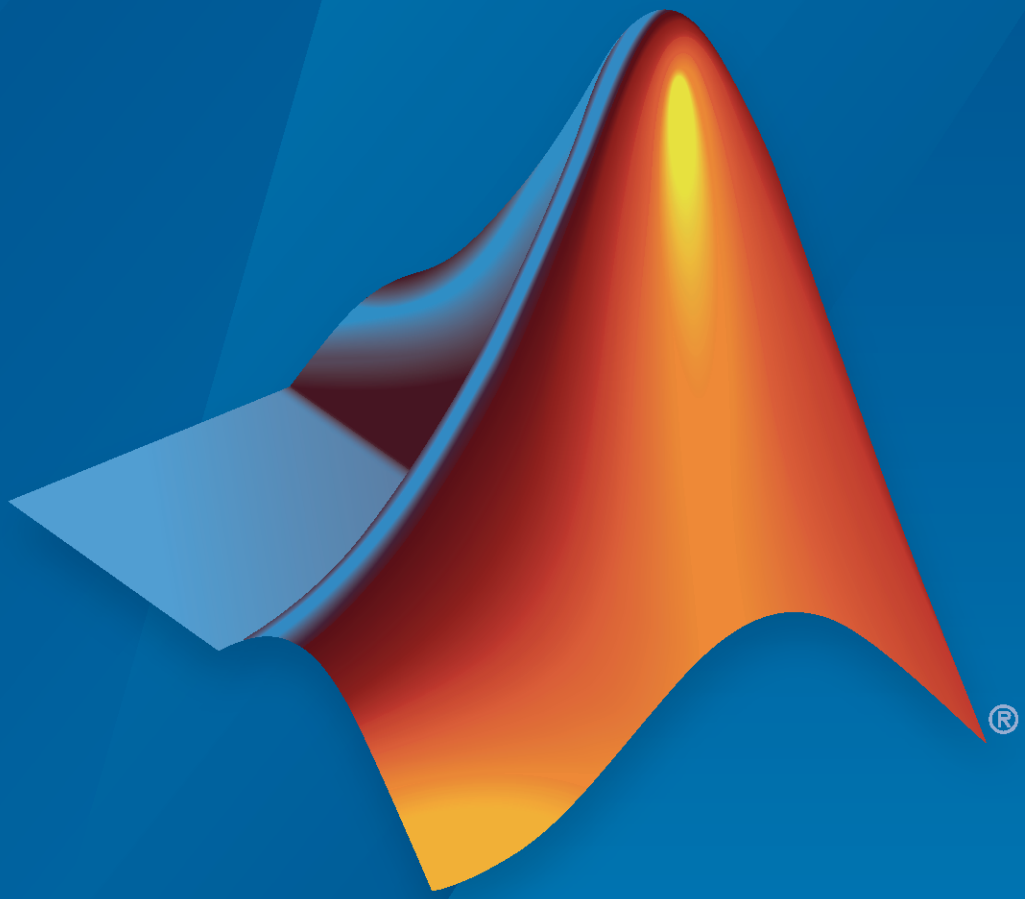
Simulink®

User's Guide



MATLAB® & SIMULINK®

R2020b

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® User's Guide

Revision History

November 1990	First printing	New for Simulink 1
December 1996	Second printing	Revised for Simulink 2
January 1999	Third printing	Revised for Simulink 3 (Release 11)
November 2000	Fourth printing	Revised for Simulink 4 (Release 12)
July 2002	Fifth printing	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	Sixth printing	Revised for Simulink 5.0 (Release 14)
October 2004	Seventh printing	Revised for Simulink 6.1 (Release 14SP1)
March 2005	Online only	Revised for Simulink 6.2 (Release 14SP2)
September 2005	Eighth printing	Revised for Simulink 6.3 (Release 14SP3)
March 2006	Online only	Revised for Simulink 6.4 (Release 2006a)
March 2006	Ninth printing	Revised for Simulink 6.4 (Release 2006a)
September 2006	Online only	Revised for Simulink 6.5 (Release 2006b)
March 2007	Online only	Revised for Simulink 6.6 (Release 2007a)
September 2007	Online only	Revised for Simulink 7.0 (Release 2007b)
March 2008	Online only	Revised for Simulink 7.1 (Release 2008a)
October 2008	Online only	Revised for Simulink 7.2 (Release 2008b)
March 2009	Online only	Revised for Simulink 7.3 (Release 2009a)
September 2009	Online only	Revised for Simulink 7.4 (Release 2009b)
March 2010	Online only	Revised for Simulink 7.5 (Release 2010a)
September 2010	Online only	Revised for Simulink 7.6 (Release 2010b)
April 2011	Online only	Revised for Simulink 7.7 (Release 2011a)
September 2011	Online only	Revised for Simulink 7.8 (Release 2011b)
March 2012	Online only	Revised for Simulink 7.9 (Release 2012a)
September 2012	Online only	Revised for Simulink 8.0 (Release 2012b)
March 2013	Online only	Revised for Simulink 8.1 (Release 2013a)
September 2013	Online only	Revised for Simulink 8.2 (Release 2013b)
March 2014	Online only	Revised for Simulink 8.3 (Release 2014a)
October 2014	Online only	Revised for Simulink 8.4 (Release 2014b)
March 2015	Online only	Revised for Simulink 8.5 (Release 2015a)
September 2015	Online only	Revised for Simulink 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Simulink 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Simulink 8.7 (Release 2016a)
September 2016	Online only	Revised for Simulink 8.8 (Release 2016b)
March 2017	Online only	Revised for Simulink 8.9 (Release 2017a)
September 2017	Online only	Revised for Simulink 9.0 (Release 2017b)
March 2018	Online only	Revised for Simulink 9.1 (Release 2018a)
September 2018	Online only	Revised for Simulink 9.2 (Release 2018b)
March 2019	Online only	Revised for Simulink 9.3 (Release 2019a)
September 2019	Online only	Revised for Simulink 10.0 (Release 2019b)
March 2020	Online only	Revised for Simulink 10.1 (Release 2020a)
September 2020	Online only	Revised for Simulink 10.2 (Release 2020b)

Contents

Introduction to Simulink

1 Simulink Basics

2 Simulation Stepping

3 How Simulink Works

Modeling Dynamic Systems

4 Creating a Model

5 Model Advisor

6 Upgrade Advisor

7 Working with Sample Times

8 Referencing a Model

9 Simulink Units

10 Conditional Subsystem

11 Messages in Simulink

12 Modeling Variant Systems

13 Managing Model Configurations

14 Configuring Models for Targets with Multicore Processors

17 Project File Management

18 Project Dependency Analysis

19 Project Source Control

22 Large-Scale Modeling

23 Power Window Example

24 Schedule Editor

Simulating Dynamic Systems

25 Running Simulations

26 Running a Simulation Programmatically

27 Multiple Simulations

28 Visualizing and Comparing Simulation Results

29 Inspecting and Comparing Simulation Data

30 Analyzing Simulation Results

31 Improving Simulation Performance and Accuracy

32 Performance Advisor

33 Solver Profiler

34 Simulink Debugger

35 Accelerating Models

36 Working with Blocks

37 Working with Block Parameters

38 Working with Lookup Tables

39 Working with Block Masks

40 Creating Custom Blocks

41 Working with Block Libraries

42 Integrate C Code in Simulink Models

43 Integrate Algorithms Using C Function

44 Using the MATLAB Function Block

45 System Objects in Simulink

46 Manage and Create a Blockset Using Blockset Designer

47 FMUs and Co-Simulation in Simulink

48 Design Considerations for C/C++ Code Generation

49 Functions, Classes, and System Objects Supported for Code Generation

50 System Objects Supported for Code Generation

51 Defining MATLAB Variables for C/C++ Code Generation

52 Defining Data for Code Generation

53 Code Generation for Variable-Size Data

54 Code Generation for MATLAB Structures

55 Code Generation for Cell Arrays

56 Code Generation for Categorical Arrays

57 Code Generation for Datetime Arrays

58 Code Generation for Duration Arrays

59 Code Generation for Tables

60 Code Generation for Timetables

61 Code Generation for MATLAB Classes

62 Code Generation for Function Handles

63 Defining Functions for Code Generation

64 Calling Functions for Code Generation

65 Improve Run-Time Performance of MATLAB Function Block

66 Troubleshooting MATLAB Code in MATLAB Function Blocks

Working with Data

67

68 Enumerations and Modeling

69 Create Data to Use as Simulation Input

73 Working with Data Stores

74 Simulink Data Dictionary

Managing Signals

75 Working with Signals

76 Using Composite Signals

77 Working with Variable-Size Signals

Customizing Simulink Environment and Printed Models

78 Customizing the Simulink User Interface

79 Frames for Printed Models

Running Models on Target Hardware

80 About Run on Target Hardware Feature

81 Running Simulations in Fast Restart

82 Component Verification

83 Simulation Testing Using Model Verification Blocks

Introduction to Simulink

Simulink Basics

The following sections explain how to perform basic tasks when using the Simulink product.

Programmatic Modeling Basics

In this section...

You can perform most Simulink modeling basics programmatically at the MATLAB® Command Window, such as creating models, adding blocks to models, and setting parameters. These examples show some of these commands and how you can use them.

Load a Model

Loading a model brings it into memory but does not open it in the Simulink Editor for editing. After you load a model, you can work with it programmatically. You can use the Simulink Editor to edit the model only if you open the model.

To load a system, use the `load_system` command. For example, to load the `vdp` model, at the MATLAB command prompt, enter:

```
load_system('vdp')
```

Create a Model and Specify Parameter Settings

You can write a function that creates a model and uses the settings that you prefer. For example, this function creates a model that has a green background and uses the ode3 solver:

```
function new_model(modelname)
% NEW_MODEL Create a new, empty Simulink model
%    NEW_MODEL('MODELNAME') creates a new model with
%    the name 'MODELNAME'. Without the 'MODELNAME'
%    argument, the new model is named 'my_untitled'.

if nargin == 0
     modelname = 'my_untitled';
end

% create and open the model
open_system(new_system(modelname));

% set default screen color
set_param(modelname,'ScreenColor','green');

% set default solver
```

```
set_param(modelname,'Solver','ode3');

% save the model
save_system(modelname);
```

Programmatically Load Variables When Opening a Model

If you assign a variable as a block parameter value, you must define the value of the variable in the model. See “Create a Model” on page 1-8. You can define the variable programmatically using the `PreloadFcn` callback with the `set_param` function. Use the function in this form:

```
set_param('mymodel','PreloadFcn','expression')
```

`expression` is a MATLAB command or a MATLAB script on your MATLAB search path. This command sets the model `PreloadFcn` callback to the value that you specify. Save the model to save the setting.

For example, when you define the variables in a MATLAB script `loadvar.m` for the model `modelname.slx`, use this command:

```
set_param('modelname','PreloadFcn','loadvar')
```

To assign the variable `K` the value `15`, use this command:

```
set_param('modelname','PreloadFcn','K=15')
```

After you save the model, the `PreloadFcn` callback executes when you next open the model.

Programmatically Add and Connect Blocks

This example shows how to use functions to add blocks and connect the blocks programmatically. Once you have added blocks to the model, you use three different approaches to connect them: routed lines, port handles, and port IDs. Routed lines allow you to specify the exact (x,y) coordinates of all connecting line segment endpoints. Port handles and port IDs allow connecting lines to block ports without having to know the port location coordinates.

Create and open a blank model named ‘`mymodel`’.

Add blocks, including a subsystem block. Use the `position` array in the `set_param` function to set the size and position of the blocks. Set the upper left and lower right block corners using (x,y) coordinates.

```
add_block('simulink/Sources/Sine Wave','mymodel/Sine1');
set_param('mymodel/Sine1','position',[140,80,180,120]);
add_block('simulink/Sources/Pulse Generator','mymodel/Pulse1');
set_param('mymodel/Pulse1','position',[140,200,180,240]);
add_block('simulink/Ports & Subsystems/Subsystem','mymodel/Subsystem1');
set_param('mymodel/Subsystem1','position',[315,120,395,200]);
add_block('simulink/Sinks/Scope','mymodel/Scope1');
set_param('mymodel/Scope1','position',[535,140,575,180]);
```

Inside `Subsystem1`, delete the default connection between `In1` and `Out1`. Also, add a second input port by copying and renaming `In1` from the block library.

```
delete_line('mymodel/Subsystem1','In1/1','Out1/1');
add_block('simulink/Sources/In1','mymodel/Subsystem1/In2');
```

Reposition the internal input and output port blocks inside `Subsystem1`.

```
set_param('mymodel/Subsystem1/In1','position',[50,50,90,70]);
set_param('mymodel/Subsystem1/In2','position',[50,130,90,150]);
set_param('mymodel/Subsystem1/Out1','position',[500,80,540,100]);
```

Insert and position an `Add` block inside `Subsystem1`.

```
add_block('simulink/Math Operations/Add','mymodel/Subsystem1/Add1');
set_param('mymodel/Subsystem1/Add1','position',[250,80,290,120]);
```

Next, add lines to connect all the blocks in the model. Start by connecting the `Sine1` and `Pulse1` blocks using routed lines.

Find the (x,y) coordinates of the `Sine1` output port.

```
Sine1_Port = get_param('mymodel/Sine1','PortConnectivity')
```

```
Sine1_Port =

  struct with fields:

       Type: '1'
   Position: [185 100]
   SrcBlock: []
    SrcPort: []
   DstBlock: [1×0 double]
    DstPort: [1×0 double]
```

`get_param` shows that the port Position is [185 100].

Find the (x,y) coordinates of the `Pulse1` output port.

```
Pulse1_Port = get_param('mymodel/Pulse1','PortConnectivity')
```

```
Pulse1_Port =

  struct with fields:

       Type: '1'
   Position: [185 220]
   SrcBlock: []
    SrcPort: []
   DstBlock: [1×0 double]
    DstPort: [1×0 double]
```

`get_param` shows that the port position is [185 220].

Connect the output of `Sine1` to the first input of `Subsystem1` using three segments of routed line.

```
add_line('mymodel', [185 100; 275 100]);
add_line('mymodel', [275 100; 275 140]);
add_line('mymodel', [275 140; 310 140]);
```

Connect the output of `Pulse1` to the second input of `Subsystem1` using three segments of routed line.

```
add_line('mymodel', [185 220; 275 220]);
add_line('mymodel', [275 220; 275 180]);
add_line('mymodel', [275 180; 310 180]);
```

Use `get_param` to get the port handles of the blocks being connected. Then use the block port handles to connect the output of `Subsystem1` to the input of `Scope1`.

```
SubsysPortHandles = get_param('mymodel/Subsystem1','PortHandles');
ScopePortHandles = get_param('mymodel/Scope1','PortHandles');
add_line('mymodel',SubsysPortHandles.Outport(1),...
ScopePortHandles.Inport(1));
```

Use port names and IDs to connect the `Add1` block inside `Subsystem1` to the subsystem inputs and outputs. Simulink uses the most direct path to connect the ports.

```
add_line('mymodel/Subsystem1','In1/1','Add1/1');
add_line('mymodel/Subsystem1','In2/1','Add1/2');
add_line('mymodel/Subsystem1','Add1/1','Out1/1');
```

Name a Signal Programmatically

1. Select the block that is the source for the signal line.
2. Use `get_param` to assign the port handle of the currently selected block to the variable *p*. Use `get_param` to assign the name of the signal line from that port to the variable *l*. Then set the name of the signal line to `'s9'`.

```
p = get_param(gcb,'PortHandles')
l = get_param(p.Outport,'Line')
set_param(l,'Name','s9')
```

Arrange Model Layouts Automatically

You can use the `Simulink.BlockDiagram.arrangeSystem` command to lay out your model. This command aligns input blocks on the left, output blocks on the right, and model elements in columns between the inputs and outputs. The command affects only one layer at a time.

You can use the `Simulink.BlockDiagram.routeLine` command to route existing lines of your model. Routing existing lines improves line route quality and avoids overlaps of a line with other lines and obstacles in the model.

While you can use these commands with any open model, they are particularly useful with models you build programmatically. For an example, see “Arrange Programmatically Populated Model”.

Open the Same Model in Multiple Windows

When you open a model, the model appears in a Simulink Editor window. For example, if you have one model open and then you open a second model, the second model appears in a second window.

To open the same model in two Simulink Editor windows, at the MATLAB command prompt, enter the `open_system` command and use the `window` argument. For example, if you have the `vdp` model open, to open another instance of the `vdp` model, enter:

```
open_system('vdp','window')
```

Locate Diagram Elements Using Highlighting

To highlight a block, line, port, or annotation in an open model, use `hilite_system`.

Specify Colors Programmatically

You can use the `set_param` command at the MATLAB command line or in a MATLAB program to set parameters that determine the background color of a diagram and the background color and foreground color of diagram elements. The following table summarizes the parameters that control block diagram colors.

Parameter	Determines
`ScreenColor`	Block diagram background
`BackgroundColor`	Block and annotation background
`ForegroundColor`	Block and annotation foreground

Set the color parameter to either a named color or an RGB value.

- Named color: `'automatic'`, `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'gray'`, `'lightBlue'`, `'orange'`, `'darkGreen'`
- RGB value: `'[r,g,b]'`

  where `r`, `g`, and `b` are the red, green, and blue components of the color normalized to the range `0.0` to `1.0`.

For example, the following command sets the background color of the currently selected system or subsystem to a light green color:

```
set_param(gcs,'ScreenColor','[0.3, 0.9, 0.5]')
```

See Also

`Simulink.BlockDiagram.routeLine` | `add_block` | `add_line` | `delete_block` | `delete_line` | `gcb` | `get_param` | `hilite_system` | `load_system` | `new_system` | `open_system` | `save_system` | `set_param`

More About

- “Common Block Properties”
- “Specify Model Colors” on page 36-12

Simulink Identifiers

A Simulink Identifier (SID) is a unique and unmodifiable designation automatically assigned to a Simulink block, model annotation, or Stateflow® object within a Stateflow chart. The SID helps to identify specific instances of these components in your diagram, especially when sharing models between people within a team.

The SID has these characteristics:

- Persistent within the lifetime of a Simulink block, model annotation, or Stateflow object
- Saved in the model file
- Remains the same if the block or object name changes
- Cannot be modified

The SID format is:

```
model_name:sid_number
```

- `model_name` is the name of the model where the block, annotation, or Stateflow object resides.
- `sid_number` is a unique number within the model, assigned by Simulink.

The SID includes additional colons in certain cases, for example on an instance of a block from a user library.

See Also

`Simulink.ID.getHandle` | `Simulink.ID.getSID` | `Simulink.ID.hilite`

More About

- “Programmatic Modeling Basics” on page 1-2

Build and Edit a Model Interactively

In this section...

Learn the basics of how to create a model, add blocks to it, connect blocks, and simulate the model. You also learn how to organize your model with subsystems, name parts of a model, and modify a model.

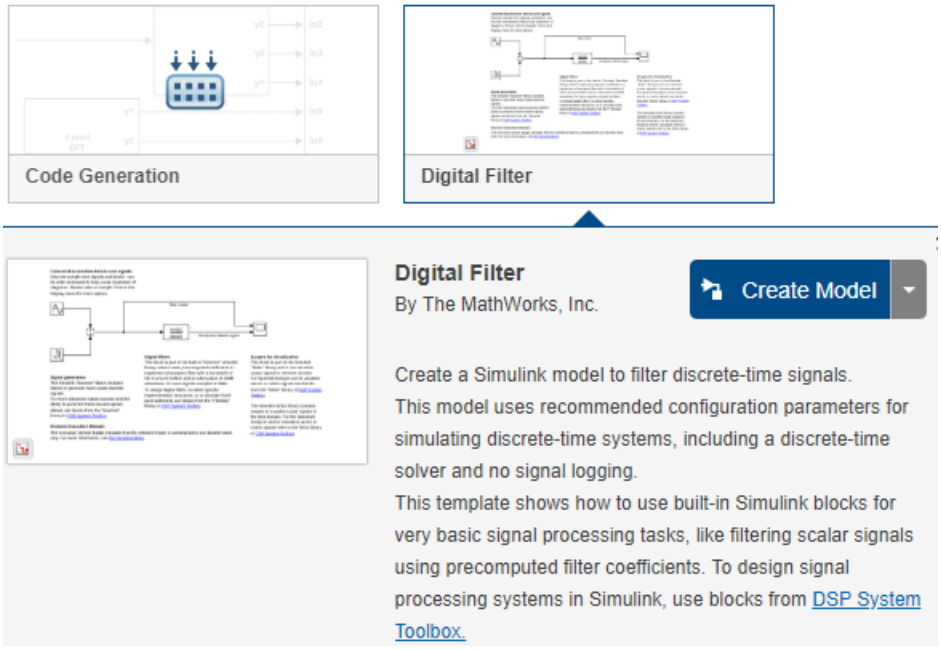
For a summary of how to connect blocks and add ports interactively, see “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61.

Create a Model

1. On the MATLAB **Home** tab, click **Simulink**.
2. In the Simulink start page, choose a template or search the templates.

   Model templates are starting points to apply common modeling approaches. They help you reuse settings and block configurations and share knowledge. Use model and project templates to apply best practices and take advantage of previous modeling solutions.

   Click the title of a template to read the description.



Search for templates by entering text in the search box. You can enter flexible search options including OR, AND, NOT, wildcards, fuzzy (~), etc. Use the Lucene search engine query parser syntax.

To locate templates that are not on the MATLAB search path, click **Open**. Model templates have the extension `.sltx`.

3. After selecting the template you want, click **Create Model**.

   To use a template without reading the description, click the template image. Alternatively, press **Ctrl+N** to use your default template.

   A new model using the template settings and contents opens in the Simulink Editor.

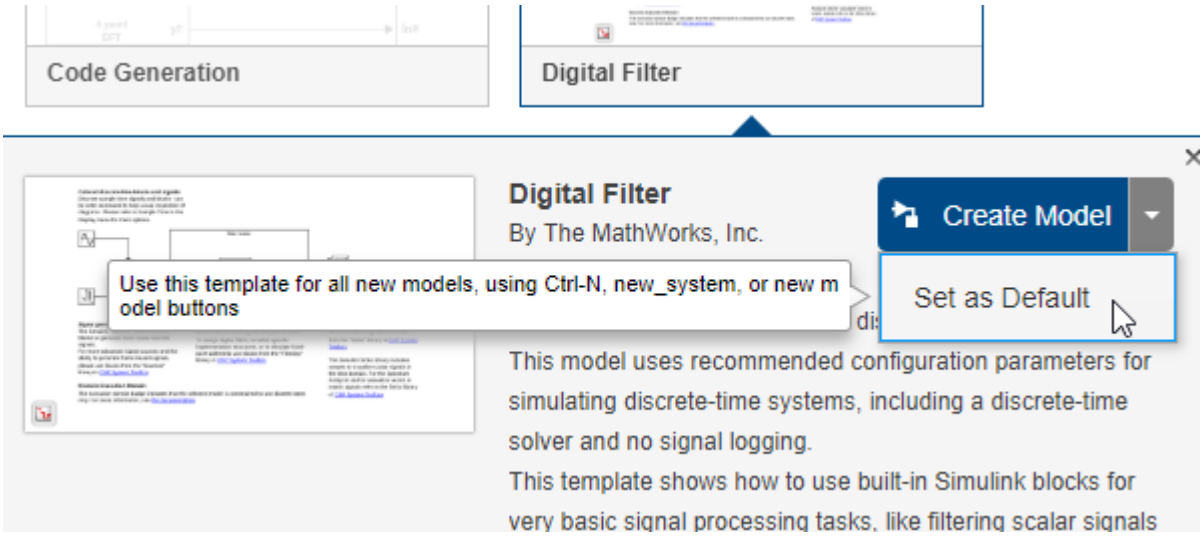
If the built-in templates do not meet your needs, try searching on the **Examples** tab, or you can create your own templates. See “Create a Template from a Model” on page 4-2. On the **Examples** tab, enter search terms to find examples titles and descriptions of interest, or open further examples on the web by clicking **View All** next to a product name.

Use Customized Settings When Creating New Models

You can specify a model template to use for all new models.

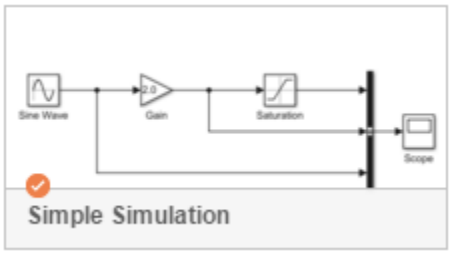
1. Create a model with the configuration settings and blocks you want, then export the model to a template. See “Create a Template from a Model” on page 4-2.
2. To reuse these settings in every new model, make the new template your default model template using the Simulink start page or the `Simulink.defaultModelTemplate` function.

   In the start page, click the title of a template to expand the description, then click the down arrow next to **Create Model** and select **Set As Default**.



After you set a default model template, every new model uses that template, for example, when you press **Ctrl+N**, when you use new model buttons, or when you use `new_system`. In the Simulink Editor, your default template appears at the top of the list when, on the **Simulation** tab, you select **New**.

The default template shows a tick mark in the start page.



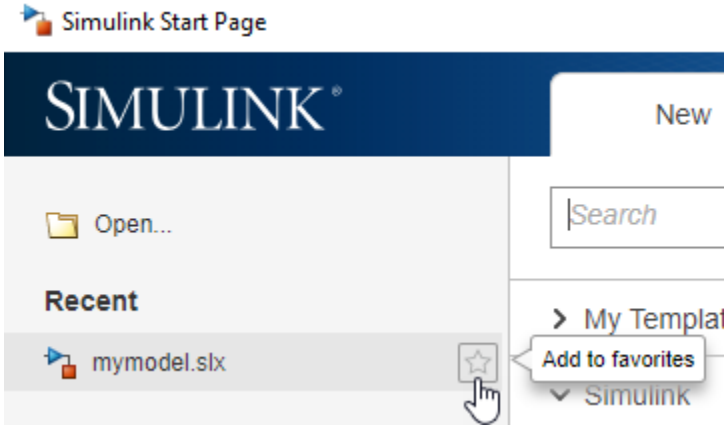
Open a Model

Opening a model loads the model into memory and displays it in the Simulink Editor. Use any of these techniques:

- On the MATLAB **Home** tab, click **Simulink**. In the Simulink Start Page, select a recent model or project from the list, or click **Open**.
- In the Simulink Editor, on the **Simulation** tab, select **Open > Recent Models** and choose a recent model.
- At the MATLAB command prompt, enter the name of the model without a file extension, for example, `vdp`. The model must be in the current folder or on the MATLAB search path.
- In the Simulink Library Browser, click the **Open model or library** button .
- Open the model using the Current Folder browser or your operating system file browser.

Tip Set favorites to easily get back to your favorite models and projects in the start page.

In the Simulink start page recent files list, you can add files to favorites. The Favorites list then appears above recent files in the start page, so that you can easily reopen your favorite models and projects.



To edit or clear the list of recent files in the start page, right-click a recent file and use the context menu.

Alternatively, use `Simulink.history.clear` to clear the Simulink history programmatically.

Note To open a model created in a later version of Simulink software in an earlier version, first export the model to the earlier version. See “Export a Model to a Previous Simulink Version” on page 1-31.

Load Variables When Opening a Model

As you build models, you sometimes define variables for a model. For example, suppose that you have a model that contains a Gain block. You can specify the variable `K` as the gain rather than setting the value on the block. When you use this approach, you must define the variable `K` for the model to simulate.

You can use a model callback to load variables when you open a model.

1. In a model that uses the Gain block, set the block **Gain** value to `K`.
2. Define the variable in a MATLAB script. In MATLAB, select **New** > **Script**. In the script, enter your variable definitions:

   ```
   K=27
   ```
3. Save the script as `loadvar.m`.
4. In the model, open the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**. With no selection at the top level of a model, you can use the Property Inspector to set model properties. Otherwise, on the **Modeling** tab, click **Model Settings**.
5. In the **Callbacks** section of the model properties, select `PreLoadFcn` as the callback that you want to define. In the pane, enter `loadvar`.
6. Save the model.

The next time that you open the model, the `PreloadFcn` callback loads the variables into the MATLAB workspace.

To learn about callbacks, see “Callbacks for Customized Model Behavior” on page 4-44. To define a callback for loading variables programmatically, see “Programmatically Load Variables When Opening a Model” on page 1-3.

Open a Model with Different Character Encoding

If you open an MDL file 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 an MDL file 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.

SLX files do not warn because they can store characters from any encoding.

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 of the current MATLAB session to match the model character encoding.

Simulink can check if models contain characters unsupported in the current locale. For more details, see “Check model for foreign characters” and “Save Models with Different Character Encodings” on page 1-30.

Simulink Model File Types

New models that you create have the `.slx` extension by default. Models created before R2012b have the extension `.mdl`. Models you can edit can have the `.slx` or `.mdl` extension, depending on when they were created or whether you converted them. See “Save Models in the SLX File Format” on page 1-28.

`.slxp` and `.mdlp` extensions denote protected models that you cannot open and edit. See “Reference Protected Models from Third Parties” on page 8-13. Model templates have the extension `.sltx`.

Simulink libraries also use the `.slx` extension, but you cannot simulate them. To learn more, see “Create a Custom Library” on page 41-2.

See Also

`Simulink.createFromTemplate` | `Simulink.defaultModelTemplate` | `Simulink.findTemplates` | `open_system` | `simulink`

Related Examples

- “Add Blocks and Set Parameters” on page 1-13
- “Extend the Model” on page 1-16
- “Simulate the Model and View Results” on page 1-22
- “Edit and Simulate the Model” on page 1-24
- “Save the Model” on page 1-27

Add Blocks and Set Parameters

Add Blocks to the Model

A minimal model takes an input signal, operates on it, and outputs the result. In the Library Browser, the Sources library contains blocks that represent input signals. The Sinks library has blocks for capturing and displaying outputs. The other libraries contain blocks you can use for a variety of purposes, such as math operations.

In this example of a basic model, the input is a sine wave, the operation is a gain (which increases the signal value by multiplying), and you output the result to a scope. Try different techniques to explore the library and to add blocks to your model.

1. Open the Sources library. In the tree view of the Library Browser, click the **Sources** library.
2. In the right pane, pause on the Sine Wave block to see a tooltip describing its purpose.
3. Add a block to your model. Right-click the Sine Wave block and select **Add block to model untitled**. To learn more about the block, right-click the block and select **Help**.
4. Add a block to your model by dragging. In the library tree view, click **Math Operations**. In the Math Operations library, locate the Gain block and drag it to your model to the right of the Sine Wave block.
5. In the library tree view, click **Simulink** to view the sublibraries as icons in the right pane. This view is an alternative way to navigate the library structure. Double-click the **Sinks** library icon.
6. In the Sinks library, locate the Scope block and add it to your model using the context menu or by dragging it.

   The figure shows your model so far.

Note The editor names blocks as you add them. For example, it names the first Gain block that you add Gain, the next Gain1, and so on. By default, the Simulink Editor hides these names. However, you can see the name by selecting the block. You can also explicitly name a block so that the name appears. You can display all names given by the editor. In the **Debug** tab, select **Information Overlays > Hide Automatic Block Names**. For more information on displaying block names, see “Manage Block Names and Ports” on page 36-10.

Align and Connect Blocks

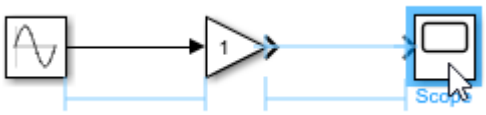
Connect the blocks to create the relationships between model elements that are needed to make the model operate. Reading the model is easier when you line up blocks according to how they interact with each other. Shortcuts help you to align and connect blocks.

1. Drag a Gain block so it lines up with a Sine Wave block. An alignment guide appears when the blocks line up horizontally.

2 Release the block, and a blue arrow appears as a preview of the suggested connection.

3 To make the connection, click the arrow. A solid line appears in place of the guide.

4 Line up and connect the Scope block to the Gain block using the same technique. Additional guides appear when multiple blocks are within range.



Tip For additional alignment options, on the **Format** tab, click options in the **Align** section.

Set Block Parameters

You can set parameters on most blocks. Parameters help you to specify how a block operates in the model. You can use the default values or you can set values. Use the Property Inspector to set parameters. Alternatively, you can double-click most blocks to set the parameters using a block dialog box.

In your model, set the sine wave amplitude and the gain value.

1 Display the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**.

2 Select the Sine Wave block.

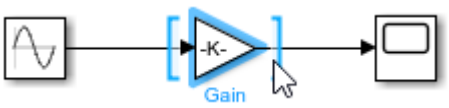
3 In the Property Inspector, set the **Amplitude** parameter to 2.

4 For blocks whose value appears on the icon, you can edit the parameter interactively. Select the Gain block. Pause on the block. A blue underline appears under the number.

5 Set the **Gain** parameter to `300000`. Click the underlined number, delete it, and enter `300000`.

Blocks such as Constant and Gain blocks display a parameter value only when it fits on the block icon.

6 To resize the block so that it displays the parameter value, click the interactive cue.



Alternatively, use one of these options to resize the block so that it displays the parameter value:

- In the Simulink Toolstrip, on the **Format** tab, select **Fit to Content**.
- In the Simulink Editor, on the action bar that appears when you select the block and pause on the ellipsis, select **Fit to Content**.

In a block dialog box or in the Property Inspector, when you set a block parameter value to a variable or function, Simulink provides a suggested list to select from based on the current text typed in the edit field. The suggestions include relevant variables or objects from every workspace (base, model, and mask), data dictionary, and referenced dictionary visible to the editable block parameter. Autocomplete is available for variables, fields of structures and objects, and functions on the MATLAB path.

See Also

`Simulink.createFromTemplate` | `Simulink.defaultModelTemplate` | `Simulink.findTemplates` | `open_system` | `simulink`

Related Examples

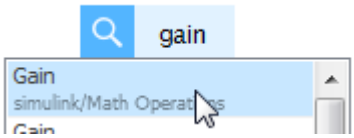
- “Extend the Model” on page 1-16
- “Simulate the Model and View Results” on page 1-22
- “Edit and Simulate the Model” on page 1-24
- “Save the Model” on page 1-27

Extend the Model

Add More Blocks

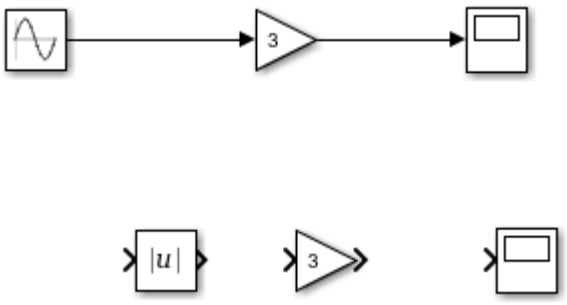
Suppose that you want to perform another gain but on the absolute value of the output from the Sine Wave block. Add blocks for this purpose, trying different techniques for locating blocks in the library and adding them to your model.

1. If you know the name of the block that you want to add, you can use a shortcut. Double-click where you want to add the block, and type the block name, in this case Gain.



   The list of suggestions shown are dynamically ranked based on your recent block usage history.
2. Click the block name or, with the block name highlighted, press **Enter**. You can use the arrow keys to highlight the block name if it is not first in the list.
3. Some blocks display a prompt for you to enter a value for one of the block parameters. The Gain block prompts you to enter the **Gain** value. Type `3` and press **Enter**.
4. To perform an absolute value, add an Abs block. Suppose you do not know the library a block is in or the full name of the block. You can search for it using the search box in the Library Browser. Enter `abs` in the search box and press **Enter**. When you find the Abs block, add it to the left of the new Gain block.
5. Add another Scope block. You can right-click the existing Scope block and drag to create the copy or use the **Copy** and **Paste** commands.

   The figure shows the current state of the model.

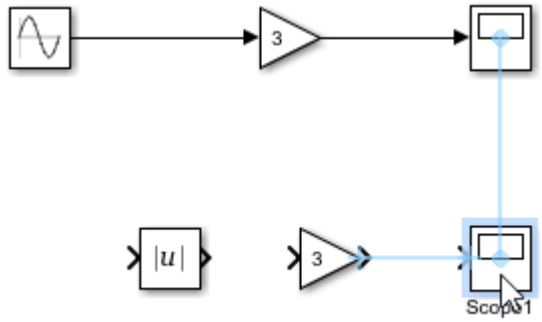


Branch a Connection

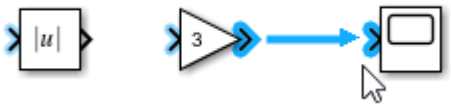
The input to the second Gain block is the absolute value of the output from the Sine Wave block. To use a single Sine Wave block as the input to both gain operations, create a branch from the Sine Wave block output signal.

1. For the first set of blocks in your model, you used the horizontal alignment guides to help you align and connect them. You can also use guides to align blocks vertically. Drag the second Scope

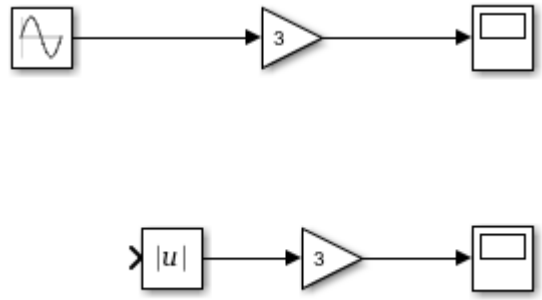
block so that it lines up under the first one. Release it when the vertical alignment guide shows that the blocks are aligned.



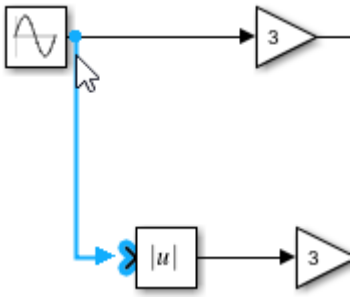
2 You can click two ports to connect them. After you click the first port, compatible ports appear highlighted. Click another port to connect.



Align and connect the blocks as shown.

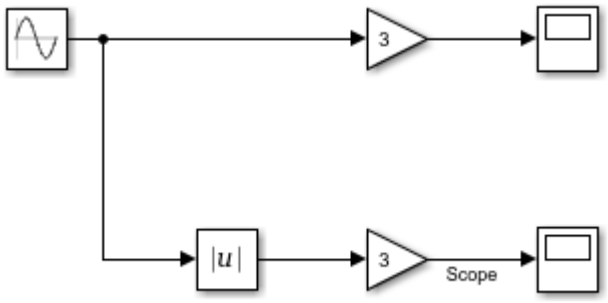


3 Create a branch from the Sine Wave block output to the Abs block. Click the input port of the Abs block. Move the cursor toward the output signal line from the Sine Wave block. A preview line appears. Click to create the branch.



Alternatively, you can start the branch by clicking the line segment and then moving the cursor toward the port.

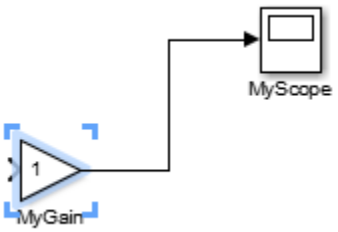
4 Name signals. Double-click the signal between the lower Gain block and the Scope block and type `Scope`. Double-click the line and not a blank area of the canvas.



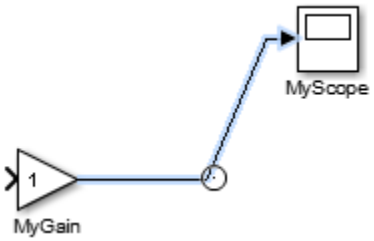
Try these methods to add or connect blocks:

- Drag from a block port and release so that a red, dotted line appears. Double-click the end of the line to use the block insertion shortcut. Suggested blocks for the current context appear on the menu. You can select one of the listed blocks.

  To improve the menu suggestions based on your model designs, see “Improve Quick Block Insert Results” on page 78-22.
- Double-click and then type the name of a block to get a list of blocks that starts with the characters you typed. For custom library blocks, you can type the block keyword if the library author assigned one. The list is ranked based on your recent block-usage history.
- After you click a port, hold **Shift** as you connect to another port. Holding **Shift** puts you in a mode in which you can make multiple, consecutive connections. For example, while holding **Shift**, you can branch a new signal line and connect it to another port or signal line with one click.
- Select the first block and **Ctrl+**click the block you want to connect it to. This technique is useful when you want to connect blocks that have multiple inputs and outputs, such as multiple blocks to a bus or two subsystems with multiple ports. As with clicking two ports, this technique is useful when you do not want to align blocks. The signal line bends as needed to make the connection.



To approximate a diagonal line from line segments, press **Shift** and drag a vertex.

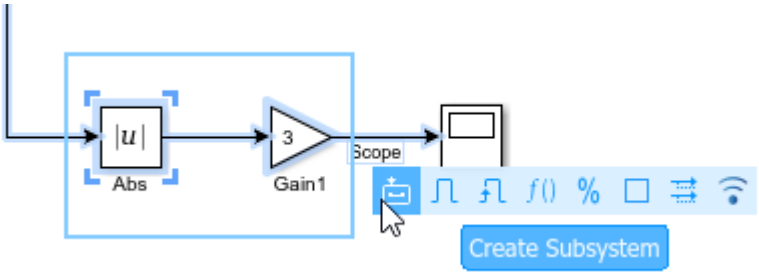


Tip To improve the shape of a signal line, select the line and, from the action bar, select **Auto-route Line**. The line redraws if a better route between model elements is possible. You can select **Auto-route Lines** from the action bar to improve lines with a single block selected or with multiple model elements selected by dragging a selection box.

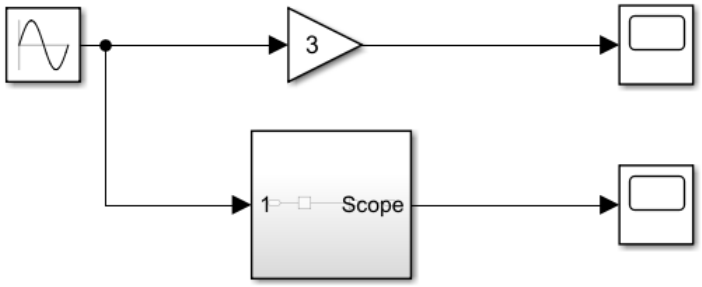
Organize Your Model Into Components

You can group blocks in subsystems and label blocks, subsystems, and signals. For more information about subsystems, see “Create Subsystems” on page 4-15.

1. Drag a selection box around the Abs block and the Gain block next to it.
2. Move the cursor over the ellipses that appear at the corner of the box where you ended the selection. From the action bar, select **Create Subsystem**.



A subsystem block appears in the model in place of the selected blocks. The subsystem block displays the input and output port indexes.



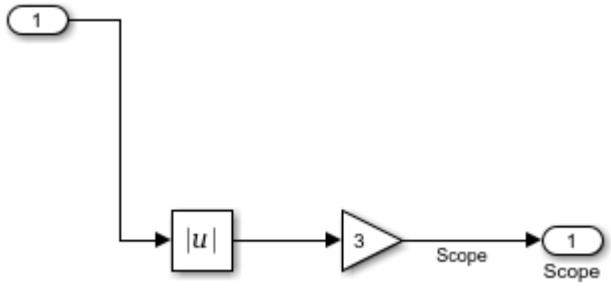
Note You can place a port on any side of the subsystem block. To move a port, click and drag it around the block.

To resize the subsystem block for the best fit in your model, drag the block handles.

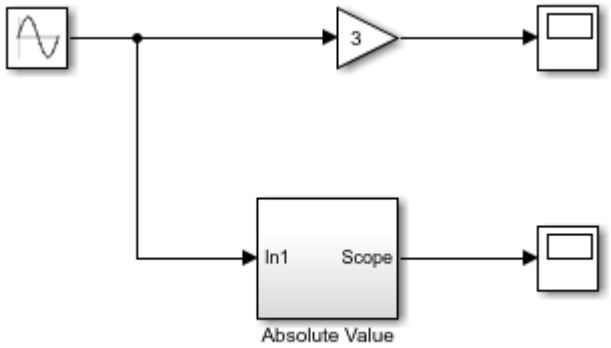
3. Give the subsystem a meaningful name. Select the block, double-click the name, and type `Absolute Value`. Naming a block causes the name to appear in the model.
4. Open the Absolute Value subsystem by double-clicking it.

Tip To use the Explorer Bar to navigate the model hierarchy, right-click the subsystem and select **Open in New Tab**.

The subsystem contains the blocks and signal that you selected as the basis of the subsystem. They are connected in sequence to two new blocks: an Inport block and an Outport block. Inport and Outport blocks correspond to the input and output ports on the subsystem. Creating the subsystem from a selection that includes a named signal adds the name of the signal to the corresponding Inport or Outport block.



5. Click the **Up to Parent** button to return to the top level of the model.
6. The figure shows the model after you create the subsystem and name it.



See Also

Related Examples

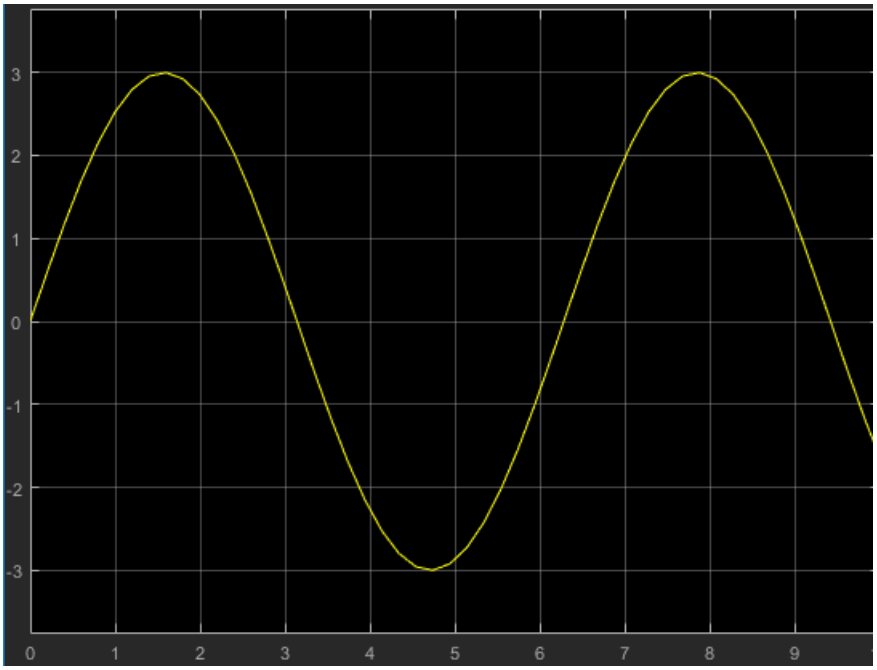
- “Add Blocks and Set Parameters” on page 1-13
- “Simulate the Model and View Results” on page 1-22
- “Edit and Simulate the Model” on page 1-24
- “Save the Model” on page 1-27
- “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61

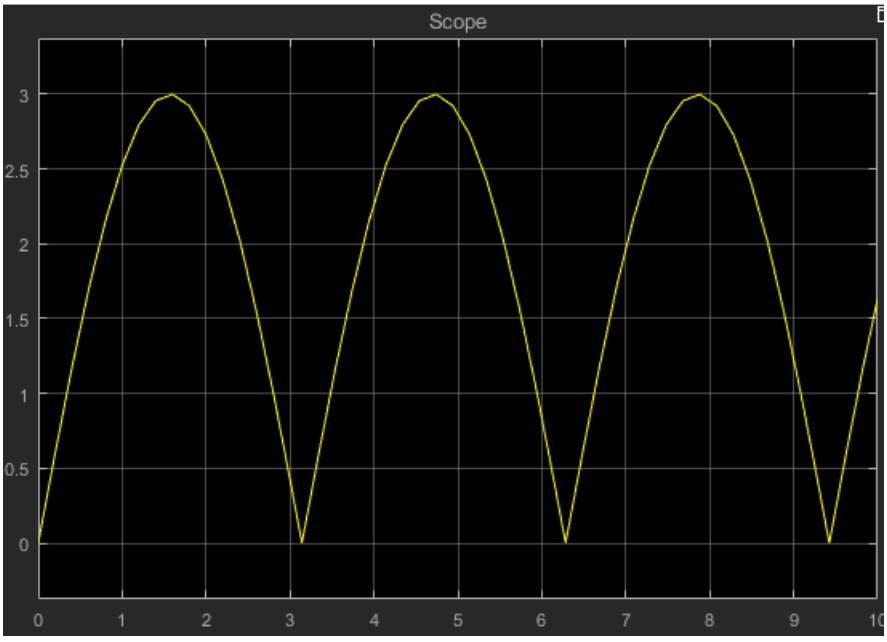
Simulate the Model and View Results

1. You can simulate a model by clicking the **Run** button  .

   In this example, simulation runs for 10 seconds, the default setting.
2. Double-click both Scope blocks to open them and view the results.

   The figure shows the two results. In the second plot, as expected, the absolute value of the sine wave is always positive.





See Also

Related Examples

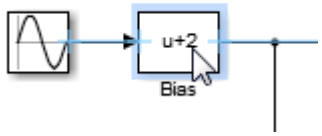
- “Add Blocks and Set Parameters” on page 1-13
- “Extend the Model” on page 1-16
- “Edit and Simulate the Model” on page 1-24
- “Save the Model” on page 1-27

Edit and Simulate the Model

You can add blocks to a signal, remove blocks from models, and redraw connections. To modify this model, add a bias to the input to both branches of your model. Also, replace one of the scopes with a different sink. Add more blocks to the subsystem and another output.

1. Add a Bias block to the model and set the **Bias** parameter to 2.
2. Drag the block onto the signal line after the Sine Wave block but before the branch. If you need to make room for the block, drag the Sine Wave block to the left or move the end of the branch by dragging it to the right.

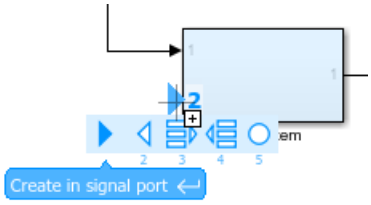
   When you drag the block onto the signal line, the block connects to the signal line at both ends. Release the block when you are satisfied with the position.



3. Remove the top Scope block. If you want to disconnect it from the model, but do not want to delete it, press and hold **Shift** and drag the block. Cut or delete it using **Ctrl-X** or the **Delete** key. The broken connection appears as a red dotted line.

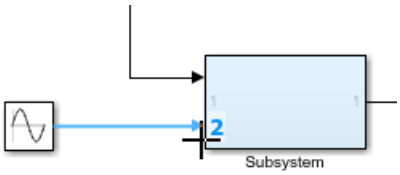
   Tip When you delete a block that has one input and one output, a prompt appears between the broken connection lines. Click the prompt to connect the signals.

4. Add a To Workspace block to the model at the end of the broken connection. The To Workspace block outputs the results to a variable in the MATLAB workspace.
5. Add a Sine Wave block to the model and set the amplitude to 5. Place it to the left of the subsystem.
6. Add another input to the subsystem. Drag a line from the new Sine Wave block to the left side of the subsystem. A new port, In2, appears on the subsystem.

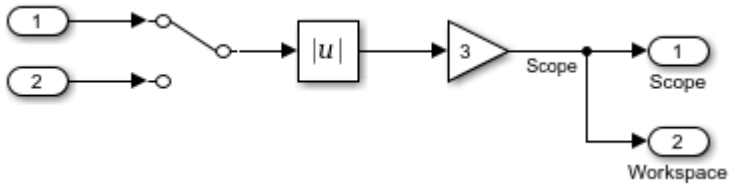


   For certain blocks, dragging a line to it adds an input port or output port. For example, a port appears on a subsystem when you connect a line to it. Other blocks that add ports include the Bus Creator, Scope, and Add, Sum, and Product blocks.

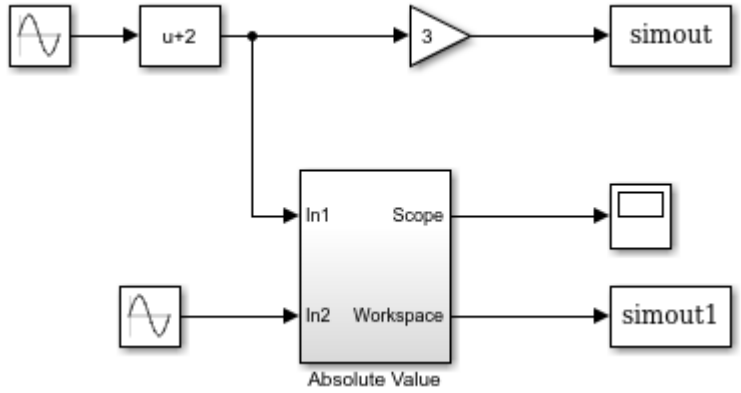
7. Add an output to the subsystem. Add another To Workspace block to the model and place it to the right of the subsystem. Drag a line from its input port to the right side of the subsystem. A new port, Out2, appears on the subsystem.



8 Open the subsystem and rename the Out2 block Workspace. Add a Manual Switch block to the subsystem. Resize it and connect it as shown. Branch the signal after the Gain block to direct the output to the To Workspace block.



Then, return to the top level of the model. The figure shows the current model.



9 Simulate the model.

- The `simout` and `simout1` variables appear in the MATLAB workspace. Double-click each variable to explore the results.
- If you want to use the second sine wave as input to the subsystem algorithm, open the subsystem and double-click the switch. The input changes to In2. Simulate again.

Tip To toggle between simulating the model with and without the effects of the Bias block, right-click the Bias block and select **Comment Through**. The block stays in the model but does not affect the operation. Right-click the Bias block and select **Uncomment** to enable the block. The **Comment Out** command comments out the block's output signal, so signal data does not pass through. Try each of these commands to better understand their effects.

See Also

Related Examples

- “Add Blocks and Set Parameters” on page 1-13
- “Extend the Model” on page 1-16
- “Simulate the Model and View Results” on page 1-22
- “Save the Model” on page 1-27

Save the Model

In this section...

How to Tell If a Model Needs Saving

To tell whether a model needs saving, look at the title bar in the Simulink Editor. If a model in the model hierarchy needs saving, an asterisk appears in the title bar (known as the dirty flag: *).

To determine programmatically whether a model needs saving, use the model parameter `Dirty`. For example:

```
if strcmp(get_param(gcs,'Dirty'),'on')
    save_system;
end
```

Save a Model

To save a model for the first time, in the Simulink Editor, on the **Simulation** tab, click **Save**. Provide a location and name for the model file. For name requirements, see “Model Names” on page 1-28.

To save a previously saved model:

- To replace the file contents, in the Simulink Editor, on the **Simulation** tab, click **Save**.
- To save the top model with a new name or location, or to change from MDL to SLX format, in the Simulink Editor, on the **Simulation** tab, select **Save** > **Save As**.

  Note For details about the SLX format, see “Upgrade Models to SLX” on page 1-29.
- To save a referenced model with a new name, location, or format, open it as a top model, then on the **Simulation** tab, select **Save** > **Save As**.
- To save the top model in a format compatible with the earlier version, on the **Simulation** tab, select **Save** > **Previous Version**. See “Export a Model to a Previous Simulink Version” on page 1-31.

Model Names

Model file names must start with a letter and can contain letters, numbers, and underscores. The file name must not be:

- A language keyword (e.g., `if`, `for`, `end`)
- A reserved name: `'simulink'`, `'sl'`, `'sf'`
- A MATLAB software command

The total number of characters in the model name must not be greater than a certain maximum, usually 63 characters. To find out whether the maximum for your system is greater than 63 characters, use the MATLAB `namelengthmax` command.

To understand how MATLAB determines which function to call when you specify a model name, see "Function Precedence Order".

What Happens When You Save a Model?

Simulink saves the model (block diagram) and block properties in the model file.

If you have any pre- or post-save functions, they execute in this order:

1. All block `PreSaveFcn` callback routines execute first, then the model `PreSaveFcn` callback routine executes.
2. Simulink writes the model file.
3. All block `PostSaveFcn` callback routines execute, then the model `PostSaveFcn` executes.

During the save process, Simulink maintains a temporary backup copy (named *`modelname`*`.bak`) for restoring in case of an error. If an error occurs during saving or during any callback during the save process, Simulink:

- Restores the original file
- Writes any content saved before the error occurred in a file named *`modelname`*`.err`
- Issues an error message

When saving a model loaded from an SLX file, the original SLX file must still be present. Simulink performs incremental loading and saving of SLX files, so if the original file is missing at save-time, Simulink warns that it cannot reconstruct the file fully.

Save Models in the SLX File Format

Save New Models as SLX

Simulink saves new models and libraries in the SLX format by default, with file extension `.slx`. SLX is a compressed package that conforms to the Open Packaging Conventions (OPC) interoperability standard. SLX stores model information using Unicode® UTF-8 in XML and other international formats. Saving Simulink models in the SLX format:

- Typically reduces file size compared to MDL. The file size reduction between MDL and SLX varies depending on the model.
- Solves some problems in previous releases with loading and saving MDL files containing Korean and Chinese characters.

- Enables incremental loading and saving. Simulink optimizes performance and memory usage by loading only required parts of the model and saving only modified parts of the model.

You can specify your file format for saving new models and libraries with the Simulink preference "File format for new models and libraries".

Upgrade Models to SLX

If you upgrade an MDL file to SLX file format, the file contains the same information as the MDL file, and you always have a backup file. All functionality and APIs that currently exist for working with models, such as the `get_param` and `set_param` commands, are also available when using the SLX file format. If you upgrade an MDL file to SLX file format without changing the model name or location, then Simulink creates a backup file by renaming the MDL (if writable).

If you save an existing MDL file by clicking **Save** on the **Simulation** tab, Simulink respects the file's current format and saves your model in MDL format.

To save an existing MDL file in the SLX file format:

1. On the **Simulation** tab, select **Save** > **Save As**.
2. Leave the default **Save as type** as SLX, and click **Save**.

   Simulink saves your model in SLX format, and creates a backup file by renaming the MDL (if writable) to `mymodel.mdl.`*`releasename`*, e.g., `mymodel.mdl.R2010b`.

Alternatively, use `save_system`:

```
save_system mymodel mymodel.slx
```

This command creates `mymodel.slx`, and if the existing file `mymodel.mdl` is writable it is renamed `mymodel.mdl.`*`releasename`*.

SLX files take precedence over MDL files, so if both exist with the same name and you do not specify a file extension, you load the SLX file.

Simulink Projects can help you migrate files to SLX. For an example, see "Convert from MDL to SLX in a Project and Preserve Revision History".

Caution If you use third-party source control tools, be sure to register the model file extension `.slx` as a binary file format. If you do not, these third-party tools might corrupt SLX files when you submit them.

Operations with Possible Compatibility Considerations when using SLX	What Happens	Action
Hard-coded references to file names with extension `.mdl`.	Scripts cannot find or process models saved with new file extension `.slx`.	Make your code work with both the `.mdl` and `.slx` extension. Use functions like `which` and `what` instead of file names.

Operations with Possible Compatibility Considerations when using SLX	What Happens	Action
Third-party source control tools that assume a text format by default.	Binary format of SLX files can cause third-party tools to corrupt the files when you submit them.	Register `.slx` as a binary file format with third-party source control tools. Also recommended for `.mdl` files. See “Register Model Files with Source Control Tools” on page 19-8.
Changing character encoding.	Some cases are improved, e.g., SLX solves some problems in previous releases with loading and saving MDL files containing Korean and Chinese characters. However, sharing models between different locales remains problematic.	See “SLX Files and Character Encodings” on page 1-31.

The format of content within MDL and SLX files is subject to change. To operate on model data, use documented APIs (such as `get_param`, `find_system`, and `Simulink.MDLInfo`.

Save Models with Different Character Encodings

MDL Files and Character Encodings

When you save a model, the current character encoding is used to encode the text stored in the model file. With MDL files, this can lead to model corruption if you save a model whose original encoding differs from current encoding.

If you change character encoding, it is possible to introduce characters that cannot be represented in the current encoding. If this is the case, the model is saved as *`model`*`.mdl.err`, where *`model`* is the model name, leaving the original model file unchanged. Simulink also displays an error message that specifies the line and column number of the first character which cannot be represented.

To recover from this error, either:

- Save the model in SLX format (see “Save Models in the SLX File Format” on page 1-28).
- Locate and remove characters one by one.

  1. Use a text editor to find the character in the `.err` file at the position specified by the save error message.
  2. Find and delete the corresponding character in the open model and resave the model.
  3. Repeat this process until you are able to save the model without error.

It’s possible that your model’s original encoding can represent all the text changes made in the current session, albeit incorrectly. For example, suppose you open a model whose original encoding is A in a session whose current encoding is B. Further suppose that you edit the model to include a

character that has different encodings in A and B and then save the model. If in addition the encoding for x in B is the same as the encoding for y in A, and if you insert x in the model while B is in effect, save the model, and then reopen the model with A in effect the Simulink software will display x as y. To alert you to the possibility of such corruptions, a warning message appears whenever you save a model in which the current and original encoding differ but the original encoding can encode, possibly incorrectly, all of the characters to be saved in the model file.

SLX Files and Character Encodings

Saving Simulink models in the SLX format typically reduces file size and solves some problems in previous releases with loading and saving MDL files containing Korean and Chinese characters.

Considerations for choosing a model file format:

- Use SLX if you are loading and saving models with Korean or Chinese characters
- Use SLX if you would benefit from a compressed model file
- Whether you use SLX or MDL, Simulink can detect and warn if models contain characters unsupported in the current locale. For SLX, you can use the Model Advisor to help you, see “Check model for foreign characters”.

Export a Model to a Previous Simulink Version

You can export (save) a model created with the latest version of the Simulink software in a format used by an earlier version. For example, to share a model with colleagues who only have access to a previous version of the Simulink product.

To export a model in an earlier format:

1. In the Simulink Editor, on the **Simulation** tab, click **Save**. This saves a copy in the latest version of Simulink. This step avoids compatibility problems.
2. In the Simulink Editor, on the **Simulation** tab, select **Save > Previous Version**.
3. In the Export to Previous Version dialog box, from the **Save as type** list, select the previous version to which to export the model. The list supports 7 years of previous releases.
4. Click **Save**.

When you export a model to a previous version’s format, the model is saved in the earlier format, regardless of whether the model contains blocks and features that were introduced after that version. If the model does contain blocks or use features that postdate the earlier version, the model might not give correct results when you run it in the earlier version of Simulink software. In addition, Simulink converts blocks that postdate an earlier version into yellow empty masked Subsystem blocks. For example, if you use `save_system` to export a model to Release R2007b, and the model contains Polynomial blocks, Simulink converts the Polynomial blocks into yellow empty masked Subsystem blocks. Simulink also removes any unsupported functionality from the model. See `save_system`.

Save from One Earlier Simulink Version to Another

You can open a model created in an earlier version of Simulink and export that model to a different earlier version. To prevent compatibility problems, use the following procedure if you need to save a model from one earlier version to another earlier version.

1. Use the current version of Simulink to open the model created with the earlier version.
2. Before you make any changes, save the model in the current version by clicking **Save** on the **Simulation** tab.

   After saving the model in the current version, you can change and resave it as needed.
3. Save the model in the earlier version of Simulink. On the **Simulation** tab, select **Save** > **Previous Version**.
4. Start the earlier Simulink version and use it to open the model that you exported to that earlier version.
5. Save the model in the earlier version.

You can now use the model in the earlier version of Simulink exactly as you could if it had been created in that version.

See also the Simulink preferences that can help you work with models from earlier versions:

- “Do not load models created with a newer version of Simulink”
- “Save backup when overwriting a file created in an older version of Simulink”

See Also

`save_system`

Related Examples

- “Add Blocks and Set Parameters” on page 1-13
- “Extend the Model” on page 1-16
- “Simulate the Model and View Results” on page 1-22
- “Edit and Simulate the Model” on page 1-24

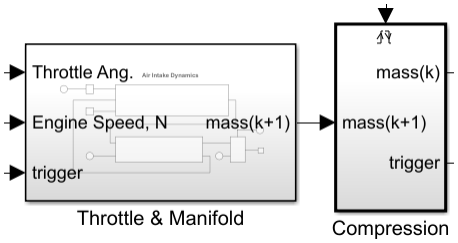
Preview Content of Model Components

To display a representation of the contents of a hierarchical model element, such as a subsystem, on the block, you can use content preview. Content preview helps you see the contents of a hierarchical element without navigating the hierarchy. Content preview shows blocks, signals, signal and block labels, sample time color coding, signal highlighting, and Stateflow animations.

By default, content preview is on for new hierarchical elements. You can change this setting by using Simulink Preferences. To open the Simulink Preferences dialog box, on the **Modeling** tab, select **Environment > Simulink Preferences**. On the **Editor** pane, you can clear the **Content preview displays for new hierarchical elements** option.

To toggle content preview for a hierarchical element, select the block that you want to enable or disable content preview for, and then on the **Format** tab, click **Content Preview**. Content preview settings for blocks apply across Simulink sessions.

In this figure, the Throttle & Manifold subsystem has content preview enabled.



Simulink scales the content preview to fit the size of the block. To improve the readability of the content preview, you can zoom in on or resize the block.

You can enable content preview on:

- Subsystem blocks
- Model blocks
- Variant Subsystem blocks
- Stateflow charts, subcharts, and graphical functions

In some cases, content preview does not appear when it is enabled. Content preview does not appear for:

- Hierarchical block icons when they are smaller than their default size in the Library Browser
- Masked blocks whose **Icon transparency** property is set to `Opaque` or `Opaque with ports`
- Masked blocks that have a mask icon image
- Subsystem blocks when they have the **Read/Write permissions** block parameter set to `NoReadOrWrite`
- Model blocks whose referenced models are protected models
- Model blocks whose referenced models are not loaded
- Models that have the classic diagram theme enabled in Simulink Editor preferences

Note A slight delay can occur when drawing models with many hierarchical elements that contain many blocks and have content preview enabled.

Programmatic Use

Parameter: `ContentPreviewEnabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Model | Subsystem | Variant Subsystem, Variant Model

Bookmark Parts of Model

In this section...

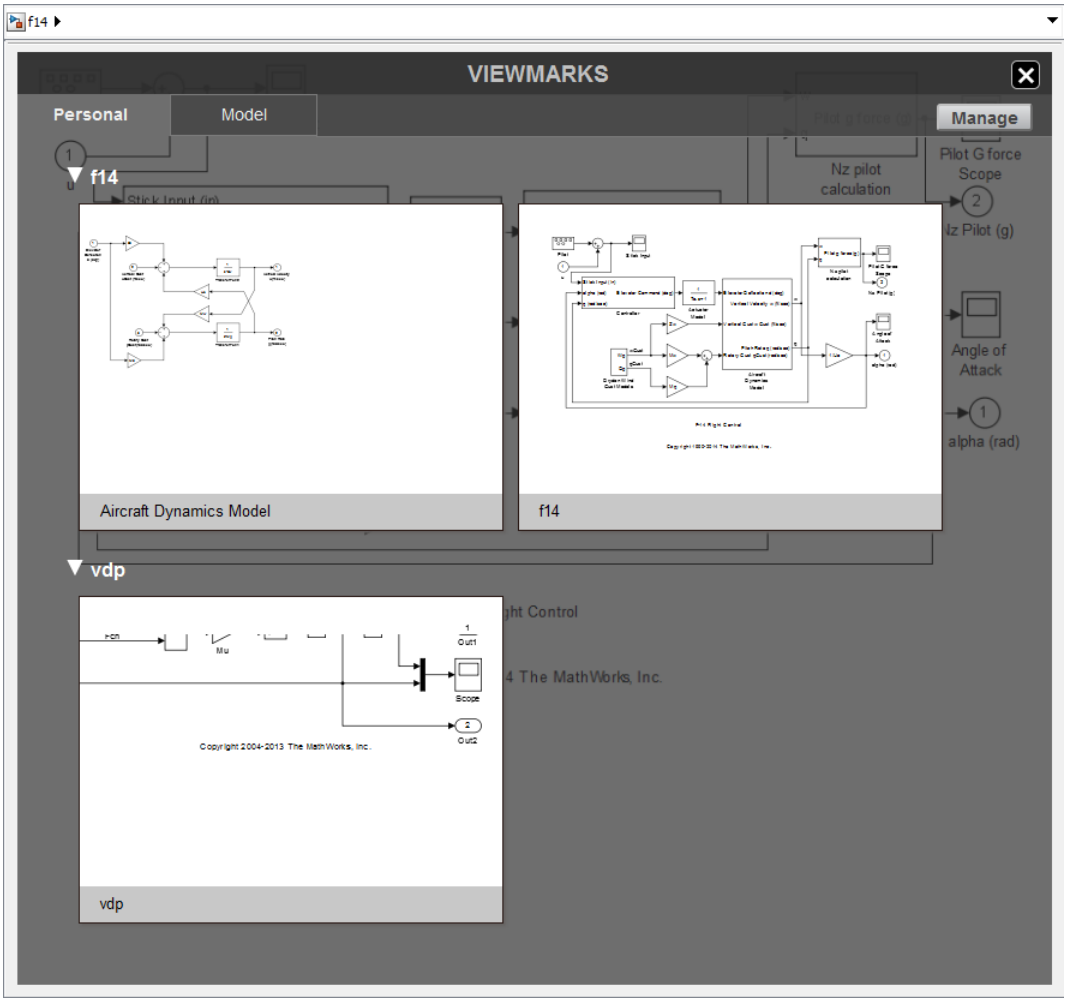
What Are Viewmarks?

Viewmarks are bookmarks to parts of a model. Use viewmarks to capture graphical views of a model so you can navigate directly to that view. You can capture viewmarks for specific levels in a model hierarchy. You can also pan and zoom to capture a point of interest.

Some examples of ways you can use viewmarks include:

- Navigate to specific locations in complex models without opening multiple Simulink Editor tabs or windows.
- Review model designs.
- Visually compare versions of a model.
- Share views of a model by storing viewmarks within the model.

You manage viewmarks in the viewmarks gallery. By default, viewmarks are stored locally on your computer. If you want to include a viewmark to share with a model, see “Save a Viewmark with the Model” on page 1-37. The figure shows the viewmark gallery.



Create a Viewmark

1. Navigate to the part of the model that you want to capture in a viewmark.
2. Pan and zoom to the part of the system that you want to capture.
3. Resize the Simulink Editor window so that it frames the part of the model you want to capture.
4. In the palette, click the **Viewmark This View** button .

The viewmark displays briefly and becomes part of the viewmarks gallery.

Open and Navigate Viewmarks

The viewmarks gallery has two tabs. The **Personal** tab consists of viewmarks that you created in a model and are stored locally on your computer. The **Model** tab consists of viewmarks that are saved in the Simulink model file.

1. In the Simulink Editor palette, click the **Viewmarks** button .
2. Select the tab (**Personal** or **Model**) that contains your viewmark, and then click the viewmark.

The Simulink Editor opens the model, if necessary, and displays the part of the model captured in the viewmark.

Manage Viewmarks

In the viewmarks gallery, you can rename viewmarks, add or edit a description for viewmark, and delete viewmarks. You also use the viewmarks gallery to save viewmarks with the model. You can manage viewmarks only in models saved in SLX format. In models saved in MDL format, the **Manage** button appears dimmed.

- To rename a viewmark, click the name and edit it.
- To add a description, click the viewmark Description button  and enter a description.
- To delete a viewmark, click the **Delete** button  on the viewmark. To delete all the viewmarks for a model, hover over the model name and click **Delete**.

You can replace the generated viewmark name.

1. Place the cursor in the viewmark name edit box.
2. Enter the new name.

You can also add a viewmark description. For example, you can add a description of the part of the model in the viewmark or add review comments.

1. Hover over the viewmark.
2. Click the **Description** button  .
3. In the description edit box, enter the description.

Save a Viewmark with the Model

1. In the Simulink Editor palette, click the **Viewmarks** button  .
2. In the viewmarks gallery, click **Manage**.
3. Select the check box in the viewmarks you want to copy to the model.
4. Click the **Add viewmarks to model** button  .

These viewmarks become part of the model and are saved with the model.

Refresh a Viewmark

A viewmark is a static view of a part of a model. For currently loaded models, you can refresh a viewmark so that it reflects the current model. Open the viewmark gallery and click the **Refresh** button  on the viewmark.

If the viewmark shows a subsystem that has been removed, then the viewmark appears dimmed.

See Also

“Print to a PDF” on page 1-57

Update Diagram and Run Simulation

Updating the Diagram

You can leave many attributes of a block diagram, such as signal data types and sample times, unspecified. The Simulink software then infers the values of block diagram attributes, based on the block connectivity and attributes that you specify. The process that Simulink uses is known as updating the diagram.

Simulink attempts to infer the most appropriate values for attributes that you do not specify. If Simulink cannot infer an attribute, it halts the update and displays an error.

Simulation Updates the Diagram

Simulink updates the diagram at the start of a simulation. The updated diagram provides the simulation with the results of the latest changes that you have made to a model.

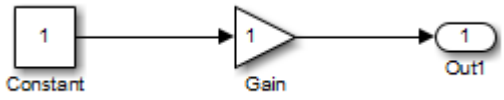
Update Diagram While Editing

As you create a model, at any point you can update the diagram. Updating the diagram periodically can help you to identify and fix potential simulation issues as you develop the model. This approach can make it easier to identify the sources of problems by focusing on a set of recent changes. Also, the update diagram processing takes less time than performing a simulation, so you can identify issues more efficiently.

To update the diagram, from the **Modeling** tab, click **Update Model**, or press **Ctrl+D**.

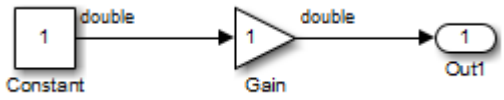
This example shows the effects of updating the diagram.

1. Create the following model.



2. On the **Debug** tab, select **Information Overlays** > **Base Data Types**.

   The data types of the output ports of the Constant and Gain blocks appear. The data type of both ports is `double`, the default value.

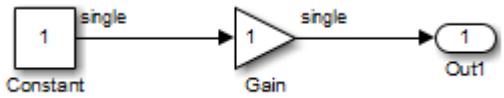


3. In the Constant block dialog box, set **Output data type** to `single`.

   The output port data type displays on the block diagram do not show this change.

4. On the **Modeling** tab, click **Update Model**.

   The updated block diagram shows the changes to the output data types of the Constant and Gain blocks.



When you update a block diagram, by default, a Gain block chooses an output data type based on the data types of the input signal and the **Gain** parameter. In this example, the block chooses the same data type as the input signal.

Simulate a Model

For any type of model you build in Simulink, you need to know how to simulate. Simulating performs the operations specified by the blocks in the model and its specific configuration and produces results. See “Simulation” for complete information, such as how to configure your model for simulation.

Use either of these methods to simulate a model:

- Press **Ctrl+T**.
- Click **Run**  .

See Also

Related Examples

- “Simulate a Model Interactively” on page 25-2
- “Run Multiple Simulations” on page 27-2
- “Run Simulations Programmatically” on page 26-2

Print Model Diagrams

In this section...

Print Interactively or Programmatically

You can print a block diagram:

- Interactively, in the Simulink Editor, on the **Simulation** tab, click **Print**
- Programmatically, use the MATLAB `print` command

To control some additional aspects of printing a block diagram, use the `set_param` command with model parameters. You can use `set_param` with the interactive and programmatic printing interface.

Printing Options

In addition to printing a model using default settings, you can:

- “Select the Systems to Print” on page 1-45.
- “Specify the Page Layout and Print Job” on page 1-47
- “Print Multiple Pages for Large Models” on page 1-49
- “Print Using Print Frames” on page 79-12
- “Print to a PDF” on page 1-57
- “Add a Log of Printed Models” on page 1-50
- “Add a Sample Time Legend” on page 1-51
- “Print Models to Image File Formats” on page 1-60

Canvas Color

By default, the canvas (background) of the printed model is white. To match the color of the model, set the **Simulink Preferences > General > Print** preference.

See Also

`print`

Related Examples

- “Basic Printing” on page 1-42
- “Print from the MATLAB Command Line” on page 1-52
- “Print Model Reports” on page 1-58

More About

- “Tiled Printing” on page 1-48
- “Print Frames” on page 79-2

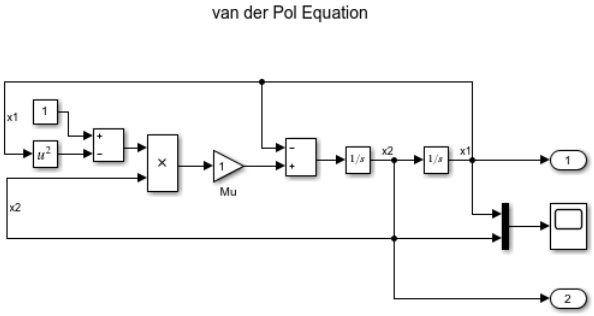
Basic Printing

In this section...

Print the vdp Model Using Default Settings

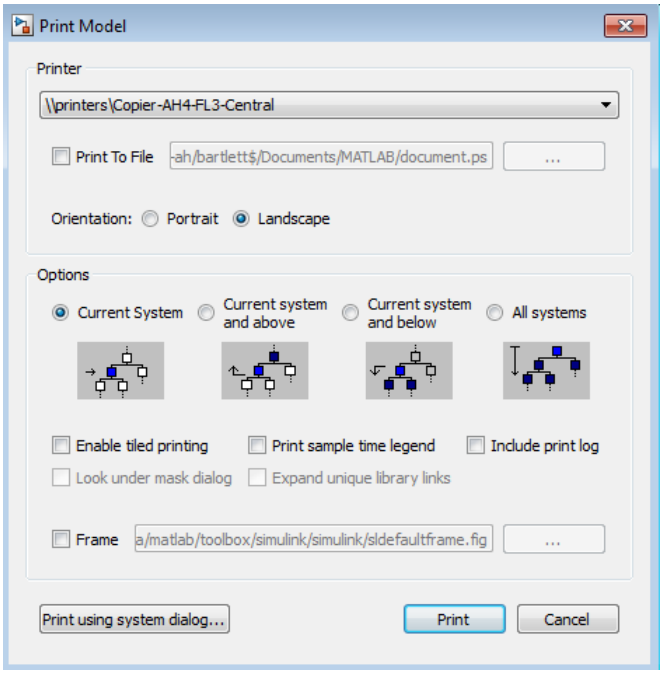
The default print settings produce good quality printed output for quickly capturing a model in printed form.

1 Open the vdp model.

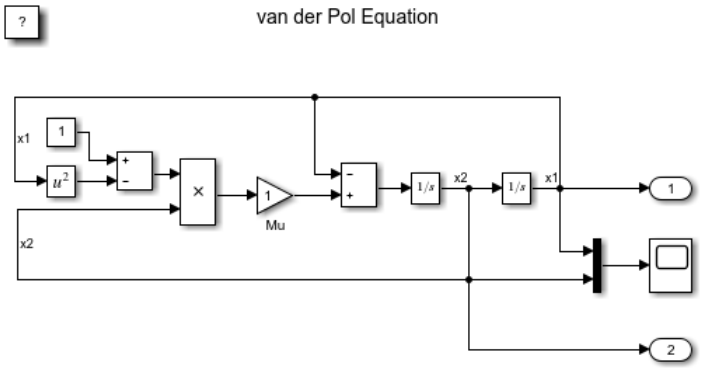


2 In the Simulink Editor, on the **Simulation** tab, click **Print**.

3 In the Print Model dialog box, use the default settings. Click **Print**.



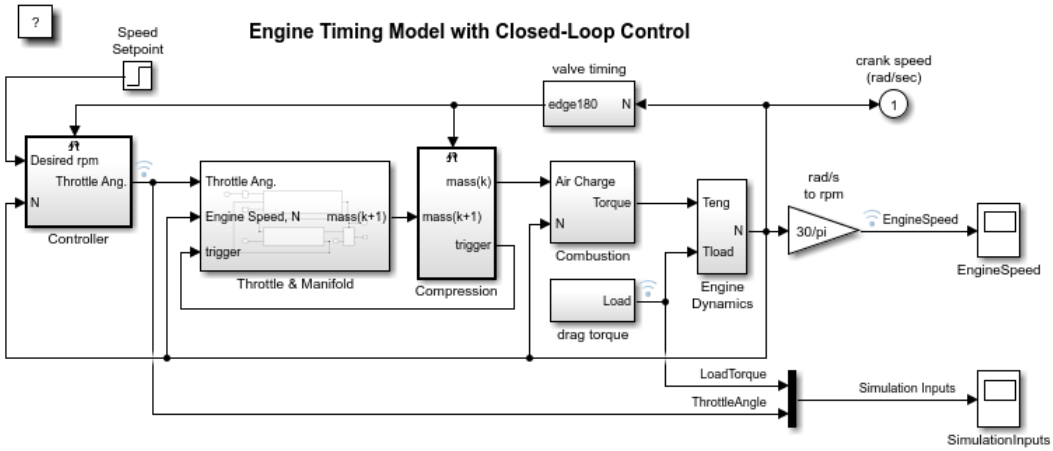
The output looks like this. The model, as it appears in the Simulink Editor, prints on a single page, using portrait orientation and not using a print frame.



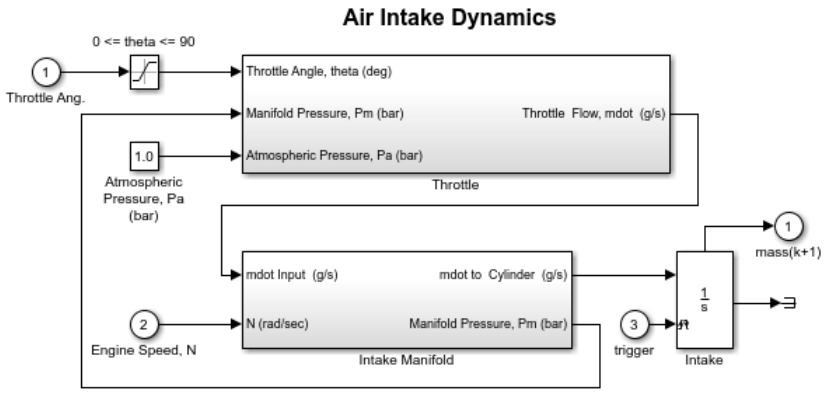
Print a Subsystem Hierarchy

You can print levels in nested subsystems.

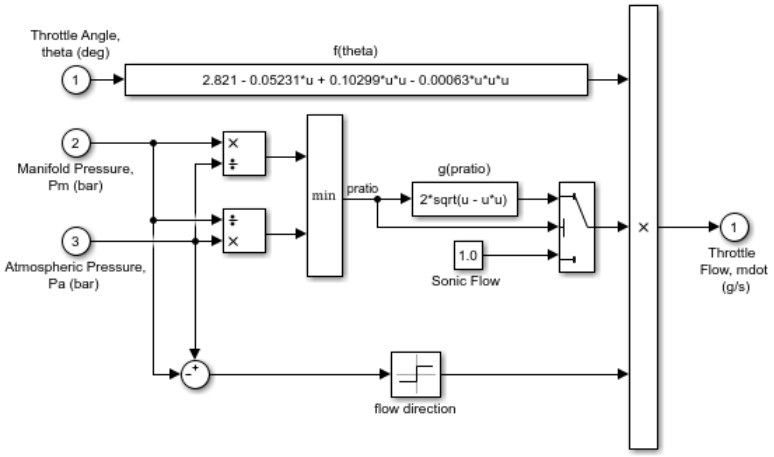
1 Open the `sldemo_enginewc` model.



2 Open the Throttle & Manifold subsystem.



3 Open the Throttle subsystem.



4 In the Simulink Editor, on the **Simulation** tab, click **Print**.

5 In the Print Model dialog box, select **Current system and above** and click **Print**.

The printed output shows the Throttle subsystem (the current system) and the two levels above it in the subsystem hierarchy.

See Also

Related Examples

- “Print from the MATLAB Command Line” on page 1-52
- “Specify the Page Layout and Print Job” on page 1-47
- “Print Subsystems” on page 1-45

More About

- “Print Model Diagrams” on page 1-40

Select the Systems to Print

In this section...

Print Current System

To select a specific system in a model to print, display that system in the currently open Simulink Editor tab. On the **Simulation** tab, click **Print**.

Print Subsystems

For models with subsystems, use the Simulink Editor and the Print Model dialog box to specify the systems in the model to print.

By default, Simulink does not print masked subsystems or library links. For information about masked subsystem and library link printing, see “Print Masked Subsystems and Library Links” on page 1-46.

Print All Subsystems in a Model

Use this procedure to print all of the subsystems in a model, including hierarchies of subsystems.

1. Display the top-level model in the currently open Simulink Editor tab.
2. In the Simulink Editor, on the **Simulation** tab, click **Print**.
3. In the Print Model dialog box, select **All systems**.
4. Click **Print**.

Print the Contents of a Specific Subsystem

In the currently open Simulink Editor tab, display the subsystem that you want to print and click **Print**.

Print a Subsystem Hierarchy

Use this procedure to print nested subsystems.

1. In the current tab of the Simulink Editor, display the subsystem level that you want to use as the starting point for printing the subsystem hierarchy.
2. In the Print Model dialog box, select one of the following:
   - **Current system and below**
   - **Current system and above**
3. Click **Print**.

Simulink prints the hierarchy for all of the subsystems in the current tab.

Print Masked Subsystems and Library Links

To print the contents of masked subsystems, in the Print Model dialog box, click **Look under mask dialog**.

To print the contents of library links, in the Print Model dialog box, click **Expand unique library links**. Simulink prints one copy, regardless of how many copies of the block the model contains.

If a subsystem is both a masked subsystem and a library link, Simulink uses the **Look under mask dialog** setting and ignores the **Expand unique library links** setting.

Print a Model Referencing Hierarchy

To print a model referencing hierarchy, open each level of the hierarchy and print that level.

Clicking **All systems** does not print different levels in the model referencing hierarchy.

You cannot print the contents of protected models.

See Also

More About

- “Design Model Architecture”

Specify the Page Layout and Print Job

In this section...

Page and Print Job Setup

Use the Print Model dialog box to specify the page orientation (portrait or landscape) for the current printing session.

To open the print dialog box for your operating system, in the Print Model dialog box, click **Print using system dialog**. The operating system print dialog box provides additional printing options for models, such as page range, copies, double-sided printing, printing in color (if your print driver supports color printing), and nonstandard paper sizes.

Set Paper Size and Orientation Without Printing

To specify paper size and orientation without printing, use the Page Setup dialog box. To open the dialog box, on the **Simulation** tab, select **Print > Page Setup**.

Only the paper size and orientation are used.

See Also

Related Examples

- “Basic Printing” on page 1-42

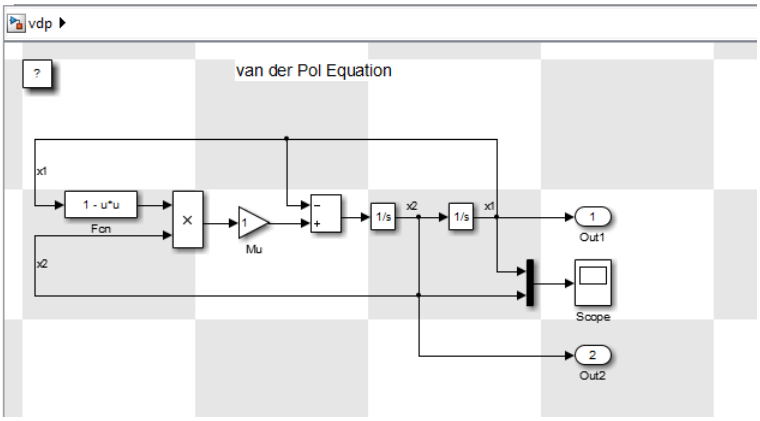
More About

- “Print Model Diagrams” on page 1-40

Tiled Printing

By default, each block diagram is scaled during the printing process so that it fits on a single page. In the case of a large diagram, this automatic scaling can make the printed image difficult to read.

Tiled printing enables you to print even the largest block diagrams without sacrificing clarity and detail. Tiled printing allows you to distribute a block diagram over multiple pages. For example, you can use tiling to divide a model as shown in the figure, with each white box and each gray box representing a separate printed page.



You can control the number of pages over which Simulink prints the block diagram.

Also, you can set different tiled-print settings for each of the systems in your model.

Note If you enable the print frame option, then Simulink does not use tiled printing.

See Also

Related Examples

- “Print Using Print Frames” on page 79-12
- “Print Multiple Pages for Large Models” on page 1-49

Print Multiple Pages for Large Models

1. In the Simulink Editor, open the model in the current tab.
2. On the **Simulation** tab, click **Print**.
3. In the Print Model dialog box, select **Enable tiled printing**.

   The default **Enable tiled printing** setting in the Print Model dialog box is the same as the **Print > Enable Tiled Printing** setting. If you change the Print Model dialog box **Enable tiled printing** setting, the Print Model dialog box setting takes precedence.
4. Confirm that tiling divides the model into separate pages the way you want it to appear in the printed pages. In the Simulink Editor, on the **Simulation** tab, select **Print > Show Page Boundaries**. The gray and white squares indicate the page boundaries.
5. Optionally, from the MATLAB command line, specify the model scaling, tile margins, or both. See “Set Tiled Page Scaling and Margins” on page 1-55.
6. Optionally, specify a subset of pages to print. In the Print Model dialog box, specify the **Page Range**.
7. Click **Print**.

See Also

Related Examples

- “Use Tiled Printing” on page 1-55

More About

- “Tiled Printing” on page 1-48

Add a Log of Printed Models

A print log lists the blocks and systems printed. To print the print log when you print a model:

1. In the Simulink Editor, open the model whose print job you want to log.
2. On the **Simulation** tab, click **Print**.
3. In the Print Model dialog box, select **Include print log**.
4. Click **Print**.

The print log appears on the last page.

For example, here is the print log for the `sldemo_enginewc` model, with **All systems** enabled and **Enable tiled printing** cleared.

```
  Page      System Name
-----------------------------------------------------------------
--------------
     1      sldemo_enginewc
     2      sldemo_enginewc/Combustion
     3      sldemo_enginewc/Compression
     4      sldemo_enginewc/Controller
     5      sldemo_enginewc/Engine Dynamics
     6      sldemo_enginewc/More Info
     7      sldemo_enginewc/Throttle & Manifold
     8      sldemo_enginewc/Throttle & Manifold/Intake Manifold
     9      sldemo_enginewc/Throttle & Manifold/Throttle
    10      sldemo_enginewc/drag torque
    11      sldemo_enginewc/valve timing
    12      sldemo_enginewc/valve timing/TDC and BDC detection
    13      sldemo_enginewc/valve timing/positive edge to dual
edge conversion
```

See Also

More About

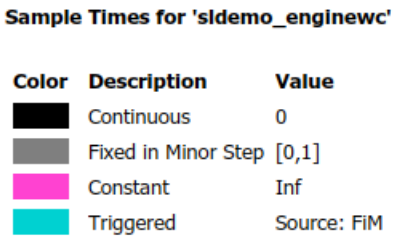
- “Print Model Diagrams” on page 1-40

Add a Sample Time Legend

You can print a legend that contains sample time information for your entire system, including any subsystems. The legend appears on a separate page from the model. To print a sample time legend:

1. In the Simulink Editor, on the **Modeling** tab, click **Update Model**.
2. On the **Simulation** tab, click **Print**.
3. In the Print Model dialog box, select **Print sample time legend**.
4. Click **Print**.

A sample time legend appears on the last page. For example, here is the sample time legend for the `sldemo_enginewc` model, with **All systems** enabled.



Sample Times for 'sldemo_enginewc'

Color	Description	Value
	Continuous	0
	Fixed in Minor Step	[0,1]
	Constant	Inf
	Triggered	Source: FiM

See Also

Related Examples

- “View Sample Time Information” on page 7-9

More About

- “Print Model Diagrams” on page 1-40

Print from the MATLAB Command Line

In this section...

Printing Commands

The MATLAB `print` command provides several options for printing Simulink models. For example, print the `Compression` subsystem in the `sldemo_enginewc` model to your default printer:

```
open_system('sldemo_enginewc');
print -sCompression
```

Tip When you use the `print` command, you can print only one system. To print multiple levels in a model, use multiple `print` commands, one for each system that you want to print. To print multiple systems in a model, consider using the Print Model dialog box in the Simulink Editor. For details, see “Select the Systems to Print” on page 1-45.

You can use `set_param` and the following parameters to specify printing options for models.

Model Parameters for Printing

Parameter	Description	Values
`PaperOrientation`	Printing paper orientation.	`'portrait' | {'landscape'}`
`PaperPosition`	When `PaperPositionMode` is set to `manual`, this parameter determines the position and size of a diagram on paper and the size of the diagram exported as a graphic file in the units specified by `PaperUnits`.	vector — `[left, bottom, width, height]`
`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” on page 1-48 for more information.	`{'auto'} | 'manual' | 'tiled'`
`PaperSize`	Size of `PaperType` in `PaperUnits`.	vector — `[width height]` (read only)

Parameter	Description	Values
PaperType	Printing paper type.	'usletter' | 'uslegal' | 'a0' | 'a1' | 'a2' | 'a3' | 'a4' | 'a5' | 'b0' | 'b1' | 'b2' | 'b3' | 'b4' | 'b5' | 'arch-A' | 'arch-B' | 'arch-C' | 'arch-D' | 'arch-E' | 'A' | 'B' | 'C' | 'D' | 'E' | 'tabloid'
PaperUnits	Printing paper size units.	'normalized' | {'inches'} | 'centimeters' | 'points'
TiledPageScale	Scales the size of the tiled page relative to the model.	{'1'}
TiledPaperMargins	Controls the size of the margins associated with each tiled page. Each element in the vector represents a margin at the particular edge.	vector — [left, top, right, bottom]

You can use `orient` to control the paper orientation.

Print Systems with Multiline Names or Names with Spaces

To print a system whose name appears on multiple lines, assign the newline character to a variable and use that variable in the `print` command. This example shows how to print a subsystem whose name, `Aircraft Dynamics Model`, appears on three lines.

```
open_system('f14');
open_system('f14/Aircraft Dynamics Model');
sys = sprintf('f14/Aircraft\nDynamics\nModel');
print (['-s' sys])
```

To print a system whose name includes one or more spaces, specify the name as a character vector. For example, to print the Throttle & Manifold subsystem, enter:

```
open_system('sldemo_enginewc');
open_system('sldemo_enginewc/Throttle & Manifold');
print (['-sThrottle & Manifold'])
```

Set Paper Orientation and Type

To set just the paper orientation, use the MATLAB `orient` command.

You can also set the paper orientation by using `set_param` with the `PaperOrientation` model parameter. Set the paper type with the `PaperType` model parameter.

Position and Size a System

To position and size the model diagram on the printed page, use `set_param` command with the `PaperPositionMode` and `PaperPosition` model parameters.

The value of the `PaperPosition` parameter is a vector of form `[left bottom width height]`. The first two elements specify the bottom-left corner of a rectangular area on the page, measured from the bottom-left corner. The last two elements specify the width and height of the rectangle.

If you set the `PaperPositionMode` parameter to `manual`, Simulink positions (and scales, if necessary) the model to fit inside the specified print rectangle. If `PaperPositionMode` is `auto`, Simulink centers the model on the printed page, scaling the model, if necessary, to fit the page.

For example, to print the `vdp` model in the lower-left corner of a U.S. letter-size page in landscape orientation:

```
open_system('vdp');
set_param('vdp', 'PaperType', 'usletter');
set_param('vdp', 'PaperOrientation', 'landscape');
set_param('vdp', 'PaperPositionMode', 'manual');
set_param('vdp', 'PaperPosition', [0.5 0.5 4 4]);
print -svdp
```

Use Tiled Printing

Enable Tiled Printing

1. Use `set_param` to set the `PaperPositionMode` parameter to `tiled`.
2. Use the `print` command with the `-tileall` argument.

For example, to enable tiled printing for the `Compression` subsystem in the `sldemo_enginewc` model:

```
open_system('sldemo_enginewc');
set_param('sldemo_enginewc/Compression', 'PaperPositionMode', ...
'tiled');
print('-ssldemo_enginewc/Compression', '-tileall')
```

Display Tiled Page Boundaries

To display the page boundaries programmatically, use the `set_param` command, with the model parameter `ShowPageBoundaries` set to `on`. For example:

```
open_system('sldemo_enginewc');
set_param('sldemo_enginewc', 'ShowPageBoundaries', 'on')
```

Set Tiled Page Scaling and Margins

To scale the block diagram so that more or less of it appears on a single tiled page, use `set_param` with the `TiledPageScale` parameter. By default, the value is 1. Values greater than 1 proportionally scale the model to use a smaller percentage of the tiled page, while values between 0 and 1 proportionally scale the model to use a larger percentage of the tiled page. For example, a `TiledPageScale` of `0.5` makes the printed diagram appear twice its size on a tiled page, while a `TiledPageScale` value of 2 makes the printed diagram appear half its size on a tiled page.

By decreasing the margin sizes, you can increase the printable area of the tiled pages. To specify the margin sizes associated with tiled pages, use `set_param` with the `TiledPaperMargins` parameter. Each margin to 0.5 inches by default. The value of `TiledPaperMargins` is a vector that specifies margins in this order: `[left top right bottom]`. Each element specifies the size of the margin at

a particular edge of the page. The value of the `PaperUnits` parameter determines the units of measurement for the margins.

Specify Range of Tiled Pages to Print

To specify a range of tiled page numbers programmatically, use `print` with the `-tileall` argument and the `-pages` argument. Append to `-pages` a two-element vector that specifies the range.

Note Simulink uses a row-major scheme to number tiled pages. For example, the first page of the first row is 1, the second page of the first row is 2, and so on.

For example, to print the second, third, and fourth pages:

```
open_system('vdp');
print('-svdp','-tileall','-pages[2 4]')
```

See Also

`orient` | `print`

Related Examples

- “Select the Systems to Print” on page 1-45

Print to a PDF

You can print a model to a `.pdf` file. Simulink creates one file for all of the systems in the model.

1. In the Simulink Editor, on the **Simulation** tab, click **Print**.
2. Select **Print to File**.
3. Specify a location and file name to save the new `.pdf` file. Include the extension `.pdf` in the file name.
4. Click **Print**.

See Also

More About

- “Print Model Diagrams” on page 1-40

Print Model Reports

A model report is an HTML document that describes the structure and content of a model. The report includes block diagrams of the model and its subsystems and the settings of its block parameters.

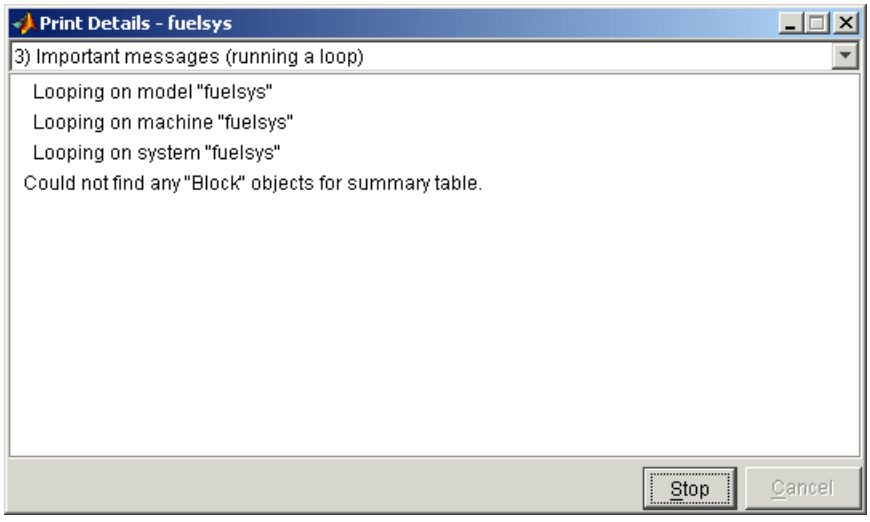
Tip If you have the Simulink Report Generator™ installed, you can generate a detailed report about a system. To do so, in the Simulink Editor, on the **Modeling** tab, select **Compare > System Design Description Report**. For more information, see "System Design Description" (Simulink Report Generator).

To generate a model report for the current model:

1. In the Simulink Editor, on the **Simulation** tab, select **Print > Print Details**.
2. In the Print Details dialog box, select report options. For details, see "Model Report Options" on page 1-58.
3. Click **Print**.

The Simulink software generates the HTML report and displays the report in your default HTML browser.

While generating the report, Simulink displays status messages on a messages pane that replaces the options pane on the Print Details dialog box.



Select the detail level of the messages from the list at the top of the messages pane. When the report generation process begins, the **Print** button changes to a **Stop** button. To stop the report generation, click **Stop**. When the report generation process finishes, the **Stop** button changes to an **Options** button. Clicking this button redisplays the report generation options, allowing you to generate another report without having to reopen the Print Details dialog box.

Model Report Options

Use the Print Details dialog box allows you to specify the following report options.

Directory

The folder where the HTML report is stored. The options include your system's temporary folder (the default), your system's current folder, or another folder whose path you specify in the adjacent edit field.

Increment filename to prevent overwriting old files

Creates a unique report file name each time you generate a report for the same model in the current session. This preserves each report.

Current object

Include only the currently selected object in the report.

Current and above

Include the current object and all levels of the model above the current object in the report.

Current and below

Include the current object and all levels below the current object in the report.

Entire model

Include the entire model in the report.

Look under mask dialog

Include the contents of masked subsystems in the report.

Expand unique library links

Include the contents of library blocks that are subsystems. The report includes a library subsystem only once even if it occurs in more than one place in the model.

See Also

More About

- “Print Model Diagrams” on page 1-40
- “Masking Fundamentals” on page 39-2

Print Models to Image File Formats

To print the current view of your model to an image file format such as `.png` or `.jpeg`, use the `-device` argument with the MATLAB `print` command. For example, to print the `vdp` model to a `.png` format, use this command:

```
print -dpng -svdp vdp_model.png
```

By default, the canvas (background) of the exported model matches the color of the model. To use a white or transparent canvas for model files that you export to another file format, set the **Simulink Preferences** > **Export** preference.

Copy Model Views to Third-Party Applications

On Microsoft® Windows® platforms, you can copy the current view of your model in either bitmap or metafile format. You can then paste the clipboard image to a third-party application such as word processing software.

On Macintosh platforms, when you copy a model view to the clipboard, Simulink saves the model in a scalable format, in addition to a bitmap format. When you paste from the clipboard to an application, the application selects the format that best meets its requirements.

By default, the canvas (background) of the copied model matches the color of the model. To use a white or transparent canvas for model files that you copy to another application, set the **Simulink Preferences** > **Clipboard** preference.

1. To copy a Simulink model to the operating system clipboard, in the Simulink Editor, on the **Format** tab, select **Screenshot**, then the desired format.
2. Paste the model from the clipboard to a third-party application.

See Also

Related Examples

- “Set Simulink Preferences”

Keyboard Shortcuts and Mouse Actions for Simulink Modeling

Keyboard shortcuts and mouse actions can help you efficiently model systems in Simulink. To open this page from a model, use **Shift + ?** or click the **Keyboard Shortcuts** button  on the navigation bar.

Note The following keyboard shortcuts are based on Windows. On Mac keyboards, use **command** (⌘) instead of **Ctrl**.

Perform File and Clipboard Operations

Task	Shortcut
Open model	**Ctrl** + **O**
Create model	**Ctrl** + **N**
Print model	**Ctrl** + **P**
Save all modified models in current model hierarchy	**Ctrl** + **S**
Save current referenced model	**Ctrl** + **Shift** + **S**
Close model	**Ctrl** + **W**
Cut	**Ctrl** + **X**
Paste	**Ctrl** + **V**
Duplicate	**Ctrl** + **C**, **Ctrl** + **V**
Undo	**Ctrl** + **Z**
Redo	**Ctrl** + **Y**
Find search string	**Ctrl** + **F**

Zoom, Scroll, and Change Current Window

Task	Action
Fit diagram to screen	Spacebar
Zoom in	**Ctrl** + **+**
Zoom out	**Ctrl** + **-**
Zoom to 100%	**Ctrl** + **0** or **Alt** + **1**
Zoom with scroll wheel	Scroll wheel **Ctrl** + scroll wheel when **Scroll wheel controls zooming** is cleared
Zoom in on object	Drag the **Zoom** button  from the palette to the object.

Task	Action
Scroll	Spacebar + drag. Scroll wheel + drag.
Change tabs	**Ctrl** + **Shift** + **Tab**
Go to previous window	**Alt** + **Tab**

Navigate Model Hierarchy

Task	Shortcut
Open	Double-click. **Enter**
Go to parent of current subsystem or referenced model	**Esc**
Look under block mask	**Ctrl** + **U**
For linked blocks, go to library of parent block	**Ctrl** + **L**
Open the **Model Explorer**	**Ctrl** + **H**

Modify Block Diagram Contents

Task	Action
Insert block at current location	Double-click and start typing the block name, then select the block from the menu. **Ctrl** + **. (period)** twice and start typing the block name, then use the down arrow key to select the block from the menu and **Enter** to insert the block. On AZERTY keyboards, use **Ctrl** + **Shift** + **. (period)** instead of **Ctrl** + **. (period)**.
Get prompted for suggested block based on context	Double-click the end of a line drawn from an existing block, then select a suggestion from the list or start typing to select one from the library.
Open the **Library Browser**	**Ctrl** + **Shift** + **L**
Insert annotation at current location	Double-click and type the annotation content, then select the annotation option. **Ctrl** + **. (period)** twice and type the annotation content, then use the down arrow key to select the annotation option and **Enter** to insert the annotation. On AZERTY keyboards, use **Ctrl** + **Shift** + **. (period)** instead of **Ctrl** + **. (period)**.

Task	Action
Copy selected objects	Drag objects with the right mouse button. **Ctrl** + drag.
Copy selected objects between Simulink Editor windows	Drag objects between windows.
Delete selected objects	**Delete** or **Backspace**

Select Objects

Objects include blocks, signal lines, signal labels, and annotations. You cannot select lines and ports with the arrow keys.

Task	Action
Select an object and clear selection from other objects	Click. Arrow keys (**M** to toggle between moving selection and moving objects)
Select multiple adjacent objects	Drag from empty position. **Shift** + arrow keys
Select multiple objects that are not adjacent	**Shift** + click. **Ctrl** + arrow keys to change the current object, **Ctrl** + spacebar to add the current object to the selection On macOS, to add the current object to the selection, use **command** + **shift** + spacebar or **shift** + spacebar.
Select all blocks, lines, and annotations	**Ctrl** + **A**

Modify Objects

Task	Shortcut
Set main parameter for selected block	**Alt** + **Enter**
Open or hide the **Property Inspector**	**Ctrl** + **Shift** + **I**
Connect blocks	Click a port, then click the port you want it to connect to. Select the first block, then **Ctrl** + click a second block. Drag from port to port. Click a port and hold **Shift** as you connect to the next port. Hold **Shift** to make multiple, consecutive connections.

Task	Shortcut
Branch line	Click a port, move the cursor near the line you want to branch, and then click after you see the preview. Select a line, move the cursor toward the element you want to connect with, and then click the port. **Ctrl** + drag line. Right mouse button + drag.
Add ports to blocks that take multiple inputs	Drag toward the port from a compatible block or signal line. Drag from the edge of the block that you want to add the port to. Click the edge of the port, and then drag. Click the edge of the port, move the cursor over the signal type — signal, bus, or connector — from the menu, and drag.
Comment through block	**Ctrl** + **Shift** + **Y**
Comment out or uncomment block	**Ctrl** + **Shift** + **X**
Disconnect block	**Shift** + drag block.

Name Objects

Task	Action
Rename selected object	Click or double-click the label. Select the object and use **F2**. On Mac keyboards, you can use **command** + **return** instead of **F2**.
Name signal line	Double-click the signal and type its name.
Display name on branch of a named signal line	Double-click the branch.
Name every branch of a signal	Right-click the signal, select **Properties**, and use the dialog box.
Delete signal label and name	Delete characters in the label, or delete the name in Signal Properties dialog box.
Delete signal label only	Right-click the label and select **Delete Label**.
Copy signal label	**Ctrl** + drag the signal label.

Modify Block Diagram Appearance

Task	Action
Move any object, including signal labels	Drag object.
Move selected blocks and annotations	Arrow keys (**M** to toggle between moving selection and moving objects)
Resize block, maintaining ratio of width and height	**Shift** + drag handle.
Resize block from the center	**Ctrl** + drag handle.
Route lines around blocks	**Shift** + drag while drawing.
Rotate blocks clockwise	**Ctrl** + **R**
Rotate blocks counterclockwise	**Ctrl** + **Shift** + **R**
Flip blocks	**Ctrl** + **I**
Create subsystem from selection	**Ctrl** + **G**
Create subsystem or area from empty selection	To create the highlighted option, which you can change by using the arrow keys or mouse: • Click inside the selection box. • Press **Enter**. Press the number key associated with an option.
Change signal label font	Select the signal line (not the label), then on the **Format** tab, click the **Font Properties** button arrow, then click **Fonts for Model**.
Mask block	**Ctrl** + **M**
Refresh Model blocks	**Ctrl** + **K**

Perform Actions

Task	Shortcut
Perform generic action	Double-click, select the **Actions** tab, and start typing the action name or description, then select the action from the menu. **Ctrl** + **. (period)** and start typing the action name or description, then use the down arrow key to select the action and **Enter** to perform it. On AZERTY keyboards, use **Ctrl** + **Shift** + **. (period)** instead of **Ctrl** + **. (period)**.
Perform context-sensitive action	Select object before opening action menu.

Update, Simulate, and Generate Code for Models

Task	Shortcut
Open Configuration Parameters dialog box	**Ctrl** + **E**
Update model	**Ctrl** + **D**
Start simulation	**Ctrl** + **T**
Stop simulation	**Ctrl** + **Shift** + **T**
Build model (for code generation)	**Ctrl** + **B**

Debug Models

Task	Shortcut
Step	**F10**
Step in	**F11**
Step out	**Shift** + **F11**
Run	**F5**
Set or clear breakpoint	**F12**

See Also

Library Browser | Simulink Editor

More About

- “Build and Edit a Model Interactively” on page 1-8

Simulation Stepping

How Simulation Stepper Helps With Model Analysis

Simulation Stepper enables you to step through major time steps of a simulation. Using discrete time steps, you can step forward or back to a particular instant in simulation time. At each time step, Stepper displays all of the simulation data the model produces.

Use Simulation Stepper to analyze your model in these ways:

- Step forward and back through a simulation.
- Pause a simulation in progress and step back.
- Continue running a simulation after stepping back.
- Analyze plotted data in your model at a particular moment in simulation time.
- Set conditions before and during simulation to pause a simulation.

See Also

Related Examples

- “Step Through a Simulation” on page 2-12

More About

- “How Stepping Through a Simulation Works” on page 2-3
- “How Simulation Stepper Differs from Simulink Debugger” on page 2-5

How Stepping Through a Simulation Works

In this section...

These topics explain how Simulation Stepper steps through a simulation.

Simulation Snapshots

When you set up Simulation Stepper, you specify:

- The number of time steps where Stepper creates ‘snapshots’
- The number of steps to skip between snapshots
- The total number of snapshots stored

A simulation snapshot contains simulation state (SimState) and information related to logged data and visualization blocks. Simulation Stepper stores simulation states in snapshots at the specified interval of time steps when it steps forward through a simulation.

It is important to understand the difference between a Simulation Stepper step and a simulation time step. A simulation time step is the fixed amount of time by which the simulation advances. A Simulation Stepper step is where Simulation Stepper creates a snapshot. Each step (that Simulation Stepper takes) consists of one or more simulation time steps (that you specify).

When you step back through a simulation, the software uses simulation snapshots, stored as SimStates, to display previous states of the simulation. The model does not simulate in reverse when stepping back. Therefore, to enable the step back capability, you must first simulate the model or step it forward to save snapshots.

Keep in mind that snapshots for stepping back are available only during a single simulation. The Simulation Stepper does not save the steps from one simulation to the next.

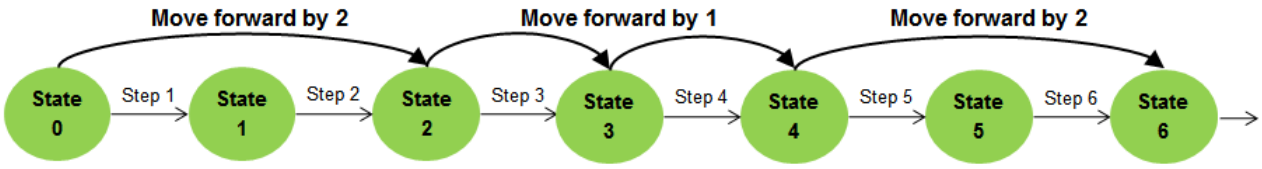
How Simulation Stepper Uses Snapshots

A simulation snapshot captures all the information required to continue a simulation from that point. When you set up simulation stepping, you specify:

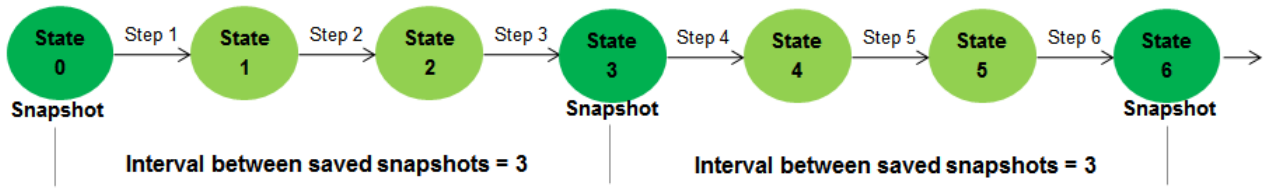
- The maximum number of snapshots to capture while simulating forward. The greater the number, the more memory the simulation uses and the longer the simulation takes to run.
- The number of time steps to skip between snapshots. This setting enables you to save snapshots of simulation state when stepping forward at periodic intervals, such as every three steps. This interval is independent of the number of forward or backward time steps taken. Because taking simulation snapshots affects simulation speed, saving snapshots less frequently can improve simulation speed.

The figure shows how you can step through a simulation depending on how you set the parameters in the Simulation Stepping Options dialog box. Because you can change the stepping parameters as you

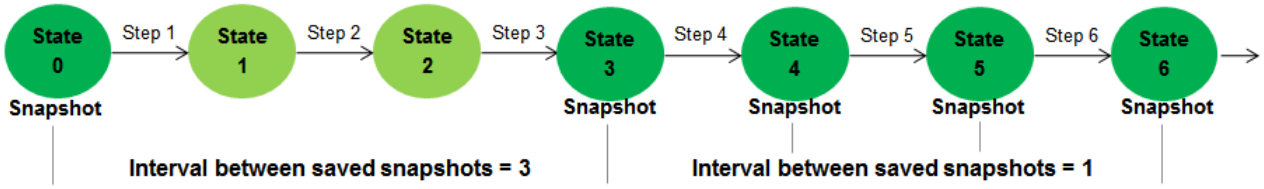
step through the simulation, you can step through a simulation as shown in this figure: sometimes by single steps and sometimes by two or more steps.



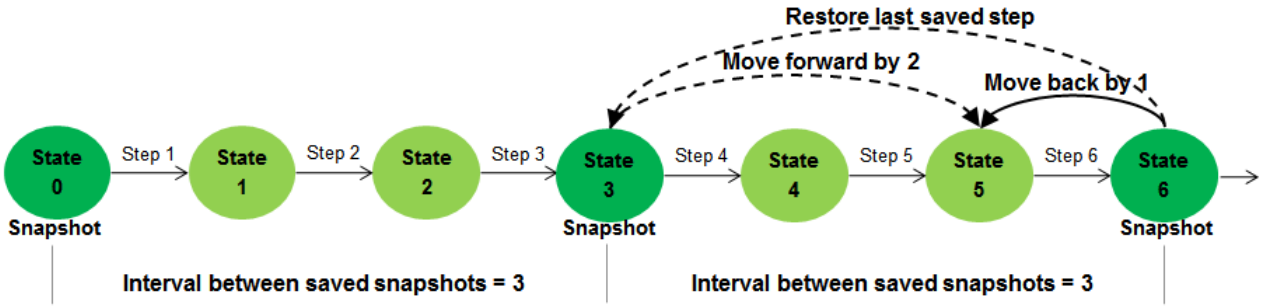
In the figure, the interval for snapshot captures is three.



This next figure shows the advantage of changing the stepping options while stepping forward. At the fourth step, the interval between stored steps changed the snapshot steps from three to one. This enables you to capture more snapshots around a simulation time of interest.



The next figure shows how the snapshot settings of Simulation Stepper can change what happens when stepping back. Suppose that the interval between snapshots is set to three, and starting at state six, the stepper **Move back/forward by** setting is set to one. The stepper first restores the simulation state to the last saved snapshot (state three), and then simulates two major times steps to arrive at the desired state (state five).



Thus, when you step back to a particular time step in a simulation, Simulation Stepper restores the last saved snapshot before that time step. Then, it steps forward to the time step you specify. This capability is helpful for memory usage and simulation performance.

How Simulation Stepper Differs from Simulink Debugger

Simulation Stepper and Simulink Debugger both enable you to start, stop, and step through a model simulation. Both tools allow you to use breakpoints as part of a debugging session. However, you use Simulation Stepper and Simulink Debugger for different purposes. The table shows the actions you can perform with each tool.

Action	Simulation Stepper	Simulink Debugger
Look at state of system after executing a major time step.	✓	✓
Observe dynamics of the entire model from step to step.	✓	
Step simulation back.	✓	
Pause across major steps.	✓	
Control a Stateflow debugging session.	✓	
Step through simulation by major steps.	✓	
Monitor single block dynamics (for example, output and update) during a single major time step.		✓
Look at state of system while executing a major time step.		✓
Observe solver dynamics during a single major step.		✓
Show various stages of Simulink simulation.		✓
Pause within a major step.		✓
Step through a simulation block by block.		✓
Access via a command-line interface.		✓

Understanding the simulation process can help you to better understand the differences between Simulation Stepper and Simulink Debugger.

See Also

Related Examples

- “Step Through a Simulation” on page 2-12

More About

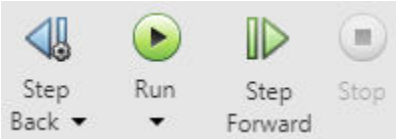
- “Use Simulation Stepper” on page 2-7

Use Simulation Stepper

In this section...

Simulation Stepper Access

You run Simulation Stepper and access the settings from the Simulink Toolstrip.



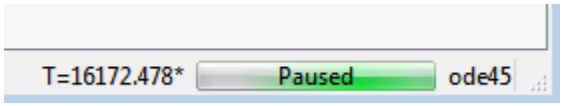
Click the **Configure simulation stepping** button  to open the Simulation Stepping Options dialog box.

Use the dialog box to enable stepping back through a simulation. When stepping back is enabled, after you start the simulation, you can use the **Step Back** button  to step back.

If you clear the **Enable previous stepping** check box, the software clears the stored snapshot cache.

Simulation Stepper Pause Status

The status bar at the bottom of the Simulink Editor displays the simulation time of the last completed simulation step. While a simulation is running, the editor updates the time display to indicate the simulation progress. This display is approximate because the status bar updates only at every major time step and not at every simulation time step. When you pause a simulation, the status bar display time catches up to the actual time of the last completed step.



The value (the time of the last completed step) that is displayed on the status bar is not always the same as the time of the solver. This happens because different solvers use different ways to propagate the simulation time in a single iteration of the simulation loop. Simulation Stepper pauses at a single position within the simulation loop. Some solvers perform their time advance before Simulation Stepper pauses. However, other solvers perform their time advance after Simulation Stepper pauses, and the time advance then becomes part of the next step. As a result, for continuous and discrete solvers, the solver time is always one major step ahead of the time of the last model output.

When this condition occurs, and the simulation is paused, the status bar time displays an asterisk. The asterisk indicates that the solver in this simulation has already advanced past the displayed time (which is the time of the last completed simulation step).

Tune Parameters

While using Simulation Stepper, when the simulation is paused, you can change tunable parameters, including some solver settings. However, changes to the solver step size take effect when the solver advances the simulation time. For some solvers, this occurs after the next simulation step is taken.

Simulation Stepper takes into account the size of a movement (**Move back/forward by**) and the frequency of saving steps (**Interval between stored back steps**). If you specify a frequency that is larger than the step size, Simulation Stepper first steps back to the last saved step and then simulates forward until the total step count difference reaches the size of the desired movement. Simulation Stepper applies values for tunable parameters when simulating forward. For this reason, if you change any tunable parameter before stepping back, the resulting simulation output might not match the previous simulation output at that step before the parameter change. This can cause unexpected results when stepping forward from the snapshot to the chosen time step.

For example, assume a snapshot save frequency of three and a step size of one. The stepper first steps back to the last saved step, up to three steps, and then simulates forward until the total step count difference reaches one. If you change tunable parameters before stepping back, the resulting simulation output might not match the previous simulation output at that step.

Referenced Models

When using Simulation Stepper and the Model block, the referenced model shares the stepping options of the top model throughout a simulation. As a result, changing Simulation Stepper settings for the referenced model during simulation changes the Simulation Stepper settings of the top model. When the simulation ends, the settings of the referenced model revert to the original values; the Stepper settings of the top model stay at the changed settings.

- When the model is not simulating, the top model and referenced model retain their own independent stepping options.
- When the model is simulating and you change a referenced model stepping option, the top model stepping option changes to the same value.
- When the model is simulating and you change a top model stepping option, the referenced model stepping option changes to the same value.
- When the model stops simulating, the referenced model stepping options revert to how they were set before simulation started; the top model keeps the values set during simulation.

Simulation Stepper and Interval Logging

When you change the logging interval of a simulation before rolling back, Simulink does not log data for time steps that were outside the original logging interval until the first forward step after a rollback operation. For more information, see “Logging intervals”.

Simulation Stepper and Stateflow Debugger

When you debug a Stateflow chart (for example, when the simulation stops at a Stateflow breakpoint), Simulation Stepper adds buttons to control the Stateflow debugging session. When the

Stateflow debugging session ends, the Simulation Stepper interface returns to the default. For more information about controlling the Stateflow debugger using the Simulink Toolstrip, see “Control Chart Execution After a Breakpoint” (Stateflow).

See Also

Related Examples

- “Step Through a Simulation” on page 2-12
- “Set Conditional Breakpoints for Stepping a Simulation” on page 2-14

More About

- “How Stepping Through a Simulation Works” on page 2-3
- “Simulation Stepping Options”
- “Simulation Stepper Limitations” on page 2-10

Simulation Stepper Limitations

In this section...

Interface

- There is no command-line interface for Simulation Stepper.

Model Configuration

- Simulation stepping (forward and backward) is available only for Normal and Accelerator modes.
- The step back capability relies on SimState technology for saving and restoring the state of a simulation. As a result, the step back capability is available only for models that support SimState. For more information, see “Save and Restore Simulation Operating Point” on page 25-41.
- Simulation Stepper steps through the major time steps of a simulation without changing the course of a simulation. Choosing a refine factor greater than unity produces loggable outputs at times between the major time steps of the solver. These times are not major time steps, and you cannot step to a model state at those times.
- If you run a simulation with stepping back enabled, the Simulink software checks whether the model can step back. If it cannot, a warning appears at the MATLAB command prompt. For some simulations, Simulink cannot step back. The step back capability is then disabled until the end of that simulation. Then the setting resets to the value you requested.
- When you place custom code in **Configuration Parameters > Simulation Target > Custom Code > Initialize function** in the **Model Configuration Parameters** dialog box, this gets called only during the first simulation in Simulation Stepper.

Blocks

- Some blocks do not support stepping back for reasons other than SimState support. These blocks are:
  - S-functions that have P-work vectors but do not declare their SimState compliance level or declare it to be unknown or disallowed (see “S-Function Compliance with the ModelOperatingPoint”)
  - Simscape™ Multibody™ First Generation blocks
  - Legacy (pre-R2016a) SimEvents® blocks
- MATLAB Function blocks generally support stepping back. However, the use of certain constructs in the MATLAB code of these blocks can prevent the block from supporting stepping back. These scenarios prevent the MATLAB Function blocks from stepping back:
  - Persistent variables of opaque data type. Attempts to step back under this condition cause an error message based on the specific variable type.
  - Extrinsic functions calls that can contain state (such as properties of objects or persistent data of functions). No warnings or error messages appear, but the result likely will be incorrect.

  - Calls to custom C code (through MEX function calls) that do not contain static variables. No warnings or error messages appear, but the result likely will be incorrect.
- Some visualization blocks do not support stepping back. Because these blocks are not critical to the state of the simulation, no errors or warnings appear when you step back in a model that contains these blocks:
  - `XY Graph`
  - `Auto Correlator`
  - `Cross Correlator`
  - `Spectrum Analyzer`
  - `Averaging Spectrum Analyzer`
  - `Power Spectral Density`
  - `Averaging Power Spectral Density`
  - `Floating Bar Plot`
  - `3Dof Animation`
  - `MATLAB Animation`
  - `VR Sink`
  - Any blocks that implement custom visualization in their output method (for example, an S-function that outputs to a MATLAB figure) are not fully supported for stepping back because the block method `Output` does not execute while stepping back. While the state of such blocks remains consistent with the simulation time (if the blocks comply with SimState), the visualization component is inconsistent until the next step forward in the simulation.

  Because these blocks do not affect the numerical result of a simulation, stepping back is not disabled for these blocks. However, the values these blocks output are inaccurate until the simulation steps forward again.

See Also

Related Examples

- “Step Through a Simulation” on page 2-12

More About

- “How Simulation Stepper Helps With Model Analysis” on page 2-2

Step Through a Simulation

Step Forward and Back

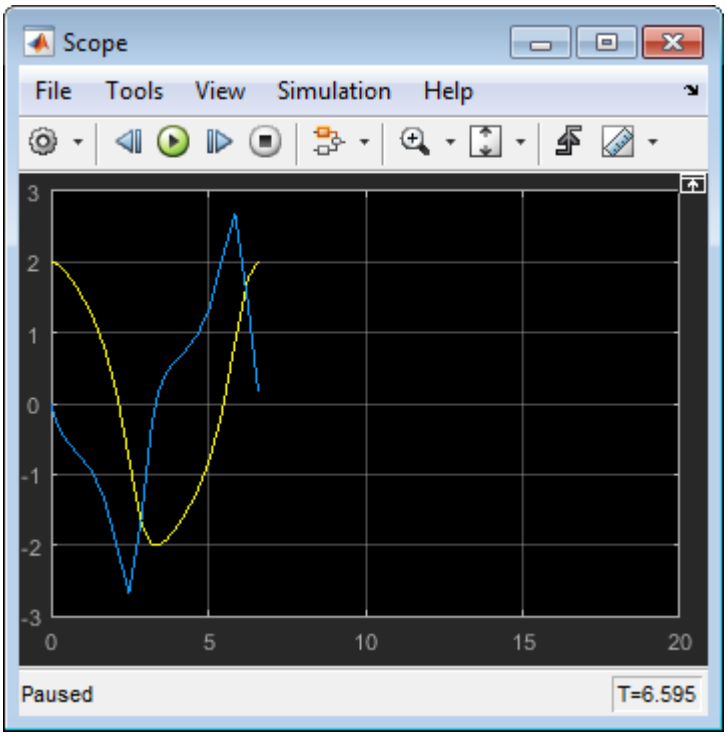
This example shows how to step forward and back through a simulation.

1. At the MATLAB prompt, type

   ```
   vdp
   ```

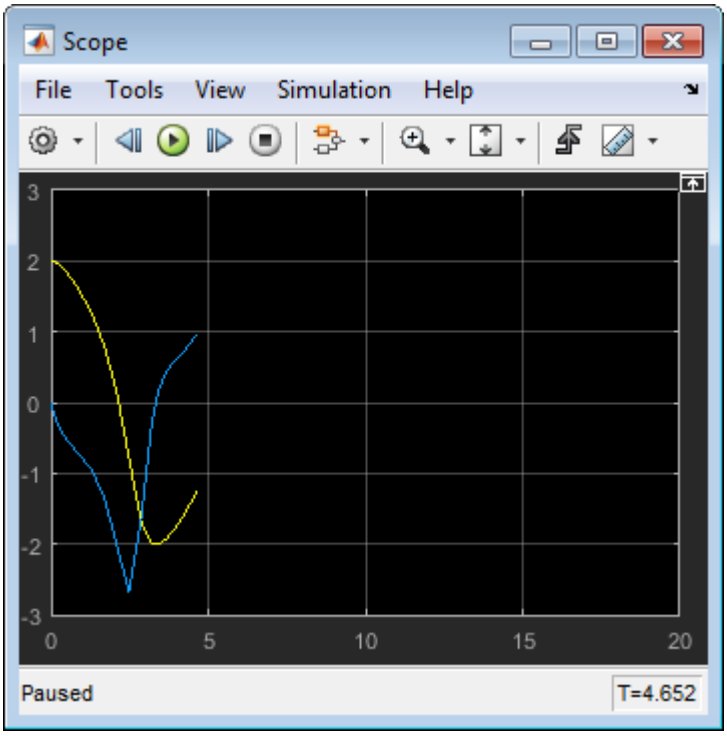
2. In the Simulink Editor for the `vdp` model, click  to open the Simulation Stepping Options dialog box.
3. In the dialog box, select the **Enable stepping back** check box, and then click **OK**.
4. On the **Simulation** tab, click the **Step Forward** button  one time.

   The simulation simulates one step, and the software stores a simulation snapshot for that step.
5. Click the Step Forward button again to step forward again and store simulation data. A total of 25 forward steps produces these simulation results:



6. You must step forward to create the simulation state that the step backward operation requires. This means you must first step forward before you can step backward through the same steps.

   On the **Simulation** tab, click the **Step Back** button  four times to step backward to the simulation snapshot shown below.



See Also

Related Examples

- “Set Conditional Breakpoints for Stepping a Simulation” on page 2-14

More About

- “How Simulation Stepper Helps With Model Analysis” on page 2-2
- “How Stepping Through a Simulation Works” on page 2-3

Set Conditional Breakpoints for Stepping a Simulation

A conditional breakpoint is triggered based on a specified expression evaluated on a signal. When the breakpoint is triggered, the simulation pauses.

Set conditional breakpoints to stop Simulation Stepper when a specified condition is met. One example of a use for conditional breakpoints is when you want to examine results after a certain number of iterations in a loop.

Simulation Stepper allows you to set conditional breakpoints for scalar signals. These breakpoints appear for signals:

Breakpoint	Description
	Enabled breakpoint. Appears when you add the conditional breakpoint.
	Enabled breakpoint hit. Appears when the simulation reaches the condition specified and triggers the breakpoint.
	Disabled breakpoint. Appears when you disable a conditional breakpoint.
	Invalid breakpoint. Appears when the software determines that a breakpoint is invalid for the signal. An enabled breakpoint image changes to this one when, during simulation, the software determines that the conditional breakpoint is invalid.

When setting conditional breakpoints, keep in mind that:

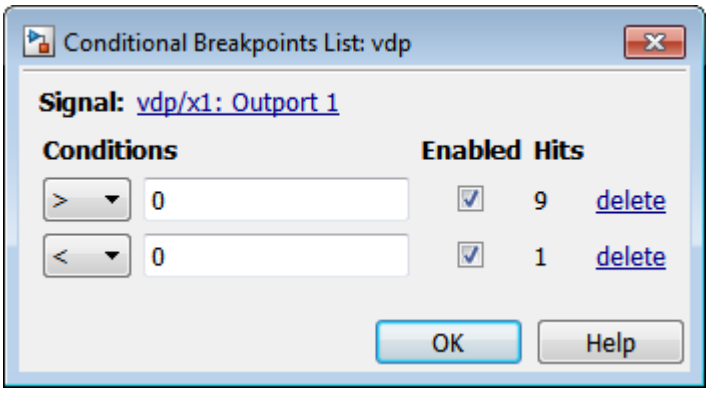
- When simulation arrives at a conditional breakpoint, simulation does not stop when the block is executed. Instead, simulation stops after the current simulation step completes.
- You can add multiple conditional breakpoints to a signal line.

Add and Edit Conditional Breakpoints

1. In a model, right-click a signal and select **Add Conditional Breakpoint**.
2. In the **Add Conditional Breakpoint** dialog box, from the drop-down list, select the condition for the signal. For example, select greater than or less than.
3. Enter the signal value where you want simulation to pause and click **OK**. For the condition values:

   - Use numeric values. Do not use expressions.
   - Do not use `NaN`.

   The affected signal line displays a conditional breakpoint icon:  .
4. Click the breakpoint to view and edit all conditions set for the signal.



5 Simulate the model and notice that the model pauses as simulation steps through the conditional breakpoints.

Conditional Breakpoints Limitations

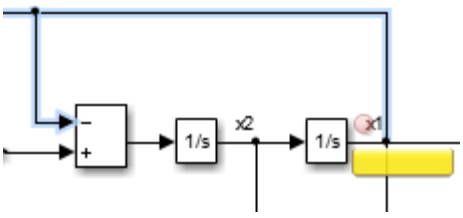
- You can set conditional breakpoints only on real scalar signals of these data types:
  - double
  - single
  - int
  - bool
  - fixed point (based on the converted double value)
- You cannot set conditional breakpoints (or port value display labels) on non-Simulink signals, such as Simscape or SimEvents signals.
- Conditional breakpoints also have the limitations that port value display have (“Port Value Display Limitations” on page 36-23).
- Conditional Breakpoints only work on the first iteration of For Each Subsystems

Observe Conditional Breakpoint Values

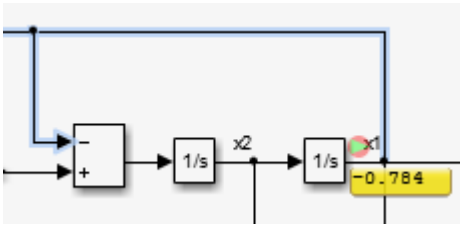
To observe a conditional breakpoint value of a block signal, use data tips to display block port values. You can add data tips before or after you add conditional breakpoints.

1 Enable the value display for a signal. Right-click the signal line that has a conditional breakpoint and select **Show Value Label of Selected Port**.

The data tip for the value display appears.



2 Simulate the model and observe the conditional breakpoint and data tip when the simulation triggers the breakpoint.



See Also

Related Examples

- “Step Through a Simulation” on page 2-12

More About

- “How Stepping Through a Simulation Works” on page 2-3

Simulation Pacing

Simulation pacing enables you to slow down a simulation to understand and observe the system's behavior. Visualizing simulations at a slower rate makes it easier to understand underlying system design, identify design issues and demonstrate near real-time behavior. You can view the results and inspect your system while the simulation is in progress. Slow down animations in scopes, to observe how and when the system changes state.

Simulation pacing is useful in scenarios where one simulation-second is completed in a few wall clock time milliseconds. You can also manually interact with the model while it is running at a slower rate, which can help you to explore how your changes affect the simulation behavior. Note that the simulation will run at an approximation of the specified pace.

To start, specify the rate of simulation. This rate is a ratio of elapsed simulation time to elapsed wall clock time.

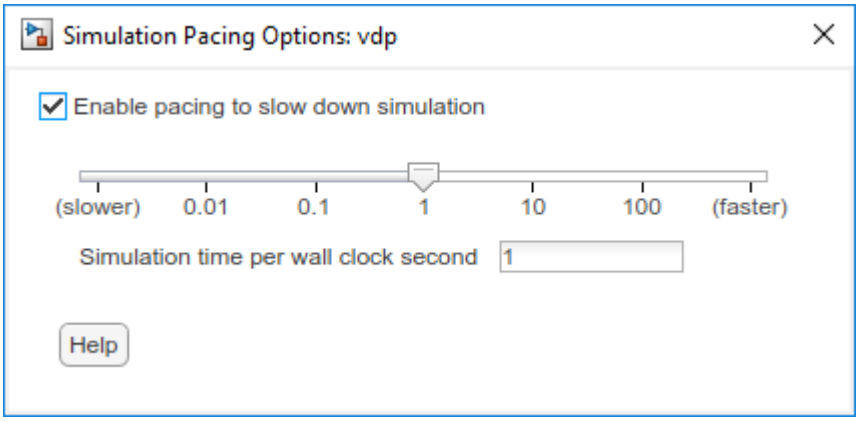
Use Simulation Pacing

This example shows how to use simulation pacing during a simulation.

1. Open the `vdp` model.

   ```
   vdp
   ```

2. In the **Simulation** tab, select **Run** > **Simulation Pacing**



3. In the Simulation Pacing Options dialog, select **Enable pacing to slow down simulation**. On enabling, the specified pace gets automatically applied to the simulation.
4. Select the pace at which the model should run by using the slider or entering the pace in the **Simulation time per wall clock second** field. This field also accepts values outside of the range on the slider. The value entered in the field can only be a finite positive number.
5. On simulating the model, you can watch the simulation progress at the specified pace in the scope. You can also change the pace through the dialog box and enable/disable pacing while your simulation is running.

To enable Pacing from the command line, use `set_param(model, 'EnablePacing', 'on')` command. To enter the value use `set_param(model, 'PacingRate', value)` where the value is a finite positive number.

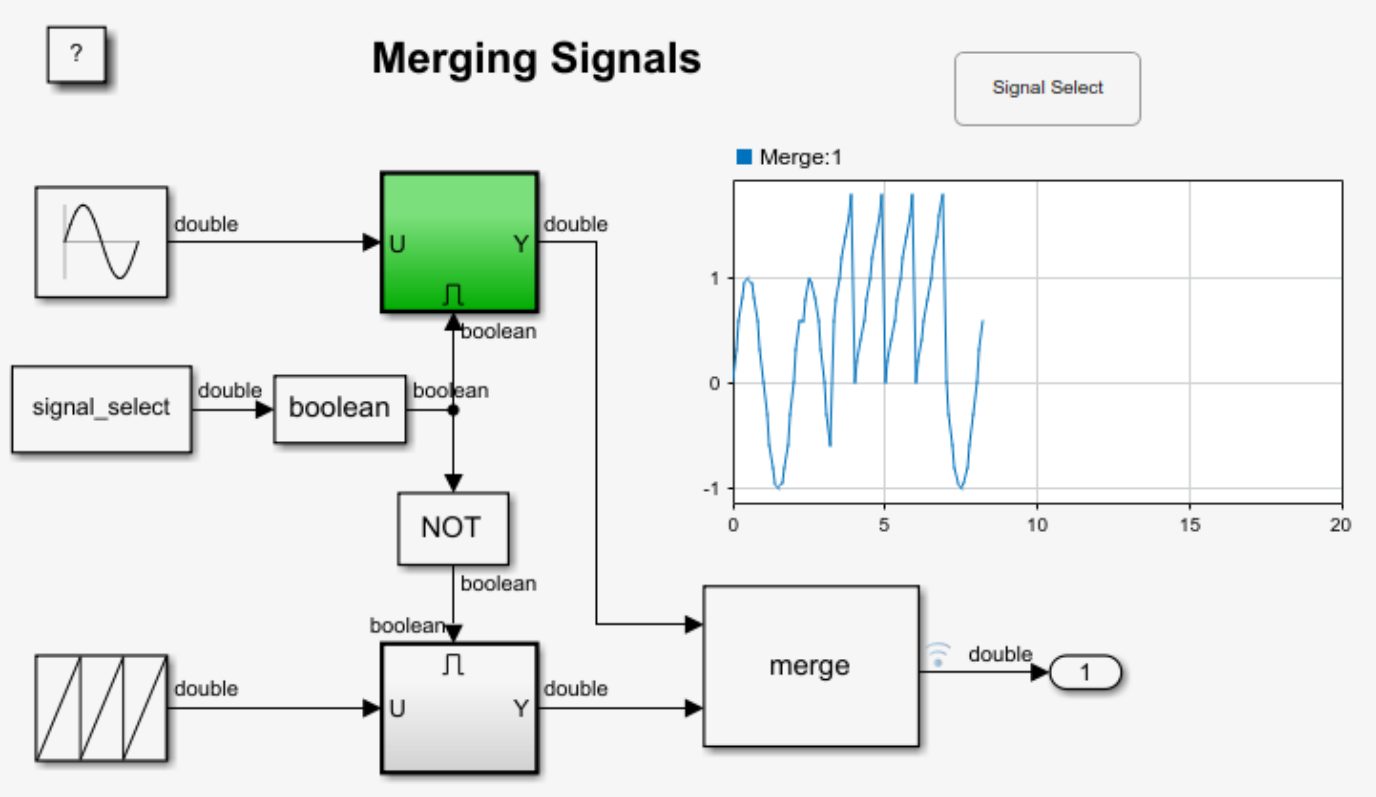
The `run` button changes to  , when a simulation is run with Pacing enabled. The status bar indicates if a simulation is running with pacing enabled. It also indicates if the simulation can not run at the specified pace.

Use Simulation Pacing with Dashboard Blocks

You can use Dashboard blocks to view signals and tune variables and parameters in your model while slowing a simulation using simulation pacing. Using Dashboard blocks with simulation pacing allows you to build an intuitive understanding of your model and how it responds to changes as you tune parameters in your model.

The example model uses simulation pacing to slow the simulation pace to approximately equivalent to wall clock time, or one simulation second per one wall clock second. You can push the Signal Select button to select whether the sine wave or sawtooth input signal passes to the output, which is visualized using a Dashboard Scope block.

Open the model and run a simulation. During the simulation, press and release the Signal Select button and observe the effect on the output signal.



For more information about creating the example model, see “Control Merging Signals with the Push Button Block”.

Limitations

- Simulation pacing is not compatible with:
  - Simulation stepper
  - Profiler
- Simulation pacing is not supported in rapid accelerator mode

See Also

Simulation Pace

More About

- “How Stepping Through a Simulation Works” on page 2-3
- “How Profiler Captures Performance Data” on page 31-5
- “Tune and Visualize Your Model with Dashboard Blocks” on page 29-164

How Simulink Works

Simulation Phases in Dynamic Systems

In this section...

Model Compilation

The first phase of simulation occurs when the system’s model is open and you simulate the model. In the Simulink Editor, click **Run**. Running the simulation causes the Simulink engine to invoke the model compiler. The model compiler converts the model to an executable form, a process called compilation. In particular, the compiler:

- Evaluates the model's block parameter expressions to determine their values.
- Determines signal attributes, e.g., name, data type, numeric type, and dimensionality, not explicitly specified by the model and checks that each block can accept the signals connected to its inputs.
- Propagates the attributes of a source signal to the inputs of the blocks that it drives in order to compute previously unspecified attributes in the blocks.
- Performs block reduction optimizations.
- Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain (see “Compare Solvers” on page 3-6).
- Determines the block execution order by task-based sorting.
- Determines the sample times of all blocks in the model whose sample times you did not explicitly specify (see “How Propagation Affects Inherited Sample Times” on page 7-30).

These events are essentially the same as what occurs when you update a diagram (“Update Diagram and Run Simulation” on page 1-38). The difference is that the Simulink software starts model compilation as part of model simulation, where compilation leads directly into the linking phase, as described in “Link Phase” on page 3-2. In contrast, you start an explicit model update as a standalone operation on a model.

Link Phase

In this phase, the Simulink engine allocates memory needed for working areas (signals, states, and run-time parameters) for execution of the block diagram. It also allocates and initializes memory for data structures that store run-time information for each block. For built-in blocks, the principal run-time data structure for a block is called the SimBlock. It stores pointers to a block's input and output buffers and state and work vectors.

Method Execution Lists

In the Link phase, the Simulink engine also creates method execution lists. These lists list the most efficient order in which to invoke a model's block methods to compute its outputs. The block execution order lists generated during the model compilation phase are used to construct the method execution lists.

Block Priorities

You can assign update priorities to blocks. The output methods of higher priority blocks are executed before those of lower priority blocks. These priorities are honored only if they are consistent with its block execution order.

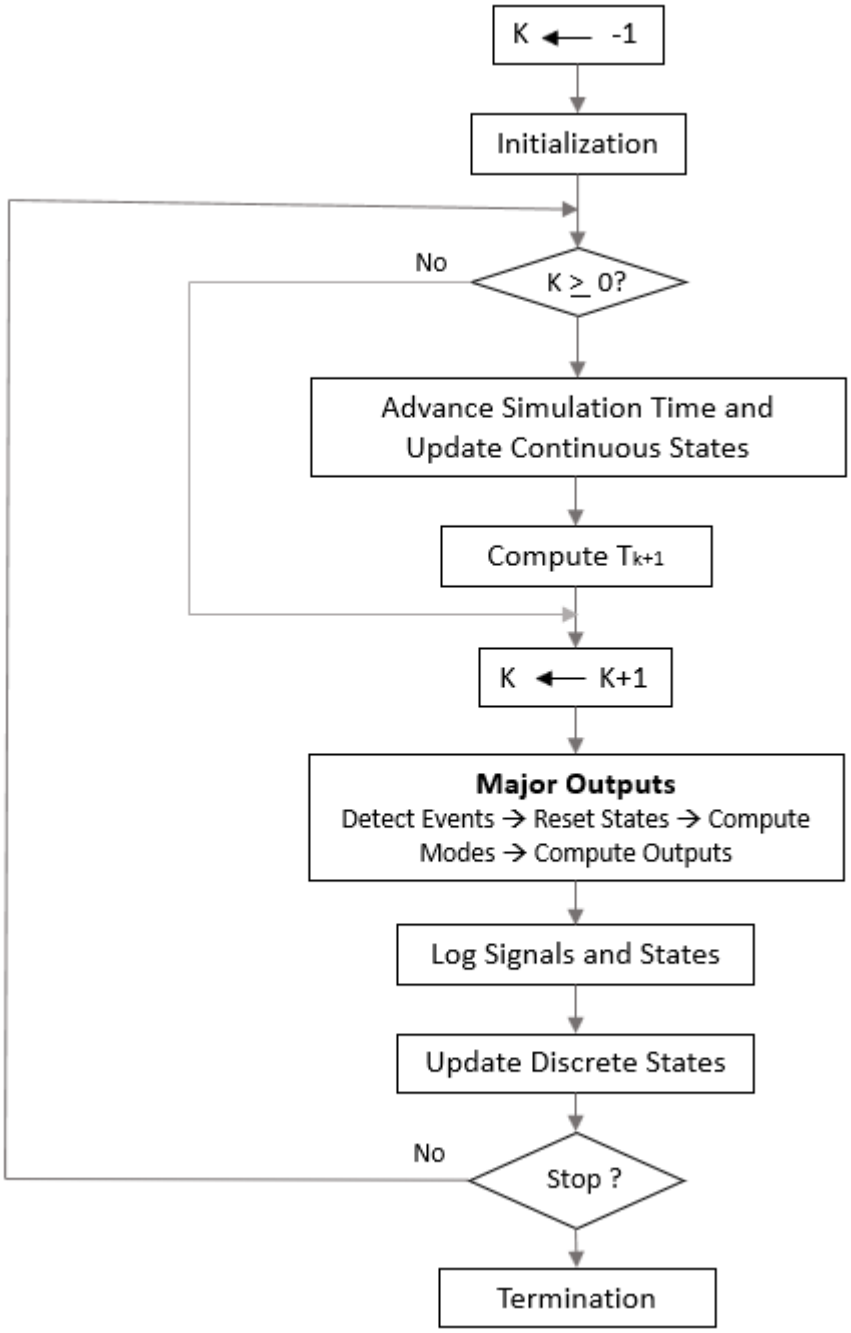
Simulation Loop Phase

Once the Link Phase completes, the simulation enters the simulation loop phase. In this phase, the Simulink engine successively computes the states and outputs of the system at intervals from the simulation start time to the finish time, using information provided by the model. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called the step size. The step size depends on the type of solver (see “Compare Solvers” on page 3-6) used to compute the system's continuous states, the system's fundamental sample time (see “Sample Times in Systems” on page 7-23), and whether the system's continuous states have discontinuities (see “Zero-Crossing Detection” on page 3-10).

The Simulation Loop phase has two subphases: the Loop Initialization phase and the Loop Iteration phase. The initialization phase occurs once, at the start of the loop. The iteration phase is repeated once per time step from the simulation start time to the simulation stop time.

At the start of the simulation, the model specifies the initial states and outputs of the system to be simulated. At each step, new values for the system's inputs, states, and outputs are computed, and the model is updated to reflect the computed values. At the end of the simulation, the model reflects the final values of the system's inputs, states, and outputs. The Simulink software provides data display and logging blocks. You can display and/or log intermediate results by including these blocks in your model.

The following flowchart explains how the simulation loop works where k denotes the major step



counter:

Loop Iteration

At each time step, the Simulink engine:

1. Computes the model outputs.

   The Simulink engine initiates this step by invoking the Simulink model Outputs method. The model Outputs method in turn invokes the model system Outputs method, which invokes the

Outputs methods of the blocks that the model contains in the order specified by the Outputs method execution lists generated in the Link phase of the simulation (see “Compare Solvers” on page 3-6).

The system Outputs method passes the following arguments to each block Outputs method: a pointer to the block's data structure and to its SimBlock structure. The SimBlock data structures point to information that the Outputs method needs to compute the block's outputs, including the location of its input buffers and its output buffers.

2. Computes the model's states.

   The Simulink engine computes a model's states by invoking a solver. Which solver it invokes depends on whether the model has no states, only discrete states, only continuous states, or both continuous and discrete states.

   If the model has only discrete states, the Simulink engine invokes the discrete solver selected by the user. The solver computes the size of the time step needed to hit the model's sample times. It then invokes the Update method of the model. The model Update method invokes the Update method of its system, which invokes the Update methods of each of the blocks that the system contains in the order specified by the Update method lists generated in the Link phase.

   If the model has only continuous states, the Simulink engine invokes the continuous solver specified by the model. Depending on the solver, the solver either in turn calls the Derivatives method of the model once or enters a subcycle of minor time steps where the solver repeatedly calls the model's Outputs methods and Derivatives methods to compute the model's outputs and derivatives at successive intervals within the major time step. This is done to increase the accuracy of the state computation. The model Outputs method and Derivatives methods in turn invoke their corresponding system methods, which invoke the block Outputs and Derivatives in the order specified by the Outputs and Derivatives methods execution lists generated in the Link phase.

3. Optionally checks for discontinuities in the continuous states of blocks.

   A technique called zero-crossing detection is used to detect discontinuities in continuous states. See “Zero-Crossing Detection” on page 3-10 for more information.

4. Computes the time for the next time step.

Steps 1 through 4 are repeated until the simulation stop time is reached.

See Also

More About

- “Compare Solvers” on page 3-6
- “Simulate a Model Interactively” on page 25-2

Compare Solvers

A dynamic system is simulated by computing its states at successive time steps over a specified time span, using information provided by the model. The process of computing the successive states of a system from its model is known as solving the model. No single method of solving a model suffices for all systems. Accordingly, Simulink provides a set of programs, known as *solvers*, each of which embodies a particular approach to solving a model. The Configuration Parameters dialog box allows you to choose the solver best suited to your model.

Fixed-Step Versus Variable-Step Solvers

You can choose the solvers provided by Simulink based on the way they calculate step size: fixed-step and variable-step.

Fixed-step solvers solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as the step size. You can specify the step size or let the solver choose the step size. Generally, decreasing the step size increases the accuracy of the results while increasing the time required to simulate the system.

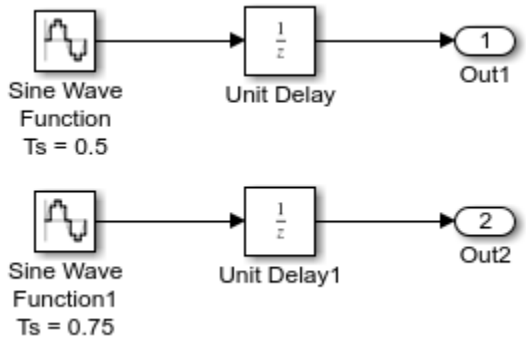
Variable-step solvers vary the step size during the simulation. They reduce the step size to increase accuracy when a model's states are changing rapidly and increase the step size to avoid taking unnecessary steps when the model's states are changing slowly. Computing the step size adds to the computational overhead at each step but can reduce the total number of steps, and hence the simulation time required to maintain a specified level of accuracy for models with rapidly changing or piecewise continuous states.

Fixed-step and variable-step solvers compute the next simulation time as the sum of the current simulation time and the step size. The **Type** control on the **Solver** configuration pane allows you to select the type of solver. With a fixed-step solver, the step size remains constant throughout the simulation. With a variable-step solver, the step size can vary from step to step, depending on the model dynamics. In particular, a variable-step solver increases or reduces the step size to meet the error tolerances that you specify.

The choice between these types depends on how you plan to deploy your model and the model dynamics. If you plan to generate code from your model and run the code on a real-time computer system, choose a fixed-step solver to simulate the model. You cannot map the variable-step size to the real-time clock.

If you do not plan to deploy your model as generated code, the choice between a variable-step and a fixed-step solver depends on the dynamics of your model. A variable-step solver might shorten the simulation time of your model significantly. A variable-step solver allows this saving because, for a given level of accuracy, the solver can dynamically adjust the step size as necessary. This approach reduces the number of steps required. The fixed-step solver must use a single step size throughout the simulation, based on the accuracy requirements. To satisfy these requirements throughout the simulation, the fixed-step solver typically requires a small step.

The `ex_multirate` example model shows how a variable-step solver can shorten simulation time for a multirate discrete model.



The model generates outputs at two different rates: every 0.5 s and every 0.75 s. To capture both outputs, the fixed-step solver must take a time step every 0.25 s (the *fundamental sample time* for the model).

```
[0.0 0.25 0.5 0.75 1.0 1.25 1.5 ...]
```

By contrast, the variable-step solver has to take a step only when the model generates an output.

```
[0.0 0.5 0.75 1.0 1.5 ...]
```

This scheme significantly reduces the number of time steps required to simulate the model.

Continuous Versus Discrete Solvers

Simulink provides both continuous and discrete solvers.

When you select a solver type, you can also select a specific solver. Both sets of solvers include discrete and continuous solvers. Discrete and continuous solvers rely on the model blocks to compute the values of any discrete states. Blocks that define discrete states are responsible for computing the values of those states at each time step. However, unlike discrete solvers, continuous solvers use numerical integration to compute the continuous states that the blocks define. When choosing a solver, determine first whether to use a discrete solver or a continuous solver.

If your model has no continuous states, then Simulink switches to either the fixed-step discrete solver or the variable-step discrete solver. If your model has only continuous states or a mix of continuous and discrete states, choose a continuous solver from the remaining solver choices based on the dynamics of your model. Otherwise, an error occurs.

Continuous solvers use numerical integration to compute a model's continuous states at the current time step based on the states at previous time steps and the state derivatives. Continuous solvers rely on the individual blocks to compute the values of the model's discrete states at each time step.

Discrete solvers exist primarily to solve purely discrete models. They compute the next simulation time step for a model and nothing else. In performing these computations, they rely on each block in the model to update its individual discrete states. They do not compute continuous states.

The solver library contains two discrete solvers: a fixed-step discrete solver and a variable-step discrete solver. The fixed-step solver by default chooses the step size and simulation rate fast enough to track state changes in the fastest block in your model. The variable-step solver adjusts the simulation step size to keep pace with the actual rate of discrete state changes in your model. This

adjustment can avoid unnecessary steps and shorten simulation time for multirate models. See “Sample Times in Systems” on page 7-23 for more information.

Note You must use a continuous solver to solve a model that contains both continuous and discrete states because discrete solvers cannot handle continuous states. If, on the other hand, you select a continuous solver for a model with no states or discrete states only, Simulink software uses a discrete solver.

Explicit Versus Implicit Continuous Solvers

You represent an explicit system by an equation

$$\dot{x} = f(x)$$

For any given value of x, you can compute $\dot{x}$ by substituting x in $f(x)$ and evaluating the equation.

Equations of the form

$$F(\dot{x}, x) = 0$$

are considered to be implicit. For any given value of x, you must solve this equation to calculate $\dot{x}$.

A linearly implicit system can be represented by the equation

$$M(x) \cdot \dot{x} = f(x)$$

$M(x)$ is called the mass matrix and $f(x)$ is the forcing function. A system becomes linearly implicit when you use physical modeling blocks in the model.

While you can apply an implicit or explicit continuous solver to solve all these systems, implicit solvers are designed specifically for solving stiff problems. Explicit solvers solve nonstiff problems. An ordinary differential equation problem is said to be stiff if the desired solution varies slowly, but there are closer solutions that vary rapidly. The numerical method must then take small time steps to solve the system. Stiffness is an efficiency issue. The more stiff a system, the longer it takes to for the explicit solver to perform a computation. A stiff system has both slowly and quickly varying continuous dynamics.

When compared to explicit solvers, implicit solvers provide greater stability for oscillatory behavior. However, implicit solvers are also computationally more expensive. They generate the Jacobian matrix and solve the set of algebraic equations at every time step using a Newton-like method. To reduce this extra cost, the implicit solvers offer a **Solver Jacobian method** parameter that allows you to improve the simulation performance of implicit solvers. See “Choose a Jacobian Method for an Implicit Solver” on page 25-9 for more information. Implicit solvers are more efficient than explicit solvers for solving a linearly implicit system.

One-Step Versus Multistep Continuous Solvers

The Simulink solver library provides both one-step and multistep solvers. The one-step solvers estimate $y(t_n)$ using the solution at the immediately preceding time point, $y(t_{n-1})$, and the values of the derivative at multiple points between t_n and t_{n-1}. These points are minor steps.

Multistep solvers use the results at several preceding time steps to compute the current solution. Simulink provides one explicit multistep solver, `ode113`, and one implicit multistep solver, `ode15s`. Both are variable-step solvers.

Single-Order Versus Variable-Order Continuous Solvers

This distinction is based on the number of orders that the solver uses to solve the system of equation. Two variable-order solvers, `ode15s` and `ode113`, are part of the solver library. They use multiple orders to solve the system of equations. Specifically, the implicit, variable-step `ode15s` solver uses first-order through fifth-order equations, while the explicit, variable-step `ode113` solver uses first-order through thirteenth-order equations. For `ode15s`, you can limit the highest order applied via the **Maximum Order** parameter. For more information, see “Maximum Order” on page 25-18.

See Also

“Zero-Crossing Detection” on page 3-10 | “Simulink Models” | “Simulation Phases in Dynamic Systems” on page 3-2

Zero-Crossing Detection

A variable-step solver dynamically adjusts the time step size, causing it to increase when a variable is changing slowly and to decrease when the variable changes rapidly. This behavior causes the solver to take many small steps in the vicinity of a discontinuity because the variable is rapidly changing in this region. This improves accuracy but can lead to excessive simulation times.

Simulink uses a technique known as *zero-crossing detection* to accurately locate a discontinuity without resorting to excessively small time steps. Usually this technique improves simulation run time, but it can cause some simulations to halt before the intended completion time.

Simulink uses two algorithms for this purpose: nonadaptive and adaptive. For information about these techniques, see “Zero-Crossing Algorithms” on page 3-25.

Demonstrating Effects of Excessive Zero-Crossing Detection

This example provides three models that illustrate zero-crossing behavior: `example_bounce_two_integrators`, `example_doublebounce`, and `example_bounce`.

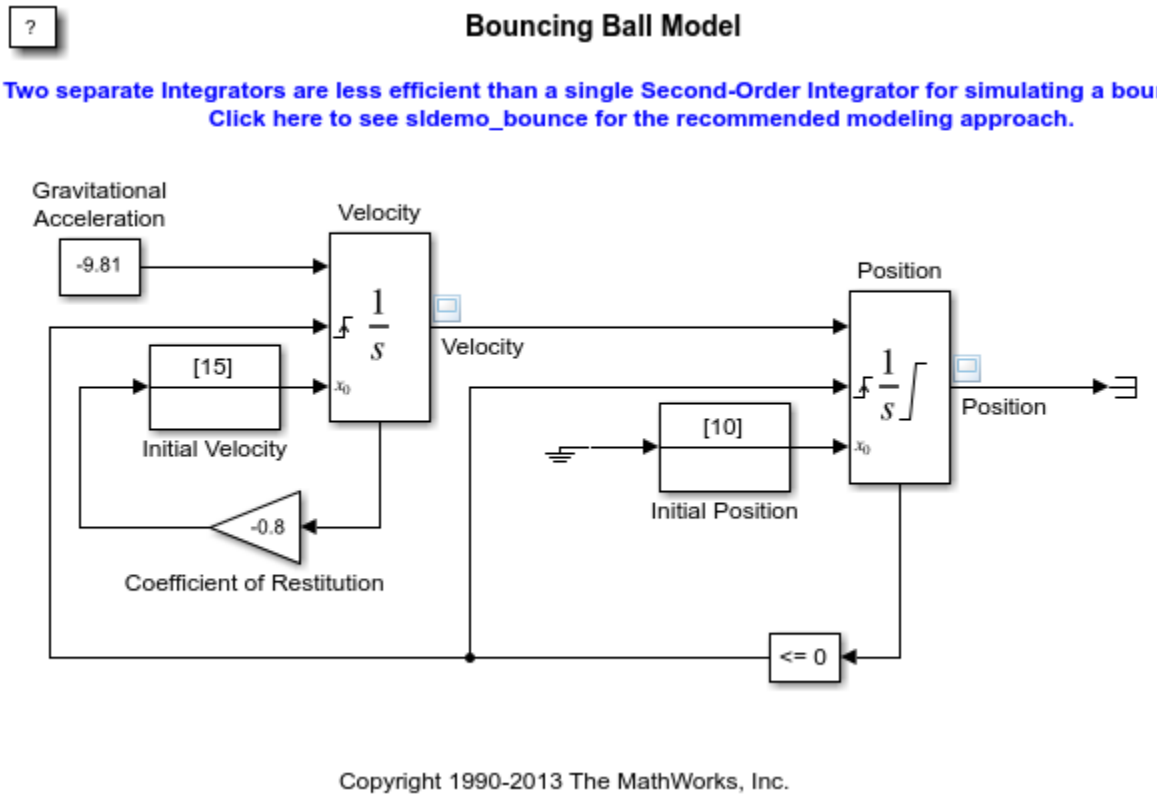
The `example_bounce_two_integrators` model demonstrates how excessive zero crossings can cause a simulation to halt before the intended completion time unless you use the adaptive algorithm.

The `example_bounce` model uses a better model design than `example_bounce_two_integrators`.

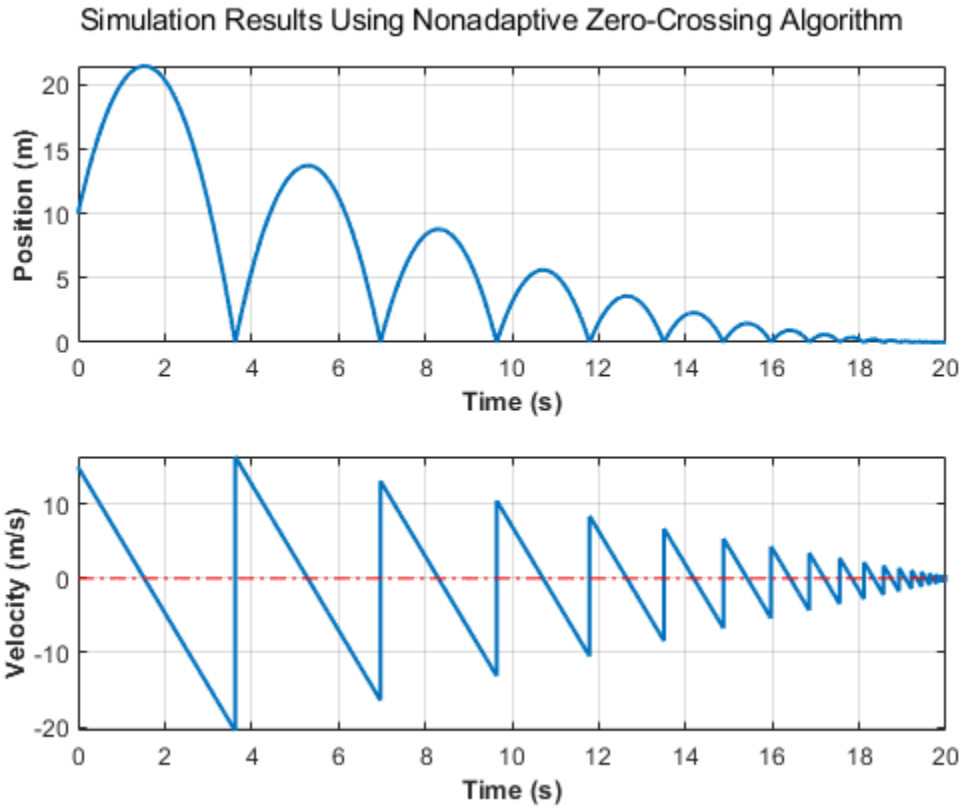
The `example_doublebounce` model demonstrates how the adaptive algorithm successfully solves a complex system with two distinct zero-crossing requirements.

Consider the `example_bounce_two_integrators` model. It uses two single integrators to compute the vertical velocity and position of the ball over the time of the simulation.

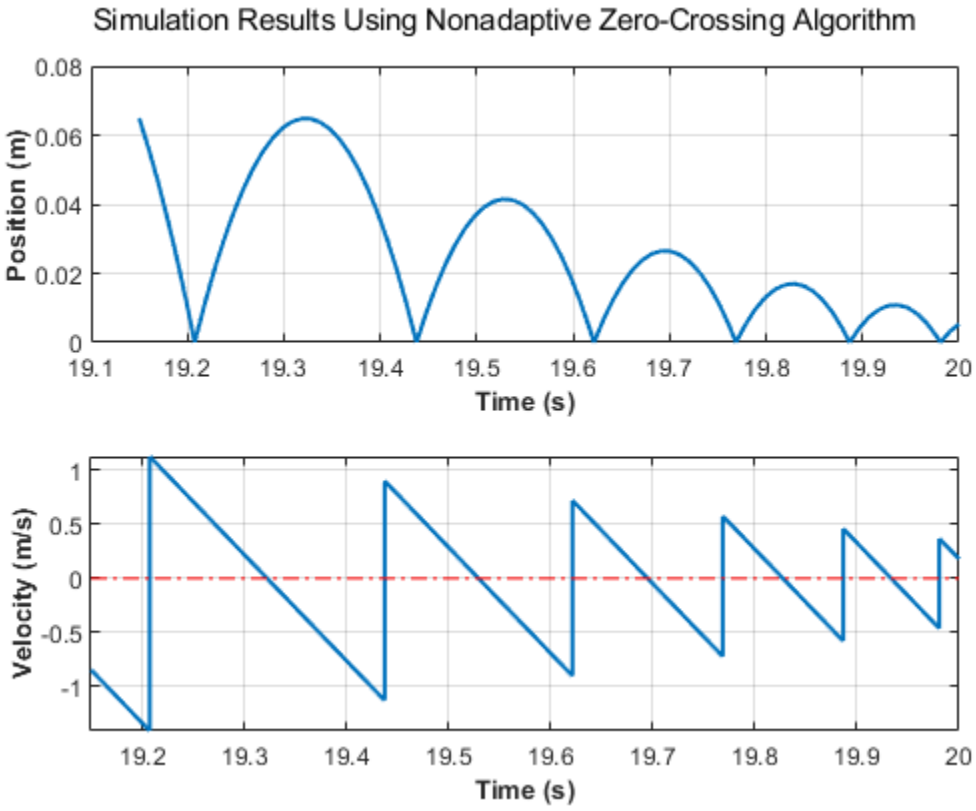
1. Open the model by running `open_system('example_bounce_two_integrators')` at the command line.
2. Once the block diagram appears, set the **Solver details > Zero-crossing options > Algorithm** parameter in the **Solver** pane of the Model configuration parameters to `Nonadaptive`. Set the stop time of the model to 20 s. You can change this setting in the Simulink toolstrip or the **Solver** pane of the model configuration parameters.
3. Simulate the model.



You can now view and analyze the simulation results.



Upon closer examination of the last portion of the simulation, you will see that velocity hovers just above zero.

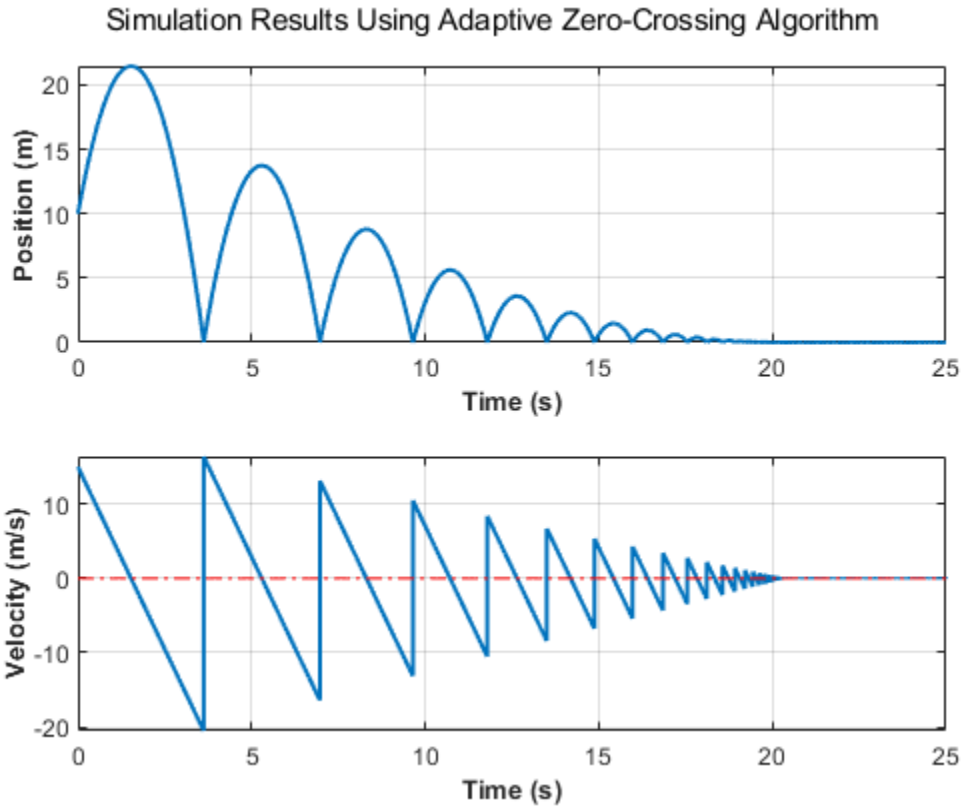


Change the simulation **Stop time** to 25 s and simulate the model. The simulation stops with an error due to excessive consecutive zero-crossing events at the Compare To Zero and Position blocks.

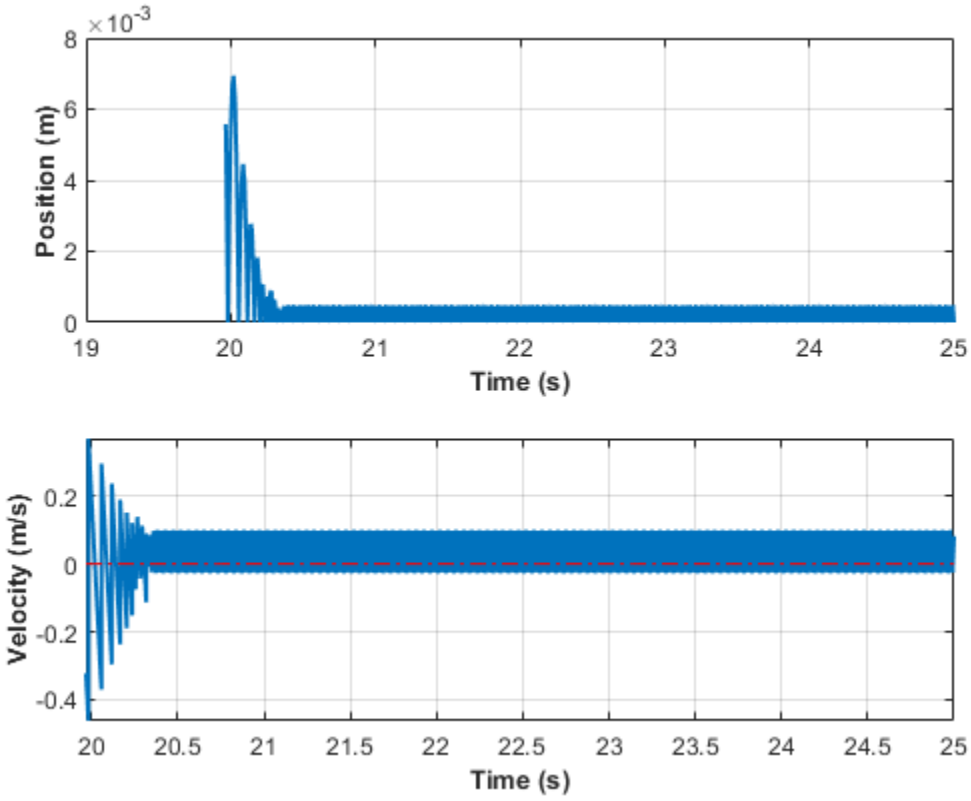
```
Simulink will stop the simulation of model 'example_bounce_two_integrators' because the 2 zero cr
 ---------------------------------------------------------------------------------
Number of consecutive zero-crossings : 1000
          Zero-crossing signal name : RelopInput
                         Block type : RelationalOperator
                         Block path : 'example_bounce_two_integrators/Compare To Zero/Compare'
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Number of consecutive zero-crossings : 500
          Zero-crossing signal name : IntgLoLimit
                         Block type : Integrator
                         Block path : 'example_bounce_two_integrators/Position'
---------------------------------------------------------------------------------
```

Although you can increase this limit by adjusting the **Model Configuration Parameters > Solver > Number of consecutive zero crossings** parameter, making that change still does not allow the simulation to go on for 25 s.

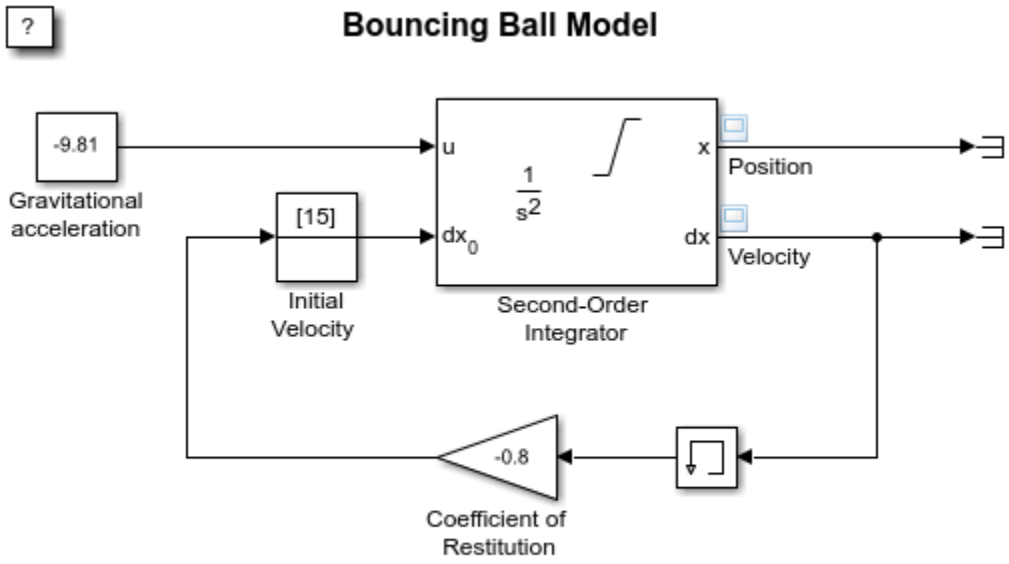
Change the **Solver details > Zero-crossing options > Algorithm** parameter in the **Solver** pane of the Model configuration parameters to `Adaptive` and simulate the model again for 25 s.



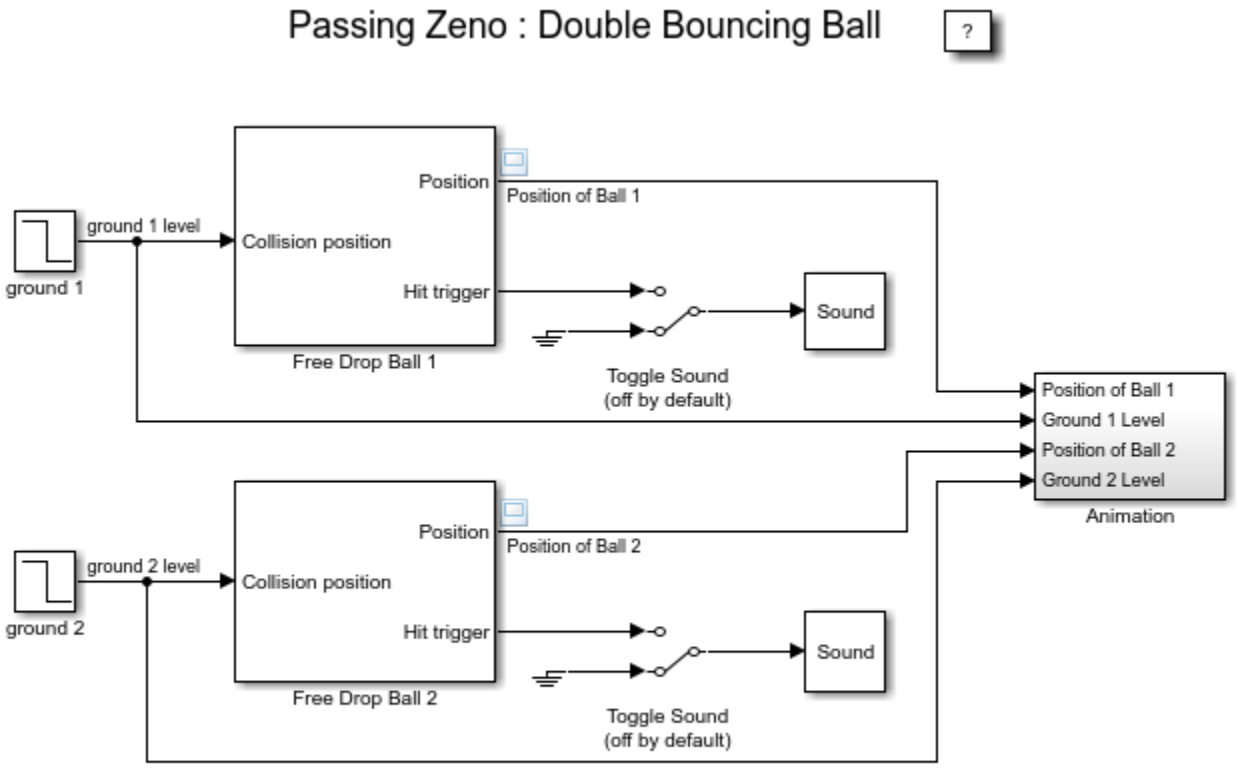
Zooming in on the last 5 s of the simulation, you can see that the results are more complete and closer to the expected analytical solution of the dynamics of a bouncing ball. The amount of chatter you see is a consequence of the system's states approaching zero and is expected in numerical simulations.

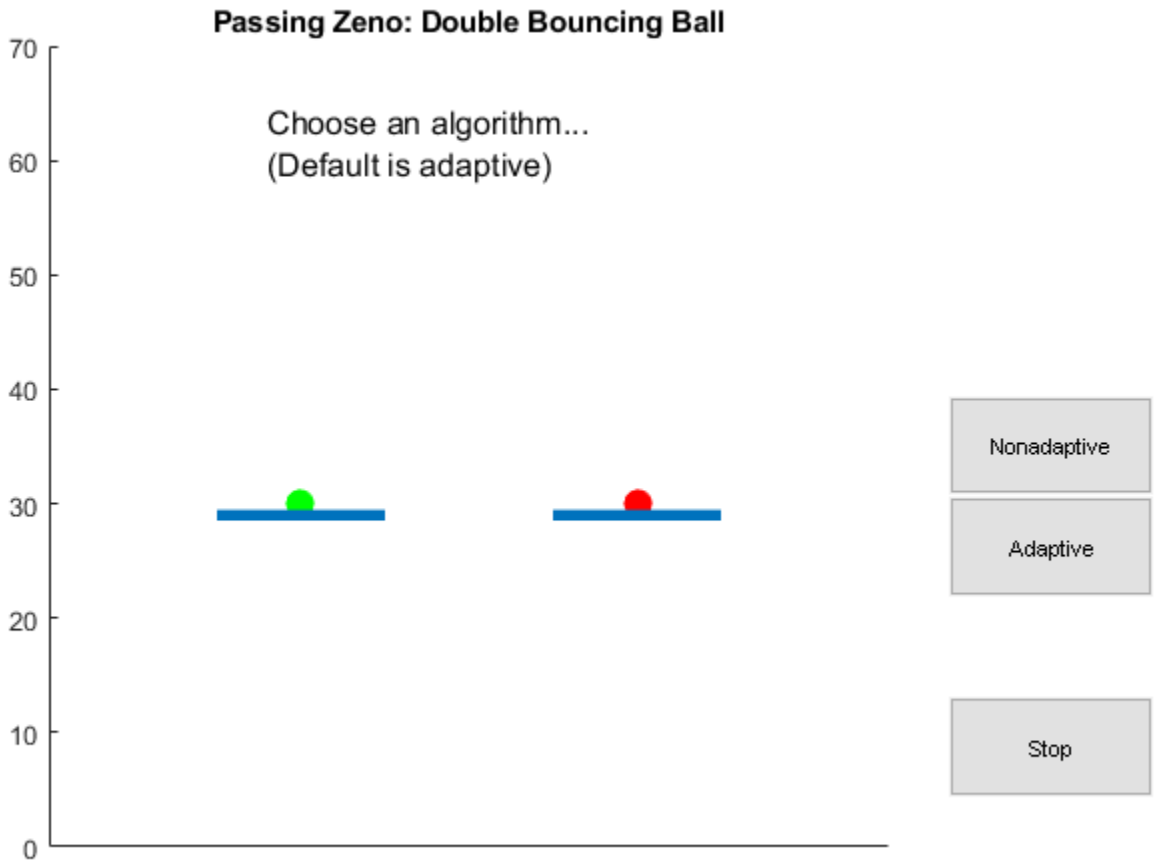


The `example_bounce` model uses a Second-Order Integrator block to model the dynamics of the bouncing ball. This is the preferred method to model the double integration of the ball's dynamics for solver performance. To compare the solver performance for `example_bounce_two_integrators` and `example_bounce`, try running the Solver Profiler on both models. For a detailed comparison of both mo0dels, see “Simulation of a Bouncing Ball”.



For a side-by-side comparison of adaptive and nonadaptive zero-crossing detection algorithms, see “Double Bouncing Ball: Use of Adaptive Zero-Crossing Location”.





Preventing Excessive Zero Crossings

Use the following table to prevent excessive zero-crossing errors in your model.

Change Type	Change Procedure	Benefits
Increase the number of allowed zero crossings	Increase the value of the **Number of consecutive zero crossings**. option on the **Solver** pane in the Configuration Parameters dialog box.	This may give your model enough time to resolve the zero crossing.
Relax the **Signal threshold**	Select **Adaptive** from the **Algorithm** pull down and increase the value of the **Signal threshold** option on the **Solver** pane in the Configuration Parameters dialog box.	The solver requires less time to precisely locate the zero crossing. This can reduce simulation time and eliminate an excessive number of consecutive zero-crossing errors. However, relaxing the **Signal threshold** may reduce accuracy.

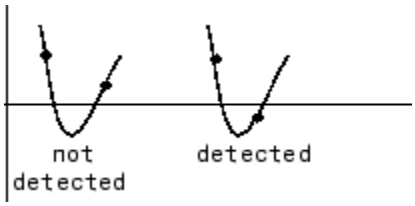
Change Type	Change Procedure	Benefits
Use the **Adaptive** algorithm	Select **Adaptive** from the **Algorithm** drop-down on the **Solver** pane in the Configuration Parameters dialog box.	This algorithm dynamically adjusts the zero-crossing threshold, which improves accuracy and reduces the number of consecutive zero crossings detected. With this algorithm you have the option of specifying both the **Time tolerance** and the **Signal threshold**.
Disable zero-crossing detection for a specific block	1 Clear the **Enable zero-crossing detection** check box on the block's parameter dialog box. 2 Select `Use local settings` from the **Zero-crossing control** pull down on the **Solver** pane of the Configuration Parameters dialog box.	Locally disabling zero-crossing detection prevents a specific block from stopping the simulation because of excessive consecutive zero crossings. All other blocks continue to benefit from the increased accuracy that zero-crossing detection provides.
Disable zero-crossing detection for the entire model	Select `Disable all` from the **Zero-crossing control** pull down on the **Solver** pane of the Configuration Parameters dialog box.	This prevents zero crossings from being detected anywhere in your model. A consequence is that your model no longer benefits from the increased accuracy that zero-crossing detection provides.
If using the `ode15s` solver, consider adjusting the order of the numerical differentiation formulas	Select a value from the `Maximum order` pull down on the **Solver** pane of the Configuration Parameters dialog box.	For more information, see “Maximum order”.
Reduce the maximum step size	Enter a value for the `Max step size` option on the **Solver** pane of the Configuration Parameters dialog box.	The solver takes steps small enough to resolve the zero crossing. However, reducing the step size can increase simulation time, and is seldom necessary when using the adaptive algorithm.

How the Simulator Can Miss Zero-Crossing Events

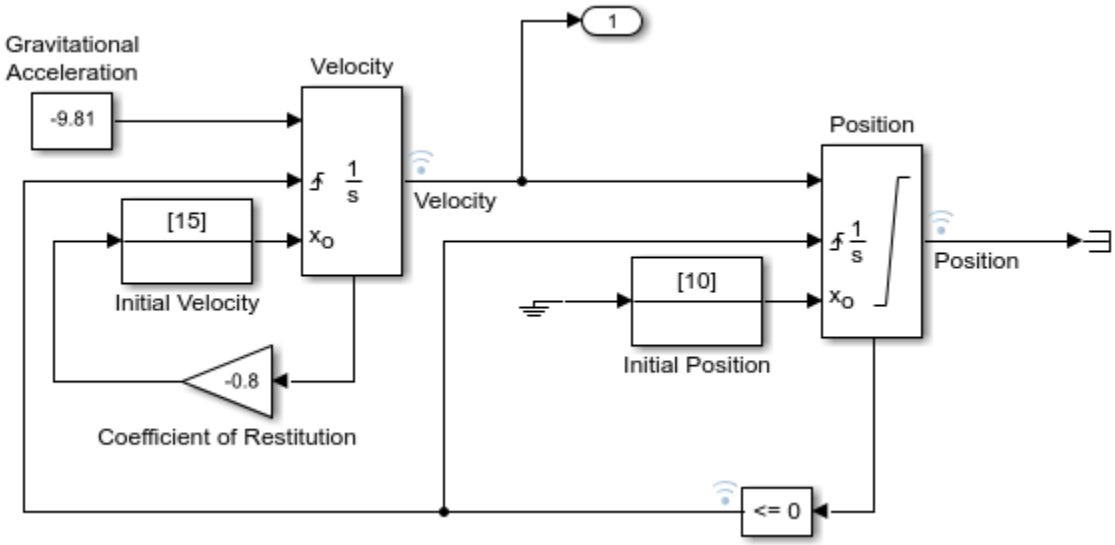
The bounce and double-bounce models, in “Simulation of a Bouncing Ball” and “Double Bouncing Ball: Use of Adaptive Zero-Crossing Location” show that high-frequency fluctuations about a discontinuity (chattering) can cause a simulation to prematurely halt.

It is also possible for the solver to entirely miss zero crossings if the solver error tolerances are too large. This is possible because the zero-crossing detection technique checks to see if the value of a signal has changed sign after a major time step. A sign change indicates that a zero crossing has occurred, and the zero-crossing algorithm searches for the precise crossing time. However, if a zero crossing occurs within a time step, but the values at the beginning and end of the step do not indicate a sign change, the solver steps over the crossing without detecting it.

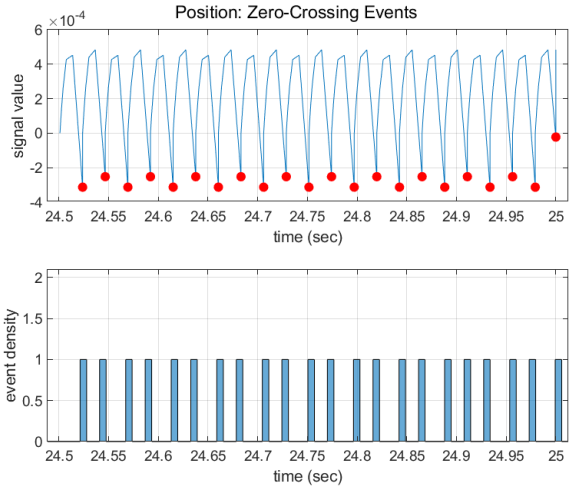
The following figure shows a signal that crosses zero. In the first instance, the integrator steps over the event because the sign has not changed between time steps. In the second, the solver detects sign change and therefore detects the zero-crossing event.

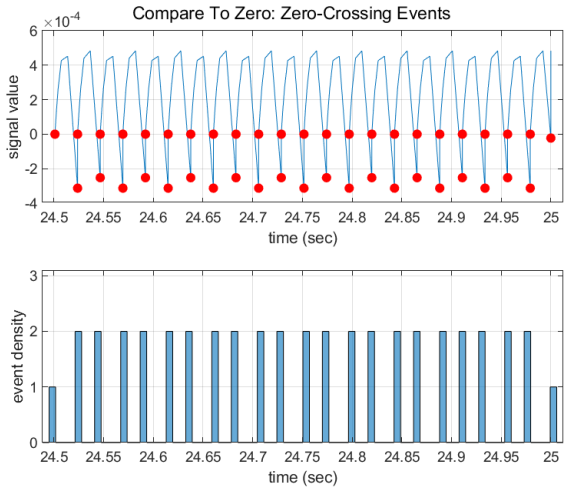


Consider the two-integrator implementation of the bounce model.



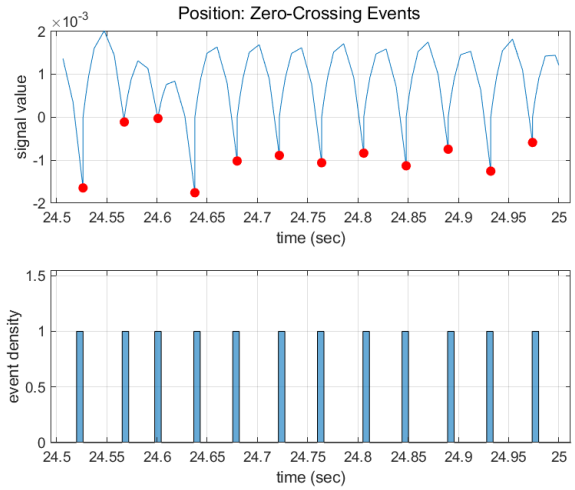
Profiling of the last 0.5 s of the simulation using the Solver Profiler shows that the simulation detects 44 zero-crossing events at the Compare To Zero block and 22 events at the output of the Position block.

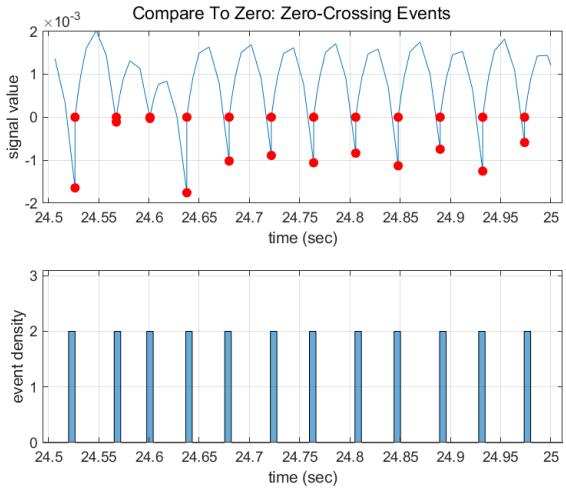




Increase the value of the **Relative tolerance** parameter to `1e-2` instead of the default `1e-3`. You can change this parameter in the **Solver Details** section of the **Solver** pane in the **Configuration Parameters** dialog or using `set_param` to specify `RelTol` as `'1e-2'`.

Profiling the last 0.5 s of the simulation with the new relative tolerance of the solver shows that it detects only 24 zero-crossing events at the Compare To Zero block and 12 events at the output of the Position block.





Zero-Crossing Detection in Blocks

A block can register a set of zero-crossing variables, each of which is a function of a state variable that can have a discontinuity. The zero-crossing function passes through zero from a positive or negative value when the corresponding discontinuity occurs. The registered zero-crossing variables are updated at the end of each simulation step, and any variable that has changed sign is identified as having had a zero-crossing event.

If any zero crossings are detected, the Simulink software interpolates between the previous and current values of each variable that changed sign to estimate the times of the zero crossings, that is, the discontinuities.

Note The Zero-Crossing detection algorithm can bracket zero-crossing events only for signals of data type `double`.

Blocks That Register Zero Crossings

The following table lists blocks that register zero crossings and explains how the blocks use the zero crossings.

Block	Number of Zero Crossing Detections
Abs	One, to detect when the input signal crosses zero in either the rising or falling direction.
Backlash	Two, one to detect when the upper threshold is engaged, and one to detect when the lower threshold is engaged.
Compare To Constant	One, to detect when the signal equals a constant.
Compare To Zero	One, to detect when the signal equals zero.
Dead Zone	Two, one to detect when the dead zone is entered (the input signal minus the lower limit), and one to detect when the dead zone is exited (the input signal minus the upper limit).

Block	Number of Zero Crossing Detections
Enable	One, If an Enable port is inside of a Subsystem block, it provides the capability to detect zero crossings. For details, “Using Enabled Subsystems” on page 10-10.
From File	One, to detect when the input signal has a discontinuity in either the rising or falling direction
From Workspace	One, to detect when the input signal has a discontinuity in either the rising or falling direction
Hit Crossing	One or two. If there is no output port, there is only one zero crossing to detect when the input signal hit the threshold value. If there is an output port, the second zero crossing is used to bring the output back to 0 from 1 to create an impulse-like output.
If	One, to detect when the If condition is met.
Integrator	If the reset port is present, to detect when a reset occurs. If the output is limited, there are three zero crossings: one to detect when the upper saturation limit is reached, one to detect when the lower saturation limit is reached, and one to detect when saturation is left.
MinMax	One, for each element of the output vector, to detect when an input signal is the new minimum or maximum.
Relational Operator	One, to detect when the specified relation is true.
Relay	One, if the relay is off, to detect the switch-on point. If the relay is on, to detect the switch-off point.
Saturation	Two, one to detect when the upper limit is reached or left, and one to detect when the lower limit is reached or left.
Second-Order Integrator	Five, two to detect when the state x upper or lower limit is reached, two to detect when the state dx/dt upper or lower limit is reached, and one to detect when a state leaves saturation.
Sign	One, to detect when the input crosses through zero.
Signal Editor	One, to detect when the input signal has a discontinuity in either the rising or falling direction
Step	One, to detect the step time.
Switch	One, to detect when the switch condition occurs.
Switch Case	One, to detect when the case condition is met.
Trigger	One, If a Triggered port is inside of a Subsystem block, it provides the capability to detect zero crossings. For details, see “Using Triggered Subsystems” on page 10-17.
Enabled and Triggered Subsystem	Two, one for the enable port and one for the trigger port. For details, see: “Using Enabled and Triggered Subsystems” on page 10-21

Note Zero-crossing detection is also available for a Stateflow chart that uses continuous-time mode. See “Configure a Stateflow Chart for Continuous-Time Simulation” (Stateflow) for more information.

Implementation Example: Saturation Block

An example of a Simulink block that registers zero crossings is the Saturation block. Zero-crossing detection identifies these state events in the Saturation block:

- The input signal reaches the upper limit.
- The input signal leaves the upper limit.
- The input signal reaches the lower limit.
- The input signal leaves the lower limit.

Simulink blocks that define their own state events are considered to have *intrinsic zero crossings*. Use the Hit Crossing block to receive explicit notification of a zero-crossing event. See “Blocks That Register Zero Crossings” on page 3-22 for a list of blocks that incorporate zero crossings.

The detection of a state event depends on the construction of an internal zero-crossing signal. This signal is not accessible by the block diagram. For the Saturation block, the signal that is used to detect zero crossings for the upper limit is `zcSignal = UpperLimit - u`, where `u` is the input signal.

Zero-crossing signals have a direction attribute, which can have these values:

- *rising* — A zero crossing occurs when a signal rises to or through zero, or when a signal leaves zero and becomes positive.
- *falling* — A zero crossing occurs when a signal falls to or through zero, or when a signal leaves zero and becomes negative.
- *either* — A zero crossing occurs if either a rising or falling condition occurs.

For the Saturation block upper limit, the direction of the zero crossing is *either*. This enables the entering and leaving saturation events to be detected using the same zero-crossing signal.

See Also

More About

- “Algebraic Loop Concepts” on page 3-27
- “Compare Solvers” on page 3-6

Zero-Crossing Algorithms

The Simulink software includes two zero-crossing detection algorithms: Nonadaptive and Adaptive.

To choose the algorithm, either use the **Algorithm** option in the Solver pane of the Configuration Parameter dialog box, or use the `ZeroCrossAlgorithm` command. The command can either be set to `'Nonadaptive'` or `'Adaptive'`.

The Nonadaptive algorithm is provided for backwards compatibility with older versions of Simulink and is the default. It brackets the zero-crossing event and uses increasingly smaller time steps to pinpoint when the zero crossing has occurred. Although adequate for many types of simulations, the Nonadaptive algorithm can result in very long simulation times when a high degree of 'chattering' (high frequency oscillation around the zero-crossing point) is present.

The Adaptive algorithm dynamically turns the bracketing on and off, and is a good choice when:

- The system contains a large amount of chattering.
- You wish to specify a guard band (tolerance) around which the zero crossing is detected.

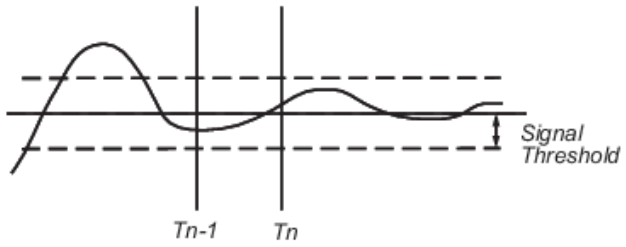
The Adaptive algorithm turns off zero-crossing bracketing (stops iterating) if either of the following are satisfied:

- The zero crossing error is exceeded. This is determined by the value specified in the **Signal threshold** option in the Solver pane of the Configuration Parameters dialog box. This can also be set with the `ZCThreshold` command. The default is `Auto`, but you can enter any real number greater than zero for the tolerance.
- The system has exceeded the number of consecutive zero crossings specified in the **Number of consecutive zero crossings** option in the Solver pane of the Configuration Parameters dialog box. Alternatively, this can be set with the `MaxConsecutiveZCs` command.

Signal Threshold for Adaptive Zero-Crossing Detection

The Adaptive algorithm automatically sets a tolerance for zero-crossing detection. Alternatively, you can set the tolerance by entering a real number greater than or equal to zero in the Configuration Parameters Solver pane, `Signal threshold` pull down. This option only becomes active when the zero-crossing algorithm is set to `Adaptive`.

This graphic shows how the Signal threshold sets a window region around the zero-crossing point. Signals falling within this window are considered as being at zero.



The zero-crossing event is bracketed by time steps T_{n-1} and T_n. The solver iteratively reduces the time steps until the state variable lies within the band defined by the signal threshold, or until the number of consecutive zero crossings equals or exceeds the value in the Configuration Parameters Solver pane, `Number of consecutive zero crossings` pull down.

It is evident from the figure that increasing the signal threshold increases the distance between the time steps which will be executed. This often results in faster simulation times, but might reduce accuracy.

Algebraic Loop Concepts

In this section...

In a Simulink model, an algebraic loop occurs when a signal loop exists with only direct feedthrough blocks within the loop. Direct feedthrough means that Simulink needs the value of the block’s input signal to compute its output at the current time step. Such a signal loop creates a circular dependency of block outputs and inputs in the same time-step. This results in an algebraic equation that needs solving at each time-step, adding computational cost to the simulation.

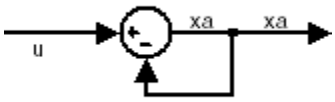
Some examples of blocks with direct feedthrough inputs are:

- Math Function
- Gain
- Product
- State-Space, when the D matrix coefficient is nonzero
- Sum
- Transfer Fcn, when the numerator and denominator are of the same order
- Zero-Pole, when the block has as many zeros as poles

Nondirect feedthrough blocks maintain a State variable. Two examples are Integrator and Unit Delay.

Tip To determine if a block has direct feedthrough, read the **Characteristics** section of the block reference page.

The figure shows an example of an algebraic loop. The Sum block is an algebraic variable x_a that is constrained to equal the first input u minus x_a (for example, $x_a = u - x_a$).



The solution of this simple loop is $x_a = u/2$.

Mathematical Interpretation

Simulink contains a suite of numerical solvers for simulating ordinary differential equations (ODEs), which are systems of equations that you can write as

$$\dot{x} = f(x, t),$$

where x is the state vector and t is the independent time variable.

Some systems of equations contain additional constraints that involve the independent variable and the state vector, but not the derivative of the state vector. Such systems are called differential algebraic equations (DAEs),

The term algebraic refers to equations that do not involve any derivatives. You can express DAEs that arise in engineering in the semi-explicit form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{x_a}, t)$$
$$0 = \mathbf{g}(\mathbf{x}, \mathbf{x_a}, t),$$

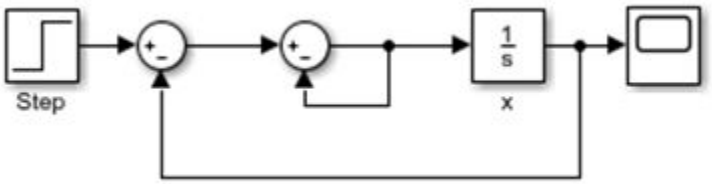
where:

- **f** and **g** can be vector functions.
- The first equation is the differential equation.
- The second equation is the algebraic equation.
- The vector of differential variables is **x**.
- The vector of algebraic variables is $\mathbf{x_a}$.

In Simulink models, algebraic loops are algebraic constraints. Models with algebraic loops define a system of differential algebraic equations. Simulink solves the algebraic equations (the algebraic loop) numerically for x_a at each step of the ODE solver.

The model in the figure is equivalent to this system of equations in semi-explicit form:

$$\dot{x} = f(x, x_a, t) = x_a$$
$$0 = g(x, x_a, t) = -x + u - 2x_a .$$



At each step of the ODE solver, the algebraic loop solver must solve the algebraic constraint for x_a before calculating the derivative $\dot{x}$.

Physical Interpretation

Algebraic constraints:

- Occur when modeling physical systems, often due to conservation laws, such as conservation of mass and energy
- Occur when you choose a particular coordinate system for a model
- Help impose design constraints on system responses in a dynamic system

Use Simscape to model systems that span mechanical, electrical, hydraulic, and other physical domains as physical networks. Simscape constructs the DAEs that characterize the behavior of a model. The software integrates these equations with the rest of the model and then solves the DAEs directly. Simulink solves the variables for the components in the different physical domains simultaneously, avoiding problems with algebraic loops.

Artificial Algebraic Loops

An *artificial algebraic loop* occurs when an atomic subsystem or Model block causes Simulink to detect an algebraic loop, even though the contents of the subsystem do not contain a direct feedthrough from the input to the output. When you create an atomic subsystem, all Inport blocks are direct feedthrough, resulting in an algebraic loop.

Start with the included model, which represents a simple proportional control of the plant described by

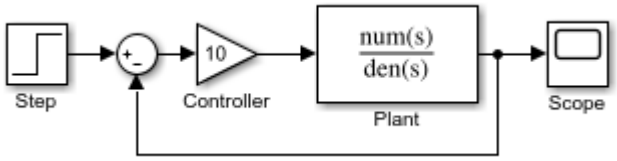
$$G(s) = \frac{1}{s^2 + 2s + 1}$$

which can be rewritten in state-space form as

$$\dot{x} = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

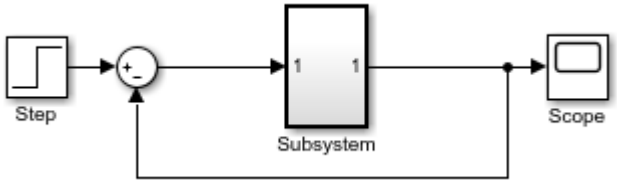
$$y = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

The system has neither algebraic variables nor direct feedthrough and does not contain an algebraic loop.



Modify the model as described in the following steps:

1. Enclose the Controller and Plant blocks in a subsystem.
2. In the subsystem dialog box, select **Treat as atomic unit** to make the subsystem atomic.
3. In the **Diagnostics** pane of the Model Configuration Parameters, set the **Algebraic Loop** parameter to `error`.



When simulating this model, an algebraic loop occurs because the subsystem is direct feedthrough, even though the path within the atomic subsystem is not direct feedthrough. Simulation stops with an algebraic loop error.

How the Algebraic Loop Solver Works

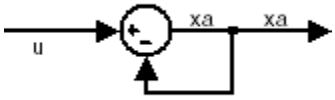
When a model contains an algebraic loop, Simulink uses a nonlinear solver at each time step to solve the algebraic loop. The solver performs iterations to determine the solution to the algebraic constraint, if there is one. As a result, models with algebraic loops can run more slowly than models without algebraic loops.

Simulink uses a dogleg trust region algorithm to solve algebraic loops. The tolerance used is smaller than the ODE solver `Reltol` and `Abstol`. This is because Simulink uses the “explicit ODE method” to solve Index-1 differential algebraic equations (DAEs).

For the algebraic loop solver to work,

- There must be one block where the loop solver can break the loop and attempt to solve the loop.
- The model should have real double signals.
- The underlying algebraic constraint must be a smooth function

For example, suppose your model has a Sum block with two inputs—one additive, the other subtractive. If you feed the output of the Sum block to one of the inputs, you create an algebraic loop where all of the blocks include direct feedthrough.



The Sum block cannot compute the output without knowing the input. Simulink detects the algebraic loop, and the algebraic loop solver solves the loop using an iterative loop. In the Sum block example, the software computes the correct result this way:

$$x_a(t) = u(t) / 2. \tag{3-1}$$

The algebraic loop solver uses a gradient-based search method, which requires continuous first derivatives of the algebraic constraint that correspond to the algebraic loop. As a result, if the algebraic loop contains discontinuities, the algebraic loop solver can fail.

For more information, see Solving Index-1 DAEs in MATLAB and Simulink [1]

Trust-Region and Line-Search Algorithms in the Algebraic Loop Solver

The Simulink algebraic loop solver uses one of two algorithms to solve algebraic loops:

- Trust-Region
- Line-Search

1. Shampine, Lawrence F., M.W.Reichelt, and J.A.Kierzenka. ”Solving Index-1 DAEs in MATLAB and Simulink.”*Siam Review.*Vol.18,No.3,1999,pp.538–552.

By default, the algebraic loop solver uses the trust-region algorithm.

If the algebraic loop solver cannot solve the algebraic loop with the trust-region algorithm, try simulating the model using the line-search algorithm.

To switch to the line-search algorithm, at the MATLAB command line, enter:

```
set_param(model_name, 'AlgebraicLoopSolver', 'LineSearch');
```

To switch back to the trust-region algorithm, at the MATLAB command line, enter:

```
set_param(model_name, 'AlgebraicLoopSolver', 'TrustRegion');
```

For more information, see:

- Shampine and Reichelt's nleqn.m code
- The Fortran program HYBRD1 in the User Guide for MINPACK-1 [2]
- Powell's "A Fortran subroutine for solving systems in nonlinear equations," in *Numerical Methods for Nonlinear Algebraic Equations*[3]
- "Trust-Region Methods for Nonlinear Minimization" (Optimization Toolbox).
- "Line Search" (Optimization Toolbox).

Limitations of the Algebraic Loop Solver

Algebraic loop solving is an iterative process. The Simulink algebraic loop solver is successful only if the algebraic loop converges to a definite answer. When the loop fails to converge, or converges too slowly, the simulation exits with an error.

The algebraic loop solver cannot solve algebraic loops that contain any of the following:

- Blocks with discrete-valued outputs
- Blocks with nondouble or complex outputs
- Discontinuities
- Stateflow charts

Implications of Algebraic Loops in a Model

If your model contains an algebraic loop:

- You cannot generate code for the model.
- The Simulink algebraic loop solver might not be able to solve the algebraic loop.
- While Simulink is trying to solve the algebraic loop, the simulation can execute slowly.

  For most models, the algebraic loop solver is computationally expensive for the first time step. Simulink solves subsequent time steps rapidly because a good starting point for x_a is available from the previous time step.

2. More,J.J.,B.S.Garbow, and K.E.Hillstrom. *User guide for MINPACK-1*. Argonne, IL:Argonne National Laboratory,1980.
3. Rabinowitz, Philip, ed. *Numerical Methods for Nonlinear Algebraic Equations*, New York: Gordon and Breach Science Publishers, 1970.

See Also

“Compare Solvers” on page 3-6 | “Zero-Crossing Detection” on page 3-10 | Algebraic Constraint | Descriptor State-Space

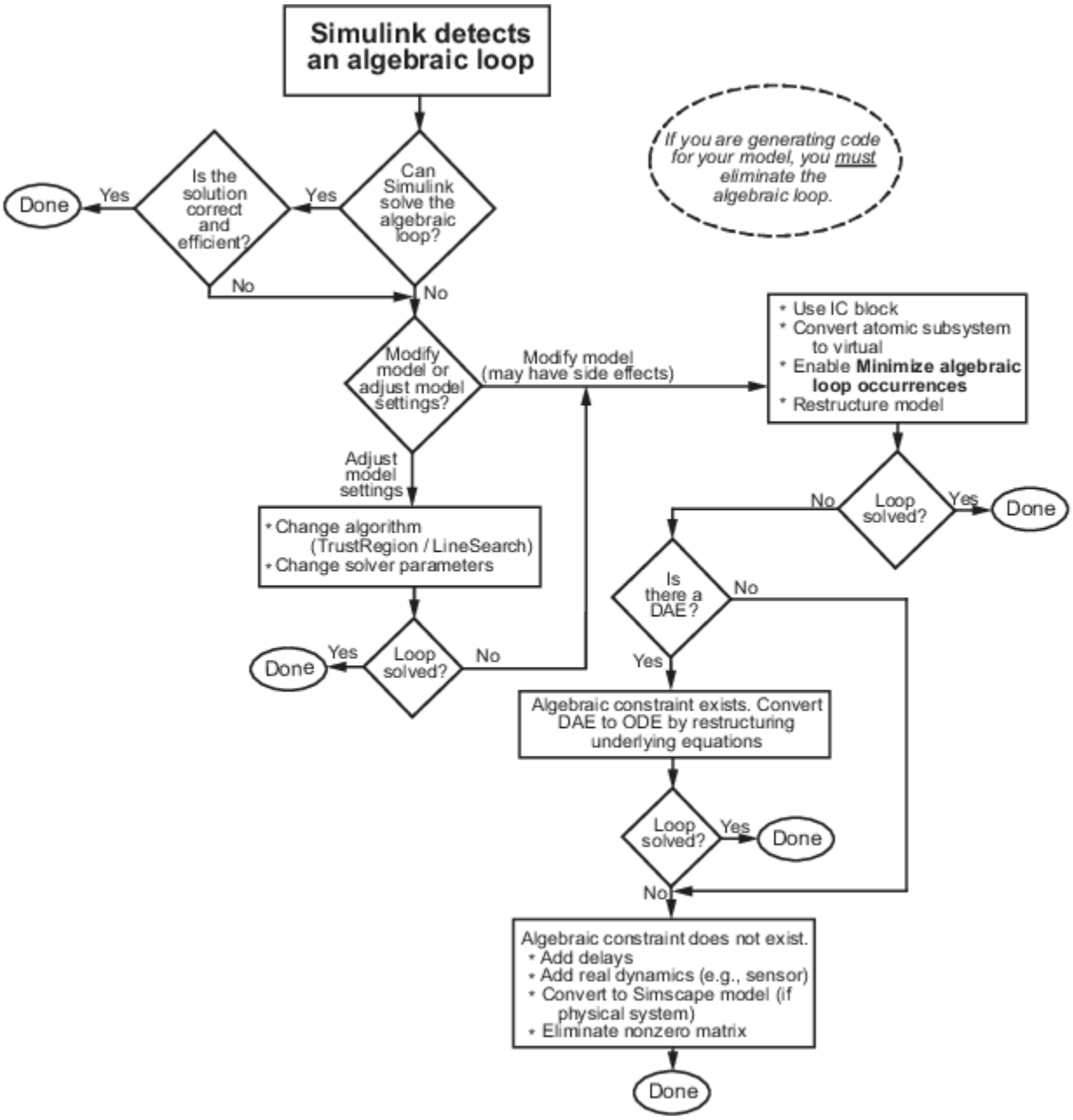
More About

- “Identify Algebraic Loops in Your Model” on page 3-33
- “Remove Algebraic Loops” on page 3-36
- “Modeling Considerations with Algebraic Loops” on page 3-52

Identify Algebraic Loops in Your Model

If Simulink reports an algebraic loop in your model, the algebraic loop solver may be able to solve the loop. If Simulink cannot solve the loop, there are several techniques to eliminate the loop.

Use this workflow to decide how you want to eliminate an algebraic loop.



Highlight Algebraic Loops in the Model

Use `getAlgebraicLoops` to identify algebraic loops in a model and highlight them in the Simulink Editor. With this approach:

- You can traverse multiple layers of model hierarchy to locate algebraic loops.
- You can identify real and artificial algebraic loops.
- You can visualize all loops in your model simultaneously.
- You do not need to drill in and out of the model, across boundaries.
- You do not need to detect loops in serial order. Also, you do not need to compile the model every time you detect and solve a loop. Therefore you can solve loops quickly.

You perform algebraic loop highlighting on an entire model, not on specific subsystems.

1. Open the model.
2. In the **Diagnostics** pane of Model Configuration Parameters, set **Algebraic loop** to `none` or `warning`. Setting this parameter to `error` prevents the model from compiling.
3. Compile the model without any errors. The model must compile before you can highlight any algebraic loops.
4. At the MATLAB command prompt, enter:

   ```
   Simulink.BlockDiagram.getAlgebraicLoops(bdroot)
   ```

The `getAlgebraicLoops` function highlights algebraic loops in the model, including algebraic loops in subsystems. It also creates a report with information about each loop:

- Solid lines represent real algebraic loops.
- Dotted lines represent artificial algebraic loops.
- A red highlight appears around a block assigned with an algebraic variable.
- The **Loop ID** helps you identify the system that contains a particular loop.

Customize the report by selecting or clearing the **Visible** check box for a loop.

Once you have identified algebraic loops in a model, you can remove them by editing the model. Close the highlight report and make changes to the model. You can edit the model only after you close the report.

Simulink does not save loop highlighting. Closing the model or exiting the display removes the loop highlighting.

Use the Algebraic Loop Diagnostic

Simulink detects algebraic loops during simulation initialization, for example, when you update your diagram. You can set the Algebraic loop diagnostic to report an error or warning if the software detects any algebraic loops in your model.

In the **Diagnostics** pane of the Model Configuration Parameters, set the **Algebraic loop** parameter:

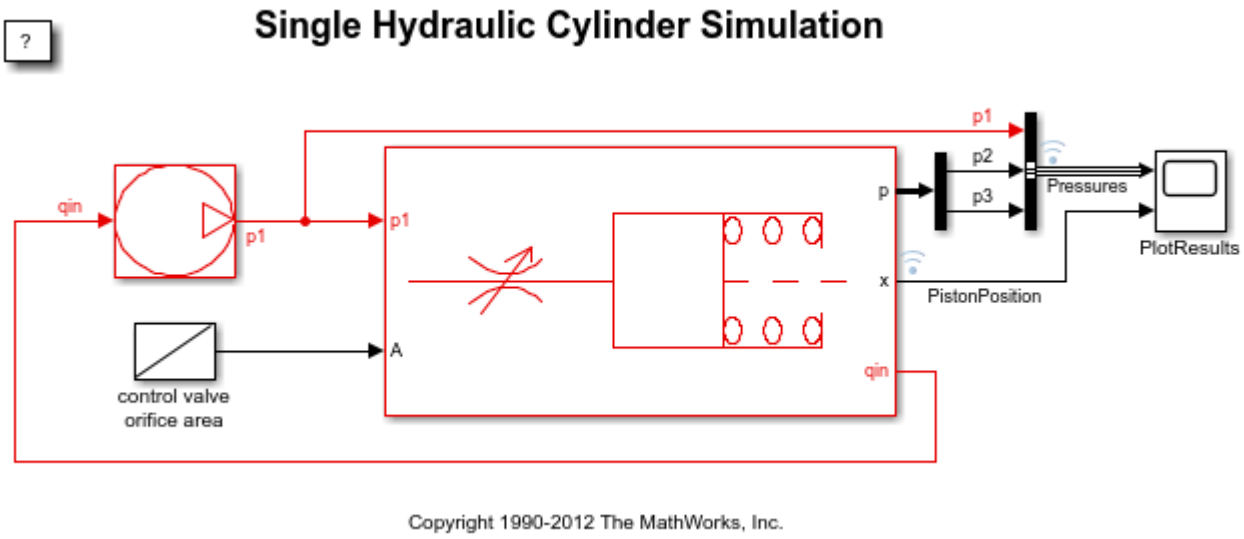
- `none` -- Simulink tries to solve the algebraic loop; reports an error only if the algebraic loop cannot be solved.

- `warning` -- Algebraic loops result in warnings. Simulink tries to solve the algebraic loop; reports an error only if the algebraic loop cannot be solved.
- `error` -- Algebraic loops stop the initialization. Review the loop manually before Simulink tries to solve the loop.

This example shows how to use the algebraic loop diagnostic to highlight algebraic loops in the `sldemo_hydcyl` model.

1. Open the `sldemo_hydcyl` model
2. In the **Diagnostics** pane of the Model Configuration Parameters, set the **Algebraic loop** parameter to `error`.
3. Try to simulate the model

When Simulink detects an algebraic loop during initialization, the simulation stops. The Diagnostic Viewer displays an error message and lists all the blocks in the model that are part of that algebraic loop. In the model, red highlights show the blocks and signals that make up the loop.



To remove the highlights, close the Diagnostic Viewer.

See Also

More About

- “Algebraic Loop Concepts” on page 3-27
- “Remove Algebraic Loops” on page 3-36
- “Modeling Considerations with Algebraic Loops” on page 3-52

Remove Algebraic Loops

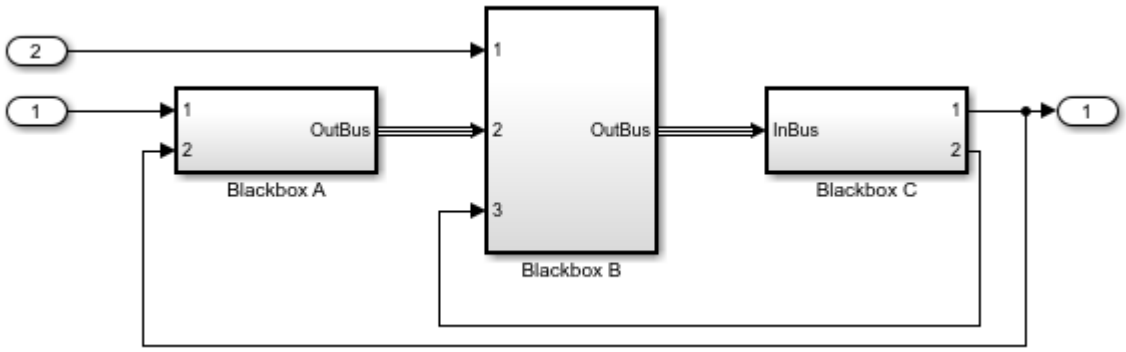
Use these techniques to remove algebraic loops in a model.

Introduce a Delay to Remove Algebraic Loops

This example demonstrates how to remove algebraic loops in a model by introducing delays between blocks in a loop. This is one approach to remove algebraic loops in larger models where such loops can occur due to feedback between atomic subsystems.

Consider the model attached with this example. There are two algebraic loops caused by the atomic subsystems in the model

- `Blackbox A` -> `Blackbox B` -> `Blackbox C` -> `Blackbox A`
- `Blackbox B` -> `Blackbox C` -> `Blackbox B`



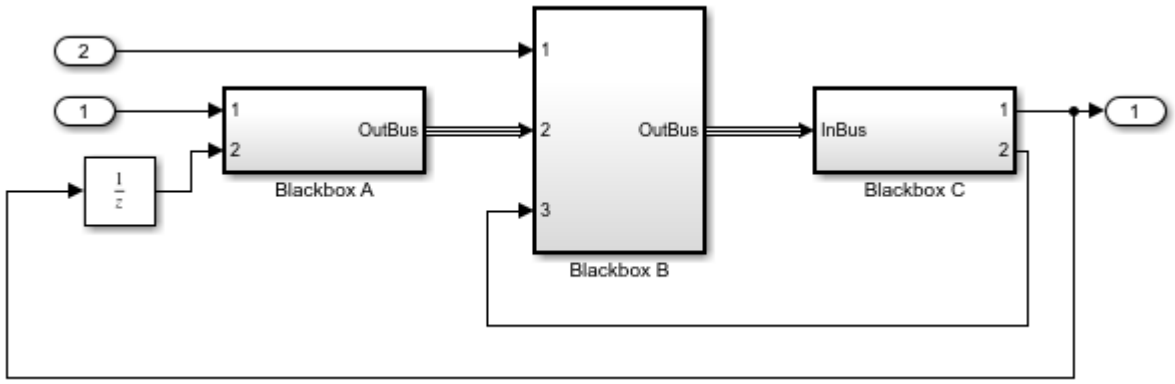
When you update this model, Simulink® detects the loop `Blackbox A` -> `Blackbox B` -> `Blackbox C` -> `Blackbox A`

Since you do not know the contents of these subsystems, break the loops by adding a Unit Delay block outside the subsystems. There are three ways to use the Unit Delay block to break these loops:

- Add a Unit Delay between `Blackbox A` and `Blackbox C`.
- Add a Unit Delay between `Blackbox B` and `Blackbox C`.
- Add Unit Delay blocks to both algebraic loops.

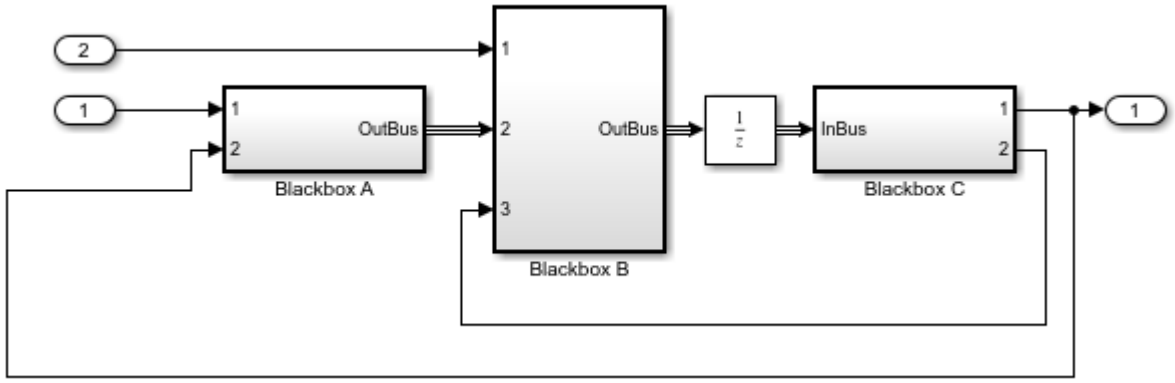
Add a unit delay between `BlackBox A` and `BlackBox C`

If you add a unit delay on the feedback signal between the subsystems `Blackbox A` and `Blackbox C`, you introduce the minimum number of unit delays (1) to the system. By introducing the delay before `Blackbox A`, `Blackbox B` and `Blackbox C` use data from the current time step.



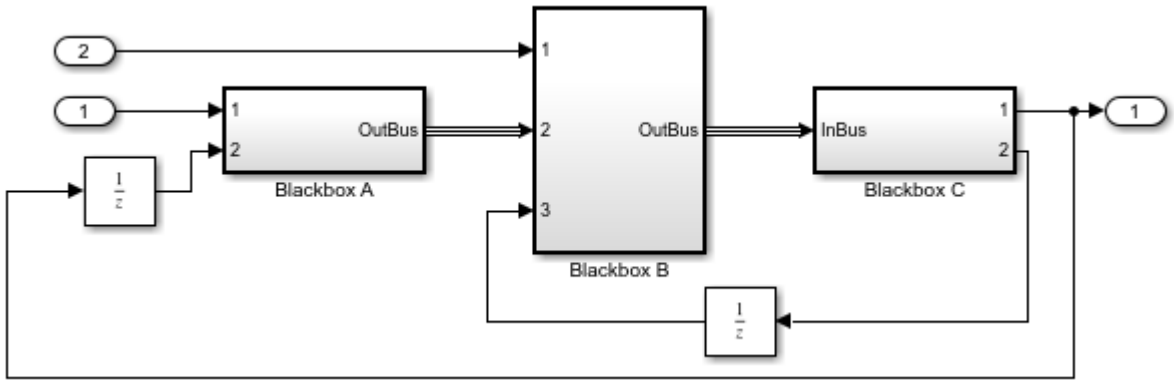
Add a unit delay between `BlackBox B` and `BlackBox C`

If you add a unit delay between the subsystems `Blackbox B` and `Blackbox C`, you break the algebraic loop between `Blackbox B` and `Blackbox C`. In addition, you break the loop between `Blackbox A` and `Blackbox C`, because that signal completes the algebraic loop. By inserting the Unit Delay block before `Blackbox C`, `Blackbox C` now works with data from the previous time step only.



Add unit delays to both algebraic loops

In the example here, you insert Unit Delay blocks to break both algebraic loops. In this model, BlackBox_A and BlackBox_B use data from the previous time step. BlackBox_C uses data from the current time step.



Solve Algebraic Loops Manually

If Simulink cannot solve the algebraic loop, the software reports an error. Use one of these techniques to solve the loop manually:

- Restructure the underlying DAEs using techniques such as differentiation or change of coordinates. These techniques put the DAEs in a form that is easier for the algebraic loop solver to solve.
- Convert the DAEs to ODEs, which eliminates any algebraic loops.
- “Create Initial Guesses Using the IC and Algebraic Constraint Blocks” on page 3-38

Create Initial Guesses Using the IC and Algebraic Constraint Blocks

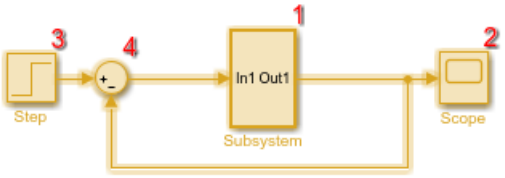
Your model might contain loops for which the loop solver cannot converge without a good, initial guess for the algebraic states. You can specify an initial guess for the algebraic state variables, but use this technique only when you think the loop is legitimate.

There are two ways to specify an initial guess:

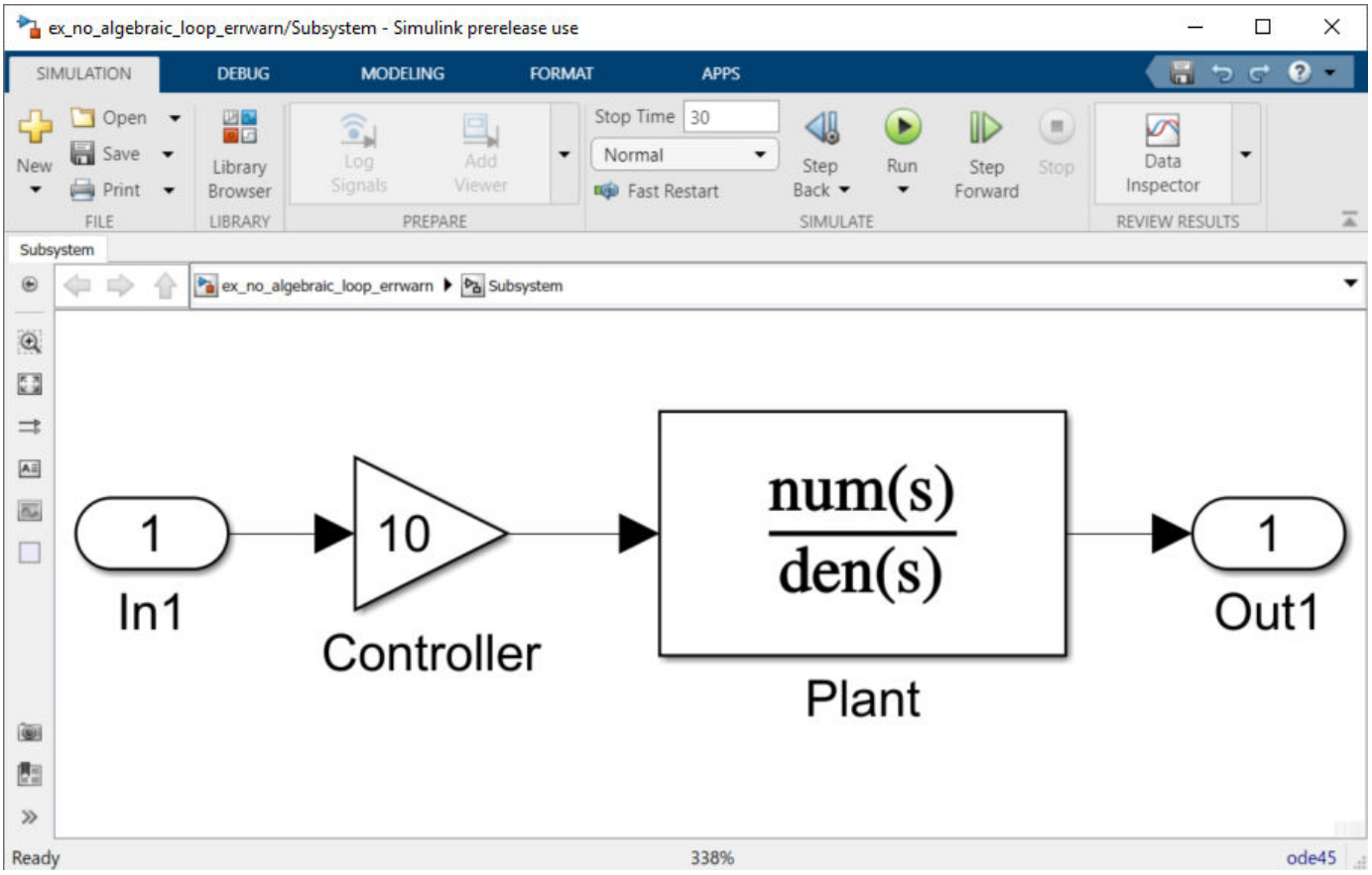
- Place an IC block in the algebraic loop.
- Specify an initial guess for a signal in an algebraic loop using an Algebraic Constraint block.

How Simulink Eliminates Artificial Algebraic Loops

When you enable **Minimize algebraic loop occurrences**, Simulink tries to eliminate artificial algebraic loops. In this example, the model contains an atomic subsystem that causes an artificial algebraic loop.



The contents of the atomic subsystem are not direct feedthrough, but Simulink identifies the atomic subsystem as direct feedthrough.



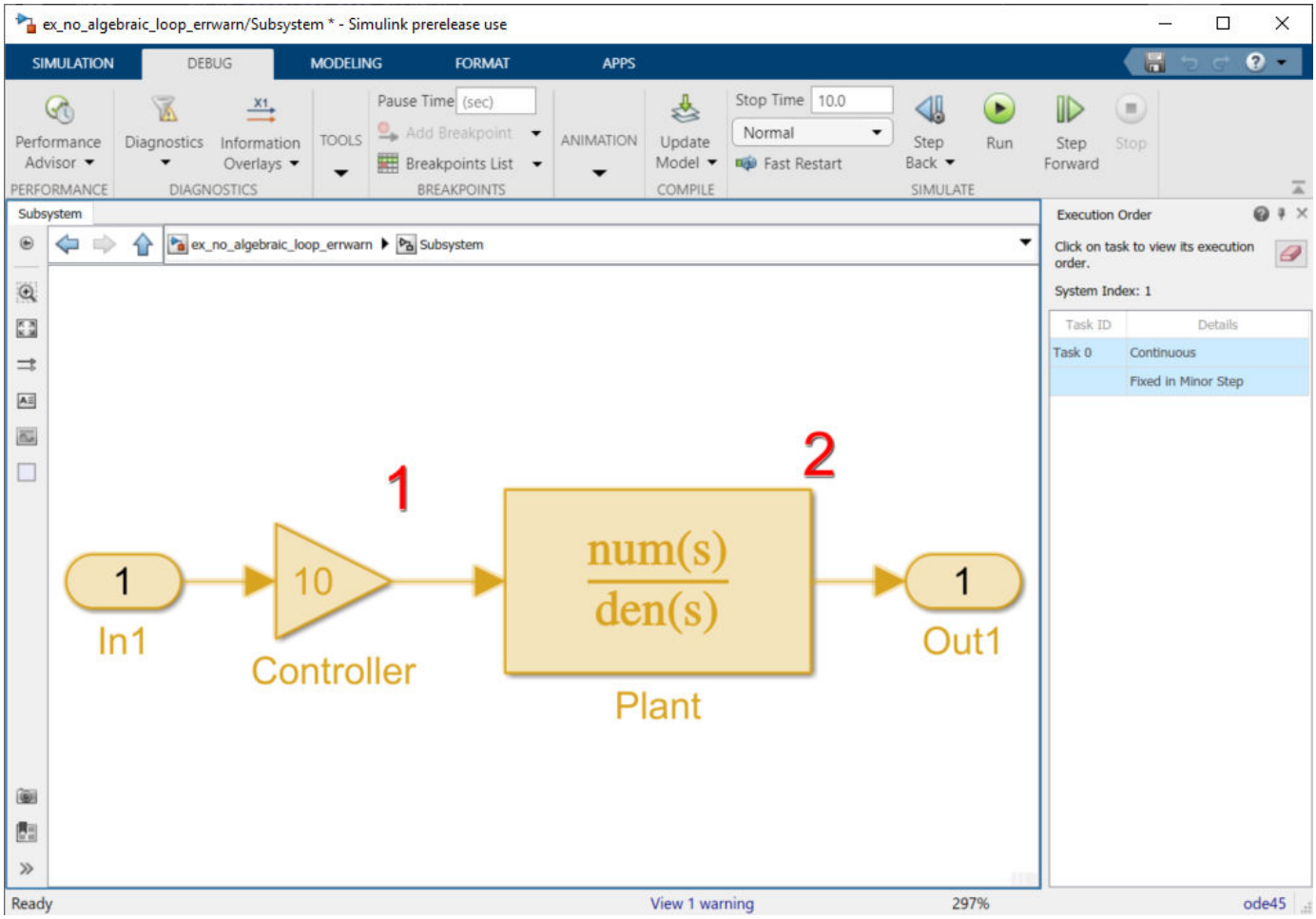
If the **Algebraic loop** diagnostic is set to `error`, simulating the model results in an error because the model contains an artificial algebraic loop involving its atomic subsystem.

To eliminate this algebraic loop,

1. Create the model from the preceding graphics, with the atomic subsystem that causes the artificial algebraic loop.
2. In the **Diagnostics** pane of Model Configuration Parameters, set the **Algebraic loop** parameter to `warning` or `none`.
3. In the **Data Import/Export** pane, make sure the **Signal logging** parameter is disabled. If signal logging is enabled, Simulink cannot eliminate artificial algebraic loops.
4. To display the block execution order for this model and the atomic subsystem, in the **Debug** tab, select **Information Overlays** > **Execution Order**.

   Reviewing the execution order can help you understand how to eliminate the artificial algebraic loop.

   All the blocks in the subsystem execute at the same level: 1. (0 is the lowest level, indicating the first blocks to execute.)

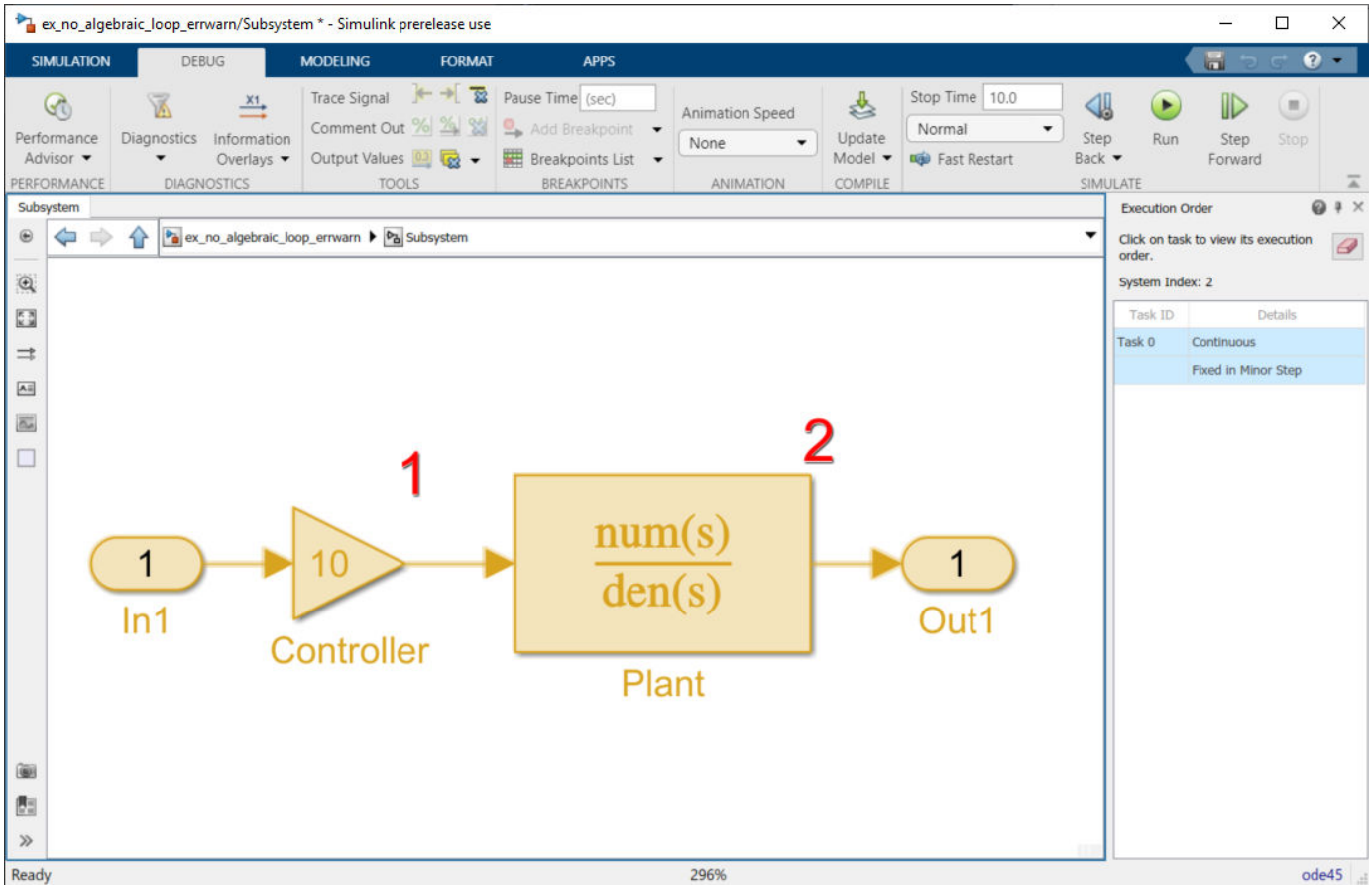


Note For more information about block execution order, see “Control and Display Execution Order” on page 36-25.

5 In the top-level model’s **Subsystem Parameters** dialog box, select **Minimize algebraic loop occurrences**. This parameter directs Simulink to try to eliminate the algebraic loop that contains the atomic subsystem, when it simulates the model. Save the changes.

6 In the **Modeling** tab, click **Update Model** to recalculate the execution order.

Now there are two levels of execution order inside the subsystem: 1 and 2.

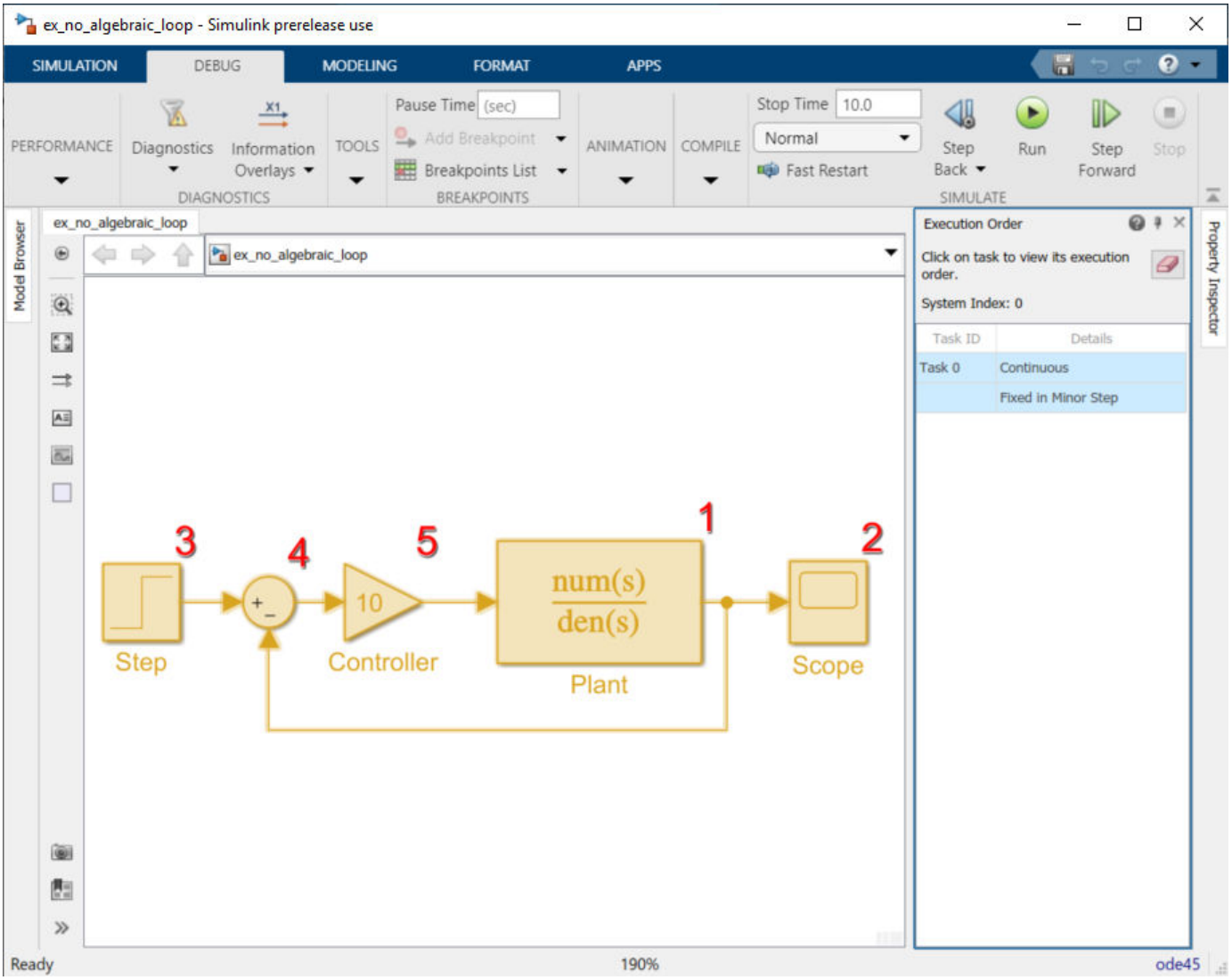


To eliminate the artificial algebraic loop, Simulink tries to make the input of the subsystem or referenced model non-direct feedthrough.

When you simulate a model, all blocks execute methods in this order:

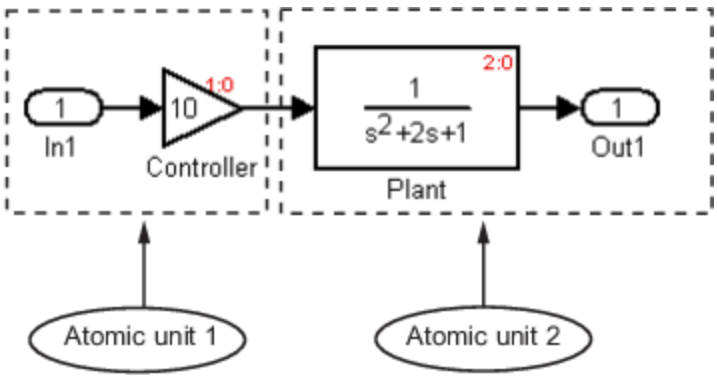
1. `mdlOutputs`
2. `mdlDerivatives`
3. `mdlUpdate`

In the original version of this model, the execution of the `mdlOutputs` method starts with the Plant block because the Plant block is non-direct feedthrough. The execution finishes with the Controller block.



For more information about these methods, see “Simulink Models”.

If you enable the **Minimize algebraic loop occurrences** parameter for the atomic subsystem, Simulink divides the subsystem into two atomic units.



These conditions are true:

- Atomic unit 2 is not direct feedthrough.
- Atomic unit 1 has only a `mdlOutputs` method.

Only the `mdlDerivatives` or `mdlUpdate` methods of Atomic unit 2 need the output of Atomic unit 1. Simulink can execute what normally would have been executed during the `mdlOutput` method of Atomic unit 1 in the `mdlDerivatives` methods of Atomic unit 2.

The new execution order for the model is:

1. `mdlOutputs` method of model
   a. `mdlOutputs` method of Atomic unit 2
   b. `mdlOutputs` methods of other blocks
2. `mdlDerivatives` method of model
   a. `mdlOutputs` method of Atomic unit 1
   b. `mdlDerivatives` method of Atomic unit 2
   c. `mdlDerivatives` method of other blocks

For the **Minimize algebraic loop occurrences** technique to be successful, the subsystem or referenced model must have a non-direct-feedthrough block connected directly to an Inport. Simulink can then set the `DirectFeedthrough` property of the block Inport to `false` to indicate that the input port does not have direct feedthrough.

Eliminate Artificial Algebraic Loops Caused by Atomic Subsystems

If an atomic subsystem causes an artificial algebraic loop, convert the atomic subsystem to a virtual subsystem. This change has no effect on the behavior of the model. When the subsystem is atomic and you simulate the model, Simulink invokes the algebraic loop solver. The solver terminates after one iteration. The algebraic loop is automatically solved because there is no algebraic constant. After you make the subsystem virtual, Simulink does not invoke the algebraic loop solver during simulation.

To convert an atomic subsystem to a virtual subsystem:

1. Open the model that contains the atomic subsystem.
2. Right-click the atomic subsystem and select **Subsystem Parameters**.
3. Clear the **Treat as atomic unit** parameter.
4. Save the changes.

If you replace the atomic subsystem with a virtual subsystem and the simulation still fails with an algebraic loop error, examine the model for one of these:

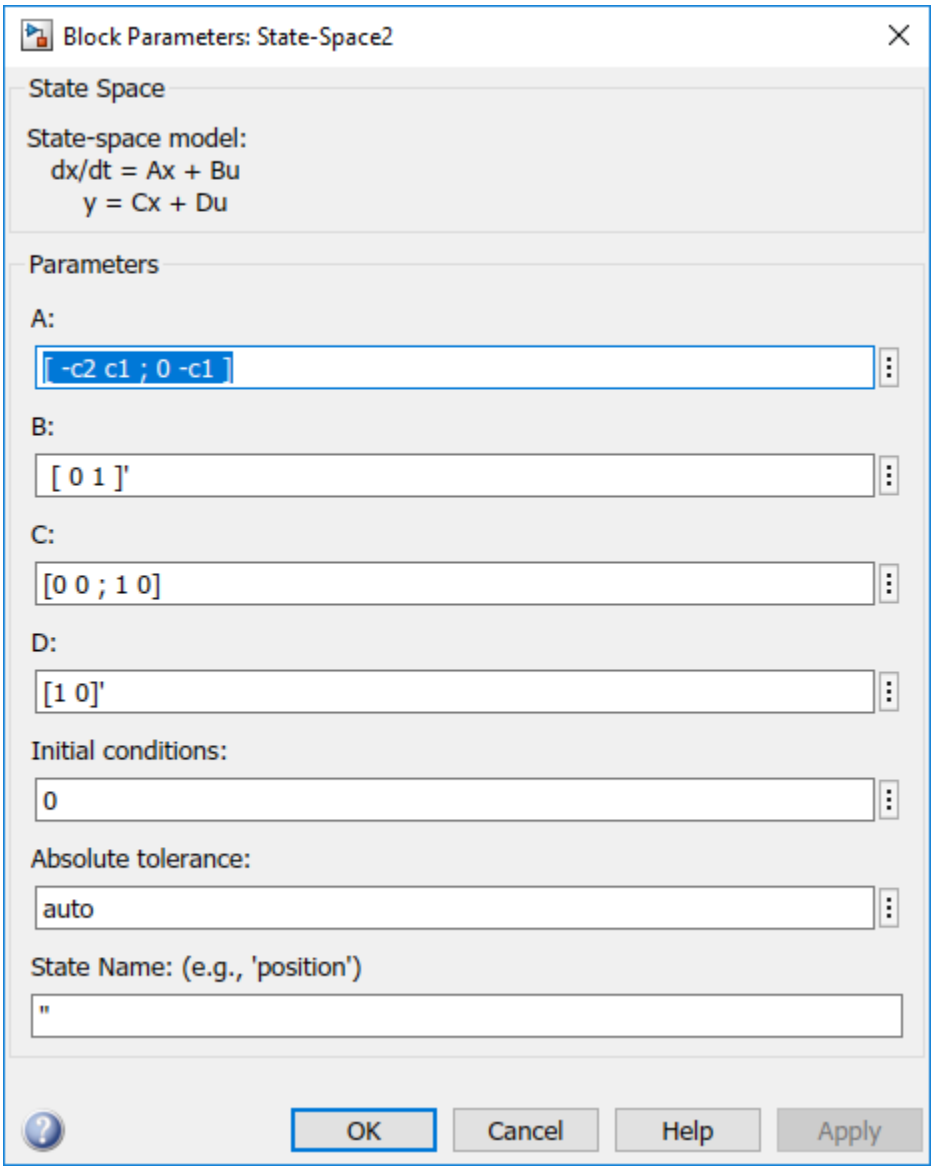
- An algebraic constraint
- An artificial algebraic loop that was not caused by this atomic subsystem

Bundled Signals That Create Artificial Algebraic Loops

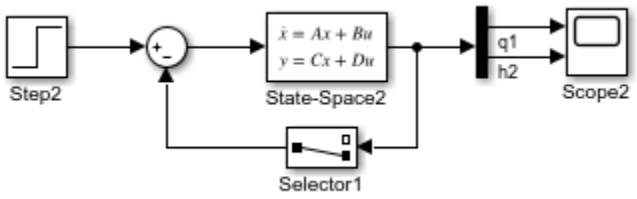
Some models bundle signals together. This bundling can cause Simulink to detect an algebraic loop, even when an algebraic constraint does not exist. If you redirect one or more signals, you may be able to remove the artificial algebraic loop.

In this example, a linearized model simulates the dynamics of a two-tank system fed by a single pump. In this model:

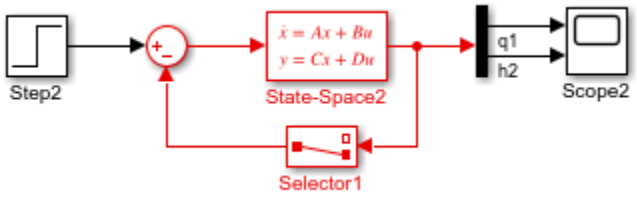
- Output `q1` is the rate of the fluid flow into the tank from the pump.
- Output `h2` is the height of the fluid in the second tank.
- The State-Space block defines the dynamic response of the tank system to the pump operation:



- The output from the State-Space block is a vector that contains `q1` and `h2`.

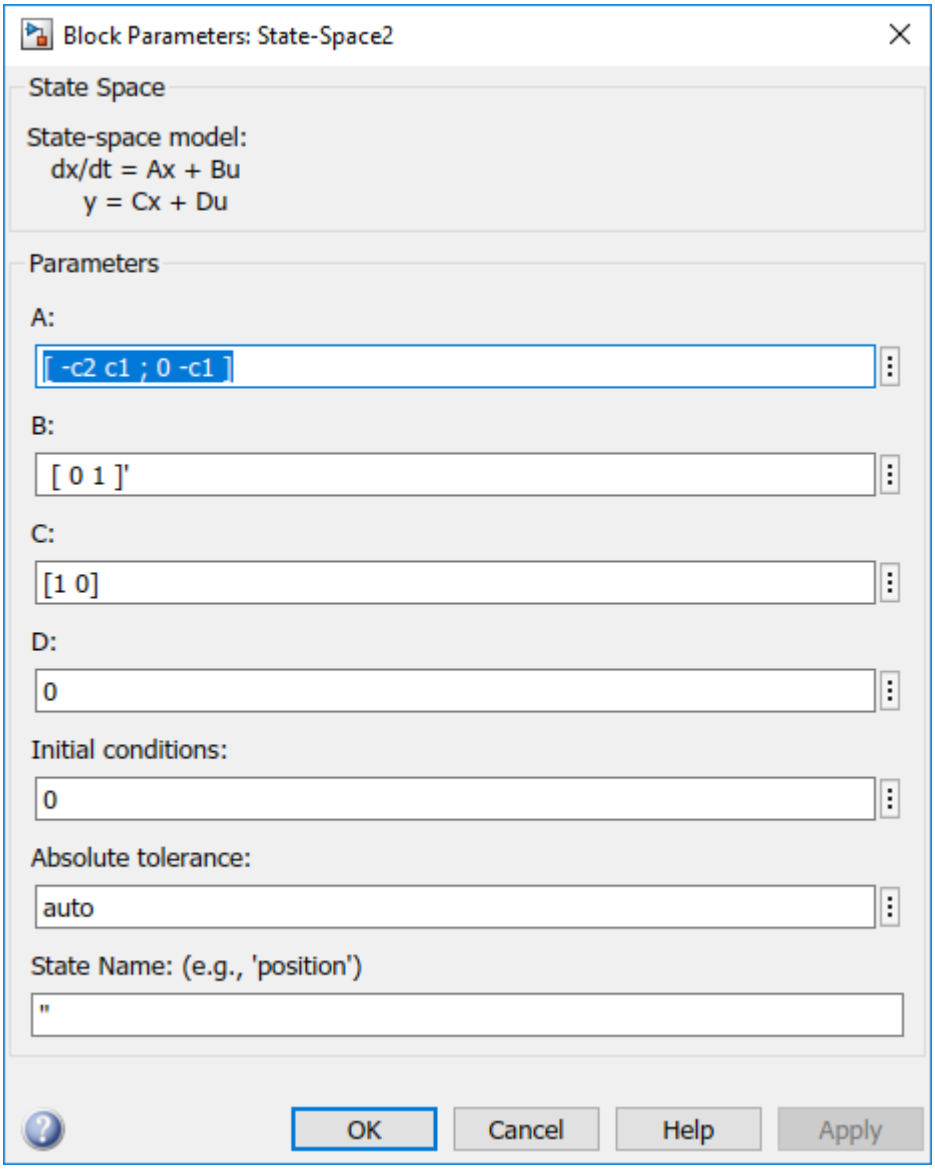


If you simulate this model with the **Algebraic loop** parameter set to `warn` or `error`, Simulink identifies the algebraic loop.



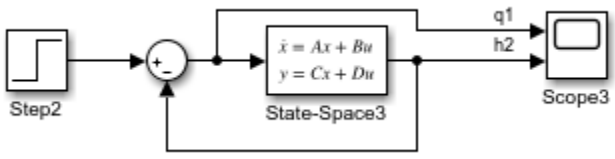
To eliminate this algebraic loop:

1 Change the `C` and `D` matrices as follows:



2 Pass `q1` directly to the Scope instead of through the State-Space block.

Now, the input (`q1`) does not pass directly to the output (the `D` matrix is 0), so the State-Space block no longer has direct feedthrough. The feedback signal has only one element now, so the Selector block is no longer necessary, as you can see in the following model.



Model and Block Parameters to Diagnose and Eliminate Artificial Algebraic Loops

There are two parameters to consider when you think that your model has an artificial algebraic loop:

- **Minimize algebraic loop occurrences** parameter — Specify that Simulink try to eliminate any artificial algebraic loops for:
  - Atomic subsystems — In the Subsystem Parameters dialog box, select **Minimize algebraic loop occurrences**.
  - Model blocks — For the referenced model, in the **Model Referencing** pane of Configuration Parameters, select **Minimize algebraic loop occurrences**.
- **Minimize algebraic loop** parameter — Specifies what diagnostic action Simulink takes if the **Minimize algebraic loop occurrences** parameter has no effect.

  The **Minimize algebraic loop** parameter is in the **Diagnostics** pane of Configuration Parameters. The diagnostic actions for this parameter are:

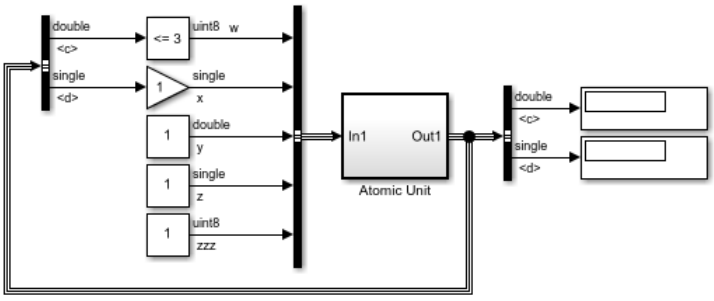
Setting	Simulation Response
`none`	Simulink takes no action.
`warning`	Simulink displays a warning that the **Minimize algebraic loop occurrences** parameter has no effect.
`error`	Simulink terminates the simulation and displays an error that the **Minimize algebraic loop occurrences** parameter has no effect.

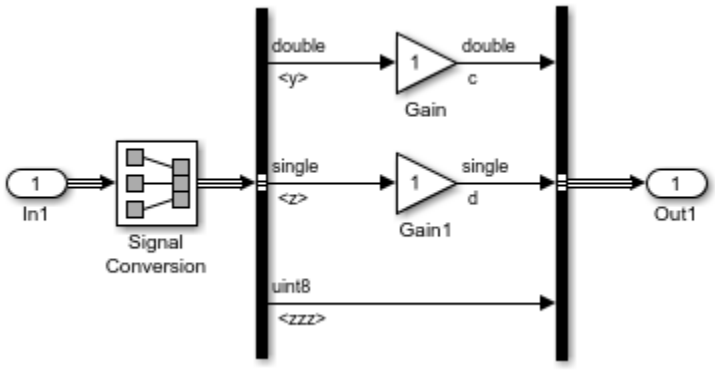
Block Reduction and Artificial Algebraic Loops

When you enable the **Block reduction** optimization in Model Configuration Parameters, Simulink collapses certain groups of blocks into a single, more efficient block, or removes them entirely. Enabling block reduction results in faster execution during model simulation and in generating code.

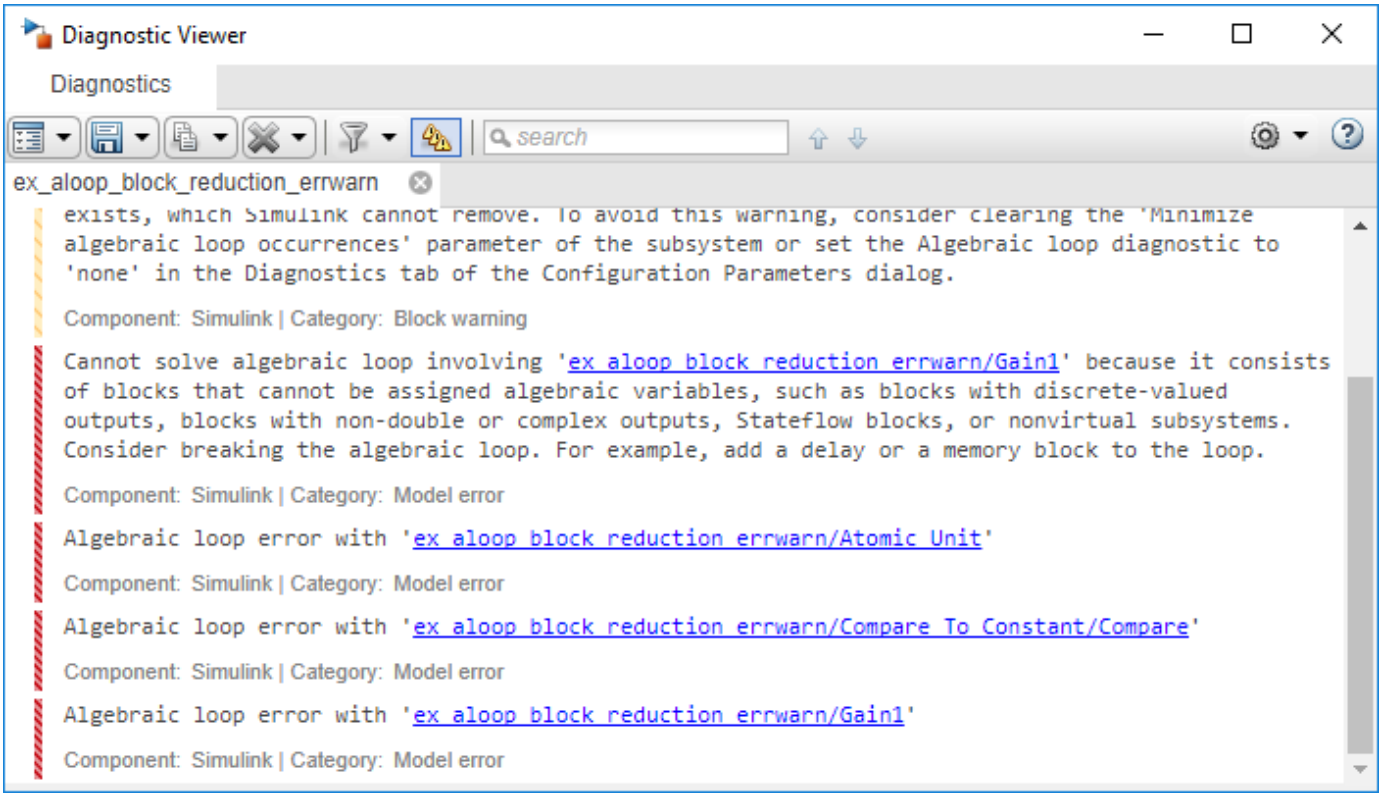
Enabling block reduction can also help Simulink solve artificial algebraic loops.

Consider the following example model.

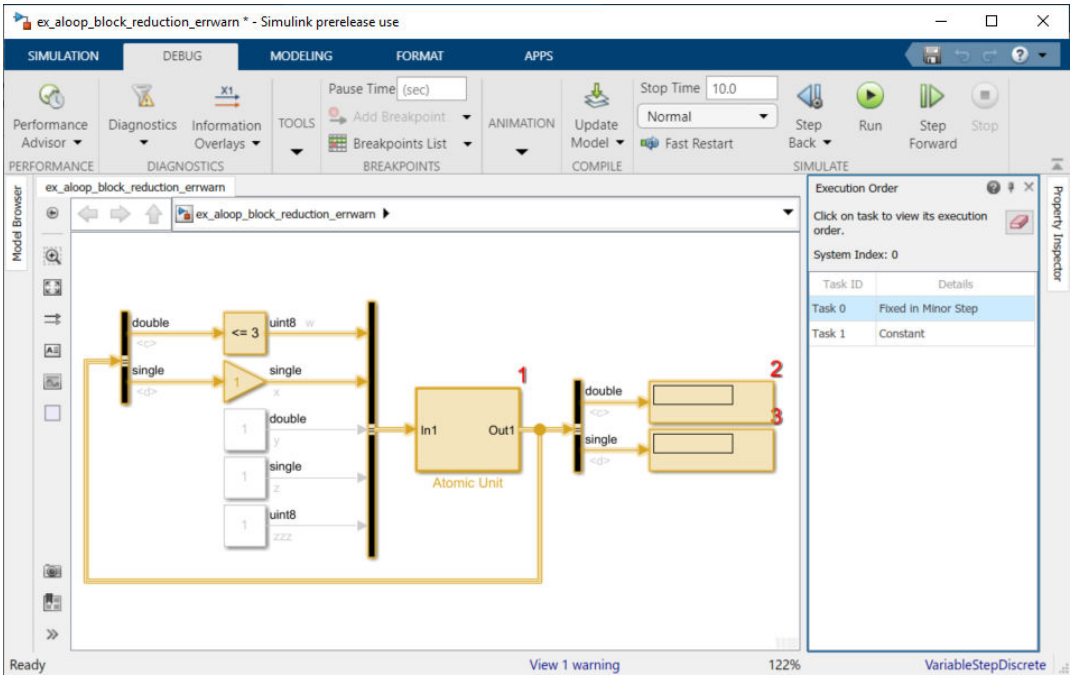




Initially, block reduction is turned off. When you simulate this model, the Atomic Unit subsystem and Gain and Compare to Constant blocks are part of an algebraic loop that Simulink cannot solve.



If you enable block reduction and execution order, and simulate the model again, Simulink does not display the execution order for blocks that have been reduced. You can now quickly see which blocks have been reduced.



The Compare to Constant and Gain blocks have been eliminated from the model, so they no longer generate an algebraic loop error. The Atomic Unit subsystem generates a warning:

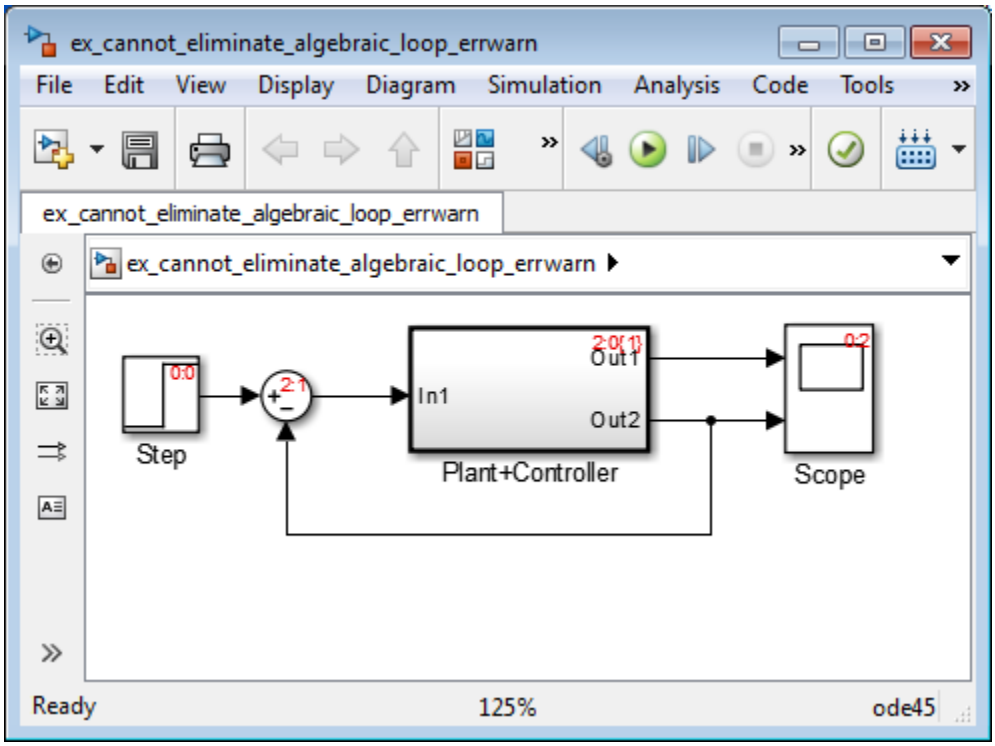
```
Warning: If the inport 'ex_aloop_block_reduction_errwarn/
Atomic Unit/In1' of subsystem 'ex_aloop_block_reduction_errwarn/
Atomic Unit' involves direct feedback, then an algebraic loop
exists, which Simulink cannot remove. Consider clearing the
'Minimize algebraic loop occurrences' parameter to avoid this
warning.
```

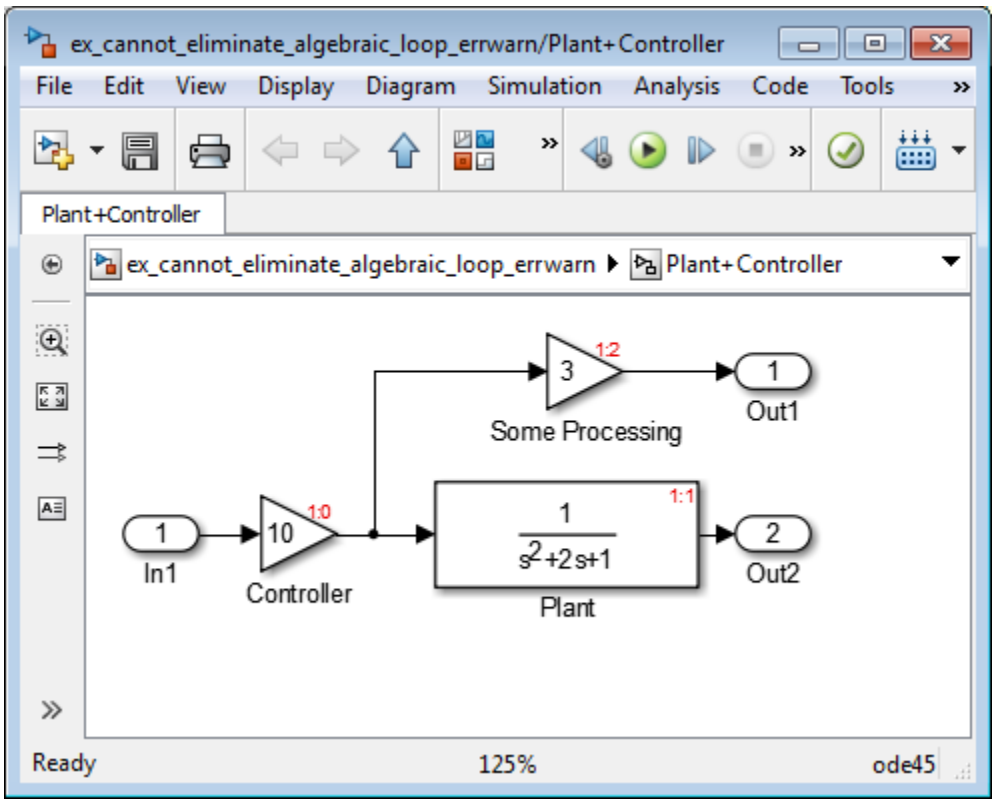
Tip Use Bus Selector blocks to pass only the required signals into atomic subsystems.

When Simulink Cannot Eliminate Artificial Algebraic Loops

Setting the **Minimize algebraic loop occurrences** parameter does not always work. Simulink cannot change the `DirectFeedthrough` property of an Inport block for an atomic subsystem if the Inport block is connected to an Outport block only through direct-feedthrough blocks.

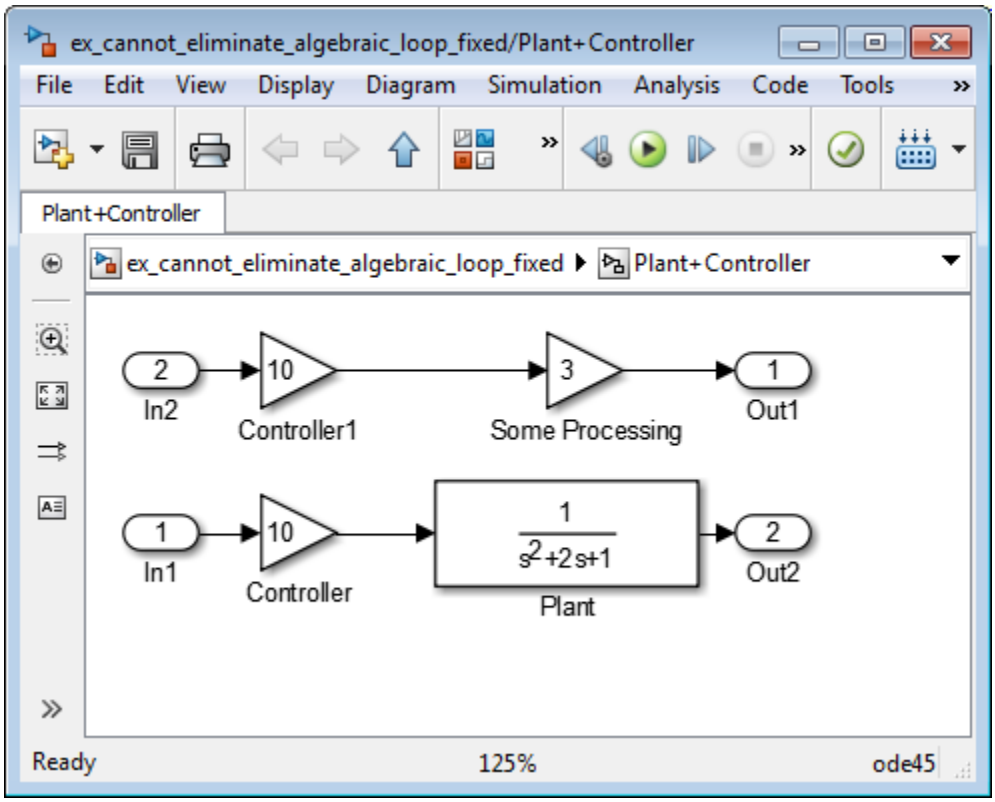
In this model, the subsystem Plant+Controller causes an algebraic loop, but it has an extra Gain block and an extra output.


ex_cannot_eliminate_algebraic_loop_errwarn
File Edit View Display Diagram Simulation Analysis Code Tools
ex_cannot_eliminate_algebraic_loop_errwarn
ex_cannot_eliminate_algebraic_loop_errwarn
0:0
Step
2:1
In1
2:0(1)
Out1
Out2
Plant+Controller
0:2
Scope
Ready
125%
ode45


ex_cannot_eliminate_algebraic_loop_errwarn/Plant+Controller
File Edit View Display Diagram Simulation Analysis Code Tools
Plant+Controller
ex_cannot_eliminate_algebraic_loop_errwarn
Plant+Controller
1:2
3
Some Processing
1
Out1
1
In1
1:0
10
Controller
1:1
1
s2+2s+1
Plant
2
Out2
Ready
125%
ode45

Simulink cannot move the `mdlOutputs` method of the Controller block to the `mdlDerivative` method of an Atomic unit 1 because the output of the atomic subsystem depends on the output of the Controller block. You cannot make the subsystem non-direct-feedthrough.

You can modify this model to eliminate the artificial algebraic loop by redefining the atomic subsystem by adding additional Inport and Gain blocks, as you can see in the model here. Doing so makes In1 non-direct-feedthrough and In2 direct feedthrough, thus breaking the algebraic loop.



See Also

More About

- “Algebraic Loop Concepts” on page 3-27
- “Identify Algebraic Loops in Your Model” on page 3-33
- “Modeling Considerations with Algebraic Loops” on page 3-52

Modeling Considerations with Algebraic Loops

Managing Large Models with Artificial Algebraic Loops

Adopt these design techniques for large models with algebraic loops:

- Avoid creating loops that contain discontinuities or nondouble data types. The Simulink algebraic loop solver is gradient-based and must solve algebraic constraints to high precision.
- Develop a scheme for clearly identifying atomic subsystems as direct feedthrough or not direct feedthrough. Use a visual scheme such as coloring the blocks or defining a block-naming convention.
- If you plan to generate code for your model, enable the **Minimize algebraic loop occurrences** parameter for all atomic subsystems. When possible, make sure that the input ports for the atomic subsystems are connected directly to non-direct-feedthrough blocks.
- Avoid combining non-direct-feedthrough and direct-feedthrough paths using the Bus Creator or Mux blocks. Simulink may not be able to eliminate any resulting artificial algebraic loops. Instead, consider clustering the non-direct-feedthrough and direct-feedthrough objects in separate subsystems.

  Use Bus Selector blocks to pass only the required signals into atomic subsystems.

Model Blocks and Direct Feedthrough

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 can give unexpected results. See:

- “Highlight Algebraic Loops in the Model” on page 3-34 for information about seeing algebraic loops graphically.
- “Display Algebraic Loop Information” on page 34-25 for information about tracing algebraic loops in the debugger.
- The “Model Configuration Parameters: Diagnostics” pane “Algebraic loop” option for information on detecting algebraic loops automatically.

Direct Model Block Feedthrough Caused by Referenced Model Structure

A Model block can 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” on page 3-36
- Manually insert the number of Unit Delay blocks needed to break the algebraic loop.

Direct Model Block Feedthrough Caused by Model Configuration

Generic Real Time (`grt`) and Embedded Real Time (`ert`) based targets provide the **Single output/ update function** option on the **Configuration Parameters** dialog. This option controls whether

generated code has separate output and update functions, or a combined output/update function. See:

- “Configure C Code Generation for Model Entry-Point Functions” (Simulink Coder) for information about separate and combined output and update functions.
- “Single output/update function” (Simulink Coder) 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” on page 3-36
- Manually insert one or more Unit Delay blocks as needed to break the algebraic loop.

Changing Block Priorities When Using Algebraic Loop Solver

During the updating phase of simulation, Simulink determines the simulation execution order of block methods. This block invocation ordering is the execution order.

If you assign priorities to nonvirtual blocks to indicate to Simulink their execution order relative to other blocks, the algebraic loop solver does not honor these priorities when attempting to solve any algebraic loops.

See Also

More About

- “Algebraic Loop Concepts” on page 3-27
- “Identify Algebraic Loops in Your Model” on page 3-33
- “Remove Algebraic Loops” on page 3-36

Artificial Algebraic Loops

An *artificial algebraic loop* occurs when an atomic subsystem or Model block causes Simulink to detect an algebraic loop, even though the contents of the subsystem do not contain a direct feedthrough from the input to the output. When you create an atomic subsystem, all Inport blocks are direct feedthrough, resulting in an algebraic loop.

Start with the included model, which represents a simple proportional control of the plant described by

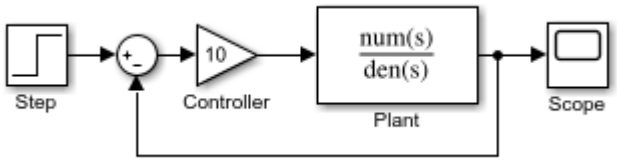
$$G(s) = \frac{1}{s^2 + 2s + 1}$$

which can be rewritten in state-space form as

$$\dot{x} = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

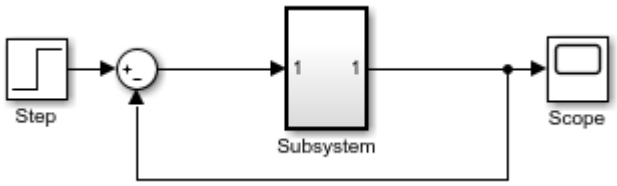
$$y = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

The system has neither algebraic variables nor direct feedthrough and does not contain an algebraic loop.



Modify the model as described in the following steps:

1. Enclose the Controller and Plant blocks in a subsystem.
2. In the subsystem dialog box, select **Treat as atomic unit** to make the subsystem atomic.
3. In the **Diagnostics** pane of the Model Configuration Parameters, set the **Algebraic Loop** parameter to `error`.



When simulating this model, an algebraic loop occurs because the subsystem is direct feedthrough, even though the path within the atomic subsystem is not direct feedthrough. Simulation stops with an algebraic loop error.

Modeling Dynamic Systems

Creating a Model

Create a Template from a Model

Create a Simulink template from a model to reuse or share the settings and contents of the model without copying the model each time. Create templates only from models that do not have external file dependencies, for example, model references, data dictionary, scripts, S-functions, or other file dependencies. If you want to include other dependent files, use a project template instead. See “Using Templates to Create Standard Project Settings” on page 16-32.

1. In the model, on the **Simulation** tab, select **Save** > **Template**.
2. In the Export *`modelname`* to Model Template dialog box, edit the template title, select or create a group, and enter a description of the template.

   The Simulink start page displays the title and description you specify and stores the template under the group you select, for example, *`My Templates`*.
3. In the **Template file** box, select a file name and location for the template SLTX file.

   Tip Save the template on the MATLAB path to make it visible on the Simulink start page. If you save to a location that is not on the path, the new template is visible on the start page only in the current MATLAB session. Saving the template does not add the destination folder to the path.
4. (Optional) To specify a thumbnail image for the template, click **Change**, then select an image file.
5. Click **Export**.

Edit a Template

To edit a model file template, open the template from the **Current Folder** or from the Simulink start page. Make the desired changes.

1. Open the template for editing.
   - From the **Current Folder**, navigate to the template (`*.sltx`). Right-click the template and select **Open Template for Editing**.
   - From the Simulink start page, point to the template and click the arrow to see the template information. On the **Create** button, click the arrow and select **Edit Template**.
2. Make the changes you want.
3. You can also edit the template properties.
   - For a model template, on the **Simulation** tab, click **Template Properties**.
   - For a subsystem template, on the **Subsystem** tab, click **Template Properties**.
   - For a library template, on the **Library** tab, click **Template Properties**.
4. Save the template for your changes to take effect.

See Also

`Simulink.exportToTemplate`

Related Examples

- “Using Templates to Create Standard Project Settings” on page 16-32

Describe Models Using Notes and Annotations

You can describe your model with notes and annotations to help others to understand it. You can add notes to any system in the model hierarchy by entering text, showing website content, or inheriting note content from the parent system. For each system, you can also choose not to show any notes.

Annotations are visual elements that you can use to add descriptive notes and callouts to your model. You can also add annotations that perform an action when you click them.

Text annotations can contain any combination of:

- Text
- Images
- Equations using LaTeX and MathML commands
- Hyperlinks that open a website or perform MATLAB functions

Also, you can create an image-only annotation.

Manage Notes

To get started, in the **Modeling** tab, under **Design**, click **Notes**. If the model has notes associated with it, they appear in a pane to the right of the model. As you navigate the hierarchy, notes for each system appear. If the model does not have notes, you can add them.

Notes are stored in a file with the extension `.mldatx`. If you want your model to have notes, first create the notes file. See “Add and Edit Notes” on page 4-3. After you create the file, the notes you add are saved automatically.

You can have multiple notes files associated with the same model, for example, for users with different roles. The person using the model can then select among notes files that match your model. Notes files contain model name and version information to ensure the notes file and model match.

The `.mldatx` file is saved separate from the model. If you move your model to a different folder, the `.mldatx` file does not also move, but the association remains if the file is on the MATLAB path.

Read and Edit Modes

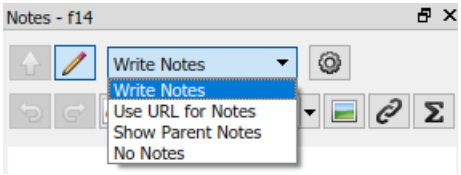
Use the **Notes** pane to edit and read notes. Use the **Read/Edit Notes** toggle  to switch between modes. When you click **Edit Notes**, the editing features are enabled. When you click **Read Notes**, the editing features are not available. Instead, the content displays in read-only format. As you navigate the model, the **Notes** pane updates with the content for the current system.

Add and Edit Notes

1. In the model, on the **Modeling** tab, under **Design**, click **Notes**. The notes interface appears to the right of the model and includes instructions to get started.
2. Click the **Create a notes file** button.
3. Enter a name for the notes file, or use the default name, and click **Save**.
4. The **Notes** pane is in editing mode. You can add notes in these ways:

    - Enter text in the text editor.

- From the menu, select **Use URL for Notes** and enter a URL whose content you want to show as notes for the current system.
- Navigate to a different system in the model and enter text, use a URL, or select **Show Parent Notes**.
- Select **No Notes**.



5 When you have finished adding and editing, click the **Read Notes** toggle button  to put the notes in reading mode.

To edit notes from reading mode, navigate to the system whose content you want to edit and click the **Edit Notes** toggle button  .

Manage Annotations

To create a text annotation, use one of these options:

- Double-click the canvas where you want to create the annotation and select **Create Annotation** from the menu.
- Click the annotation box  on the Simulink Editor palette and then click the canvas.
- Drag the annotation box  on the Simulink Editor palette to the canvas.
- Drag text from another application to the canvas.
- Paste text from the clipboard. Right-click the canvas and select **Paste**.

After you add the text annotation, you can:

- Apply formatting changes to text or insert an image, table, or equation using the formatting toolbar.
- Apply additional formatting, using the **Paragraph** menu on the context menu. For example, you can create bullet and numbered lists from this menu.
- Add hyperlinks using the context menu. You can use hyperlinks to open a website or make an annotation interactive using MATLAB commands.
- Apply properties using the Property Inspector. To view the Property Inspector, in the **Modeling** tab, under **Design**, click **Property Inspector**.

Resize an Annotation

An annotation resizes as you enter content. You can also resize an annotation by dragging the corners. For example, you can hold **Shift** as you drag to resize proportionally.

After you resize an annotation, the annotation stays that size until you resize it again, regardless of the content size. To revert to the original height or width of the annotation, in the Property Inspector, under **Appearance**, clear the **Fixed height** or **Fixed width** check box.

Make an Annotation Interactive

To make the annotation interactive, use a hyperlink on any content of a text annotation.

1. In the annotation, select the content that you want to make interactive. To make the entire annotation interactive, select all the content.
2. Right-click and select **Hyperlink** from the context menu.
3. In the Hyperlink dialog box, either:
   - Select **URL Address** as the target and enter the web address in the **Code** box.
   - Select **MATLAB Code** as the target and enter MATLAB functions in the **Code** box.
4. Click **OK**.

For an alternative approach, see “Annotation Callback Functions” on page 4-9.

Add an Image Annotation

When you want to resize or move an image independently from text, create an image annotation. For example, you can size and position your company logo at a particular location in the model. You can also invoke MATLAB functions with a click on the image. To add an annotation that contains an image, drag the image from your file system to the canvas.

Tip To include an image such as a logo in every new model, add the image to your default template. See “Create a Template from a Model” on page 4-2.

Alternatively, you can drag an **Image** box from the palette onto the canvas. Then you can either:

- Double-click the image box and browse to an image.
- Paste an image from the clipboard. Right-click the image box and select **Paste Image**.
- Drag an image from your local file system to the Simulink Editor canvas.

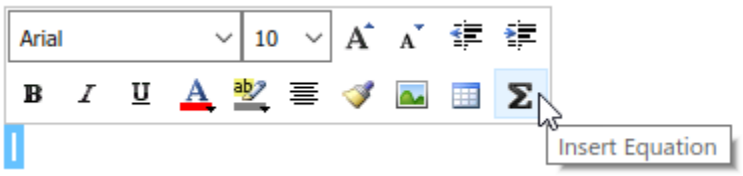
Tip If you resize an image, you can reset it to its original size. Right-click the image and select **Format > Restore Size**.

To associate an action with an image:

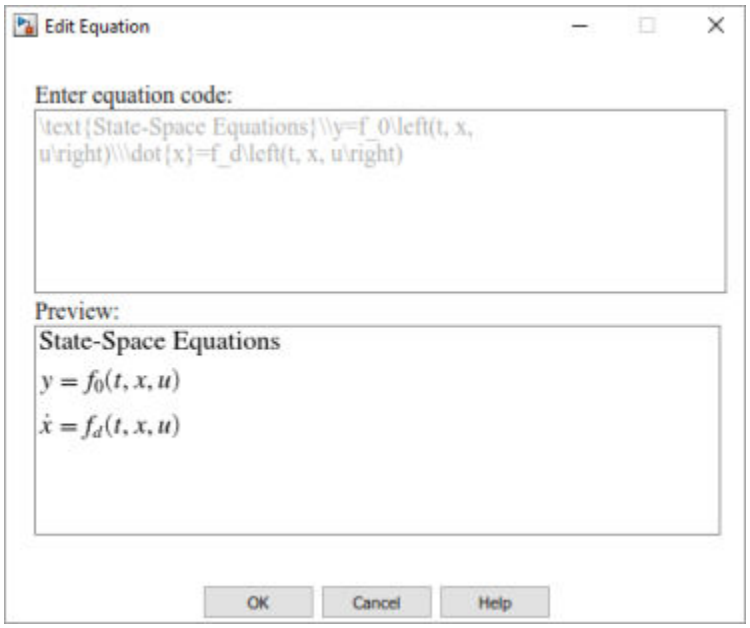
1. Select the image.
2. In the Property Inspector, under **ClickFcn**, add the MATLAB functions that you want to invoke with a click on the image.

Add Equations in an Annotation

You can add equations to your annotation by clicking the **Insert Equation** button in the annotation formatting toolbar.



In the Edit Equation dialog box, enter LaTeX or MathML code to generate equations. For LaTeX commands, see “Insert LaTeX Equation”.



To edit equation code, double-click the equation in the annotation. Similar to text in annotations, you can format equations in annotations by using the formatting toolbar. You can change the font color, text highlight color, font size, and location of equations in annotations.

You can add TeX formatting commands to your annotation for mathematical and other symbols and Greek letters.

1. Add supported TeX commands to your annotation. For example, add this text:

   ```
   \sigma \kappa \mu
   ```

2. With the annotation selected, or with the text cursor in the annotation, in the Property Inspector, under **Appearance**, select **Enable TeX commands**.

   When you click outside the annotation, the TeX commands appear as symbols in the annotation.

The table shows the TeX characters supported in Simulink annotations.

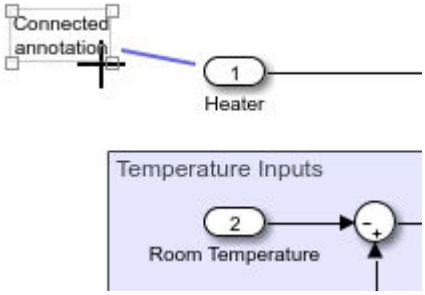
Supported TeX Characters		
alpha	forall	supseteq
beta	exists	supset
gamma	ast	subseteq
delta	cong	subset
epsilon	sim	int
zeta	leq	in
eta	infty	o
theta	clubsuit	copyright
vartheta	diamondsuit	0
iota	heartsuit	ldots
kappa	spadesuit	varpi
lambda	leftarrow	times
mu	uparrow	cdot
nu	rightarrow	vee
xi	downarrow	wedge
pi	circ	perp
rho	pm	mid
sigma	geq	Leftarrow
varsigma	propto	Rightarrow
tau	partial	Uparrow
upsilon	bullet	Downarrow
phi	div	prime
chi	neq	nabla
psi	equiv	surd
omega	approx	angle
Gamma	aleph	neg
Delta	Im	lceil
Theta	Re	rceil

Supported TeX Characters		
Lambda	otimes	lfloor
Xi	oplus	rfloor
Pi	oslash	langle
Sigma	cap	rangle
Upsilon	cup	
Phi		
Psi		
Omega		

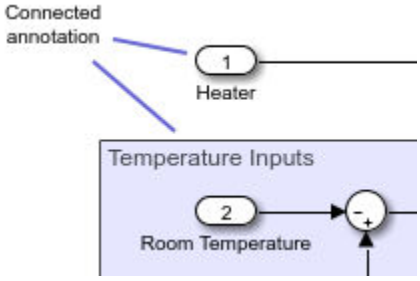
Associate Annotations with Blocks and Areas

You can add a line between an annotation and a block or area in a model. These annotation connectors attach dynamically at both ends, so that they move and resize as necessary to maintain the connection.

1 Place the cursor over the annotation outline where you want the line to start.



2 When the cursor is a crosshair, drag the line to the block or area where you want the line to end.



Tip To specify the color or width of an annotation connector, right-click it and use the **Format** menu.

Hide an Annotation

By default, all annotations appear in the model. To hide an annotation, first configure it for hiding by converting it to markup. Then, in the **Format** tab, click **Show Markup**.

You can configure an annotation so that you can hide or display it.

1. Right-click the annotation.
2. From the context menu, select **Convert to Markup**.

A markup annotation has a light-blue background, regardless of the background color you set. If you change a markup annotation back to a regular annotation, the annotation returns to the background color you set.

To change a markup annotation to a regular annotation (one that you cannot hide), from the annotation context menu, select **Convert to Annotation**.

To hide all markup annotations, in the **Format** tab, click **Show Markup**.

To display hidden markup annotations, in the **Format** tab, click **Show Markup**.

Note In a model reference hierarchy, **Show Markup** and **Hide Markup** apply only to the current model reference level.

Annotation Callback Functions

You can associate these callback functions with annotations.

Click Function

You can make an annotation interactive using a link. Alternatively, you can make an annotation interactive by adding a click function callback. A click function is a MATLAB function that Simulink invokes when you click an annotation.

You can add a click function callback programmatically or interactively. To create a click function programmatically, see `Simulink.Annotation`. To create one interactively, see “Associate a Click Function with an Annotation” on page 4-9.

The text for annotations associated with a click function appears in blue.

Load Function

Simulink invokes a load function when you load the model that contains the associated annotation. To associate a load function with an annotation, set the `LoadFcn` property of the annotation to the desired function (see `Simulink.Annotation`).

Delete Function

A delete function is invoked before you delete an annotation. To associate a delete function with an annotation, set the `DeleteFcn` property of the annotation to the desired function (see `Simulink.Annotation`).

Associate a Click Function with an Annotation

You can interactively associate a click function with an annotation.

1. Add an annotation.
2. Open the Annotation Properties dialog box. Right-click the annotation and select **Properties**.
3. Open the **ClickFcn** tab. In the text box under **ClickFcn**, enter the MATLAB code that defines the click function, and click **OK**.

Tip Alternatively, you can use the annotation text as the click function. Then, in the Annotation Properties dialog box, select the **Use annotation text as click callback** check box.

Select and Edit Click-Function Annotations

If you associate an annotation with a click function, clicking invokes the function rather than selecting the annotation. To select it instead, drag a selection box around it. To edit it, right-click it and select **Edit Text** or **Properties**.

See Also

`Simulink.Annotation`

Related Examples

- “Create and Edit Annotations Programmatically” on page 4-11

Create and Edit Annotations Programmatically

Annotations are visual elements that you can use to add descriptive notes and callouts to your model. In addition to text-only annotations, you can create annotations that:

- Open websites
- Perform MATLAB commands
- Display images
- Visually differentiate areas of block diagrams

The following examples show how to programmatically create, edit, and delete annotations.

Create Annotation Programmatically

Programmatically create, modify, and view an annotation.

Open a new model.

```
open_system(new_system)
```

Create an annotation with default properties using the `Simulink.Annotation` function.

```
a = Simulink.Annotation(gcs,'This is an annotation.');
```

After creating the annotation, use dot notation to set property values. For example, apply an 18-point font and light blue background to the annotation.

```
a.FontSize = 18;
a.BackgroundColor = 'lightBlue';
```

To view and briefly highlight the new annotation, use the `view` function.

```
view(a)
```

Programmatically Find and Modify Existing Annotations

Programmatically find and modify the properties of an annotation.

Open the `vdp` model.

```
vdp
```

To find the annotations in the model, use the `find_system` function.

```
h = find_system(gcs,'FindAll','on','Type','annotation');
```

To identify the annotations, query the text inside the annotations by using the `get_param` function.

```
get_param(h,'PlainText')
```

```
ans = 2x1 cell
    {'Copyright 2004-2020 The MathWorks, Inc.'}
    {'van der Pol Equation'                    }
```

Suppose you want to apply a light blue background color to the `'van der Pol Equation'` annotation.

Get the `Simulink.Annotation` object by specifying the corresponding index of the array.

```
a = get_param(h(2),'Object');
```

Use dot notation to set the value of the `BackgroundColor` property.

```
a.BackgroundColor = 'lightBlue';
```

Delete Annotation

Programmatically delete an annotation.

Open the `vdp` model.

```
vdp
```

To get the handles for the annotations in the model, use the `find_system` function.

```
h = find_system(gcs,'FindAll','on','Type','annotation');
```

To identify the annotations, query the text inside the annotations.

```
get_param(h,'PlainText')
```

```
ans = 2x1 cell
    {'Copyright 2004-2020 The MathWorks, Inc.'}
    {'van der Pol Equation'                    }
```

To delete the title of the model (`'van der Pol Equation'`), get the `Simulink.Annotation` object that corresponds to the second handle.

```
a = get_param(h(2),'Object');
```

Delete the annotation from the model.

```
delete(a)
```

Create Annotations That Contain Hyperlinks

For rich-text annotations, you can use HTML formatting to add a hyperlink to text within the annotation.

Open a new model.

```
open_system(new_system)
```

Create two annotations, moving one of the annotations so that it does not overlap the other.

```
a1 = Simulink.Annotation(gcs,'This is an annotation.');
a2 = Simulink.Annotation(gcs,'This is another annotation.');
a2.Position = [0 20 28 34];
```

To create a hyperlink in the annotation, set `Interpreter` to `'rich'` and define the hyperlink in the `Text` property.

```
a1.Interpreter = 'rich';
a1.Text = 'Go to <a href="www.mathworks.com">www.mathworks.com</a>.';
```

You can also embed MATLAB functions in the hyperlink.

```
a2.Interpreter = 'rich';
a2.Text = '<a href="matlab:magic(4)">Generate magic square</a>.';
```

For more information, see “Create Hyperlinks that Run Functions”.

Add Image to Model

Add an image to your model, such as a logo, by creating an image-only annotation.

Open a new model and create an annotation in it.

```
open_system(new_system)
a = Simulink.Annotation(gcs,'This is an annotation.');
```

Change the annotation to display only the specified image.

```
img = fullfile(matlabroot,'toolbox','matlab','imagesci','peppers.png');
setImage(a,img)
```

Create Area Programmatically

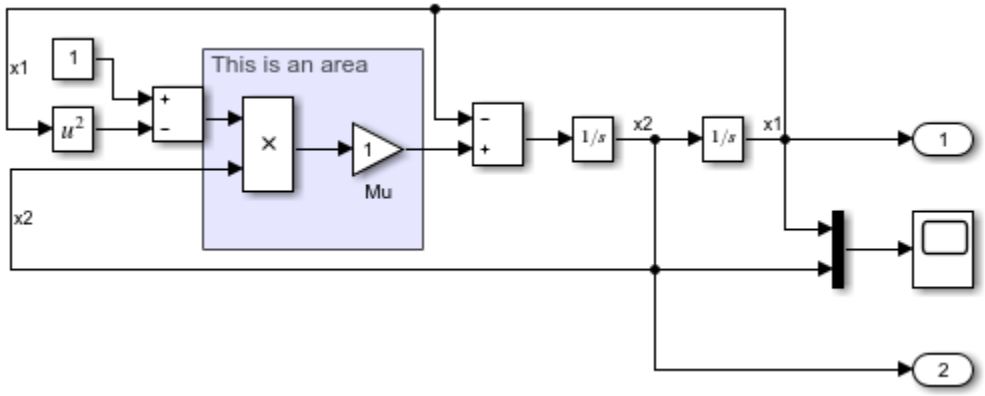
Create an area annotation in a model.

Open the `vdp` model.

```
open_system('vdp')
```

Create an area that includes some of the blocks in the model.

```
add_block('built-in/Area','vdp/This is an area','Position',[120,100,230,200])
```



Create and Hide Markup Annotation

To create annotations that can be easily hidden, create markup annotations.

Open a new model.

```
open_system(new_system)
```

Create two annotations, and move the second annotation so that it does not overlap the first annotation.

```
a1 = Simulink.Annotation(gcs,'This is a model annotation.');
a2 = Simulink.Annotation(gcs,'This is a markup annotation.');
a2.Position = [0 20 28 34];
```

By default, you create model annotations, which appear in the model.

Change the second annotation to a markup annotation.

```
a2.MarkupType = 'markup';
```

Configure the current model to hide markup annotations.

```
set_param(gcs,'ShowMarkup','off');
```

Both annotations remain, despite the markup annotation being hidden.

```
ah = find_system(gcs,'FindAll','on','Type','annotation');
at = get_param(ah,'Text')
```

```
at = 2x1 cell
    {'This is a markup annotation.'}
    {'This is a model annotation.' }
```

Find Annotation Executing Callback Function

If an annotation invoked a currently executing callback function, use the `getCallbackAnnotation` to determine which annotation invoked it. The function returns the corresponding `Annotation` object. This function is also useful if you write a callback function in a separate MATLAB file that contains multiple callback calls.

See Also

`Simulink.Annotation` | `add_block` | `delete (Annotation)` | `setImage (Annotation)` | `view (Annotation)`

Related Examples

- “Describe Models Using Notes and Annotations” on page 4-3

Create Subsystems

In this section...

As a model increases in size and complexity, you can simplify it by grouping blocks into subsystems. A subsystem is a set of blocks that you group into a single Subsystem block.

Using subsystems:

- Establishes a hierarchical block diagram where a Subsystem block is on one layer and the blocks that make up the subsystem are on another.
- Keeps functionally related blocks together.
- Helps reduce the number of blocks displayed in your model window.
- Establishes an interface with inputs and outputs.

When you make a copy of a subsystem, that copy is independent of the source subsystem. To reuse the contents of a subsystem across a model or across models, consider referenced subsystems, referenced models, or subsystems linked to a block in a custom library. For more information, see “Choose Among Types of Model Components” on page 22-4.

Types of Subsystems

A subsystem can be virtual or nonvirtual. A virtual subsystem provides graphical hierarchy in a model and does not impact the execution of a model. A nonvirtual subsystem executes as a single unit within a model.

Tip For controllers and other standalone components, define a hard boundary around the related blocks by using a nonvirtual subsystem or referenced model. Defining a hard boundary upfront avoids costly refactoring when you want to generate code for the component.

Simulink classifies nonvirtual subsystems into these types:

Type of Subsystem	Description
Atomic Subsystem	Subsystem that executes as a single unit.
Enabled Subsystem	Subsystem whose execution is enabled by external input.
Triggered Subsystem	Subsystem whose execution is triggered by external input.
Function-Call Subsystem	Subsystem whose execution is controlled by an external function-call input.

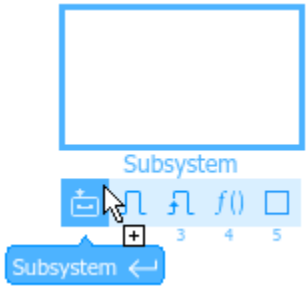
Type of Subsystem	Description
Enabled and Triggered Subsystem	Subsystem whose execution is enabled and triggered by external inputs.
Resettable Subsystem	Subsystem whose block states reset with an external trigger.
If Action Subsystem	Subsystem whose execution is enabled by an If block.
Switch Case Action Subsystem	Subsystem whose execution is controlled by a Switch Case block.
While Iterator Subsystem	Subsystem that repeats execution during a simulation time step while a logical condition is true.
For Iterator Subsystem	Subsystem that repeats execution during a simulation time step for a specified number of iterations.
For Each Subsystem	Subsystem that repeats execution on each element or subarray of input signal and concatenates results.

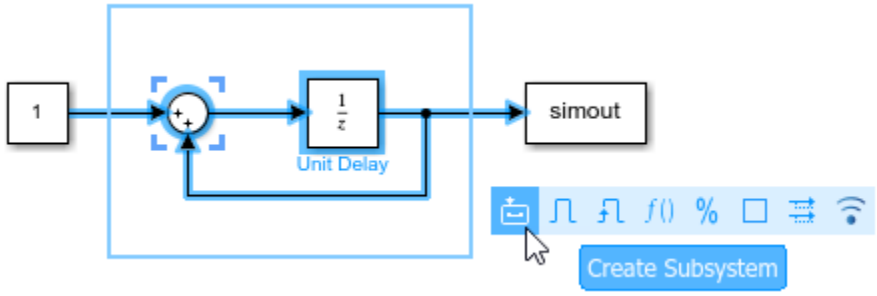
Create Subsystems

To create a subsystem, you can:

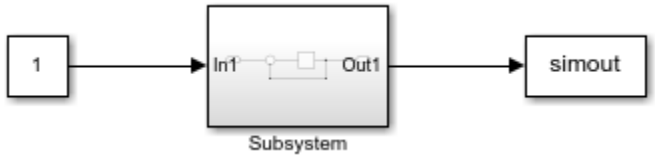
- In the Simulink Editor, double-click and start typing the subsystem type, then select the corresponding block from the menu.
- In the Simulink Editor, drag a selection box to outline the subsystem that you want to create, then select the subsystem type.
- Drag a Subsystem block from the Library Browser.
- Copy and paste a Subsystem block from a model.

When you create a subsystem from a selection box, the selection can be empty or can contain multiple blocks in one area of the model.





From the action bar, select the type of subsystem that you want to create.



When the selection contains blocks that correspond to input and output ports, the new subsystem includes copies of those blocks. The new subsystem does not contain copies of blocks that correspond to control ports.

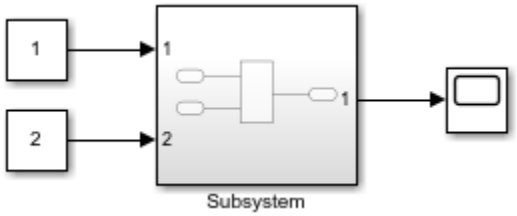
You can change the type of subsystem after creation.

- To make a subsystem execute as a unit, click the Subsystem block. On the **Subsystem** tab, select **Is Atomic Subsystem**.
- To make a subsystem execute conditionally, add a block that corresponds to a control port.
- To make a subsystem execute unconditionally, remove blocks that correspond to control ports.

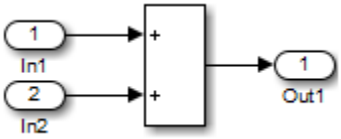
Add Ports to Subsystems

The ports on a Subsystem block correspond to blocks inside the subsystem.

For example, this Subsystem block has two input ports and one output port.

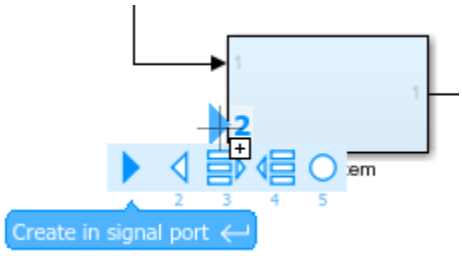


The subsystem contains two Inport blocks (`In1` and `In2`) and one Outport block (`Out1`) that correspond to the ports on the Subsystem block.

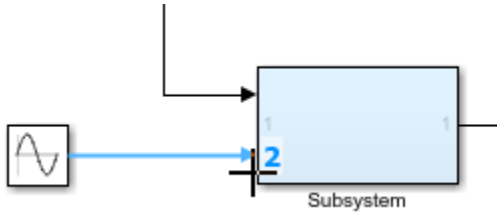


To add ports to a subsystem:

- Click the edge of the Subsystem block, then select the type of port to create.



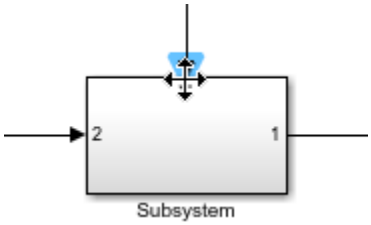
- Drag a line to the edge of the Subsystem block.



- Open the subsystem by double-clicking the Subsystem block, then add the corresponding blocks to the subsystem.

Type of Port	Corresponding Block
Signal port, input	Inport block
Signal port, output	Outport block
Bus port, input	In Bus Element block
Bus port, output	Out Bus Element block
Control port, enabled	Enable block
Control port, triggered	Trigger block
Control port, function-call	Trigger block with **Trigger type** set to `function-call`
Control port, reset	Reset block
Control port, action	Action Port block
Connection port	Connection Port block

To change the location of a port on a Subsystem block, drag the port to a new location on any side of the Subsystem block.



By default, Simulink labels the ports on a Subsystem block. To specify how Simulink labels the ports of a subsystem:

1. Select the Subsystem block.
2. On the **Format** tab of the Simulink Toolstrip, select one of the labeling options from the **Port Labels** menu. For more information, see **Show port labels**.

Configure Subsystems

You can change the name of the Subsystem block and modify the block the way that you do with any other block. For example, you can:

- Apply block masks to hide the subsystem content, making it appear as an atomic block with its own icon and parameter dialog box. For more information, see “Masking Fundamentals” on page 39-2.
- Use block callbacks to perform actions in response to subsystem modeling actions such as handling an error, deleting a block or line in a subsystem, or closing a subsystem. For more information on block properties, such as callbacks, see “Specify Block Properties” on page 36-4.

Restrict Subsystem Access

The **Read/Write permissions** parameter of a Subsystem block controls the level of access allowed for the subsystem.

Note Restricting read or write access does not prevent the access restrictions from being changed. To hide proprietary information, consider using a protected model. For more information, see “Protected Models for Model Reference”.

When a subsystem is stored in a custom library, you can use the **Read/Write permissions** parameter on the parent library block to control access for the linked instances of the block. As long as the library link remains intact, the restricted access can prevent people from viewing or modifying the contents of the subsystem while still allowing them to employ it in a model. For more information, see “Linked Blocks” on page 41-10.

See Also

`Simulink.BlockDiagram.copyContentsToSubsystem`

More About

- “Navigate Model Hierarchies” on page 4-20
- “Expand Subsystem Contents” on page 4-33
- “Conditionally Executed Subsystems Overview” on page 10-3

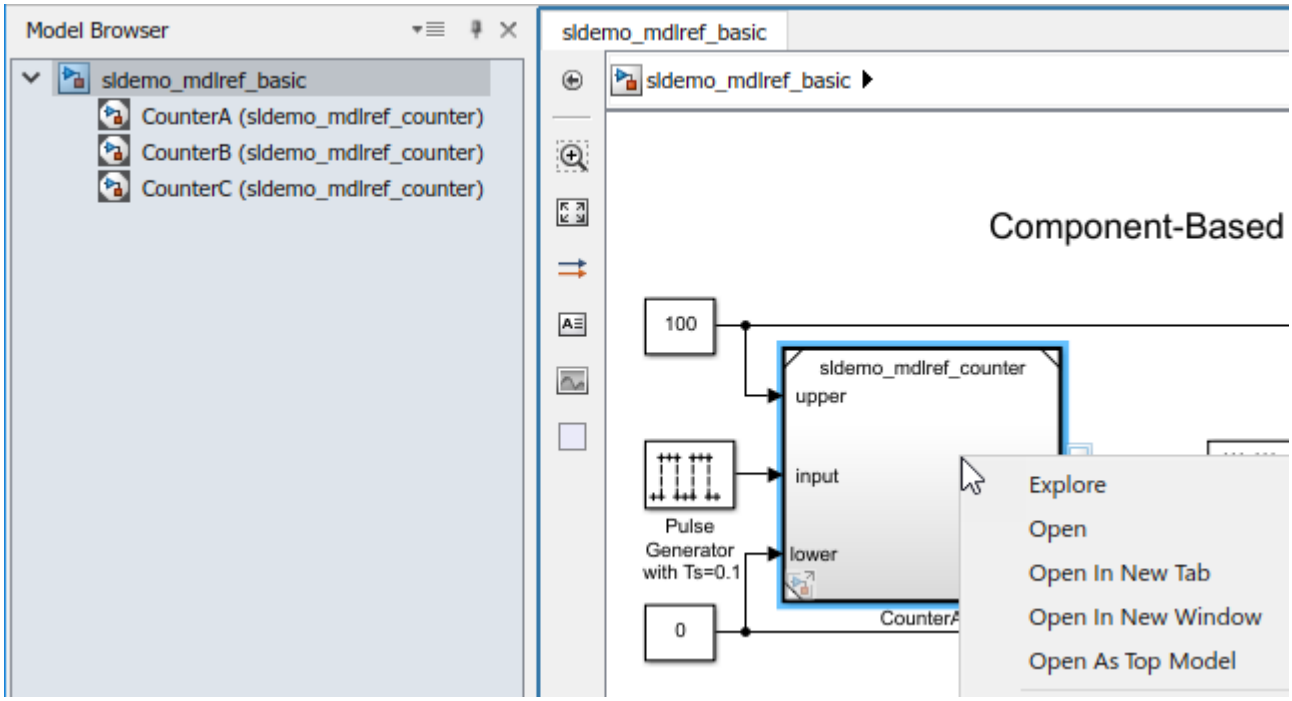
Navigate Model Hierarchies

Subsystems and model references allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy using the Model Browser or with Simulink Editor model navigation commands.

Open a Subsystem or Referenced Model

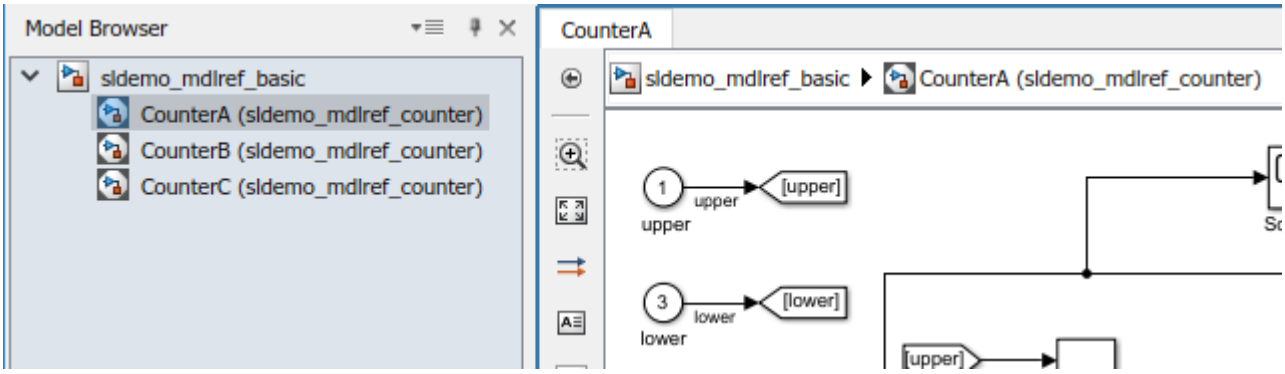
To open a model component using the Simulink Editor context menu:

1. In the Simulink Editor, right-click the Subsystem block, Model block, or canvas.

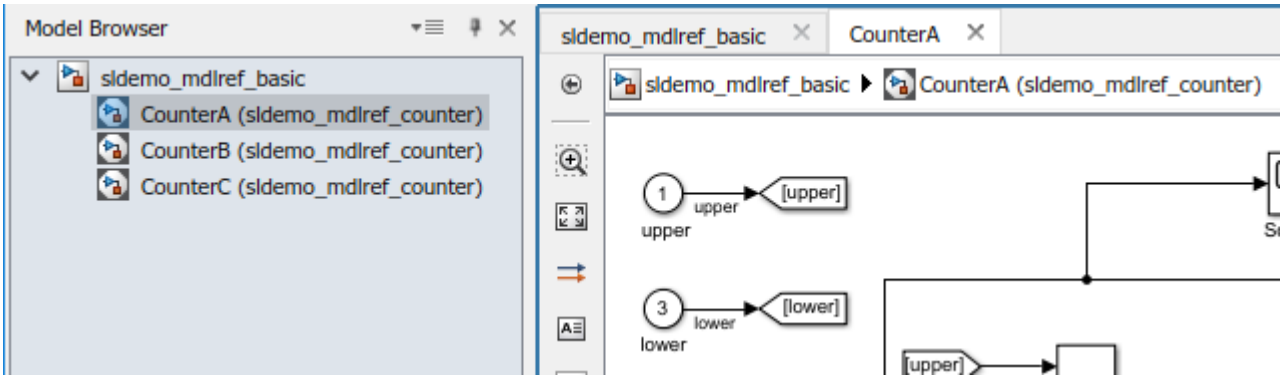


2. From the context menu, select one of these options:

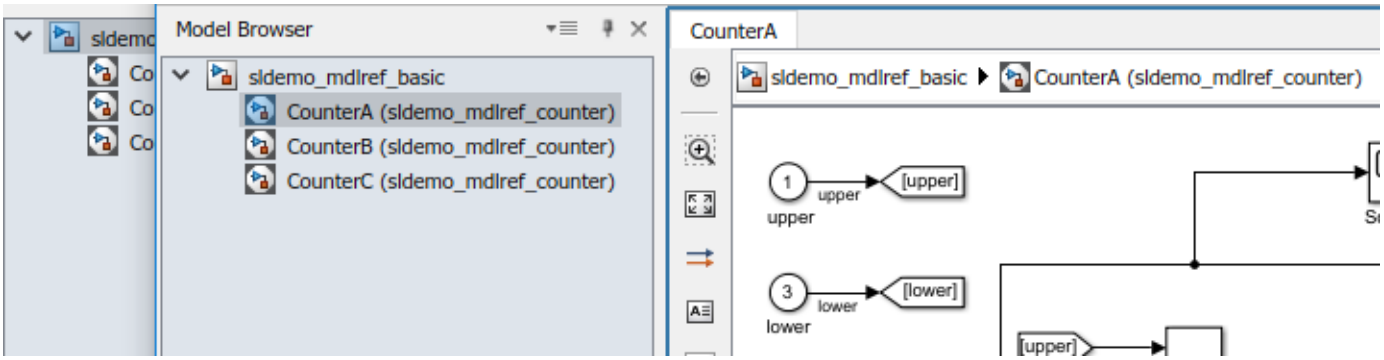
   - **Open** — Open the subsystem or referenced model in the same window and tab as used for the top model.



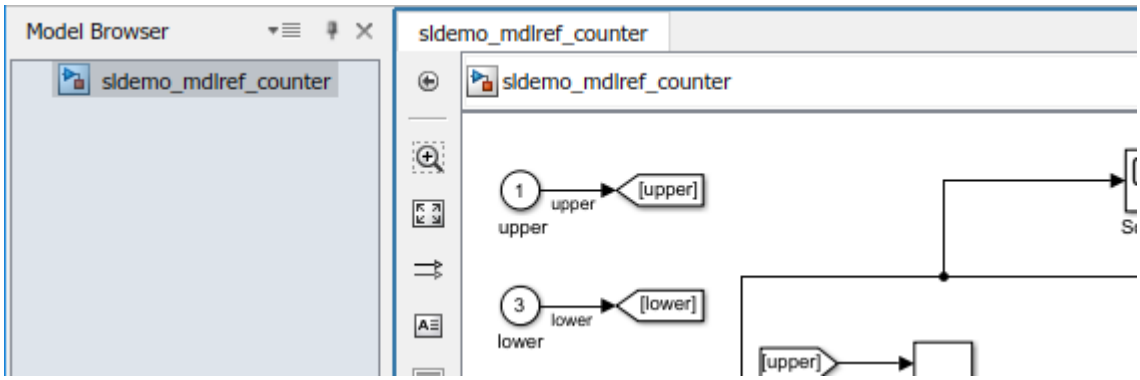
- **Open In New Tab** — Open the subsystem or referenced model in a new tab.



- **Open In New Window** — Open the subsystem or referenced model in a new Simulink Editor window.



- **Open As Top Model** — Open a referenced model as a top model in a new Simulink Editor window.



For any operation to open a subsystem or referenced model, you can use a keyboard shortcut to have it open in a new tab or window:

Where to Open the Subsystem or Referenced Model	Keyboard Shortcut
In a new tab	Hold the **Ctrl** key while opening the subsystem or referenced model.

Where to Open the Subsystem or Referenced Model	Keyboard Shortcut
In a new window	Hold the **Shift** key while opening the subsystem or referenced model.

Tip To navigate up and out of a subsystem or referenced model, use the **Navigate** arrows at the top left. The subsystem or referenced model you navigated from appears highlighted so that you can identify where you came from.

See Also

Related Examples

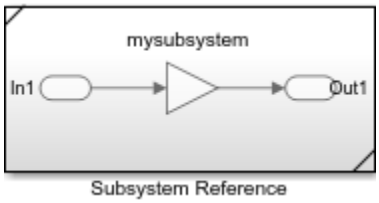
- “Exploring the Model Hierarchy”
- “Preview Content of Model Components” on page 1-33

Subsystem Reference

Subsystem reference allows you to save the contents of a subsystem in a separate SLX file and reference it using a Subsystem Reference block. You can create multiple instances referencing the same subsystem file. When you edit any instance of a referenced subsystem, the changes are saved in the separate SLX file in which the subsystem is stored and all the referenced instances of that file are synchronized.

When you save a subsystem to a separate file you can reuse it multiple times by using **Subsystem Reference** blocks referencing the same subsystem file.

You can identify a Subsystem Reference block by the triangles in the opposite corners of the block icon.



A referenced subsystem supports all the semantics of a regular subsystem. A referenced subsystem adapts itself to the context of the parent model and has identical execution behavior when compared to a nonreferenced subsystem.

Create a Subsystem Block Diagram

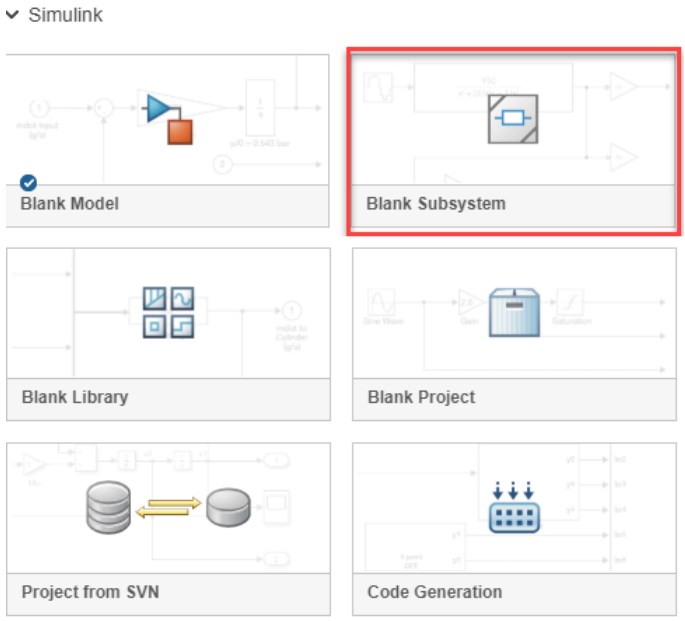
A subsystem file stores the content of a subsystem block diagram in an SLX file.

A subsystem file:

- Cannot be simulated.
- Does not have a configuration set.
- Does not have a model workspace.
- Does not have code generation capability.

To create a subsystem block diagram:

1. Open Simulink.
2. On the Start Page, in the Simulink product group, click **Blank Subsystem**.



Alternatively, in the Simulink toolstrip, on the **Simulation** tab, select **New** and click **Blank Subsystem**. A Subsystem window opens.

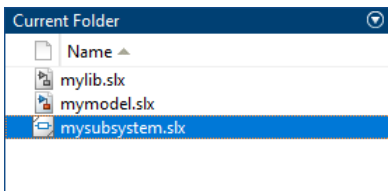
3. Add content and click **Save** .
4. Specify a file name in the **Save As** dialog box. The file name must be a valid MATLAB name.

   This creates a new subsystem file at the location specified.

To create a subsystem file programmatically, use the command:

```
new_system(subsystemfilename,'subsystem')
```

Once you create the subsystem file programmatically, it appears in the MATLAB File Browser as:



For more information on controlling subsystem files programmatically, see “Control Referenced Subsystem Programmatically” on page 4-29.

Reference a Subsystem File in a Model

1. Open a model in which you want to reference a subsystem block diagram.
2. Add a Subsystem Reference block using the Simulink Library Browser or the Quick Block Insert menu. An empty Subsystem Reference block is added to the Simulink canvas.
3. In the Simulink toolstrip, on the **Referenced Subsystem** tab, specify the name of the subsystem block diagram file in the **File Name** field, or click **Browse** to navigate to it.

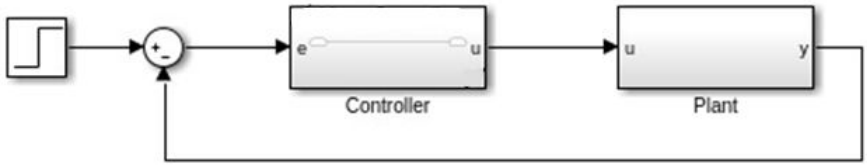
Alternatively, you can double-click the **Subsystem Reference** block to specify the subsystem block diagram file.

4 Click **OK** to save the changes.

Convert an Existing Subsystem to a Referenced Subsystem

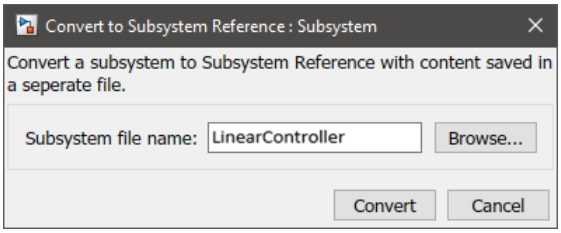
You can convert an existing Subsystem block to a Subsystem Reference block.

Consider a model with two Subsystem blocks as shown.



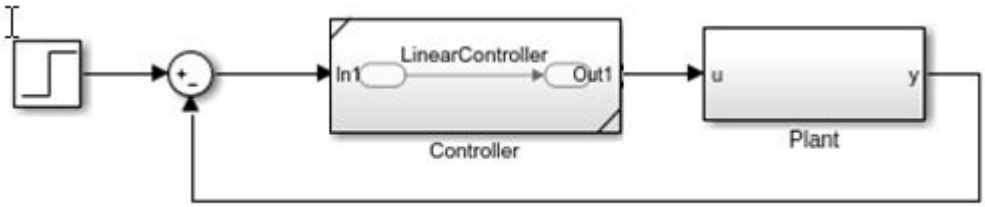
In this model, you have two Subsystem blocks – a Controller subsystem and a Plant subsystem. To convert the Controller Subsystem block to a referenced subsystem:

1 Select the Controller Subsystem block and on the **Subsystem Block** tab, select **Convert** and click **Convert to Subsystem Reference**.



2 Specify a name for the subsystem component in the **Subsystem file name** field and click **Convert**.The name must be a valid MATLAB name. The conversion creates a subsystem file in the current directory. To create the file in a specific location, click **Browse** and navigate to the save location.

The Subsystem block changes into a Subsystem Reference block with the name of the subsystem file displayed at the top of the block icon.



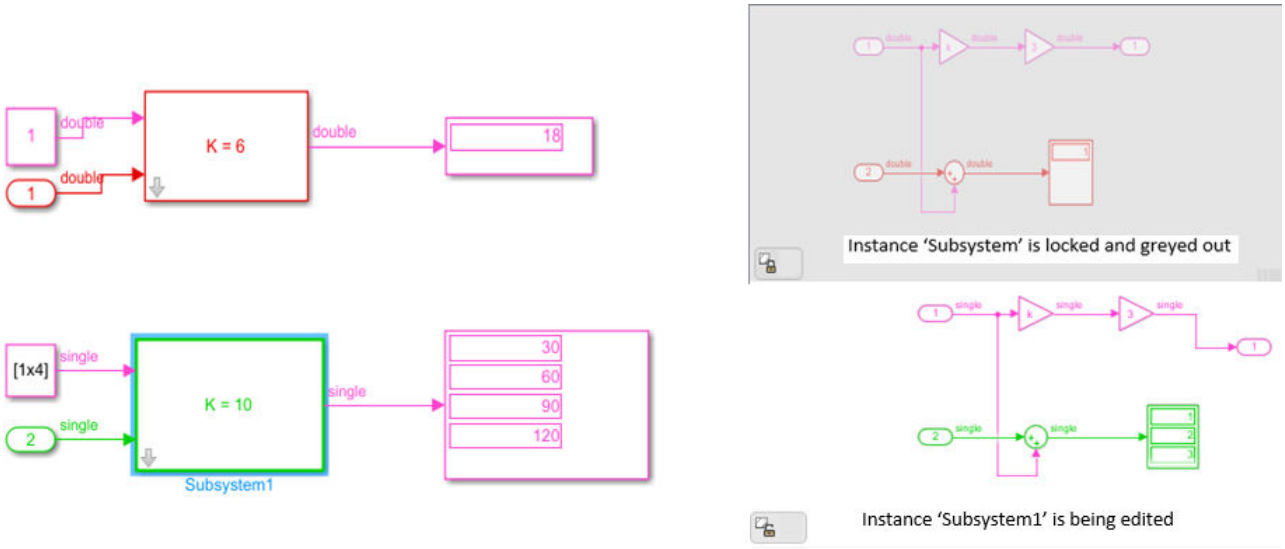
Tip When you convert a linked block to a referenced subsystem, do so in the parent library block of that linked block. Once you convert a parent library block to a referenced subsystem, all its linked block instances are also converted to referenced subsystems.

You cannot convert a subsystem to a referenced subsystem when the subsystem:

- Has no read/write permissions.
- Has a mask that is trying to modify its contents.

Edit and Save Referenced Subsystem

In a model containing multiple instances of a referenced subsystem, you can edit any instance and upon saving the changes the updates propagate to all the referenced instances. When you actively edit an instance of a referenced subsystem, all the other referenced instances are locked and are unavailable for edit.



A badge is shown at the bottom left corner of an open subsystem file to denote the availability of the file for edit. The badge shows two states:

-  indicates that the subsystem file is available for edit. Right-click the badge to see all the active instances of that subsystem and to navigate to each of them.
-  indicates that the subsystem file is unavailable for edit, as another instance is being actively edited. Right-click the badge to open the instance being edited.

Once you have edited an instance in a model, you can save the changes from the **Save** option available in the **Simulation** tab of the model window. The **Save All** drop-down gives you two options to save your changes:

- **Save All** – To save all the updates in the current model.
- **Save Specific Referenced File** – To save a specific subsystem file when you have made changes to multiple subsystem files.

Note If you edit any instance of a referenced subsystem in a model, save the changes before updating or simulating the model. Unsaved changes in referenced instances cause error during simulation.

Add a System Mask for Subsystem Reference

You can mask a subsystem file using a system mask. When you create a system mask for a subsystem file, all the referenced instances share the same system mask.

To mask a subsystem file:

1. Open the subsystem file to be masked.
2. In the Simulink toolstrip, on the **Subsystem** tab, click **Create System Mask**. Alternatively, right-click anywhere on the canvas and select **Mask** and click **Create System Mask**. The Mask Editor dialog opens.
3. Add mask parameters and click **OK**.

Note You cannot directly mask a Subsystem Reference block. To mask a Subsystem Reference block, select the block. On the **Referenced Subsystem** tab, click **Create System Mask**. This action opens the subsystem file being referenced and creates a mask on it.

For more information on creating and editing System masks, see “Introduction to System Mask” on page 39-48.

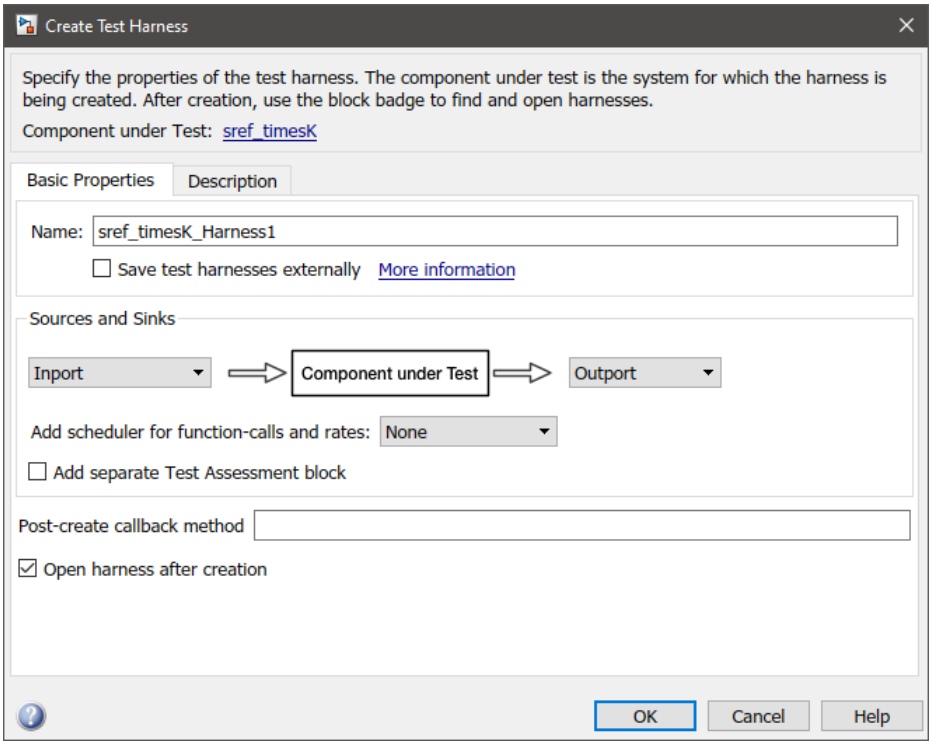
Simulate a Subsystem Block Diagram with a Test Harness

A subsystem block diagram cannot be simulated like a model or subsystem. However, you can create test harnesses on a subsystem block diagram and simulate the test harness. This action helps you to check for any errors or discrepancies while editing a subsystem block diagram. You can associate more than one test harness to a subsystem file and set a default test harness for the subsystem from a set of available test harnesses.

To simulate with a test harness:

1. Open a subsystem block diagram.
2. In the Simulink toolstrip, on the **Subsystem** tab, click **Add Test Harness**.

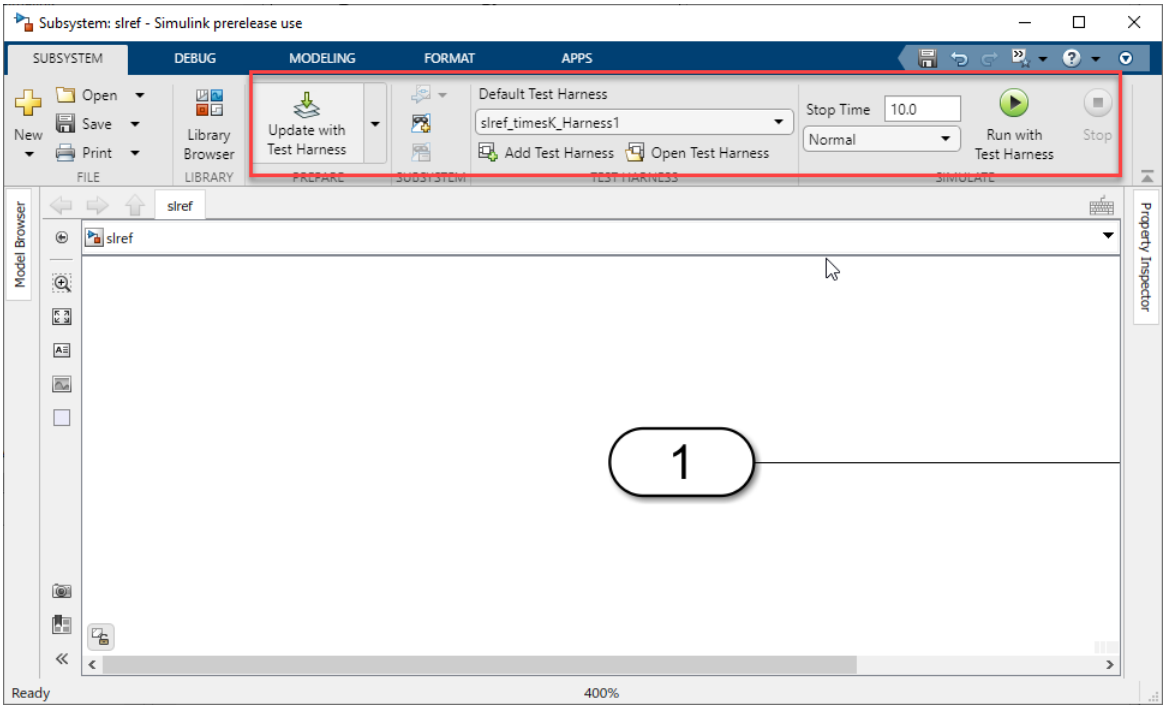
   The **Create Test Harness** window appears.



3 Specify a name for the new test harness and click **OK**. This becomes the default test harness.

4 Click **Run with Test Harness** on the toolstrip to simulate the test harness.

You can also stop the simulation with the **Stop** button, set **Stop Time** for the default simulation, and update the block diagram with the **Update with Test Harness** button.



Set the Test harness using the Command line

You can set the default test harness of a subsystem block diagram using the command:

```
set_param('<subsystemfilename>','DefaultTestHarness','<testHarnessName>');
```

Subsystem Reference Compatibility with Previous Versions

When you export a model containing referenced subsystems to a version prior to R2019b, all the Subsystem Reference blocks are converted to independent Subsystem blocks.

Subsystem files created in R2019b cannot be exported to a prior version. For information on how to export a simulink model to a previous version, see “Export a Model to a Previous Simulink Version” on page 1-31.

Control Referenced Subsystem Programmatically

You can create a referenced subsystem, find available referenced subsystems in a model, change the referenced subsystem file in a block, and check the block diagram type of an SLX file using a command-line interface.

Create a Referenced Subsystem

You can create a referenced subsystem using the new_system command:

```
new_system(subsystemfilename,'SubSystem')
```

Find Subsystem Reference in a Model

You can find if a model contains a referenced subsystem using the `Simulink.findBlocksOfType` function:

```
Simulink.findBlocksOfType(bdroot, 'SubSystem','ReferencedSubsystem','.',Simulink.FindOptions('Reg
```

You can also use the find_system command:

```
find_system(bdroot, 'RegExp','on','BlockType','SubSystem','ReferencedSubsystem', '.')
```

Both return the number of Subsystem Reference blocks in the model. By default, `find_system` lists all the child blocks inside a subsystem reference instance.

If you do not want `find_system` to look inside a referenced subsystem, use `find_system` with `LookInsideSubsystemReference` set to `off`. By default, `LookInsideSubsystemReference` is set to `on`.

Change the Referenced File for a Subsystem

You can change the subsystem file being referenced in a Subsystem Reference block through command-line interface using the `set_param` command:

```
set_param(gcb, 'ReferencedSubsystem', '<subsystemfilename>')
```

This command changes the file being currently referenced by the Subsystem Reference block and replaces it with the new subsystem file you specify.

Check if the SLX File is a Subsystem Block Diagram Type

You can check if an SLX file is a subsystem block diagram type that can be placed in a **Subsystem Reference** block using any of these commands:

```
bdIsSubsystem(bdname)
```

This command returns `logical 1` if bdname.slx is a Subsystem block diagram type and `logical 0` if it is not. When using this command, make sure that bdname.slx is loaded.

```
get_param(bdname,'BlockDiagramType')
```

This command returns `Subsystem` if bdname.slx is a Subsystem block diagram type. When using this command, make sure that bdname.slx is loaded.

```
Simulink.MDLInfo(bdname)
```

This command gives the entire model information where the `BlockDiagramType` property is shown as `Subsystem` if bdname.slx is a Subsystem block diagram type.

Best Practices

While using a referenced subsystem in a model:

- To mask a referenced subsystem, use a System mask.
- Do not reference a parent subsystem because it creates a reference loop.
- Only the subsystem file type can be referenced by a Subsystem Reference block.

See Also

Related Examples

- “Component-Based Modeling Guidelines” on page 22-2
- “Choose Among Types of Model Components” on page 22-4
- “Reference a Subsystem File in a Model” on page 4-31

Reference a Subsystem File in a Model

You can save a subsystem in an SLX file and reference it from a model. Such a modeling pattern helps re-usability. Consider the `slexSSRef_model` model. This model contains two subsystems referencing the same subsystem file - `TimesK.slx`. The `TimesK.slx` subsystem contains Inport, Gain, and Outport blocks and is masked using Model Mask. An **Edit** parameter is added to the mask to pass value using the variable `k`. This model is configured to demonstrate these capabilities of a referenced subsystem.

Instance Specific Parameterization

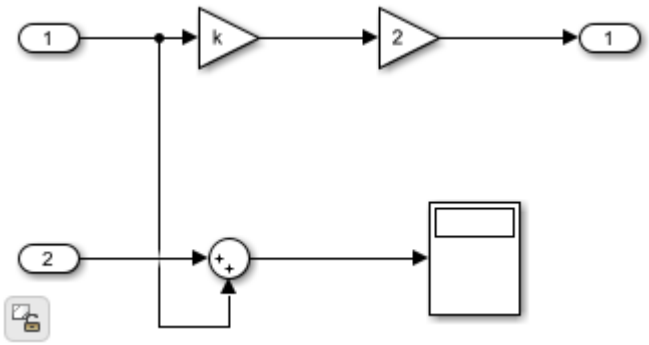
You can specify different parameter values for each instance of a referenced subsystem. For example, here the input value (k) for the **Edit** parameter from Subsystem1 and Subsystem2 are `5` and `10` respectively.

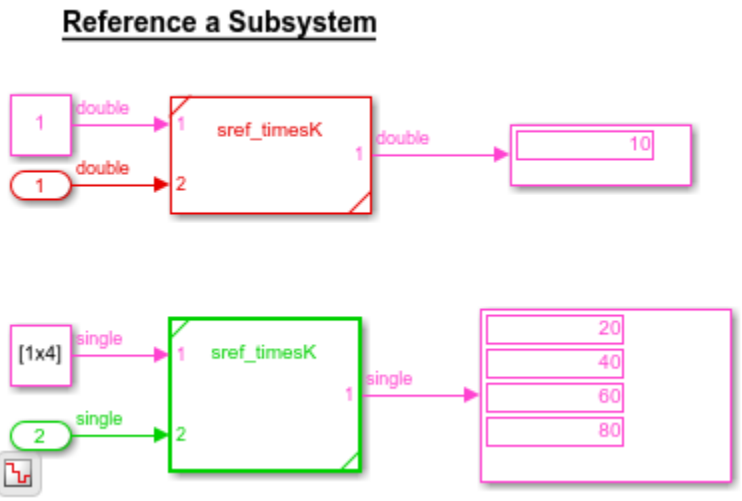
Instance Specific Debugging

In this example, you have two instances of subsystem, referencing the saved subsystem file, `TimesK`. Each instance is driven by its own set of inputs in the top model. When you need to debug a referenced subsystem, you can dive into each instance and look for errors and debug them.

Dimension Adaptability

While referencing subsystems, the top model can drive multiple dimensions. In this model, the Subsystem1 instance is driven by an one-dimensional signal and the Subsystem2 instance is driven by a four-dimensional signal. In both instances, you can see that the referenced subsystems adapt to have one dimension and four dimensions respectively in the output.





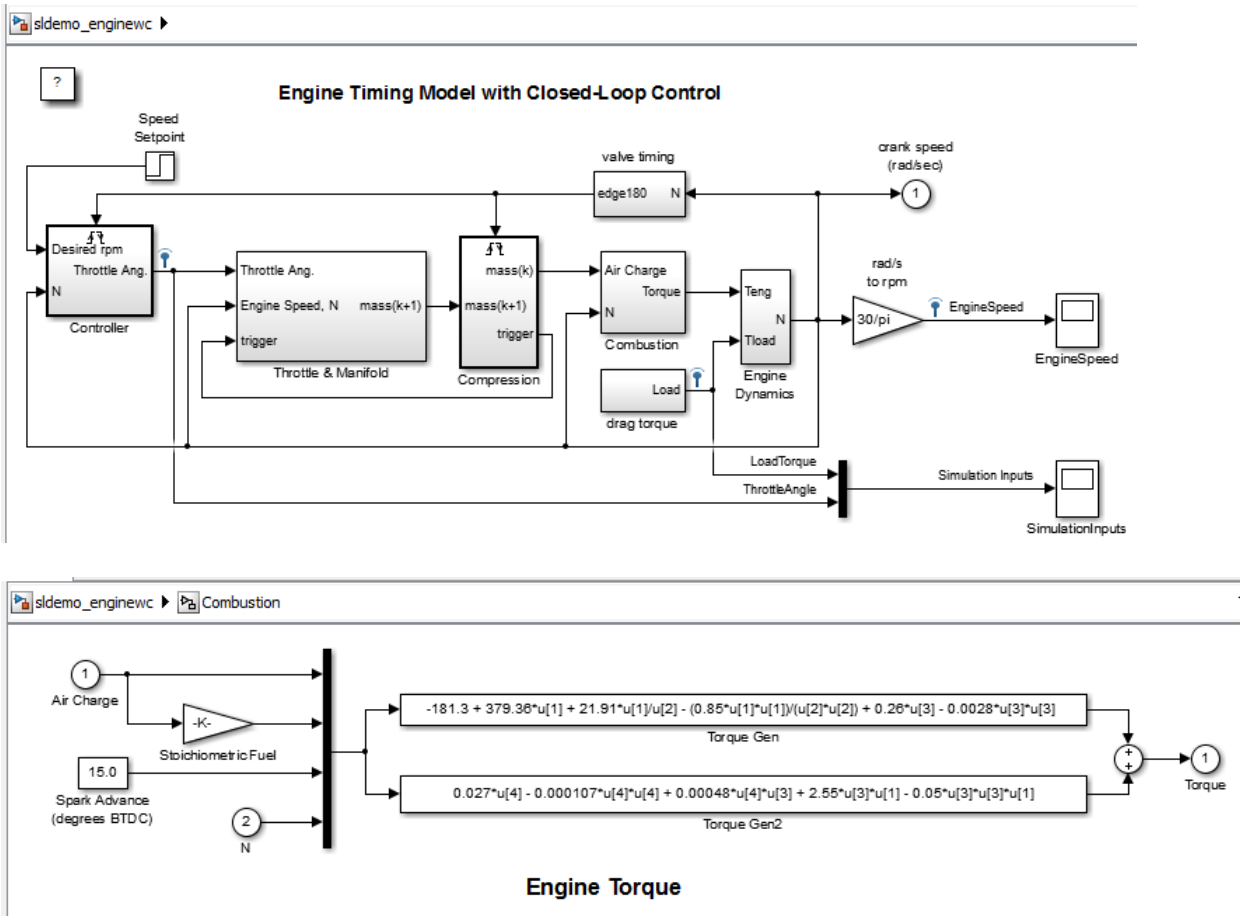
See Also

“Subsystem Reference” on page 4-23

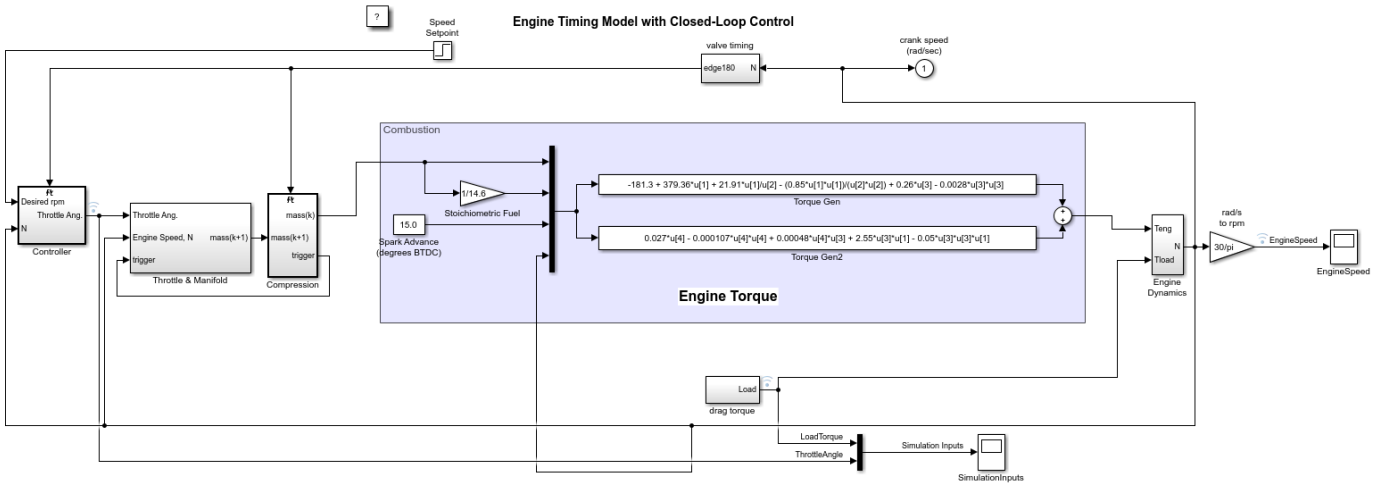
Expand Subsystem Contents

To move the contents of a subsystem into the containing system, you can expand the subsystem.

For example, the `sldemo_enginewc` model includes the Combustion subsystem.



After you expand the Combustion subsystem, the top level of the `sldemo_enginewc` model includes the blocks and signals of the Combustion subsystem. The expansion removes the Subsystem block and the Inport and Outport blocks.



Why Expand a Subsystem?

Expand a subsystem if you want to flatten a model hierarchy by bringing the contents of a subsystem up one level.

Expanding a subsystem is useful when refactoring a model. Flattening a model hierarchy can be the end result, or just one step in refactoring. For example, you could pull a set of blocks up to the parent system by expanding the subsystem, deselect the blocks that you want to leave in the parent, and then create a subsystem from the remaining selected blocks.

What Subsystems Can You Expand?

You can expand virtual subsystems that are not masked, linked, or commented. If you try to expand a masked, linked, or commented subsystem using the Simulink Editor, a message gives you the option of having Simulink modify the subsystem so that you can then expand it.

Kind of Subsystem	Modification
Masked subsystem	Removes all masking information
Library links	Breaks the link
Commented-out subsystem	Uncomments the subsystem

You cannot expand these subsystems:

- Atomic subsystems
- Conditional subsystems
- Configurable subsystems
- Variant subsystems
- Subsystems with the **Read/Write permissions** parameter set to `ReadOnly` or `NoReadWrite`
- Subsystems with an `InitFcn`, `StartFcn`, `PauseFcn`, `ContinueFcn`, or `StopFcn` callback
- Subsystems with linked requirements (using Simulink Requirements™ software)

Expand a Subsystem

To interactively expand a subsystem, right-click a Subsystem block and, from the context menu, select **Subsystem & Model Reference** > **Expand Subsystem**.

To programmatically expand a subsystem, use the `Simulink.BlockDiagram.expandSubsystem` function.

Tip Subsystem expansion applies to the currently selected subsystem level. Simulink does not expand other subsystems in a nested subsystem hierarchy.

To improve readability when you expand nested subsystems, start by expanding the highest-level subsystem that you want to expand, and then work your way down the hierarchy as far as you want to expand.

Results of Expanding a Subsystem

When you expand a subsystem, Simulink:

- Removes the Subsystem block
- Removes the root Inport, root Outport, and Simscape Connection Port blocks that were in the subsystem
- Connects the signal lines that went to the input and output ports of the subsystem directly to the ports of the blocks in the model that connected to the subsystem
- Distributes blocks and routes signals for readability.

Block Paths

The paths for blocks that were in the subsystem that you expanded change. After expansion, update scripts and test harnesses that rely on the hierarchical paths to blocks that were in the subsystem that you expanded.

Signal Names and Properties

If you expand a subsystem with a missing connection on the outside or inside of the subsystem, Simulink keeps the line labels, but uses the signal name and properties from just one of the lines. For lines corresponding to:

- A subsystem input port, Simulink uses the signal name and properties from the signal in the system in which the subsystem exists
- A subsystem output port, Simulink uses the signal name and properties from the subsystem

Display Layers

The display layers of blocks (in other words, which blocks appear in front or in back for overlapping blocks) does not change after expansion. Blocks in front of the Subsystem block remain above the expanded contents, and blocks below the Subsystem block remain under the expanded contents.

Execution Order and Block Priorities

When you compile a model, Simulink sorts the blocks in terms of the order of block execution. Expanding a subsystem can change block path names, which, in rare cases, can impact the block execution order.

If you explicitly set block execution order by setting block priorities within a subsystem, Simulink removes those block priority settings when you expand that subsystem.

Data Stores

Expanding a subsystem that contains a Data Store Memory block that other subsystems read from or write to can change the required data store write and read sequence. You may need to restructure your model. For details, see “Order Data Store Access” on page 73-19.

See Also

More About

- “Types of Subsystems” on page 4-15
- “Create Subsystems” on page 4-15
- “Navigate Model Hierarchies” on page 4-20

Use Control Flow Logic

In this section...

What is a Control Flow Subsystem

A *control flow subsystem* executes one or more times at the current time step when enabled by a control flow block. A control flow block implements control logic similar to that expressed by control flow statements of programming languages (e.g., `if-then`, `while-do`, `switch`, and `for`).

Equivalent C Language Statements

You can use block diagrams to model control flow logic equivalent to the following C programming language statements:

- `for`
- `if-else`
- `switch`
- `while`

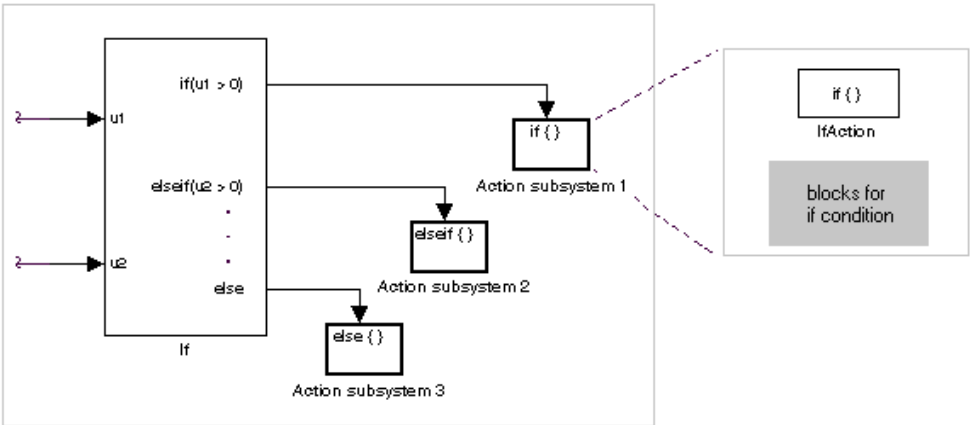
Conditional Control Flow Logic

You can use the following blocks to perform conditional control flow logic.

C Statement	Equivalent Blocks
`if-else`	If, If Action Subsystem
`switch`	Switch Case, Switch Case Action Subsystem

If-Else Control Flow

The following diagram represents `if-else` control flow.



Construct an `if-else` control flow diagram as follows:

1. Provide data inputs to the If block for constructing if-else conditions.

   In the If block parameters dialog box, set inputs to the If block. Internally, the inputs are designated as `u1, u2,..., un` and are used to construct output conditions.

2. In the If block parameters dialog box, set output port if-else conditions for the If block.

   In the If block parameters dialog box, set Output ports. Use the input values `u1, u2, ..., un` to express conditions for the if, elseif, and else condition fields in the dialog box. Of these, only the if field is required. You can enter multiple elseif conditions and select a check box to enable the else condition.

3. Connect each condition output port to an Action subsystem.

   Connect each if, elseif, and else condition output port on the If block to a subsystem to be executed if the port's case is true.

   Create these subsystems by placing an Action Port block in a subsystem. This creates an atomic Action subsystem with a port named Action, which you then connect to a condition on the If block.

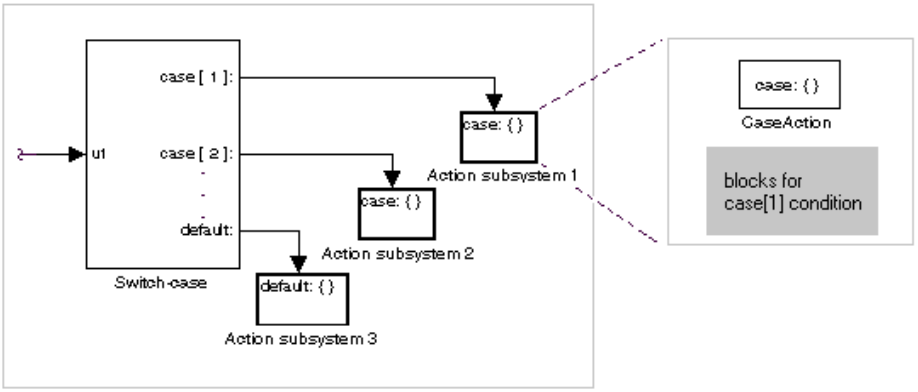
   Once connected, the subsystem takes on the identity of the condition it is connected to and behaves like an enabled subsystem.

For more detailed information, see the If and Action Port blocks.

Note All blocks in an Action subsystem driven by an If or Switch Case block must run at the same rate as the driving block.

Switch Control Flow

The following diagram represents `switch` control flow.



Construct a `switch` control flow statement as follows:

1. Provide a data input to the argument input of the Switch Case block.

   The input to the Switch Case block is the argument to the `switch` control flow statement. This value determines the appropriate case to execute. Noninteger inputs to this port are truncated.

2. Add cases to the Switch Case block based on the numeric value of the argument input.

   Using the parameters dialog box of the Switch Case block, add cases to the Switch Case block. Cases can be single or multivalued. You can also add an optional default case, which is true if no other cases are true. Once added, these cases appear as output ports on the Switch Case block.

3. Connect each Switch Case block case output port to an Action subsystem.

   Each case output of the Switch Case block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing an Action Port block in a subsystem. This creates an atomic subsystem with a port named Action, which you then connect to a condition on the Switch Case block. Once connected, the subsystem takes on the identity of the condition and behaves like an enabled subsystem. Place all the block programming executed for that case in this subsystem.

For more detailed information, see documentation for the Switch Case and Action Port blocks.

Note After the subsystem for a particular case executes, an implied break executes, which exits the `switch` control flow statement altogether. Simulink `switch` control flow statement implementations do not exhibit the “fall through” behavior of C `switch` statements.

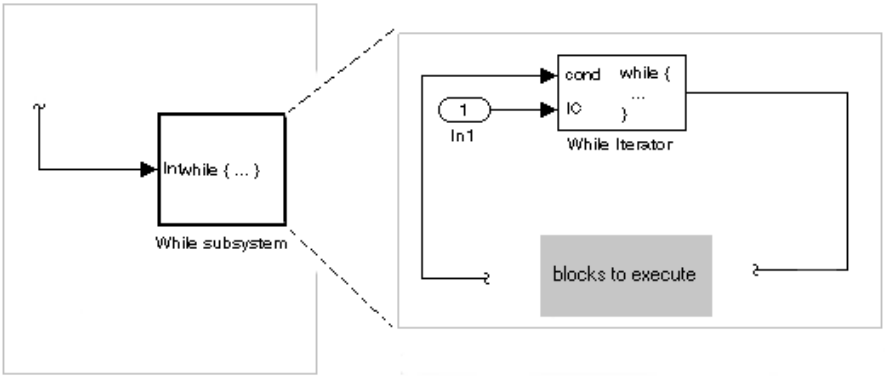
While and For Loops

Use the following blocks to perform `while` and `for` loops.

C Statement	Equivalent Blocks
`do-while`	While Iterator Subsystem
`for`	For Iterator Subsystem
`while`	While Iterator Subsystem

While Loops

The following diagram illustrates a `while` loop.



In this example, Simulink repeatedly executes the contents of the While subsystem at each time step until a condition specified by the While Iterator block is satisfied. In particular, for each iteration of the loop specified by the While Iterator block, Simulink invokes the update and output methods of all

the blocks in the While subsystem in the same order that the methods would be invoked if they were in a noniterated atomic subsystem.

Note Simulation time does not advance during execution of a While subsystem's iterations. Nevertheless, blocks in a While subsystem treat each iteration as a time step. As a result, in a While subsystem, the output of a block with states (that is, a block whose output depends on its previous input), reflects the value of its input at the previous iteration of the `while` loop. The output does *not* reflect that block's input at the previous simulation time step. For example, a Unit Delay block in a While subsystem outputs the value of its input at the previous iteration of the `while` loop, not the value at the previous simulation time step.

Construct a `while` loop as follows:

1. Place a While Iterator block in a subsystem.

   The host subsystem label changes to `while {...}`, to indicate that it is modeling a while loop. These subsystems behave like triggered subsystems. This subsystem is host to the block programming that you want to iterate with the While Iterator block.

2. Provide a data input for the initial condition data input port of the While Iterator block.

   The While Iterator block requires an initial condition data input (labeled `IC`) for its first iteration. This must originate outside the While subsystem. If this value is nonzero, the first iteration takes place.

3. Provide data input for the conditions port of the While Iterator block.

   Conditions for the remaining iterations are passed to the data input port labeled `cond`. Input for this port must originate inside the While subsystem.

4. (Optional) Set the While Iterator block to output its iterator value through its properties dialog.

   The iterator value is 1 for the first iteration and is incremented by 1 for each succeeding iteration.

5. (Optional) Change the iteration of the While Iterator block to `do-while` through its properties dialog.

   This changes the label of the host subsystem to `do {...} while`. With a `do-while` iteration, the While Iteration block no longer has an initial condition (IC) port, because all blocks in the subsystem are executed once before the condition port (labeled `cond`) is checked.

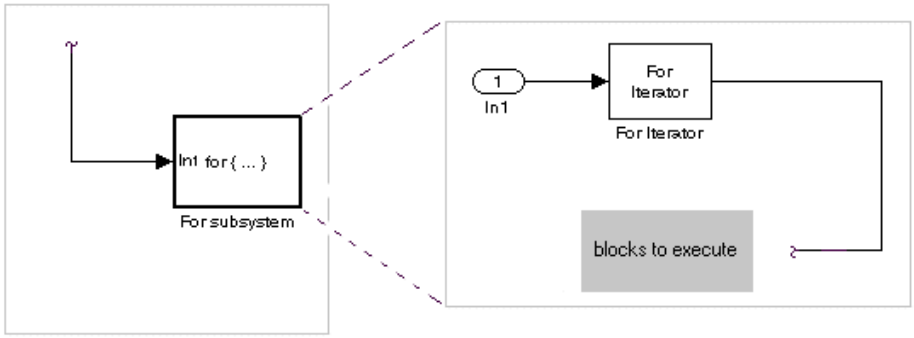
6. Create a block diagram in the subsystem that defines the subsystem's outputs.

   Note The diagram must not contain blocks with continuous states (for example, blocks from the Continuous block library). The sample times of all the blocks must be either inherited (`-1`) or constant (`inf`).

For more information, see the While Iterator block.

Modeling For Loops

The following diagram represents a `for` loop:



In this example, Simulink executes the contents of the For subsystem multiples times at each time step. The input to the For Iterator block specifies the number of iterations. For each iteration of the `for` loop, Simulink invokes the update and output methods of all the blocks in the For subsystem in the same order that it invokes the methods if they are in a noniterated atomic subsystem.

Note Simulation time does not advance during execution of a For subsystem's iterations. Nevertheless, blocks in a For subsystem treat each iteration as a time step. As a result, in a For subsystem, the output of a block with states (that is, a block whose output depends on its previous input) reflects the value of its input at the previous iteration of the `for` loop. The output does *not* reflect that block's input at the previous simulation time step. For example, a Unit Delay block in a For subsystem outputs the value of its input at the previous iteration of the `for` loop, not the value at the previous simulation time step.

Construct a `for` loop as follows:

1. Drag a For Iterator Subsystem block from the Library Browser or Library window into your model.
2. (Optional) Set the For Iterator block to take external or internal input for the number of iterations it executes.

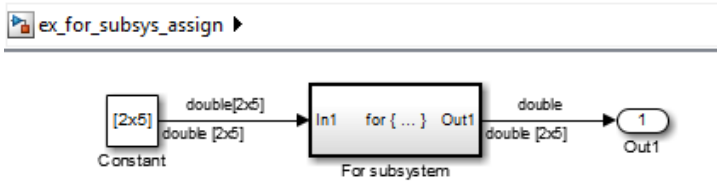
   Through the properties dialog of the For Iterator block you can set it to take input for the number of iterations through the port labeled `N`. This input must come from outside the For Iterator Subsystem.

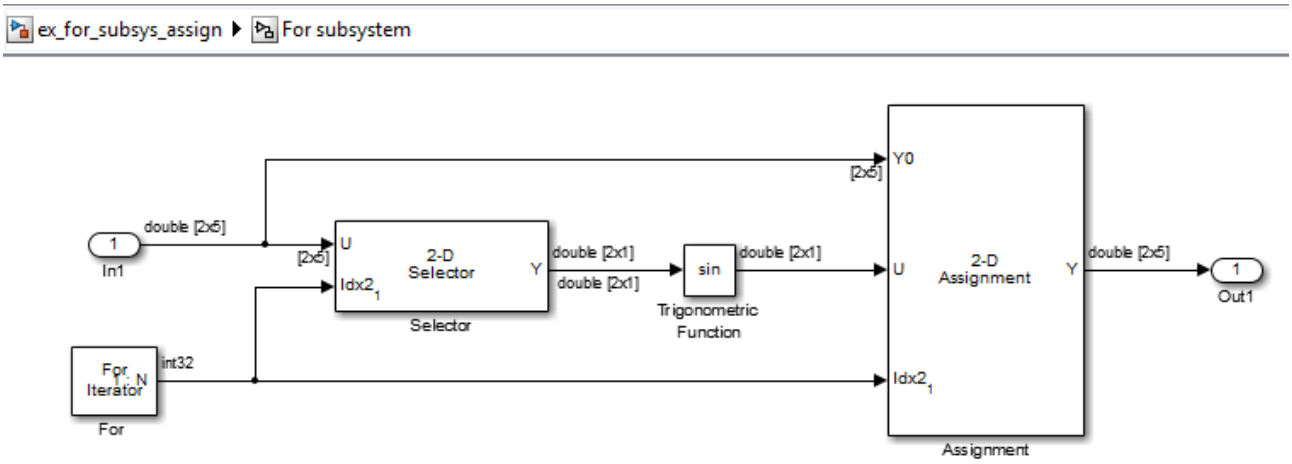
   You can also set the number of iterations directly in the properties dialog.
3. (Optional) Set the For Iterator block to output its iterator value for use in the block programming of the For Iterator Subsystem.

   The iterator value is 1 for the first iteration and is incremented by 1 for each succeeding iteration.
4. Create a block diagram in the subsystem that defines the subsystem's outputs.

   Note The diagram must not contain blocks with continuous states (for example, blocks from the Continuous block library). The sample times of all the blocks must be either inherited (`-1`) or constant (`inf`).

The For Iterator block works well with the Assignment block to reassign values in a vector or matrix. The following example shows the use of a For Iterator block. Note the matrix dimensions in the data being passed.





The above example outputs the sine value of an input 2-by-5 matrix (2 rows, 5 columns) using a For subsystem containing an Assignment block. The process is as follows.

1. A 2-by-5 matrix is input to the Selector block and the Assignment block.
2. The Selector block strips off a 2-by-1 matrix from the input matrix at the column value indicated by the current iteration value of the For Iterator block.
3. The sine of the 2-by-1 matrix is taken.
4. The sine value 2-by-1 matrix is passed to an Assignment block.
5. The Assignment block, which takes the original 2-by-5 matrix as one of its inputs, assigns the 2-by-1 matrix back into the original matrix at the column location indicated by the iteration value.

   The rows specified for reassignment in the property dialog for the Assignment block in the above example are [1,2]. Because there are only two rows in the original matrix, you could also have specified -1 for the rows, (that is, all rows).

Note The Trigonometric Function block is already capable of taking the sine of a matrix. The above example uses the Trigonometric Function block only as an example of changing each element of a matrix with the collaboration of an Assignment block and a For Iterator block.

See Also

Assignment | For Iterator | For Iterator Subsystem | While Iterator | While Iterator Subsystem | While Iterator Subsystem

Callbacks for Customized Model Behavior

In this section...

Model, Block, and Port Callbacks

Callbacks are commands you can define that execute in response to a specific modeling action, such as opening a model or stopping a simulation. Callbacks define MATLAB expressions that execute when the block diagram or a block is acted upon in a particular way.

Simulink provides model, block, and port callback parameters that identify specific kinds of model actions. You provide the code for a callback parameter. Simulink executes the callback code when the associated modeling action occurs.

For example, the code that you specify for the `PreLoadFcn` model callback parameter executes before the model loads. You can provide code for `PreLoadFcn` that loads the variables that model uses into the MATLAB workspace.

What You Can Do with Callbacks

Callbacks are a powerful way to customize your Simulink model. A callback executes when you perform actions on your model, such as double-clicking a block or starting a simulation. You can use callbacks to execute MATLAB code. You can use model, block, or port callbacks to perform common tasks, such as:

- “Load Variables When Opening a Model” on page 1-11
- “Specify Block Callbacks” on page 4-49
- “Automate Simulation Tasks Using Callbacks” on page 26-6

Avoid run Commands in Callback Code

Do not call the `run` command from within model or block callback code. Doing so can result in unexpected behavior (such as errors or incorrect results) if you load, compile, or simulate a Simulink model.

See Also

Related Examples

- “Model Callbacks” on page 4-45
- “Block Callbacks” on page 4-49
- “Port Callbacks” on page 4-55
- “Callback Tracing” on page 4-56
- “Fast Restart Methodology” on page 81-7

Model Callbacks

In this section...

Model callbacks execute at specified action points, for example after you load or save the model.

You can set most of the same callbacks for libraries. Only the callbacks that can execute for a library are available to set for a library. For example, you cannot set the `InitFcn` callback for a library, which is called as part of simulation, because you cannot simulate a library.

Create Model Callbacks

1. In the Simulink Editor, open the Property Inspector. In the **Modeling** tab, under **Design**, click **Property Inspector**.
2. With no selection at the top level of your model, in the **Properties** tab, in the **Callbacks** section, select the callback you want to set.
3. In the box, enter the functions you want the callback to perform.

To create a model callback programmatically, use the `set_param` function to assign MATLAB code to a model callback parameter. See “Model Callback Parameters” on page 4-46

Referenced Model Callbacks

In a model hierarchy, the execution of callbacks reflects the order in which the top model and the models it references execute their callbacks. For example, suppose:

- Model A:
  - References model B in accelerator mode.
  - Has a `PostLoadFcn` callback that creates variables in the MATLAB workspace.
  - Has the `Rebuild` configuration parameter set to `Always`, `If changes detected`, or `If any changes in known dependencies detected`.
- Model B:
  - Has a `CloseFcn` callback that clears the MATLAB workspace.
  - Has not been built or is out of date.

Simulating model A triggers rebuilding the referenced model B. When Simulink rebuilds model B, it opens and closes model B, which invokes the model B `CloseFcn` callback. `CloseFcn` clears the MATLAB workspace, including the variables created by the model A `OpenFcn` callback.

Instead of using a `CloseFcn` callback for model B, you can use a `StopFcn` callback in model A to clear the variables used by the model from the MATLAB workspace. Alternatively, you can use a data dictionary for the data to avoid the need to have variables in the base workspace.

If a model references multiple instances of the same model in normal mode, callbacks execute for each instance.

For models referenced in accelerator mode, Simulink does not execute some callbacks. If everything is up to date and the **Rebuild** configuration parameter is set to `If any changes in known dependencies detected`, then the referenced model does not compile and its `InitFcn` callbacks do not execute. Callbacks such as `StartFcn` and `StopFcn` do not execute because referenced models in accelerator mode use an S-function, which starts and stops instead of the referenced model.

Note Simulation outputs are not available in the `StopFcn` callbacks for command-line simulations.

Model Callback Parameters

Model Loading and Closing Callback Parameters

Model Callback Parameter	When Executed

`PreLoadFcn`	This callback is executed before the model is loaded.
	
	Do not use model parameters in a `PreLoadFcn` model callback because parameters are loaded after the model is loaded. Instead, use a `PostLoadFcn` callback to work with model parameters when the model is loaded.
	
	Defining a callback code for this parameter is useful for loading variables that the model uses.
	
	If you want to call your model from a MATLAB file without opening your model, use the `load_system` function so that the `PreLoadFcn` executes.
	
	For examples, see:
	
	• “Load Variables When Opening a Model” on page 1-11
	• “Introduction to Managing Data with Model Reference”
	• “Manage Simulation Targets for Referenced Models” on page 8-50
	
	Limitations include:
	
	• For the `PreLoadFcn` callback, `get_param` does not return the model parameter values because the model is not yet loaded. Instead, `get_param` returns:
	  • The default value for a standard model parameter such as `solver`
	  • An error message for a model parameter added with `add_param`
	• Programmatic access to Scopes is not supported.

Model Callback Parameter	When Executed
`PostLoadFcn`	After the model is loaded. Defining callback code for this parameter may be useful for generating an interface requiring a loaded model. Limitations include: • If you make structural changes with `PostLoadFcn`, the function does not set the model `Dirty` flag to indicate unsaved changes. When you close the model, Simulink does not prompt you to save. • Programmatic access to Scopes is not supported. Because the Simulink Editor opens after this callback executes, the `PostLoadFcn` callback is not suitable for setting up the model view, for example setting a zoom factor. Save zoom information with the model to open it with a particular zoom factor.
`CloseFcn`	Before the block diagram is closed. Any `ModelCloseFcn` and `DeleteFcn` callbacks set on blocks in the model are called prior to the model `CloseFcn` callback. The `DestroyFcn` callback of any blocks in the model is called after the model `CloseFcn` callback.

Model Saving Callback Parameters

Model Callback Parameter	When Executed
`PreSaveFcn`	Before the model is saved.
`PostSaveFcn`	After the model is saved. **Note** If you make structural changes with `PostSaveFcn`, the function does not set the model `Dirty` flag to indicate unsaved changes. When you close the model, Simulink does not prompt you to save.

Model Simulation Callback Parameters

Model Callback Parameter	When Executed
`InitFcn`	Called during update phase before block parameters are evaluated. This is called during model update and simulation. For more information on `InitFcn` callback, see “Initialization Function” on page 12-108 For examples, see: • “Create Programmatic Hyperlinks” on page 25-64 • “Track Object Using MATLAB Code” on page 44-134
`StartFcn`	Called before the simulation phase. This is not called during model update.
`PauseFcn`	After the simulation pauses.

Model Callback Parameter	When Executed
`ContinueFcn`	Before the simulation continues.
`StopFcn`	After the simulation stops. Output is written to workspace variables and files before the `StopFcn` is executed. Simulation outputs are not available in the `StopFcn` callbacks for command-line simulations.

See Also

Related Examples

- “Callbacks for Customized Model Behavior” on page 4-44
- “Block Callbacks” on page 4-49
- “Port Callbacks” on page 4-55
- “Callback Tracing” on page 4-56
- “Fast Restart Methodology” on page 81-7

Block Callbacks

In this section...

Specify Block Callbacks

1. Open the Property Inspector. In the **Modeling** tab, under **Design**, click **Property Inspector**.
2. Select the block whose callback you want to specify. In the **Properties** tab of the Property Inspector, in the **Callbacks** section, select the callback you want to define.
3. In the box, enter the functions you want the callback to perform.

To specify a block callback programmatically, use `set_param` to assign MATLAB code to the block callback parameter.

Block Callback Parameters

If a block callback executes before or after a modeling action takes place, that callback occurs immediately before or after the action.

Block Opening Callback Parameters

Block Callback Parameter	When Executed
`OpenFcn`	When the block is opened. Generally, use this parameter with Subsystem blocks. The callback executes when you double-click the block or when you use `open_system` with the block as an argument. The `OpenFcn` parameter overrides the normal behavior associated with opening a block, which is to display the block dialog box or to open the subsystem. Examples of tasks that you can use `OpenFcn` for include defining variables for a block, making a call to MATLAB to produce a plot of simulated data, or generating a graphical user interface. After you add an `OpenFcn` callback to a block, double-clicking the block does not open the block dialog box. Also, the block parameters do not appear in the Property Inspector when the block is selected. To set the block parameters, select **Block Parameters** from the block context menu. For examples of using `OpenFcn` with model referencing, see: • In the Introduction to Managing Data with Model Reference example, click the question mark block at the top and then select `Detailed Workflow for Managing Data with Model Reference`. • “Manage Simulation Targets for Referenced Models” on page 8-50
`LoadFcn`	After the block diagram is loaded. For Subsystem blocks, the `LoadFcn` callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a `LoadFcn` callback defined.

Block Editing Callback Parameters

Block Callback Parameter	When Executed
`MoveFcn`	When the block is moved or resized.
`NameChangeFcn`	After a block name or path changes. When a Subsystem block path changes, the Subsystem block calls the `NameChangeFcn` callback of its descendant blocks and then calls the `NameChangeFcn` callback on itself.

Block Callback Parameter	When Executed
`PreCopyFcn`	Before a block is copied. The `PreCopyFcn` is also executed if `add_block` is used to copy the block. If you copy a Subsystem block that contains a block for which the `PreCopyFcn` callback is defined, that callback executes also. The block `CopyFcn` callback is called after all `PreCopyFcn` callbacks are executed.
`CopyFcn`	After a block is copied. The callback is also executed if `add_block` is used to copy the block. If you copy a Subsystem block that contains a block for which the `CopyFcn` parameter is defined, the callback is also executed.
`ClipboardFcn`	When the block is copied or cut to the system clipboard.
`PreDeleteFcn`	Before a block is graphically deleted (for example, when you graphically delete the block or invoke `delete_block` on the block). The `PreDeleteFcn` is not called when the model containing the block is closed. The block's `DeleteFcn` is called after the `PreDeleteFcn`, unless the `PreDeleteFcn` invokes the `error` command, either explicitly or via a command used in the `PreDeleteFcn`.
`DeleteFcn`	After a block is graphically deleted (for example, when you graphically delete the block, invoke `delete_block` on the block, or close the model containing the block). When the `DeleteFcn` is called, the block handle is still valid and can be accessed using `get_param`. If the block is graphically deleted by invoking `delete_block` or by closing the model, after deletion the block is destroyed from memory and the block's `DestroyFcn` is called. For Subsystem blocks, the `DeleteFcn` callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a `DeleteFcn` callback defined.
`DestroyFcn`	When the block has been destroyed from memory (for example, when you invoke `delete_block` on either the block or a subsystem containing the block or close the model containing the block). If the block was not previously graphically deleted, the block`DeleteFcn` callback is called prior to the `DestroyFcn`. When the `DestroyFcn` is called, the block handle is no longer valid.
`UndoDeleteFcn`	When a block deletion is undone.

Block Compilation and Simulation Callback Parameters

Block Callback Parameter	When Executed
`InitFcn`	Before the block diagram is compiled and before block parameters are evaluated. For more information on `InitFcn` callback, see “Initialization Function” on page 12-108.

Block Callback Parameter	When Executed
`StartFcn`	After the block diagram is compiled and before the simulation starts. In the case of an S-Function block, `StartFcn` executes immediately before the first execution of the block's `mdlProcessParameters` function. For more information, see "S-Function Callback Methods".
`ContinueFcn`	Before the simulation continues.
`PauseFcn`	After the simulation pauses.
`StopFcn`	At any termination of the simulation. In the case of an S-Function block, `StopFcn` executes after the block's `mdlTerminate` function executes. For more information, see "S-Function Callback Methods".

Block Saving and Closing Callback Parameters

Block Callback Parameter	When Executed
`PreSaveFcn`	Before the block diagram is saved. For Subsystem blocks, the `PreSaveFcn` callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a `PreSaveFcn` callback defined.
`PostSaveFcn`	After the block diagram is saved. For Subsystem blocks, the `PostSaveFcn` callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a `PostSaveFcn` callback defined.
`CloseFcn`	When the block is closed using `close_system`. The `CloseFcn` is not called when you interactively close the block parameters dialog box, when you interactively close the subsystem or model containing the block, or when you close the subsystem or model containing a block using `close_system`. For example, to close all open MATLAB windows, use a command such as: `set_param('my_model','CloseFcn','close all')`
`ModelCloseFcn`	Before the block diagram is closed. When the model is closed, the block's `ModelCloseFcn` is called prior to its `DeleteFcn`. For Subsystem blocks, the `ModelCloseFcn` callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a `ModelCloseFcn` callback defined.

Subsystem Block Callback Parameters

You can use the other block callback parameters with Subsystem blocks, but the callback parameters in this table are specific to Subsystem blocks.

Note A callback for a masked subsystem cannot directly reference the parameters of the masked subsystem (see “Create Block Masks”). Simulink evaluates block callbacks in the MATLAB base workspace, whereas the mask parameters reside in the masked subsystem's private workspace. A block callback, however, can use `get_param` to obtain the value of a mask parameter. For example, here `gain` is the name of a mask parameter of the current block:

```
get_param(gcb, 'gain')
```

Block Callback Parameter	When Executed
`DeleteChildFcn`	After a block or line is deleted in a subsystem. If the block has a `DeleteFcn` or `DestroyFcn` callback, those callbacks execute prior to the `DeleteChildFcn` callback.

`ErrorFcn`	When an error has occurred in a subsystem.
	
	Use the following form for the callback code for the `ErrorFcn` parameter:
	
	`newException = errorHandler(subsys, ...`
	`errorType, originalException)`
	
	where
	
	• `errorHandler` is the name of the function.
	• `subsys` is a handle to the subsystem in which the error occurred.
	• `errorType` is a character vector indicating the type of error that occurred.
	• `originalException` is an MSLException (see “Error Handling in Simulink Using MSLException” on page 26-19).
	• `newException` is an MSLException specifying the error message to be displayed to the user.
	
	If you provide the original exception, then you do not need to specify the subsystem and the error type.
	
	The following command sets the `ErrorFcn` of the subsystem `subsys` to call the `errorHandler` callback:
	
	`set_param(subsys,'ErrorFcn','errorHandler')`
	
	In such calls to `set_param`, do not include the input arguments of the callback code. Simulink displays the error message returned by the callback.

Block Callback Parameter	When Executed
`ParentCloseFcn`	Before closing a subsystem containing the block or when the block is made part of a new subsystem using either: • The `new_system` function • The **Create Subsystem** icon on the **Multiple** tab, in the Simulink Editor. When you close the model, Simulink does not call the `ParentCloseFcn` callbacks of blocks at the root model level.

See Also

Related Examples

- “Callbacks for Customized Model Behavior” on page 4-44
- “Model Callbacks” on page 4-45
- “Port Callbacks” on page 4-55
- “Callback Tracing” on page 4-56
- “Fast Restart Methodology” on page 81-7

Port Callbacks

Block input and output ports have a single callback parameter, `ConnectionCallback`. This parameter allows you to set callbacks on ports that are triggered every time the connectivity of these ports changes. Examples of connectivity changes include adding a connection from the port to a block, deleting a block connected to the port, and deleting, disconnecting, or connecting branches or lines to the port.

Use `get_param` to get the port handle of a port and `set_param` to set the callback on the port. The callback code must have one input argument that represents the port handle. The input argument is not included in the call to `set_param`.

For example, suppose the currently selected block has a single input port. The following code sets `foo` as the connection callback on the input port:

```
phs = get_param(gcb, 'PortHandles');
set_param(phs.Inport, 'ConnectionCallback', 'foo');
```

where, `foo` is defined as:

```
function foo(portHandle)
```

See Also

Related Examples

- “Callbacks for Customized Model Behavior” on page 4-44
- “Model Callbacks” on page 4-45
- “Block Callbacks” on page 4-49
- “Callback Tracing” on page 4-56
- “Fast Restart Methodology” on page 81-7

Callback Tracing

Use callback tracing to determine the callbacks that Simulink invokes and the order that it invokes them when you open, edit, or simulate a model.

To enable callback tracing, do one of the following:

- In the Simulink Preferences dialog box, select the **Callback tracing** preference.
- Execute `set_param(0, 'CallbackTracing', 'on')`.

  The `CallbackTracing` parameter causes the callbacks to appear in the MATLAB Command Window as they are invoked. This option applies to all Simulink models, not just models that are open when you enable the preference.

See Also

Related Examples

- “Callbacks for Customized Model Behavior” on page 4-44
- “Model Callbacks” on page 4-45
- “Block Callbacks” on page 4-49
- “Port Callbacks” on page 4-55
- “Fast Restart Methodology” on page 81-7

Manage Model Versions and Specify Model Properties

In this section...

How Simulink Helps You Manage Model Versions

In Simulink, you can manage multiple versions of a model using these techniques:

- Use Projects to manage your project files, connect to source control, review modified files, and compare revisions. See “Project Management”.
- Use model file change notification to manage work with source control operations and multiple users. See “Model File Change Notification” on page 4-57.
- See `Simulink.MDLInfo` to extract information from a model file without loading the block diagram into memory. You can use `MDLInfo` to query model version and Simulink version, find the names of referenced models without loading the model into memory, and attach arbitrary metadata to your model file.

Model File Change Notification

You can use a Simulink preference to specify whether to notify you if the model has changed on disk. You can receive this notification when updating or simulating the model, first editing the model, or saving the model. The model can change on disk, for example, with source control operations and multiple users.

In the Simulink Editor, on the **Modeling** tab, select **Environment > Simulink Preferences**. In the **Model File** pane, under **Change Notification**, select the appropriate action.

- If you select **First editing the model**, the file has changed on disk, and the block diagram is unmodified in Simulink:
  - Any interactive operation that modifies the block diagram (e.g., adding a block) causes a warning to appear.
  - Any command-line operation that modifies the block diagram (such as a call to `set_param`) causes a warning to appear.
- If you select **Saving the model**, and the file has changed on disk:
  - Saving the model in the Simulink Editor causes a message to appear.
  - The `save_system` function reports an error, unless you use the `OverwriteIfChangedOnDisk` option.

To programmatically check whether the model has changed on disk since it was loaded, use the function `slIsFileChangedOnDisk`.

For more options that help you work with source control and multiple users, see “Project Management”.

Manage Model Properties

You can use the Property Inspector to view and edit model version properties, description, and callback functions. To open the Property Inspector, in the **Modeling** tab, under **Design**, click **Property Inspector**. Model properties or, if you are in a library model, library properties, appear in the Property Inspector when nothing is selected at the top level of a model.

Specify the Current User

When you create or update a model, your name is logged in the model. Simulink assumes that your name is specified by at least one of the `USER`, `USERNAME`, `LOGIN`, or `LOGNAME` environment variables. If your system does not define any of these variables, Simulink does not update the user name in the model.

UNIX® systems define the `USER` environment variable and set its value to the name you use to log in to your system. Thus, if you are using a UNIX system, you do not have to take further action for Simulink to identify you as the current user.

Windows systems can define environment variables for user name that Simulink expects, depending on the version of Windows installed on your system and whether it is connected to a network. Use the MATLAB function `getenv` to determine which of the environment variables is defined. For example, at MATLAB command prompt, enter:

```
getenv('user')
```

This function determines whether the `USER` environment variable exists on your Windows system. If it does not, set it.

Model Information

The **Info** tab summarizes information about the current version of the model, such as modifications, version, and last saved date. You can view and edit model information and enable, view, and edit the model's change history.

Use the **Description** section to enter a description of the model. You can then view the model description by entering `help` followed by the model name at the MATLAB command prompt.

- **Model version**

  Version number for this model. The major model version is incremented by the number of releases passed since the model was last saved. The minor model version is reset to zero for every new release of Simulink and is incremented by one each time you save the model within the same release.
- **Created by**

  Name of the person who created this model based on the value of the `USER` environment variable when the model is created.
- **Created on**

  Date and time this model was created. Do not change this value.
- **Last saved by**

  Name of the person who last saved this model based on the value of the `USER` environment variable when the model is saved.

- **Last saved on**

  Date that this model was last saved, based on the system date and time.

Properties

You can view the source file location, set the model compression level, specify where to save model design data, and define callbacks in the **Properties** tab of the model properties.

Note Library properties also enable you to specify the mapping from old library blocks to new library blocks. For information on using forwarding tables for this purpose, see “Forwarding Tables” on page 41-34.

Set SLX Compression Level

In the **Properties** tab of the **Property Inspector**, you can select one of three **SLX Compression** options:

- `None` applies no compression during the save operation.
- `Normal`, the default, creates the smallest file size.
- `Fastest` creates a smaller file size than you would get by selecting `None`, but provides a faster save time than `Normal`.

To set the compression level programmatically, use `SLXCompressionType`.

Tip You can reduce your Git™ repository size by saving Simulink models without compression. Turning off compression results in larger SLX files on disk but reduces repository size.

To use this setting with new SLX files, create your models using a model template with SLX Compression set to none. See “Create a Template from a Model” on page 4-2. For existing SLX files, set compression and then save the model.

Define Location of Design Data

Use the **Design Data** section to specify the location of the design data that your model uses. You can define design data in the base workspace or in a data dictionary. See “Migrate Single Model to Use Dictionary” on page 74-6.

Callbacks

Use the **Callbacks** section to specify functions to invoke at specific points in the simulation of the model. Select the callback from the list. In the box, enter the function you want to invoke for the selected callback. For information on these callbacks, see “Create Model Callbacks” on page 4-45.

Access Model Information Programmatically

Some version information is stored as model parameters in a model. You can access this information programmatically using the Simulink `get_param` function.

The table describes the model parameters used by Simulink to store version information.

Property	Description
Created	Date created.
Creator	Name of the person who created this model.
Description	User-entered description of this model. Enter or edit a description on the Description tab of the Model Properties dialog box. You can view the model description by typing `help 'mymodelname'` at the MATLAB command prompt.
Dirty	If the parameter is `on`, the model has unsaved changes.
FileName	Absolute path where the model is saved.
LastModifiedBy	Name of the user who last saved the model.
LastModifiedDate	Date when the model was last saved.
MetaData	Names and attributes of arbitrary data associated with the model. For more details, see `Simulink.MDLInfo.getMetadata`.
ModifiedByFormat	Format of the `ModifiedBy` parameter. The value can include the tag `%<Auto>`. The Simulink software replaces the tag with the current value of the `USER` environment variable.
ModifiedDateFormat	Format used to generate the value of the `LastModifiedDate` parameter. The value can include the tag `%<Auto>`. Simulink replaces the tag with the current date and time when saving the model.
ModelVersion	The major model version is incremented by the number of releases passed since the model was last saved. The minor model version is reset to zero for every new release of Simulink and is incremented by one each time you save the model within the same release.
ModelVersionFormat	The value contains the model format version as `%<AutoIncrement:#.#>` where # is an integer. Simulink increments # when saving the model.
PreviousFileName	When a `PreSaveFcn` or `PostSaveFcn` callback is running, `PreviousFileName` indicates the absolute path of the model before the save operation started. To find the current absolute path of the model, use `FileName` instead.
SavedSinceLoaded	Indicates whether the model has been saved since it was loaded. `'on'` indicates the model has been saved.
VersionLoaded	Simulink version that last saved the model, e.g., `'7.6'`.

`LibraryVersion` is a block parameter for a linked block. `LibraryVersion` is the `ModelVersion` of the library at the time the link was created.

For source control version information, see instead “Project Management”.

See Also

Model Info

Related Examples

- “Project Management”
- “Model File Change Notification” on page 4-57
- `Simulink.MDLInfo`

Model Discretizer

In this section...

What Is the Model Discretizer?

Model Discretizer selectively replaces continuous Simulink blocks with discrete equivalents. Discretization is a critical step in digital controller design and for hardware in-the-loop simulations.

You can use the Model Discretizer to:

- Identify a model's continuous blocks
- Change a block's parameters from continuous to discrete
- Apply discretization settings to all continuous blocks in the model or selected blocks
- Create configurable subsystems that contain multiple discretization candidates along with the original continuous block(s)
- Switch among the different discretization candidates and evaluate the resulting model simulations

The Model Discretizer does not support masked subsystems.

Requirements

To use Model Discretizer

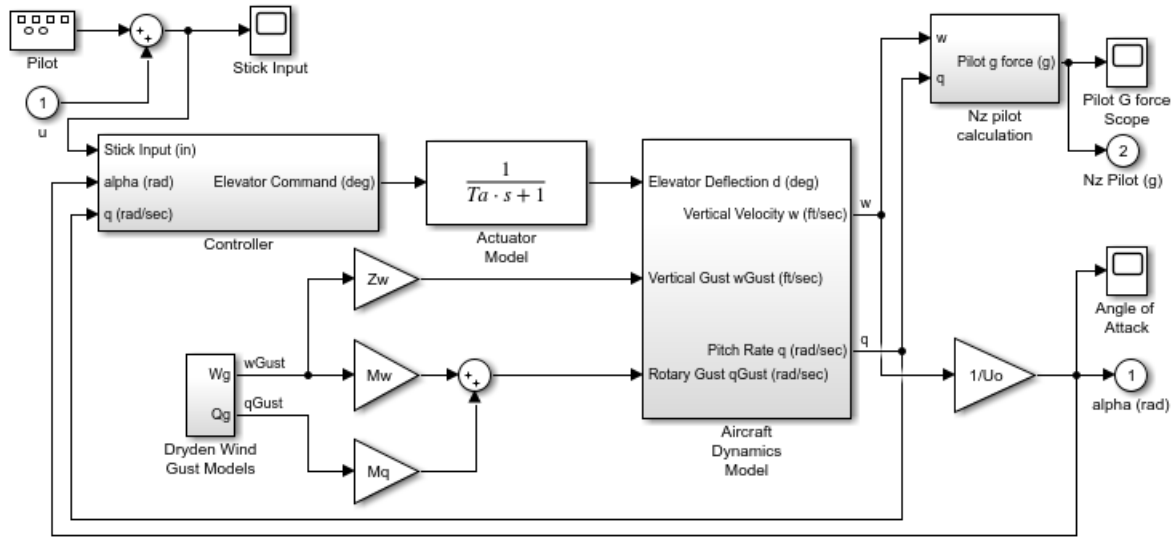
- You must have a Control System Toolbox™ license, Version 5.2 or later.
- Make sure your model does not contain any obsolete blocks and is upgraded to the current Simulink version. For more information, see “Model Upgrades”

Discretize a Model with the Model Discretizer

To discretize a model:

- Start the Model Discretizer
- Specify the Transform Method
- Specify the Sample Time
- Specify the Discretization Method
- Discretize the Blocks

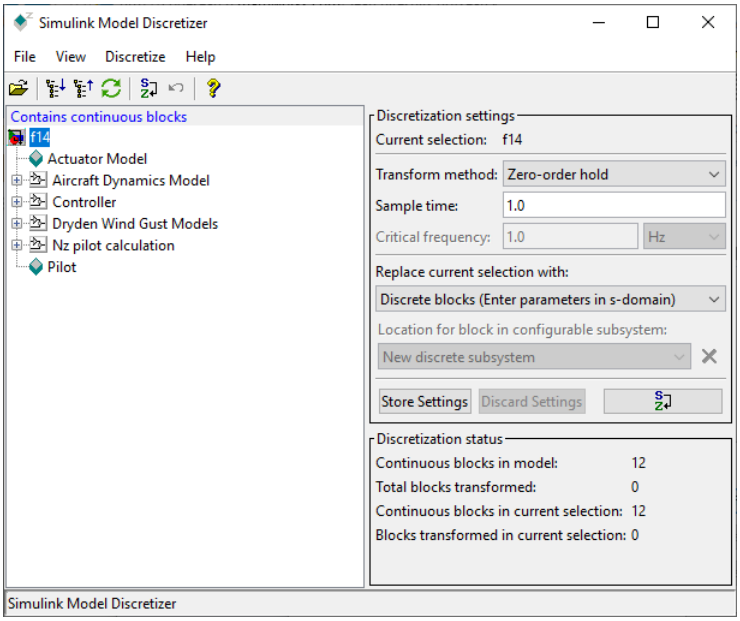
The `f14` model shows the steps in discretizing a model.



Start Model Discretizer

To open the tool, in the Simulink Editor, on the **Apps** tab, under **Apps**, under **Control Systems**, click **Model Discretizer**.

The **Simulink Model Discretizer** opens.



Alternatively, you can open Model Discretizer from the MATLAB Command Window using the `slmdldiscui` function.

The following command opens the **Simulink Model Discretizer** window with the `f14` model:

```
slmdldiscui('f14')
```

To open a new model or library from Model Discretizer, select **File** > **Load model**.

Specify the Transform Method

The transform method specifies the type of algorithms used in the discretization. For more information on the different transform methods, see the Control System Toolbox.

The **Transform method** list contains the following options:

- `Zero-order hold`

  Zero-order hold on the inputs.
- `First-order hold`

  Linear interpolation of inputs.
- `Tustin`

  Bilinear (Tustin) approximation.
- `Tustin with prewarping`

  Tustin approximation with frequency prewarping.
- `Matched pole-zero`

  Matched pole-zero method (for SISO systems only).

Specify the Sample Time

Enter the sample time in the **Sample time** field. For the Model Discretizer, this value must be numeric.

You can specify an offset time by entering a two-element vector for discrete blocks or configurable subsystems. The first element is the sample time and the second element is the offset time. For example, an entry of [1.0 0.1] would specify a 1.0 second sample time with a 0.1 second offset. If no offset is specified, the default is zero.

You can enter workspace variables when discretizing blocks in the s-domain. See “Discrete blocks (Enter parameters in s-domain)” on page 4-65.

Specify the Discretization Method

Specify the discretization method in the **Replace current selection with** field. The options are

- “Discrete blocks (Enter parameters in s-domain)” on page 4-65

  Creates a discrete block whose parameters are retained from the corresponding continuous block.
- “Discrete blocks (Enter parameters in z-domain)” on page 4-65

  Creates a discrete block whose parameters are “hard-coded“ values placed directly into the block's dialog.
- “Configurable subsystem (Enter parameters in s-domain)” on page 4-66

  Create multiple discretization candidates using s-domain values for the current selection. A configurable subsystem can consist of one or more blocks.
- “Configurable subsystem (Enter parameters in z-domain)” on page 4-66

Create multiple discretization candidates in z-domain for the current selection. A configurable subsystem can consist of one or more blocks.

Discrete blocks (Enter parameters in s-domain)

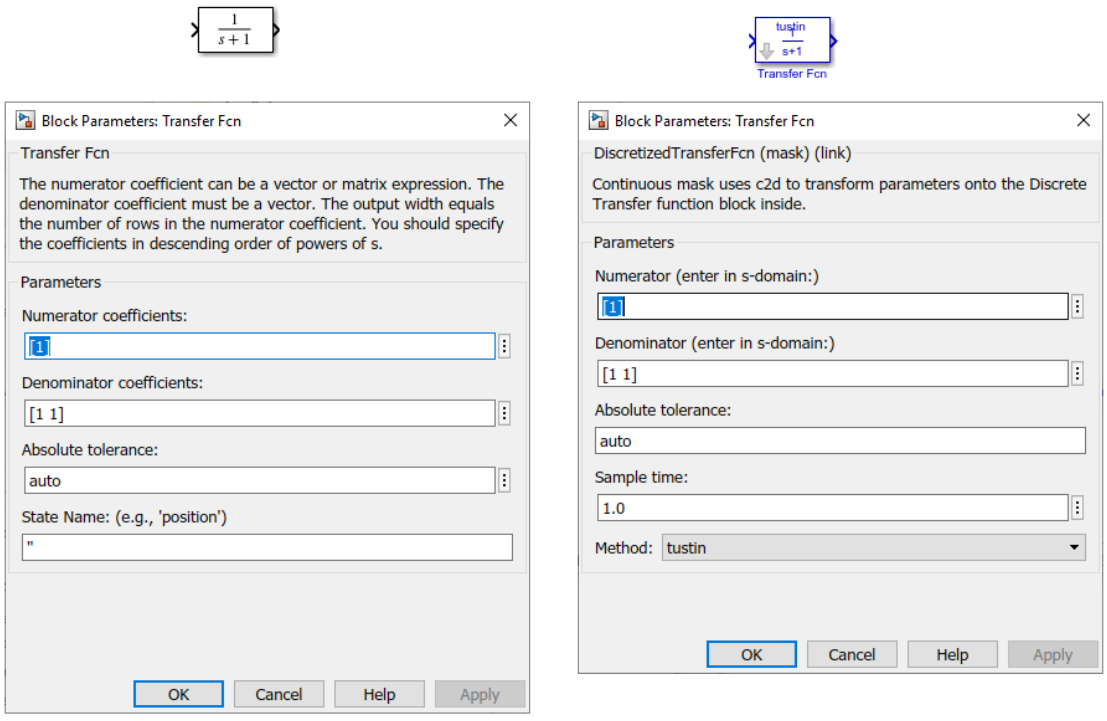
Creates a discrete block whose parameters are retained from the corresponding continuous block. The sample time and the discretization parameters are also on the block's parameter dialog box.

The block is implemented as a masked discrete block that uses `c2d` to transform the continuous parameters to discrete parameters in the mask initialization code.

These blocks have the unique capability of reverting to continuous behavior if the sample time is changed to zero. Entering the sample time as a workspace variable (`'Ts'`, for example) allows for easy changeover from continuous to discrete and back again. See “Specify the Sample Time” on page 4-64.

Note If you generated code from a model, parameters are not tunable when **Default parameter behavior** is set to `Inlined` in the model's Configuration Parameters dialog box.

The following figure shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the s-domain with the Tustin transform method. The block parameters dialog box for each block appears below the block.



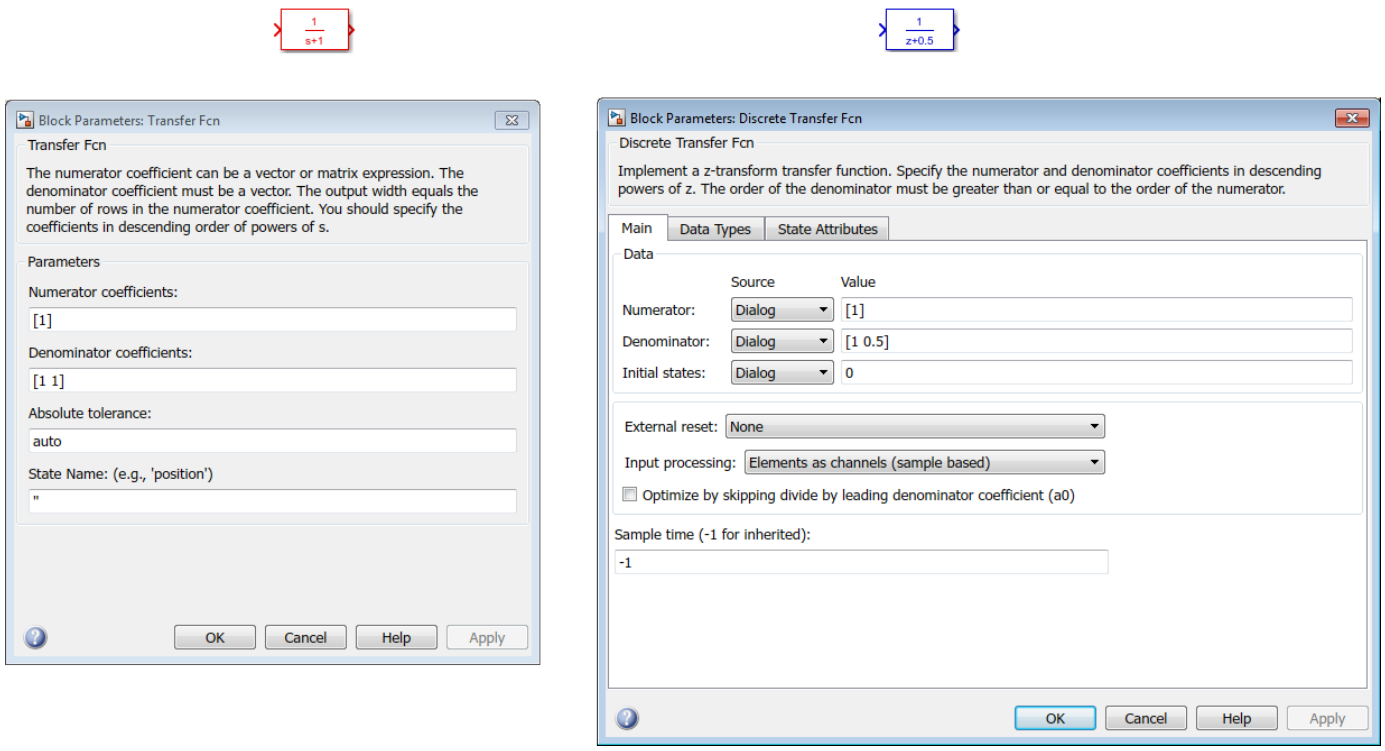
Discrete blocks (Enter parameters in z-domain)

Creates a discrete block whose parameters are “hard-coded” values placed directly into the block's dialog box. Model Discretizer uses the `c2d` function to obtain the discretized parameters, if needed.

For more help on the `c2d` function, type the following in the Command Window:

```
help c2d
```

The following figure shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the z-domain. The block parameters dialog box for each block appears below the block.



Note If you want to recover exactly the original continuous parameter values after the Model Discretization session, you should enter parameters in the s-domain.

Configurable subsystem (Enter parameters in s-domain)

Create multiple discretization candidates using s-domain values for the current selection. A configurable subsystem can consist of one or more blocks.

The **Location for block in configurable subsystem** field becomes active when this option is selected. This option allows you to either create a new configurable subsystem or overwrite an existing one.

Note The current folder must be writable in order to save the library or libraries for the configurable subsystem option.

Configurable subsystem (Enter parameters in z-domain)

Create multiple discretization candidates in z-domain for the current selection. A configurable subsystem can consist of one or more blocks.

The **Location for block in configurable subsystem** field becomes active when this option is selected. This option allows you to either create a new configurable subsystem or overwrite an existing one.

Note The current folder must be writable in order to save the library or libraries for the configurable subsystem option.

Configurable subsystems are stored in a library containing the discretization candidates and the original continuous block. The library will be named `<model name>_disc_lib` and it will be stored in the current. For example a library containing a configurable subsystem created from the `f14` model will be named `f14_disc_lib`.

If multiple libraries are created from the same model, then the filenames will increment accordingly. For example, the second configurable subsystem library created from the `f14` model will be named `f14_disc_lib2`.

You can open a configurable subsystem library by right-clicking on the subsystem in the model and selecting **Library Link > Go to library block** from the context menu.

Discretize the Blocks

To discretize blocks that are linked to a library, you must either discretize the blocks in the library itself or disable the library links in the model window.

You can open the library from Model Discretizer by selecting **Load model** from the **File** menu.

You can disable the library links by right-clicking on the block and selecting **Library Link > Disable Link** from the context menu.

There are two methods for discretizing blocks.

Select Blocks and Discretize

1. Select a block or blocks in the Model Discretizer tree view pane.

   To choose multiple blocks, press and hold the **Ctrl** button on the keyboard while selecting the blocks.

   Note You must select blocks from the Model Discretizer tree view. Clicking blocks in the editor does not select them for discretization.

2. Select **Discretize current block** from the **Discretize** menu if a single block is selected or select **Discretize selected blocks** from the **Discretize** menu if multiple blocks are selected.

   You can also discretize the current block by clicking the **Discretize** button, shown below.

Store the Discretization Settings and Apply Them to Selected Blocks in the Model

1. Enter the discretization settings for the current block.
2. Click **Store Settings**.

This adds the current block with its discretization settings to the group of preset blocks.

3. Repeat steps 1 and 2, as necessary.
4. Select **Discretize preset blocks** from the **Discretize** menu.

Deleting a Discretization Candidate from a Configurable Subsystem

You can delete a discretization candidate from a configurable subsystem by selecting it in the **Location for block in configurable subsystem** field and clicking the **Delete** button.

Undoing a Discretization

To undo a discretization, click the **Undo discretization** button.

Alternatively, you can select **Undo discretization** from the **Discretize** menu.

This operation undoes discretizations in the current selection and its children. For example, performing the undo operation on a subsystem will remove discretization from all blocks in all levels of the subsystem's hierarchy.

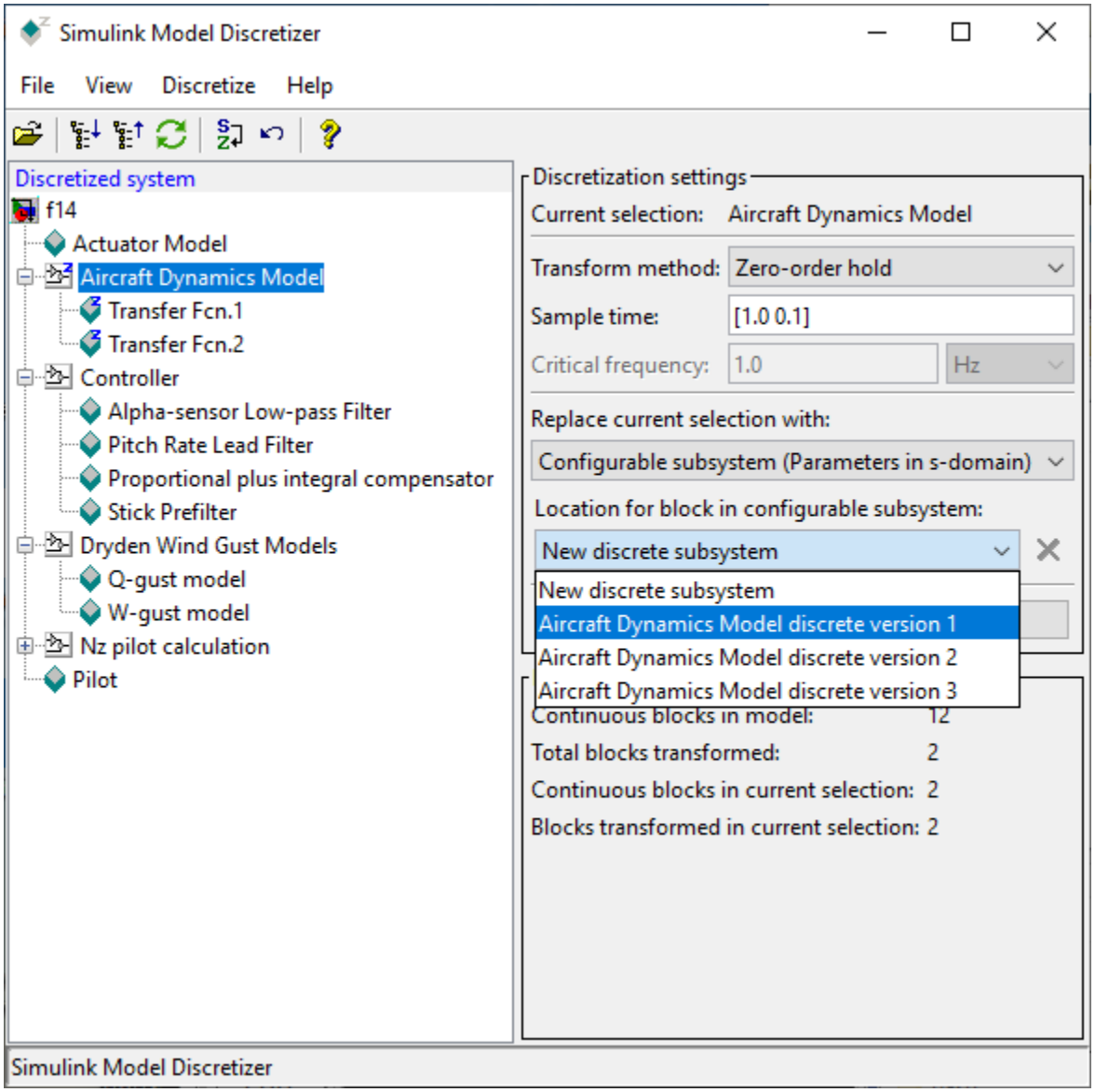
View the Discretized Model

Model Discretizer displays the model in a hierarchical tree view.

Viewing Discretized Blocks

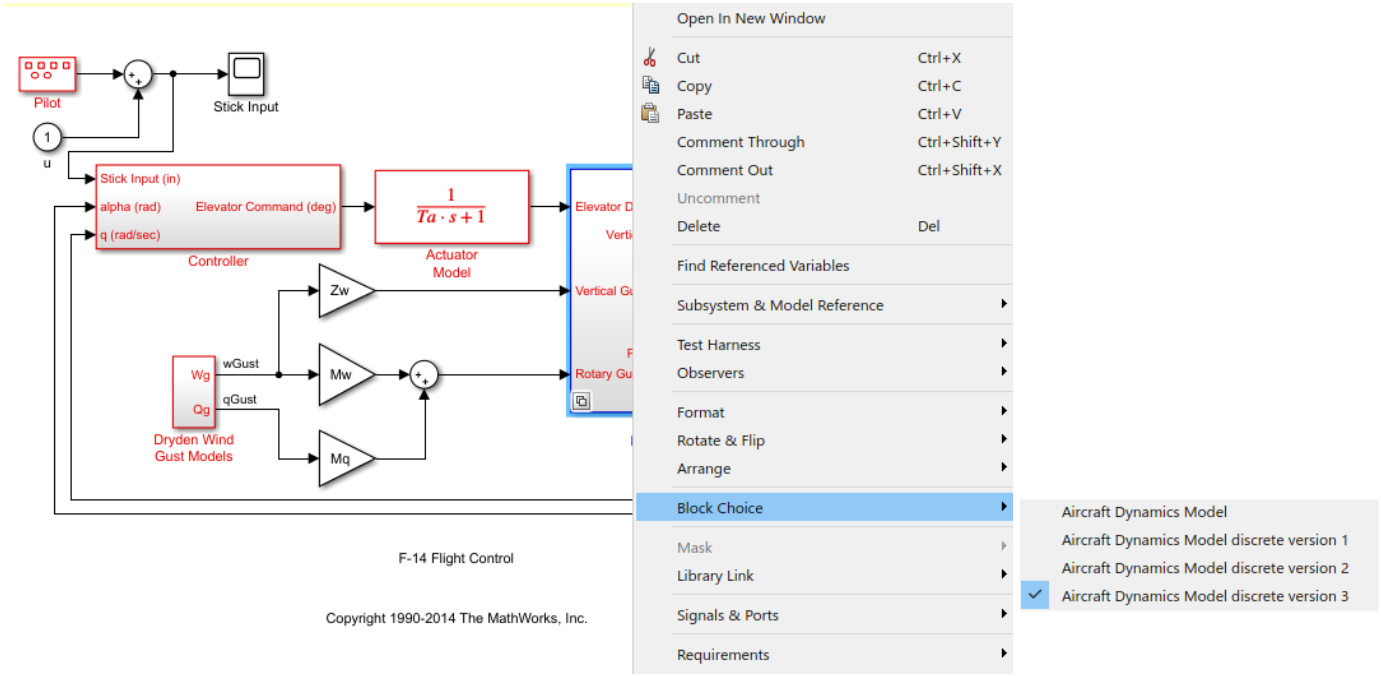
The block's icon in the tree view becomes highlighted with a "**z**" when the block has been discretized.

The following figure shows that the Aircraft Dynamics Model subsystem has been discretized into a configurable subsystem with three discretization candidates.

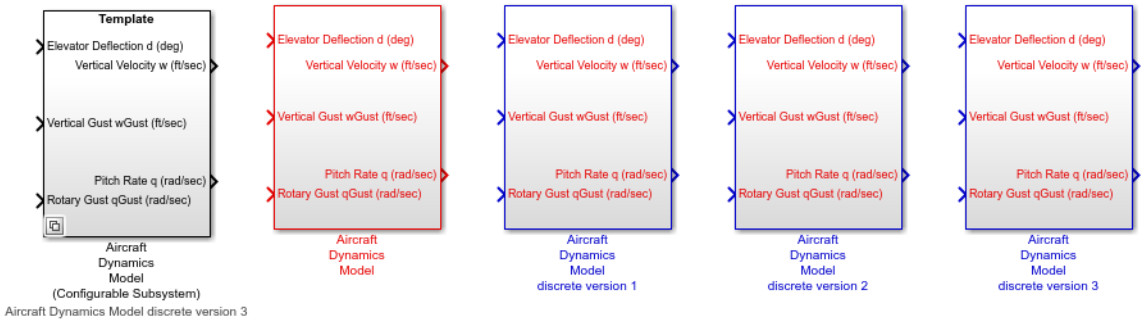


The other blocks in this `f14` model have not been discretized.

The following figure shows the Aircraft Dynamics Model subsystem of the `f14` example model after discretization into a configurable subsystem containing the original continuous model and three discretization candidates.



The following figure shows the library containing the Aircraft Dynamics Model configurable subsystem with the original continuous model and three discretization candidates.



Refreshing Model Discretizer View of the Model

To refresh the Model Discretizer tree view of the model when the model has been changed, click the **Refresh** button.

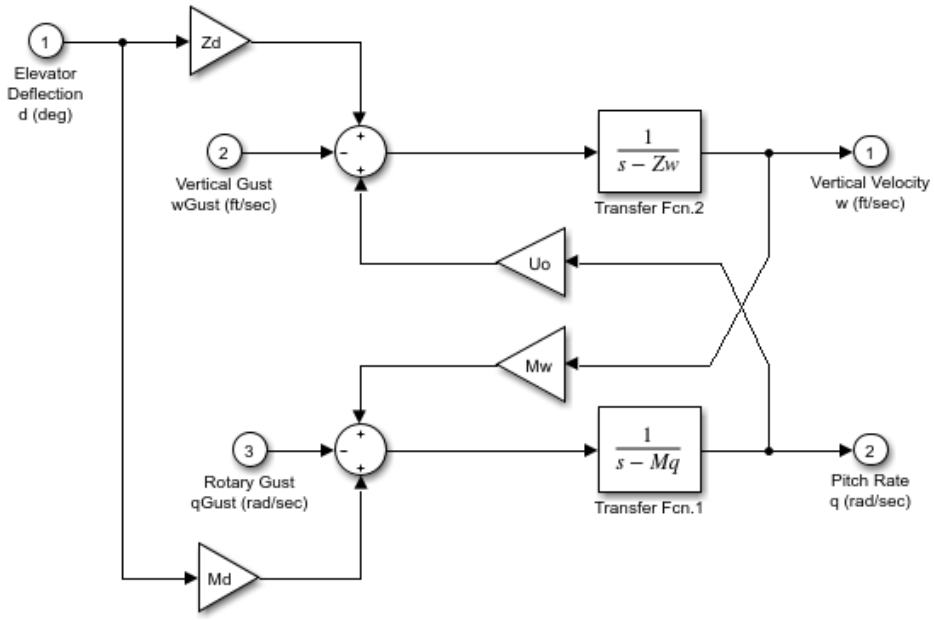
Alternatively, you can select **View** > **Refresh**.

Discretize Blocks from the Simulink Model

You can replace continuous blocks in a Simulink software model with the equivalent blocks discretized in the s-domain using the Discretizing library.

The procedure below shows how to replace a continuous Transfer Fcn block in the Aircraft Dynamics Model subsystem of the `f14` model with a discretized Transfer Fcn block from the Discretizing Library. The block is discretized in the s-domain with a zero-order hold transform method and a two second sample time.

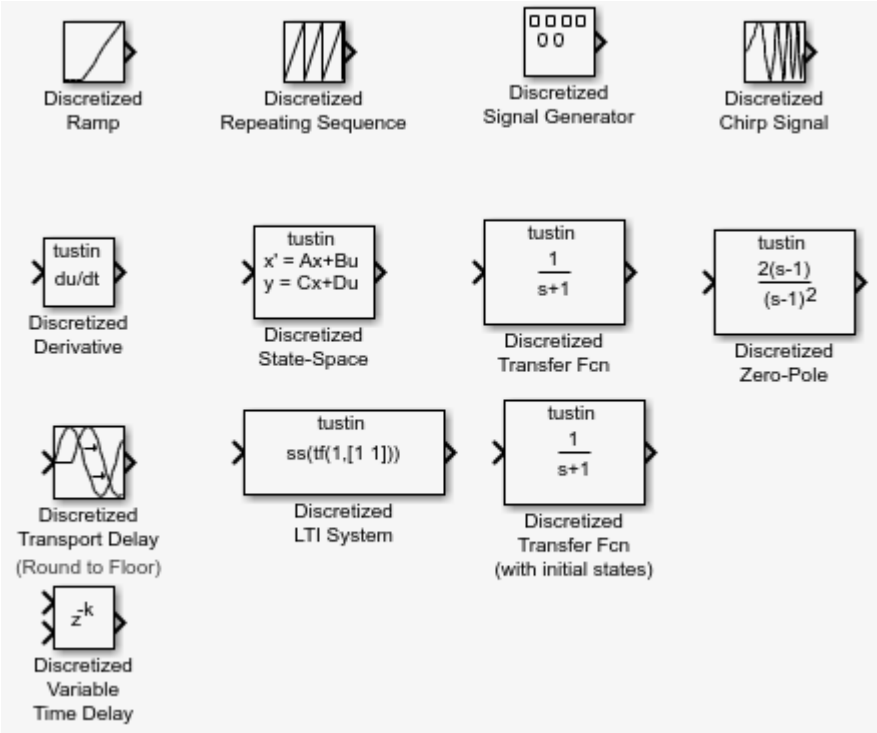
1. Open the `f14` model.
2. Open the Aircraft Dynamics Model subsystem in the `f14` model.



3. Open the Discretizing library window.

   Enter `discretizing` at the MATLAB command prompt.

   The **Library: discretizing** window opens.

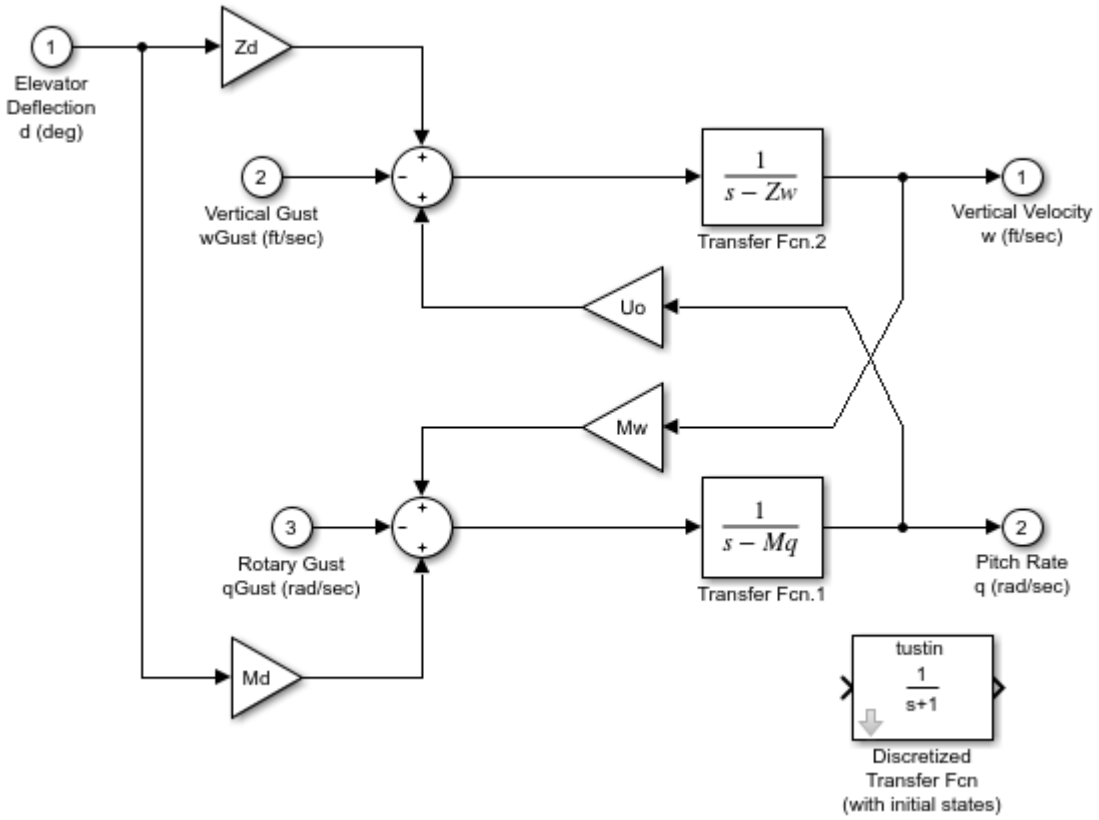


This library contains s-domain discretized blocks.

4 Add the Discretized Transfer Fcn (with initial states) block to the **f14/Aircraft Dynamics Model** window.

   a Click the Discretized Transfer Fcn block in the **Library: discretizing** window.

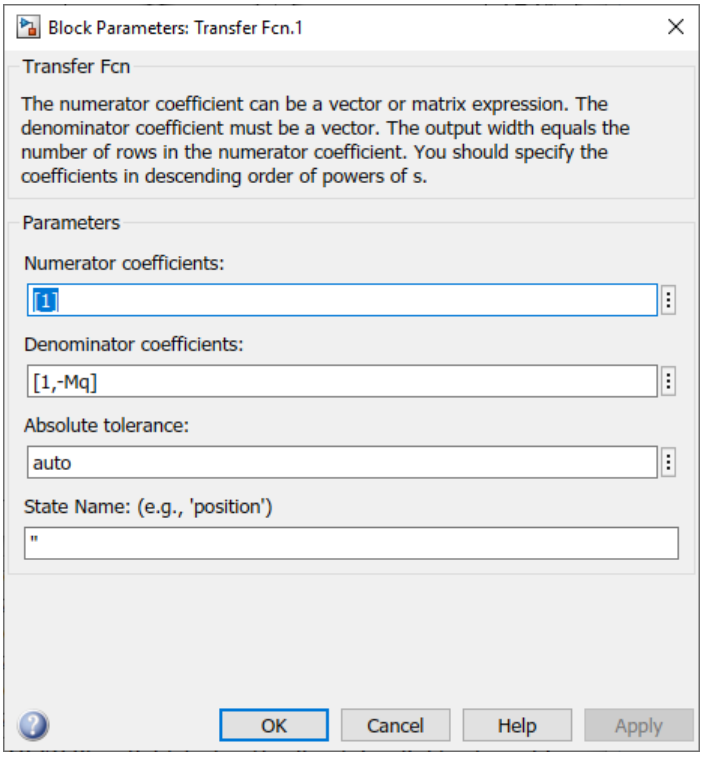
   b Drag it into the **f14/Aircraft Dynamics Model** window.



5 Open the parameter dialog box for the Transfer Fcn.1 block.

   Double-click the Transfer Fcn.1 block in the **f14/Aircraft Dynamics Model** window.

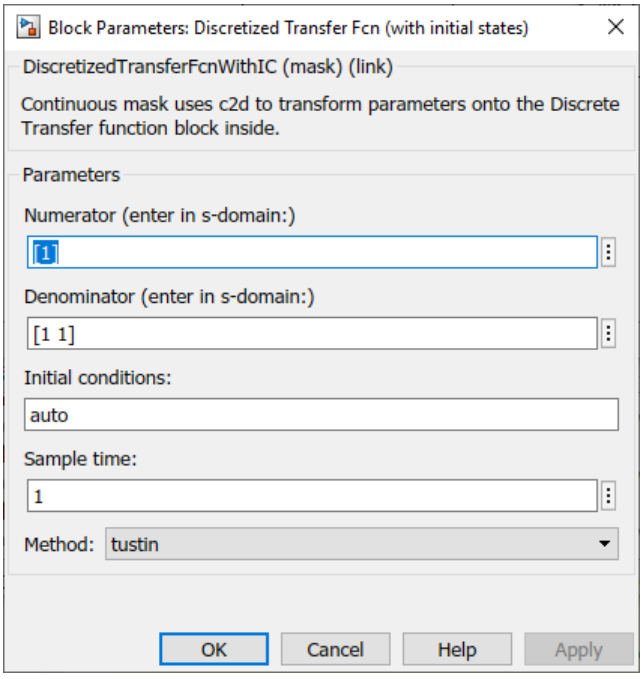
   The Block Parameters: Transfer Fcn.1 dialog box opens.



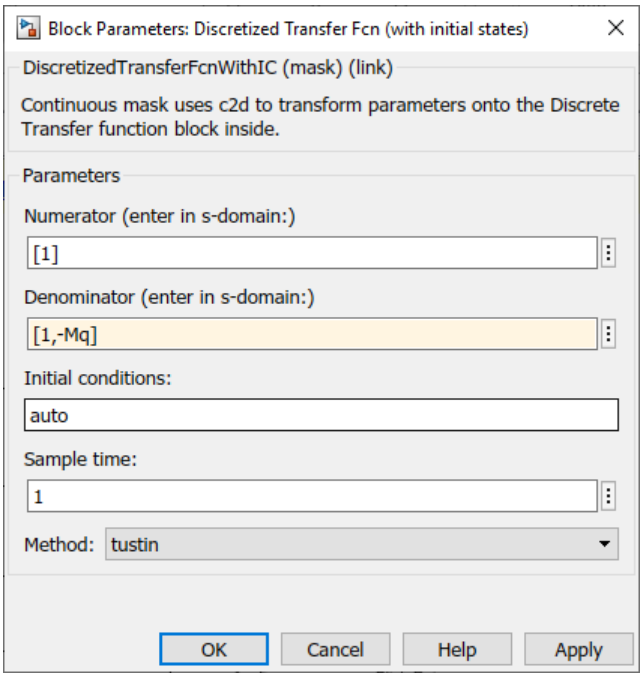
6 Open the parameter dialog box for the Discretized Transfer Fcn block.

Double-click the Discretized Transfer Fcn block in the **f14/Aircraft Dynamics Model** window.

The Block Parameters: Discretized Transfer Fcn dialog box opens.



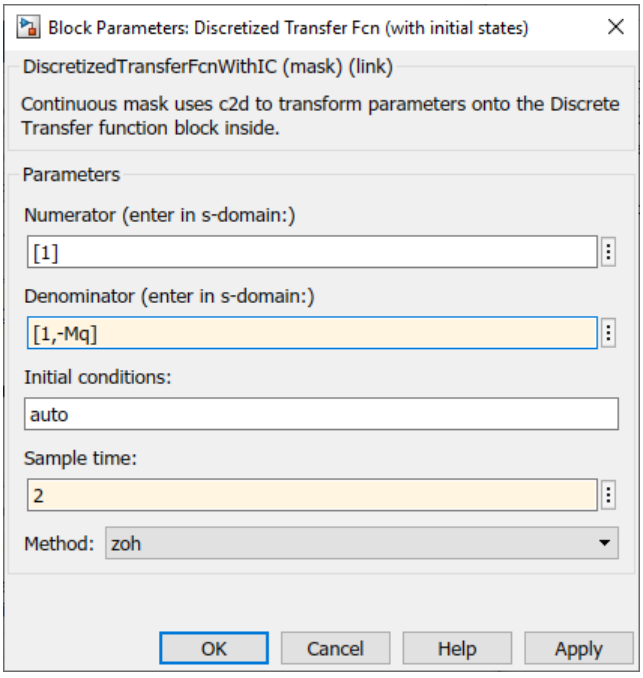
Copy the parameter information from the Transfer Fcn.1 block dialog box to the Discretized Transfer Fcn block's dialog box.



7 Enter 2 in the **Sample time** field.

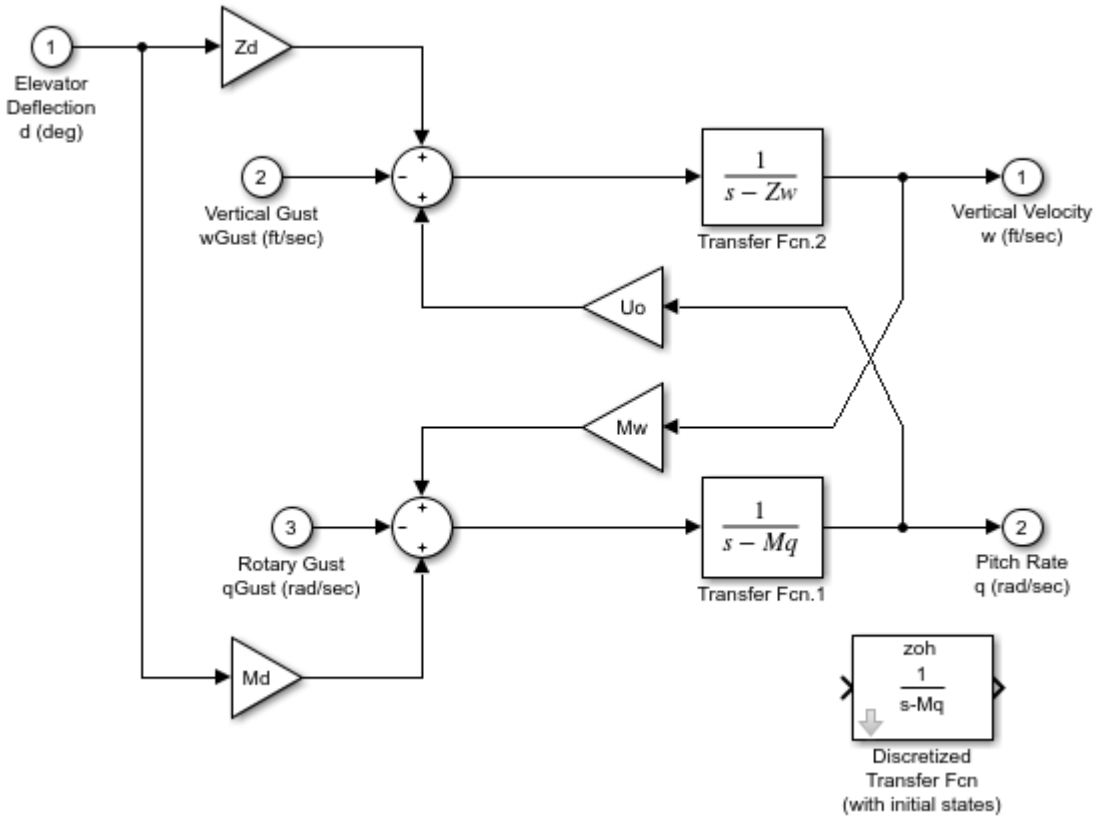
8 Select `zoh` from the **Method** dropdown list.

The parameter dialog box for the Discretized Transfer Fcn now looks like this.



9 Click **OK**.

The **f14/Aircraft Dynamics Model** window now looks like this.

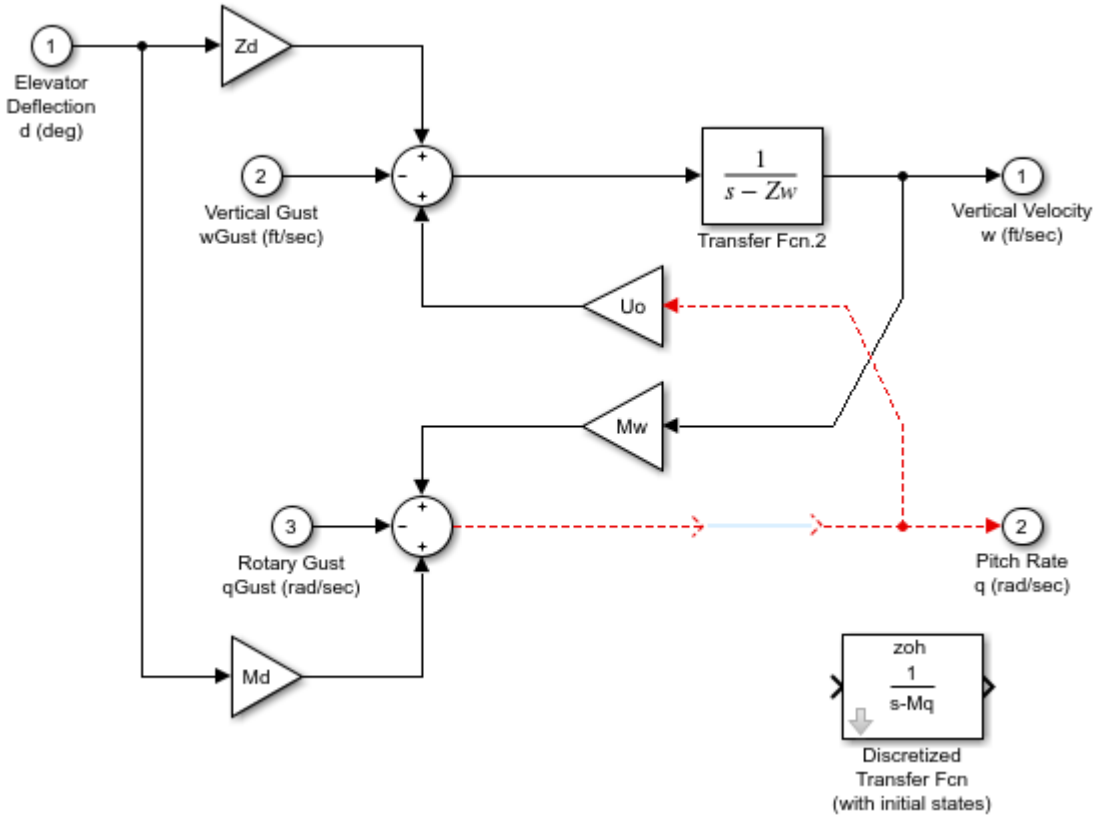


10 Delete the original Transfer Fcn.1 block.

a Click the Transfer Fcn.1 block.

b Press the **Delete** key.

The **f14/Aircraft Dynamics Model** window now looks like this.

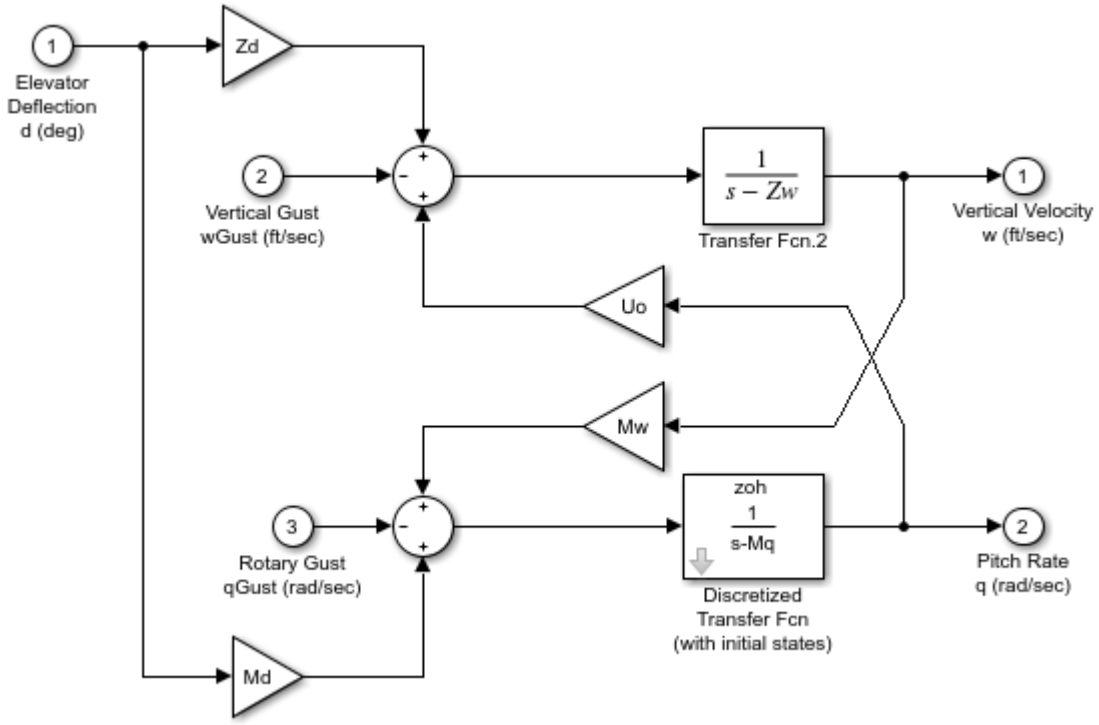


11 Add the Discretized Transfer Fcn block to the model.

a Click the Discretized Transfer Fcn block.

b Drag the Discretized Transfer Fcn block into position to complete the model.

The **f14/Aircraft Dynamics Model** window now looks like this.



Discretize a Model with the sldiscmdl Function

Use the `sldiscmdl` function to discretize Simulink software models from the MATLAB Command Window. You can specify the transform method, the sample time, and the discretization method with the `sldiscmdl` function.

For example, the following command discretizes the `f14` model in the s-domain with a 1-second sample time using a zero-order hold transform method:

```
sldiscmdl('f14',1.0,'zoh')
```

See Also

`sldiscmdl`

Related Examples

- “Discrete blocks (Enter parameters in s-domain)” on page 4-65
- “Discrete blocks (Enter parameters in z-domain)” on page 4-65
- “Configurable subsystem (Enter parameters in s-domain)” on page 4-66
- “Configurable subsystem (Enter parameters in z-domain)” on page 4-66

Model Advisor

Check Your Model Using the Model Advisor

Model Advisor Overview

The Model Advisor checks your model or subsystem for modeling conditions and configuration settings that cause inaccurate or inefficient simulation of the system that the model represents. The Model Advisor checks can help you verify compliance with industry standards and guidelines. By using the Model Advisor, you can implement consistent modeling guidelines across projects and development teams.

Upon completing the analysis of your model, the Model Advisor produces a report that lists the suboptimal conditions, settings, and modeling techniques and proposes solutions, when applicable.

You can use the Model Advisor to check your model in these ways:

- Interactively run Model Advisor checks
- Configure the Model Advisor to automatically run edit-time checks (requires Simulink Check™)

These limitations apply when you use the Model Advisor to check your model. For limitations that apply to specific checks, see the Capabilities and Limitations section in the check documentation.

- If you rename a system, you must restart the Model Advisor to check that system.
- In systems that contain a variant subsystem, the Model Advisor checks the active subsystem. To check both the active and inactive subsystems, set the `Advisor.Application` (Simulink Check) property, `AnalyzeVariants`, to `true`.
- Model Advisor does not analyze commented blocks.
- Checks do not search in model blocks or subsystem blocks with the block parameter **Read/Write** set to `NoReadorWrite`. However, on a check-by-check basis, Model Advisor checks do search in library blocks and masked subsystems.
- Unless specified otherwise in the documentation for a check, the Model Advisor does not analyze the contents of a Model block. To run checks on referenced models, use instances of the `Advisor.Application` class (Simulink Check license required).

Note Software is inherently complex and may not be free of errors. Model Advisor checks might contain bugs. MathWorks® reports known bugs brought to its attention on its Bug Report system at https://www.mathworks.com/support/bugreports/. The bug reports are an integral part of the documentation for each release. Examine bug reports for a release as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

While applying Model Advisor checks to your model increases the likelihood that your model does not violate certain modeling standards or guidelines, their application cannot guarantee that the system being developed will be safe or error-free. It is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include unintended functionality.

Model Advisor Checks Documentation

The Model Advisor only displays the checks for your installed products. This table provides links to the product-specific check documentation. A product license may be required to review some of the documentation.

Product	Model Advisor Check Documentation
Simulink	“Simulink Checks”
Embedded Coder®	“Embedded Coder Checks” (Embedded Coder)
AUTOSAR Blockset	“AUTOSAR Blockset Checks” (AUTOSAR Blockset)
Simulink Coder™	“Simulink Coder Checks” (Simulink Coder)
HDL Coder™	“HDL Code Advisor Checks” (HDL Coder)
Simulink Code Inspector™	“Simulink Code Inspector Checks” (Simulink Code Inspector)
Simulink Check	“DO-178C/DO-331 Checks” (Simulink Check) “IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks” (Simulink Check) “Model Checks for DO-254 Standard Compliance” (Simulink Check) “High Integrity System Modeling Checks” (Simulink Check) “Model Advisor Checks for MAB and JMAAB Compliance” (Simulink Check) “MISRA C:2012 Checks” (Simulink Check) “Secure Coding Checks for CERT C, CWE, and ISO/IEC TS 17961 Standards” (Simulink Check) “Model Metrics” (Simulink Check) “Clone Detection Checks” (Simulink Check)
Simulink Design Verifier™	“Simulink Design Verifier Checks” (Simulink Design Verifier)
Simulink Requirements	“Requirements Consistency Checks” (Simulink Requirements)
Simscape	Documentation is available only in the Model Advisor. To review the documentation for the check, in the Model Advisor, right-click on the check title and select **What's This?**
Simulink Control Design™	“Simulink Control Design Checks” (Simulink Control Design)

Product	Model Advisor Check Documentation
IEC Certification Kit	“IEC Certification Kit Bug Report Checks” (IEC Certification Kit) “High Integrity System Modeling Checks” (Simulink Check)
DO Qualification Kit	“DO Qualification Kit Bug Report Checks” (DO Qualification Kit) “High Integrity System Modeling Checks” (Simulink Check)

Run Model Advisor Checks and Review Results

You can use the Model Advisor to check your model interactively against modeling standards and guidelines. The following example uses the `sldemo_mdladv` model to demonstrate the execution of the Model Advisor checks using the Model Advisor.

1. Open the Model Advisor example model `sldemo_mdladv`.
2. To open the Model Advisor, in the Simulink editor, click the **Modeling** tab and select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**.
3. In the left pane of the Model Advisor, select the checks you want to run on your model:
   a. You can select the checks by using the **By Product** or **By Task** folders. If these folders are not displayed in the Model Advisor window, open **Settings** > **Preferences** and select:
      - **Show By Product Folder** — Displays checks available for each product
      - **Show By Task Folder** — Displays checks related to specific tasks
   b. You can search for and execute a specific check by enter the *Title* or *TitleID* of the check in the **Find:** field and click the **Find Next** button. The Model Advisor searches in check names, folder names, and analysis descriptions. You can use the **Source** tab to identify the *Title*, *TitleID*, and location of the MATLAB source code for each check. To display the **Source** in the right pane of the Model Advisor, open **Settings** > **Preferences** and select **Show Source Tab**.
4. Click on the folder that contains the checks and, on the right pane of the Model Advisor, select:
   - **Show report after run** to automatically generate and display the report in HTML format
   - **Run Selected Checks** to execute the analysis.

   To run a single check, right-click the check in the folder and select **Run This Check**.
5. View the results on the Model Advisor User Interface. Common check status results include
   - `Pass` − Check did not identify issues.
   - `D-Pass` − Dependent on configuration parameter or successful execution of another check.
   - `Warn` − Check has identified issues.
   - `Fail` − Check fails to execute.
6. Fix the warnings or failures as desired. For more information, see “Address Model Check Results” on page 5-9.

7 Use the **Exclusions** tab to review checks that were marked for exclusion from the analysis. To display the **Exclusions** tab in the right pane of the Model Advisor, open **Settings** > **Preferences** and select **Show Exclusion tab**.

8 View and save the report. For additional information, see “Save and View Model Advisor Check Reports” on page 5-13.

Note If you did not select **Show report after run** when you executed the checks, you can generate a report of the results after the analysis is complete. See “Generate Model Advisor Reports” (Simulink Check).

9 If desired, you can reset the status of the checks to the Not Run state. In the left pane, right-click on **Model Advisor** and select `Reset`. This action does not delete the results of the analysis from the Model Advisor.

Save Analysis Time by Running the Checks from a Previous Analysis

You can save time by consistently running the same set of checks on your model by using the Model Advisor dashboard. When you use the dashboard, the Model Advisor does not reload the checks before executing them, saving analysis time.

1 Open the Model Advisor example model `sldemo_mdladv`.

2 Select **Model Advisor** > **Model Advisor Dashboard**. A **System Selector — Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**.

3 The Model Advisor Dashboard window opens. From this dashboard, you can:

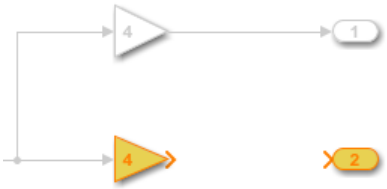
- Click the **Run checks** button to execute the same checks from the previous analysis
- Click the **Switch to standard view** button to open the Model Advisor and select different checks
- Click the **Enable Highlighting** button to view the highlighted results in the Simulink editor

4 Click the **Run checks** button to run the same checks on the model that were used in the previous analysis. If desired, click the **Enable Highlighting** button.

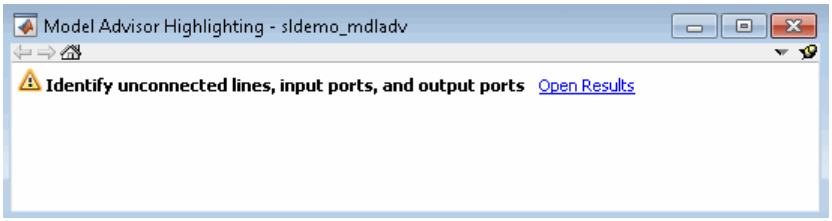
5 The Model Advisor execute the checks and updates the dashboard to reflect the results of the analysis, including the number of:

- Passed checks
- Failed checks
- Flagged checks
- Total checks

If you clicked the **Enable Highlighting** button, the flagged results are highlighted in the model.



The Model Advisor Highlighting information window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.



6 Click the **Open Report** button to open the entire report in HTML format. Alternatively, you can select the number link beside the results to filter the report results.

Run Model Checks Programmatically

If you have Simulink Check, you can create MATLAB scripts and functions so that you can run the Model Advisor programmatically. For example, you can create a `ModelAdvisor.run` function to check whether your model passes a specified set of the Model Advisor checks every time that you open the model and start a simulation.

Access Other Advisors

You can use the Model Advisor window to access other advisors:

- **Upgrade Advisor** — Use this advisor to upgrade and improve models with the current release. See “Consult the Upgrade Advisor” on page 6-2.
- **Code Generation Advisor** — Use this advisor to configure your model to meet code generation objectives. See “Application Objectives Using Code Generation Advisor” (Simulink Coder).
- **Performance Advisor** — Use this advisor to improve the simulation performance of your model. See “Improve Simulation Performance Using Performance Advisor” on page 32-2.

You can access these advisor from the lower left corner of the Model Advisor.

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” (Simulink Check)
- “Address Model Check Results” on page 5-9
- “Save and View Model Advisor Check Reports” on page 5-13
- “Run Model Advisor Checks in Background” on page 5-8

More About

- “Check Your Model Using the Model Advisor” on page 5-2
- “Optimization Tools and Techniques” (Simulink Coder)

Find Model Advisor Check IDs

An *ID* is a unique identifier for a Model Advisor check. 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, in the **Modeling** tab, select **Model Advisor** to open the Model Advisor. **2** In **Settings > Preferences**, select **Show Source Tab**. Click **Apply**. **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.
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.

If you know a check ID from a previous release, you can find the current check ID using the `ModelAdvisor.lookupCheckID` function. For example, the check ID for Check for root Outports with missing range definitions prior to Release 2018b was `mathworks.iec61508.OutportRange`. Using the `ModelAdvisor.lookupCheckID` function returns:

```
>> NewID = ModelAdvisor.lookupCheckID('mathworks.iec61508.OutportRange')

NewID =
'mathworks.hism.hisl_0026'
```

See Also

`ModelAdvisor.lookupCheckID`

Related Examples

- “Run Model Advisor Checks and Review Results” on page 5-4
- “Address Model Check Results” on page 5-9
- “Save and View Model Advisor Check Reports” on page 5-13
- “Run Model Advisor Checks in Background” on page 5-8

More About

- “Check Your Model Using the Model Advisor” on page 5-2

Run Model Advisor Checks in Background

If you have Parallel Computing Toolbox, you can run the Model Advisor in the background, so that you can continue working on your model during analysis. When you start a Model Advisor analysis run in the background, Model Advisor takes a snapshot of your model. The analysis does not reflect changes that you make to your model while Model Advisor is running in the background.

1. Open your model.
2. From the Simulink Editor, on the **Modeling** tab, select **Model Advisor > Model Advisor**. In the System Selector window, select the model or system that you want to review. The Model Advisor opens
3. In the Model Advisor window, click the **Run checks in background** toggle ().
4. In the left pane of the Model Advisor window, select the checks that you want to run.
5. In the Model Advisor window, select **Run selected checks** () button.

   Alternatively, you can use the Model Advisor dashboard to run the checks. In the Model Advisor window, switch to the Model Advisor dashboard by clicking the **Switch to Model Advisor Dashboard** toggle (). On the Model Advisor dashboard, click **Run selected checks** ().

   The Model Advisor starts an analysis on a parallel processor.
6. To stop running checks in the background, in the Model Advisor window, click **Stop background run** (). In the lower-left pane, you see a status of the analysis.

The **Explore Result** option is not available for checks that are run in the background.

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” (Simulink Check)
- “Address Model Check Results” on page 5-9
- “Save and View Model Advisor Check Reports” on page 5-13
- “Find Model Advisor Check IDs” on page 5-7

More About

- “Check Your Model Using the Model Advisor” on page 5-2

Address Model Check Results

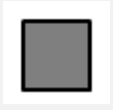
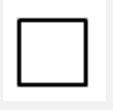
After you run the Model Advisor checks on page 5-4 to find warnings or failures in your model, you can use the Model Advisor to:

- Address Model Advisor check results by using highlighting
- Fix check warnings or failures manually or by using the Model Advisor

Address Model Check Results with Highlighting

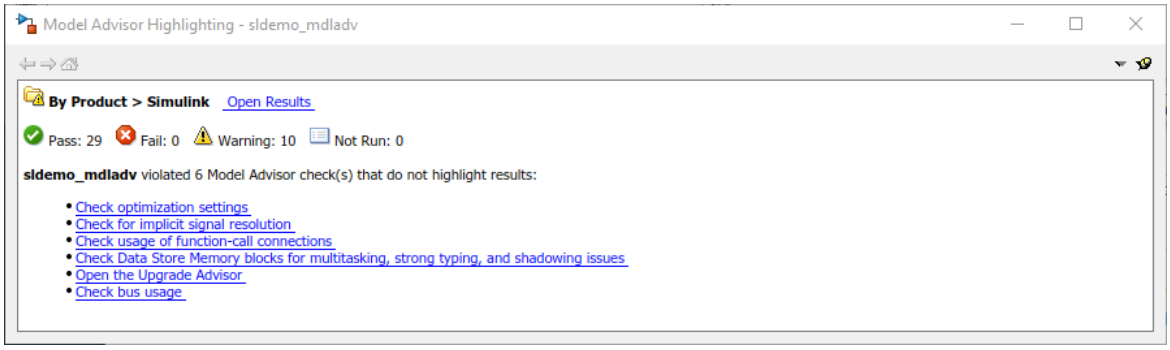
To indicate the analysis results for individual Model Advisor checks, use color highlighting on the model diagram. Highlighting is available for Simulink blocks and Stateflow charts. Blocks that pass a check, fail a check, or cause a check warning are highlighted in color in the model window. On the toolbar of the Model Advisor window, click **Enable highlighting** ( ), select **Highlighting > Enable Highlighting**.

After selecting the highlighting feature, the model window and a Model Advisor Highlighting information window open. The Model Advisor Highlighting information window provides a link to the Model Advisor window where you can review the check results.

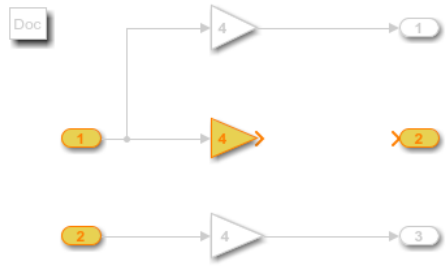
Yellow with orange border		Blocks that cause the check failure or warning.
White with orange border		Subsystem with blocks that cause the check warning or failure.
White with gray border		Blocks or subsystems without highlighting.
Gray with black border		Blocks that are excluded from the check.
White with black border		Subsystems that are excluded from the check.

If a check warns or fails, and the model window highlights blocks in gray, closely examine the results in the Model Advisor window. A Model Advisor check can fail or warn due to your parameter or diagnostic settings.

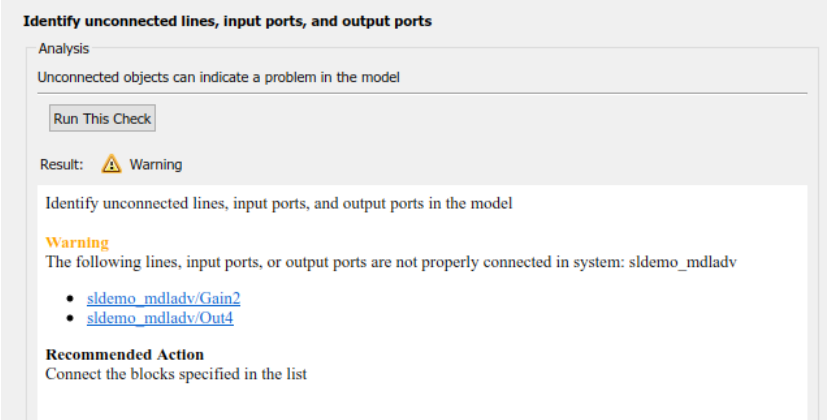
After you run a Model Advisor analysis and select the highlighting feature, checks with highlighted results are indicated with an icon in the Model Advisor window. Highlighting is not available for some checks. Selecting **By Product > Simulink** displays Model Advisor checks in the Model Advisor Highlighting window that do not highlight results.



In the left pane of the Model Advisor window, select the highlighted check **Identify unconnected lines, input ports, and output ports**. In the model editor window, the Model Advisor highlights the blocks or components related to the warning. In this case, the Model Advisor finds a `Gain` block and an `Outport` block not properly connected to the model.



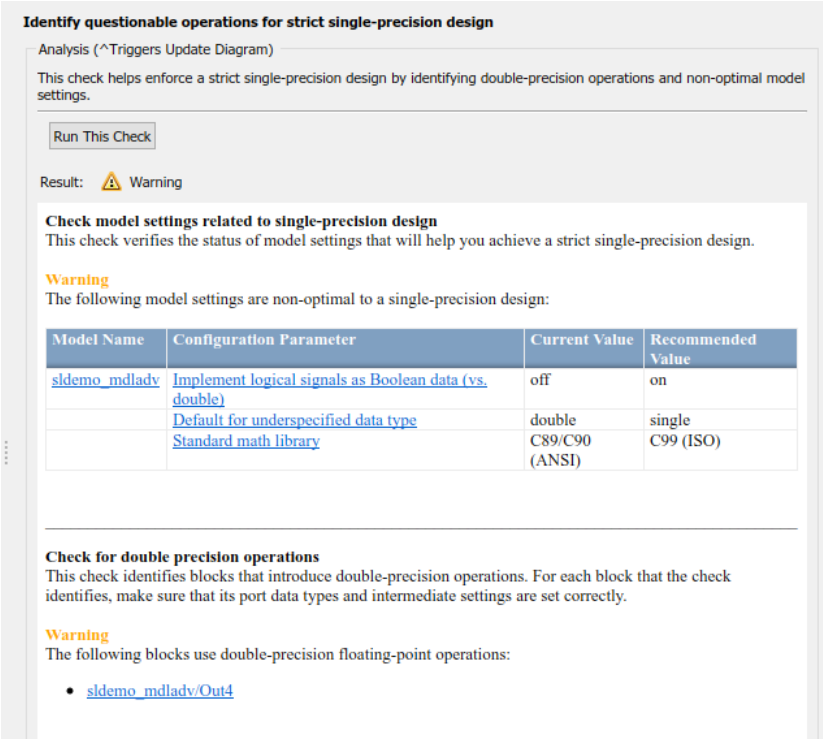
In the right pane of the Model Advisor window, there is further information about the warning.



The **Recommended Action** suggests how to fix the warning or error. In this case, connect the disconnected blocks.

In the left pane of the Model Advisor window, select the highlighted check **Identify questionable operations for strict single-precision design**. In the model editor window, the Model Advisor highlights the blocks or components related to the warning. In this case, the Model Advisor finds an `Outport` block that uses double precision due to a setting of the **Default for underspecified data type** configuration parameter.

In the right pane of the Model Advisor window, you see further detail on the single-precision warning.



The default input of the `Outport` block is set to `double`. Model Advisor generates a warning because the `Outport` block is not connected to another block. After reviewing the check results in the model window and the Model Advisor window, you can choose to fix warnings or failures.

To view model blocks that are excluded from the Model Advisor checks, on the Model Advisor window toolbar, select **Highlighting** > **Highlight Exclusions**. If you have Simulink Check, you can create or modify exclusions to the Model Advisor checks.

Fix a Model Advisor Check Warning or Failure

The Model Advisor check results identify model elements that are being flagged by the Model Advisor check. You can either manually fix the issues or use the Model Advisor to automatically apply a fix.

For more information on why a specific check does not pass, see the documentation for that check.

1. In the Model Advisor results, review the results of the analysis.
2. Fix the warning of failure by using one of these methods:

a To manually fix an issue, use the hyperlinks to open model elements that are being flagged by the Model Advisor check. You can also apply batch changes to model parameters from the command line.

b To use the Model Advisor to automatically fix an issue, in the Action box, click **Modify All** or **Modify**. The Action Result box displays a table of changes.

3 To verify that the check passes, rerun the check.

Note After you finish addressing all warning or failures, it is important that you rerun all checks to verify that there are no additional issues.

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” (Simulink Check)
- “Save and View Model Advisor Check Reports” on page 5-13
- “Find Model Advisor Check IDs” on page 5-7
- “Run Model Advisor Checks in Background” on page 5-8

More About

- “Check Your Model Using the Model Advisor” on page 5-2

Save and View Model Advisor Check Reports

When the Model Advisor runs checks, it generates an HTML report of check results. By default, the HTML report is in the `slprj/modeladvisor/model_name` folder.

If you have Simulink Check, you can generate reports in Adobe® PDF and Microsoft Word .docx formats.

Save Model Advisor Check Reports

The Model Advisor 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 creates it.

You can save a Model Advisor report to a new location.

1. In the Model Advisor window, navigate to the folder with the checks that you ran.
2. Select the folder. The right pane of the Model Advisor window displays information about that folder. The pane includes a **Report** box.
3. In the Report box, click **Generate Report**.
4. In the Generate Model Advisor Report dialog box, enter the path to the folder where you want to generate the report. Provide a file name.
5. Click **OK**. The Model Advisor saves the report in HTML format to the location that you specified.

If you rerun the Model Advisor, the report is updated in the working folder, not in the location where you archived the original report.

The full path to the report is in the title bar of the report window.

View Model Advisor Check Reports

Access a report by selecting a folder and clicking the link in the **Report** box. Or, before a Model Advisor analysis, in the right pane of the Model Advisor window, select **Show report after run**.

Tip Use the options in the Model Advisor window to interactively fix warnings and failures. Model Advisor reports are best for viewing a summary of checks.

As you run checks, the Model Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, an informational message appears in the report. Timestamps indicate when checks have been run. The time of the current run appears at the top right of the report. Checks that occurred during previous runs have a timestamp following the check name.

Goal	Action
Display results for checks that pass, warn, or fail.	Use the **Filter checks** check boxes. For example, to display results for only checks that warn, in the left pane of the report, select the Warning check box. Clear the **Passed**, **Failed**, and **Not Run** check boxes.

Goal	Action
Display results for checks with keywords or phrases in the check title.	Use the **Keywords** field. Results for checks without the keyword in the check title are not displayed in the report. For example, to display results for checks with only “setting” in the check title, in the **Keywords** field, enter “setting”.
Quickly navigate to sections of the report.	Select the links in the table-of-contents navigation pane.
Expand and collapse content in the check results.	Click **Show/Hide check details**.
Scroll to the top of the report.	Click **Scroll to top**.
Minimize folder results in the report.	Click the minus sign next to the folder name.

Printed versions of the report do not contain:

- Filtering checks, Navigation, or View panes.
- Content hidden due to filtering or keyword searching.

Some checks have input parameters specified in the right pane of the Model Advisor. For example, **Check Merge block usage** has an input parameter for **Maximum analysis time (seconds)**. When you run checks with input parameters, the Model Advisor displays the values of the input parameters in the HTML report. For more information, see the `EmitInputParametersToReport` property of the `Simulink.ModelAdvisor` class.

See Also

`Simulink.ModelAdvisor`

Related Examples

- “Run Model Advisor Checks and Review Results” (Simulink Check)
- “Address Model Check Results” on page 5-9
- “Run Model Advisor Checks in Background” on page 5-8

More About

- “Check Your Model Using the Model Advisor” on page 5-2

Upgrade Advisor

Consult the Upgrade Advisor

Use the Upgrade Advisor to help you upgrade and improve models with the current release. 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 help identify cases when 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.

You can open the Upgrade Advisor in the following ways:

- From the Model Editor, on the **Modeling** tab, select **Model Advisor > Upgrade Advisor**
- From the MATLAB command line, use the `upgradeadvisor` function:

  ```
  upgradeadvisor modelname
  ```
- Alternatively, from the Model Advisor, click **Upgrade Advisor**. This action closes the Model Advisor and opens the Upgrade Advisor.

In the Upgrade Advisor, you create reports and run checks in the same way as when using the Model Advisor.

- Select the top Upgrade Advisor node in the left pane to run all selected checks and create a report.
- Select each individual check to open a detailed view of the results in the right pane. View the analysis results for recommended actions to manually fix warnings or failures. In some cases, the Upgrade Advisor provides mechanisms for automatically fixing warnings and failures.

Caution When you fix a warning or failure, rerun all checks to update the results of all checks. If you do not rerun all checks, the Upgrade Advisor might report an invalid check result.

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. To guide you through upgrade checks to run both non-compile and compile checks, run the check **Analyze model hierarchy and continue upgrade sequence**. See “Analyze model hierarchy and continue upgrade sequence”.

For models with no hierarchy, select and run all checks except the **Analyze model hierarchy and continue upgrade sequence** check.

For more information on individual checks, see

- “Model Upgrades” for upgrade checks only
- “Simulink Checks” for all upgrade and advisor checks

Upgrade Programmatically

To analyze and upgrade models programmatically, use the `upgradeadvisor` function.

Tip For an example showing how to upgrade a whole project programmatically, see “Upgrade Simulink Models Using a Project”.

Upgrade Advisor Checks

For advice on upgrading and improving models with the current release, use the following Model Advisor checks in the Upgrade Advisor.

- “Check model for block upgrade issues”
- “Check usage of function-call connections”
- “Identify Model Info blocks that can interact with external source control tools”
- “Check for calls to slDataTypeAndScale”
- “Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter”
- “Identify Variant blocks using Variant objects with empty conditions”
- “Check that the model is saved in SLX format”
- “Check that the model or library is saved in current version”
- “Check model for SB2SL blocks”
- “Check Model History properties”
- “Identify Model Info blocks that use the Configuration Manager”
- “Identify configurable subsystem blocks for converting to variant subsystem blocks”
- “Check and update masked blocks in library to use promoted parameters”
- “Check and update mask image display commands with unnecessary imread() function calls”
- “Check Rapid accelerator signal logging”“Check get_param calls for block CompiledSampleTime”
- “Check model for parameter initialization and tuning issues”
- “Check model for block upgrade issues requiring compile time information”
- “Check usage of Merge blocks”
- “Check usage of Outport blocks”
- “Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition”
- “Check usage of Discrete-Time Integrator blocks”
- “Check model settings for migration to simplified initialization mode”
- “Check model for legacy 3DoF or 6DoF blocks”
- “Check model for Aerospace Blockset navigation blocks”
- “Check for root outports with constant sample time”
- “Analyze model hierarchy and continue upgrade sequence”
- “Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices”
- “Check for case mismatches in references to models and libraries”
- “Check configuration parameters for generation of inefficient saturation code” (Embedded Coder)

See Also

`upgradeadvisor`

Related Examples

- “Check Your Model Using the Model Advisor” on page 5-2
- “Address Model Check Results” on page 5-9

Working with Sample Times

What Is Sample Time?

The sample time of a block is a parameter that indicates when, during simulation, the block produces outputs and if appropriate, updates its internal state. The internal state includes but is not limited to continuous and discrete states that are logged.

Note Do not confuse the Simulink usage of the term sample time with the engineering sense of the term. In engineering, sample time refers to the rate at which a discrete system samples its inputs. Simulink allows you to model single-rate and multirate discrete systems and hybrid continuous-discrete systems through the appropriate setting of block sample times that control the rate of block execution (calculations).

For many engineering applications, you need to control the rate of block execution. In general, Simulink provides this capability by allowing you to specify an explicit `SampleTime` parameter in the block dialog or at the command line. Blocks that do not have a `SampleTime` parameter have an implicit sample time. You cannot specify implicit sample times. Simulink determines them based upon the context of the block in the system. The Integrator block is an example of a block that has an implicit sample time. Simulink automatically sets its sample time to `0`.

Sample times can be port based or block based. For block-based sample times, all of the inputs and outputs of the block run at the same rate. For port-based sample times, the input and output ports can run at different rates. To learn more about rates of execution, see “Types of Sample Time” on page 7-13.

See Also

“Specify Sample Time” on page 7-3 | “Types of Sample Time” on page 7-13 | “Sample Times in Systems” on page 7-23 | “Sample Times in Subsystems” on page 7-22

Specify Sample Time

In this section...

Designate Sample Times

Simulink allows you to specify a block sample time directly as a numerical value or symbolically by defining a sample time vector. In the case of a discrete sample time, the vector is [T_s, T_o] where T_s is the sampling period and T_o is the initial time offset. For example, consider a discrete model that produces its outputs every two seconds. If your base time unit is seconds, you can directly set the discrete sample time by specifying the numerical value of `2` as the `SampleTime` parameter. Because the offset value is zero, you do not need to specify it; however, you can enter `[2,0]` in the **Sample time** field.

For nondiscrete blocks, the components of the vector are symbolic values that represent one of the types in “Types of Sample Time” on page 7-13. The following table summarizes these types and the corresponding sample time values. The table also defines the explicit nature of each sample time type and designates the associated color and annotation. Because an *inherited sample time* is explicit, you can specify it as `[-1, 0]` or as `-1`. Whereas, a triggered sample time is implicit; only Simulink can assign the sample time of `[-1, -1]`. (For more information about colors and annotations, see “View Sample Time Information” on page 7-9.)

Designations of Sample Time Information

Sample Time Type	Sample Time	Color	Annotation	Explicit
Discrete	[T_s, T_o]	In descending order of speed: red, green, blue, light blue, dark green, orange	D1, D2, D3, D4, D5, D6, D7,... Di	Yes
Continuous	[0, 0]	black	Cont	Yes
Fixed in minor step	[0, 1]	gray	FiM	Yes
Inherited	[–1, 0]	N/A	N/A	Yes
Constant	[Inf, 0]	magenta	Inf	Yes
Variable	[–2,T_{vo}]	brown	V1, V2,... Vi	No
Controllable	[base, -2i], i = 0, 1, 2, ...,	brown	Ctrl1, Ctrl2, Ctrl3, ... Ctrli	Yes
Hybrid	N/A	yellow	N/A	No
Triggered	`Source: D`1, Source: D2, ...Source: Di	cyan	T1, T2,... Ti	No
Asynchronous	[–1, –n]	purple	A1, A2,... Ai	No
Dataflow	N/A	light purple	N/A	No

The color that is assigned to each block depends on its sample time relative to other sample times in the model. This means that the same sample time may be assigned different colors in a parent model and in models that it references. (See “Model References”.)

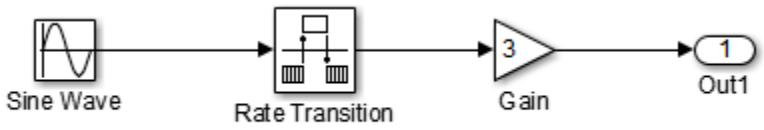
For example, suppose that a model defines three sample times: 1, 2, and 3. Further, suppose that it references a model that defines two sample times: 2 and 3. In this case, blocks operating at the 2 sample rate appear as green in the parent model and as red in the referenced model.

It is important to note that Mux and Demux blocks are simply grouping operators; signals passing through them retain their timing information. For this reason, the lines emanating from a Demux block can have different colors if they are driven by sources having different sample times. In this case, the Mux and Demux blocks are color coded as hybrids (yellow) to indicate that they handle signals with multiple rates.

Similarly, Subsystem blocks that contain blocks with differing sample times are also colored as hybrids, because there is no single rate associated with them. If all the blocks within a subsystem run at a single rate, the Subsystem block is colored according to that rate.

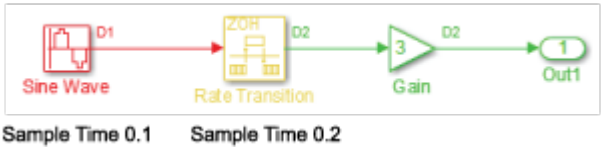
You can use the explicit sample time values in this table to specify sample times interactively or programmatically for either block-based or port-based sample times.

The following model, `ex_specify_sample_time`, serves as a reference for this section.



ex_specify_sample_time

In this example, set the sample time of the input sine wave signal to `0.1`. The goal is to achieve an output sample time of `0.2`. The Rate Transition block serves as a zero-order hold. The resulting block diagram after setting the sample times and simulating the model is shown in the following figure. (The colors and annotations indicate that this is a discrete model.)



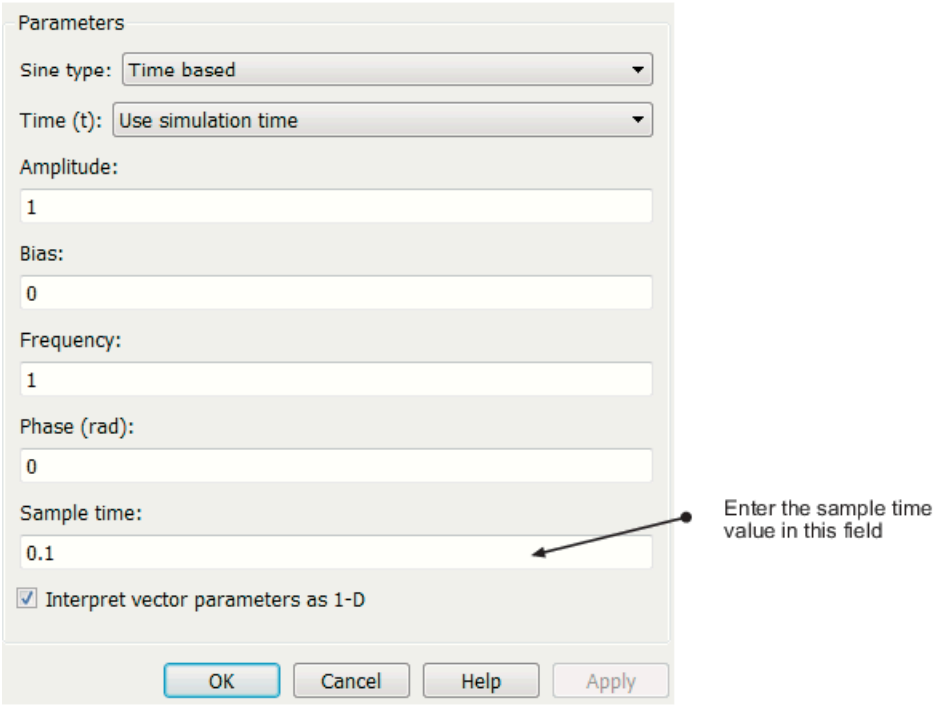
ex_specify_sample_time after Setting Sample Times

Specify Block-Based Sample Times Interactively

To set the sample time of a block interactively:

1. In the Simulink model window, double-click the block. The block parameter dialog box opens.
2. Enter the sample time in the **Sample time** field.
3. Click **OK**.

Following is a figure of a parameters dialog box for the Sine Wave block after entering `0.1` in the **Sample time** field.

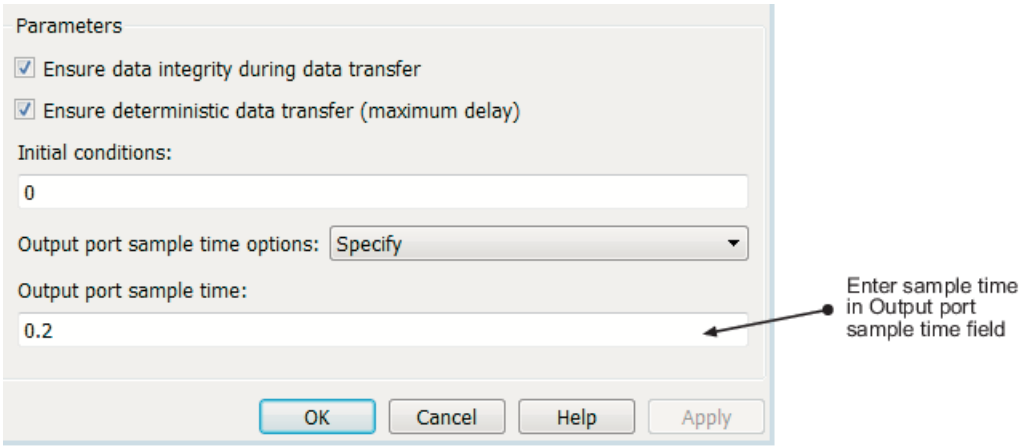


To specify and inspect block-based sample times throughout a model, consider using the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). On the **Inports/Outports**, **Signals**, and **Data Stores** tabs, set the **Change view** drop-down list to `Design` and use the **Sample Time** column. For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Specify Port-Based Sample Times Interactively

The Rate Transition block has port-based sample times. You can set the output port sample time interactively by completing the following steps:

1. Double-click the Rate Transition block. The parameters dialog box opens.
2. Leave the drop-down menu choice of the **Output port sample time options** as `Specify`.
3. Replace the `-1` in the **Output port sample time** field with `0.2`.



4 Click **OK**.

For more information about the sample time options in the Rate Transition parameters dialog box, see the Rate Transition reference page.

Specify Block-Based Sample Times Programmatically

To set a block sample time programmatically, set its `SampleTime` parameter to the desired sample time using the `set_param` command. For example, to set the sample time of the Gain block in the Specify_Sample_Time model to inherited (`-1`), enter the following command:

```
set_param('Specify_Sample_Time/Gain','SampleTime','[-1, 0]')
```

As with interactive specification, you can enter just the first vector component if the second component is zero.

```
set_param('Specify_Sample_Time/Gain','SampleTime','-1')
```

Specify Port-Based Sample Times Programmatically

To set the output port sample time of the Rate Transition block to `0.2`, use the `set_param` command with the parameter `OutPortSampleTime`:

```
set_param('Specify_Sample_Time/Rate Transition',...
'OutPortSampleTime', '0.2')
```

Access Sample Time Information Programmatically

To access all sample times associated with a model, use the API `Simulink.BlockDiagram.getSampleTimes`.

To access the sample time of a single block, use the API `Simulink.Block.getSampleTimes`.

Specify Sample Times for a Custom Block

You can design custom blocks so that the input and output ports operate at different sample time rates. For information on specifying block-based and port-based sample times for S-functions, see “Specify S-Function Sample Times” in *Writing S-Functions* of the Simulink documentation.

Determining Sample Time Units

Since the execution of a Simulink model is not dependent on a specific set of units, you must determine the appropriate base time unit for your application and set the sample time values accordingly. For example, if your base time unit is second, then you would represent a sample time of `0.5` second by setting the sample time to `0.5`.

Change the Sample Time After Simulation Start Time

To change a sample time after simulation begins, you must stop the simulation, reset the `SampleTime` parameter, and then restart execution.

See Also

“What Is Sample Time?” on page 7-2 | “Types of Sample Time” on page 7-13

View Sample Time Information

In this section...

Simulink models can display color coding and annotations that represent specific sample times. Each sample time type has one or more colors associated with it. You can display the blocks and signal lines in color, the annotations in black, or both. To select one of these options:

1. To enable colors, in the Simulink model window, on the **Debug** tab, select **Information Overlays** > **Colors**.
2. To enable annotations, on the **Debug** tab, select **Information Overlays** > **Text**.

Selecting both **Colors** and **Text** displays both the colors and the annotations. Regardless of your choice, Simulink performs an **Update Model** automatically.

To turn off the colors and annotations:

1. On the **Debug** tab, select **Information Overlays** > **Colors** to disable colors.
2. On the **Debug** tab, select **Information Overlays** > **Text** to disable annotations.

Simulink performs another **Update Model** automatically.

Your Sample Time Display choices directly control the information that the Timing Legend displays.

Note The discrete sample times in the table Designations of Sample Time Information represent a special case. Five colors indicate the speed through the fifth fastest discrete rate. A sixth color, orange, represents all rates that are slower than the fifth discrete rate. You can distinguish between these slower rates by looking at the annotations on their respective signal lines.

Inspect Sample Time Using Timing Legend

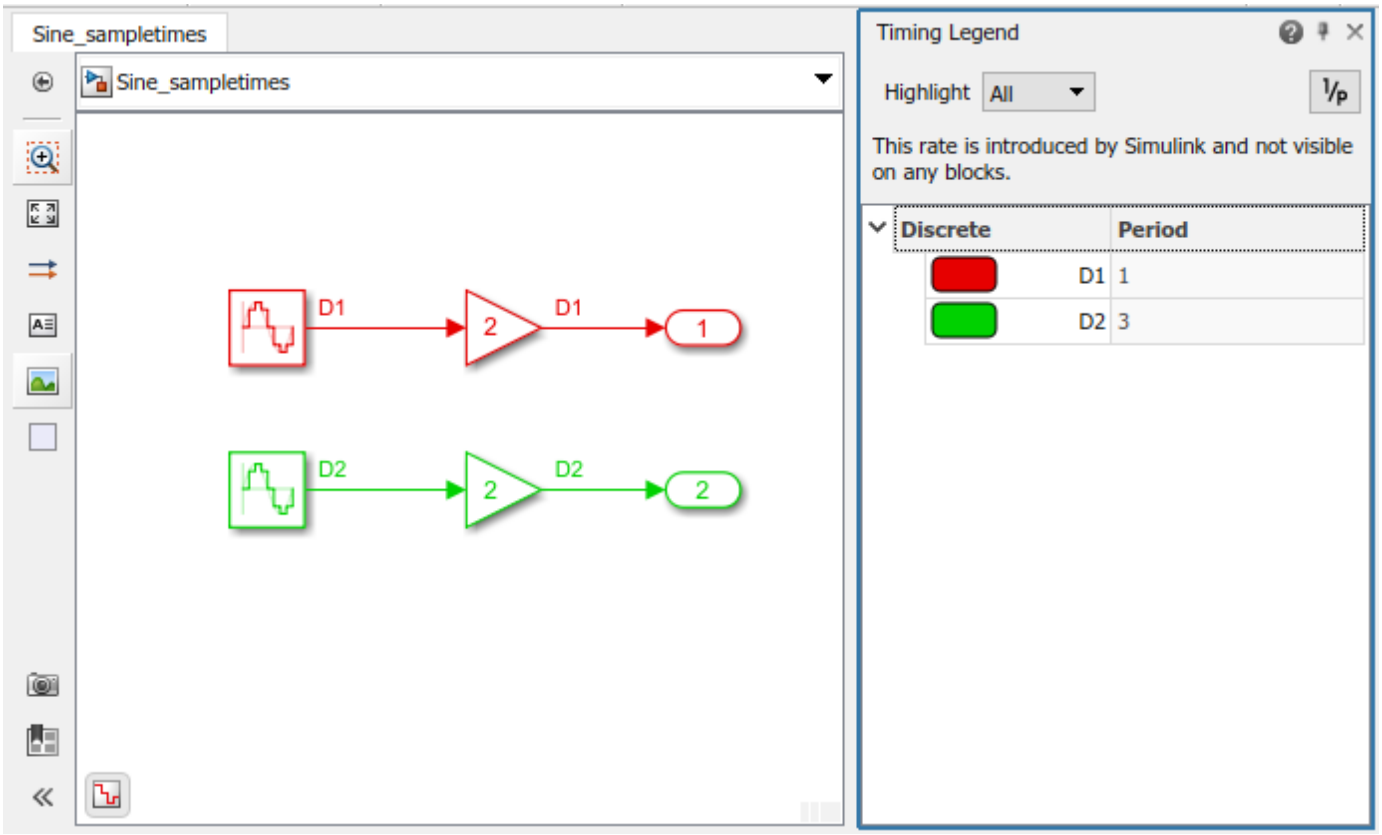
You can view the Timing Legend for an individual model or for multiple models. Additionally, you can prevent the legend from automatically opening when you select options on the **Sample Time** menu.

To assist you with interpreting a block diagram, the Timing Legend contains the sample time color, annotation, and value for each sample time in the model. To view the legend:

1. In the Simulink model window, on the **Modeling** tab, click **Update Model**.
2. On the **Debug** tab, select **Information Overlays** > **Legend**or press **Ctrl + J**.

In addition, when you select **Colors** or **Text**, Simulink updates the model diagram and opens the legend by default. The legend contents reflect your choices. If you turn colors on, the legend displays the color and the value of the sample time. Similarly, if you turn annotations on, the annotations appear in the legend.

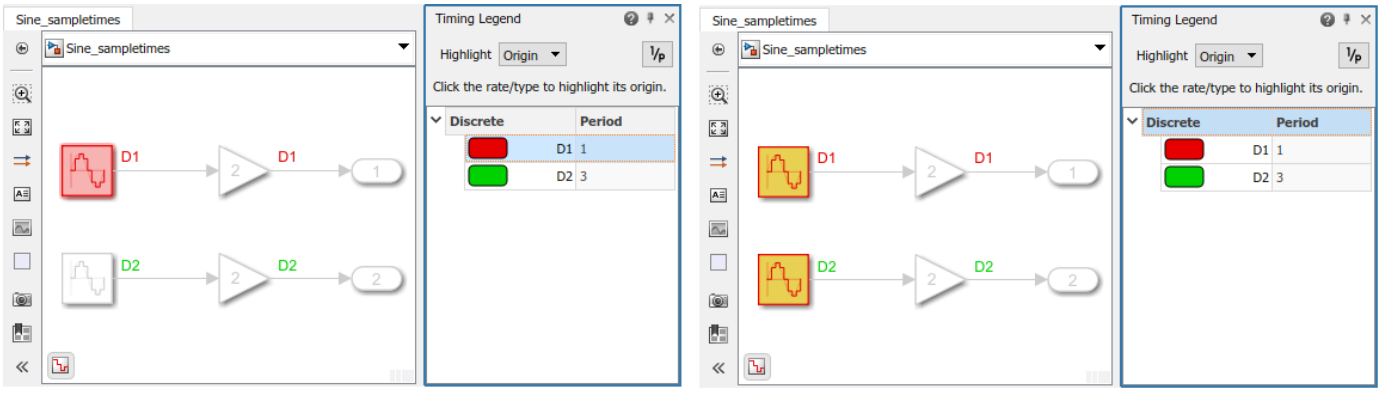
The legend displays sample times present in the model, classified by the type of the sample time.



The legend provides two types of highlighting options:

- Highlighting the blocks and signals that the sample time originates from.
- Highlighting all the blocks and signals that contain the selected sample time.

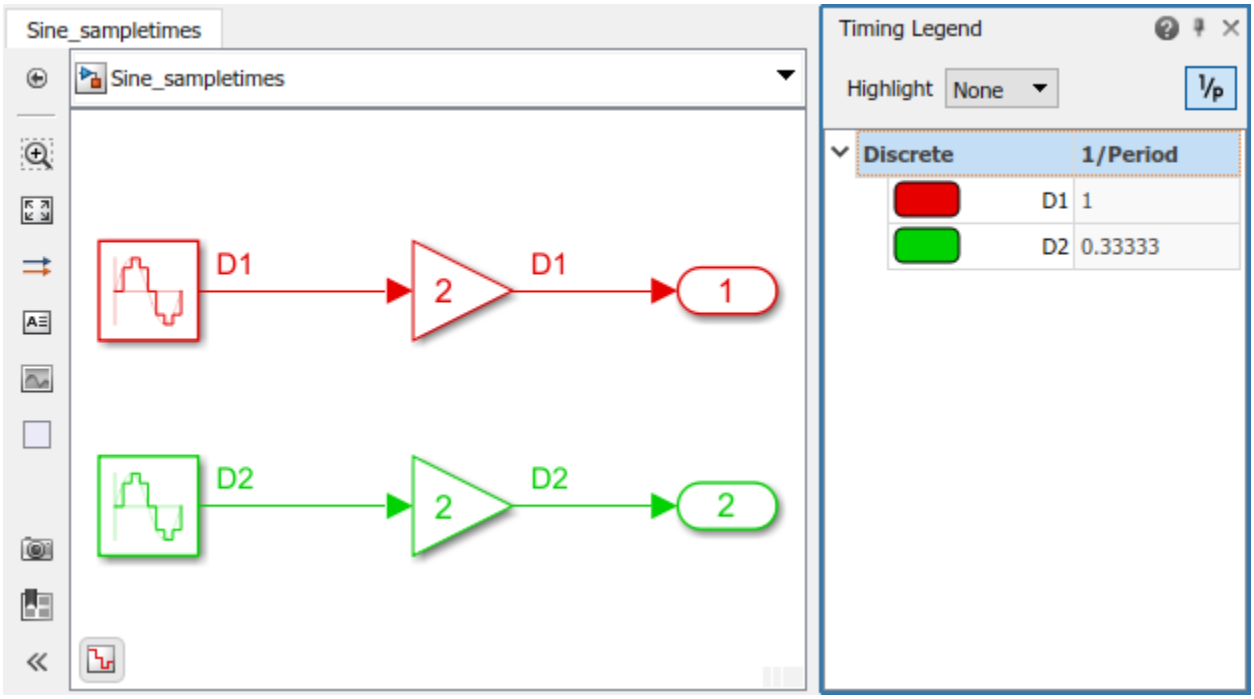
To enable highlighting of the origin of the sample times, click the `Origin` option from the **Highlight** menu. You can also click the type of the sample time to highlight all sources of a particular type of sample time.



To enable highlighting of all the blocks that contain a selected sample time, click the `All` option from the **Highlight** menu. You can also click the type of the sample time to highlight all the blocks and signals that contain the select type of sample time.

The None option from the **Highlight** menu clears current highlighting.

The button  shows discrete value as 1/period when the discrete sample time is present. When clicked, the discrete period is displayed as 1/period; for a nonzero offset, it displays as offset/period. The image shows 1/period values and the corresponding highlighted block in the model.



Note The Timing Legend displays all of the sample times in the model, including those that are not associated with any block. For example, if the fixed step size is 0.1 and all of the blocks have a sample time of 0.2, then both rates (0.1 and 0.2) appear in the legend.

For subsequent viewings of the legend, update the diagram to access the latest known information.

If you do not want to view the legend upon selecting **Sample Time Display**:

1. In the Simulink Editor, on the **Modeling** tab, select **Environment > Simulink Preferences**
2. In the **General** pane, clear **Open the timing legend when the sample time display is changed** and click **Apply**.

Inspect Sample Times Throughout a Model

The Model Data Editor (on the **Modeling** tab, click **Model Data Editor**) shows information about model data (signals, parameters, and states) in a sortable, searchable table. The **Sample Time** column shows the sample time specified for each signal in a model. After you update the block diagram, the column also shows the specific sample that each signal uses (for example, for signals for which you specify inherited sample time, `-1`). You can also use this column to specify sample times.

For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

See Also

“What Is Sample Time?” on page 7-2 | “Specify the Sample Time” on page 4-64 | “Types of Sample Time” on page 7-13

Types of Sample Time

In this section...

Discrete Sample Time

Given a block with a discrete sample time, Simulink executes the block output or update method at times

$$t_n = nT_s + |T_o|$$

where the sample time period T_s is always greater than zero and less than the simulation time, T_{sim}. The number of periods (n) is an integer that must satisfy:

$$0 \le n \le \frac{T_{sim}}{T_s}$$

As simulation progresses, Simulink computes block outputs only once at each of these fixed time intervals of t_n. These simulation times, at which Simulink executes the output method of a block for a given sample time, are referred to as *sample time hits*. Discrete sample times are the only type for which sample time hits are known *a priori*.

If you need to delay the initial sample hit time, you can define an offset, T_o.

The Unit Delay block is an example of a block with a discrete sample time.

Continuous Sample Time

Unlike the discrete sample time, continuous sample hit times are divided into major time steps and minor time steps, where the minor steps represent subdivisions of the major steps. The solver produces a result at each major time step. It uses results at the minor time steps to improve the accuracy of the result at the major time step.

The ODE solver you choose integrates all continuous states from the simulation start time to a given major or minor time step. The solver determines the times of the minor steps and uses the results at the minor time steps to improve the accuracy of the results at the major time steps. However, you see the block output only at the major time steps.

To specify that a block, such as the Derivative block, is continuous, enter `[0, 0]` or `0` in the **Sample time** field of the block dialog.

Fixed-in-Minor-Step

If the sample time of a block is set to `[0, 1]`, the block becomes *fixed-in-minor-step*. For this setting, Simulink does not execute the block at the minor time steps; updates occur only at the major time steps. This process eliminates unnecessary computations of blocks whose output cannot change between major steps.

While you can explicitly set a block to be fixed-in-minor-step, more typically Simulink sets this condition as either an inherited sample time or as an alteration to a user specification of `0` (continuous). This setting is equivalent to, and therefore converted to, the fastest discrete rate when you use a fixed-step solver.

Inherited Sample Time

If a block sample time is set to `[−1, 0]` or `−1`, the sample time is *inherited* and Simulink determines the best sample time for the block based on the block context within the model. Simulink performs this task during the compilation stage; the original inherited setting never appears in a compiled model. Therefore, you never see inherited ([–1, 0]) in the Sample Time Legend. (See “View Sample Time Information” on page 7-9.)

There are some blocks in which the sample time is inherited (-1) by default. For these blocks, the parameter is not visible on the block dialog box unless it is set to a noninherited value. Examples of these blocks include the Gain and Rounding Function blocks. As a good modeling practice, do not change the **Sample time** parameter for these blocks. For more information, see “Blocks for Which Sample Time Is Not Recommended” on page 7-17.

All inherited blocks are subject to the process of sample time propagation, as discussed in “How Propagation Affects Inherited Sample Times” on page 7-30

Constant Sample Time

In Simulink, a constant is a symbolic name or expression whose value you can change only outside the algorithm or through supervisory control. Blocks, like the constant block, whose outputs do not change during normal execution of the model, are always considered to be constant.

Simulink assigns constant sample time to these blocks. They run their block output method:

- At the start of a simulation.
- In response to runtime changes in the environment, such as tuning a parameter.

For constant sample time, the block sample time assignment is `[inf,0]` or `[inf]`.

For a block to allow constant sample time, these conditions hold:

- The block has no continuous or discrete states.
- The block does not drive an output port of a conditionally executed subsystem (see “Using Enabled Subsystems” on page 10-10).

S-Function Blocks

The Simulink block library includes several blocks, such as the MATLAB S-Function block, the Level-2 MATLAB S-Function block, and the C S-Function block, whose ports can produce outputs at different sample rates. It is possible for some of the ports of these blocks to have a constant sample time.

Variable Sample Time

Blocks that use a variable sample time have an implicit `SampleTime` parameter that the block specifies; the block tells Simulink when to run it. The compiled sample time is [–2, T_{vo}] where T_{vo} is a unique variable offset.

The Pulse Generator block is an example of a block that has a variable sample time. Since Simulink supports variable sample times for variable-step solvers only, the Pulse Generator block specifies a discrete sample time if you use a fixed-step solver.

To learn how to write your own block that uses a variable sample time, see “C MEX S-Function Examples”.

Controllable Sample Time

A block can be configured to use a controllable sample time with a resolution T_{base}. T_{base} is the smallest allowable time interval between block executions. To set T_{base} in your own C S-Function block, use `ssSetControllableSampleTime`.

A block using controllable sample time can be dynamically set to execute at n multiples of T_{base}. The time of the block's next execution is

$T_{next} = n\ T_{base} + T$

You can set n in your C S-Function block using `ssSetNumTicksToNextHitForControllableSampleTime`.

Triggered Sample Time

If a block is inside of a triggered-type (e.g., function-call, enabled and triggered, or iterator) subsystem, the block may be constant or have a triggered sample time. You cannot specify the triggered sample time type explicitly. However, to achieve a triggered type during compilation, you must set the block sample time to inherited (–1). Simulink then determines the specific times at which the block computes its output during simulation. One exception is if the subsystem is an asynchronous function call, as discussed in the following section.

Asynchronous Sample Time

An asynchronous sample time is similar to a triggered sample time. In both cases, it is necessary to specify an inherited sample time because the Simulink engine does not regularly execute the block. Instead, a run-time condition determines when the block executes. For the case of an asynchronous sample time, an S-function makes an asynchronous function call.

The differences between these sample time types are:

- Only a function-call subsystem can have an asynchronous sample time. (See “Using Function-Call Subsystems” on page 10-34.)
- The source of the function-call signal is an S-function having the option SS_OPTION_ASYNCHRONOUS.
- The asynchronous sample time can also occur when a virtual block is connected to an asynchronous S-function or an asynchronous function-call subsystem.

- The asynchronous sample time is important to certain code-generation applications. (See "Asynchronous Events" (Simulink Coder) in the *Simulink Coder User's Guide*.)
- The sample time is $[-1, -n]$.

For an explanation of how to use blocks to model and generate code for asynchronous event handling, see "Rate Transitions and Asynchronous Blocks" (Simulink Coder) in the *Simulink Coder User's Guide*.

See Also

"What Is Sample Time?" on page 7-2 | "View Sample Time Information" on page 7-9 | "Specify Sample Time" on page 7-3

Blocks for Which Sample Time Is Not Recommended

In this section...

Some blocks do not enable you to set the **Sample Time** parameter by default. However, you can see and set the **Sample Time** parameter for these blocks in an existing model if the sample time is set to a value other than the default of `-1` (inherited sample time). The **Sample Time** parameter is not available on certain blocks because specifying a sample time that is not `-1` on blocks such as the Gain, Sum, and n-D Lookup Table causes sample rate transition to be implicitly mixed with block algorithms. This mixing can often lead to ambiguity and confusion in Simulink models.

In most modeling applications, you specify rates for a model on the boundary of your system instead of on a block within the subsystem. You specify the system rate from incoming signals or the rate of sampling the output. You can also decide rates for events you are modeling that enter the subsystem as trigger, function-call, or enable/disable signals. Some global variables (such as Data Store Memory blocks) might need additional sample time specification. If you want to change rate within a system, use a Rate Transition block, which is designed specifically to model rate transitions.

In a future release, you might not be able see or set this parameter on blocks where it is not appropriate.

Best Practice to Model Sample Times

Use these approaches instead of setting the **Sample Time** parameter in the blocks where it is not appropriate:

- Adjust your model by specifying **Sample Time** only in the blocks listed in “Appropriate Blocks for the Sample Time Parameter” on page 7-17, and set **Sample Time** to `-1` for all other blocks. To change the sample time for multiple blocks simultaneously, use Model Explorer. For more information, see **Model Explorer**.
- Use the Rate Transition block to model rate transitions in your model.
- Use the Signal Specification block to specify sample time in models that don’t have source blocks, such as algebraic loops.
- Specify the simulation rate independently from the block sample times, using the Model Parameter dialog box.

Once you have completed these changes, verify whether your model gives the same outputs as before.

Appropriate Blocks for the Sample Time Parameter

Specify sample time on the boundary of a model or subsystem, or in blocks designed to model rate transitions. Examples include:

- Blocks in the Sources library
- Blocks in the Sinks library

- Trigger ports (if **Trigger type** is set to `function-call`) and Enable ports
- Data Store Read and Data Store Write blocks, as the Data Store Memory block they link to might be outside the boundary of the subsystem
- Rate Transition block
- Signal Specification block
- Blocks in the Discrete library
- Message Receive block
- Function Caller block

Specify Sample Time in Blocks Where Hidden

You can specify sample time in the blocks that do not display the parameter on the block dialog box. If you specify value other than `-1` in these blocks, no error occurs when you simulate the model. However, a message appears on the block dialog box advising to set this parameter to `-1` (inherited sample time). If you promote the sample time block parameter to a mask, this parameter is always visible on the mask dialog box.

To change the sample time in this case, use the `set_param` command. For example, select a block in the Simulink Editor and, at the command prompt, enter:

```
set_param(gcb,'SampleTime','2');
```

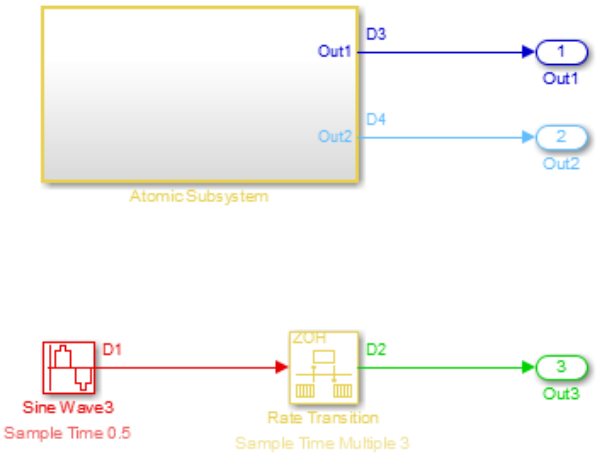
See Also

“Resolve Rate Transitions” on page 7-27 | “What Is Sample Time?” on page 7-2 | “Sample Times in Subsystems” on page 7-22 | “Sample Times in Systems” on page 7-23

Block Compiled Sample Time

During the compilation phase of a simulation, Simulink determines the sample time of a block from the `SampleTime` parameter (if the block has an explicit sample time), the block type (if it has an implicit sample time), or by the model content. This compiled sample time determines the sample rate of a block during simulation. You can determine the compiled sample time of any block in a model by first updating the model and then getting the block `CompiledSampleTime` parameter, using the `get_param` command.

For example, consider the model `ex_compiled_sample_new`.



Use `get_param` to obtain the block `CompiledSampleTime` parameter for each of the blocks in this example.

```
get_param('model_name/block_name','CompiledSampleTime');
```

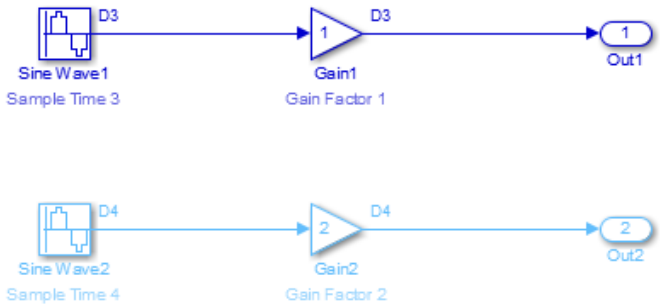
For the Sine Wave3 block,

```
get_param('ex_compiled_sample_new/Sine Wave3','CompiledSampleTime');
```

displays

```
0.5000   0
```

The atomic subsystem contains sine wave blocks with sample times of 3 and 4.



When calculating the block `CompiledSampleTime` for this subsystem, Simulink returns a cell array of the sample times present in the subsystem.

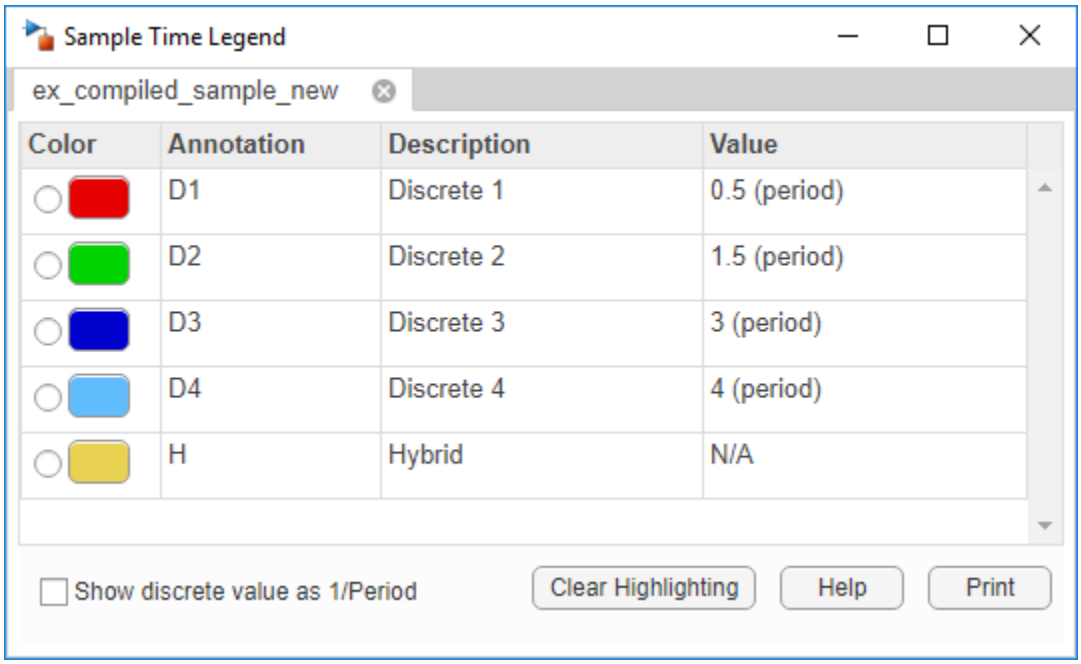
```
3   0
4   0
```

The greatest common divisor (GCD) of the two rates is 1. However, this is not necessarily one of the rates in the model.

The Rate Transition block in this model serves as a Zero-Order Hold. Since the `Sample Time Multiple` parameter is set to 3, the input to the rate transition block has a sample rate of 0.5 while the output has a rate of 1.5.

```
rt=get_param('ex_compiled_sample_new/Rate Transition',...
'CompiledSampleTime');
rt{:}

0.5000   0
1.5000   0
```

The Sample Time Legend shows all of the sample rates present in the model.



To inspect compiled sample times throughout a model, you can use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). After you update the block diagram, the right side of the **Sample Time** column shows compiled sample times for signals and data stores. For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

See Also

Related Examples

- “Sample Times in Subsystems” on page 7-22
- “View Sample Time Information” on page 7-9

Sample Times in Subsystems

Subsystems fall into two categories: triggered and non-triggered. For triggered subsystems, in general, the subsystem gets its sample time from the triggering signal. One exception occurs when you use a Trigger block to create a triggered subsystem. If you set the block **Trigger type** to **function-call** and the **Sample time type** to **periodic**, the `SampleTime` parameter becomes active. In this case, *you* specify the sample time of the Trigger block, which in turn, establishes the sample time of the subsystem.

There are four non-triggered subsystems:

- Virtual
- Enabled
- Atomic
- Action

Simulink calculates the sample times of virtual and enabled subsystems based on the respective sample times of their contents.

The atomic subsystem is a special case in that the subsystem block has a `SystemSampleTime` parameter. Moreover, for a sample time other than the default value of –1, the blocks inside the atomic subsystem can have only a value of `Inf`, –1, or the identical (discrete) value of the subsystem `SampleTime` parameter. If the atomic subsystem is left as inherited, Simulink calculates the block sample time in the same manner as the virtual and enabled subsystems. However, the main purpose of the subsystem `SampleTime` parameter is to allow for the simultaneous specification of a large number of blocks, within an atomic subsystem, that are all set to inherited. To obtain the sample time set on an atomic subsystem, use this command at the command prompt:

```
get_param(AtomicSubsystemBlock,‘SystemSampleTime’);
```

Finally, the sample time of the action subsystem is set by the If block or the Switch Case block.

For non-triggered subsystems where blocks have different sample rates, Simulink returns the Compiled Sample Time for the subsystem as a cell array of all the sample rates present in the subsystem. To see this, use the `get_param` command at MATLAB prompt.

```
get_param(subsystemBlock,'CompiledSampleTime')
```

See Also

More About

- “Block Compiled Sample Time” on page 7-19
- “Sample Times in Systems” on page 7-23
- “Specify Execution Domain” on page 7-33

Sample Times in Systems

In this section...

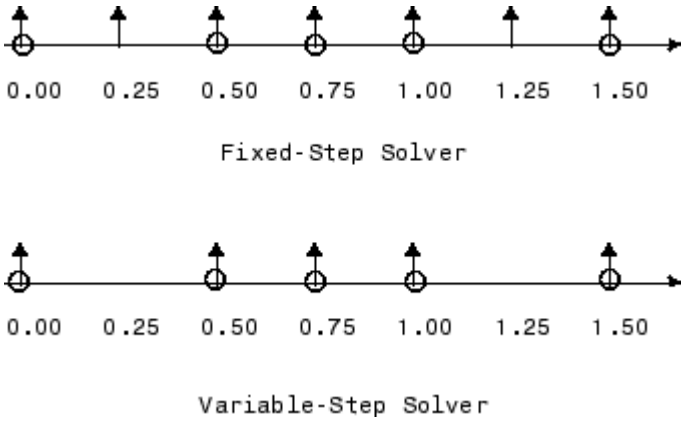
Purely Discrete Systems

A purely discrete system is composed solely of discrete blocks and can be modeled using either a fixed-step or a variable-step solver. Simulating a discrete system requires that the simulator take a simulation step at every sample time hit. For a *multirate discrete system*—a system whose blocks Simulink samples at different rates—the steps must occur at integer multiples of each of the system sample times. Otherwise, the simulator might miss key transitions in the states of the system. The step size that the Simulink software chooses depends on the type of solver you use to simulate the multirate system and on the fundamental sample time.

The *fundamental sample time* of a multirate discrete system is the largest double that is an integer divisor of the actual sample times of the system. For example, suppose that a system has sample times of 0.25 and 0.50 seconds. The fundamental sample time in this case is 0.25 seconds. Suppose, instead, the sample times are 0.50 and 0.75 seconds. The fundamental sample time is again 0.25 seconds.

The importance of the fundamental sample time directly relates to whether you direct the Simulink software to use a fixed-step or a variable-step discrete solver to solve your multirate discrete system. A fixed-step solver sets the simulation step size equal to the fundamental sample time of the discrete system. In contrast, a variable-step solver varies the step size to equal the distance between actual sample time hits.

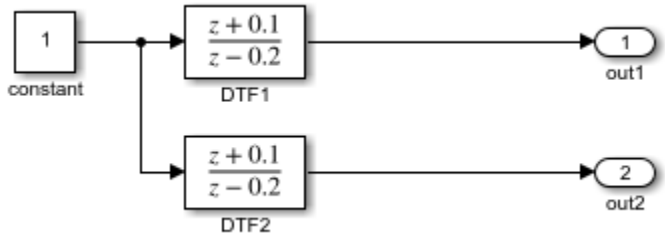
The following diagram illustrates the difference between a fixed-step and a variable-step solver.



In the diagram, the arrows indicate simulation steps and circles represent sample time hits. As the diagram illustrates, a variable-step solver requires fewer simulation steps to simulate a system, if the fundamental sample time is less than any of the actual sample times of the system being simulated. On the other hand, a fixed-step solver requires less memory to implement and is faster if one of the system sample times is fundamental. This can be an advantage in applications that entail generating code from a Simulink model (using Simulink Coder). In either case, the discrete solver provided by

Simulink is optimized for discrete systems; however, you can simulate a purely discrete system with any one of the solvers and obtain equivalent results.

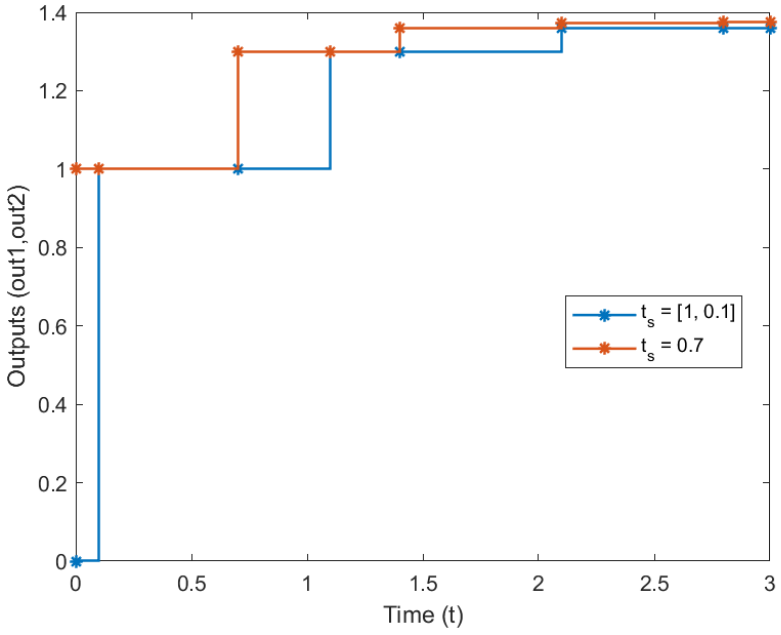
Consider the following example of a simple multirate system. For this example, the DTF1 Discrete Transfer Fcn block **Sample time** is set to `[1 0.1]` [], which gives it an offset of `0.1`. The **Sample time** of the DTF2 Discrete Transfer Fcn block is set to `0.7` , with no offset. The solver is set to a variable-step discrete solver.



Running the simulation and plotting the outputs using the `stairs` function

```
set_param(bdroot,'SolverType','Variable-Step','SolverName','VariableStepDiscrete','SaveFormat','A
simOut = sim(bdroot,'Stoptime','3');
stairs(simOut.tout,simOut.yout,'-*','LineWidth',1.2);
xlabel('Time (t)');
ylabel('Outputs (out1,out2)');
legend('t_s = [1, 0.1]','t_s = 0.7','location','best')
```

produces the following plot.



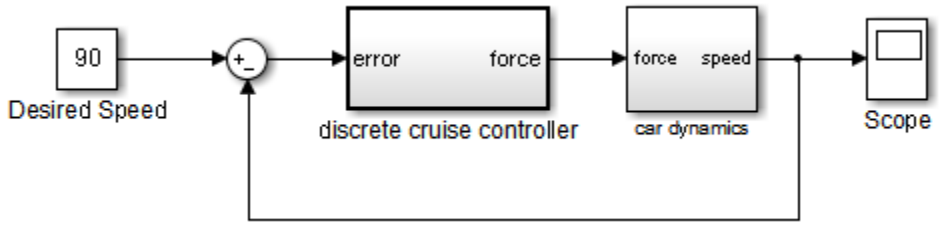
(For information on the `sim` command. see “Run Simulations Programmatically” on page 26-2.)

As the figure demonstrates, because the DTF1 block has a `0.1` offset, the DTF1 block has no output until `t = 0.1`. Similarly, the initial conditions of the transfer functions are zero; therefore, the output of DTF1, y(1), is zero before this time.

Hybrid Systems

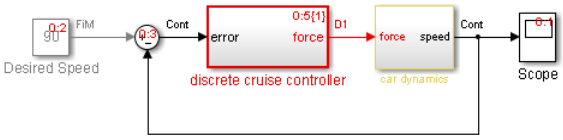
Hybrid systems contain both discrete and continuous blocks and thus have both discrete and continuous states. However, Simulink solvers treat any system that has both continuous and discrete sample times as a hybrid system. For information on modeling hybrid systems, see “Modeling Hybrid Systems”.

In block diagrams, the term hybrid applies to both hybrid systems (mixed continuous-discrete systems) and systems with multiple sample times (multirate systems). Such systems turn yellow in color when you perform an **Update Diagram** with Sample Time Display **Colors** turned 'on'. As an example, consider the following model that contains an atomic subsystem, “Discrete Cruise Controller”, and a virtual subsystem, “Car Dynamics”. (See `ex_execution_order`.)

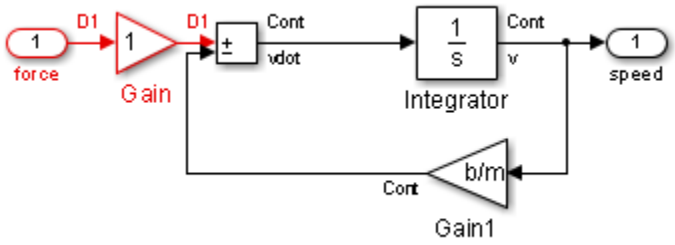


Car Model

With the **Sample Time** option set to **All**, an **Update Diagram** turns the virtual subsystem yellow, indicating that it is a hybrid subsystem. In this case, the subsystem is a true hybrid system since it has both continuous and discrete sample times. As shown below, the discrete input signal, D1, combines with the continuous velocity signal, v, to produce a continuous input to the integrator.

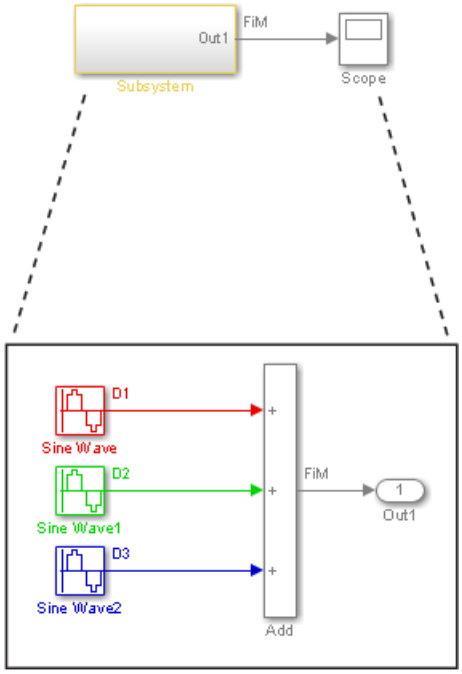


Car Model after an Update Diagram



Car Dynamics Subsystem after an Update Diagram

Now consider a multirate subsystem that contains three Sine Wave source blocks, each of which has a unique sample time — 0.2, 0.3, and 0.4, respectively.



Multirate Subsystem after an Update Diagram

An **Update Diagram** turns the subsystem yellow because the subsystem contains more than one sample time. As shown in the block diagram, the Sine Wave blocks have discrete sample times D1, D2, and D3 and the output signal is fixed in minor step.

In assessing a system for multiple sample times, Simulink does not consider either constant [inf, 0] or asynchronous [–1, –n] sample times. Thus a subsystem consisting of one block that outputs constant value and one block with a discrete sample time will not be designated as hybrid.

The hybrid annotation and coloring are very useful for evaluating whether or not the subsystems in your model have inherited the correct or expected sample times.

See Also

“Blocks for Which Sample Time Is Not Recommended” on page 7-17 | “View Sample Time Information” on page 7-9

Resolve Rate Transitions

In general, a rate transition exists between two blocks if their sample times differ, that is, if either of their sample-time vector components are different. The exceptions are:

- Blocks that output constant value never have a rate transition with any other rate.
- A continuous sample time (black) and the fastest discrete rate (red) never has a rate transition if you use a fixed-step solver.
- A variable sample time and fixed in minor step do not have a rate transition.

You can resolve rate transitions manually by inserting rate transition blocks and by using two diagnostic tools. For the single-tasking execution mode, the **Single task rate transition** diagnostic allows you to set the level of Simulink rate transition messages. The **Multitask rate transition** diagnostic serves the same function for multitasking execution mode. These execution modes directly relate to the type of solver in use: Variable-step solvers are always single-tasking; fixed-step solvers may be explicitly set as single-tasking or multitasking.

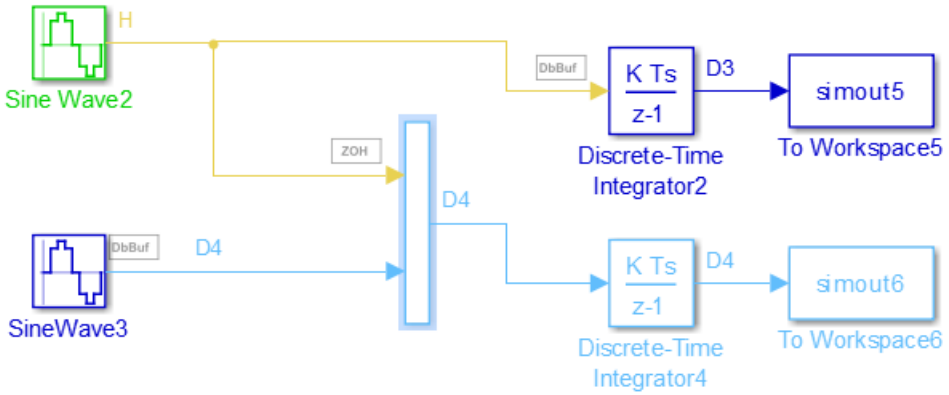
Automatic Rate Transition

Simulink can detect mismatched rate transitions in a multitasking model during an update diagram and automatically insert Rate Transition blocks to handle them. To enable this, in the **Solver** pane of model configuration parameters, select **Automatically handle rate transition for data transfer**. The default setting for this option is off. When you select this option:

- Simulink handles transitions between periodic sample times and asynchronous tasks.
- Simulink inserts hidden Rate Transition blocks in the block diagram.
- Automatically inserted Rate Transition blocks operate in protected mode for periodic tasks and asynchronous tasks. You cannot alter this behavior. For periodic tasks, automatically inserted Rate Transition blocks operate with the level of determinism specified by the **Deterministic data transfer** parameter in the **Solver** pane. The default setting is `Whenever possible`, which enables determinism for data transfers between periodic sample-times that are related by an integer multiple. For more information, see “Deterministic data transfer”. To use other modes, you must insert Rate Transition blocks and set their modes manually.

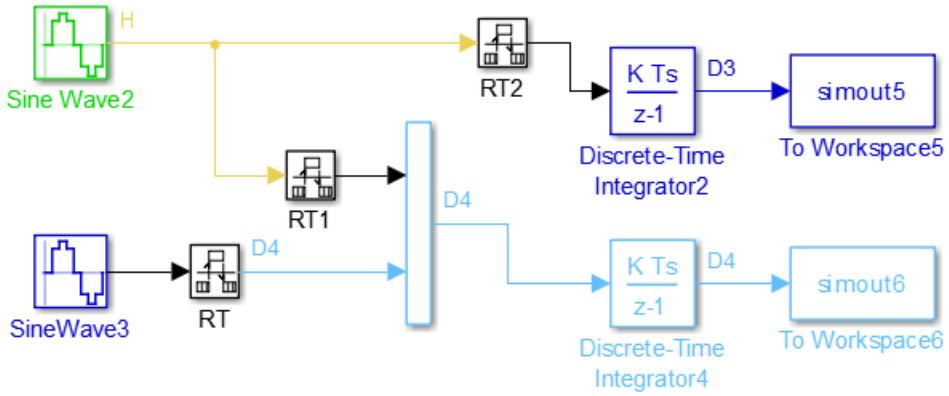
Visualize Inserted Rate Transition Blocks

When you select the **Automatically handle rate transition for data transfer** option, Simulink inserts Rate Transition blocks in the paths that have mismatched transition rates. These blocks are hidden by default. To visualize the inserted blocks, update the diagram. Badge labels appear in the model and indicate where Simulink inserted Rate Transition blocks during the compilation phase. For example, in this model, three Rate Transition blocks were inserted between the two Sine Wave blocks and the Multiplexer and Integrator when the model compiled. The ZOH and DbBuf badge labels indicate these blocks.



You can show or hide badge labels. On the **Debug** tab, select **Information Overlays > Automatic Rate Transitions**.

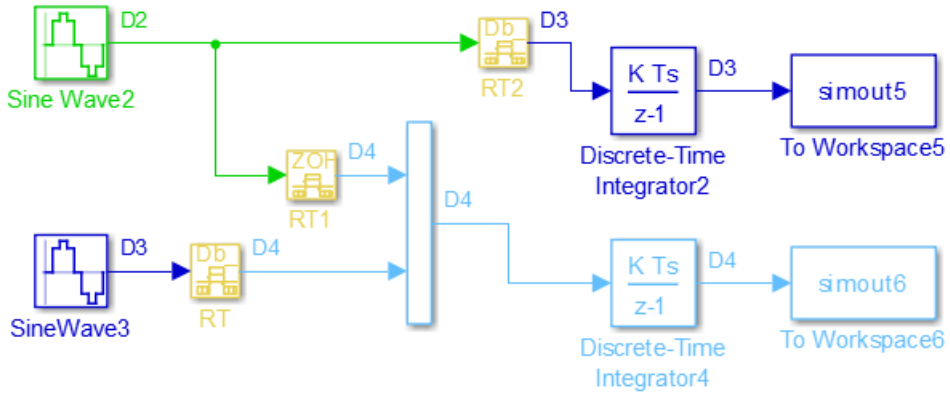
To configure the hidden Rate Transition blocks, right click on a badge label and click on **Insert rate transition block** to make the block visible.



When you make hidden Rate Transition blocks visible:

- You can see the type of Rate Transition block inserted as well as the location in the model.
- You can set the **Initial Conditions** of these blocks.
- You can change data transfer and sample time block parameters.

Validate the changes to your model by updating your diagram.



Displaying inserted Rate Transition blocks is not compatible with export-function models.

To learn more about the types of Rate Transition blocks, see Rate Transition.

Note Suppose you automatically insert rate transition blocks and there is a virtual block specifying sample time upstream of the block you insert. You cannot click the badge of the inserted block to configure the block and make it visible because the sample time on the virtual block causes a rate transition as well. In this case, manually insert a rate transition block before the virtual block. To learn more about virtual blocks, see “Nonvirtual and Virtual Blocks” on page 36-2.

See Also

Related Examples

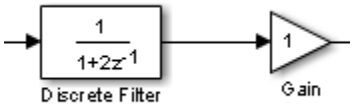
- “Handle Rate Transitions” (Simulink Coder)

More About

- “Time-Based Scheduling and Code Generation” (Simulink Coder)

How Propagation Affects Inherited Sample Times

During a model update, for example at the beginning of a simulation, Simulink uses a process called sample time propagation to determine the sample times of blocks that inherit their sample times. The figure below illustrates a Discrete Filter block with a sample time period T_s driving a Gain block.



Because the output of the Gain block is the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to the sample rate of the filter. The establishment of such effective rates is the fundamental mechanism behind sample time propagation in Simulink.

Process for Sample Time Propagation

Simulink uses the following basic process to assign sample times to blocks that inherit their sample times:

1. Propagate known sample time information forward.
2. Propagate known sample time information backward.
3. Apply a set of heuristics to determine additional sample times.
4. Repeat until all sample times are known.

Simulink Rules for Assigning Sample Times

A block having a block-based sample time inherits a sample time based on the sample times of the blocks connected to its inputs, and in accordance with the following rules:

Rule	Action
All of the inputs have the same sample time and the block can accept that sample time	Simulink assigns the sample time to the block
The inputs have different discrete sample times and all of the input sample times are integer multiples of the fastest input sample time	Simulink assigns the sample time of the fastest input to the block . (This assignment assumes that the block can accept the fastest sample time.)
The inputs have different discrete sample times, some of the input sample times are not integer multiples of the fastest sample time, and the model uses a variable-step solver	Simulink assigns a fixed-in-minor-step sample time to the block.

Rule	Action
The inputs have different discrete sample times, some of the input sample times are not integer multiples of the fastest sample time, the model uses a fixed-step solver, and Simulink can compute the greatest common integer divisor (GCD) of the sample times coming into the block	Simulink assigns the GCD sample time to the block. Otherwise, Simulink assigns the fixed step size of the model to the block.
The sample times of some of the inputs are unknown, or if the block cannot accept the sample time	Simulink determines a sample time for the block based on a set of heuristics.

See Also

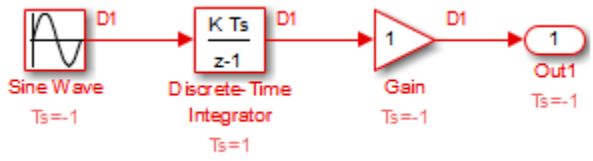
“Blocks for Which Sample Time Is Not Recommended” on page 7-17

More About

• “Backpropagation in Sample Times” on page 7-32

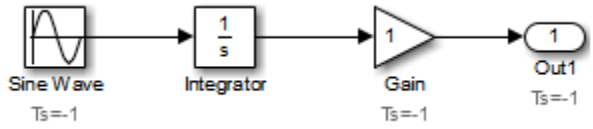
Backpropagation in Sample Times

When you update or simulate a model that specifies the sample time of a source block as inherited (–1), the sample time of the source block may be backpropagated; Simulink may set the sample time of the source block to be identical to the sample time specified by or inherited by the block connected to the source block. For example, in the model below, the Simulink software recognizes that the Sine Wave block is driving a Discrete-Time Integrator block whose sample time is 1; so it assigns the Sine Wave block a sample time of 1.



You can verify this sample time setting by selecting **Information Overlays > Colors** from the **Debug** tab of theSimulinktoolstrip and noting that both blocks are red. Because the Discrete-Time Integrator block looks at its input only during its sample hit times, this change does not affect the results of the simulation, but does improve the simulation performance.

Now replacing the Discrete-Time Integrator block with a continuous Integrator block, as shown in the model below, causes the Sine Wave and Gain blocks to change to continuous blocks. You can test this change by, on the **Modeling** tab, selecting **Update Model** to update the colors. Both blocks now appear black.



Note Backpropagation makes the sample times of model sources dependent on block connectivity. If you change the connectivity of a model whose sources inherit sample times, you can inadvertently change the source sample times. For this reason, when you update or simulate a model, by default, Simulink displays warnings at the command line if the model contains sources that inherit their sample times.

See Also

“View Sample Time Information” on page 7-9 | “How Propagation Affects Inherited Sample Times” on page 7-30

Specify Execution Domain

Execution domain specification allows you to set a model and its subsystems and referenced models to simulate as discrete-time or data-driven systems. Use this setting to separate the discrete dynamics from the rest of its dynamics, for example, in the design of a deployable controller for a plant that is modeled with continuous-time dynamics.

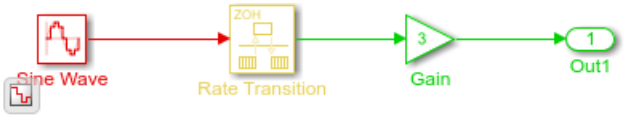
To simulate a computationally intensive signal processing or multirate signal processing system, you can also assign a dataflow domain. Dataflow domains simulate using a model of computation synchronous dataflow, which is data-driven and statically scheduled. For more information, see "Dataflow Domain" (DSP System Toolbox).

You can create subsystems that maintain their discrete execution domain irrespective of their environment. By constraining a subsystem to be discrete, you can increase reusability of your subsystem as a component. To improve code generation, this specification reduces unnecessary update methods, reduces major time step checks, and increases reusability of generated code.

Domain Specification Badge

The domain specification badge indicates the execution domain computed to a model or subsystem when you update the model diagram. You can toggle the visibility of the domain specification badge by turning on the **Sample Time Display**. For more information on visualizing sample time, see "View Sample Time Information" on page 7-9. The badge is visible at the bottom left corner of the Simulink Editor.

The model below shows a discrete Sine Wave block whose rate is reduced by the Rate Transition block before driving the Gain block.



Observe that the model receives the *Discrete* execution domain because its contents are all discrete.

You can also toggle the visibility of the badge by enabling or disabling the **Set Domain Specification** parameter in the **Execution** tab of the **Property Inspector**.

Types of Execution Domains

You can instruct Simulink to assign the execution domain (along with the allowed sample times) via the **Property Inspector**.

Specification	Discrete	Other	Dataflow
`Deduce from contents`	X	X	-
`Discrete`	X	-	-
`Dataflow`	-	-	X

-  `Deduce from contents` Let Simulink assign the execution domain based on the contents of the subsystem.
-  `Discrete` Constrain all blocks in a subsystem to be discrete.
-  `Dataflow` Simulate a computationally-intensive signal processing or multi-rate signal processing system. This setting requires the DSP System Toolbox™.

When you update the model diagram or simulate the model, the badge displays the computed execution domain for the model component. There are three execution domains in Simulink:

-  **Discrete** Blocks have discrete states and sample times. Allowed samples times include “Discrete Sample Time” on page 7-13, “Controllable Sample Time” on page 7-15, and “Asynchronous Sample Time” on page 7-15.
-  **Dataflow** Dataflow domains simulate using computation synchronous dataflow, which is data-driven and statically scheduled. This execution domain requires the DSP System Toolbox. For more information, see “Specifying Dataflow Domains” (DSP System Toolbox).
-  **Other** Blocks are not strictly discrete.

  Subsystems that receive the **Other** execution domain include:

  - Subsystems whose blocks have continuous states and sample times, including “Continuous Sample Time” on page 7-13, “Fixed-in-Minor-Step” on page 7-14, and “Variable Sample Time” on page 7-15
  - Subsystems with a mixture of continuous and discrete sample times.
  - Subsystems with “Asynchronous Sample Time” on page 7-15.
  - Triggered Subsystem
  - Function-Call Subsystem
  - Enabled and Triggered Subsystem
  - Initialize Function
  - Reset Function
  - Terminate Function
  -

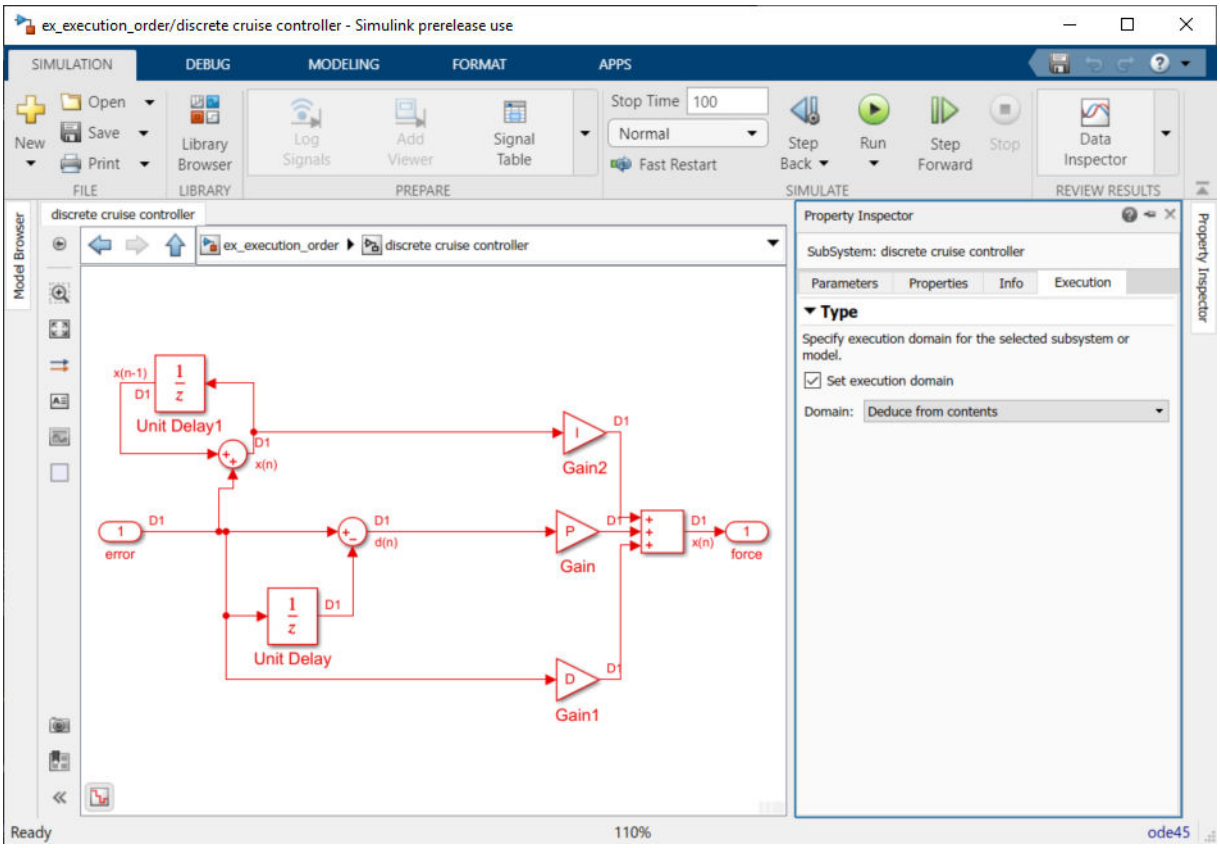
If a subsystem has continuous, variable, fixed-in-minor step, “Constant Sample Time” on page 7-14, or a mixture of sample times, you can use the badge to enable or disable domain specification. The subsystem still receives the **Other** time domain.

The domain specification badge is not actionable when the currently selected subsystem or model is a linked block, inside a library block, or a conditionally executed subsystem that receives the **Other** domain. To change the execution domain of a linked library block, break the link to the parent library block. See “Disable or Break Links to Library Blocks” on page 41-20.

Set Execution Domain

You can set the domain specification per subsystem and at the root level of the model using the **Execution** tab of the Property Inspector. To enable the Property Inspector for the model, on the

Modeling tab, under **Design**, click **Property Inspector**, or press **Ctrl+Shift+I** on your keyboard. If the domain specification badge is displayed, you can also open the **Execution** settings in the Property Inspector by clicking the badge. See “Domain Specification Badge” on page 7-33.



Select the **Set Execution Domain** check box. You can now specify the **Domain**.

Note Changing the domain specification at the root level of the model does not change the setting for its child subsystems.

You can also enable this setting from the command line using `set_param` to set the `SetExecutionDomain` parameter `'on'` or `'off'`.

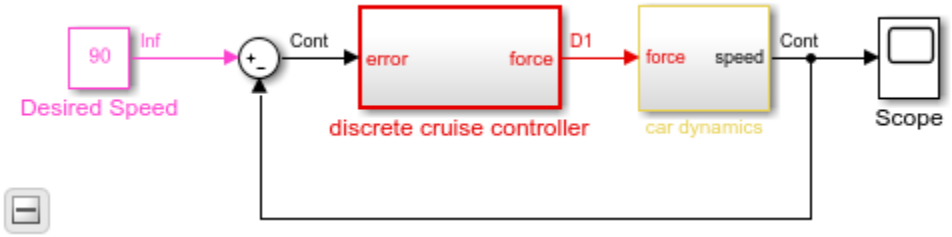
Once enabled, the default setting for the **Domain** parameter is `Deduce from contents`. When you update the diagram, the execution domain is deduced from the characteristics of the blocks in the currently open subsystem. For example, a system that has only discrete blocks is in the **Discrete** execution domain. See “Types of Execution Domains” on page 7-33.

The badge shows the current specification setting. If you set the subsystem domain to `Deduce from contents`, the badge text displays **Deduce** until you update the diagram. Once you update the model diagram, the badge shows the computed execution domain, as described in “Types of Execution Domains” on page 7-33. When you enable **Set domain specification** and **Domain** is set to `Deduce from Contents`, Simulink computes the execution domain of the currently focused subsystem based on the blocks and sample times inside the subsystem.

To set the **Domain** parameter from the command line, use `set_param` to change `ExecutionDomainType` to either `'Deduce'` or `'Discrete'`. You can also get the computed execution domain after you update the diagram using the `CompiledExecutionDomain` parameter of the subsystem.

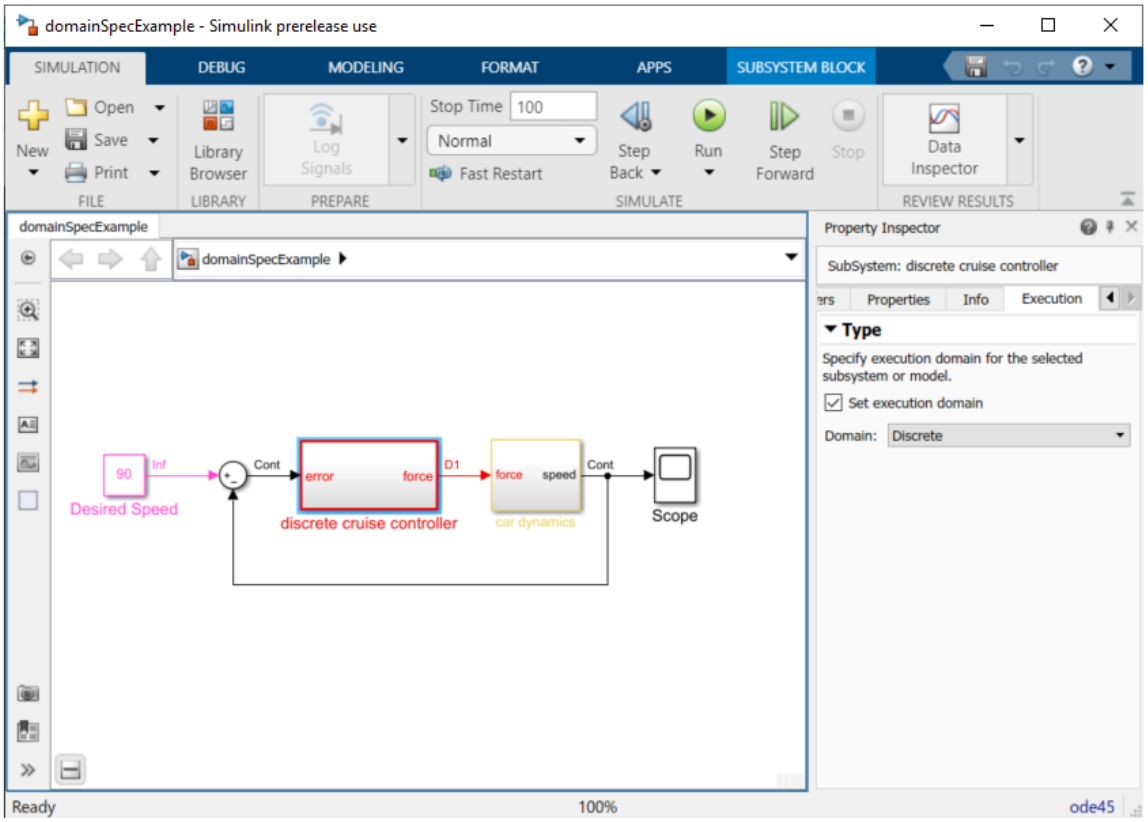
Enforce Discrete Execution Domain for a Subsystem

This model shows how to specify execution domains for the constituent subsystems of a model. The model has a discrete cruise controller subsystem that tracks the reference speed set in the Desired Speed block. A car dynamics subsystem models the continuous-time dynamics of the car.

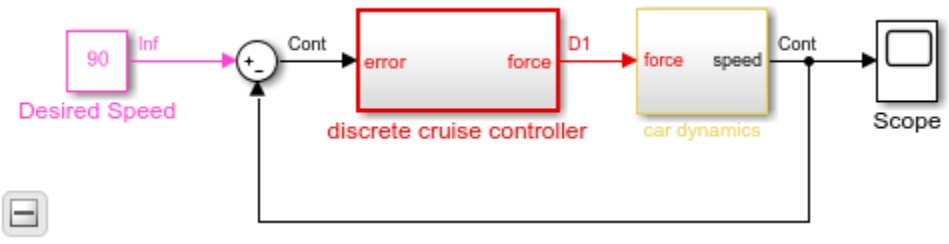


Notice that the discrete cruise controller of the model has a hybrid sample time due to the presence of a continuous-time signal from the output of the car dynamics at the input port of the controller.

To enforce discrete-time execution of the controller, select the subsystem and open the **Execution** tab of the **Property Inspector** by clicking on the Domain badge at the bottom-left corner of the Simulink Editor.

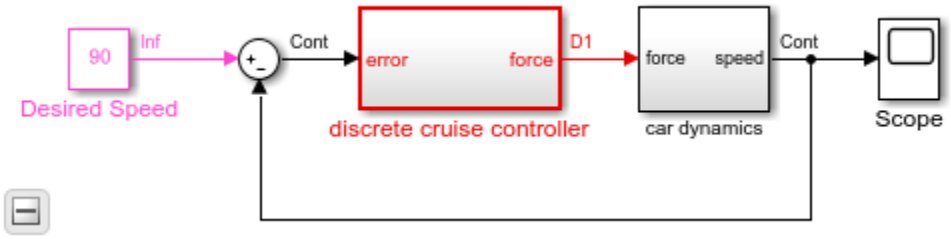


Enable the **Set execution domain** parameter and set **Domain** to `Discrete`. Update the model diagram or simulate the model.



Note that the discrete cruise controller subsystem is now discrete.

You can also set the execution domain of the car dynamics to `Deduce from Contents`. The car dynamics subsystem receives the Hybrid sample time and the **Other** execution domain. If you wish, set the **Sample Time** parameter of the Inport block in this subsystem to 0.



See Also

“What Is Sample Time?” on page 7-2 | “Sample Times in Subsystems” on page 7-22 | “How Propagation Affects Inherited Sample Times” on page 7-30 | “Dataflow Domain” (DSP System Toolbox)

See Also

Referencing a Model

Model Reference Basics

You can include one model in another by using a Model block. Each instance of a Model block is a model reference. For simulation and code generation, blocks within a referenced model execute together as a unit. The model that contains a referenced model is a parent model. A collection of parent and referenced models constitutes a model hierarchy.

A model can function as both a standalone model and a referenced model, without changing the model or any entities derived from it. To use a referenced model as a standalone model, the referenced model cannot depend on data that is available only from a higher-level model.

Model Reference Advantages

Like subsystems, model references allow you to organize large models hierarchically. Like libraries, model references allow you to define a set of blocks once and use it repeatedly. Model references provide several advantages that are unavailable with subsystems and libraries. Several of these advantages result from referenced models compiling independent from the context of the Model block, including:

- **Modular development**

  You can develop a referenced model independently from the models that use it.
- **Model protection**

  With a Simulink Coder license, you can obscure the contents of a referenced model, allowing you to distribute the model without revealing its intellectual property.

  With a Simulink license, you can reference a protected model provided by a third party. Depending on the granted protected-model permissions, you can view, simulate, and generate code for the protected model.
- **Inclusion by reference**

  You can reference a model multiple times without making redundant copies, and multiple models can reference the same model.
- **Incremental loading**

  Simulink software loads a referenced model when it is needed, which speeds up model loading.
- **Accelerated simulation**

  Simulink software can convert a referenced model to code and simulate the model by running the code, which is faster than interactive simulation.
- **Incremental code generation**

  Accelerated simulation generates code only if the model has changed since the code was previously generated.
- **Independent configuration sets**

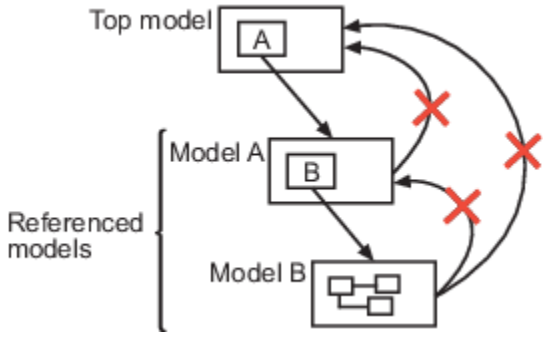
  The configuration set used by a referenced model can differ from the configuration set of its parent or other referenced models.

For a video summarizing model reference advantages, see Modular Design Using Model Referencing.

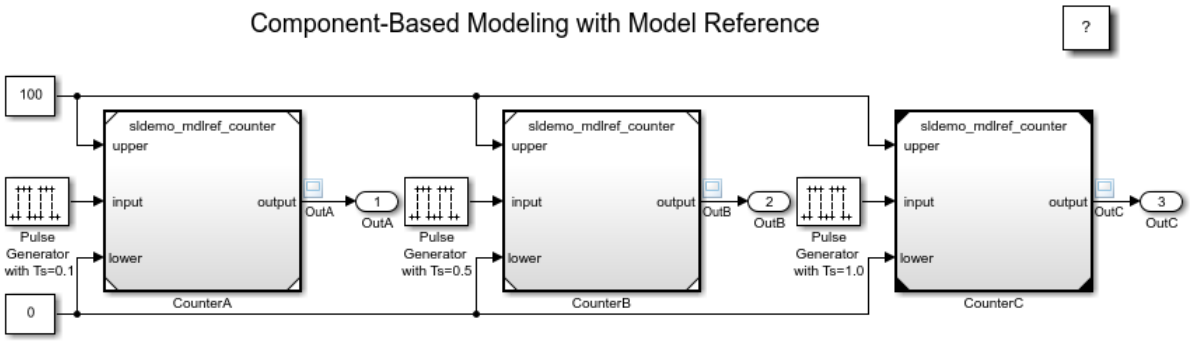
To compare model references, subsystems, and libraries, see “Choose Among Types of Model Components” on page 22-4. You can use multiple componentization techniques in the same model.

Model Hierarchies

Referenced models can contain Model blocks that reference lower-level models. The top model is the top model in a hierarchy of referenced models. Where only one level of model reference exists, the parent model and top model are the same. To prevent cyclic inheritance, a Model block cannot refer directly or indirectly to a model that is superior to it in the model hierarchy. This figure shows cyclic inheritance.



A parent model can contain multiple Model blocks that reference the same referenced model, as long as the referenced model does not define global data. For example, the `sldemo_mdlref_basic` model includes Model blocks that reference three instances of the same referenced model, `sldemo_mdlref_counter`.



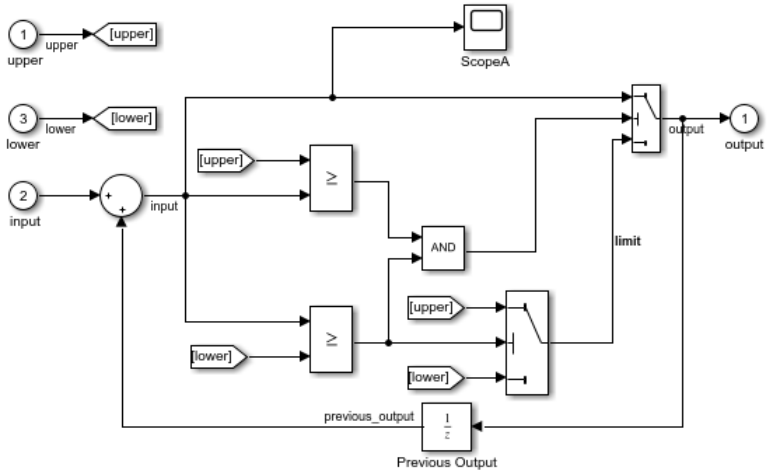
The referenced model can also appear in other parent models at any level.

Model Block and Referenced Model Interface

A Model block displays input, output, and control ports that correspond to root-level input, output, and control ports of the model it references. To connect the referenced model to other elements of the parent model, use these Model block ports. Connecting a signal to a Model block port connects the signal to the corresponding port in the referenced model.

In model `sldemo_mdlref_basic`, each Model block has three inputs: two Constant blocks and a Pulse Generator block. Each Model block has one output signal logged to a scope. Because the input signal from each Pulse Generator block uses a different sample time, the output signal from each Model block differs for each model instance.

To connect to the parent model, referenced model `sldemo_mdlref_counter` includes three Inport blocks (**upper**, **lower**, and **input**) and one Outport block (**output**).



Signal attributes in the referenced model are independent from the context of the Model block. For example, signal dimensions and data types do not propagate across the Model block boundary. To define signal attributes in the referenced model, define block parameters for root-level Inport and In Bus Element blocks.

For more information, see “Model Reference Interface and Boundary” on page 8-31.

Model Workspaces and Data Dictionaries

Each model has its own workspace for storing variable values. In a model hierarchy, each model workspace acts as a unique namespace. Therefore, you can use the same variable name in multiple model workspaces. To share data among models, you can use a data dictionary.

Duplicate data definitions can exist in a model reference hierarchy under these conditions:

- Each model in the hierarchy can see only one definition.
- Definitions must be the same across models in the hierarchy.

For more information on where you can store variables and objects, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.

Referenced Model Execution

To use an external signal to control whether a Model block executes during simulation, see “Modify Referenced Models for Conditional Execution” on page 8-24.

Variant Subsystem blocks can contain Model blocks as variant systems. For information on variant systems, see “What Are Variants and When to Use Them” on page 12-2.

By default, a block parameter has the same value in each Model block instance of a reusable referenced model. To specify a different block parameter value for each instance of a reusable referenced model, create model arguments. For example, if you add a Gain block to model

`sldemo_mdlref_counter`, model arguments allow each of the three instances of this model to use different gain values. See “Parameterize Instances of a Reusable Referenced Model” on page 8-64.

With a model mask, you can control the appearance of Model blocks and customize the way the blocks display model arguments. For model mask requirements, see “Model Masks” on page 8-6.

Referenced Model Simulation and Code Generation

You can simulate a referenced model either interpretively (in normal mode) or by compiling the referenced model to code and executing the code (in accelerator mode). For details, see “Choose Simulation Modes for Model Hierarchies” on page 8-39.

Simulink cache files contain build artifacts that can speed up simulation and code generation. For more information and an example workflow, see “Share Simulink Cache Files for Faster Simulation” on page 8-54.

To learn about generating code for a model reference hierarchy, see “Generate Code for Model Reference Hierarchy” (Simulink Coder).

See Also

Blocks
Model

Related Examples

- “Reference Existing Models” on page 8-11
- “Reference Protected Models from Third Parties” on page 8-13
- “Convert Subsystems to Referenced Models” on page 8-18

More About

- “Component-Based Modeling Guidelines” on page 22-2
- “Model Reference Requirements and Limitations” on page 8-6

Model Reference Requirements and Limitations

Before referencing models, consider model reference requirements and limitations. By understanding the requirements and limitations upfront, you are better prepared to reference models successfully.

Model Reuse

You can reference a model more than once in a model hierarchy unless the referenced model has any of these properties:

- The model references another model that is set to single instance.
- The model contains To File blocks.
- The model contains an internal signal or state with a storage class that is not supported for multi-instance models. Internal signals and states must have the storage class set to `Auto` or `Model default` and the default storage class for internal data must be a multi-instance storage class.
- The model uses any of these Stateflow constructs:
  - Exported Stateflow graphical functions
  - Machine-parented data
- The referenced model executes in accelerator mode and contains an S-function that is either not inlined or is inlined but does not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`.
- The model contains a function-call subsystem that:
  - Simulink forces to be a function
  - Is called by a wide signal

If the referenced model has any of these properties, only one instance of the model can appear in the model hierarchy. The model must have **Total number of instances allowed per top model** set to `One`.

Model Masks

You can use masked blocks in a referenced model. Also, you can mask a referenced model (see “Create and Reference a Masked Model” on page 39-49).

To successfully use masks, consider these requirements and limitations:

- If a mask specifies the name of a referenced model, the mask must provide the name of the referenced model directly. You cannot use a workspace variable to provide the name.
- The mask workspace of a Model block is not available to the referenced model. Any variable that the referenced model uses must resolve to either of these workspaces:
  - A workspace that the referenced model defines
  - The MATLAB base workspace
- Mask callbacks cannot add Model blocks, change the Model block name, or change the Model block simulation mode.

S-Functions in Referenced Models

Different types of S-functions provide different levels of support for model references.

S-Function Type	Models Referenced in Normal Mode	Models Referenced in Accelerator Mode
Level-1 MATLAB S-function	Not supported	Not supported
Level-2 MATLAB S-function	Supported	Supported — requires a TLC file
Handwritten C MEX S-function	Supported — can be inlined with a TLC file	Supported — can be inlined with a TLC file
S-Function Builder	Supported	Supported
Legacy Code Tool	Supported	Supported

When you use S-functions in referenced models, consider these requirements and limitations.

S-Function Consideration	Requirements and Limitations
Sample Time Inheritance	If an S-function depends on an inherited sample time, the S-function must explicitly declare a dependence on the inherited sample time. To control sample-time inheritance, use `ssSetModelReferenceSampleTimeInheritanceRule` differently based on whether an S-function permits or precludes inheritance. For details, see “S-Functions That Specify Sample Time Inheritance Rules” (Simulink Coder).
Accelerator Mode Referenced Models	For accelerator mode referenced models that contain an S-function that requires inlining using a Target Language Compiler file, the S-function must use the `ssSetOptions` macro to set the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` option in its `mdlInitializeSizes` method. The simulation target does not inline the S-function unless the S-function sets this option. A referenced model cannot use noninlined S-functions in these cases: • The model uses a variable-step solver. • The model is referenced more than once in the model hierarchy. To work around this limitation, use normal mode or: **1** Make copies of the referenced model. **2** Assign different names to the copies. **3** Reference a different copy at each location that needs the model. • The S-function uses character vector parameters. A referenced model in accelerator mode cannot use S-functions generated by Simulink Coder software.

S-Function Consideration	Requirements and Limitations
Normal Mode Referenced Models	Under certain conditions, when a C S-function appears in a referenced model that executes in normal mode, successful execution is impossible. For details, see “S-Functions in Normal Mode Referenced Models”. To specify whether an S-function can be used in a normal mode referenced model, use the `ssSetModelReferenceNormalModeSupport` SimStruct function. For an S-function to work with multiple instances of referenced models in normal mode, the S-function must indicate explicitly that it supports multiple `exec` instances. For details, see “Supporting the Use of Multiple Instances of Referenced Models That Are in Normal Mode”.
Protected Models	A protected model cannot use noninlined S-functions directly or indirectly.

Model Architecture Requirements and Limitations

Element	Requirements and Limitations
Goto and From blocks	Goto and From blocks cannot cross model reference boundaries.
Iterator subsystems	If the referenced model contains Assignment blocks, you can place the Model block in an iterator subsystem only if the Assignment blocks are also in an iterator subsystem.
Configurable subsystems	In a configurable subsystem with a Model block, during model update, do not change the subsystem that the configurable subsystem selects.
`InitFcn` callback	An `InitFcn` callback in a top model cannot change parameters used by referenced models.
Printing referenced models	You cannot print a referenced model from a top model.

Signal Requirements and Limitations

Signal	Requirements and Limitations
0-based or 1-based indexing information propagation	In two cases, Simulink does not propagate 0-based or 1-based indexing information to referenced model root-level ports connected to blocks that: • Accept indexes (such as the Assignment block) • Produce indexes (such as the For Iterator block) An example of a block that accepts indexes is the Assignment block. An example of a block that produces indexes is the For Iterator block. The two cases result in a lack of propagation that can cause Simulink to fail to detect incompatible index connections. These two cases are: • If a root-level input port of the referenced model connects to index inputs in the model that have different 0-based or 1-based indexing settings, Simulink does not set the 0-based or 1-based indexing property of the root-level Inport block. • If a root-level output port of the referenced model connects to index outputs in the model that have different 0-based or 1-based indexing settings, Simulink does not set the 0-based or 1-based indexing property of the root-level Outport block.
Asynchronous rates	Referenced models can only use asynchronous rates if the model meets *both* of these conditions: • An external source drives the asynchronous rate through a root-level Inport block. • The root-level Inport block outputs a function-call signal. See Asynchronous Task Specification.
User-defined data type input or output	A referenced model can input or output only the user-defined data types that are fixed point or that `Simulink.DataType` or `Simulink.Bus` objects define.
Buses	If you use a virtual bus as an input or an output for a referenced model, the bus cannot contain a variable-sized signal element. See “Model Reference Requirements for Nonvirtual Buses” on page 76-55.
Signal objects	A signal that connects to a Model block is functionally the same signal outside and inside the block. Therefore, that signal is subject to the restriction that a given signal can have at most one associated signal object. See `Simulink.Signal` for more information.

Simulation Requirements and Limitations

Simulation Property	Requirements and Limitations
Continuous sample time propagation	A continuous sample time cannot be propagated to a Model block that is sample-time independent.
Sample times and solvers	The solver of the top model controls all continuous sample times in a model hierarchy. For example, for a fixed-step solver, all continuous rates in referenced models run at the fixed-step size of the top model. For information about how sample times impact solvers, see “Types of Sample Time” on page 7-13.
State initialization	To initialize the states of a model that references other models with states, specify the initial states in structure or structure with time format. For more information, see “State Information for Referenced Models” on page 72-79.
Parameter tunability	When you simulate a model that references other models, under some circumstances, you lose some tunability of block parameters (for example, the **Gain** parameter of a Gain block). For more information, see “Tunability Considerations and Limitations for Other Modeling Goals” on page 37-36.

Code Generation Requirements and Limitations

By understanding code generation requirements and limitations upfront, you are better prepared to properly set up the model hierarchy for code generation. See “Set Configuration Parameters for Code Generation of Model Hierarchies” (Simulink Coder) and “Code Generation Limitations for Model Reference” (Simulink Coder).

See Also

Related Examples

- “Compare Capabilities of Model Components” on page 22-8
- “Set Configuration Parameters for Model Hierarchies” on page 8-60
- “Code Generation Limitations for Model Reference” (Simulink Coder)

Reference Existing Models

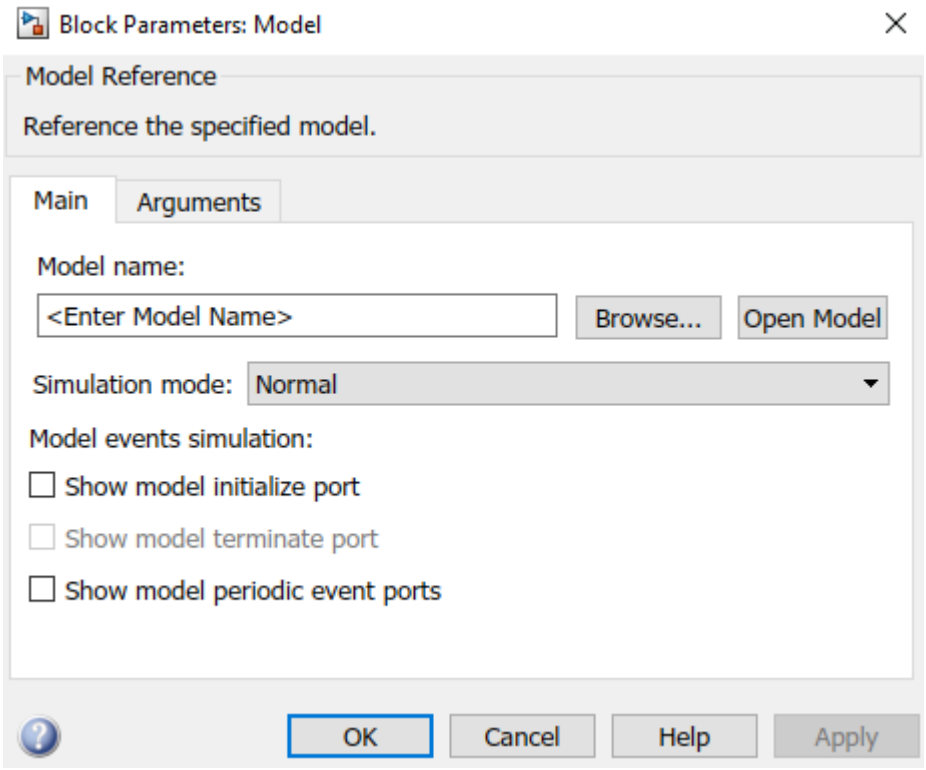
A model becomes a referenced model when a Model block in another model references it. Any model can function as a referenced model, and can continue to function as a separate model.

For a video explaining how to create model references, see Getting Started with Model Referencing.

To reference an existing model in another model, follow these steps.

1. If the folder containing the model you want to reference is not on the MATLAB path, add the folder to the MATLAB path.
2. In the referenced model, set **Total number of instances allowed per top model** to:
   - `One` to use the model at most once in a model hierarchy.
   - `Multiple` to use the model more than once in a model hierarchy. To reduce overhead, specify `Multiple` only when necessary.
   - `Zero` to preclude referencing the model.
3. Create an instance of the Model block in the parent model. The new block is initially unresolved because it does not specify a referenced model.

4. To open the block parameters dialog box, double-click the Model block.



5. Enter the name of the referenced model in the **Model name** field. This name must contain fewer than 60 characters, exclusive of the `.slx` or `.mdl` suffix.

6 Click **OK**. If the referenced model contains root-level inputs or outputs, the Model block displays corresponding input and output ports.

7 Use the Model block ports to connect referenced model signals to ports in the parent model. See “Model Reference Interface and Boundary” on page 8-31.

See Also

Blocks
Model

Related Examples

- “Reference Protected Models from Third Parties” on page 8-13
- “Convert Subsystems to Referenced Models” on page 8-18
- “Modify Referenced Models for Conditional Execution” on page 8-24

More About

- “Inspect Model Hierarchies” on page 8-28
- “Model Reference Requirements and Limitations” on page 8-6

Reference Protected Models from Third Parties

To deliver a model without revealing its intellectual property, third parties can protect the model before delivery. A protected model is a referenced model that does not support editing. The protected model author chooses whether to enable read-only view, simulation, code generation, and password protection. When an operation is password-protected, the AES-256 encryption technology protects the supporting file contents.

Note Creating a protected model requires a Simulink Coder license.

To identify protected models, look for:

- Files in the MATLAB Current Folder browser with a badge icon  and an `.slxp` extension. Protected models do not appear in the model hierarchy in the Model Explorer.
- Model blocks in the Simulink Editor with a badge icon in the lower left corner:

If available, a protected model report describes the supported functionality. To open the report, use one of these options:

- In the MATLAB Current Folder browser, right-click the protected model and click **Open Report**.
- In the Simulink Editor, right-click the protected-model badge icon on the Model block and click **Display Report**.

Load Supporting Files for Protected Model

You may receive the protected model on its own, in a project archive, or in a package.

- If you receive a project archive (`.mlproj`), extract the protected model and any supporting files. Double-click the project archive (`.mlproj`) in the Current Folder browser and specify the destination folder. Alternatively, right-click on the project archive and click **Extract Here**. The project opens, and the extracted files are in a new folder named after the project archive.
- If you receive a protected model or supporting files by another method, follow any provided instructions. For example, you might need to load a MAT-file that contains workspace definitions. You can configure a callback function, such as `LoadFcn`, to load the MAT-file automatically. See “Callbacks for Customized Model Behavior” on page 4-44.

Verify Digital Signature of Protected Model

If the author signed the protected model, verify the digital signature. In the Current Folder browser, double-click the protected model. In the **Details** pane, the **Signed by** field indicates the publisher that signed the model and whether the signature is verified by a trusted certificate authority. Verification fails in the following cases:

- The protected model was changed after it was signed.
- The protected model was not signed.
- The protected model was signed with an expired certificate.
- The protected model was self-signed with a certificate issued by the author.
- The protected model was signed with a missing or invalid certificate.
- The certificate of the CA is missing in your system or is invalid.

To verify the signature on protected models by default, in the Simulink Preferences dialog box, select **Verify digital signature of protected model before opening**.

View Protected Model Contents

Web view allows you to view this protected model information:

- System contents
- Block parameters
- Signal properties

To access the read-only view, you must have access to the licenses used in the protected model. If available, the protected model report shows the required licenses.

To open the read-only view, you can double-click the Model block referencing the protected model or the `.slxp` file in the Current Folder browser. If the read-only view is password protected, right-click the protected-model badge icon and select **Authorize**. In the **Model view** box, enter the password, then click **OK**.

You can navigate the model hierarchy by double-clicking Model or Subsystem blocks. Alternatively, you can navigate to the **View All** tab and select the system that you want to view. You cannot view the content of protected referenced models in the protected model.

To view block parameters and signal properties, select a block or a signal line, respectively.

To search in Web view, click the search button, then enter the name or value that you want to search in the box that appears. The elements of the model that the search returns appear highlighted. The search results include the name and parent of each returned element.

Test Protected Model in Isolated Environment

With the protected model, you may receive a harness model, which typically has the suffix `_harness`. A harness model provides an isolated environment for you to test the protected model. If any supporting files are missing, simulating or generating code for the harness model can help identify them.

To create a harness model, right-click the protected model file in the Current Folder browser, then click **Create Harness Model**. The created harness model is set up for simulation of the protected model.

Reference Protected Model

To reference a protected model:

1. Ensure that the protected model is on the MATLAB path.
2. If you have a harness model, copy the Model block from the harness model into your model. Otherwise, reference the protected model in a new Model block. Open the Block Parameters dialog box and enter the name of the protected model in the **Model name** field.

   When a Model block references a protected model, the **Simulation mode** of the block becomes `Accelerator`. You cannot change this mode or use this Model block in External mode.
3. If the protected model is password protected, right-click the protected-model badge icon on the Model block and click **Authorize**. In the **Model view** box, enter the password, then click **OK**.
4. Connect signal lines to the Model block that match its input and output port requirements. See “Model Reference Interface and Boundary” on page 8-31.

   Note that the protected model cannot use noninlined S-functions directly or indirectly.
5. Provide any needed model argument values. See “Parameterize a Referenced Model” on page 8-65.

If you plan to simulate the protected model, use the same platform used to create the protected model. The software stores the protected model as a compiled MEX binary.

Use Models Protected in Previous Releases

Check that the Simulink release you are using supports the protected model.

If available, the protected model report shows the release versions used to create the protected model. Use the same Simulink release that was used to create the protected model unless the protected model also supports later releases, as described by this table.

Modeling Requirement	Cross-Release Protected Model Support
Read-only view	**Supported with limitations** — R2019a and later releases • Protected model must have been created in R2018b or a later release.
Simulation in normal mode	**Supported with limitations** — R2019a and later releases • Protected model must have been created in R2018b or a later release.

Modeling Requirement	Cross-Release Protected Model Support
Simulation in accelerator mode	**Supported with limitations** — R2020a and later releases • Protected model must have been created in R2018b or a later release. • You must directly simulate the parent of the protected model. Simulating a model hierarchy that contains the parent model as a referenced model is not supported. To interactively simulate the parent model, you must open it as the top model in its own window. For more information, see “Navigate Model Hierarchies” on page 4-20. • The protected model must use Just-in-Time acceleration and must not require C code generation. For more information on acceleration modes, see “How Acceleration Modes Work” on page 35-3.
Simulation in SIL or PIL mode	**Supported with limitations** — R2020b and later releases • Protected model contains ERT, ERT-based, AUTOSAR, or GRT code generated in R2018b or a later release. You can run Model block SIL and PIL simulations that reference the protected model. For more information, see “Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code” (Embedded Coder). If ERT or AUTOSAR code in the protected model requires shared utility code, use `sharedCodeUpdate` to copy the required code to an existing shared utility code folder. For more information, see “Cross-Release Shared Utility Code Reuse” (Embedded Coder). • You cannot use `Simulink.ModelReference.ProtectedModel.addTarget` or `Simulink.ModelReference.ProtectedModel.removeTarget` to add or remove generated code from a protected model created in a different release.
Simulation in rapid accelerator mode	**Not supported**

Modeling Requirement	Cross-Release Protected Model Support
Code generation	**Supported with limitations** — R2020b and later releases • Protected model contains ERT, ERT-based, AUTOSAR, or GRT code generated in R2018b or a later release, with a `Model reference` code interface. For more information, see “Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code” (Embedded Coder). If ERT or AUTOSAR code in the protected model requires shared utility code, use `sharedCodeUpdate` to copy the required code to an existing shared utility code folder. For more information, see “Cross-Release Shared Utility Code Reuse” (Embedded Coder). • You cannot use `Simulink.ModelReference.ProtectedModel.addTarget` or `Simulink.ModelReference.ProtectedModel.removeTarget` to add or remove generated code from a protected model created in a different release.

See Also

`Simulink.ProtectedModel.getPublisher` | `Simulink.ProtectedModel.verifySignature`

More About

- “Reference Existing Models” on page 8-11
- “Inspect Model Hierarchies” on page 8-28
- “Protect Models to Conceal Contents” (Simulink Coder)

Convert Subsystems to Referenced Models

Model reference offers benefits for modeling large, complex systems and for team-based development. Many large models use a combination of subsystems and referenced models. To decide whether to convert a subsystem to a referenced model, see “Choose Among Types of Model Components” on page 22-4.

Prepare Subsystem for Conversion

Preparing a subsystem for conversion can eliminate or reduce the number of issues identified during conversion. Addressing these issues before conversion can be more efficient than switching repeatedly between the diagnostic messages and the Simulink Editor.

To prepare the subsystem:

1. Set the **Signal resolution** configuration parameter to `Explicit only` or `None`.

   Tip You can automatically fix this issue during conversion.

2. Configure the Subsystem block interface.

   Tip You can automatically fix these interface issues during conversion.

Subsystem Interface	What to Look For	Model Modification
Goto or From blocks	Goto or From blocks crossing the subsystem boundary	Replace From blocks that have a corresponding Goto block that crosses the subsystem boundary with an Inport block. Replace each Goto block that has corresponding From blocks that cross the subsystem boundary with an Outport block. Connect the Inport and Outport blocks to the corresponding subsystem ports.
Data stores	Data Store Memory blocks accessed by Data Store Read or Data Store Write blocks from outside of the subsystem	Replace the Data Store Memory block with a global data store. Define a global data store using a `Simulink.Signal` object. For details, see “Data Stores with Signal Objects” on page 73-16.

Subsystem Interface	What to Look For	Model Modification
Tunable parameters	Global tunable parameters in the dialog box opened using the **Configuration Parameters** > **Code Generation** > **Optimization** > **Configure** button	To create a `Simulink.Parameter` object for each tunable parameter, use `tunablevars2parameterobjects`. The `Simulink.Parameter` objects must have a storage class other than `Auto`. For more information, see “Parameterize Instances of a Reusable Referenced Model” on page 8-64 and “Tunable Parameters” on page 10-69.

3 Configure the Subsystem block contents.

Subsystem Configuration	What to Look For	Model Modification
Block execution order	Virtual subsystem that does not force contained blocks to execute consecutively.	Select the Subsystem block, and then on the **Subsystem Block** tab, select **Is Atomic Subsystem**.
Function calls	Function-call signals that cross virtual subsystem boundaries	Move the Function-Call Generator block into the subsystem that you want to convert. **Note** If you convert an export-function subsystem, then you do not need to move the Function-Call Generator block.
	Function-call outputs	Change the function-call outputs to data triggers.
	Wide function-call ports	Eliminate wide signals for function-call subsystems.
Sample times	An Inport block sample time that does not match the sample time of the block driving the Inport block	Insert Rate Transition blocks where appropriate.
Inport blocks	Merged Inport blocks	Configure the model to avoid merged Inport blocks. See the Merge block documentation.
Constant blocks	Constant blocks that provide input for subsystems	Move Constant blocks into the subsystem.
Buses	Buses that enter and exit a subsystem	Match signal names and bus element names for blocks inside the subsystem. To find signal names that do not match bus element names, use the **Signal label mismatch** diagnostic.

Subsystem Configuration	What to Look For	Model Modification
	Duplicate signal names in buses	Make signal names of bus elements unique.
	Signal names that are not valid MATLAB identifiers. A valid identifier is a character vector that meets these conditions: • The name contains letters, digits, or underscores. • The first character is a letter. • The length of the name is less than or equal to the value returned by the `namelengthmax` function.	Change any invalid signal names to be valid MATLAB identifiers.

4 Make sure that the model containing the subsystem that you want to convert compiles successfully.

Convert Subsystems to Referenced Models

To convert a subsystem to a referenced model, open the **Model Reference Conversion Advisor** by selecting a Subsystem block and, on the **Subsystem Block** tab, selecting **Convert > Model Block**. The Model Reference Conversion Advisor lets you interactively specify conversion parameters and fix issues that the advisor finds.

To make the conversion process faster:

- In the Model Reference Conversion Advisor, select **Fix errors automatically (if possible)**. This option automatically fixes some conversion issues, but you do not control the fixes.
- Close any open Scope block windows before starting the conversion.

To leave the Subsystem block in place and create a separate model from the contents of the Subsystem block, clear **Replace the content of a subsystem with a Model block**.

To compare top-model simulation results before and after conversion:

- Enable signal logging for output signals of interest.
- Select **Check simulation results after conversion** and **Replace the content of a subsystem with a Model block**.
- Set the **Stop time**, **Absolute tolerance**, and **Relative tolerance**.
- Set the **Model block simulation mode** option in the advisor to the same simulation mode as the original model.

After you set the conversion settings, click **Convert** and address any identified issues.

Alternatively, in the MATLAB command window, use the `Simulink.SubSystem.convertToModelReference` function. You can convert multiple Subsystem

blocks using one `Simulink.SubSystem.convertToModelReference` command. However, you cannot convert a parent subsystem and a child of that subsystem at the same time.

Conversion Results

After all conversion checks pass, Simulink:

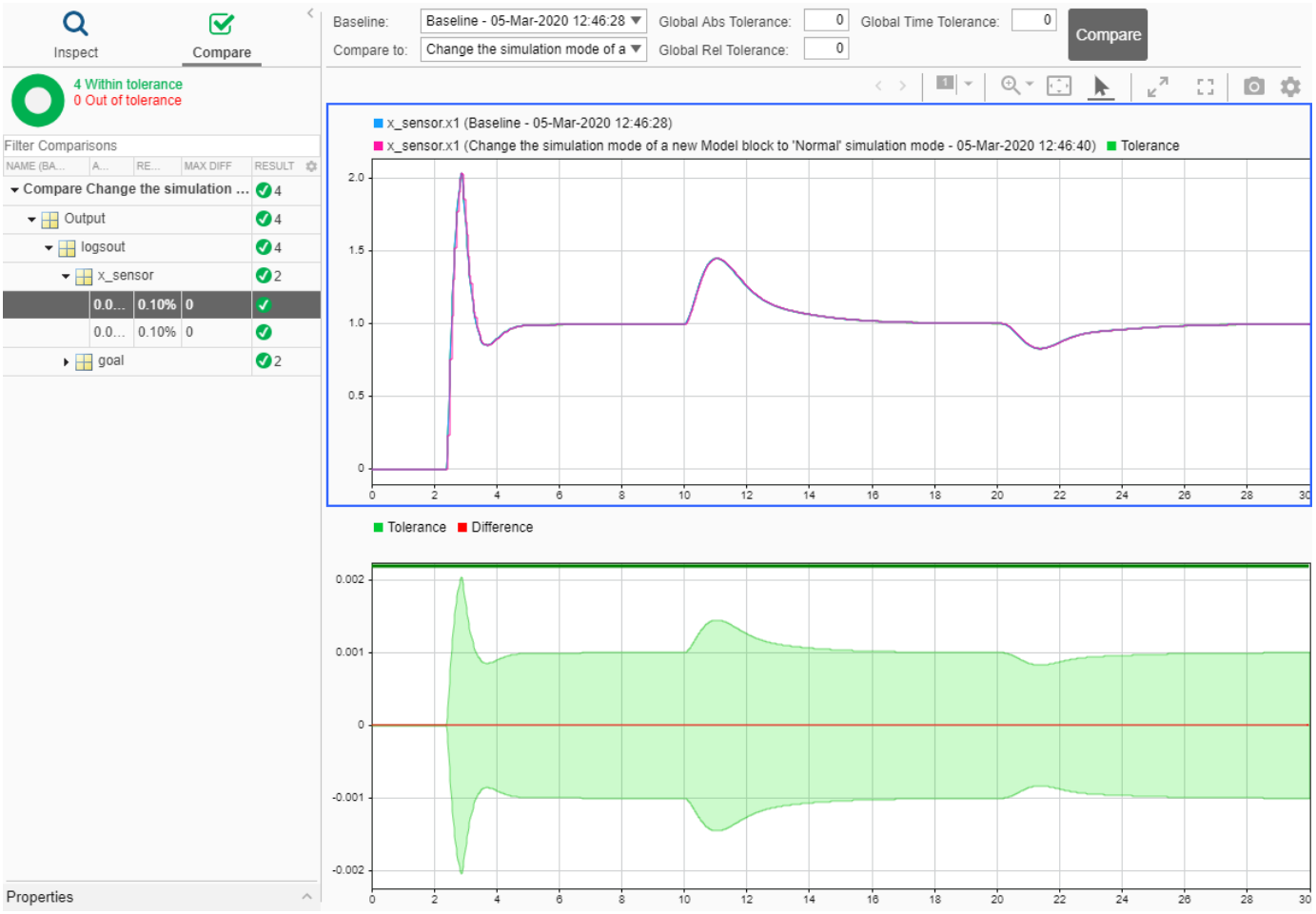
- Creates a referenced model from the subsystem.
- Creates the `Simulink.Bus` objects, `Simulink.Signal` objects, and tunable parameters that the referenced model requires.
- By default, replaces the Subsystem block with a Model block that references the new model.
- Inserts the Model block in a Subsystem block if the automatic fixes added ports to the Model block interface.
- Creates an HTML conversion summary report in the `slprj` folder. This report summarizes the results of the conversion process, including the results of the fixes that the advisor performed. This report also describes the elements that it copies.
- Optionally checks the consistency of simulation results before and after conversion.

Simulink copies the following elements from the original model to the new referenced model.

- **Configuration set** — If the parent model uses:
  - A configuration set that is not a referenced configuration set, the advisor copies the entire configuration set to the referenced model
  - A referenced configuration set, then both the parent and referenced models use the same referenced configuration set
- **Variables** — The advisor copies only the model workspace variables that the subsystem used in the original model to the model workspace of the referenced model. If the model that contained the subsystem uses a data dictionary, then the referenced model uses the same data dictionary.
- **Requirements links** — The advisor copies requirements links created with Simulink Requirements software to the Model block from the original Subsystem block.

Compare Simulation Results Before and After Conversion

After you successfully complete conversion, use the **Click here to view the comparison results** link. The results display in the **Simulation Data Inspector**. A green check mark indicates that simulation results are within tolerance between the baseline model and the model with the new referenced model.



For more information, see “Compare Simulation Data” on page 29-130.

Revert Conversion

If you are not satisfied with the conversion results, you can restore the model to its initial state. Use one of these approaches:

- At any point during the conversion, select **File > Load Restore Point**.
- After you successfully run the **Complete conversion** check, use the **Click here to restore the original model** link.

Integrate Referenced Model into Parent Model

After you complete the conversion, update the model as necessary to meet your modeling requirements. For example, you can manually replace a Subsystem block with a Model block that references the created referenced model.

If you want to simulate the model with external data, check that the root Inport blocks in the new referenced model have the appropriate **Interpolate data** parameter setting. See the documentation for the **Interpolate data** parameter of the Inport block.

See Also

Blocks
Model

More About

- “Reference Existing Models” on page 8-11
- “Choose Among Types of Model Components” on page 22-4
- “Model Reference Requirements and Limitations” on page 8-6
- “Inspect Model Hierarchies” on page 8-28

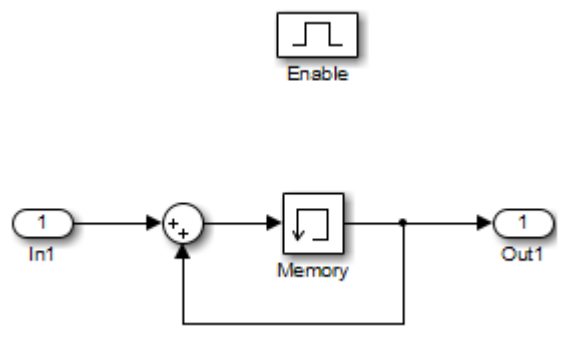
Modify Referenced Models for Conditional Execution

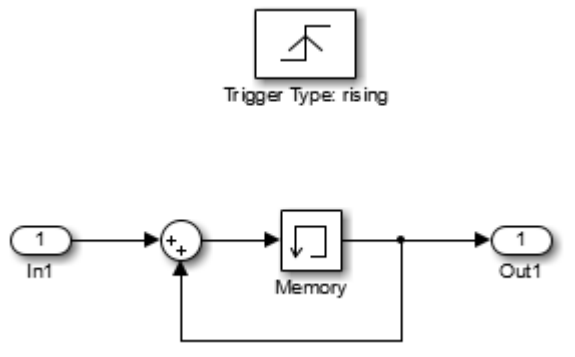
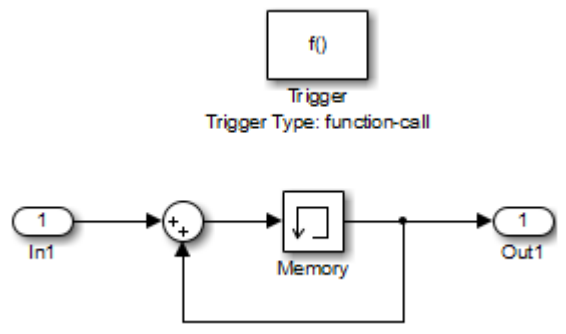
A conditionally executed referenced model, or conditional model, allows you to control its execution with an external signal. The external signal, called the control signal, is attached to the control input port. Conditional models are useful when you create complex model hierarchies that contain components whose execution depends on other components.

Conditional Models

You can set up referenced models to execute conditionally, similar to conditional subsystems. For information about conditional subsystems, see “Conditionally Executed Subsystems Overview” on page 10-3.

Simulink software supports these conditional model types:

Conditional Model	Description
Enabled	An enable port executes a referenced model at each simulation step for which the control signal has a positive value. To add an enable port to a Model block, insert an Enable block in the referenced model. This image displays the contents of a simple enabled referenced model.  To see an example of an enabled *subsystem*, see `enablesub`. A corresponding enabled referenced model uses the same blocks as are in the enabled subsystem.

Conditional Model	Description
Triggered	A trigger port executes a referenced model each time a trigger event occurs. To add a trigger port to a Model block, insert a Trigger block in the referenced model. This image displays the contents of a simple triggered referenced model.  For an example of a triggered model, see model `sldemo_mdlref_datamngt`.
Triggered and Enabled	A Model block can have both trigger and enable ports. If the enable control signal has a positive value at the time step for which a trigger event occurs, a triggered and enabled model executes once.
Function-Call	A function-call port executes a referenced model each time a function-call event occurs. To add a function-call port to a Model block, insert a Trigger block in the referenced model. Then, open the Block Parameters dialog box and set the **Trigger type** to `function-call`. A Stateflow chart, a Function-Call Generator block, a Hit Crossing block, or an appropriately configured custom S-function can provide function-call events. See “Using Function-Call Subsystems” on page 10-34. This image displays the contents of a simple function-call referenced model.  For an example of a function-call model, see model `sldemo_mdlref_fcncall`.

Requirements for Conditional Models

Conditional models must meet the requirements for:

- Conditional subsystems (see “Conditionally Executed Subsystems and Models”)
- Referenced models (see “Reference Existing Models” on page 8-11)

Conditional models must also meet the requirements specific to each type of conditional model.

Conditional Model	Requirements
Enabled	• Multi-rate enabled models cannot use multi-tasking solvers. Use single-tasking. • For models with enable ports at the root, if the model uses a fixed-step solver, the fixed-step size of the model must not exceed the rate for any block in the model. • Signal attributes of the enable port in the referenced model must be consistent with the input that the Model block provides to that enable port.
Triggered	Signal attributes of the trigger port in the referenced model must be consistent with the input that the Model block provides to that trigger port.
Triggered and Enabled	See requirements for triggered models and enabled models.
Function-Call	• A function-call model cannot have an output port driven only by Ground blocks, including hidden Ground blocks inserted by Simulink. To meet this requirement, do the following: 1 Insert a Signal Conversion block into the signal connected to the output port. 2 Enable the **Exclude this block from 'Block reduction' optimization** option of the inserted block. • The parent model must trigger the function-call model at the rate specified by the **Configuration Parameters** > **Solver** `'Fixed-step size'` option if the function-call model meets both these conditions: • It specifies a fixed-step solver. • It contains one or more blocks that use absolute or elapsed time. Otherwise, the parent model can trigger the function-call model at any rate. • A function-call model must not have direct internal connections between its root-level input and output ports. Simulink does not honor the `None` and `Warning` settings for the **Invalid root Inport/Outport block connection** diagnostic for a referenced function-call model. It reports all invalid root port connections as errors. • If the **Sample time type** is `periodic`, the sample-time period must not contain an offset. • The signal connected to a function-call port of a Model block must be scalar.

Modify a Referenced Model for Conditional Execution

1 At the root level of the referenced model, insert one of the following blocks:

Type of Model	Blocks to Insert
Enabled	Enable
Triggered	Trigger
Triggered and Enabled	Trigger and Enable
Function-Call	Trigger

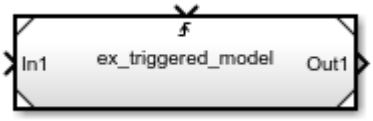
For an enabled model, go to Step 3.

2 For the Trigger block, set the **Trigger type** parameter:

Type of Model	Trigger Type Parameter Setting
Triggered Triggered and enabled	One of the following: • `rising` • `falling` • `either`
Function-Call	`function-call`

3 Use the Model block ports to connect the referenced model to other ports in the parent model.

- The top of the Model block displays an icon that corresponds to the control signal type expected by the referenced model. For a triggered model, the top of the Model block displays this icon.



See Also

Blocks
Enable | Function-Call Subsystem | Trigger

More About

- “Simulate Conditionally Executed Referenced Models” on page 8-43
- “Conditionally Executed Subsystems Overview” on page 10-3
- “Export-Function Models Overview” on page 10-97

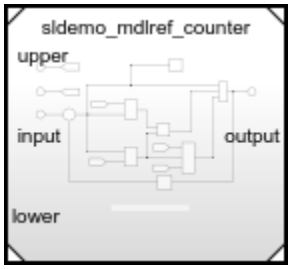
Inspect Model Hierarchies

To better understand a model hierarchy, you can use Simulink tools, functions, parameters, or preferences to:

- Preview model contents
- Visualize the model hierarchy
- List referenced models and Model blocks
- Display model version numbers

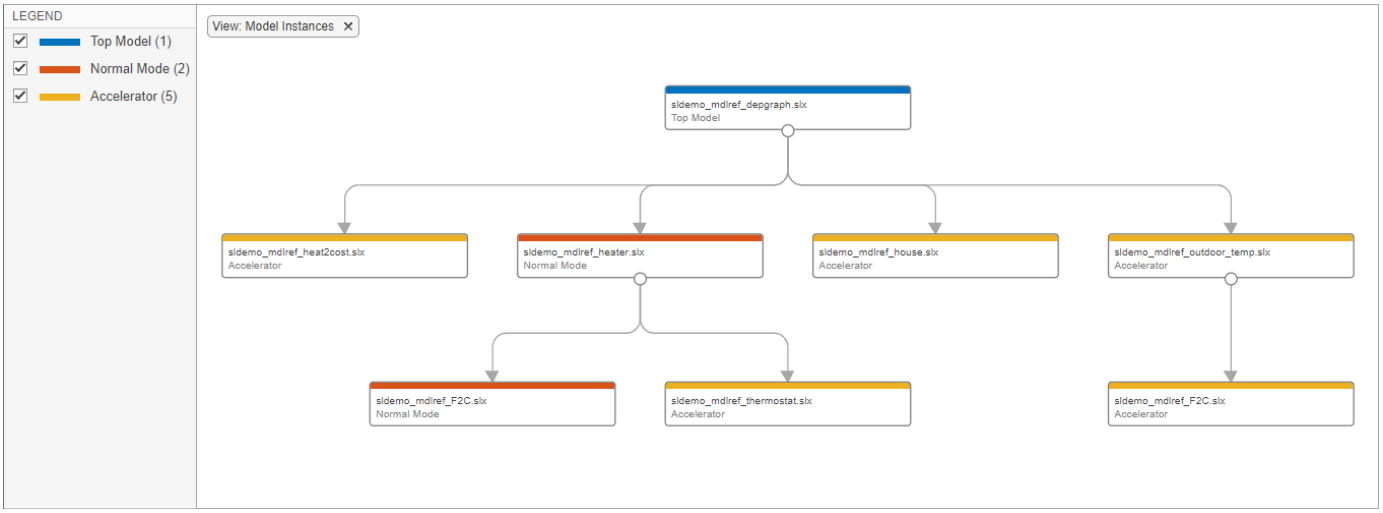
Content Preview

Content preview displays a representation of the contents of a referenced model on the Model block. This preview helps you to understand at a glance the kind of processing performed by the referenced model without opening the referenced model. See “Preview Content of Model Components” on page 1-33.



Model Dependency Graph

The Dependency Analyzer shows the structure of the model hierarchy and lets you open constituent models. The **Model Instances** view displays Model blocks differently to indicate Normal, Accelerator, SIL, and PIL modes. See “Analyze Model Dependencies” on page 17-40.



The `depview` function opens the model dependency graph.

List of Model References

The `find_mdlrefs` function lists all models that are directly or indirectly referenced by a given model. This function also provides the paths to the related Model blocks.

Find Referenced Models in Model Hierarchy

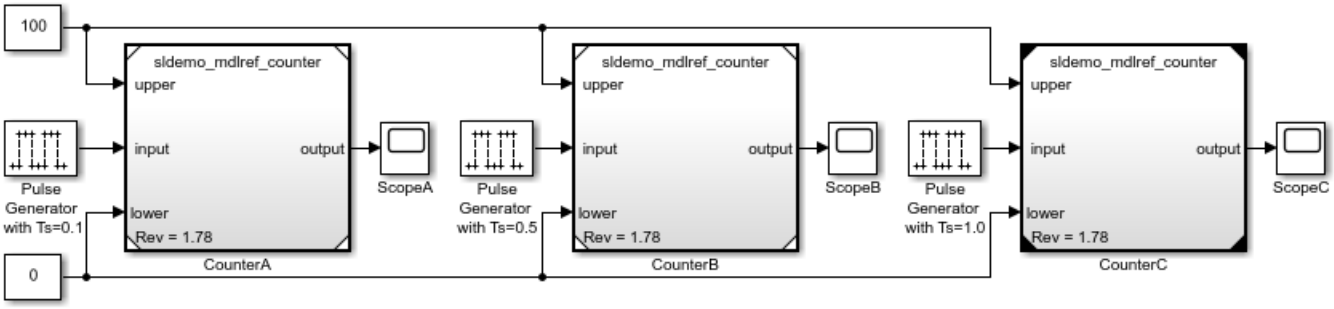
Find referenced models and Model blocks for all models referenced by the specified model.

```
load_system('sldemo_mdlref_basic');
[myModels,myModelBlks] = find_mdlrefs('sldemo_mdlref_basic')
```

```
myModels = 2x1 cell
    {'sldemo_mdlref_counter'}
    {'sldemo_mdlref_basic'  }


myModelBlks = 3x1 cell
    {'sldemo_mdlref_basic/CounterA'}
    {'sldemo_mdlref_basic/CounterB'}
    {'sldemo_mdlref_basic/CounterC'}
```

Model Version Numbers

To display the version numbers of referenced models, in the parent model, on the **Debug** tab, click **Information Overlays**, then under **Blocks**, select **Ref. Model Version**. The version of each Model block instance appears on each Model block.



For information on model versions, see “Manage Model Versions and Specify Model Properties” on page 4-57.

See Also

Functions

`depview` | `find_mdlrefs`

More About

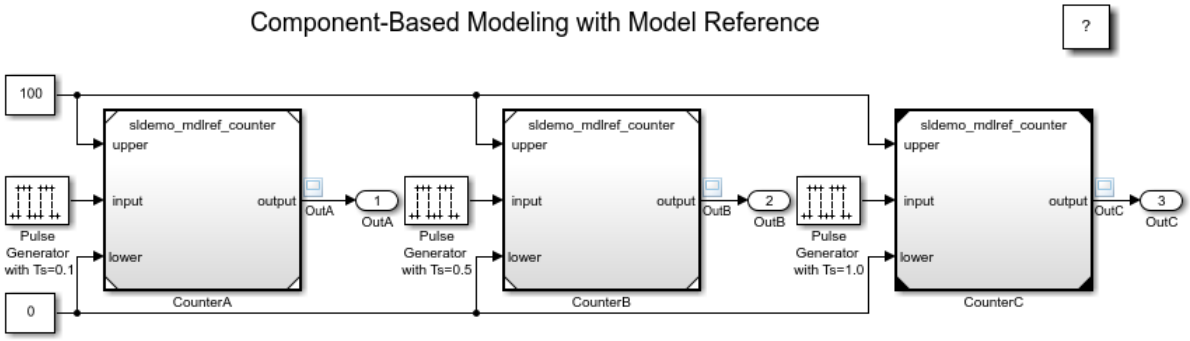
- “Preview Content of Model Components” on page 1-33
- “Analyze Model Dependencies” on page 17-40

- “Manage Model Versions and Specify Model Properties” on page 4-57
- “Export Signal Data Using Signal Logging” on page 72-41

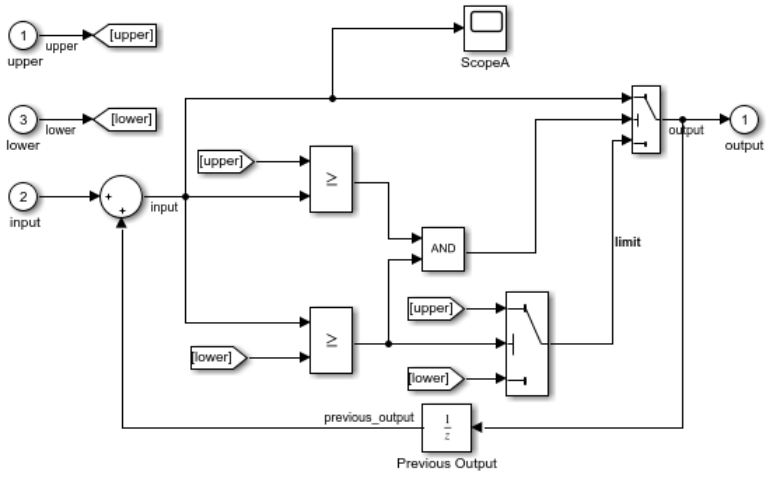
Model Reference Interface and Boundary

A Model block has input, output, and control ports that correspond to root-level input, output, and control ports of the model it references. A referenced model can include Inport, Outport, In Bus Element, Out Bus Element, Trigger, and Enable blocks to get input from the parent model and to provide output to the parent model. The input signals for the Model block must be valid for the corresponding input blocks of the referenced model. The output signals for the Model block are the referenced model root-level output block signals.

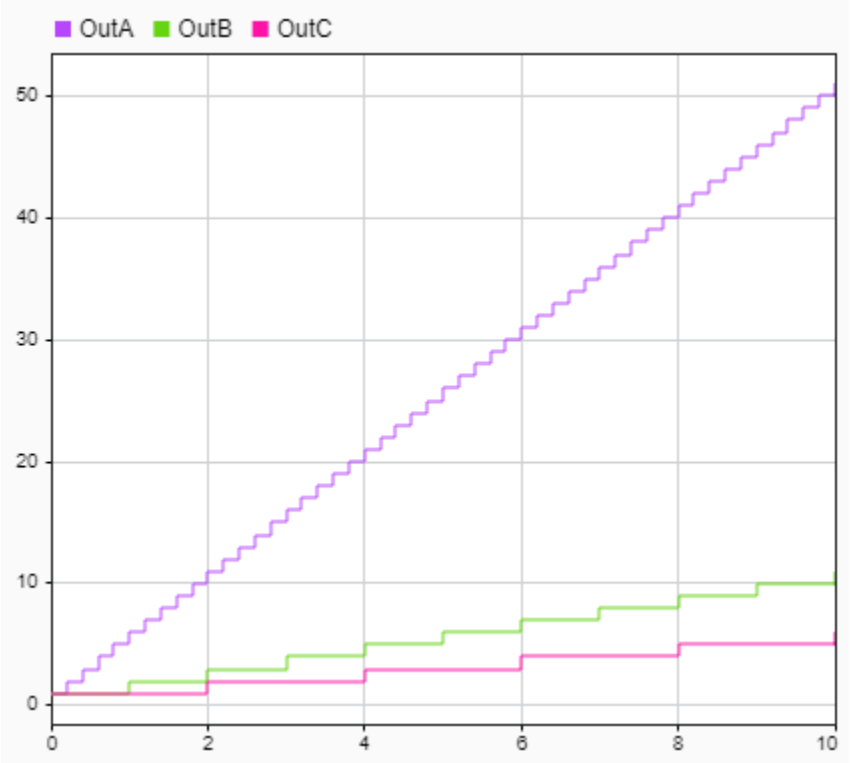
In `sldemo_mdlref_basic`, each Model block has three inputs: two Constant blocks and a Pulse Generator block. Each Model block has one output signal logged to a scope. Because the input signal from each Pulse Generator block uses a different sample time, the output signal from each Model block differs for each model instance.



To connect to the parent model, referenced model `sldemo_mdlref_counter` includes three Inport blocks (**upper**, **lower**, and **input**) and one Outport block (**output**).



To view how the output signal for each Model block differs, you can use the **Simulation Data Inspector**.



Refresh Model Blocks

Refreshing a Model block updates its internal representation to reflect changes to the interface of the referenced model. For example, when the referenced model gains or loses a port, refreshing the Model block updates its ports.

When a referenced model is loaded, the Model blocks that reference it automatically refresh. When a referenced model is not loaded, the corresponding Model blocks refresh when you perform actions such as:

- Opening the parent model
- Selecting a Model block
- Simulating the model hierarchy
- Generating code for the model hierarchy

When you select a Model block, you can refresh all Model blocks in a model hierarchy by clicking the **Refresh** button arrow on the **Model Block** tab, then clicking **Refresh Blocks**.

To be notified when Simulink detects Model blocks that might not match their referenced models, change the default setting for these diagnostic configuration parameters:

- **Model block version mismatch**
- **Port and parameter mismatch**

When these configuration parameters are set to `error` for a model, the Model blocks in that model do not automatically refresh. To refresh a Model block when these configuration parameters are set to `error`:

- Select the Model block. On the **Model Block** tab, click **Refresh**.
- Use the `Simulink.ModelReference.refresh` function.

Signal Propagation

Signal attributes in a referenced model are independent from the context of the Model block. For example, signal dimensions and data types do not propagate across the Model block boundary. To define signal attributes in a referenced model, define block parameters for root-level Inport and In Bus Element blocks.

For signals that connect to Outport blocks to propagate out of a referenced model to the parent model, the signal names must explicitly appear on the signal lines.

For virtual buses that cross model reference boundaries, use In Bus Element and Out Bus Element blocks.

For nonvirtual buses that cross model reference boundaries, use Inport and Outport blocks that specify the same bus object as the related buses in the parent model. See “Nonvirtual Buses at Model Interfaces” on page 76-55.

For an example of a model hierarchy that uses buses, see `sldemo_mdlref_bus`.

A referenced model can only provide input or get output for user-defined data types that are fixed point or that `Simulink.DataType` or `Simulink.Bus` objects define.

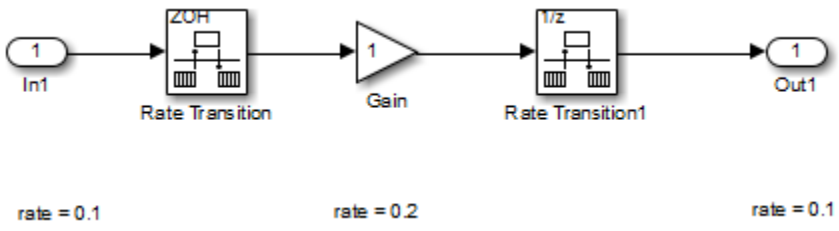
Signal Logging in Referenced Models

In a referenced model, you can log any signal configured for signal logging. Use the Signal Logging Selector to select a subset or all the signals configured for signal logging in a model hierarchy. For details, see “Override Signal Logging Settings” on page 72-57.

You can use the Simulation Data Inspector to view and analyze signals logged in referenced models. You can view signals on multiple plots, zoom, and use data cursors to understand and evaluate the data. Also, you can compare signal data from multiple simulations. For an example of viewing signals with referenced models, see “Viewing Signals in Model Reference Instances”.

Sample Time Requirements

The first nonvirtual block that connects to a referenced model root-level input or output block must have the same sample time as the related port. If the sample times are different, use Rate Transition blocks to match input and output sample times, as shown in this diagram.



Share Data Among Referenced Model Instances

By default, each Model block instance reads from and writes to a separate copy of the signals and block states in the model. Therefore, the instances do not interact with each other through shared signal or state data.

To share a piece of data between all of the instances (for example, an accumulator or a fault indicator), model the data as a data store.

- To restrict access to data so that only the blocks in the referenced model can read from and write to it, use a Data Store Memory block in the model and select the **Share across model instances** parameter. For an example, see “Share Data Store Between Instances of a Reusable Algorithm”.
- To allow access to data outside the referenced model, use a global data store, which is a `Simulink.Signal` object in the base workspace or a data dictionary. Data outside the referenced model can be in the parent model or in other referenced models.

For more information about data stores, see “Model Global Data by Creating Data Stores” on page 73-10.

See Also

Blocks
In Bus Element | Inport | Model | Out Bus Element | Outport

More About

- “Nonvirtual Buses at Model Interfaces” on page 76-55
- “Model Reference Requirements and Limitations” on page 8-6
- “Modify Referenced Models for Conditional Execution” on page 8-24

Referenced Model Sample Times

How Sample-Time Inheritance Works for Model Blocks

The sample times of a Model block are the sample times of the model that it references. If the referenced model must run at specific rates, the model specifies the required rates. Otherwise, the referenced model inherits its sample time from the parent model.

Placing a Model block in a triggered, function call, or iterator subsystem relies on the ability to inherit sample times. Also, allowing a Model block to inherit sample time maximizes its reuse potential. For example, a model can fix the data types and dimensions of all its input and output signals. You could reuse the model with different sample times (for example, discrete at 0.1 or discrete at 0.2, triggered).

Conditions for Inheriting Sample Times

A referenced model inherits its sample time if the model:

- Does not have any continuous states
- Specifies a fixed-step solver and the **Fixed-step size** is `auto`
- Contains no blocks that specify sample times (other than inherited or constant)
- Does not contain any S-functions that use their specific sample time internally
- Has only one sample time (not counting constant and triggered sample time) after sample time propagation
- Does not contain any blocks, including Stateflow charts, that use absolute time, as listed in “Blocks That Depend on Absolute Time” on page 8-36
- Does not contain any blocks whose outputs depend on inherited sample time, as listed in “Blocks Whose Outputs Depend on Inherited Sample Time” on page 8-36.

You can use a referenced model that inherits its sample time anywhere in a parent model. By contrast, you cannot use a referenced model that has intrinsic sample times in a triggered, function call, or iterator subsystem. To avoid rate transition errors, ensure that blocks connected to a referenced model with intrinsic samples times operate at the same rates as the referenced model.

Note A continuous sample time cannot be propagated to a Model block that is sample-time independent.

For more information, see “Blocks Whose Outputs Depend on Inherited Sample Time” on page 8-36.

Determining Sample Time of a Referenced Model

To determine whether a referenced model can inherit its sample time, set the **Periodic sample time constraint** configuration parameter to `Ensure sample time independent`. If the model is unable to inherit sample times, this setting causes Simulink to display an error message when building the model. See “Periodic sample time constraint” for more about this option.

To determine the intrinsic sample time of a referenced model, or the fastest intrinsic sample time for multirate referenced models:

1. Update the model that references the model
2. Select a Model block within the parent model
3. Enter the following at the MATLAB command line:

```
get_param(gcb, 'CompiledSampleTime')
```

Blocks That Depend on Absolute Time

The following Simulink blocks depend on absolute time, and therefore preclude a referenced model from inheriting sample time:

- Backlash (only when the model uses a variable-step solver and the block uses a continuous sample time)
- Chirp Signal
- Clock
- Derivative
- Digital Clock
- Discrete-Time Integrator (only when used in triggered subsystems)
- From File
- From Workspace
- Pulse Generator
- Ramp
- Rate Limiter
- Repeating Sequence
- Signal Generator
- Sine Wave (only when the **Sine type** parameter is `Time-based`)
- `stateflow` (when the chart uses absolute-time temporal logic, or the reserved word `t` to reference time)
- Step
- To File
- To Workspace (only when logging to `Timeseries` or `Structure With Time` format)
- Transport Delay
- Variable Time Delay
- Variable Transport Delay

Some blocks other than Simulink blocks depend on absolute time. See the documentation for the blocksets that you use.

Blocks Whose Outputs Depend on Inherited Sample Time

Using a block whose output depends on an inherited sample time in a referenced model can cause simulation to produce unexpected or erroneous results. When building a referenced model that does not need a specified rate, Simulink checks for blocks whose outputs are functions of the inherited sample time. This checking includes examining S-Function blocks. If Simulink finds any such blocks, it specifies a default sample time. If you have set the **Configuration Parameters** > **Solver** >

Periodic sample time constraint to `Ensure sample time independent`, Simulink displays an error. See “Periodic sample time constraint” for more about this option.

The outputs of the following built-in blocks depend on inherited sample time. The outputs of these blocks preclude a referenced model from inheriting its sample time from the parent model:

- Discrete-Time Integrator
- From Workspace (if it has input data that contains time)
- Probe (if probing sample time)
- Rate Limiter
- Rate Limiter Dynamic
- Sine Wave

Simulink assumes that the output of an S-function does not depend on inherited sample time unless the S-function explicitly declares the contrary. See “Specify S-Function Sample Times” for information on how to create S-functions that declare whether their output depends on their inherited sample time.

In referenced models that inherit their sample time, avoid S-functions in referenced models that fail to declare whether output depends on inherited sample time. Excluding those kinds of S-functions helps to avoid simulation errors. By default, Simulink warns you if your model contains such blocks when you update or simulate the model. See “Unspecified inheritability of sample time” for details.

Sample Time Consistency

Use consistent sample time rates to promote the reliable use of a model referenced by another model. Make the rates of root Inport and Outport blocks in a referenced model consistent with the rates of blocks reading from and writing to those blocks. Simulink generates an error when there are sample time mismatches between:

- The sample times of root Inport blocks and the sample times of blocks to which the Inport block inputs.
- The sample times of root Outport blocks and the sample times of blocks that input to the Outport block.

To address an error that flags a sample time inconsistency in a referenced model, you can use one of these approaches.

Top-Level Inport or Outport Block Sample Time	Possible Solution
Different from all the blocks to which it connects, and those blocks all have the same sample time as each other	Set the sample time of the Inport or Outport block so that it matches the sample time of the block to which it connects.
Different from one or more blocks and the same as one or more blocks	For blocks that do not match the Inport or Outport block, insert Rate Transition blocks on the signal that connects to the Inport or Outport block.

Sample Rates and Solvers

The solver of the top model controls all continuous sample times in a model hierarchy. For example, for a fixed-step solver, all continuous rates in referenced models run at the fixed-step size of the top model. For information about how sample times impact solvers, see “Types of Sample Time” on page 7-13.

See Also

Related Examples

- “Specify Sample Time” on page 7-3
- “View Sample Time Information” on page 7-9

More About

- “What Is Sample Time?” on page 7-2
- “Types of Sample Time” on page 7-13

Choose Simulation Modes for Model Hierarchies

When you simulate a model hierarchy, you should consider how top models and referenced models execute.

- Top model — Supports all Simulink simulation modes. To speed up execution of a top model, you can use Simulink accelerator or rapid accelerator mode.
- Referenced model — Although you can specify any simulation mode for a model, when you reference that model, the Model block for each instance of the referenced model controls the simulation mode of the instance. The simulation mode of a parent model can override the simulation mode of a Model block.

For information on simulation modes, see “Choosing a Simulation Mode” on page 35-10 and “How Acceleration Modes Work” on page 35-3.

Model Reference Simulation Modes

You can specify any of these simulation modes for a Model block:

- Normal
- Accelerator
- Software-in-the-loop (SIL) — requires Embedded Coder
- Processor-in-the-loop (PIL) — requires Embedded Coder

When you choose between normal and accelerator mode, you must make a tradeoff between flexibility and speed. Normal mode supports more Simulink and Stateflow features in referenced models, such as scopes, port value display, and debugging tools. Accelerator mode supports fewer features in referenced models, but simulates model hierarchies faster.

Modeling Requirement	Normal Mode	Accelerator Mode
Simulation speed	Models execute slower in normal mode than accelerator mode. However, referenced models that execute in normal mode do not delay simulation to build and compile simulation targets because normal mode executes referenced models interpretively.	Models execute faster in accelerator mode than normal mode. Before simulating the model, Simulink must build and compile simulation targets, which can be undesirable for prototyping. For more information, see “Manage Simulation Targets for Referenced Models” on page 8-50.

Modeling Requirement	Normal Mode	Accelerator Mode
Debugging	With the Simulink Debugger, you can set a breakpoint inside a referenced model that executes in normal mode. For more information, see “Simulink Debugger”. With the Simulink Profiler, you can enable profiling for a referenced model that executes in normal mode. Enabling profiling on a parent model does not enable profiling for referenced models. See “How Profiler Captures Performance Data” on page 31-5.	For referenced models that execute in accelerator mode, specifications made and actions taken by the Simulink Debugger and Simulink Profiler are ignored.
Testing	Simulink Coverage™ model coverage analysis supports referenced models that execute in normal mode.	Simulink Coverage model coverage analysis ignores referenced models that execute in accelerator mode.
Tunability	You can tune block parameters during simulation for referenced models that execute in normal mode.	If a referenced model that executes in accelerator mode uses variables in the base workspace or a data dictionary to set parameter values, you can tune the values of those variables. For more information see “Tunability Considerations and Limitations for Other Modeling Goals” on page 37-36.
Data logging	Data logging provides extensive support for referenced models that execute in normal mode.	For referenced models that execute in accelerator mode, To Workspace blocks log data only if they use the `Timeseries` format for saving.
Data visualization	You can view instance-specific simulation results with the Simulation Data Inspector. These visualization methods show data for only one instance of a referenced model that executes in normal mode. • Scope, Floating Scope, and Scope Viewer blocks in the referenced model • Runtime displays, such as port values For more information, see “Simulate Multiple Referenced Model Instances in Normal Mode” on page 8-44.	You can view instance-specific simulation results with the Simulation Data Inspector. These visualization methods show no data for referenced models that execute in accelerator mode. • Scope, Floating Scope, and Scope Viewer blocks in the referenced model • Runtime displays, such as port values Top model Scope blocks can display data for referenced models that execute in accelerator mode if you use the Signal & Scope Manager to add test points in the referenced model. Adding or removing a test point requires rebuilding the model reference simulation target for a model.

Modeling Requirement	Normal Mode	Accelerator Mode
Diagnostics	Configuration parameter settings are applied as specified.	For models referenced in accelerator mode, Simulink can ignore some configuration parameter settings. For details, see “Diagnostics That Are Ignored in Accelerator Mode” on page 8-62.
Runtime checks	Runtime checks are enabled.	Some blocks include runtime checks that are disabled when you include the block in a referenced model in accelerator mode. Examples of these blocks include Assignment, Selector, MATLAB Function, and MATLAB System blocks.
Linearization analysis and optimization	Normal mode allows block-by-block linearization of a referenced model, which achieves an accurate linearization.	In accelerator mode, discrete states of model references are not exposed to linearization. These discrete states are not perturbed during linearization and, therefore, are not truly free in the trimming process. The outputs of random blocks are not kept constant during trimming. Outputs that are not kept constant can affect the optimization process.
Extrinsic functions	A MATLAB Function block in a referenced model that executes in normal mode can call MATLAB functions that are declared extrinsic for code generation.	A MATLAB Function block in a referenced model that executes in accelerator mode cannot call MATLAB functions that are declared extrinsic for code generation.
S-Functions	Referenced models that execute in normal mode support more S-functions than referenced models that execute in accelerator mode. For more information see, “S-Functions in Referenced Models” on page 8-7.	Referenced models that execute in accelerator mode support fewer S-functions than referenced models that execute in normal mode. For more information see, “S-Functions in Referenced Models” on page 8-7.

SIL and PIL simulation modes execute referenced models by generating production code for embedded processors. SIL mode provides a convenient alternative to PIL simulation because it can run on a host platform when the target hardware is not available. For more information, see “SIL and PIL Simulations” (Embedded Coder).

Overridden Simulation Modes

The simulation mode of the parent model can override the simulation mode of a Model block. This table shows which simulation mode Simulink uses for a referenced model instance based on the simulation mode of the parent model and related Model block.

Simulation Mode Used by Parent Model	Simulation Mode of Model Block	
	Normal	**Accelerator**
Normal	Compatible — Referenced model simulates in normal mode.	Compatible — Referenced model simulates in accelerator mode.
Accelerator	Overridden — Referenced model simulates in accelerator mode.	Compatible — Referenced model simulates in accelerator mode.
Rapid accelerator (top model only)		

For information on SIL and PIL, see “Simulation Mode Override Behavior in Model Reference Hierarchy” (Embedded Coder).

See Also

Related Examples

- “Manage Simulation Targets for Referenced Models” on page 8-50
- “Model Reference Requirements and Limitations” on page 8-6
- “Simulate Multiple Referenced Model Instances in Normal Mode” on page 8-44
- “Choosing a Simulation Mode” on page 35-10

Simulate Conditionally Executed Referenced Models

You can run a standalone simulation of a conditionally executed referenced model, or conditional model. A standalone simulation is useful for unit testing because it provides consistent data across simulations in terms of data type, dimension, and sample time. Use normal, accelerator, or rapid accelerator mode to simulate a conditional model.

Triggered, Enabled, and Triggered and Enabled Models

Triggered, enabled, and triggered and enabled models require an external input to drive the Trigger or Enable blocks. In the **Signal Attributes** pane of the Trigger or Enable block dialog box, specify values for the signal data type, dimension, and sample time.

To run a standalone simulation, specify the inputs using the **Input** configuration parameter. For details about how to specify the input, see “Comparison of Signal Loading Techniques” on page 70-21. The following conditions apply when you use the **Input** parameter for Trigger and Enable block inputs:

- Use the last data input for the trigger or enable input. For a triggered and enabled model, use the last data input for the trigger input.
- If you do not provide any input values, the simulation uses zero as the default values.

You can log data to determine which signal caused the model to run. For the Trigger or Enable block, in the **Main** pane of the Block Parameters dialog box, select **Show output port**.

Function-Call Models

When you simulate a function-call model, the Model block conditionally executes when it receives a function-call event. A Stateflow chart, Function-Call Generator block, or S-Function block can provide function-call events.

You can also configure the model to calculate output at specific times using a variable-step solver. For more information, see “Samples to Export for Variable-Step Solvers” on page 72-38.

See Also

Blocks
Enable | Function-Call Subsystem | Trigger

More About

- “Modify Referenced Models for Conditional Execution” on page 8-24
- “Choose Simulation Modes for Model Hierarchies” on page 8-39
- “Conditionally Executed Subsystems Overview” on page 10-3
- “Export-Function Models Overview” on page 10-97

Simulate Multiple Referenced Model Instances in Normal Mode

In this section...

Normal Mode Visibility

All instances of a normal-mode referenced model are part of the simulation. However, Simulink software displays only one instance in a model window. Normal mode visibility includes the display of Scope blocks and data port values.

To set normal mode visibility, in the top model, on the **Simulation** tab, in the **Prepare** section, under **Signal Monitoring**, select **Normal Mode Visibility**. This setting determines the instance that Simulink software displays. If you do not specify normal mode visibility for a specific instance of a referenced model, Simulink software selects one instance of the referenced model to display.

After a simulation, if you try to open a referenced model from a Model block that does not have normal mode visibility, Simulink software displays a warning.

To set up your model to control which instance of a referenced model in normal mode has visibility and to ensure proper simulation of the model, see “Specify the Instance Having Normal Mode Visibility” on page 8-46.

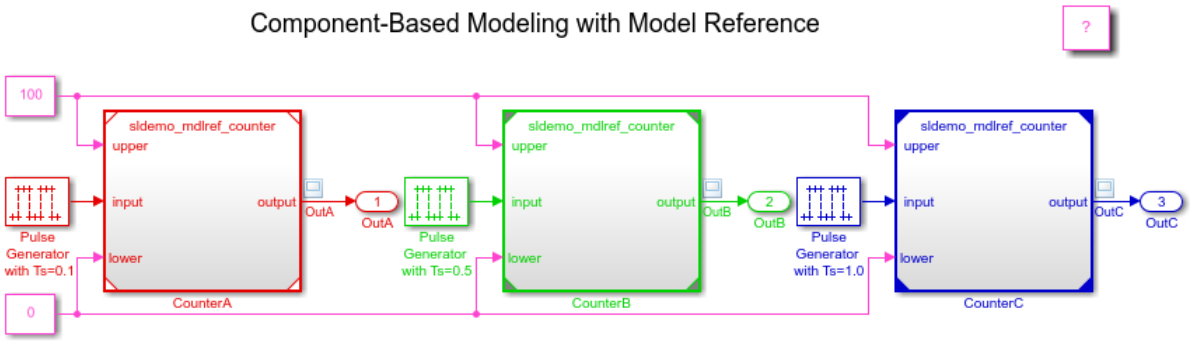
Example Models with Multiple Referenced Model Instances

The `sldemo_mdlref_basic` model and the “Visualizing Model Reference Architectures” featured example show the use of models containing multiple instances of a referenced model.

sldemo_mdlref_basic

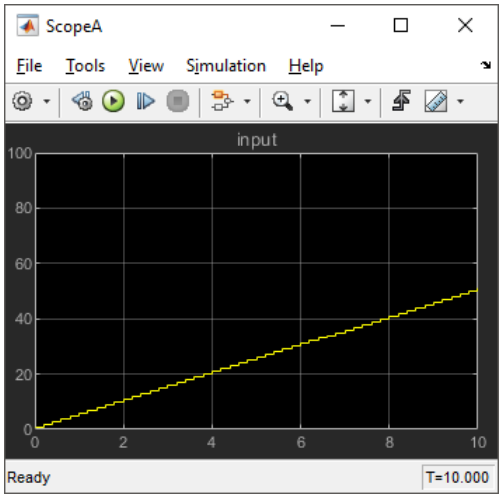
The `sldemo_mdlref_basic` model has three Model blocks (`CounterA`, `CounterB`, and `CounterC`) that each reference the `sldemo_mdlref_counter` model.

If you update the diagram, the `sldemo_mdlref_basic` displays different icons for each of the three Model blocks that reference `sldemo_mdlref_counter`.



Model Block	Icon Corners	Simulation Mode and Normal Mode Visibility Setting
`CounterA`	White	Normal mode, with normal mode visibility enabled
`CounterB`	Gray corners	Normal mode, with normal mode visibility disabled
`CounterC`	Black corner	Accelerator mode (normal mode visibility is not applicable)

Open and simulate `sldemo_mdlref_basic`. Double-click the CounterA model and open the ScopeA block.



That `ScopeA` block reflects the results of simulating the CounterA Model block, which has normal mode visibility enabled.

If you try to open `sldemo_mdlref_counter` model by double-clicking the CounterB Model block, ScopeA in `sldemo_mdlref_counter` still shows the results of the CounterA Model block because that block has normal mode visibility enabled.

Visualizing Model Reference Architectures

The featured example `Visualizing Model Reference Architectures` shows the use of the Dependency Analyzer for a model that references multiple instances of a referenced model in normal mode.

Configure Models with Multiple Referenced Model Instances

1. Set the **Total number of instances allowed per top model** parameter to `Multiple`.
2. Set each instance of the referenced model so that it uses normal mode. In the block parameters dialog box for the Model block that references the instance, set the **Simulation Mode** parameter to `Normal`. Ensure that all the ancestors in the hierarchy for that Model block are in normal mode.

   The corners of icons for Model blocks that are in normal mode can be white (empty). The corners turn gray after you update the diagram or simulate the model.

3 If necessary, modify S-functions used by the model so that they work with multiple instances of referenced models in normal mode. For details, see “Supporting the Use of Multiple Instances of Referenced Models That Are in Normal Mode”.

By default, Simulink assigns normal mode visibility to one of the instances. After you complete the configuration steps, you can specify a non-default instance to have normal mode visibility.

For more information about encapsulating a reusable algorithm in a referenced model, see “Model Reuse” on page 8-6.

Specify the Instance Having Normal Mode Visibility

Determine Which Instance Has Normal Mode Visibility

To determine which instance currently has normal mode visibility:

1 To apply the normal mode visibility setting, update the diagram and make no other changes to the model.

2 Examine the Model blocks that reference the model that you are interested in. The Model block that has white corners has normal mode visibility enabled, navigate through the model hierarchy.

When you are editing a model or during compilation, after updating the diagram, use the `ModelReferenceNormalModeVisibilityBlockPath` parameter. The result is a `Simulink.BlockPath` object that is the block path for the Model block that references the model that has normal mode visibility enabled. For example:

```
get_param('sldemo_mdlref_basic',...
 'ModelReferenceNormalModeVisibilityBlockPath')

ans =

   Simulink.BlockPath
   Package: Simulink

   Block Path:
      'sldemo_mdlref_basic/CounterA'
```

For a top model that you are simulating or that is in a compiled state, you can use the `CompiledModelBlockInstancesBlockPath` parameter. For example:

```
a = get_param('sldemo_mdlref_depgraph',...
 'CompiledModelBlockInstancesBlockPath')

a =

         sldemo_mdlref_F2C: [1x1 Simulink.BlockPath]
      sldemo_mdlref_heater: [1x1 Simulink.BlockPath]
sldemo_mdlref_outdoor_temp: [1x1 Simulink.BlockPath]
```

When you create a `Simulink.BlockPath` object for specifying normal mode visibility:

- The first character vector must represent a block that is in the top model of the model reference hierarchy.
- Character vectors must represent Model blocks that are in normal mode.

- Character vectors that represent variant models or variant subsystems must refer to an active variant.

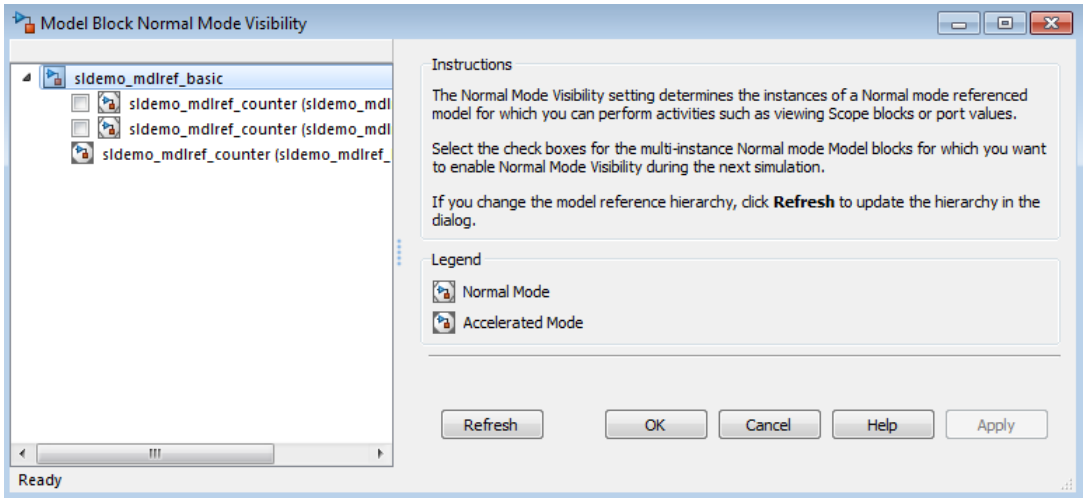
Enable Normal Mode Visibility for an Instance

Note You cannot change normal mode visibility during simulation.

To enable normal mode visibility for a different instance of the referenced model than the instance that currently has normal mode visibility:

1. Navigate to the top model.
2. On the **Simulation** tab, in the **Prepare** section, under **Signal Monitoring**, select **Normal Mode Visibility**.

   The Model Block Normal Mode Visibility dialog box appears. For example, here is the dialog box for the `sldemo_mdlref_basic` model, with the hierarchy pane expanded:

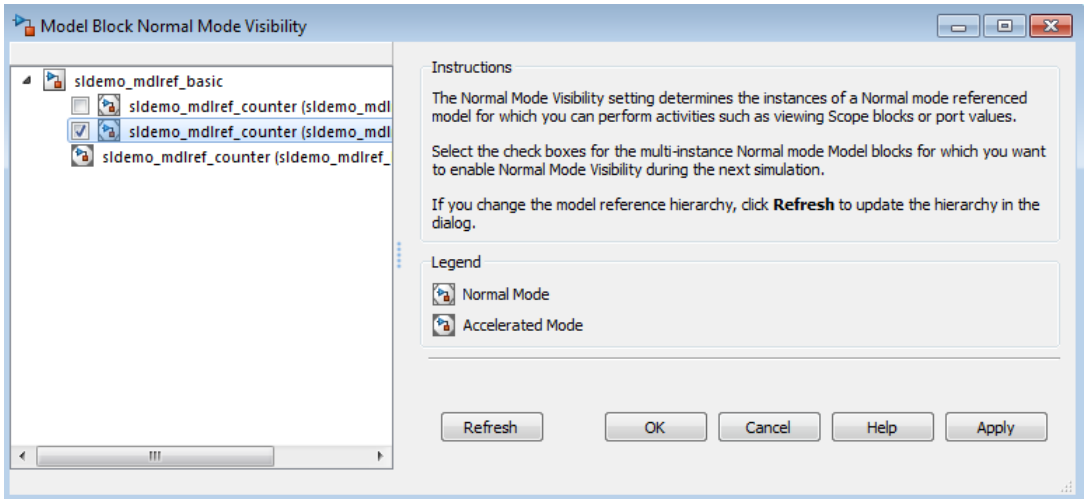


   The model hierarchy pane shows a partial model hierarchy for the model from which you opened the dialog box. The hierarchy stops at the first Model block that is not in normal mode. The model hierarchy pane does not display Model blocks that reference protected models.

   The dialog box shows the complete model hierarchy for the top model. The normal mode instances of referenced models have check boxes.

   Tip To have the model hierarchy pane of the Model Block Normal Mode Visibility dialog box reflect the current model hierarchy, click **Refresh**.

3. Select the instance of the model that you want to have normal mode visibility.



Simulink selects all ancestors of the model and clears all other instances of that model. When a model is cleared, Simulink clears all children of that model.

Tip To open a model from the Model Block Normal Mode Visibility dialog box, right-click the model in the model hierarchy pane and then click **Open**.

4 To apply the normal mode visibility setting, simulate the top model in the model hierarchy.

As an alternative to using the Model Block Normal Mode Visibility dialog box, at the MATLAB command line you can use the `ModelBlockNormalModeVisibility` parameter. For input, you can specify one of these values:

- An array of `Simulink.BlockPath` objects. For example:

```
bp1 = Simulink.BlockPath({'mVisibility_top/Model',  ...
'mVisibility_mid_A/Model'});
bp2 = Simulink.BlockPath({'mVisibility_top/Model1', ...
'mVisibility_mid_B/Model1'});
 bps = [bp1, bp2];
 set_param(topMdl,'ModelBlockNormalModeVisibility',bps);
```

- A cell array of cell arrays of character vectors, with the character vectors being paths to individual blocks and models. This example produces the same effect as the object array example:

```
p1 = {'mVisibility_top/Model','mVisibility_mid_A/Model'};
p2 = {'mVisibility_top/Model1','mVisibility_mid_B/Model1'};
set_param(topMdl,'ModelBlockNormalModeVisibility',{p1, p2});
```

- An empty array, to specify the use of the Simulink default selection of the instance that has normal mode visibility. For example:

```
set_param(topMdl, 'ModelBlockNormalModeVisibility', []);
```

  Using an empty array is equivalent to clearing all the check boxes in the Model Block Normal Mode Visibility dialog box.

See Also

Related Examples

- “Choose Simulation Modes for Model Hierarchies” on page 8-39
- “Choosing a Simulation Mode” on page 35-10
- “Reduce Update Time for Referenced Models by Using Parallel Builds” on page 8-53

More About

- “Manage Simulation Targets for Referenced Models” on page 8-50

Manage Simulation Targets for Referenced Models

A simulation target, or SIM target, is an automatically generated MEX-file that implements a referenced model that executes in accelerator mode. Simulink invokes the simulation target as needed during simulation to compute the behavior and outputs of the referenced model. Simulink uses the same simulation target for all instances of the referenced model that execute in accelerator mode. Instances of the referenced model that execute in normal mode do not use the simulation target.

To create model reference simulation targets, Simulink generates code that imposes some requirements and limitations on referenced models that execute in accelerator mode. Aside from these constraints, you can generally ignore simulation targets when you execute a referenced model in accelerator mode. For information on these constraints, see “Choose Simulation Modes for Model Hierarchies” on page 8-39.

By default, Simulink generates the simulation target for a referenced model that executes in accelerator mode if:

- The simulation target does not exist when you update the diagram of a direct or indirect parent, simulate the model hierarchy, or generate code for the model hierarchy.
- The simulation target is out of date with structural changes in the referenced model.

While generating a simulation target, the MATLAB command window displays status messages so that you can monitor the simulation target generation process.

To programmatically build a model reference simulation target, use the `slbuild` function.

Note If you have a Simulink Coder license, be careful not to confuse the simulation target of a referenced model with these other types of targets:

- Hardware target — A platform for which Simulink Coder generates code
- System target — A file that tells Simulink Coder how to generate code for particular purpose
- Rapid Simulation target (RSim) — A system target file supplied with Simulink Coder
- Model reference target — A library module that contains Simulink Coder code for a referenced model

---

Reduce Time Spent Checking For Changes

You can reduce the time that Simulink spends checking whether simulation targets require rebuilding by setting configuration parameter values as follows:

- In all referenced models throughout the hierarchy, set the **Signal resolution** configuration parameter to `Explicit only` or `None`.
- In the top model of the model hierarchy, set the **Rebuild** configuration parameter to `If any changes in known dependencies detected`. Alternatively, you can use this parameter to specify that Simulink always or never rebuilds simulation targets.

These parameters exist in the configuration set of the model; they are not parameters of the Model block. Setting these parameters for any instance of a referenced model sets it for all instances of that model.

Use Custom Code

To use custom C code with Stateflow or with MATLAB Function blocks when building a model reference simulation target, use the **Include custom code for referenced models** configuration parameter.

Caution Using custom C code for referenced models in accelerator mode can produce different results than when you simulate the model without using custom code. If the custom code includes declarations of structures for buses or enumerations, the simulation target generation fails if the build results in duplicate declarations of those structures. Also, if custom code uses a structure that represents a bus or enumeration, you can get unexpected simulation results.

Control Location of Simulation Targets

Simulink creates simulation targets in the `slprj` build folder of the current folder. If the `slprj` folder does not exist, Simulink creates it.

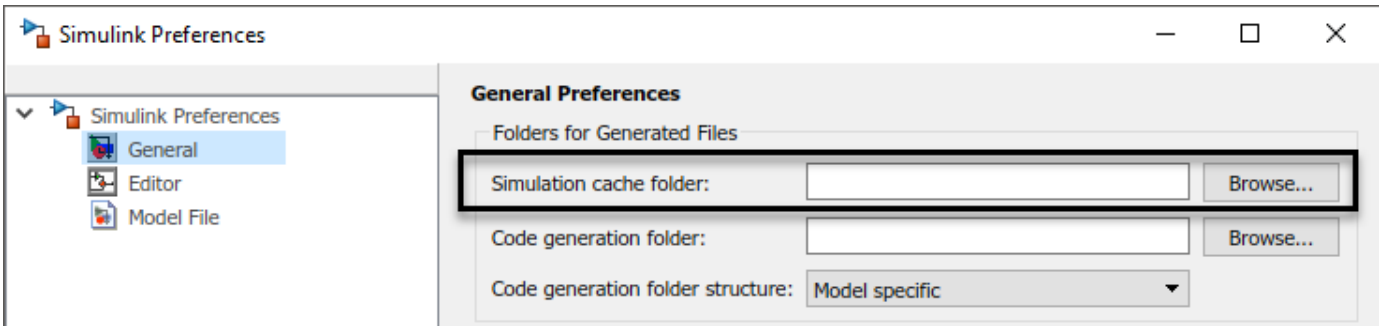
Note Simulink Coder code generation also uses the `slprj` folder. Subfolders in `slprj` provide separate places for simulation code, Simulink Coder code, and other files. For details, see "Manage Build Process Folders" (Simulink Coder).

You can place generated files in a different root folder than the current working folder. This option allows you to:

- Store generated files separate from the models and other source materials used to generate them.
- Reuse or share previously built simulation targets without having to set the current working folder to a previous working folder.
- Separate generated simulation artifacts from generated production code.

The simulation cache folder is the root folder in which to place artifacts used for simulation.

To specify a simulation cache folder to use instead of the current folder (`pwd`), in the **Simulink Preferences > General** dialog box, set the **Simulation cache folder** by entering or browsing to a folder path.



This preference provides the initial defaults for MATLAB session parameters.

Alternatively, you can set the `CacheFolder` MATLAB session parameter using the `set_param` function.

```
>> set_param(0, 'CacheFolder', fullfile('C:','Work','MyModelSimCache'))
>> get_param(0, 'CacheFolder')

ans =

C:\Work\MyModelSimCache
```

To override or restore the **Simulation cache folder** preference only for the current MATLAB session, use the `Simulink.fileGenControl` function. The values you set using `Simulink.fileGenControl` expire at the end of the current MATLAB session.

See Also

More About

- “Choose Simulation Modes for Model Hierarchies” on page 8-39
- “Share Simulink Cache Files for Faster Simulation” on page 8-54
- “Reduce Update Time for Referenced Models by Using Parallel Builds” on page 8-53

Reduce Update Time for Referenced Models by Using Parallel Builds

For models with large model reference hierarchies, you can increase the speed of diagram updates by building in parallel referenced models that are configured to run in accelerator mode. With Parallel Computing Toolbox software, you can distribute the code generation and compilation for referenced models across a parallel pool of MATLAB workers. If you also have MATLAB Parallel Server™ software, you can distribute the code generation and compilation across remote workers in your MATLAB Parallel Server configuration.

To configure parallel building of referenced models:

1. Open the Configuration Parameters dialog box for the top model of the model hierarchy.
2. Select the **Enable parallel model reference builds** check box.
3. For each MATLAB worker, you can set up a MATLAB environment that is consistent with the MATLAB environment of the client. From the **MATLAB worker initialization for builds** drop-down list, select one of these values:

   - `None` –– Simulink does not initialize workers.
   - `Copy base workspace` –– Simulink attempts to copy the base workspace to each MATLAB worker.
   - `Load top model` –– Simulink loads the top model onto each MATLAB worker.

If a parallel pool of MATLAB workers is not running when you update your model, MATLAB automatically opens a parallel pool of workers using the default cluster profile. To change the default behaviour of the worker cluster, you can modify properties of the cluster profile. If you have not touched your parallel preferences, the default profile is `local`. Control parallel behavior with the parallel preferences, including scaling up to a cluster, automatic pool creation, and preferred number of workers. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

For more general information about parallel computing, see “Run MATLAB Functions with Automatic Parallel Support” (Parallel Computing Toolbox).

See Also

More About

- “Manage Simulation Targets for Referenced Models” on page 8-50
- “Share Simulink Cache Files for Faster Simulation” on page 8-54
- “Run Parallel Simulations” on page 26-7
- “Reduce Build Time for Referenced Models by Using Parallel Builds” (Simulink Coder)

Share Simulink Cache Files for Faster Simulation

Simulink cache files contain build artifacts that can speed up simulation and code generation. To generate these build artifacts and automatically package them in Simulink cache files, perform one of these actions:

- Update the diagram for a model hierarchy that contains models referenced in accelerator mode
- Simulate a model hierarchy that contains models referenced in accelerator mode
- Simulate a top model in accelerator or rapid accelerator mode
- Generate code for a model or model hierarchy

The second time that you perform any of these actions, Simulink builds only the out-of-date files as long as the **Rebuild** configuration parameter is set to `If any changes detected` (default) or `If any changes in known dependencies detected`. With fewer files to build, the actions complete faster.

Note While you can avoid all referenced model rebuilds by setting the **Rebuild** configuration parameter to `Never`, using this setting can produce invalid simulation results.

Team members or continuous integration systems can generate Simulink cache files for models that you use. To reduce the time it takes when you build those models for the first time, you can copy the corresponding Simulink cache files to your local folder specified by the **Simulation cache folder** preference. Simulink extracts any Simulink cache file contents that differ from the contents on disk. If Simulink generates or updates the build artifacts on disk, it locally updates the corresponding Simulink cache files.

You can identify a Simulink cache file by its `.slxc` extension. Its file name matches the name of the corresponding model.

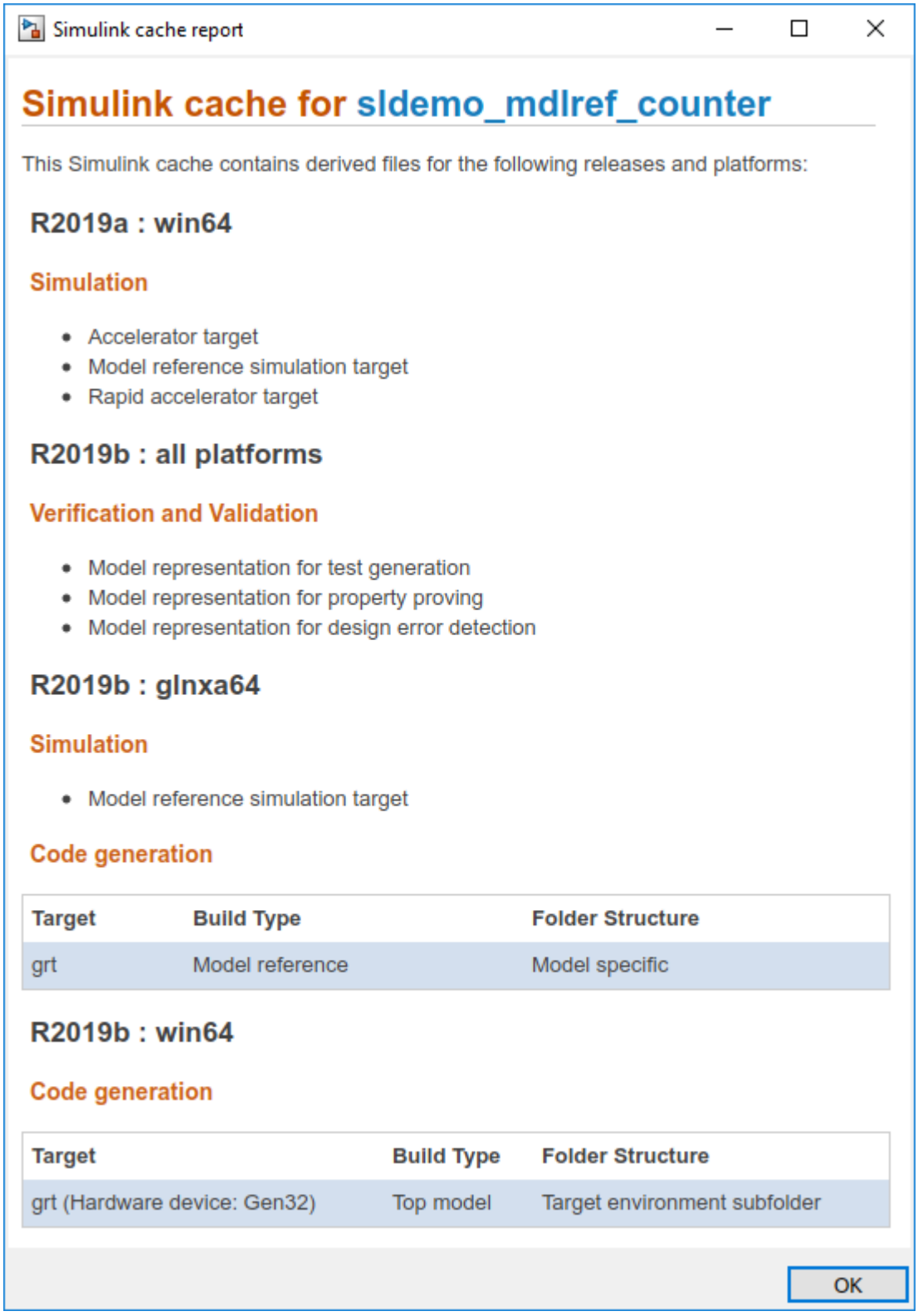
Inspect Simulink Cache File Contents

Simulink cache files can contain multiple types of build artifacts, including:

- Accelerator targets
- Rapid accelerator targets
- Model reference simulation targets
- Variable usage information
- Code generation targets (Simulink Coder, Embedded Coder)
- Model representations (Simulink Design Verifier)

Simulink cache files accumulate build artifacts for multiple platforms and Simulink releases.

To learn what a specific Simulink cache file contains, open the report by double-clicking the file.


Simulink cache report
Simulink cache for sldemo_mdlref_counter
This Simulink cache contains derived files for the following releases and platforms:
R2019a : win64
Simulation
Accelerator target
Model reference simulation target
Rapid accelerator target
R2019b : all platforms
Verification and Validation
Model representation for test generation
Model representation for property proving
Model representation for design error detection
R2019b : glnxa64
Simulation
Model reference simulation target
Code generation
Target
Build Type
Folder Structure
grt
Model reference
Model specific
R2019b : win64
Code generation
Target
Build Type
Folder Structure
grt (Hardware device: Gen32)
Top model
Target environment subfolder
OK

Alternatively, to get the contents of the Simulink cache file in a MATLAB table, use the `slxcinfo` function.

You can reuse build artifacts that support the platform and release that you are using. For example, suppose that you use a Windows machine and the R2019a Simulink release. In the Simulink cache file described by this report, you can use the simulation targets under **R2019a : win64**. A team member that uses a Linux® machine and R2019b can use the simulation and code generation targets under **R2019b : glnxa64** and the model representations under **R2019b : all platforms**.

Note If you create a Simulink cache file in R2019b, you cannot use that Simulink cache file in R2019a. A Simulink cache file can accumulate build artifacts for the release in which it was created and later releases.

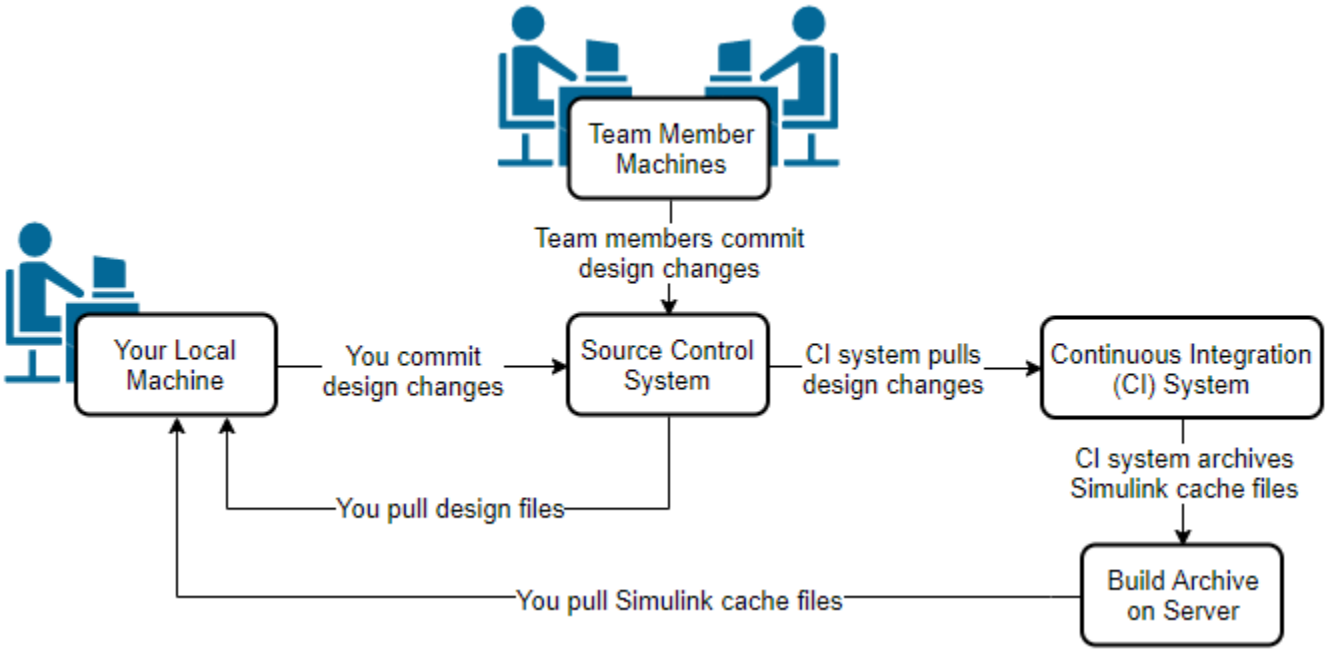
Use Simulink Cache Files

While most teams can benefit from using Simulink cache files, a development environment with these features is best suited for using Simulink cache files.

- A model hierarchy that contains many referenced models.
- A top model that simulates in accelerator or rapid accelerator mode, causing Simulink to generate a simulation target for the top model and all referenced models.
- A standardized platform, compiler, and Simulink release for the entire development team.
- Relative paths or consistent absolute paths, which you specify with the **Include directories** configuration parameter, for the entire development team.
- A source control system, such as Git, that manages design files. See “Set Up Git Source Control” on page 19-16.

  Note Since Simulink cache files are derived files, you should not manage them under source control. Even if you share Simulink cache files by storing them in a source control system, you cannot diff or merge different versions of these files.
- A CI system, such as Jenkins™, which periodically builds the latest version of the model hierarchy using a pool of parallel workers.

In this development environment, you interact with files in the source control system and build archive.



To reduce the amount of time that you spend updating simulation targets, follow these steps.

1. Pull the latest version of all design files from the source control system.
2. Copy the latest version of all Simulink cache files from the build archive and place them in your **Simulation cache folder**.
3. Open the top model and simulate it.

   Simulink extracts the required build artifacts from the Simulink cache files. The simulation completes without rebuilding any models as long as the models have not changed since the most recent build completed by the CI system.

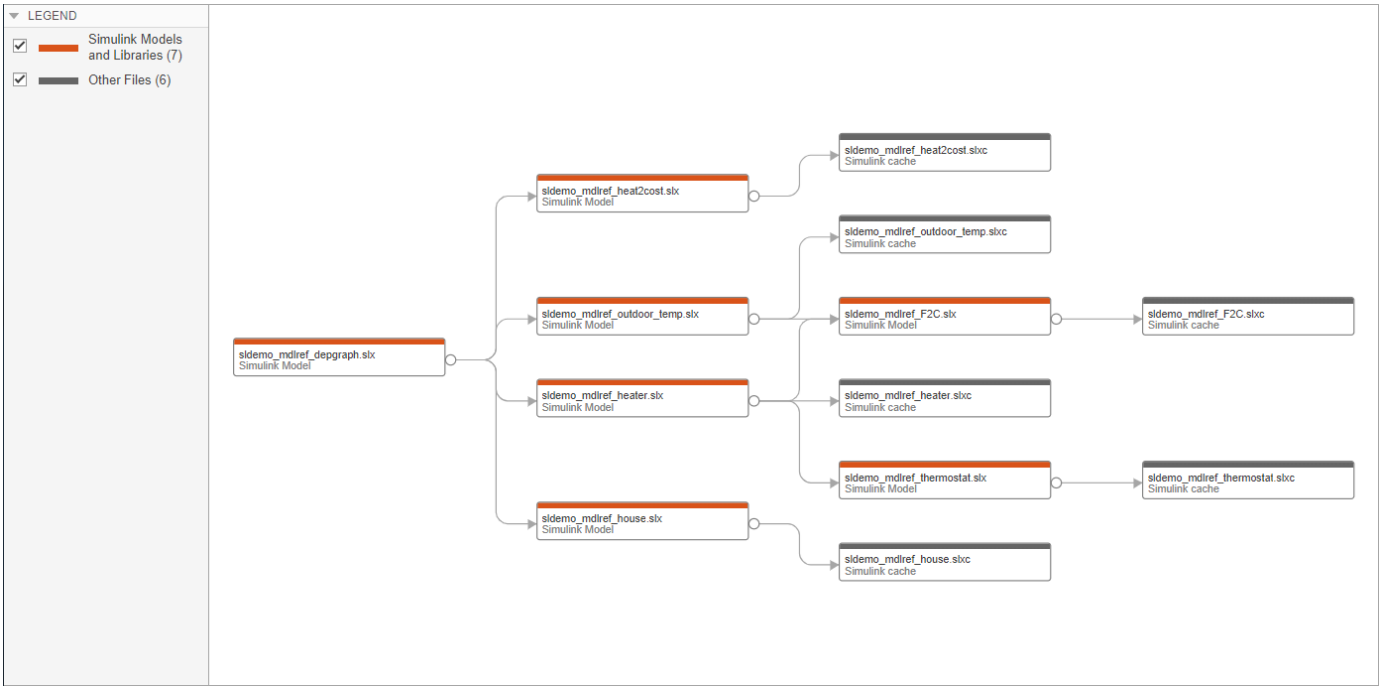
   Note To unpack the simulation and code generation targets from the Simulink cache files without updating, simulating, or generating code for the model hierarchy, use the `slxcunpack` function.
4. Change a model and simulate the model hierarchy again.

   Simulink rebuilds the necessary models and updates the local copy of the corresponding Simulink cache file.
5. Commit the updated model to the source control system. You do not commit the Simulink cache file, which is a derived file.

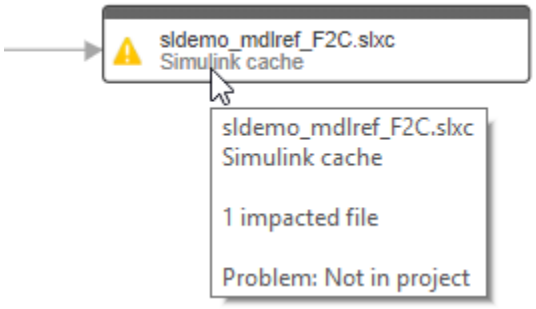
Check for Simulink Cache Files in Projects

When you create a project from a top model, the project includes the corresponding Simulink cache files for the model and its referenced models.

To view Simulink cache file dependencies in a model hierarchy, you can select **Dependency Analyzer** in the **Views** pane of the corresponding project.



If any Simulink cache files are missing from the project, the dependency analysis identifies them.



For more information, see “What Is Dependency Analysis?” on page 18-2

See Also

`slxcinfo` | `slxcunpack`

Related Examples

- “Choose Simulation Modes for Model Hierarchies” on page 8-39
- “Manage Simulation Targets for Referenced Models” on page 8-50
- “Simulink Cache Files for Incremental Code Generation” (Simulink Coder)
- “Reduce Update Time for Referenced Models by Using Parallel Builds” on page 8-53
- “Share Simulink Cache File for Faster Analysis” (Simulink Design Verifier)

External Websites

- Simulink Cache (1 min, 27 sec)
- Agile Model-Based Design: Accelerating Simulink Simulations in Continuous Integration Workflows

Set Configuration Parameters for Model Hierarchies

A referenced model uses a configuration set the same way that it would if the model executed independently. By default, every model in a hierarchy has its own configuration set. When you open a referenced model in the context of a model hierarchy, access its configuration parameters by clicking the **Model Settings** button arrow, then selecting **Model Settings** under **Referenced Model**.

Because each model can have its own configuration set, configuration parameter values can be different in different models. Furthermore, some parameter values are intrinsically incompatible with model references. The Simulink response to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, or a trivial resolution without risk exists, Simulink ignores or resolves the inconsistency without posting a warning.
- Where a nontrivial and possibly acceptable solution exists, Simulink resolves the conflict silently, resolves it with a warning, or generates an error. See “Diagnostics That Are Ignored in Accelerator Mode” on page 8-62 for details.
- Where no acceptable resolution is possible, Simulink generates an error. Change some or all parameter values to eliminate the problem.

Manage Configuration Parameters by Using Configuration References

To assign an externally stored configuration set to multiple models, you can use configuration references. Configuration references help you eliminate configuration parameter incompatibilities.

You can propagate the configuration reference of a top model to an individual referenced model or to all referenced models in the model hierarchy. For an example, see “Share a Configuration Across Referenced Models” on page 13-18.

Configuration Requirements for All Referenced Model Simulation

Some configuration parameter options can cause incompatibilities in model hierarchies. Where possible, Simulink resolves violations of these requirements automatically, but most cases require changes to the parameters in some or all models.

Dialog Box Pane	Option	Requirement
Solver	**Start time**	The compiled start time of the top model and all referenced models must be the same. The compiled start time is the first simulation step after the specified start time. Simulation steps are increments of the fastest discrete rate in the model, beginning from zero.
	Stop time	Simulink uses the **Stop time** of the top model for simulation, overriding any differing **Stop time** in a referenced model.

Dialog Box Pane	Option	Requirement
	Type	The **Type** and **Solver** of the top model apply throughout the hierarchy. See “Solver Settings” on page 8-61.
	Solver	
Data Import/Export	**Initial state**	Can be selected for the top model, but must be cleared for a referenced model.
Math and Data Types	**Application lifespan (days)**	For code generation, the setting must be the same for the parent and referenced models. For simulation, the setting can be different for the parent and referenced models.
Model Referencing	**Total number of instances allowed per top model**	Must not be `Zero` in a referenced model. Specifying `One` rather than `Multiple` is preferable or required sometimes. See “Number of Model Instances Setting” on page 8-62.
Code Generation > Optimization	**Default parameter behavior**	If the parent model has this option set to `Inlined`, then the referenced model cannot be set to `Tunable`.

Solver Settings

Model referencing works with both fixed-step and variable-step solvers. All models in a model hierarchy use the same solver, which is always the solver specified by the top model. An error occurs if the solver type specified by the top model is incompatible with the solver type specified by any referenced model.

Top Model Solver Type	Referenced Model Solver Type	Compatibility
`Fixed-step`	`Fixed-step`	Allowed
`Variable-step`	`Variable-step`	Allowed
`Variable-step`	`Fixed-step`	Allowed unless the referenced model is multirate and specifies both a discrete sample time and a continuous sample time
`Fixed-step`	`Variable-step`	Error

If an incompatibility exists between the top model solver and any referenced model solver, one or both models must change to use compatible solvers. For information about solvers, see “Compare Solvers” on page 3-6 and “Solver Selection Criteria” on page 25-5.

Number of Model Instances Setting

A referenced model must specify that it is available to be referenced, and whether it can be referenced at most once or can have multiple instances. The **Total number of instances allowed per top model** parameter provides this specification. The possible values for this parameter are:

- `Zero` — A model cannot reference this model. An error occurs if a reference to the model occurs in another model.
- `One` — A model hierarchy can reference the model at most once. An error occurs if more than one instance of the model exists. This value is sometimes preferable or required.
- `Multiple` — A model hierarchy can reference the model more than once, if it contains no constructs that preclude multiple references. An error occurs if the model cannot be referenced multiple times, even if only one reference exists.

Setting **Total number of instances allowed per top model** to `Multiple` for a model that is referenced only once can reduce execution efficiency slightly. However, this setting does not affect data values that result from simulation or from executing code Simulink Coder generates. Specifying `Multiple` when only one model instance exists avoids having to change or rebuild the model when reusing the model:

- In the same hierarchy
- Multiple times in a different hierarchy

Some model properties and constructs require setting **Total number of instances allowed per top model** to `One`. For details, see “Model Reuse” on page 8-6.

Diagnostics That Are Ignored in Accelerator Mode

For models referenced in accelerator mode, Simulink ignores the values of these configuration parameter settings if you set them to a value other than `None`:

- **Array bounds exceeded** (`ArrayBoundsChecking`)
- **Inf or NaN block output** (`SignalInfNanChecking`)
- **Simulation range checking** (`SignalRangeChecking`)
- **Division by singular matrix** (`CheckMatrixSingularityMsg`)
- **Wrap on overflow** (`IntegerOverflowMsg`)

Also, for models referenced in accelerator mode, Simulink ignores these **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” on page 73-3.

- **Detect read before write** (`ReadBeforeWriteMsg`)
- **Detect write after read** (`WriteAfterReadMsg`)
- **Detect write after write** (`WriteAfterWriteMsg`)

During model reference simulation in accelerator mode, Simulink temporarily sets several **Configuration Parameters > Diagnostics > Data Validity** parameter settings to `None`, if they are set to `Warning` or `Error`. You can use the Model Advisor to check for parameters that change. For details, see “Diagnostics That Are Ignored in Accelerator Mode” on page 8-62.

You can use the Model Advisor to identify models referenced in accelerator mode for which Simulink ignores the configuration parameters listed above.

1. On the **Modeling** tab, click **Model Advisor**.
2. Select the top model, then click **OK**.
3. Select **By Task > Model Referencing > Check diagnostic settings ignored during accelerated model reference simulation**.
4. Click the **Run This Check** button.

To see the results of running the identified diagnostics with settings to produce warnings or errors, simulate the model in normal mode. Inspect the diagnostic warnings and then simulate in accelerator mode.

Note Configuration parameters on the **Code Generation** pane of the Configuration Parameters dialog box do not affect simulation in either normal or accelerator mode. **Code Generation** parameters affect only code generation by Simulink Coder itself. Accelerator mode simulation requires code generation to create a simulation target. Simulink uses default values for all **Code Generation** parameters when generating the target, and restores the original parameter values after code generation is complete.

See Also

More About

- “Manage Configuration Sets for a Model” on page 13-5
- “Share a Configuration with Multiple Models” on page 13-10
- “Set Configuration Parameters for Code Generation of Model Hierarchies” (Simulink Coder)

Parameterize Instances of a Reusable Referenced Model

When you reference the same model using multiple Model blocks, you can configure a block parameter to use either the same value or a different value for each instance of the model. For example, you can configure the **Gain** parameter of a Gain block. To use different values, create and use a model argument to set the value of the block parameter. For some applications, you can reuse a referenced model only if you can configure each instance to use a different value for a block parameter (such as the setpoint of a controller or a filter coefficient).

Specify a Different Value for Each Instance of a Reusable Model

For a block parameter in a reusable referenced model, to specify a different value for each instance of the model:

1. Create a MATLAB variable or `Simulink.Parameter` object in the model workspace of the referenced model.
   - Use a MATLAB variable for ease of maintenance.
   - Use a `Simulink.Parameter` object for greater control over the minimum and maximum value, the data type, and other properties of the model argument.
2. Set the block parameter value in the model by using the variable or parameter object. Optionally, use the same variable or object to set other block parameter values.
3. Configure the variable or object as a model argument.

   When you simulate this model directly, the block parameters use the value that the variable or object stores in the model workspace. When this model is simulated as a referenced model, a parameter that is configured as a model argument gets its value from its parent model.
4. In each Model block that refers to the reusable model, specify an instance-specific value for the block parameter. If you do not specify a value, the argument uses the last value specified below it in the model hierarchy. In the top model you can configure the diagnostic configuration parameter **No explicit final value for model arguments** to generate an error or warning when the topmost Model block that can set the value for a model argument uses this default value instead of providing an explicit value.
5. In intermediate models, in addition to specifying an instance-specific value for the block parameter, you can specify if the parameter can be overridden at the next level of the hierarchy.

Combine Multiple Arguments into a Structure

When you configure a model to use multiple model arguments, consider using a structure instead of separate variables in the model workspace. This technique reduces the effort of maintenance when you want to add, rename, or delete arguments. Instead of manually synchronizing the arguments in the model workspace with the argument values in Model blocks, you modify structures by using the Variable Editor or the command prompt.

If you have a Simulink Coder license, this technique can also reduce the ROM consumed by the formal parameters of the referenced model functions, such as the output (`step`) function.

To create and use structures to set block parameter values, see “Organize Related Block Parameter Definitions in Structures” on page 37-19.

Parameterize a Referenced Model

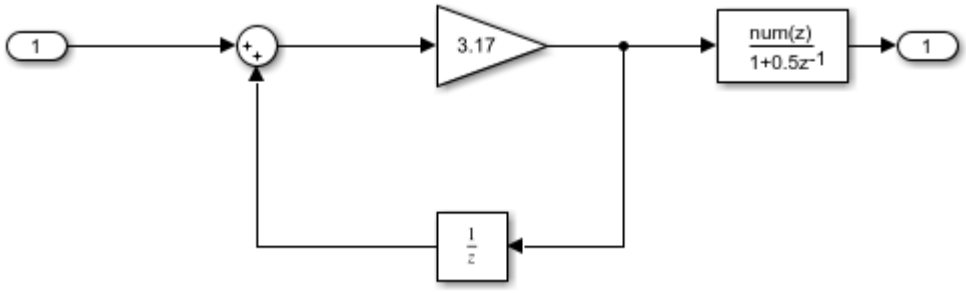
This example shows how to interactively configure multiple instances of a referenced model to use different values for the same block parameter. For an example that parameterizes a referenced model using only the command prompt, see “Parameterize a Referenced Model Programmatically” on page 8-75. For an example that involves code generation, see “Specify Instance-Specific Parameter Values for Reusable Referenced Model” (Simulink Coder).

Configure Referenced Model to Use Model Arguments

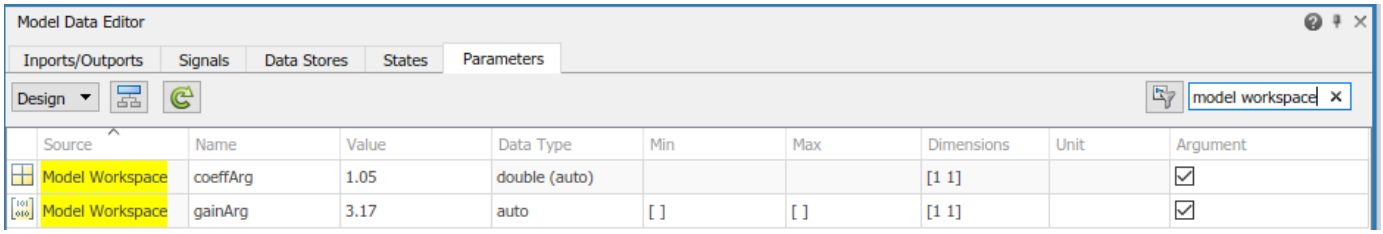
When you simulate a referenced model by itself, the parameter objects in the model workspace use the values that you specify for the `Simulink.Parameter` objects or MATLAB variables. The block parameters also use these values.

To configure the **Gain** parameter of the Gain block and the **Numerator** parameter of the Discrete Filter block as model arguments, follow these steps:

1. Create a model `ex_model_arg_ref` that contains a Gain block and a Discrete Filter block.



2. In the model, on the **Modeling** tab, click **Model Data Editor**.
3. In the Model Data Editor, select the **Parameters** tab.
4. Use the **Value** column to set the value of the **Gain** parameter to a variable, for example, `gainArg`.
5. Next to `gainArg`, click the action button ⋮ and select **Create**.
6. In the Create New Data dialog box, set **Value** to `Simulink.Parameter` and **Location** to `Model Workspace`. Click **Create**.
7. In the `Simulink.Parameter` property dialog box, set **Value** to a number, for example, `3.17`. Click **OK**.
8. Using the Model Data Editor, set the **Numerator** parameter of the Discrete Filter block.
   - Use a `Simulink.Parameter` object named `coeffArg`.
   - Store `coeffArg` in the model workspace.
   - Assign a value of `1.05` to `coeffArg`.
9. In the Model Data Editor, click the **Show/refresh additional information** button.
10. For each object, select the check box in the **Argument** column.

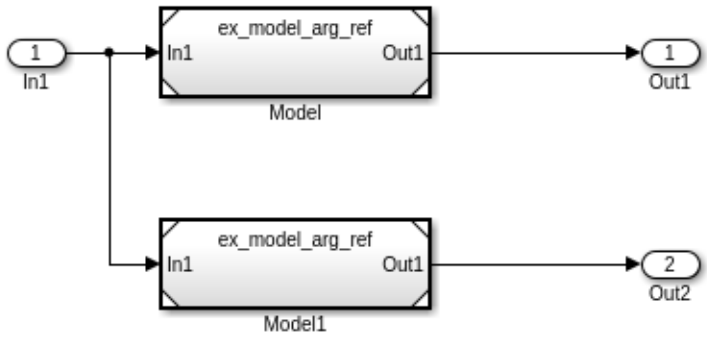


For models with many parameters, you can use the **Filter contents** box to find specific parameters quicker.

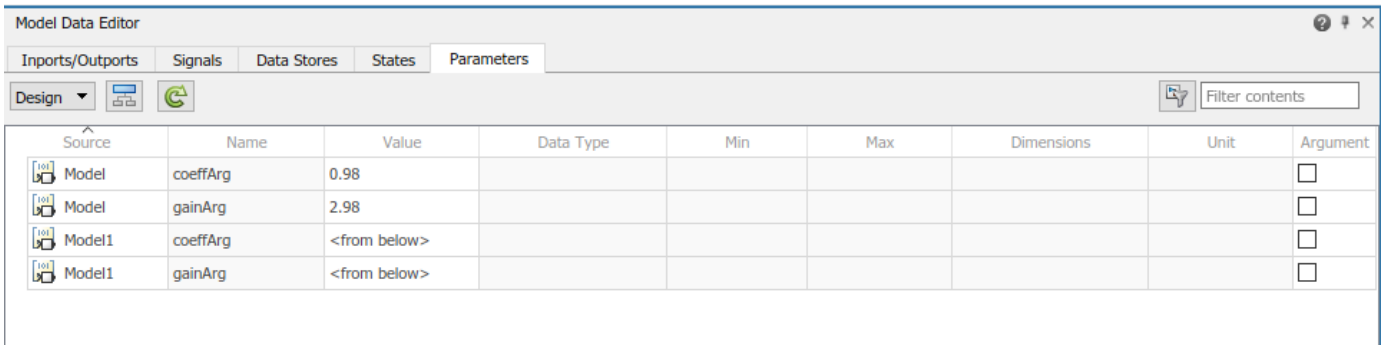
Set Model Argument Values in Parent Model

When you simulate a parent model, each instance of a reusable referenced model uses the parameter values that you specify in the parent model. This example shows how you can expose a model argument as a tunable parameter on the Model block at each level of the model hierarchy.

Create a model `ex_model_arg` that uses multiple instances of the reusable model `ex_model_arg_ref` from the previous example.

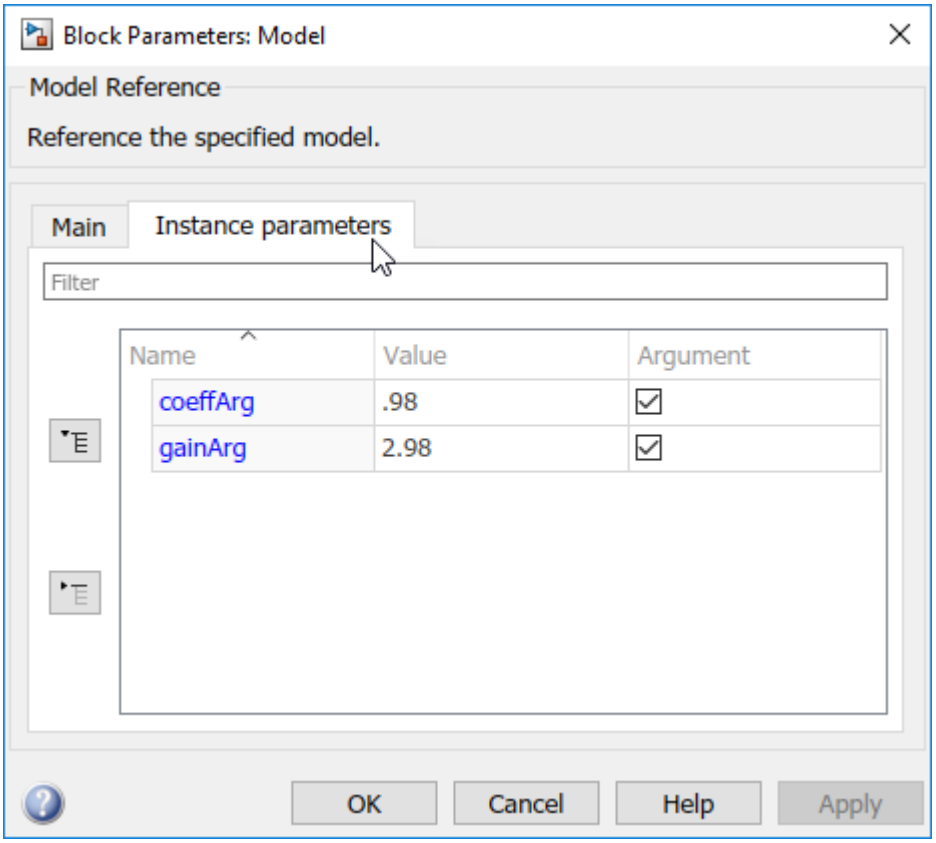


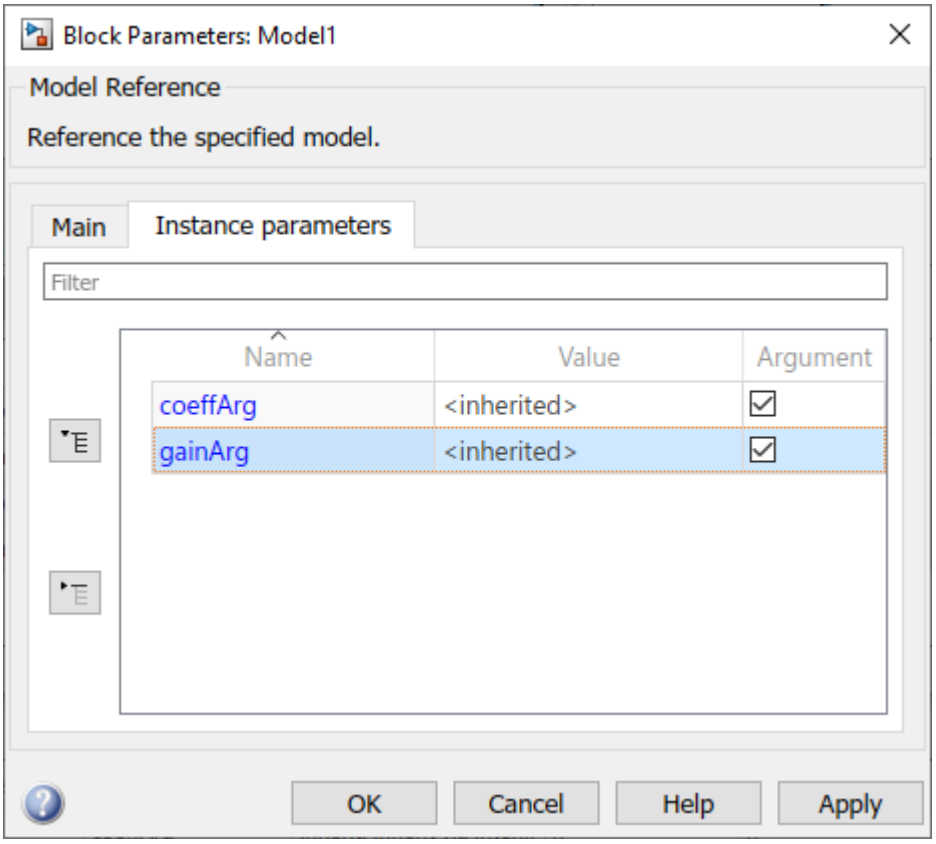
1. In the model, on the **Modeling** tab, click **Model Data Editor**.
2. In the Model Data Editor, select the **Parameters** tab. The Model Data Editor shows four rows that correspond to the instance-specific parameters that you can specify for the two Model blocks.
3. Use the Model Data Editor to set values for the parameters in `Model`. For example, use the values in this figure. For `Model1`, do not specify a value for the model arguments. By default, a model argument uses the last value specified below it in the model hierarchy (indicated by the value `<from below>`).



4 To override the value of these parameters at the next level of the model hierarchy, select the check box in the **Argument** column. By default, the check box is not selected.

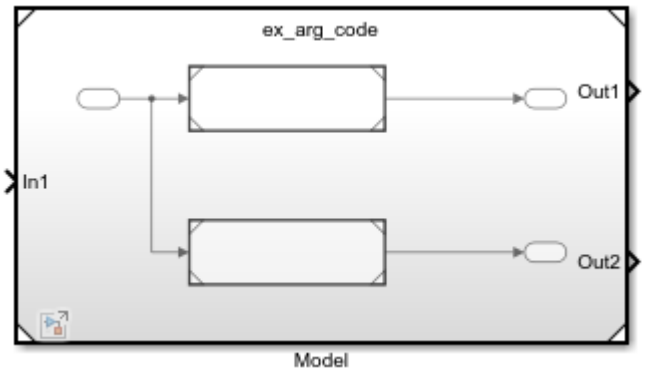
You can also configure instance-specific parameters at each Model block. In the block dialog box, select the **Instance parameters** tab.



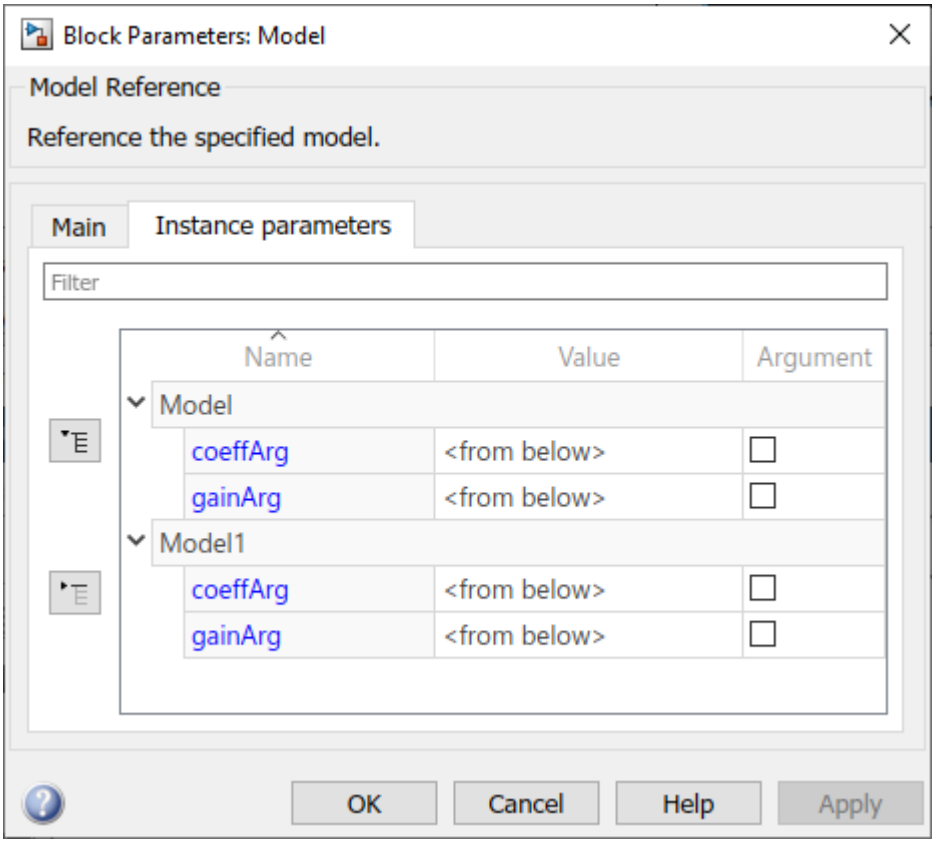


In `Model1`, when you select the **Argument** check box to expose a parameter to the parent model, the **Value** displays as `<inherited>` to indicate that the runtime value now comes from that parent.

5 Create a model `ex_model_arg_top` that contains a Model block that references `ex_model_arg`.



6 Open the block parameter dialog box for the Model block and select the **Instance parameters** tab. In this tab you can see each instance-specific parameter that was exposed as a tunable parameter in the referenced models. From here you can create a parameter value set for all instances of the `coeffArg` and `gainArg` parameters in the model hierarchy.



Group Multiple Model Arguments into Single Structure

You can use structures to reduce the effort of maintenance when you want to add, rename, or delete arguments. With structures, the mathematical functionality of the models is the same.

To replace the parameter values with structures for `ex_model_arg_ref` and `ex_model_arg`, follow these steps:

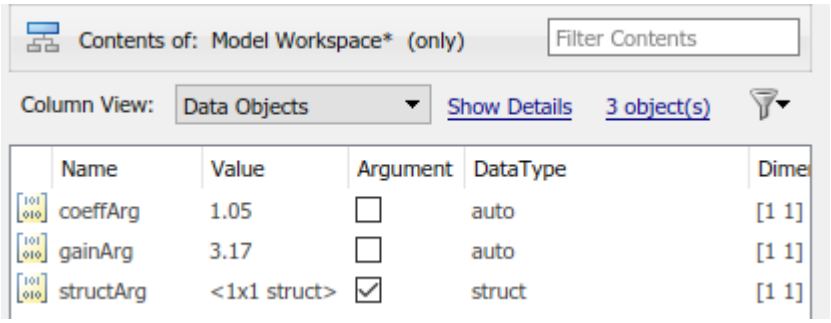
1. At the command prompt, create a structure. Add one field for each of the parameter objects in the `ex_model_arg_ref` workspace.

   ```
   structForInst1.gain = 3.17;
   structForInst1.coeff = 1.05;
   ```

2. Store the structure in a `Simulink.Parameter` object.

   ```
   structForInst1 = Simulink.Parameter(structForInst1);
   ```

3. Open the Model Explorer. In the referenced model, `ex_model_arg_ref`, on the **Modeling** tab, click **Model Explorer**.
4. Use the Model Explorer to copy the parameter object from the base workspace into the `ex_model_arg_ref` model workspace.
5. In the model workspace, rename `structForInst1` as `structArg`.
6. In the **Contents** pane, configure `structArg` as the only model argument.



7 In the `ex_model_arg_ref` model, in the Model Data Editor **Parameters** tab, set the value of the **Gain** parameter to `structArg.gain` and the value of the **Numerator** parameter to `structArg.coeff`.

8 Save the model.

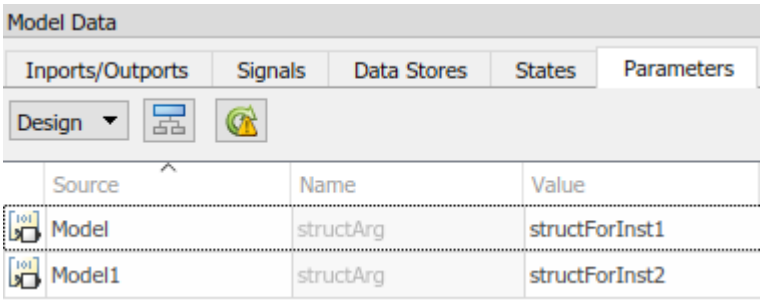
9 At the command prompt, copy the existing structure in the base workspace as `structForInst2`.

```
structForInst2 = copy(structForInst1);
```

10 Set the field values in the two structures by using the same numbers that you used to set the model argument values in the Model blocks.

```
structForInst1.Value.gain = 2.98;
structForInst1.Value.coeff = 0.98;

structForInst2.Value.gain = 3.34;
structForInst2.Value.coeff = 1.11;
```

11 In the top model, `ex_model_arg`, use the Model Data Editor to set the argument values as shown in this figure.



Use Bus Object as Data Type of Structures

You can use a `Simulink.Bus` object as the data type of the structures. The object ensures that the characteristics of the instance-specific structures, such as the names and order of fields, match the characteristics of the structure in the model workspace.

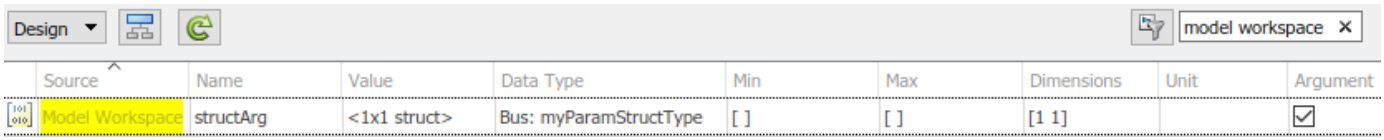
1 At the command prompt, use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object. The hierarchy of elements in the object matches the hierarchy of the structure fields. The default name of the object is `slBus1`.

```
Simulink.Bus.createObject(structForInst1.Value);
```

2 Rename the bus object as `myParamStructType` by copying it.

```
myParamStructType = copy(slBus1);
```

3. In the Model Data Editor for `ex_model_arg`, click the **Show/refresh additional information** button. The Model Data Editor now contains rows that correspond to the parameter objects in the base workspace, `structForInst1` and `structForInst2`.
4. Use the **Data Type** column to set the data type of `structForInst1` and `structForInst2` to `Bus: myParamStructType`.
5. In the Model Data Editor for `ex_model_arg_ref`, use the Model Data Editor to set the data type of `structArg` to `Bus: myParamStructType`.



Change Model Argument Name or Value

To rename a model argument in the context of the referenced model:

- Find all Model blocks that refer to the model and save the instance-specific parameter values that each block specifies. Use the `get_param` function to query the `InstanceParameters` parameter of each block, which is a structure array. The structure contains four fields: `Name`, `Value`, `Path`, and `Argument`.

  You must save the instant-specific parameter values because the renaming operation discards the values in the Model blocks.
- In the Model Data Editor, right-click the variable or object in the model workspace of the referenced model and select **Rename All**. The renaming operation changes the name of the variable or object and changes references to it throughout the model. For more information, see “Create, Edit, and Manage Workspace Variables” on page 67-106.
- Reapply the argument values to the Model blocks by using the new name of the argument. To programmatically set argument values in a Model block, see “Instance parameters”.

Customize User Interface for Reusable Components

When you design a reusable referenced model for use by other members of a team, you can apply a mask to the entire referenced model. You can then customize the way that your users interact with Model blocks, including setting instance-specific values.

Using this technique also makes it easier to programmatically specify instance-specific values. If you create and use a mask parameter named `gainMask` to programmatically set the value to `0.98` for an instance of the model named `myModelBlock`, your users can use this command at the command prompt:

```
set_param('myModelBlock','gainMask','0.98')
```

If you apply a mask to the referenced model, the model mask shows only the instance-specific parameters from the direct child model. It does not show instance-specific parameters promoted up from descendant models.

If you do not mask the model, to set the instance-specific value, use the `InstanceParameters` parameter of the block. For more information, see “Parameterize a Referenced Model Programmatically” on page 8-75.

For information about masking models, see “Introduction to System Mask” on page 39-48.

Configure Instance-Specific Data for Lookup Tables

When you use `Simulink.LookupTable` objects to store and configure lookup table data for ASAP2 or AUTOSAR code generation (for example, STD_AXIS or CURVE), you can configure the objects as model arguments. You can then specify unique table data and breakpoint data for each instance of a component.

You cannot use `Simulink.Breakpoint` objects as model arguments.

You can specify the instance-specific value of a `Simulink.LookupTable` argument as a new `Simulink.LookupTable` in the parent model or as a simple MATLAB structure or array.

When you set **Specification** to `Explicit value` or `Even spacing`, the value can be:

- The name of a valid MATLAB structure variable, such as `Model1_LUT2`
- A literal structure expression, such as `struct('Table', …, 'BP1', …, 'BP2', …)`
- Other expressions that return a valid structure, such as `Params.Model1.LUT2` or a call to a MATLAB function

When you set **Specification** to `Reference`, the value can be:

- A literal numeric array value, such as `[1 5 7; 2 8 13]`
- The name of a numeric array variable, such as `Model1_LUT2`
- Other expressions that return a valid numeric array, such as `Params.Model1.LUT2` or a call to a MATLAB function

When you specify the instance-specific value of a `Simulink.LookupTable` argument as a structure, the following rules apply:

- Each field of the model argument definition must be specified in the structure and the number of fields and the names of the fields must match.
- The dimensions of the table and the breakpoint data in the structure must match that of the model argument definition.
- If the data type of a structure field is `double`, the value is cast to the data type of the corresponding model argument field. Otherwise, the value must match the data type of the corresponding model argument field.

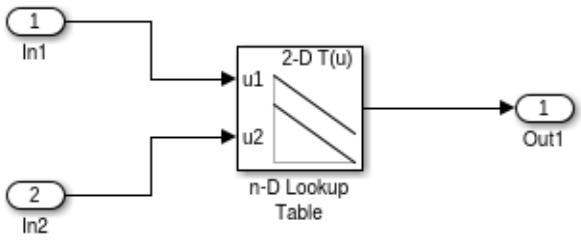
You can specify the value as a simple numeric value for any simulation mode and for code generation. For code generation, if you configure the model argument with a storage class of `Auto`, the structure or numeric array variable is not preserved in the generated code. If you set the storage class to any other value, the structure or numeric array is similar to other model arguments in that the value is used to initialize the tunable argument in the generated code.

This example shows how to specify the instance-specific value of a `Simulink.LookupTable` argument as a new `Simulink.LookupTable` and as a MATLAB structure.

For an example that parameterizes a referenced model by using lookup tables and the command prompt, see “Configure Instance-Specific Data for Lookup Tables Programmatically” on page 8-80.

Configure Model Arguments in a Referenced Model

1 Create a model `ex_arg_LUT_ref`, which represents a reusable algorithm.



2 Using the Model Explorer, add a `Simulink.LookupTable` object in the model workspace. You can use the **Add Simulink LookupTable** button . Name the object `LUTArg`.

3 Set **Number of table dimensions** to 2. In the **Table** and **Breakpoints** tabular area, use specify values for the `Table`, `BP1`, and `BP2` data. For example, configure the table and breakpoint data by entering these values in the MATLAB expression box.

- `Table` — `[3 4;1 2]`
- `BP1` — `[1 2]`
- `BP2` — `[3 4]`

When you simulate or generate code directly from `ex_arg_LUT_ref`, the model uses these values.

4 Under **Struct Type definition**, set **Name** to `LUTArg_Type`.

5 Click **Apply**.

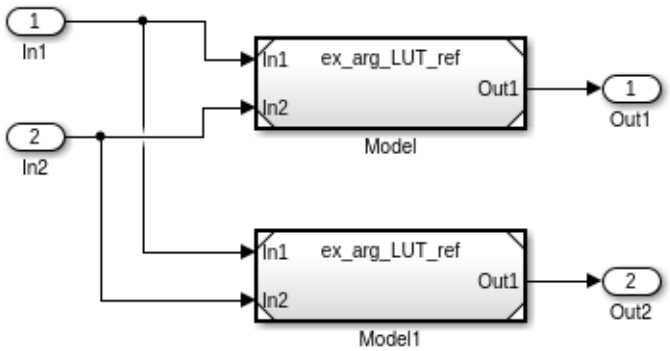
6 In the **Contents** pane, for `LUTArg`, select the check box in the **Argument** column.

7 In the referenced model, in the n-D Lookup Table block, set **Data specification** to `Lookup table object`. Set **Name** to `LUTArg`.

8 Save the model.

Create Instance-Specific Argument Values

1 Create a model `ex_arg_LUT`, which uses the reusable algorithm twice.



2 At the command prompt, create a `Simulink.LookupTable` object in the base workspace. Alternatively, you can create the `Simulink.LookupTable` object in a data dictionary.

```
LUTForInst1 = Simulink.LookupTable;
```

3 Specify breakpoint and table data for the object.

```
LUTForInst1.Table.Value = [8 7; 6 5];
LUTForInst1.Breakpoints(1).Value = [5 6];
LUTForInst1.Breakpoints(2).Value = [3 4];
```

4 Specify a structure type name. Match this name to the name specified by the object in the referenced model workspace.

```
LUTForInst1.StructTypeInfo.Name = 'LUTArg_Type';
```

5 Use a structure to create the instance-specific argument value for the second Model block. Specify the breakpoint and table data for the structure.

```
StructForInst2.Table = [9 8; 7 7];
StructForInst2.BP1 = [3 4];
StructForInst2.BP2 = [5 6];
```

6 In the `ex_arg_LUT` model, for model instance `Model`, on the **Instance parameters** tab, set the value of **LUTArg** to `LUTForInst1`.

7 For model instance `Model1`, set **LUTArg** to `StructForInst2`.

One instance of `ex_arg_LUT_ref` uses the table and breakpoint data stored in the `Simulink.LookupTable` object in the base workspace and the other instance uses the table and breakpoint data stored in the structure.

See Also

`Simulink.Breakpoint` | `Simulink.LookupTable` | `Simulink.Parameter`

Related Examples

- “Parameterize a Referenced Model Programmatically” on page 8-75
- “Configure Instance-Specific Data for Lookup Tables Programmatically” on page 8-80
- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” (Simulink Coder)

More About

- “Organize Related Block Parameter Definitions in Structures” on page 37-19
- “Parameter Interfaces for Reusable Components” on page 37-17
- “Tune and Experiment with Block Parameter Values” on page 37-31

Parameterize a Referenced Model Programmatically

This example shows how to programmatically configure multiple instances of a referenced model to use different values for the same block parameter.

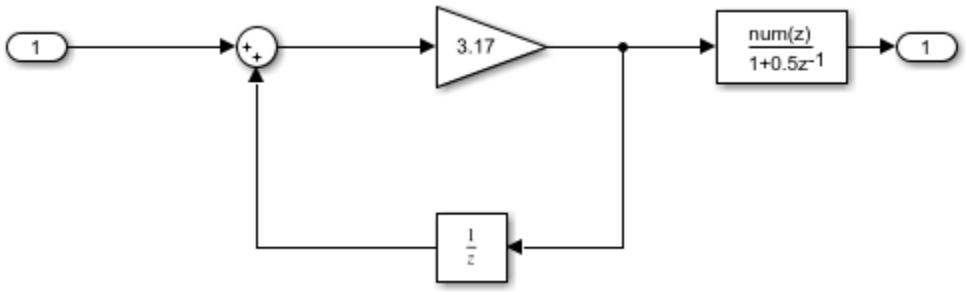
Configure Referenced Model to Use Model Arguments

When you simulate a model, the parameter objects in the model workspace use the values that you specify for the `Simulink.Parameter` objects, `Simulink.LookupTable` objects, or MATLAB® variables. The block parameters also use these values.

To configure the **Gain** parameter of a Gain block and the **Numerator** parameter of a Discrete Filter block as model arguments, follow these steps.

Open model `ex_model_arg_ref`. This model represents a reusable algorithm.

```
open_system('ex_model_arg_ref')
```



For the Gain block, set the value of the **Gain** parameter to a `Simulink.Parameter` object in the model workspace with a numeric value. For this example, name the `Simulink.Parameter` object `gainArg` and assign a value of `3.17`.

```
set_param('ex_model_arg_ref/Gain','Gain','gainArg')
modelWorkspace = get_param('ex_model_arg_ref','ModelWorkspace');
assignin(modelWorkspace,'gainArg',Simulink.Parameter(3.17));
```

For the Discrete Filter block, set the value of the **Numerator** parameter to a `Simulink.Parameter` object in the model workspace with a numeric value. For this example, name the `Simulink.Parameter` object `coeffArg` and assign a value of `1.05`.

```
set_param('ex_model_arg_ref/Discrete Filter','Numerator','coeffArg')
assignin(modelWorkspace,'coeffArg',Simulink.Parameter(1.05));
```

Specify `gainArg` and `coeffArg` as model arguments.

```
set_param('ex_model_arg_ref','ParameterArgumentNames','coeffArg,gainArg')
```

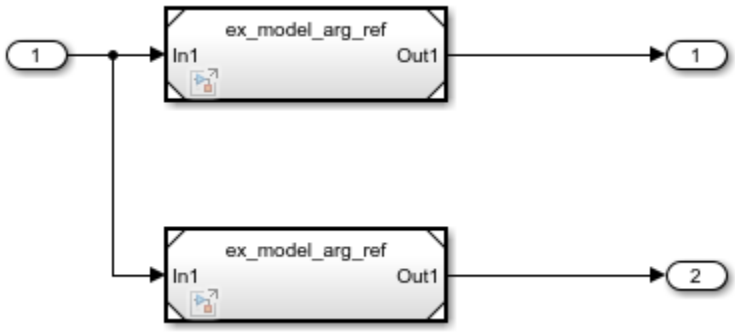
Set Model Argument Values in Parent Model

When you simulate a parent model, each instance of a reusable referenced model uses the argument values that you specify in the parent model. In this example, in the upper instance of `ex_model_arg_ref`, the parameter object `gainArg` uses the value `2.98`.

Model `ex_model_arg` contains two Model blocks that reference `ex_model_arg_ref`. To set different parameter values for the two instances of the model, follow these steps.

Open model `ex_model_arg`. This model represents a system model that uses multiple instances of the reusable algorithm.

```
open_system('ex_model_arg')
```



For both instances of model `ex_model_arg`, set values for the model arguments. If you decide to re-promote these arguments, set the `Argument` field to `true`. By default, the `Argument` field is `false`.

```
instSpecParams = get_param('ex_model_arg/Model','InstanceParameters');
instSpecParams1 = get_param('ex_model_arg/Model1','InstanceParameters');

instSpecParams(1).Value = '.98';
instSpecParams(2).Value = '2.98';
instSpecParams1(1).Value = '1.11';
instSpecParams1(2).Value = '3.34';
instSpecParams(1).Argument = true;
instSpecParams(2).Argument = true;
instSpecParams1(1).Argument = true;
instSpecParams1(2).Argument = true;

set_param('ex_model_arg/Model','InstanceParameters',instSpecParams);
set_param('ex_model_arg/Model1','InstanceParameters',instSpecParams1);
```

See Also

More About

- “Group Multiple Model Arguments into a Single Structure” on page 8-77

Group Multiple Model Arguments into a Single Structure

This example shows how to programmatically configure multiple instances of a referenced model to use different values for the same block parameter by using structures.

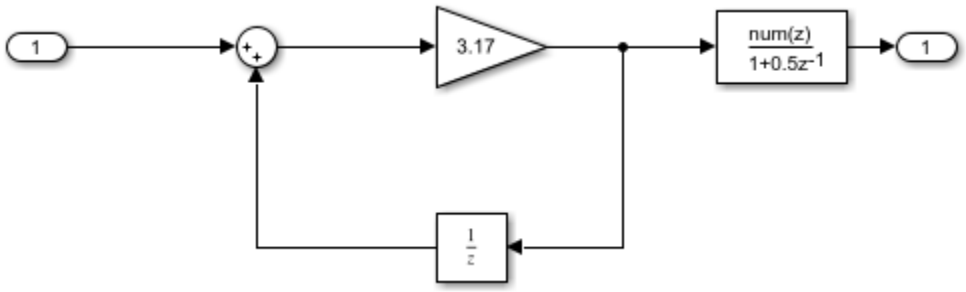
Configure Referenced Model to Use Model Arguments Grouped Into Structure

You can use structures to reduce the effort of maintenance when you want to add, rename, or delete arguments. With structures, the mathematical functionality of the models is the same.

To replace the parameter values with structures for `ex_model_arg_ref` and `ex_model_arg`, follow these steps.

Open model `ex_model_arg_ref`. This model represents a reusable algorithm.

```
open_system('ex_model_arg_ref')
```



Create a structure that contains one field for each of the parameter objects that exist in the `ex_model_arg_ref` workspace. Specify a value for each field.

```
structForInst1.gain = 3.17;
structForInst1.coeff = 1.05;
```

Store the structure in a `Simulink.Parameter` object.

```
structForInst1Param = Simulink.Parameter(structForInst1);
```

Copy the `Simulink.Parameter` object into the `ex_model_arg_ref` model workspace. For this example, name the copy of the object `structArg`.

```
modelWorkspace = get_param('ex_model_arg_ref','ModelWorkspace');
assignin(modelWorkspace,'structArg',copy(structForInst1Param));
```

Configure `structArg` as the only model argument.

```
set_param('ex_model_arg_ref','ParameterArgumentNames','structArg')
```

In the `ex_model_arg_ref` model, set the **Gain** parameter of the Gain block to `structArg.gain` and set the **Numerator** parameter of the Discrete Filter block to `structArg.coeff`.

```
set_param('ex_model_arg_ref/Gain','Gain','structArg.gain')
set_param('ex_model_arg_ref/Discrete Filter',...
    'Numerator','structArg.coeff')
```

Copy the existing structure as `structForInst2Param`.

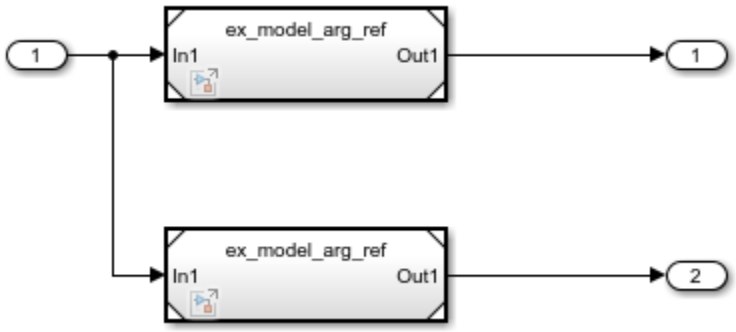
```
structForInst2Param = copy(structForInst1Param);
```

Set the field values in the two structures to the same numbers that you used to set the model argument values in the Model blocks.

```
structForInst1Param.Value.coeff = 0.98;
structForInst1Param.Value.gain = 2.98;
structForInst2Param.Value.coeff = 1.11;
structForInst2Param.Value.gain = 3.34;
```

Open model `ex_model_arg`. This model represents a system model that uses multiple instances of the reusable algorithm.

```
open_system('ex_model_arg')
```



For model instance `Model`, set **structArg** to `structForInst1Param`. For model instance `Model1`, set **structArg** to `structForInst2Param`.

```
instSpecParamsStruct = get_param('ex_model_arg/Model','InstanceParameters');
instSpecParamsStruct1 = get_param('ex_model_arg/Model1','InstanceParameters');

instSpecParamsStruct(1).Value = 'structForInst1Param';
instSpecParamsStruct1(1).Value = 'structForInst2Param';

set_param('ex_model_arg/Model','InstanceParameters',instSpecParamsStruct);
set_param('ex_model_arg/Model1','InstanceParameters',instSpecParamsStruct1);
```

Use Bus Object as Data Type of Structures

You can use a `Simulink.Bus` object as the data type of the structures. The bus object makes sure that the characteristics of the instance-specific structures, such as the names and order of fields, match the characteristics of the structure in the model workspace.

To set the data type of the structures to bus objects, follow these steps.

Use the `Simulink.Bus.createObject` function to create the bus object. The hierarchy of elements in the object matches the hierarchy of the structure fields. The default name of the object is `slBus1`.

```
Simulink.Bus.createObject(structForInst1Param.Value);
```

Rename the bus object by copying it.

```
myParamStructType = copy(slBus1);
```

Set the data type of the parameter objects in the base workspace by using the bus object.

```
structForInst1Param.DataType = 'Bus: myParamStructType';
structForInst2Param.DataType = 'Bus: myParamStructType';
```

For the `structArg` object, set `DataType` to `Bus: myParamStructType`.

```
temp = getVariable(modelWorkspace,'structArg');
temp = copy(temp);
temp.DataType = 'Bus: myParamStructType';
assignin(modelWorkspace,'structArg',copy(temp));
close_system('ex_model_arg_ref',0)
close_system('ex_model_arg',0)
```

See Also

More About

- “Parameterize a Referenced Model Programmatically” on page 8-75

Configure Instance-Specific Data for Lookup Tables Programmatically

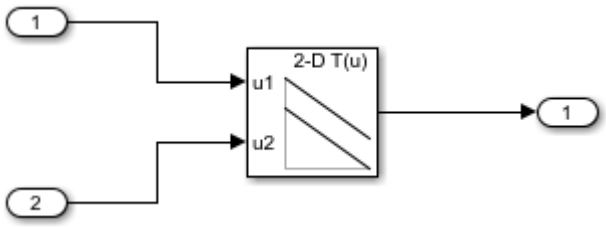
When you use `Simulink.LookupTable` objects to store and configure lookup table data for ASAP2 or AUTOSAR code generation (for example, STD_AXIS or CURVE), you can configure the objects as model arguments. You can then specify unique table data and breakpoint data for each instance of a component.

This example shows how to configure multiple instances of a referenced model to use different values for the same block parameter by using lookup tables and the command prompt.

Configure Model Arguments in Referenced Model

Open model `ex_arg_LUT_ref`, which represents a reusable algorithm.

```
open_system('ex_arg_LUT_ref')
```



Create a `Simulink.LookupTable` object in the base workspace. For this example, name the object `LUTArg`.

```
temp = Simulink.LookupTable;
```

Specify values for the table and breakpoint data. When you simulate or generate code directly from `ex_arg_LUT_ref`, the model uses these values.

```
temp.Table.Value = [3 4; 1 2];
temp.Breakpoints(1).Value = [1 2];
temp.Breakpoints(2).Value = [3 4];
```

Set the structure name to `LUTArg_Type`.

```
temp.StructTypeInfo.Name = 'LUTArg_Type';
```

Copy the structure to the model workspace.

```
mdlwks = get_param('ex_arg_LUT_ref','ModelWorkspace');
assignin(mdlwks,'LUTArg',copy(temp))
```

Specify `LUTArg` as a model argument.

```
set_param('ex_arg_LUT_ref','ParameterArgumentNames','LUTArg')
```

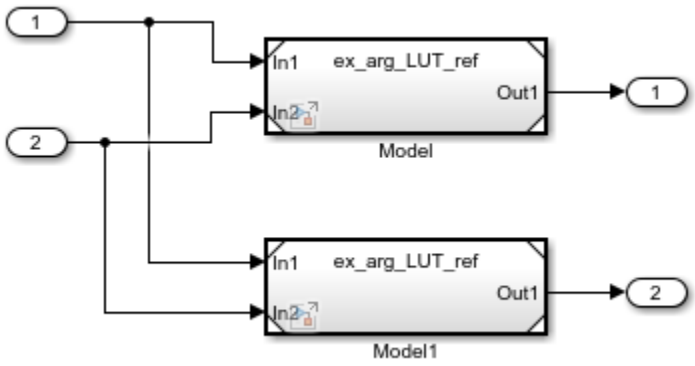
For the n-D Lookup Table block, set `'Data specification'` to `'Lookup table object'` and set the name to `LUTArg`.

```
set_param('ex_arg_LUT_ref/n-D Lookup Table',...
    'DataSpecification','Lookup table object','LookupTableObject','LUTArg')
```

Create Instance-Specific Argument Values

Open model `ex_arg_LUT`, which uses the reusable algorithm twice.

```
open_system('ex_arg_LUT')
```



Create a `Simulink.LookupTable` object in the base workspace.

```
LUTForInst1 = Simulink.LookupTable;
```

Specify table and breakpoint data for the object.

```
LUTForInst1.Table.Value = [8 7; 6 5];
LUTForInst1.Breakpoints(1).Value = [5 6];
LUTForInst1.Breakpoints(2).Value = [3 4];
```

Specify the structure name to match the name specified by the object in the referenced model workspace.

```
LUTForInst1.StructTypeInfo.Name = 'LUTArg_Type';
```

Use a structure to create the instance-specific argument value for the second Model block. Specify the breakpoint and table data for the structure.

```
StructForInst2.Table = [9 8; 7 7];
StructForInst2.BP1 = [3 4];
StructForInst2.BP2 = [5 6];
```

In the `ex_arg_LUT` model, for model instance `Model`, set the value of `LUTArg` to `LUTForInst1`. For model instance `Model1`, set the value of `LUTArg` to `StructForInst2`.

```
set_param('ex_arg_LUT/Model','ParameterArgumentValues',...
    struct('LUTArg','LUTForInst1'))
set_param('ex_arg_LUT/Model1','ParameterArgumentValues',...
    struct('LUTArg','StructForInst2'))
```

One instance of `ex_arg_LUT_ref` uses the table and breakpoint data stored in the `Simulink.LookupTable` object in the base workspace and the other instance uses the table and breakpoint data stored in the structure.

Simulink Units

Unit Specification in Simulink Models

Simulink enables you to specify physical units as attributes on signals at the boundaries of model components. Such components can be:

- Subsystems
- Referenced Simulink models
- Simulink-PS Converter and PS-Simulink Converter blocks that interface between Simulink and components developed in Simscape and its associated physical modeling products
- Stateflow charts, state transition tables, or truth tables
- MATLAB Function blocks
- Constant blocks
- Data Store Memory, Data Store Read, and Data Store Write blocks

By specifying, controlling, and visualizing signal units, you can ensure the consistency of calculations across the various components of your model. For example, this added degree of consistency checking is useful if you are integrating many separately developed components into a large, overall system model.

In Simulink models, you specify units from a unit database. The unit database comprises units from the following unit systems:

- `SI` — International System of Units
- `SI (extended)` — International System of Units (extended)
- `English` — English System of Units
- `CGS` — Centimetre-gram-second System of Units

Based on the type of system you are modeling, you can use any combination of units from these supported unit systems. For more information about supported unit systems and the units they contain, see Allowed Units.

You can assign units to signals through these blocks:

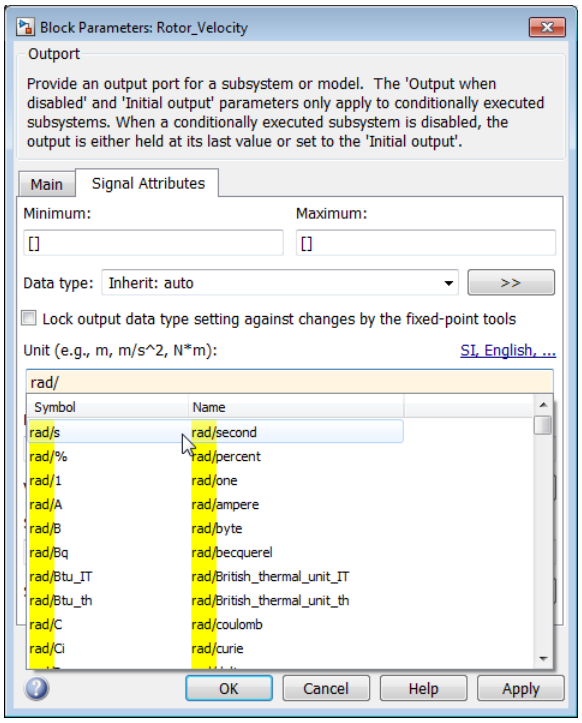
- Inport
- Outport
- Signal Specification
- MATLAB Function
- Stateflow Chart

and these objects:

- `Simulink.Signal`
- `Simulink.BusElement`
- `Simulink.Parameter`

When you add a supported block to your model, the **Unit** parameter on the block is set to `inherit` by default. This setting means that the block inherits the unit from a connecting signal that has an explicitly specified unit.

You can explicitly specify units for signals using the **Unit** parameter of a supported block. For this parameter, the dialog box provides matching suggestions to help you:



If you do not provide a correctly formed unit expression, you get an error. Correctly formed unit expressions are a combination of unit names or symbols with properly balanced parentheses and `*`, `/`, and `^` characters. Special characters such as `[`, `]`, `{`, `}`, `<`, `>`, `\`, `"`, `&`, and so forth are not supported.

By default, a block port has an empty (that is, unspecified) unit and the **Unit** parameter is set to `inherit`. When you specify a unit for one port, Simulink checks the unit setting of any port connected to it. If a port has an empty unit, you can connect it to another port that has any supported unit. If a port unit parameter is set to `inherit`, it inherits the unit from a connected port that has a specified unit.

Specify Physical Quantities

When you model a physical system, it is possible to use the same unit expression for two or more signals that represent different physical quantities. For example, a unit expression of `N*m` can represent either torque or energy. To prevent mistaken connection of two ports with the same unit but representing different physical quantities, you can add a physical quantity to the unit expression. For example, for the same unit of `N*m`, you can specify different physical quantities of `N*m@torque` and `N*m@energy`. Similar to units, the dialog box provides suggestions as you type the names of physical quantities.

Physical quantities help you to enforce an extra degree of unit consistency checking between connected ports. When you attempt to connect ports with different physical quantities, the model displays a warning.

Specify Units in Objects

By default, `Simulink.Signal`, `Simulink.BusElement`, and `Simulink.Parameter` objects have empty units. In the case of a:

- `Simulink.Signal` object, the empty unit means that the corresponding signal can inherit a unit from an upstream or downstream port.
- `Simulink.BusElement` object, the empty unit means that the corresponding bus element signal also has an empty unit. You can connect the signal to a port with any unit, but the signal does not inherit a unit from the port.
- `Simulink.Parameter` object, the object does not attach a unit to the corresponding parameter value.

If you specify a unit in a `Simulink.Signal` or `Simulink.BusElement` object, Simulink applies the attribute to the corresponding signal line when:

- The `Simulink.Signal` object resolves to a signal in the model
- You use a bus element signal that is associated with a `Simulink.Bus` object with a Bus Creator, Bus Selector, or Bus Assignment block.

For the `Simulink.Parameter` object, Simulink does not apply any attribute. For all objects, if the **Unit** parameter has a value that is not formed correctly, you see an error. If the unit is formed correctly but is undefined, you see a warning when you compile the model. If the unit expression contains special characters such as `[`, `]`, `{`, `}`, `<`, `>`, `\`, `"`, `&`, and so forth, Simulink replaces them with underscores (`_`).

Custom Unit Properties

Notes on the `Unit` and `DocUnits` properties starting in R2016a:

- The `DocUnits` property is now `Unit` for `Simulink.Parameter` or `Simulink.Signal` objects. If, in a previous release, you used the `DocUnits` parameter of a `Simulink.Parameter` or `Simulink.Signal` object to contain text that does not now comply with units specifications, simulation returns a warning when the model simulates.

  To suppress these warnings, set the configuration parameter “Units inconsistency messages” to `none`. This setting suppresses all units inconsistency check warnings.
- If you have a class that derives from `Simulink.Parameter`, `Simulink.Signal`, or `Simulink.BusElement` with a previously defined `Unit` property, Simulink returns an error like the following:

  ```
  Cannot define property 'Unit' in class 'classname' because
   the property has already been defined in the superclass 'superclass'.
  ```

  If you use this property to represent the physical unit of the signal, delete the `Unit` property from the derived class in the R2016a or later release. Existing scripts continue to work, unless you are assigning incorrectly formed unit expressions to the `Unit` field. In this case, replace the use of `Unit` with `DocUnits` to continue to be able to assign the unit expression.

Note If you store existing data in a MAT- or `.sldd` file, in a release prior to R2016a, copy the contents of the `Unit` property to the `DocUnits` first. Then, save the file in the earlier release before loading the model in R2016a or later release.

Specify Units for Temperature Signals

When modeling absolute temperature quantities, use units such as `K`, `degC`, `degF`, and `degR`. When modeling temperature *difference* quantities, use units such as `deltaK`, `deltadegC`, `deltadegF`, and `deltadegR`. If you connect a signal that has a temperature difference unit to a block that specifies an *absolute* temperature unit, Simulink detects the mismatch.

Specify Units in MATLAB Function Blocks

You can specify units for input and output data of MATLAB Function blocks by using the **Unit** parameter on the Ports and Data Manager.

During model update, Simulink checks for inconsistencies in units between input or output data ports and the corresponding signals.

Specify Units in Constant Blocks

You can specify units for output data of Constant blocks by using the **Unit** property in the `Simulink.Parameter` object.

Specify Units for Logging and Loading Signal Data

You can include units in signal data that you log or load.

You specify units for logging and loading using `Simulink.SimulationData.Unit` objects. When you log using `Dataset` or `Timeseries` format, Simulink stores the unit information using `Simulink.SimulationData.Unit` objects. If you create MATLAB timeseries data to load, you can specify `Simulink.SimulationData.Unit` object for the `Units` property of the `timeseries` object.

For details, see “Log Signal Data That Uses Units” on page 72-24 and “Load Signal Data That Uses Units” on page 70-59.

Restricting Unit Systems

By default, you can specify units from any of the supported unit systems. However, in large modeling projects, to enforce consistency, you might want to restrict the unit systems that certain components of your model can use. To specify available unit systems for a model, in the configuration parameter **Allowed unit systems**, enter `all` or a comma-separated list containing one or more of `SI`, `SI (extended)`, `CGS`, and `English`. Do not use quotation marks. If your model contains referenced models, you can use the **Allowed unit systems** to restrict units in each of those referenced models. If your model contains subsystems, you can use the Unit System Configuration block to restrict units in the subsystems. You can also optionally use a Unit System Configuration block in a model. In this case, the settings in the Unit System Configuration block override whatever you specify in **Allowed unit systems**.

To restrict unit systems in a model:

1. In the **Unit** parameter of the Inport, Outport, or Signal Specification block, click the link.

If a Unit System Configuration block exists in your model, this link opens the block dialog box. Otherwise, the link opens the **Allowed unit systems** configuration parameter.

2. Specify one or more the desired unit systems, `SI`, `SI (extended)`, `English`, or `CGS`, in a comma-delimited list, or `all`, without quotation marks.

In a parent-child relationship (for example, a top model with a referenced model or subsystem), you can specify different unit systems for each component. However, if a child propagates a unit into a parent that is not in the unit systems specified for the parent, you get a warning.

To check whether there are unit mismatches caused by restricted unit systems in your model hierarchy:

- Press **Ctrl+D** and visually inspect the model for warning badges.
- Use the Model Advisor check **Identify disallowed unit systems**.

See Also

Inport | MATLAB Function | Outport | Signal Specification | `Simulink.BusElement` | `Simulink.Parameter` | `Simulink.Signal` | Unit Conversion | Unit System Configuration

Related Examples

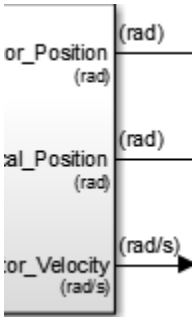
- “Update an Existing Model to Use Units” on page 9-14

More About

- “Displaying Units” on page 9-7
- “Unit Consistency Checking and Propagation” on page 9-9
- “Converting Units” on page 9-12
- “Troubleshooting Units” on page 9-24

Displaying Units

To display signal units in your model, on the **Debug** tab, select **Information Overlays** > **Units**. To select this option programmatically, use the command-line property `ShowPortUnits`.

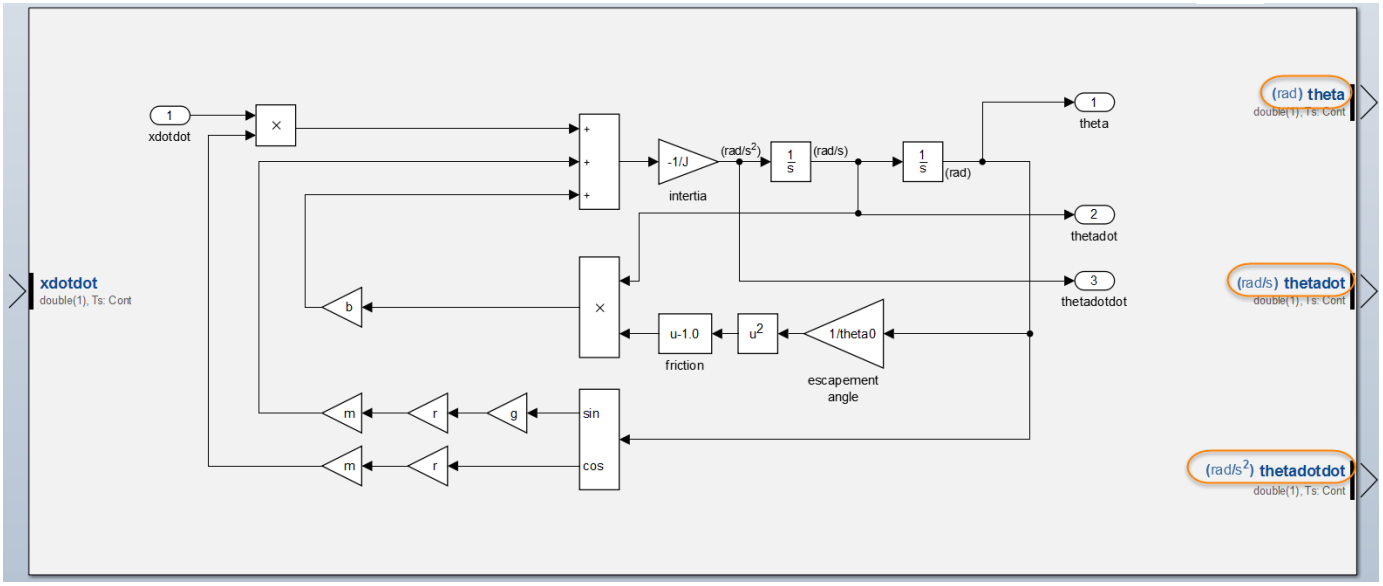


With this option selected, the model dynamically updates port and signal labels to show any changes that you make to units. You do not need to press **Ctrl+D** to update the model. When you simulate your model, the Scope block displays units for connected signals as *y*-axis labels.

Note When you explicitly specify units on input or output blocks, block port labels and signal lines display those units. If a port is set to inherit units or has empty units, port labels and signal lines do not show labels.

Note With the option to display units cleared, you do not see port and signal labels, even when you press **Ctrl+D** to update your model. However, you do see warning or error badges for any unit inconsistency problems that exist in the model.

You can also see units in the interface view of your model. On the **Modeling** tab, under **Design**, click **Model Interface**.



See Also

Inport | MATLAB Function | Outport | Signal Specification | `Simulink.BusElement` | `Simulink.Parameter` | `Simulink.Signal` | Unit Conversion | Unit System Configuration

Related Examples

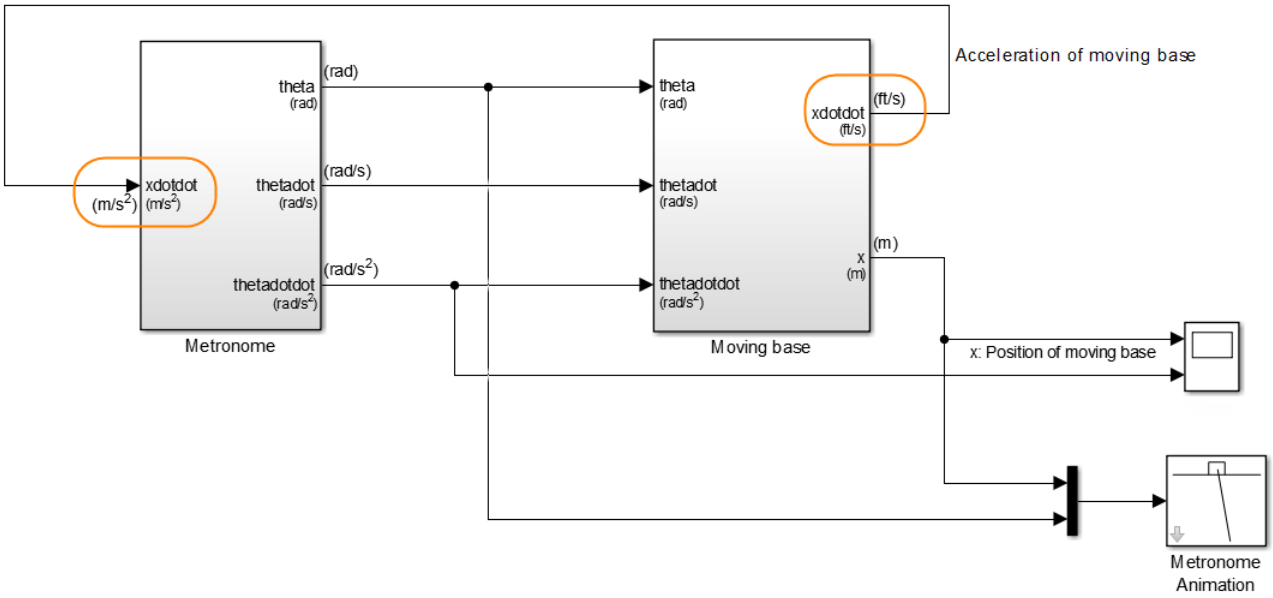
- “Update an Existing Model to Use Units” on page 9-14

More About

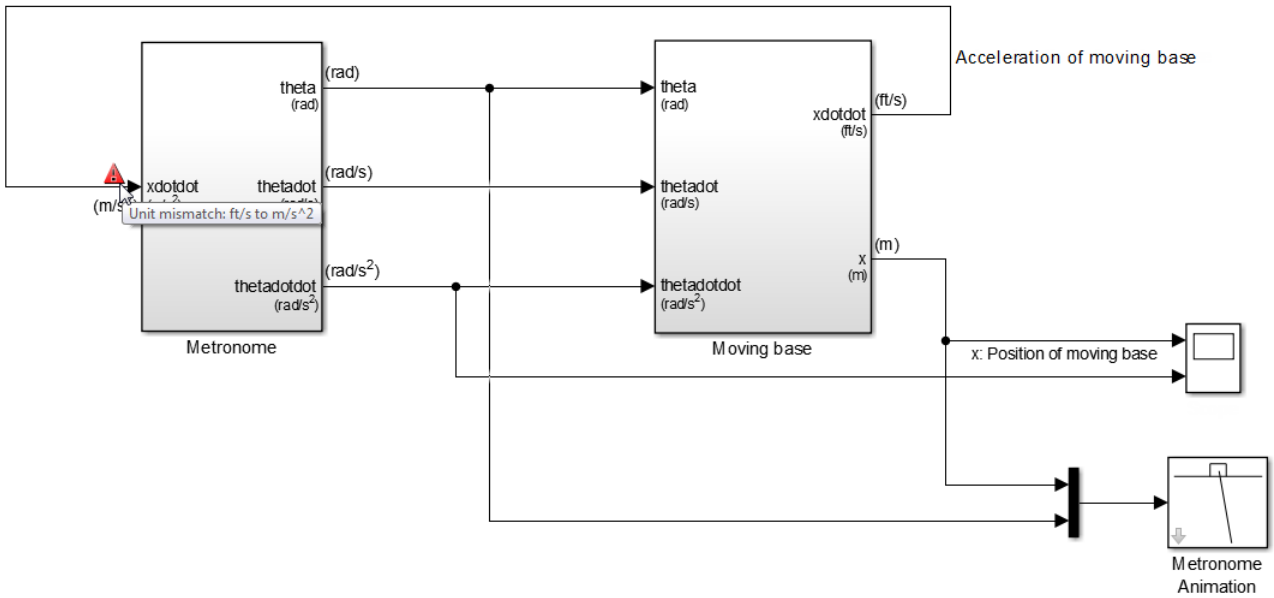
- “Unit Specification in Simulink Models” on page 9-2
- “Unit Consistency Checking and Propagation” on page 9-9
- “Converting Units” on page 9-12
- “Troubleshooting Units” on page 9-24

Unit Consistency Checking and Propagation

Simulink performs unit consistency checking between components. Ports that you connect together — sometimes via intermediate blocks that propagate units — must have the same units. For example, you cannot connect a port with unit `ft/s` to a port with unit `m/s^2`.



By default, Simulink shows the mismatch warning when it detects a mismatch in units between two connected ports. You can press **Ctrl+D** to show mismatched units in your model.



To make this connection valid, you can:

- Explicitly set both port units to the same unit.

- Set the **Unit** parameter of one of the connecting ports to `inherit`.
- Insert a Unit Conversion block between the mismatched units if they are separated by a scaling factor or offset, or if they are inverse units. These units are convertible. For more information, see “Converting Units” on page 9-12.
- Select the **Allow automatic unit conversions** configuration parameter. For more information, see “Converting Units” on page 9-12.

Note Simulink supports variations on unit expressions. For example, one port can have a unit of `m/s^2` and a connected port can have a unit of `m/s/s`. In these cases, Simulink does not display a warning for mismatched units.

When Simulink detects one of these conditions, it displays the inconsistency warning  :

- Disallowed unit system
- Undefined unit

Simulink checks the consistency of unit settings and propagates units across component boundaries. In a model that contains a referenced model, Simulink compiles the referenced model independently of the top model. This independent compilation means that the referenced model cannot inherit units from the top model.

If a port in a referenced model has **Unit** set to `inherit`, it can inherit a unit from any upstream or downstream block in the referenced model. If the port does not inherit a unit from an upstream or downstream block, you can connect it to a port in the top model with any unit.

Simulink passes units through the following blocks that do not change data, known as noncomputation blocks:

- Bus Creator
- Bus Selector
- Bus to Vector
- Data Type Conversion
- Demux
- From
- Goto
- Inport
- Merge
- Model
- Mux
- Outport
- Rate Transition
- Signal Conversion
- Signal Specification
- Subsystem
- Variant Sink

- Variant Source

Note If you supply two or more signals with different units to a Mux block, Simulink applies empty units to the vector signal that the Mux block outputs. Vector signals must have a common unit.

Note If you have a nonvirtual bus in your model (see “Types of Composite Signals” on page 76-2), Simulink sets the unit of the bus to empty. A nonvirtual bus cannot have a unit. However, if the bus element signals themselves have units, Simulink does not change these.

Simulink does not propagate units through blocks that produce new data as output. When signals with units pass through these blocks, the units of these signals become empty. Examples of blocks that do not preserve units because they produce new data as an output include:

- Sum
- Gain
- Filter
- Product

Unit Propagation Between Simulink and Simscape

When modeling physical systems, you might want to integrate components developed in Simulink with components developed in Simscape and its associated physical modeling products. Simscape components use physical signals instead of regular Simulink signals. Therefore, you need Simulink-PS Converter and PS-Simulink Converter converter blocks to connect signals between Simulink and Simscape components.

To specify units for the input and output signals of your Simscape component, you can explicitly specify the units on the converter blocks. When you specify units on a PS-Simulink Converter block that converts a signal from Simscape to Simulink, Simulink propagates the unit settings to the connected Simulink port. However, Simulink cannot propagate a signal unit from Simulink into your Simscape component. To do that, you must explicitly specify the unit on the Simulink-PS Converter block. For more information, see “Physical Units” (Simscape).

See Also

Inport | MATLAB Function | Outport | Signal Specification | `Simulink.BusElement` | `Simulink.Parameter` | `Simulink.Signal` | Unit Conversion | Unit System Configuration

Related Examples

- “Update an Existing Model to Use Units” on page 9-14

More About

- “Unit Specification in Simulink Models” on page 9-2
- “Displaying Units” on page 9-7
- “Converting Units” on page 9-12
- “Troubleshooting Units” on page 9-24

Converting Units

Simulink can convert units between ports when it detects discrepancies that have known mathematical relationships such as:

- Scaling factors
- Conversion factors and offsets, such as `°F` (Fahrenheit) to `°C` (Celsius)
- Scaled, inverse units, such as `mpg` (miles per gallon) and `L/km` (liters per kilometer).

For example, if you connect one port with a unit of `cm` to one with a unit of `mm`, Simulink can automatically scale one unit to work with the other.

To enable Simulink to convert unit mismatches in your model automatically, select the **Allow automatic unit conversions** configuration parameter.

- When Simulink successfully converts signal units at a block port, it displays  .
- When Simulink detects that an automatic conversion is not possible, it displays  .

To manually convert units separated by a conversion factor or offset:

1. Clear the **Allow automatic unit conversions** configuration parameter.
2. Insert a Unit Conversion block between the ports whose units you want to convert.

Tip Automatic conversion of units is a convenience. For better control of units, when Simulink detects a mismatch, consider modifying the units specified at one or the other of the two connected ports.

Automatic Unit Conversion Limitations

Simulink does not support automatic conversion:

- At the root level of models configured for concurrent execution or export-function models. For more information, see “Configure Your Model for Concurrent Execution” on page 14-20 and “Export-Function Models Overview” on page 10-97.
- For fixed-point and integer signals.
- At an input port of a Merge block.
- At any port of an asynchronous Rate Transition block.
- At an input port of a function-call subsystem.
- For bus signals.

See Also

Inport | MATLAB Function | Outport | Signal Specification | `Simulink.BusElement` | `Simulink.Parameter` | `Simulink.Signal` | Unit Conversion | Unit System Configuration

Related Examples

- “Update an Existing Model to Use Units” on page 9-14

More About

- “Unit Specification in Simulink Models” on page 9-2
- “Displaying Units” on page 9-7
- “Unit Consistency Checking and Propagation” on page 9-9
- “Troubleshooting Units” on page 9-24

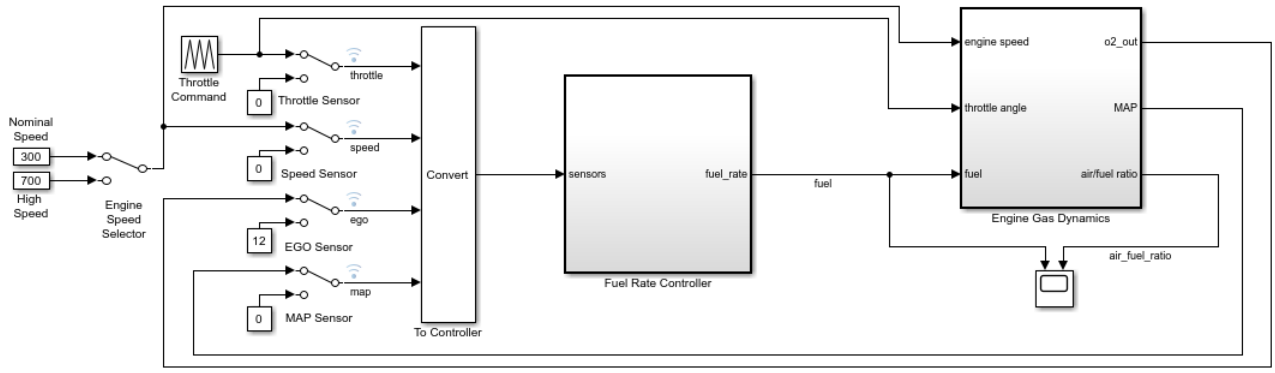
Update an Existing Model to Use Units

This example shows how to add units to an existing model. You see how to:

- Use an incremental workflow to add units to components in your model
- Integrate components that use different unit systems
- Specify units for individual elements of a bus object
- Troubleshoot unit mismatch problems

The model in the example is a fuel control system. The controller (Fuel Rate Controller) and plant (Engine Gas Dynamics) components of the model are nonvirtual subsystems. Nonvirtual subsystems have the **Treat as atomic unit** parameter selected. You introduce units to the plant before introducing units to the controller and connecting signals. You also specify units for the individual elements of a bus object in the model.

Open the `ex_units_fuelsys` example model.



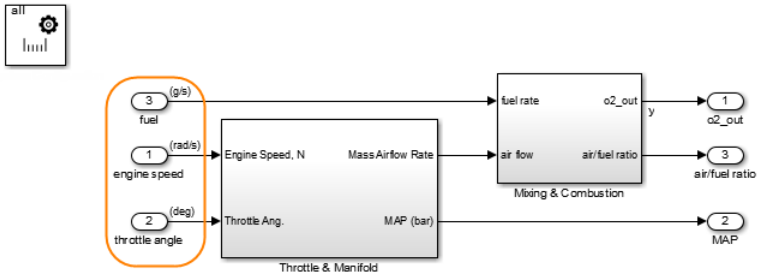
For the top model, the **Allowed unit systems** configuration parameter determines the unit systems the model can use. For each of the plant and controller subsystems, a Unit System Configuration block determines the allowed unit systems.

Component	Allowed Unit Systems
Top model	`SI`
Fuel Rate Controller subsystem (controller)	`all`
Engine Gas Dynamics subsystem (plant)	`all`

In the plant subsystem, on the **Signal Attributes** tab of each Inport block dialog box, set the **Unit** parameter to a value appropriate for the connected physical signal.

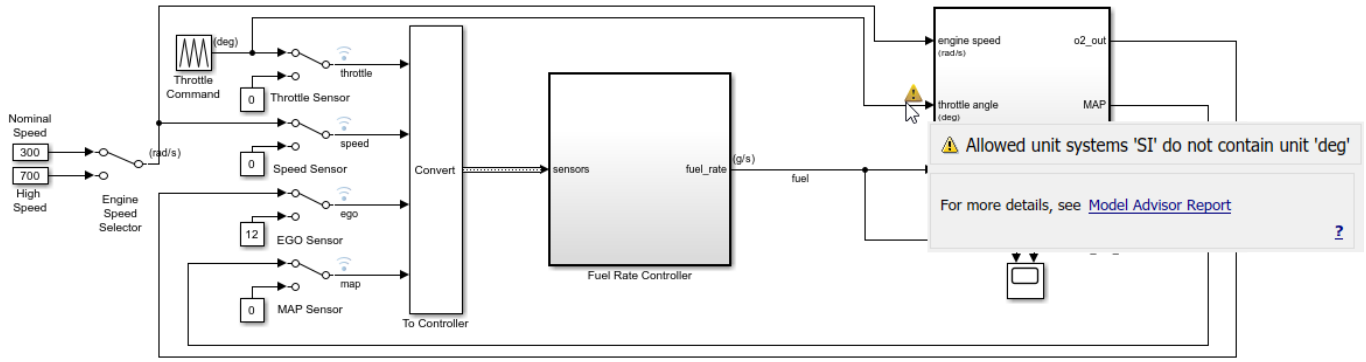
Block	Physical Signal	Unit Parameter Setting
1	engine speed	`rad/s` (radians per second)
2	throttle angle	`deg` (degrees)
3	fuel rate	`g/s` (grams per second)

To display units on ports and signals in the model, on the **Debug** tab, select **Information Overlays** > **Port Units**.



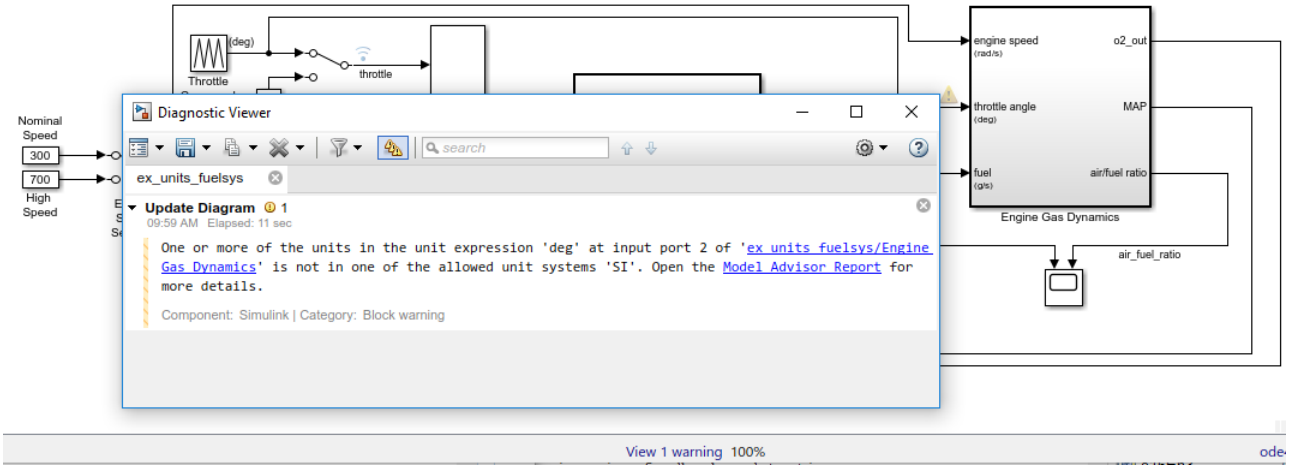
In the plant subsystem, you see units on the Inport blocks and connected signals.

Navigate back to the top model. To compile the model, press **Ctrl+D**, which also performs unit consistency checking.



The model displays a warning to indicate that there is a disallowed unit for the `throttle angle` signal. Clicking the warning icon displays a link to a Model Advisor report that gives you more detail.

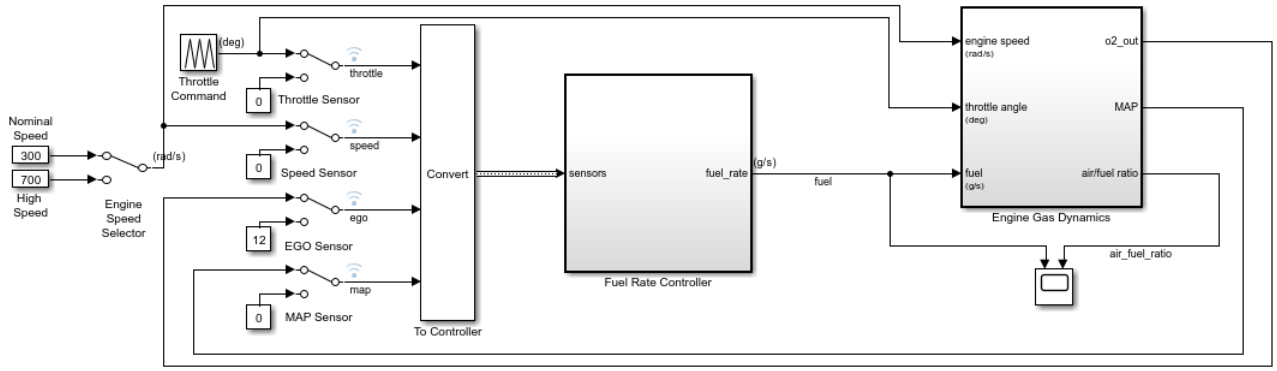
The model also displays the warning at the bottom of the model editing window.



In the plant subsystem, you specified a unit of `deg` (degrees) for the `throttle angle` signal. However, the warning message indicates that degrees are not in the SI unit system. As determined by the **Allowed unit systems** configuration parameter, SI is the only unit system that the top model currently allows. To resolve this warning, you have two options:

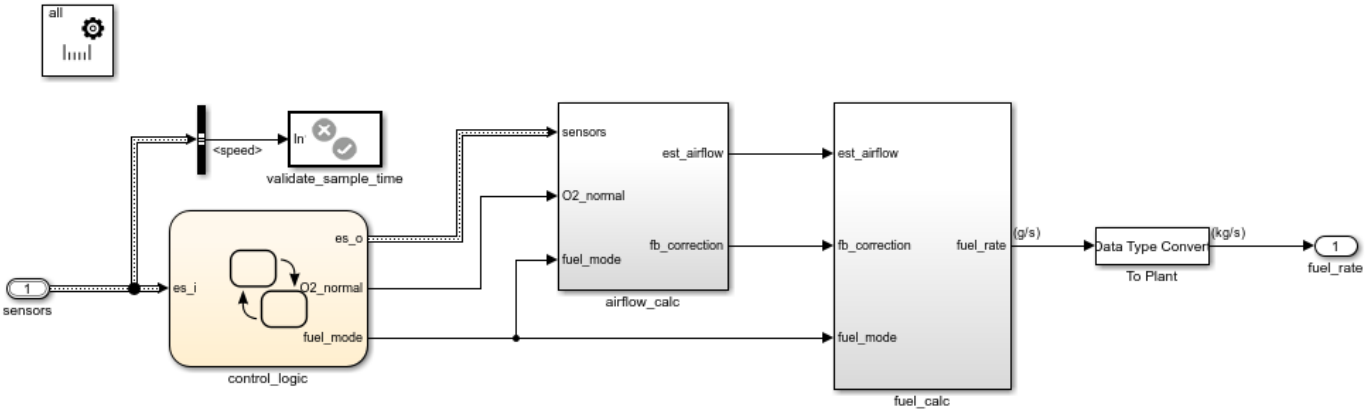
- In the plant subsystem, specify a unit for the `throttle angle` signal that the SI unit system supports. For more information about supported unit systems and the units they contain, see Allowed Units.
- In the top model, change the **Allowed unit systems** configuration parameter to expand the set of allowed unit systems.

In this case, a unit of `deg` for the `throttle angle` signal is appropriate. Instead, to resolve the warning, expand the set of allowed unit systems for the top model. Set the **Allowed unit systems** configuration parameter of the top model to `all`. To recompile the model, press **Ctrl+D**.

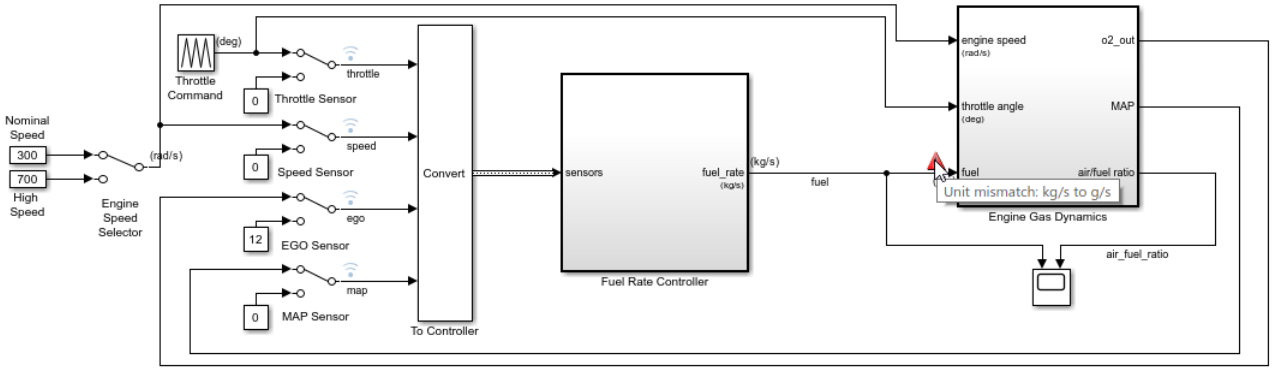


The top model no longer displays warnings.

Now that you have introduced units to the plant and successfully resolved unit inconsistency problems, you can add units to the controller. In the Fuel Rate Controller subsystem, set the **Unit** parameter of the fuel_rate Outport block to `kg/s` (kilograms per second).



Navigate back to the top model. To recompile it, press **Ctrl+D**.

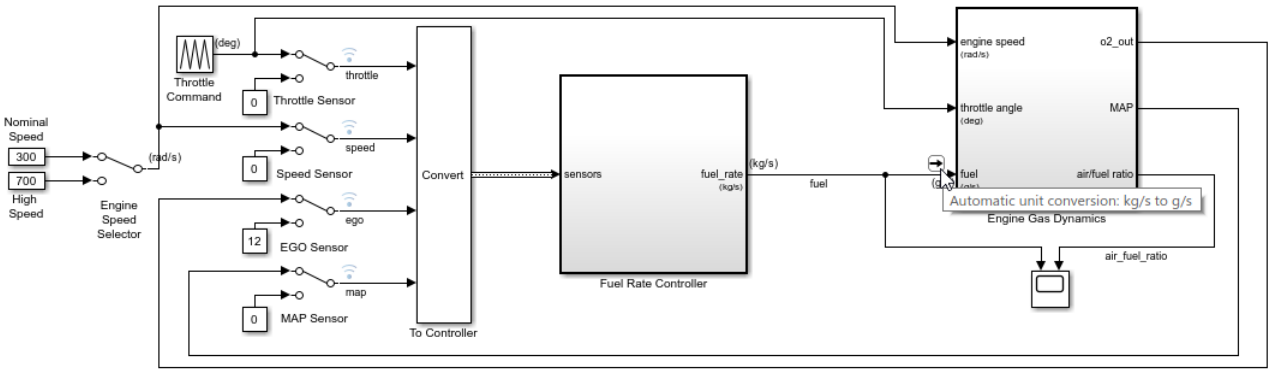


The top model now shows a warning for mismatched units between the controller and plant. To resolve this error, you can:

- Explicitly insert a Unit Conversion block between the two components.
- Select the **Allow automatic unit conversions** configuration parameter.

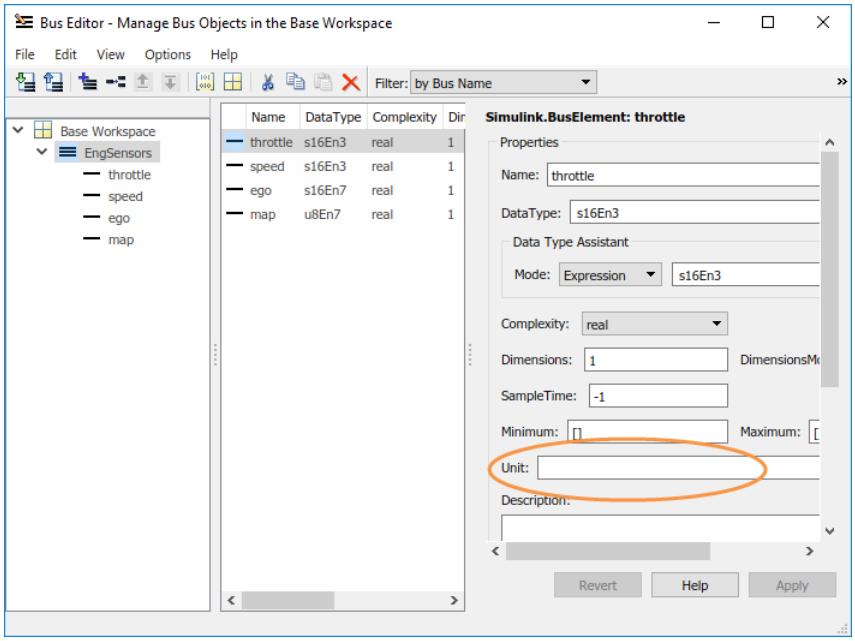
Both options convert units in the same way. A situation in which you might disallow automatic conversions and insert conversion blocks instead is when you are integrating many components in a large system model. In that case, manually inserting conversion blocks can give you an added degree of control of unit conversions in the model. Also, with a conversion block, you can control the data type of the converted signal. This is useful, for instance, when you are modeling for fixed-point precision.

In this case, to enable Simulink to resolve the unit mismatch automatically, select **Allow automatic unit conversions**. To recompile the model, press **Ctrl+D**.



Simulink automatically converts units between the controller and the plant. An automatic conversion icon replaces the warning.

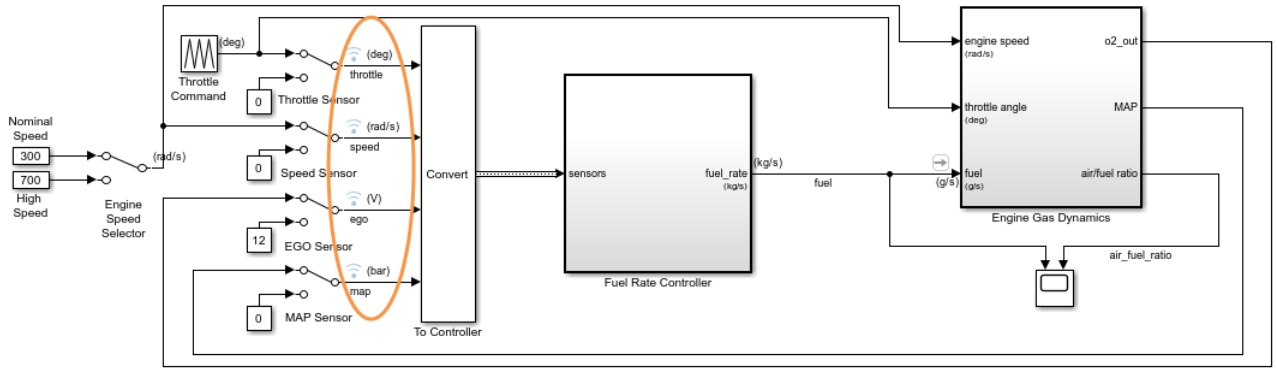
The top model includes a `EngSensors` bus object that passes various sensor signals as a composite signal to the controller. To use the Bus Editor to add units to individual elements of the bus object, on the **Modeling** tab, under **Design**, click **Bus Editor**.



For the `EngSensors` bus object, set the **Unit** parameter of each element.

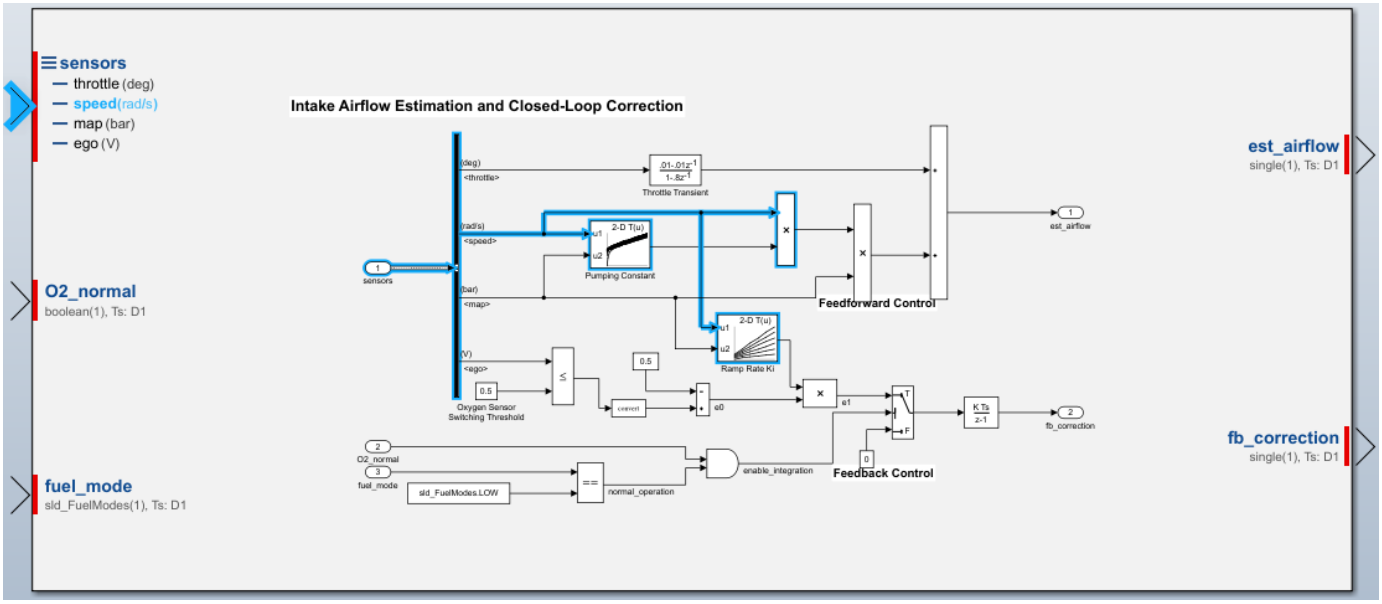
Signal	Unit Parameter Setting
`throttle`	`deg` (degrees)
`speed`	`rad/s` (radians per second)
`ego`	`V` (volts)
`map`	`bar` (bars)

To recompile the model, press **Ctrl+D**.



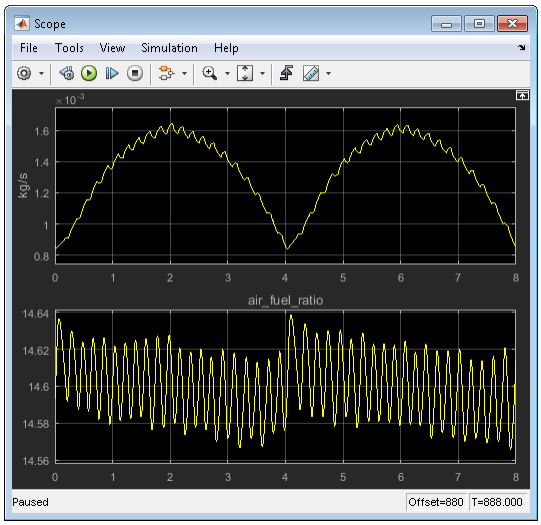
The model shows units on the individual elements of the bus object.

You can also see the units in the interface view of your model. On the **Modeling** tab, under **Design**, click **Model Interface**.



The airflow_calc block of the controller subsystem displays units on the individual elements of the bus object, both at the component interface and within the component.

After you introduce units incrementally and resolve inconsistency and mismatch issues, you can simulate the model.



For the `fuel` signal that is connected to the scope, the plot window displays the associated units of `kg/s` as a *y*-axis label.

See Also

Inport | Outport | Unit Conversion | Unit System Configuration

More About

- “Unit Specification in Simulink Models” on page 9-2
- “Nonvirtual and Virtual Blocks” on page 36-2
- “Displaying Units” on page 9-7
- “Unit Consistency Checking and Propagation” on page 9-9
- “Converting Units” on page 9-12
- “Troubleshooting Units” on page 9-24

Working with Custom Unit Databases

In Simulink models, you specify units from a unit database. The unit database comprises units from the following unit systems:

- `SI` — International System of Units
- `SI (extended)` — International System of Units (extended)
- `English` — English System of Units
- `CGS` — Centimetre-gram-second System of Units

By default, Simulink supports only the units and unit systems listed in Allowed Units. To introduce additional units from other unit systems at a system-wide level and use those new units as you would those listed in Allowed Units, create and load a new unit database with these functions:

- `createCustomDBFromExcel` — Creates a custom unit database file from an Excel spreadsheet that contains definitions for the custom unit database. On all supported platforms, the `createCustomDBFromExcel` function supports: `.xls` and `.xlsx` files.
- `rehashUnitDBs` — Loads custom unit databases by rehashing unit database files on the MATLAB path.

Custom Units Spreadsheet Format

Spreadsheets must have these columns in any order:

- **name** — Unit name, using any graphical Unicode characters except `@`, `*`, `/`, `^`, `(`, `)`, `+`, `\`, `"`, `'`, `{`, `}`, `[`, `]`, `<`, `>`, `&`, `-`, `::`, and white space.
- **symbol** — Symbol of unit, using any graphical Unicode characters except `@`, `*`, `/`, `^`, `(`, `)`, `+`, `\`, `"`, `'`, `{`, `}`, `[`, `]`, `<`, `>`, `&`, `-`, `::`, and white space.
- **asciiSymbol** — Symbol of unit, in ASCII.
- **displayName** — Name of unit displayed in model in LaTeX format.
- **definitionExpression** — Definition of the unit in terms of predefined units, such as seven base SI units.
- **conversionFactor** — Conversion factor between the unit and its definition.
- **conversionOffset** — Conversion offset between the unit and its definition.
- **physicalQuantity** — Valid physical quantities. See table 'Physical Quantities' in `showunitslist`.
- **provenance** — Optional column. List of unit provenances, separated by commas.

Follow these guidelines when developing spreadsheets:

- If the input spreadsheet contains more then one worksheet, you must name the worksheets with the prefixes `unit`, `physicalQuantity`, or `provenance`.
- If there are multiple instances of a prefix in a worksheet, the function uses all of the worksheets to create the database:
  - `unit` — Creates units
  - `physicalQuantity` — Creates physical quantities
  - `provenance` — Creates provenances

Optionally, you can define physical quantities in another worksheet of the input spreadsheet. A worksheet defining physical quantities contains these columns in any order:

- **name** — Physical quantity name, using any graphical Unicode characters except `@`, `*`, `/`, `^`, `(`, `)`, `+`, `\`, `"`, `'`, `{`, `}`, `[`, `]`, `<`, `>`, `&`, `-`, `::`, and white space.
- **definitionExpression** — Definition of physical quantity in terms of predefined physical quantities.
- **provenance** — Optional. List of physical quantity provenances, separated by commas.

Optionally, you can also define provenances in another worksheet of the input spreadsheet. A worksheet defining provenances contains these columns in any order:

- **identifier** — Identifier of provenance
- **title** — Title of provenance
- **subTitle** — Subtitle of provenance
- **organization** — Organization of provenance
- **fullName** — Full name of provenance
- **urlList** — List of URL links of provenance
- **edition** — Provenance edition
- **year** — Provenance year

Define Custom Units in Excel Spreadsheet

First, create an Excel spreadsheet following the guidelines in “Custom Units Spreadsheet Format” on page 9-21. Use unit definitions, one per row, such as:

	A	B	C	D	E	F	G	H
1	**name**	**symbol**	**asciiSymbol**	**displayName**	**definitionExpression**	**conversionFactor**	**conversionOffset**	**physicalQuantity**
2	ounce_force	ozf	ozf	{\rm{}oz_{force}}	oz*gn	1	0	force
3	troy_oz	ozt	ozt	{\rm{}oz_{troy}}	oz	1	0	mass
4	hogshead	Hhd	Hhd	{\rm{}Hhd}	gal	79	0	volume

Save this spreadsheet in a file such as `unitsDB.xlsx`. You can then create the database and load it.

Create and Load Custom Unit Database

This example shows how to create a custom units database and then load it.

Create the database using the spreadsheet included in this example.

```
createCustomDBFromExcel('unitsDB.xlsx')
```

The function creates `unitsDB.slunitdb.mldatx` in the current folder.

Load the new units database into memory.

```
rehashUnitDBs
```

To verify that the new database has been loaded, open the `slex_customunits` model and apply custom units on any of the output ports:

```
slex_customunits
```

See Also

Unit Conversion | Unit System Configuration | `createCustomDBFromExcel` | `rehashUnitDBs` | `showunitslist`

More About

- “Unit Specification in Simulink Models” on page 9-2
- “Nonvirtual and Virtual Blocks” on page 36-2
- “Displaying Units” on page 9-7
- “Unit Consistency Checking and Propagation” on page 9-9
- “Converting Units” on page 9-12
- “Troubleshooting Units” on page 9-24

Troubleshooting Units

In this section...

To help you troubleshoot issues with unit settings, Simulink uses Model Advisor checks to generate a report useful for larger models.

By default, Simulink flags unit usage issues, such as mismatched units, with warnings. Warnings enable you to continue working despite mismatched units. You can reduce the number of warnings you see by setting the configuration parameter **Units inconsistency messages** to none.

Undefined Units

Simulink does not support custom unit specifications. For more information about supported unit systems and the units they contain, see Allowed Units.

The Model Advisor check “Identify undefined units in the model” identifies undefined units.

Overflow and Underflow Errors or Warning

You can get overflow and underflow errors or warnings when using the Unit Conversion block. If you get:

- Overflow messages, change the data type at the output port to one with a better range
- Underflow messages, change the data type at the output port to one with better precision

Mismatched Units Detected

At the boundary of a component, Simulink detects if the units of two ports do not match. To see the tooltip, hover over the warning badge. If the unit is convertible, Simulink displays advice on fixing the issue.

The Model Advisor check “Identify unit mismatches in the model” identifies mismatched units.

Mismatched Units Detected While Loading

At the boundary of a component, Simulink detects if the units of two ports do not match. To see the tooltip, hover the warning badge. When possible, Simulink displays advice on fixing the issue.

The Model Advisor check “Identify unit mismatches in the model” identifies mismatched units.

Disallowed Unit Systems

Simulink supports only the unit systems listed in the tables of allowed units.

The Model Advisor check “Identify disallowed unit systems in the model” identifies unit systems that are not allowed in the configured units systems.

Automatic Unit Conversions

If the **Allow automatic unit conversions** configuration parameter is set, Simulink supports the automatic conversion of units. Simulink flags automatically converted units with the  badge.

For a list of the automatic unit conversions, use the Model Advisor check “Identify automatic unit conversions in the model”.

Unsuccessful Automatic Unit Conversions

If the **Allow automatic unit conversions** configuration parameter is set, Simulink supports the automatic conversion of units. If Simulink cannot perform the automatic unit conversion, Simulink returns a warning (). In such cases, consider manually specifying the unit.

Tip Automatic unit conversion is a convenience. For better control of units, you can manually set the units for two connecting ports.

Simscape Unit Specification Incompatible with Simulink

If these are true:

- You define a new unit to your unit registry by using the `pm_addunit` function.
- You use the new unit with the Simulink-PS Converter or PS-Simulink Converter block.
- Your new unit conflicts with an existing one in the Simulink database.

Simulink returns a warning about a potential incorrect calculation ().

See Also

Inport | MATLAB Function | Outport | Signal Specification | `Simulink.BusElement` | `Simulink.Parameter` | `Simulink.Signal` | Unit Conversion | Unit System Configuration | `createCustomDBFromExcel` | `rehashUnitDBs` | `showunitslist`

More About

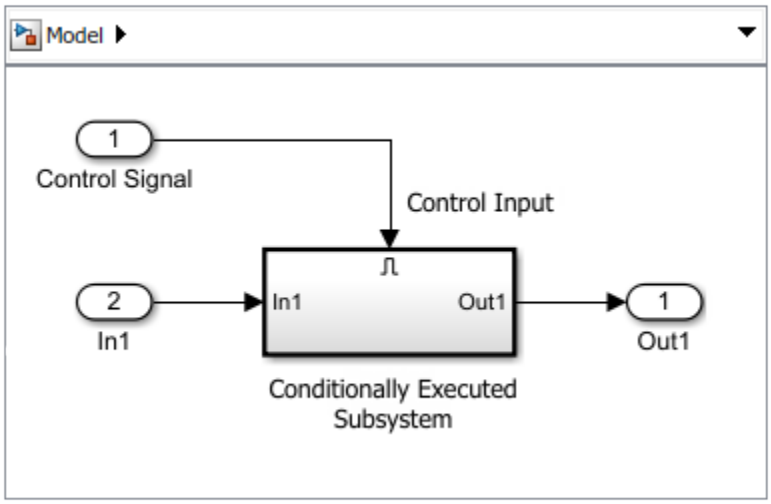
- “Unit Specification in Simulink Models” on page 9-2
- “Displaying Units” on page 9-7

Conditional Subsystem

Models with While Structures

Conditionally Executed Subsystems Overview

A conditionally executed subsystem is an atomic subsystem that allows you to control its execution with an external signal. The external signal, called the control signal, is attached to the control input port. Conditional subsystems are useful when you create complex models that contain components whose execution depends on other components.



Simulink supports these types of conditional subsystems:

- Enabled Subsystem — Executes at each time step while the control signal is positive. Execution starts at the time step when the control signal crosses zero from the negative to the positive direction. See “Using Enabled Subsystems” on page 10-10.
- Triggered Subsystem — Executes at each time step when the control signal rises or falls to zero or crosses zero. See “Using Triggered Subsystems” on page 10-17.
- Enabled and Triggered Subsystem — Executes at the time step when the enable control signal has a positive value and the trigger control signal rises or falls to zero. See “Using Enabled and Triggered Subsystems” on page 10-21.
- Function-Call Subsystem — Executes when the control signal receives a function-call event. Events can occur one or more time during a time step. A Stateflow chart, Function-Call Generator block, S-Function block, or Hit Crossing block can provide function-call events. See “Using Function-Call Subsystems” on page 10-34.

Model Examples

- “Simulink Subsystem Semantics”

See Also

Blocks
Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Related Examples

- “Using Enabled Subsystems” on page 10-10
- “Using Triggered Subsystems” on page 10-17
- “Using Enabled and Triggered Subsystems” on page 10-21
- “Using Function-Call Subsystems” on page 10-34

More About

- “Conditional Subsystem Initial Output Values” on page 10-37
- “Conditional Subsystem Output Values When Disabled” on page 10-54

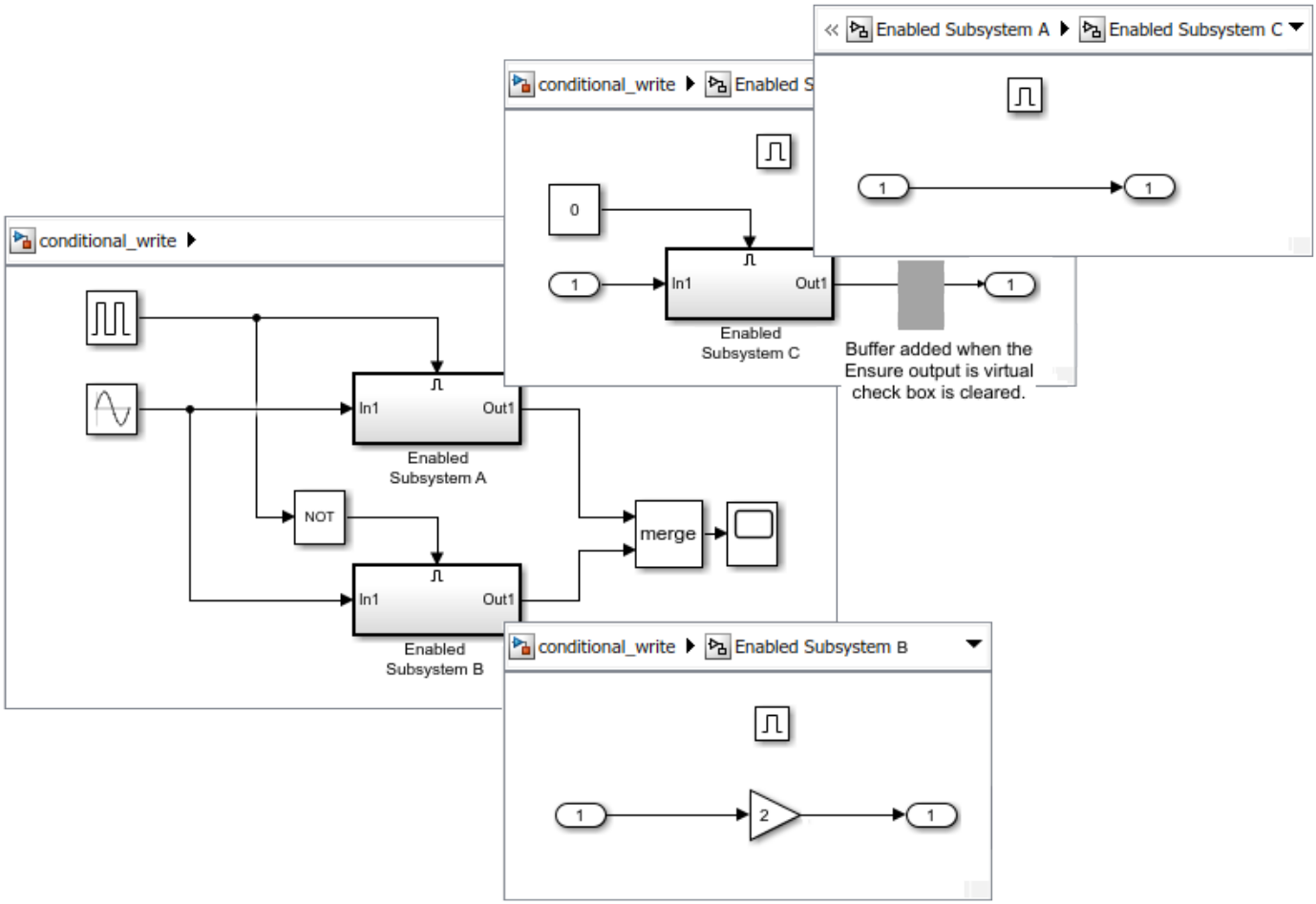
Ensure Outport is Virtual

Simulink can add a hidden signal buffer before the Outport block in a conditional subsystem or before an Outport block at the top-level of a model. This buffer ensures consistent initialization of the Outport block signal.

In a few cases, adding a signal buffer with a conditional output signal or partial write signal can cause a different simulation result. The parameter **Ensure outport is virtual** is an option on an Outport block to remove the buffer. Select this option when you are concerned with conditional or partial write signals.

Conditional Output Signal

Consider the following model. To open model, see `ex_conditional_write`.



The Merge block combines its inputs into a single signal whose value at any time is equal to the most recently computed output of its driving blocks.

For the case with most models, clear the **Ensure outport is virtual** check box on the Outport block connected to `Enabled Subsystem C`.

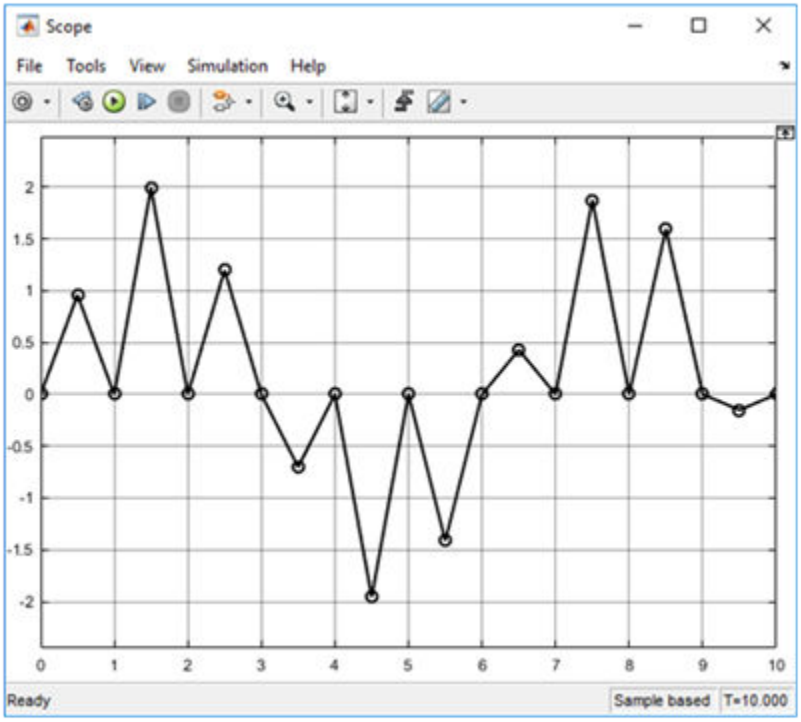
- The Outport block follows non-virtual semantics. A hidden buffer is inserted if needed before the Outport block.
- The buffer provides consistent initialization of the Outport block signal.

Time 0: A runs, C does not run, but because the buffer is in A, it runs and copies the initial value of zero to the Outport block. B does not run. The merge signal is zero from the output from A.

Time 0.5: A does not run. B runs and outputs a sine wave. The merge signal is the sine wave from B.

Time 1. A runs, C does not run, but the buffer again runs and copies over the initial value of zero to the Outport block. B does not run. The merge signal is again the initial value of A, not the last value from B.

Simulating the model with a fixed-step, produces the following result.



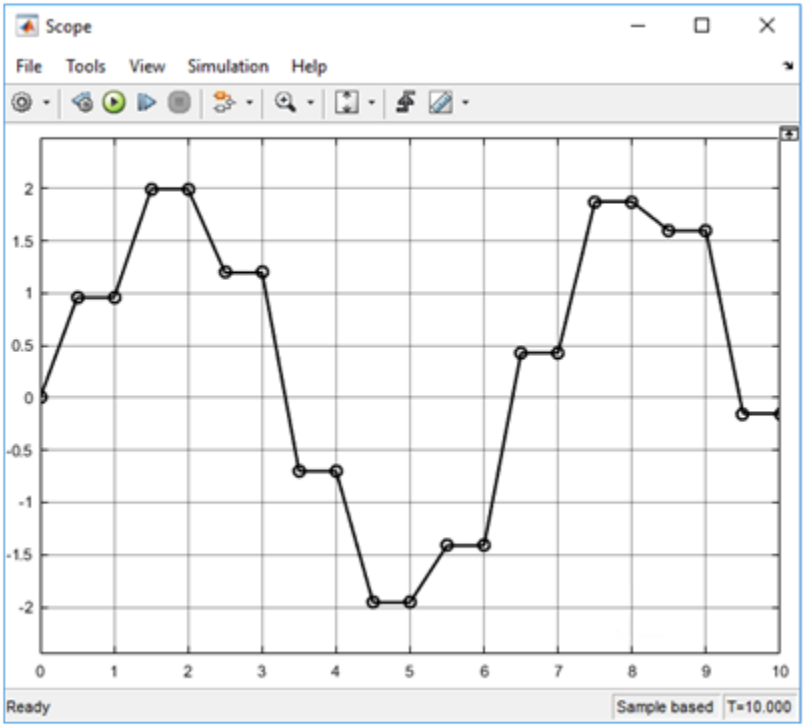
For the case where you are concerned with conditional and partial writes, select (check) the **Ensure outport is virtual** check box for the Outport block connected to Enabled Subsystem C.

- The Outport block follows virtual semantics.
- A hidden buffer is not inserted before the Outport block of the Subsystem.
- If Simulink determines a buffer is needed, an error is displayed.

Time 0: A runs, C does not run. B does not run. Merge signal is the initial value of the signal.

Time 0.5 sec: A does not run. B runs and outputs a sine wave. The merge signal is the value of the sine wave from B.

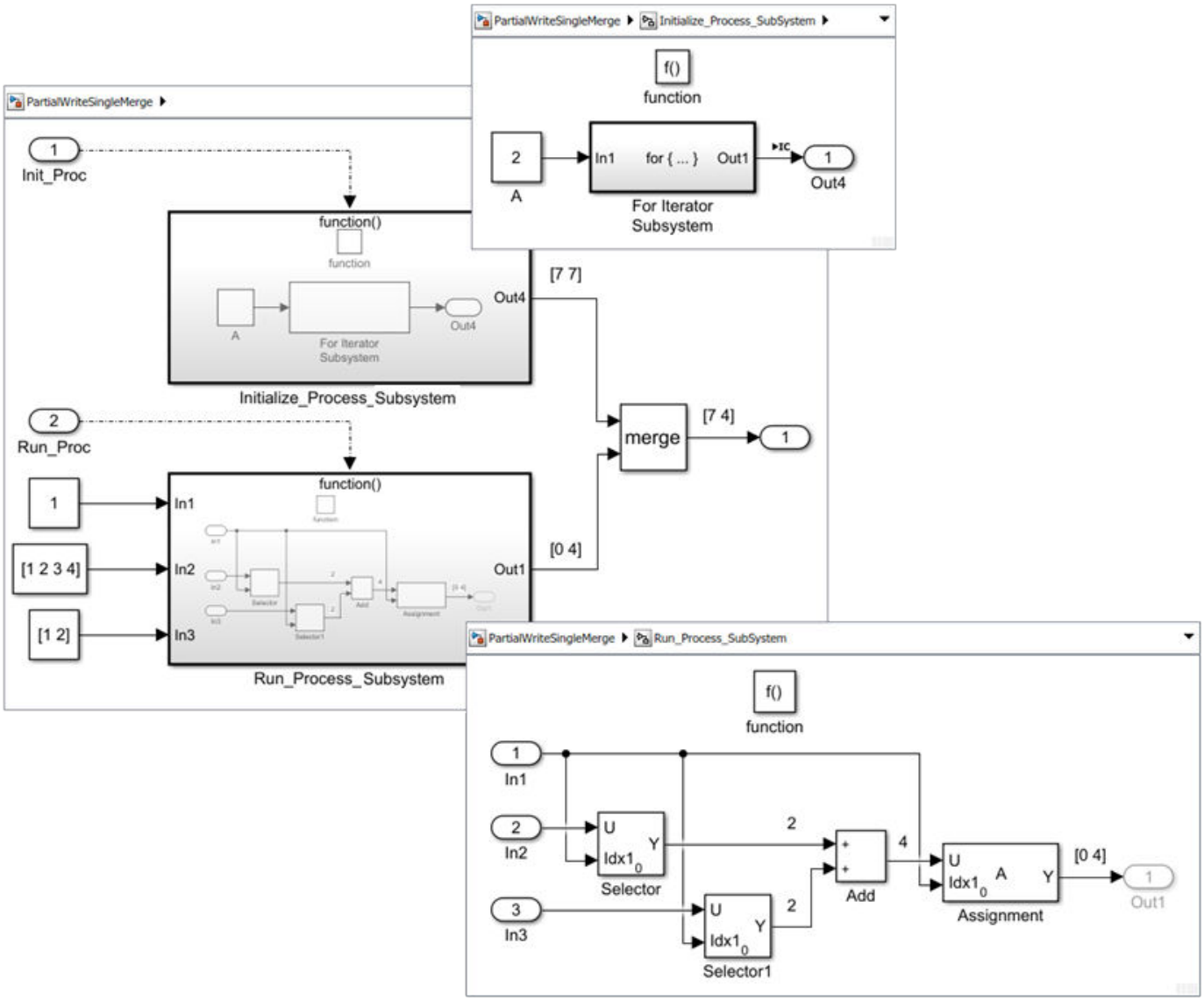
Time 1: A runs, C does not run. B does not run. The merge signal is the most recently computed output which was the sine wave from B.



Partial Write Signals With a Merge Block

A typical modeling pattern is where you want to initialize a vector signal and then periodically update partial elements of the signal based upon certain conditions or inputs. One way of modeling this pattern is to use a Merge block whose inputs are from two Function-Call Subsystem blocks. One subsystem is the initialize task while the other subsystem is a periodic write task.

The model below demonstrates this pattern. The Initialize_Process_Subsystem is called once at the beginning of a simulation to initialize a vector signal. The Run_Process_Subsystem is called to partially write to elements of the vector. However, the output from the Assignment block needs a path where hidden buffers do not make copies of the vector. Selecting the **Ensure outport is virtual** check box on the Outport block removes a hidden buffer. If Simulink determines the buffer is needed an error is displayed. To open model, see `ex_partial_write_single_merge`.



The Initialize_Process_SubSystem

- Initializes each element of a 2 element vector with a value of 7.
- Outputs the vector `[7 7]`.

The Run_Process_Subsystem

- Inputs an index value of 1, but because the **Index mode** parameter for the Selector blocks is set to `Zero-based`, they select the 2nd elements from the input vectors.
- Adds the output scalar values from the Selector blocks for a result of 4.
- Because the Assignment block **Index mode** parameter is set to `Zero-based` and the input index Idx1 is 1, the output signal needs to be a vector with length 2. After setting the **Output size** parameter to 2, the Assignment block is able to write to the 2nd element.
- Selecting the **Ensure outport is virtual** check box removes the hidden buffer.

Code generated from this model includes two functions. `Init_Proc` and `Run_Proc`.

```
/* Model step function */
void Init_Proc(void)
{
  int32_T s3_iter;

  /* Initialize signal vector elements with 7.0 */
     for (s3_iter = 0; s3_iter < 2; s3_iter++) {
    PartialWriteSingleMerge_DW.Assignment[s3_iter] = 7.0;
  }

  for (s3_iter = 0; s3_iter < 2; s3_iter++) {
    PartialWriteSingleMerge_Y.Out4[s3_iter] =
      PartialWriteSingleMerge_DW.Assignment[s3_iter];
  }
}

/* Model step function */
void Run_Proc(void)
{
    /* Write to element 1 of the output signal vector */
  PartialWriteSingleMerge_Y.Out4[1] = 4.0;
}
```

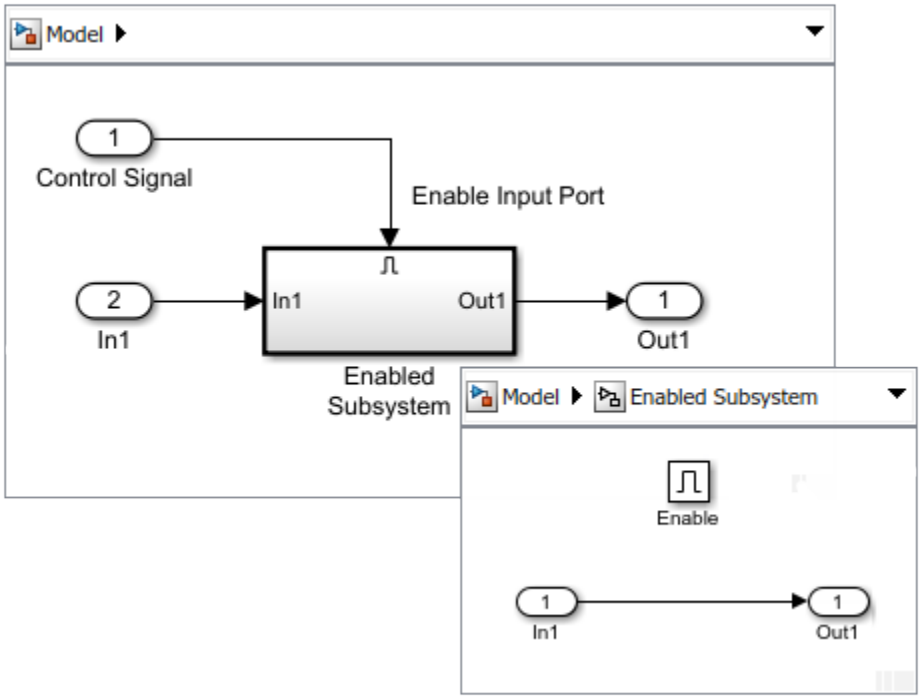
Using Enabled Subsystems

In this section...

An enabled subsystem is a conditionally executed subsystem that runs once at each major time step while the control signal has a positive value. If the signal crosses zero during a minor time step, the subsystem is not enabled or disabled until the next major time step.

The control signal can be either a scalar or a vector.

- If a scalar value is greater than zero, the subsystem executes.
- If any one of the vector element values is greater than zero, the subsystem executes.

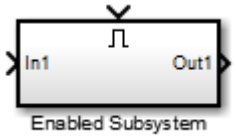


Create an Enabled Subsystem

To create an enabled subsystem:

1 Add an Enabled Subsystem block to your model.

- Copy a block from the Simulink Ports & Subsystems library to your model.
- Click the model diagram, start typing `enabled`, and then select Enabled Subsystem.



2. Set initial and disabled values for the Outport blocks. See “Conditional Subsystem Initial Output Values” on page 10-37 and “Conditional Subsystem Output Values When Disabled” on page 10-54.
3. Specify how subsystem states are handled when the subsystem is enabled.

   Open the subsystem block, and then open the parameter dialog box for the Enable port block. From the **States when enabling** drop-down list, select:

   - `held` — States maintain their most recent values.
   - `reset` — If the subsystem is disabled for at least one time step, states revert to their initial conditions.

     In simplified initialization mode (default), the subsystem elapsed time is always reset during the first execution after becoming enabled. This reset happens regardless of whether the subsystem is configured to reset on being enabled. See “Underspecified initialization detection”.

   For nested subsystems whose Enable blocks have different parameter settings, the settings for the child subsystem override the settings inherited from the parent subsystem.
4. Output the control signal from the Enable block.

   In the parameter dialog box for the Enable Block, select the **Show output port** check box.

   Selecting this parameter allows you to pass the control signal into the enabled subsystem. You can use this signal with an algorithm that depends on the value of the control signal.

Blocks in Enabled Subsystems

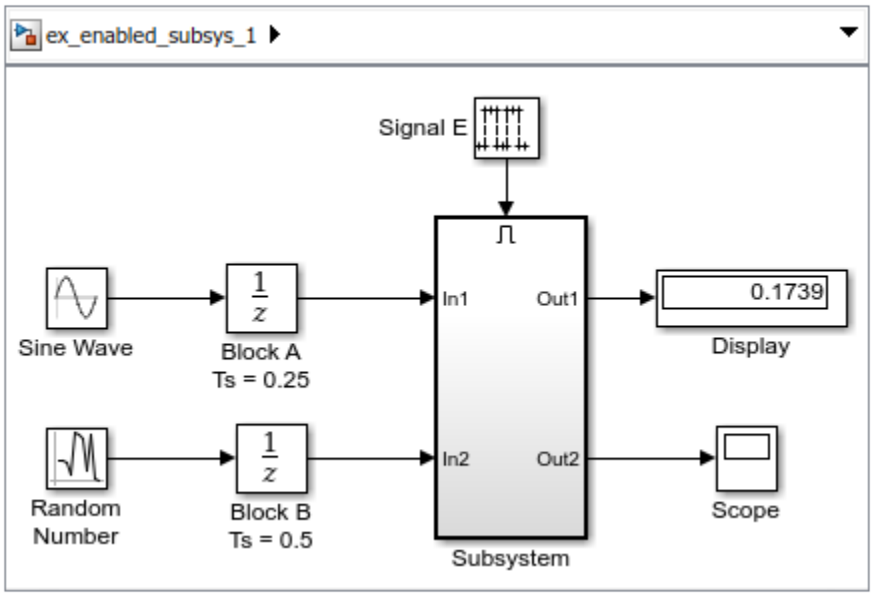
Discrete Blocks

Discrete blocks in an enabled subsystem execute only when the subsystem executes, and only when their sample times are synchronized with the simulation sample time.

Consider the ex_enabled_subsys_1 model, which contains four discrete blocks and a control signal. The discrete blocks are:

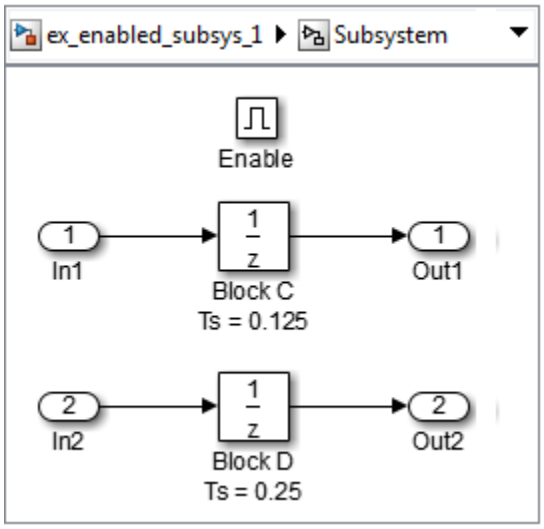
- Block A, with the sample time of 0.25 seconds
- Block B, with the sample time of 0.5 seconds

Signal E is the enable control signal generated by a Pulse Generator with a sample time of 0.125. Its output changes value from 0 to 1 at 0.375 seconds and returns to 0 at 0.875 seconds.

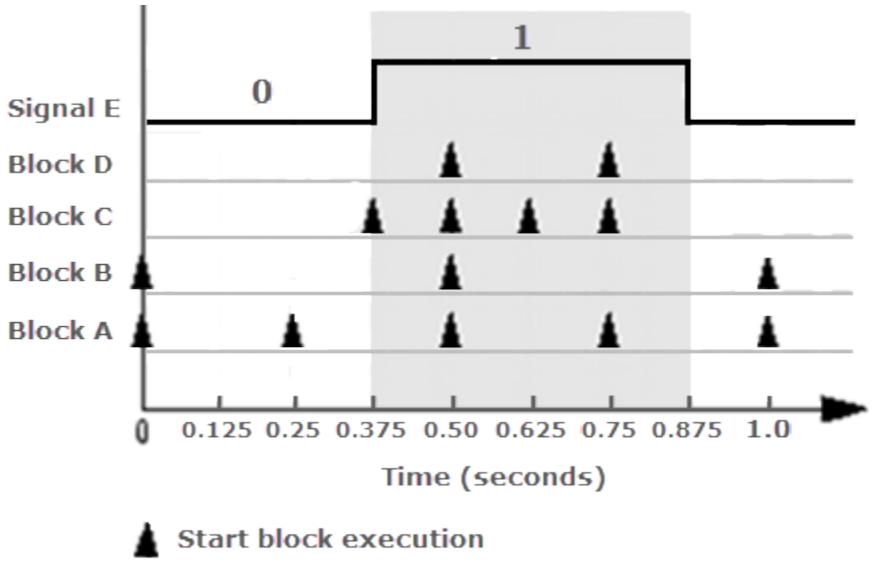


The discrete blocks in the enabled subsystem are:

- Block C, within the enabled subsystem, with the sample time of 0.125 seconds
- Block D, also within the enabled subsystem, with the sample time of 0.25 seconds



Discrete blocks execute at sample times shown.



Blocks A and B execute independently of the enable control signal because they are not part of the enabled subsystem. When the enable control signal becomes positive, blocks C and D execute at their assigned sample rates until the enable control signal becomes zero again. Block C does not execute at 0.875 seconds when the enable control signal changes to zero.

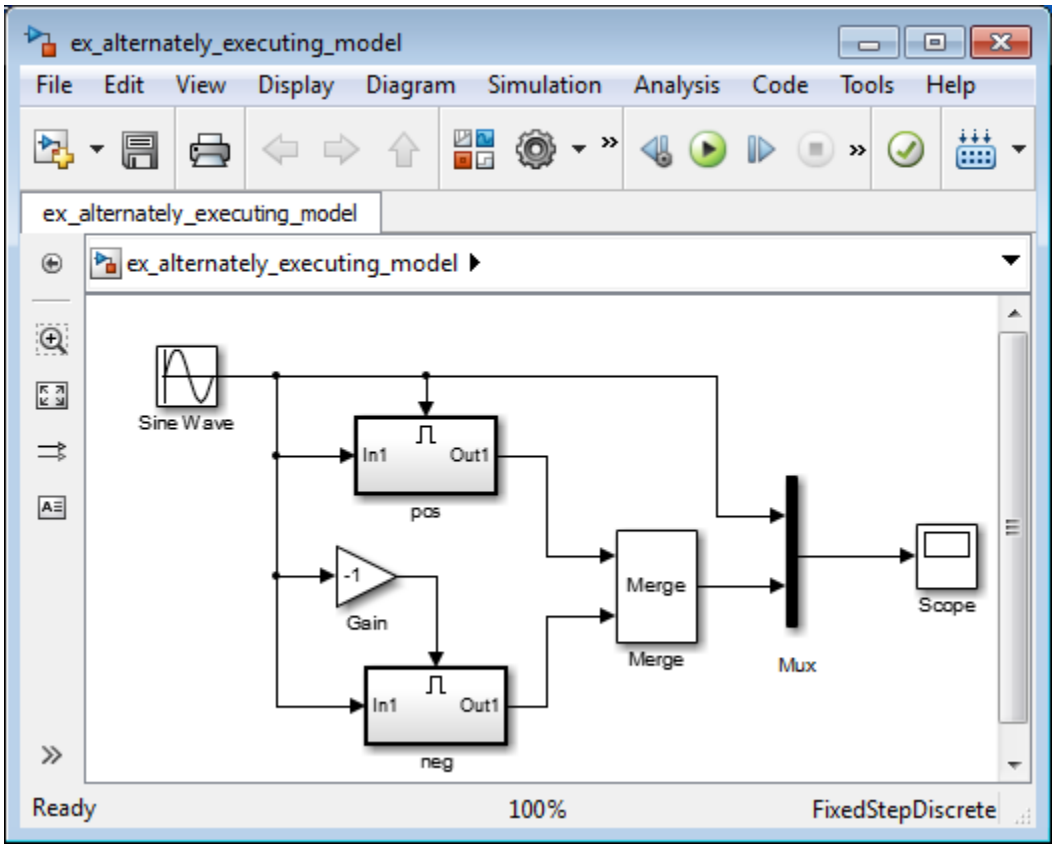
Goto Blocks

Enabled subsystems can contain Goto blocks. However, only output ports for blocks with state can connect to Goto blocks. See the `Locked` subsystem in the model `sldemo_clutch`, for an example of using Goto blocks in an enabled subsystem.

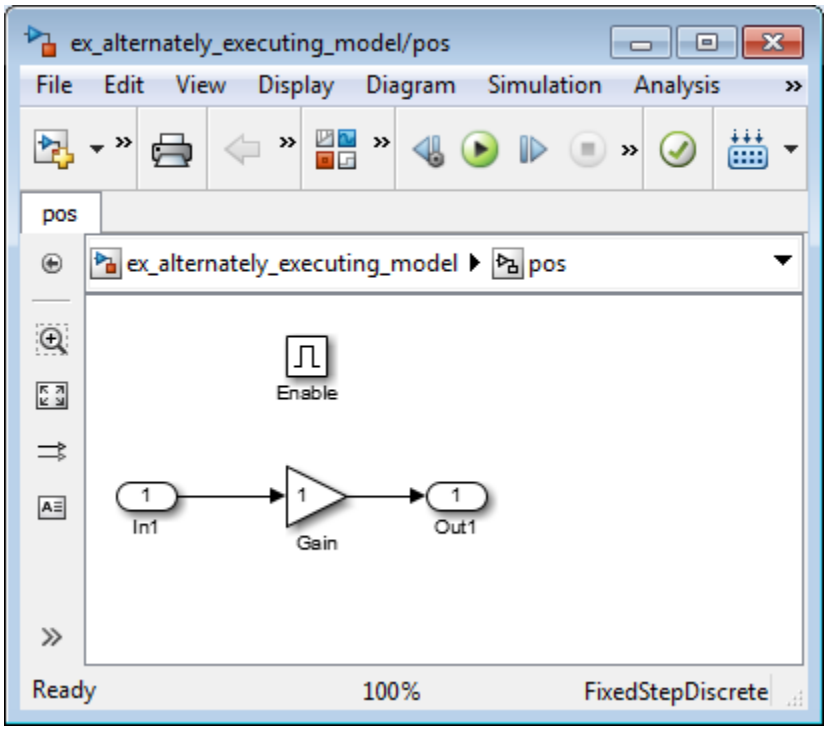
Alternately Executing Enabled Subsystem Blocks

You can use conditional subsystems with Merge blocks to create sets of subsystems that execute alternately, depending on the current state of the model.

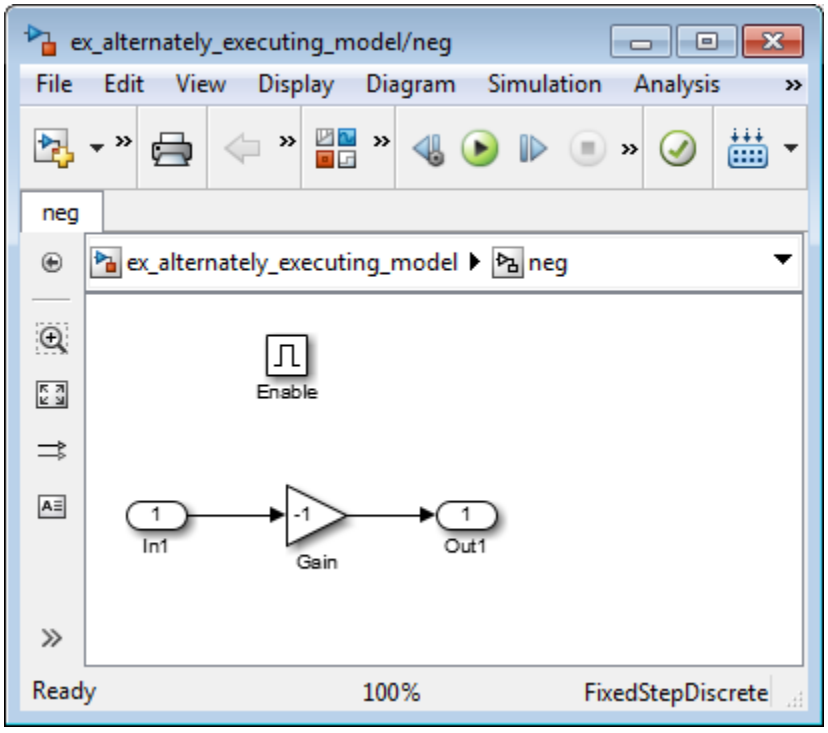
Consider a model that uses two Enabled Subsystem blocks and a Merge block to model a full-wave rectifier (a device that converts AC current to pulsating DC current). To open model, see `ex_alternately_executing_model`.



Open the `pos` subsystem. The subsystem is enabled when the AC waveform is positive and passes the waveform unchanged to its output.

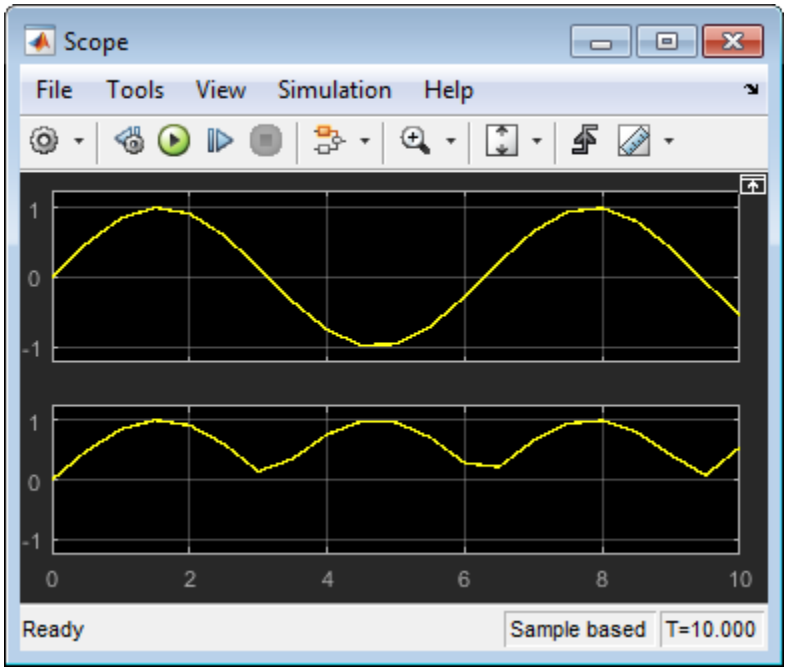


Open the neg subsystem. The subsystem is enabled when the waveform is negative and inverts the waveform.



The Merge block passes the output of the currently enabled subsystem along with the original waveform to the Scope block.

Running a simulation and then open the Scope block.



Model Examples

- “Simulink Subsystem Semantics”
- “Building a Clutch Lock-Up Model”
- “Enabled Subsystems”
- “Advanced Enabled Subsystems”

See Also

Blocks
Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Related Examples

- “Using Triggered Subsystems” on page 10-17
- “Using Enabled and Triggered Subsystems” on page 10-21
- “Using Function-Call Subsystems” on page 10-34

More About

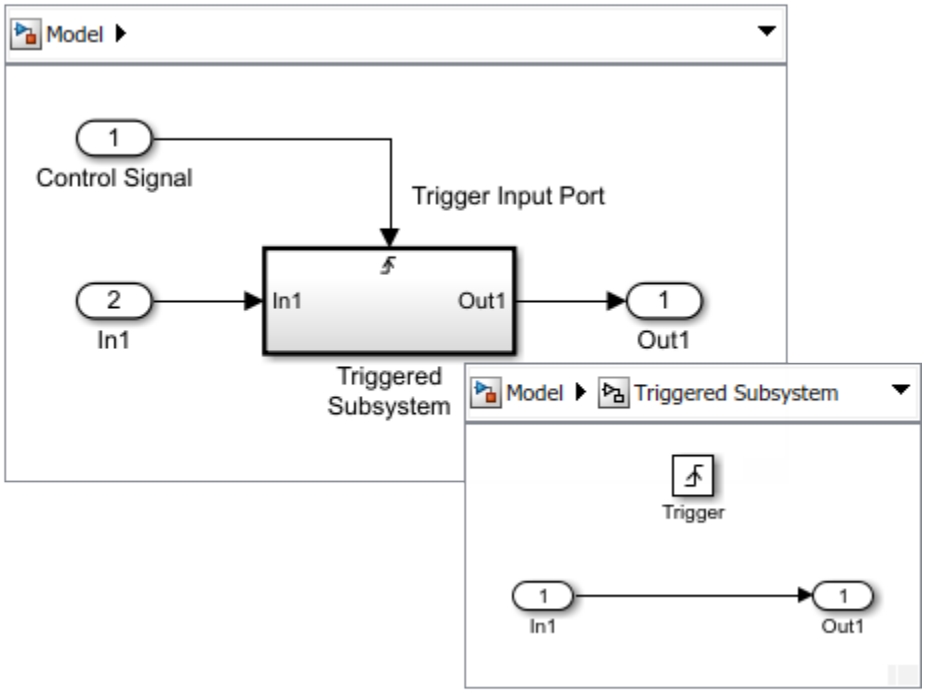
- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Initial Output Values” on page 10-37
- “Conditional Subsystem Output Values When Disabled” on page 10-54
- “Comparison of Resettable Subsystems and Enabled Subsystems” on page 10-110

Using Triggered Subsystems

In this section...

A triggered subsystem is a conditionally executed atomic subsystem that runs each time the control signal (trigger signal):

- Either rises from a negative value to a positive value or zero, or rises from a zero value to a positive value.
- Either falls from a positive value to a negative value or zero, or falls from a zero value to a negative value.
- Rises or falls through or to a zero value.



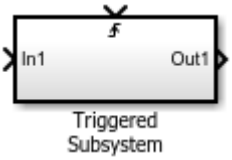
Unlike an Enabled Subsystem block, a Triggered Subsystem block always holds its outputs at the last value between triggers. Also, triggered subsystems cannot reset block states when executed; the states of any discrete block are held between triggers.

Create a Triggered Subsystem

To create a triggered subsystem:

1. Add a Triggered Subsystem block to your model.

   - Copy a block from the Simulink Ports & Subsystems library to your model.
   - Click the model diagram, start typing `trigger`, and then select Triggered Subsystem.

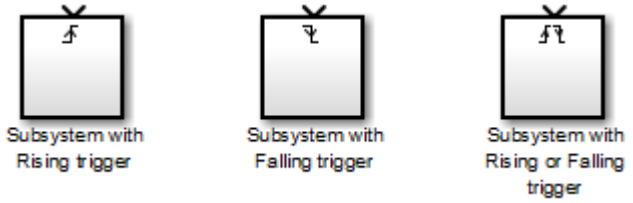


2. Set initial and disabled values for the Outport blocks. See “Conditional Subsystem Initial Output Values” on page 10-37 and “Conditional Subsystem Output Values When Disabled” on page 10-54.
3. Set how the control signal triggers execution.

   Open the subsystem block, and then open the parameter dialog box for the Trigger port block. From the **Trigger type** drop-down list, select:

   - `rising` — Trigger execution of the subsystem when the control signal rises from a negative or zero value to a positive value.
   - `falling` — Trigger execution of the subsystem when the control signal falls from a positive or zero value to a negative value.
   - `either` — Trigger execution of the subsystem with either a rising or falling control signal.

   Different symbols appear on the Trigger and Subsystem blocks to indicate rising and falling triggers.



4. Output the enable control signal from the Trigger port block. Open the Trigger port block. Select the **Show output port** check box to pass the control signal into the triggered subsystem.

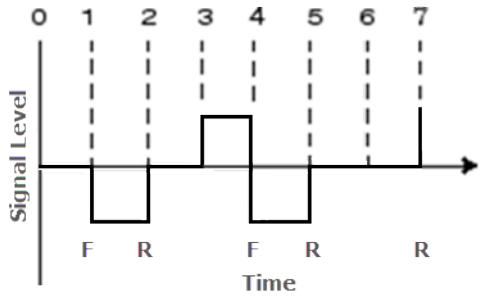
   You can use this signal with an algorithm that depends on the value of the control signal.
5. From the **Output data type** drop-down list, select `auto`, `int8`, or `double`.

   The `auto` option causes the data type of the output signal to be the data type (either `int8` or `double`) of the block port connected to the signal.

Triggering with Discrete Time Systems

For a discrete time system, the trigger control signal must remain at zero for more than one time step. This triggering strategy eliminates false triggers caused by control signal sampling.

In the following timing diagram for a discrete system, a rising trigger signal (R) does not occur at time step 3. The trigger signal remains at zero for only one time step before the signal increases from zero.



Triggered Model Versus a Triggered Subsystem

You can place a Trigger port block in a Model block (referenced model) to simplify your model design instead of using one of these blocks:

- A Triggered Subsystem block in a Model block.
- A Model block in a Triggered Subsystem block.

For information about using Trigger port blocks in referenced models, see “Modify Referenced Models for Conditional Execution” on page 8-24.

To convert a subsystem to use model referencing, see “Convert Subsystems to Referenced Models” on page 8-18.

Blocks in a Triggered Subsystem

All blocks in a triggered subsystem must have **Sample time** set to inherited (`-1`) or constant (`inf`). This requirement allows the blocks in a triggered subsystem to run only when the triggered subsystem itself runs. This requirement also means that a triggered subsystem cannot contain continuous blocks, such as an Integrator block.

Model Examples

- “Simulink Subsystem Semantics”
- “Triggered Subsystems”
- “Modeling Engine Timing Using Triggered Subsystems”

See Also

Blocks
Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Related Examples

- “Using Enabled Subsystems” on page 10-10
- “Using Enabled and Triggered Subsystems” on page 10-21
- “Using Function-Call Subsystems” on page 10-34

More About

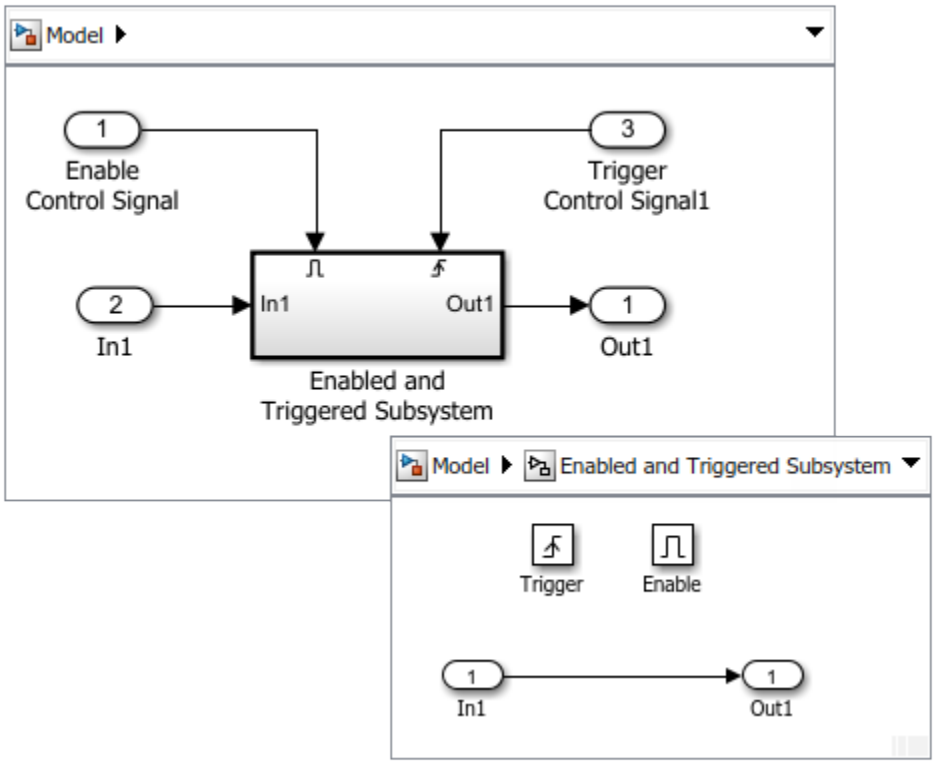
- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Initial Output Values” on page 10-37
- “Conditional Subsystem Output Values When Disabled” on page 10-54

Using Enabled and Triggered Subsystems

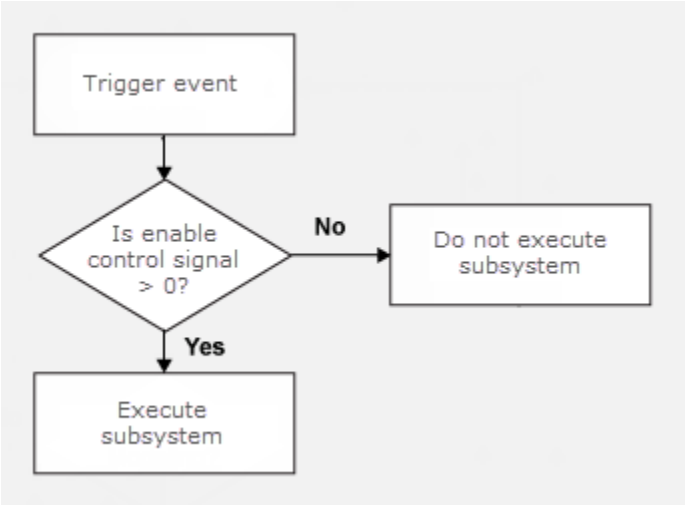
In this section...

An Enabled and Triggered Subsystem is a conditionally executed subsystem that runs once at each simulation time step when both these conditions apply:

- Enabled control signal has a positive value.
- Trigger control signal rises or falls through zero.



An Enabled and Triggered Subsystem block contains both an Enable port block and a Trigger port block. When a trigger signal rises or falls through zero, the enable input port is checked to evaluate the enable control signal. If its value is greater than zero, the subsystem is executed. When both inputs are vectors, the subsystem executes if at least one element of each vector is nonzero.

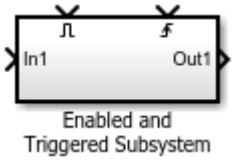


Creating an Enabled and Triggered Subsystem

To create an enabled and triggered subsystem:

1. Add an Enabled and Triggered Subsystem block to your model.

   - Copy a block from the Simulink Ports & Subsystems library to your model.
   - Click the model diagram, start typing `enabled`, and then select Enabled and Triggered Subsystem.

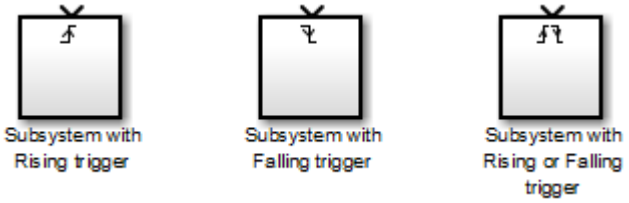
   

2. Set initial and disabled values for the Outport blocks. See “Conditional Subsystem Initial Output Values” on page 10-37 and “Conditional Subsystem Output Values When Disabled” on page 10-54.
3. Set how the control signal triggers execution.

   Open the subsystem block, and then open the block parameters dialog box for the Trigger port block. From the **Trigger type** drop-down list, select:

   - `rising` — Trigger execution of the subsystem when the control signal rises from a negative or zero value to a positive value.
   - `falling` — Trigger execution of the subsystem when the control signal falls from a positive or zero value to a negative value.
   - `either` — Trigger execution of the subsystem with either a rising or falling control signal.

   Different symbols appear on the Trigger and Subsystem blocks to indicate rising and falling triggers.



4 Specify how subsystem states are handled when enabled.

Open the subsystem block, and then open the Enable port block. From the **States when enabling** drop-down list, select:

- `held` — States maintain their most recent values.
- `reset` — States revert to their initial conditions if the subsystem is disabled for at least one time step.

In simplified initialization mode, the subsystem elapsed time is always reset during the first execution after becoming enabled. This reset happens regardless of whether the subsystem is configured to reset when enabled. See “Underspecified initialization detection”.

For nested subsystems whose Enable blocks have different parameter settings, the settings for the child subsystem override the settings inherited from the parent subsystem.

Blocks in an Enabled and Triggered Subsystem

All blocks in an enabled and triggered subsystem must have **Sample time** set to inherited (`-1` or constant (`inf`). This requirement allows the blocks in a triggered subsystem to run only when the triggered subsystem itself runs. This requirement also means that a triggered subsystem cannot contain continuous blocks, such as an Integrator block.

Model Examples

- “Simulink Subsystem Semantics”
- “Enabled Subsystems”

See Also

Blocks
Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Related Examples

- “Using Enabled Subsystems” on page 10-10
- “Using Triggered Subsystems” on page 10-17
- “Using Function-Call Subsystems” on page 10-34

More About

- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Initial Output Values” on page 10-37
- “Conditional Subsystem Output Values When Disabled” on page 10-54

Select Subsystem Execution

In this section...

A logically executed subsystem block runs one or more times at the current time step when enabled by a control block. A control block implements control logic similar to that expressed by a programming language statement (e.g., if-then, switch, while, for).

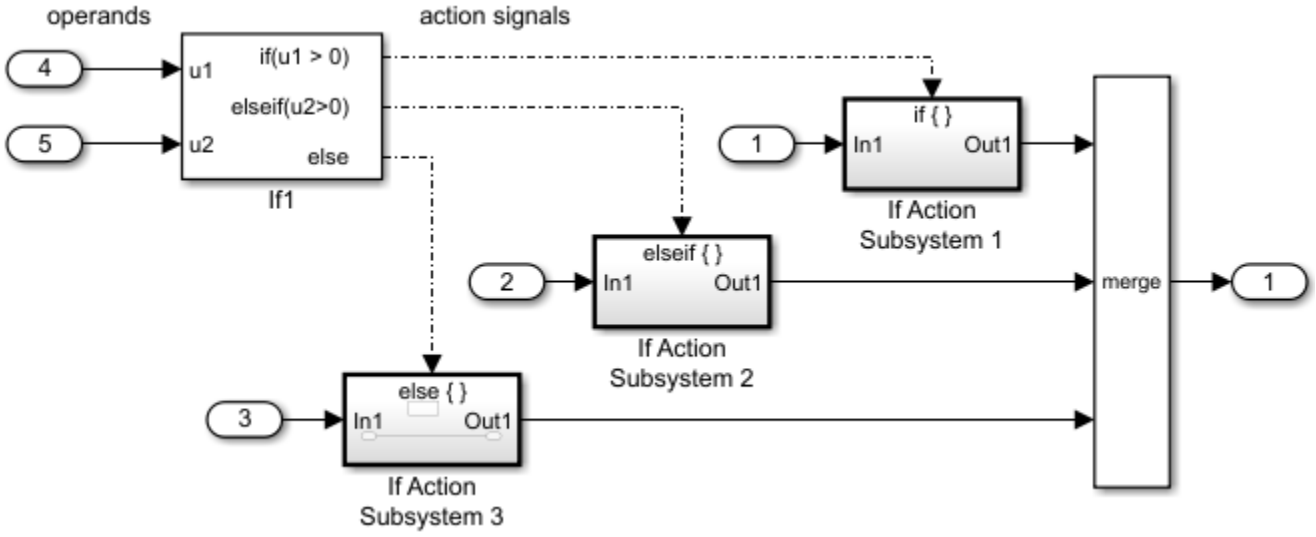
Selector subsystems are one type of logically executed subsystem that execute once during a time step in response to an action signal from a control block located external to the subsystem. Simulink supports two selector subsystem structures, if-else and switch-case.

Models with If-Else Structures

The If Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is enabled by an If block.

An external If block controls execution. The If block evaluates a logical expression and then, depending on the result of the evaluation, outputs an action signal to a If Action Subsystem block.

Consider the following model. To open model, see `ex_if_block`.



In this model, the inputs to the If block provide the operand values for the logical expressions represented as output ports. Each output port is attached to an If Action Subsystem block. The expressions in the If block are evaluated top down starting with the `if` expression. When an expression evaluates to true, its corresponding If Action Subsystem is executed and the remaining expressions are not evaluated.

The *if-else* structure in the model can be represented with the following pseudo code.

```
IF u1 > 0 THEN
    Subsystem 1
ELSEIF u2 > 0
    Subsystem 2
ELSE
    Subsystem 3
END IF
```

Create Model with If-Else Structure

To create the example model, use the following procedure.

1. Place an If block in the Simulink Editor. Double-click the block to open the block parameters dialog box.
2. In the **Number of inputs** box, enter `2`.

   Two input ports are added to the block. The inputs ports are for signals containing operand values, not necessary the number of operands. An input signal can be a vector. For example, you could specify the fifth element of a vector u in an expression as `u(5) > 0`.
3. In the **If expression** text box, enter `u1 > 0`.

   An output port is added to the block with the label `if(u1 > 0)`. This port is the only required output for an If block.
4. In the **Elseif expressions** text box, enter `u2 > 0`.

   You can enter multiple elseif expressions with a comma separating the expressions. Each expression adds an output port to the If block with a label of the form `elseif(expression)`.
5. Check the **Show else condition** check box.

   An output port is added to the block with the label `else`.
6. Add three If Action Subsystem blocks.

   These blocks are Subsystem blocks with an Action Port block. When you place an Action Port block inside a subsystem, an input port named Action is added to the block.
7. Connect each output port from the If block to the action port of an If Action Subsystem block.

   When you make the connection, the icon for the If Action Subsystem block is renamed to the type of expression that attached to it.
8. In each If Action Subsystem block, enter the Simulink blocks to be executed for the condition it handles.
9. Connect outputs from If Action Subsystem blocks to a Merge block.
10. Run a simulation.

    The action signal lines between the If block and the If Action Subsystem blocks change from a solid to a dashed line.

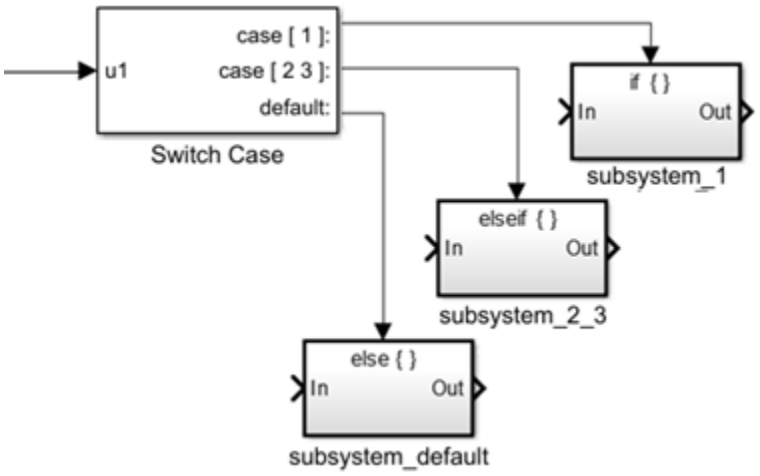
Note All blocks in an If Action Subsystem block driven by an If block must run at the same rate as the driving block.

Models with Switch Case Structure

The Switch Case Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is enabled by a Switch Case block.

An external Switch Case block controls execution. The Switch Case block evaluates a case index and then, depending on the selected case, outputs an action signal to a Switch Case Action Subsystem block.

Consider the following model with a *switch* structure. To open model, see `ex_switch_case_block`.



In this model, the input to the Switch Case block provides the index value for selecting a case represented as output ports. Each output port is attached to an If Action Subsystem block. When a case is selected, its corresponding If Action Subsystem is executed.

The *switch* structure in the model can be represented with the following pseudo code.

```
CASE u1
    u1 = 1:
        subsystem_1
        break
    u1 = 2 or 3:
        subsystem_2_3
        break
    u1 = OTHER VALUES:
        subsystem_default
        break
END CASE
```

Create Model with Switch Case Structure

To create the example model, use the following procedure.

1. Place a Switch Case block in the Simulink Editor. Double-click the block to open the block parameters dialog box.
2. In the **Case conditions** box, enter `{1, [2,3]}`.

Two cases are defined. The first case when the input value is `1`, and the second case when the input value is `2` or `3`. Cases can be single or multivalued and appear as output ports on the Switch Case block. Non-integer input values are truncated to integers.

3. Select the **Show default case** check box.

   An output port labeled `default:`is added to the block. This port sends an action signal if no other cases are selected.

4. Add three Switch Case Action Subsystem blocks.

   These blocks are Subsystem blocks with an Action Port block. When you place an Action Port block inside a subsystem, an input port named Action is added to the block.

5. Connect each output port from the Switch Case block to the action port of an Switch Case Action Subsystem block.

   When you make the connection, the icon for the Switch Case Action Subsystem block is renamed to the type of expression attached to it.

6. In each Switch Case Action Subsystem block, enter the Simulink blocks to be executed for the case it handles.

7. Run a simulation.

   The action signal lines between the Switch Case block and the Switch Case Action Subsystem blocks change from a solid to a dashed line.

Note After the subsystem for a particular case executes, an implied break terminates the execution of the Switch Case block. Simulink Switch Case blocks do not exhibit the fall -through behavior of C `switch` statements.

Model Examples

- “Simulink Subsystem Semantics”
- “Modeling Clutch Lock-Up Using If Blocks”
- “If-Then-Else Blocks”

See Also

Blocks

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

Iterate Subsystem Execution

In this section...

A logically executed subsystem block runs one or more times at the current time step when enabled by a control block. A control block implements control logic similar to that expressed by a programming language statement (e.g., if-then, switch, while, for).

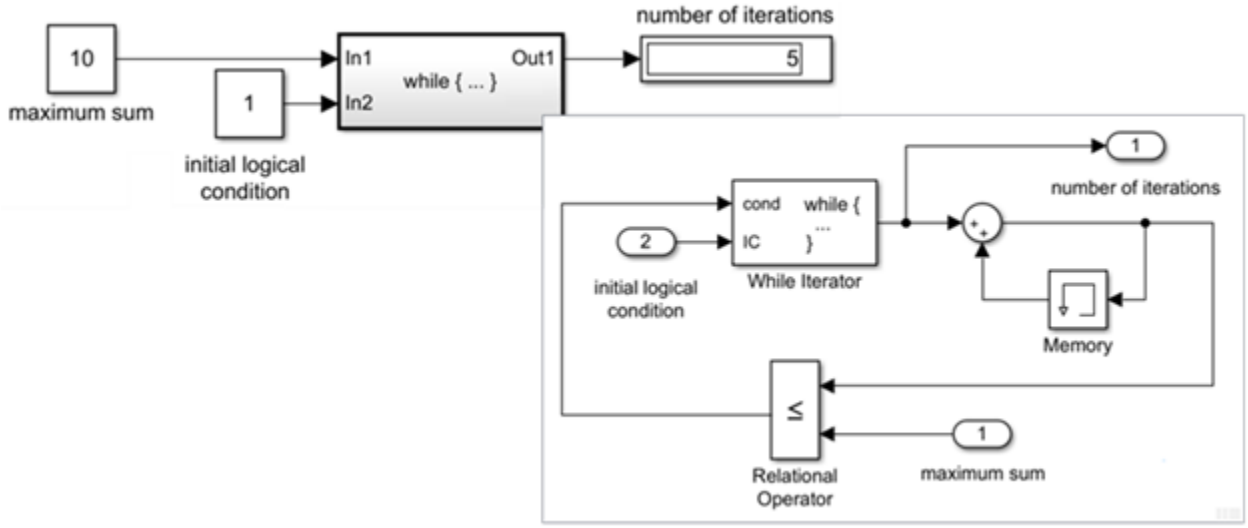
Iterator subsystems are one type of logically executed subsystem that execute one or more times during a time step in response to a control block internal to the subsystem block.

Note The While Iterator Subsystem and For Iterator Subsystem blocks must not contain blocks with continuous states (for example, blocks from the Continuous block library). The sample times of all blocks within the subsystem must be either inherited (`-1`) or constant (`inf`).

Models with While Structures

The While Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step while a logical (Boolean) expression is true.

Consider the following model. To open model, see `ex_while_iterator_block`.



An input of `1` (`true`) to the While Iterator block activates the subsystem. At each time step, the current iterative number is added to a running total until a maximum sum is reached.

The *while* structure in the model can be represented with the following pseudo code.

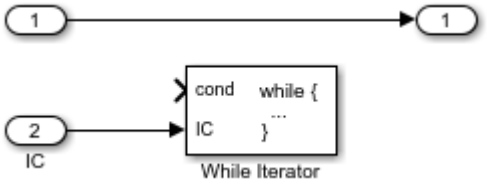
```
maximum_sum = 10;
sum = 0;
```

```
iteration_number = 0

condition = (maximum_sum > 0)
WHILE condition NOT EQUAL 0
    iteration_number = iteration_number + 1
    sum = sum + iteration_number
    IF (sum > maximum_sum OR iteration_number > maximum_iterations) THEN
        condition = 0
END WHILE
```

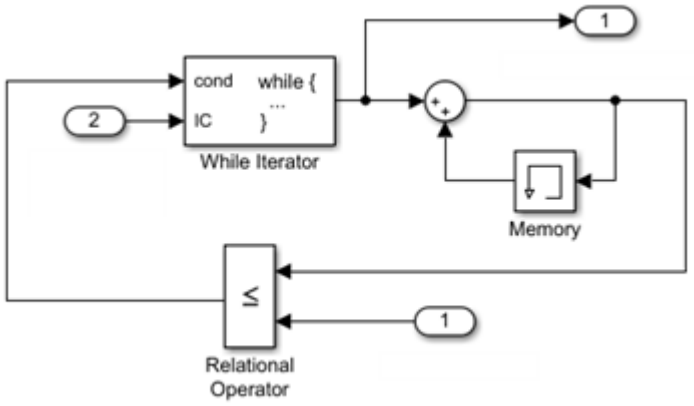
Create Model with While Structure

To create the example model, use the following procedure.

1. Place a While Iterator Subsystem block in the Simulink Editor. Double-click the subsystem block to display its contents.



2. Double-click the While Iterator block to open its block parameters dialog box. Set the **Maximum number of iterations** to `20` and **States when starting** to `reset`. Select the **Show iteration number** port check box.
3. Add Memory, Relational Operator, and Sum blocks. Connect blocks as shown. For the Memory block, select the **Inherit sample time** check box.



   The iteration number from the output of the While Iterator block is added to its previous value until the sum is greater or equal to the maximum sum from Inport block 1.
4. Navigate to the top level of the model.
5. Connect a Constant block to input port 2. This block provides the *initial logical condition* value for the While Iterator block. Set the **Constant value** to any non-zero number.

   The While Iterator block requires an initial logical condition (input port labeled `IC`) for its first iteration. This signal enables the While Iterator Subsystem block and must originate from outside the subsystem. If this value is nonzero, the first iteration takes place.

6. Connect a second Constant block to input port 1. This block provides a maximum value for the iterative algorithm. The algorithm adds successive integers until a maximum value is reached.
7. Connect a Display block to output port 1. This block shows the number of iterations from the While Integrator block output port.
8. Run a simulation.

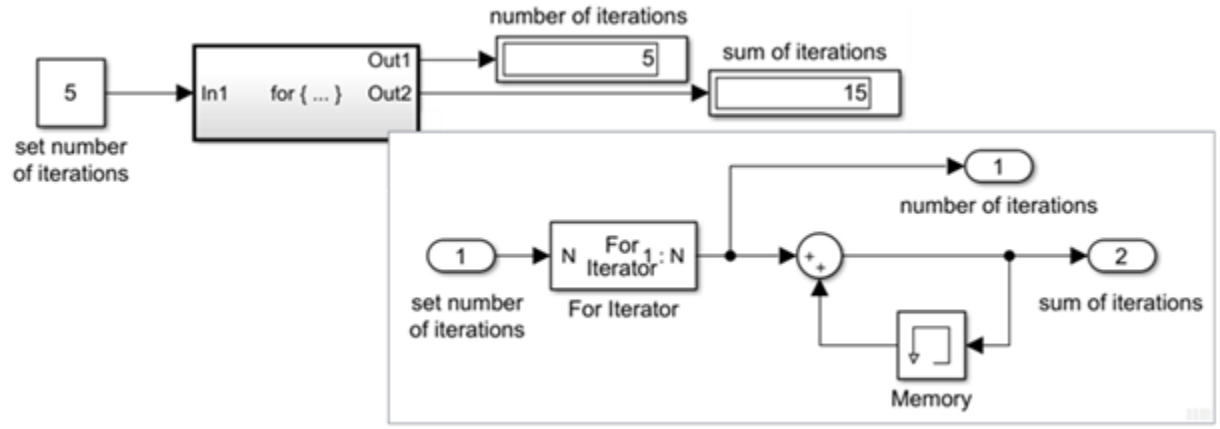
   During a single time step, the first four iteration numbers are added for a total sum (`10`). With the fifth iteration, the sum (`15`) is greater than the maximum sum (`10`), the iterations stop, and the block waits for the next time step.

Note Simulation time does not advance during iterative executions of a While Iteration Subsystem block. Nevertheless, blocks in the subsystem treat each iteration as a time step. As a result, in a While Iterator Subsystem block, the output of a block with states (that is, a block whose output depends on its previous input), reflects the value of its input at the previous iteration of the `while` loop. The output does *not* reflect the block input at the previous simulation time step. For example, a Unit Delay block in a While subsystem outputs the value of its input at the previous iteration of the `while` loop, not the value at the previous simulation time step.

Model with For Structures

The For Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step for a specified number of iterations.

Consider the following model. To open model, see `ex_for_iterator_block`.



The input to the For Iterator block specifies the number of iterations. At each time step, the current iterative number is added to a running total for `5` iterations.

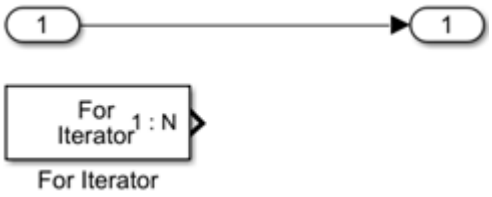
The *for* structure in the model can be represented with the following pseudo code.

```
number_of_iterations = 5
sum = 0;
iteration_number = 0

FOR iteration_number = 0 TO number_of_iterations
    iteration_number = iteration_number + 1
    sum = sum + iteration_number
END FOR
```

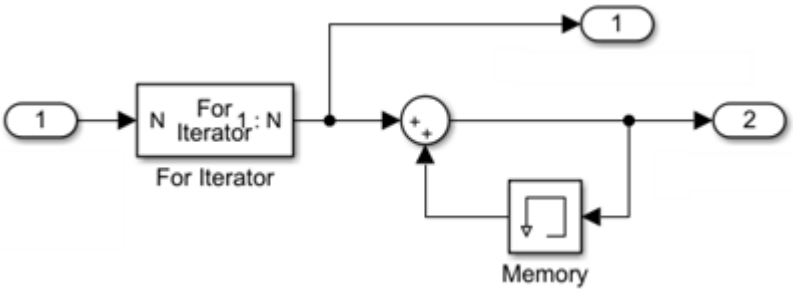
Create Model With For Structure

To create the example model, use the following procedure.

1. Place a For Iterator Subsystem block in the Simulink Editor. Double-click the subsystem block to display its contents.



2. Double-click the For Iterator block to open its block parameters dialog box. Set **States when starting** to `reset` and **Iteration limit source** to `external`.
3. Add Memory, Sum, and Outport blocks. Connect blocks as shown. For the Memory block, select the **Inherit sample time** check box.

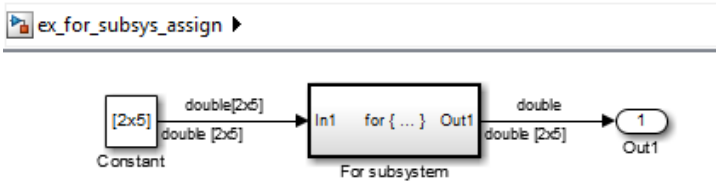


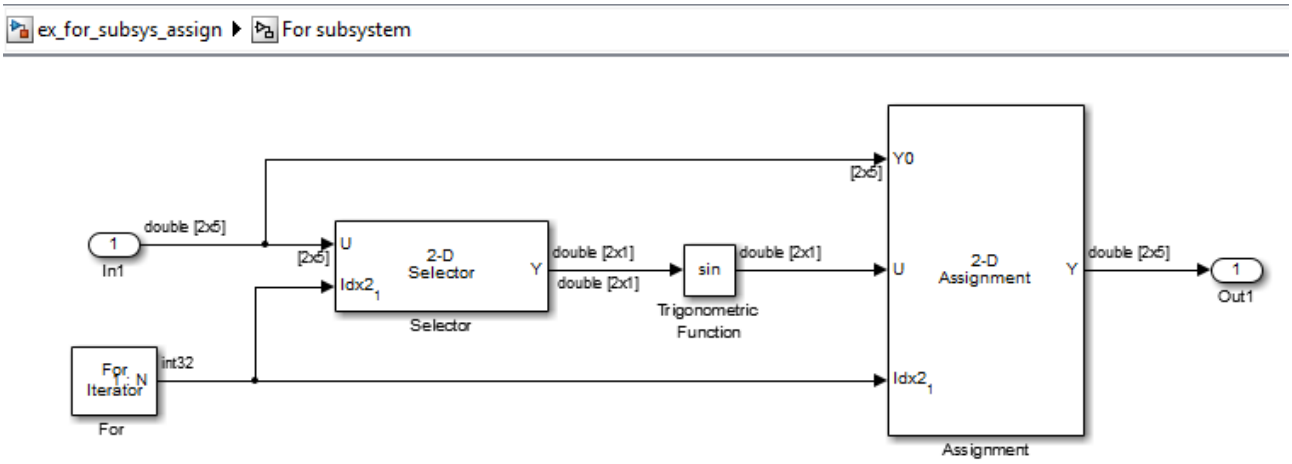
   The iteration number from the output of the For Iterator block is added to its previous value for the specified number of iterations from Inport block 1.
4. Navigate to the top level of the model.
5. Connect a Constant block to input port 1. This block provides the number of iterations for the For Iterator block. Set the **Constant value** to 5.
6. Connect Display blocks to output ports 1 and 2. These blocks shows the number of iterations from the For Integrator block output port and the sum from the Memory block.
7. Run a simulation.

   During each time step, the first five iteration numbers are added for a total sum (15).

Using Assignment Blocks

The For Iterator block works well with the Assignment block to reassign values in a vector or matrix. The following example shows the use of a For Iterator block. Note the matrix dimensions in the data being passed.





The example outputs the sine value of an input 2-by-5 matrix (2 rows, 5 columns) using a For subsystem containing an Assignment block. The process is as follows.

- A 2-by-5 matrix is input to the Selector block and the Assignment block.
- The Selector block strips off a 2-by-1 matrix from the input matrix at the column value indicated by the current iteration value of the For Iterator block.
- The sine of the 2-by-1 matrix is taken.
- The sine value 2-by-1 matrix is passed to an Assignment block.
- The Assignment block, which takes the original 2-by-5 matrix as one of its inputs, assigns the 2-by-1 matrix back into the original matrix at the column location indicated by the iteration value.

  The rows specified for reassignment in the parameter dialog box for the Assignment block in the example are [1,2]. Because there are only two rows in the original matrix, you could also have specified -1 for the rows (that is, all rows).

Note The Trigonometric Function block is already capable of taking the sine of a matrix. The example uses the Trigonometric Function block only as an example for changing each element of a matrix with the collaboration of an Assignment block and a For Iterator block.

Model Examples

- “Simulink Subsystem Semantics”

See Also

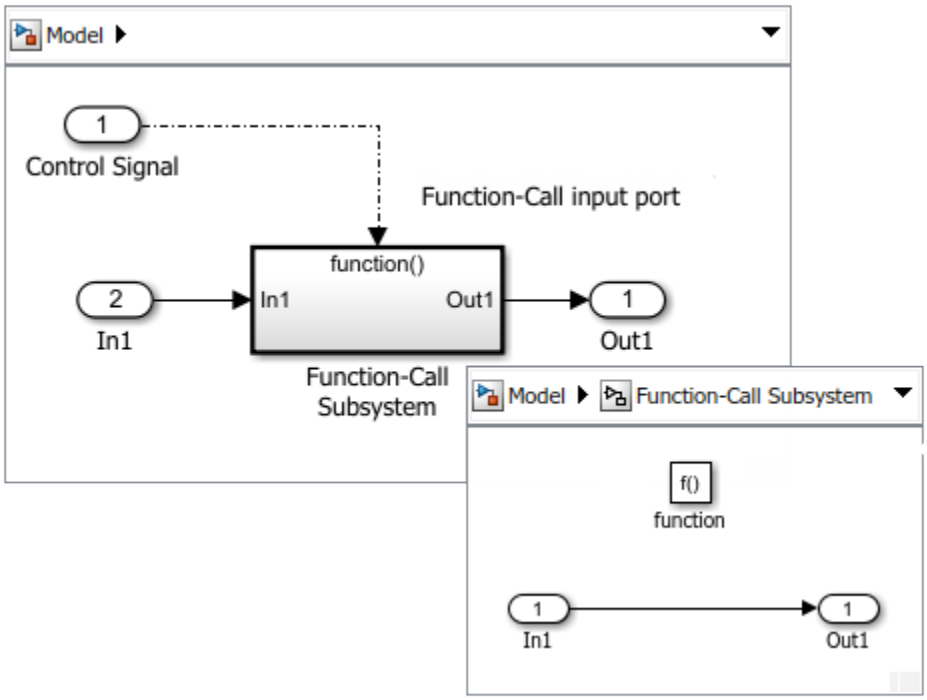
Blocks

For Iterator | For Iterator Subsystem | Subsystem | While Iterator | While Iterator Subsystem

Using Function-Call Subsystems

In this section...

A Function-Call Subsystem block is a conditionally executed subsystem that runs each time the control port receives a function-call event. A Stateflow chart, Function-Call Generator block, S-Function block, or Hit Crossing block can provide function-call events.



A function-call subsystem is analogous to a function in a procedural programming language. Invoking a function-call subsystem executes the output methods of the blocks within the subsystem in execution order. For an explanation of the Function-Call Subsystem block parameters, see Subsystem.

Creating a Function-Call Subsystem

To create a function-call subsystem:

1. Add a Function-Call Subsystem block to your model.
2. Open the subsystem block. Add a block diagram defining the algorithm that is executed when the subsystem receives a function-call event.
3. Set initial and disabled values for the Outport blocks. See “Conditional Subsystem Initial Output Values” on page 10-37 and “Conditional Subsystem Output Values When Disabled” on page 10-54.
4. Set how subsystem states are handled when the subsystem is executed:

Open the subsystem block, then open the block parameters dialog box for the Trigger block. From the **States when enabling** drop-down list, select an option:

- `held` — States maintain their most recent values.
- `reset` — States set to their initial conditions.
- `inherit` — Use the held or reset setting from the parent subsystem initiating the function-call.

For nested subsystems whose Function-Call Subsystem blocks have different parameter settings, the settings for the child subsystem override the settings inherited from the parent subsystem. See Trigger.

5 Attach a function-call initiator to the function-call input port.

If you attach an Inport block, open the block, select the Signal Attributes tab, then select the **Output function call** check box.

Sample Time Propagation in a Function-Call Subsystem

Configure a Function-Call Subsystem block by setting the **Sample time type** of its Trigger Port block to `triggered` or `periodic`.

- A triggered (aperiodic) function-call subsystem can execute zero, one, or multiple times during a time step.

  If a function-call subsystem is executed by a root-level function-call Inport block with a discrete sample time, multiple function-calls during a time step are not allowed. To allow multiple function-calls, set **Sample time** to `-1` (inherited), and set the sample time for all blocks in the function-call subsystem to `-1` (inherited).
- A periodic function-call subsystem executes once during a time step and must receive periodic function-calls. If the function-calls are aperiodic, the simulation stops and an error message is displayed. Set the sample time for all blocks in the function-call subsystem to `-1` (inherited).

Note During range checking, the minimum and maximum parameter settings are back-propagated to the actual source port of the function-call subsystem, even when the function-call subsystem is not enabled.

To prevent this back propagation:

1 Add a Signal Conversion block and a Signal Specification block after the source port.

2 Set the **Output** of the Signal Conversion block to `Signal copy`.

3 Specify the minimum and maximum values for the Signal Specification block instead of specifying them on the source port.

Model Examples

- “Simulink Subsystem Semantics”

See Also

Blocks
Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem | Subsystem | Trigger

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Generate Component Source Code for Export to External Code Base” (Embedded Coder)
- “Context-dependent inputs”
- “Check usage of function-call connections”
- “Check for potentially delayed function-call subsystem return values”

More About

- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Initial Output Values” on page 10-37
- “Conditional Subsystem Output Values When Disabled” on page 10-54

Conditional Subsystem Initial Output Values

In this section...

To initialize the output values for a conditional subsystem, initialize Outport blocks within the subsystem by using one of these methods:

- Inherit initial output values from input signals connected to the Outport blocks.
- Specify initial output values using Outport block parameters.

Note If the conditional subsystem is driving a Merge block in the same model, you do not need to specify an initial condition for the subsystem Outport block.

Inherit Initial Output Values from Input Signals

Simulink attempts to use input signals connected to conditional subsystem Outport blocks to initialize output values. This behavior is seen after setting the Outport block parameter **Source of initial output value** to `Input signal`. The exact same behavior is seen when **Source of initial output value** is set to `Dialog` and **Initial output** is set to `[]`.

Valid input signals for setting initial output values are:

- Output ports from another conditionally executed subsystem
- Output ports from a Model block with a Trigger block set to function-call
- Merge blocks
- Constant blocks
- IC (initial condition) blocks
- Simulink signal object attached to the signal line connected to the Outport block. If the `InitialValue` parameter is defined, Simulink uses this value.
- Stateflow chart

If the input signal is from a block that is not listed here, the Outport block uses the default initial value of the output data type.

Note If you are using classic initialization mode, selecting `Input signal` causes an error. To inherit the initial output value from an input signal, set the **Source of initial output value** parameter to `Dialog`, set **Output when disabled** to `held`, and set **Initial output** to the empty matrix `[]`.

Specify Initial Output Values Using Dialog Parameters

Explicitly set the initial output values in cases where you want to:

- Test the behavior of a model with various initial values.
- Set initial values to steady state and reduce simulation time.

- Eliminate having to trace input signals to determine the initial output values.

To specify initial output values

1. Open the dialog box for an Outport block within a conditional subsystem.
2. From the **Source of initial output value** drop-down list, select `Dialog`.
3. In the **Initial output** box, enter the initial value.

See Also

More About

- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Output Values When Disabled” on page 10-54

Rate-Based Models Overview

You can control the execution of model components (Subsystem and Model blocks) by using export-function models or rate-based models. Benefits of scheduling components are:

- Full control over scheduling of model components rather than letting Simulink implicitly schedule the components.
- No need to deal with data dependency issues between components. That is, there are only data transfers.

See Also

More About

- “Sorting Rules for Explicitly Scheduled Model Components” on page 10-47

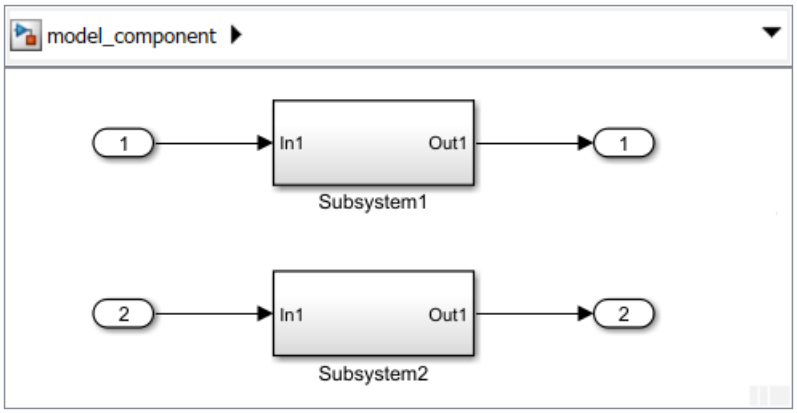
Create A Rate-Based Model

A rate-based model is a model with explicitly scheduled subsystems. You can schedule a subsystem with a periodic execution rate by specifying the **Sample time** parameter for an Inport block connected to the Subsystem block, the Subsystem block, or a block within the Subsystem block where Sample time can be specified (for example, a Delay block).

To open a completed rate-based model, see `ex_rate_based_model`.

Note Using Continuous time blocks such as Integrator blocks are not allowed. Instead use discrete time equivalent blocks

Consider the following model with two atomic Subsystem blocks. Subsystem1 multiplies its input by 2 while Subsystem2 multiplies its input by 4.

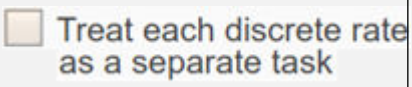
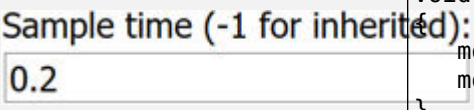
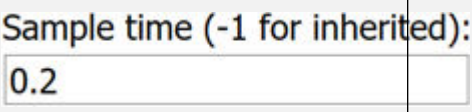


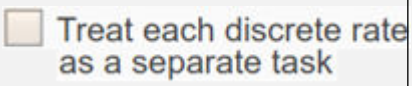
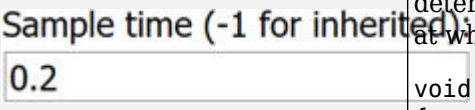
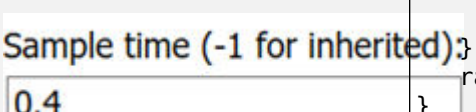
1. Open the Inport 1 dialog box. On the Signal Attributes tab, set the **Sample time** to `0.2`.
2. Open the Inport 2 dialog box. On the Signal Attributes tab, set the **Sample time** to `0.4`.
3. If a rate-based model has multiple rates, single tasking is not allowed. Select the check box for the configuration parameter **Treat each discrete rate as a separate task**.

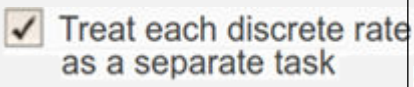
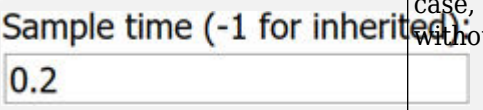
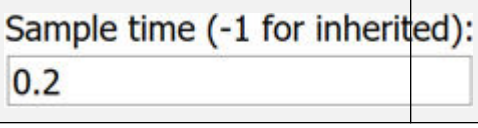
Multi-Tasking and Multi-Rate Model for Code Generation

Selecting single-tasking versus multi-tasking and single-rate versus multi-rate controls entry points in the generated code.

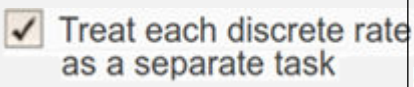
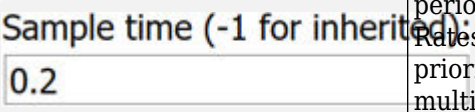
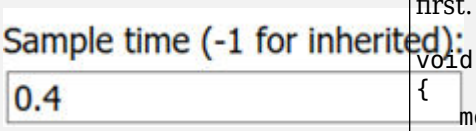
Configuration Parameter	Explicitly Scheduled Rates	Generated Code Entry Points

Single-tasking
 
 	
	Single-rate
	Subsystem1
	 
	
	Subsystem2
	 
	
		One entry-point function called periodically every 0.2 seconds
		`void model_component_step(void)`
		`{`
		`  model_component_Y.Out1 = 2.0 * model`
		`  model_component_Y.Out2 = 4.0 * model`
		`}`

Single-tasking
 
 	
	Multi-rate
	Subsystem1
	 
	
	Subsystem2
	 
	
		One entry-point function called periodically every 0.2 seconds, a schedule counter in the function determines which rates execute at which sample times.
		`void model_component_step(void)`
		`{`
		`  model_component_Y.Out1 = 2.0 * model`
		`  if (model_component_M->Timing.TaskCo`
		`  model_component_Y.Out2 = 4.0 * m`
		`}`
		`  rate_scheduler();`
		`}`

Multi-tasking  	Single-rate Subsystem1  Subsystem2  	One entry-point function, the same code as single-tasking with a single-rate. Not a use case, but the code generates without an error

Configuration Parameter	Explicitly Scheduled Rates	Generated Code Entry Points

Multi-tasking
 
 	
	Multi-rate
	Subsystem1
	 
	
	Subsystem2
	 
	
		Two entry-point functions one called periodically every 0.2 seconds and the other called periodically every 0.4 seconds. Rates are executed using a prioritized preemptive multitasking scheme. Faster rates are assigned higher priorities and thus executed first.
		`void model_component_step0(void)`
		`{`
		`  model_component_Y.Out1 = 2.0 * model`
		`}`
		
		`void model_component_step1(void)`
		`{`
		`  model_component_Y.Out2 = 4.0 * model`
		`}`

Test Rate-Based Model Simulation Using Function-Call Generators

In this section...

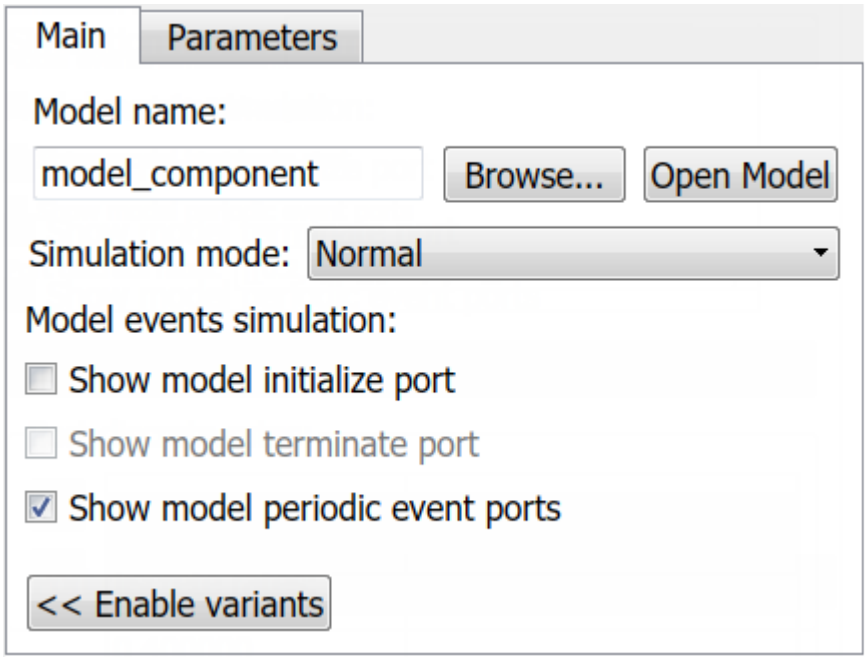
This topic describes how to reference a rate-based model in a test model, and then add periodic function-call event ports to the test model for simulation. To open an completed test harness model, see `ex_model_test_harness_with_subsystems`.

Create Test Model That References a Rate-Based Model

Testing a rate-based model includes referencing the model from a Model block in a test model, adding periodic function-call event ports to the Model block, and then connecting function-calls to the ports.

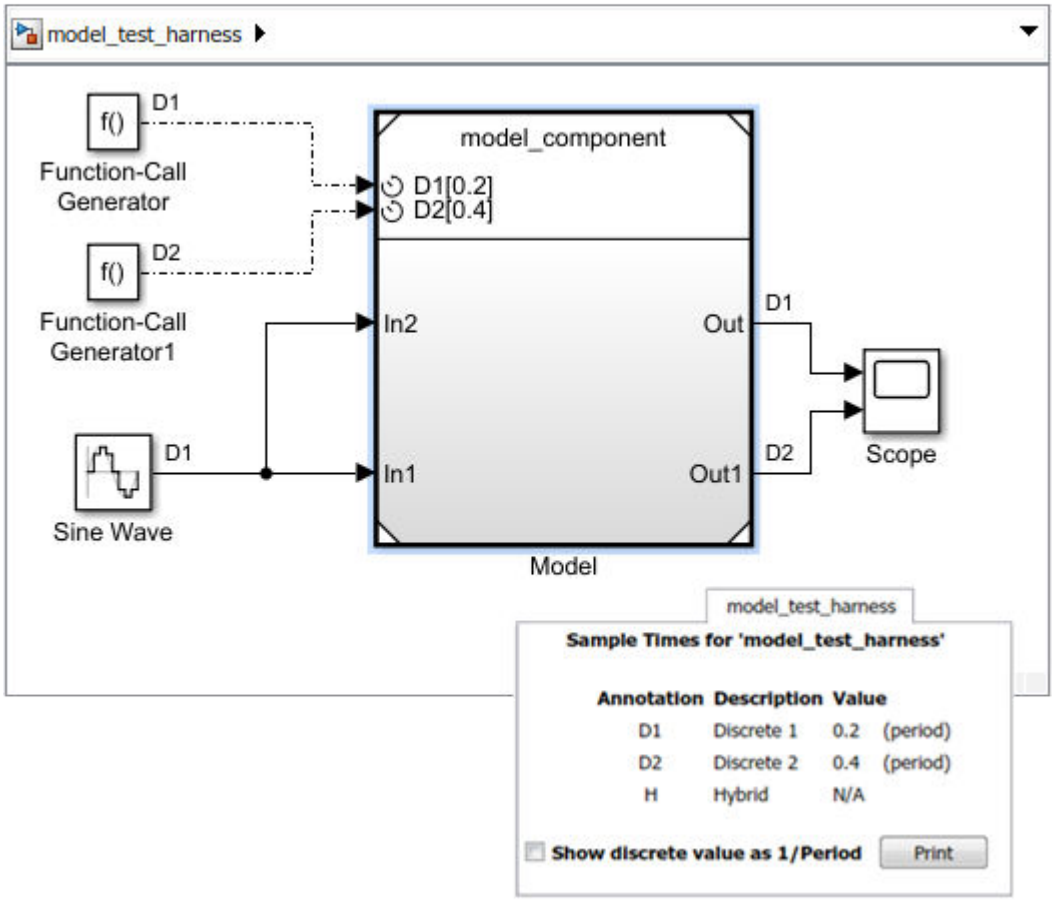
1. Create a new Simulink model.
2. Add a Model block and open the block parameters dialog box.
3. In the **Model name** box, enter the file name for the rate-based model.
4. Select the **Schedule rates** check box.

   Periodic function-call event ports are added to the Model block with the **Sample times** you specified for the Inport blocks connected to the Subsystem blocks.



5. Specify the execution rate using function-call initiators (Function-Call Generator blocks or Stateflow charts). The function-call events and scheduling of the events are located outside of the Model block referencing the rate-based model.

In this example, add Function-Call Generator blocks. Open the block dialog box for the blocks and specify **Sample time**.



Subsystems or referenced models in a rate-based model with the same sample time must have a common rate initiator. This requirement includes periodic scheduled subsystems and event-driven Function-Call Subsystem blocks with the same rate.

6 Use a fixed-step solver for simulation. Set the configuration parameters **Type** to `Fixed-step`, **Solver** to `auto`, and **Fixed-step size** to `auto`.

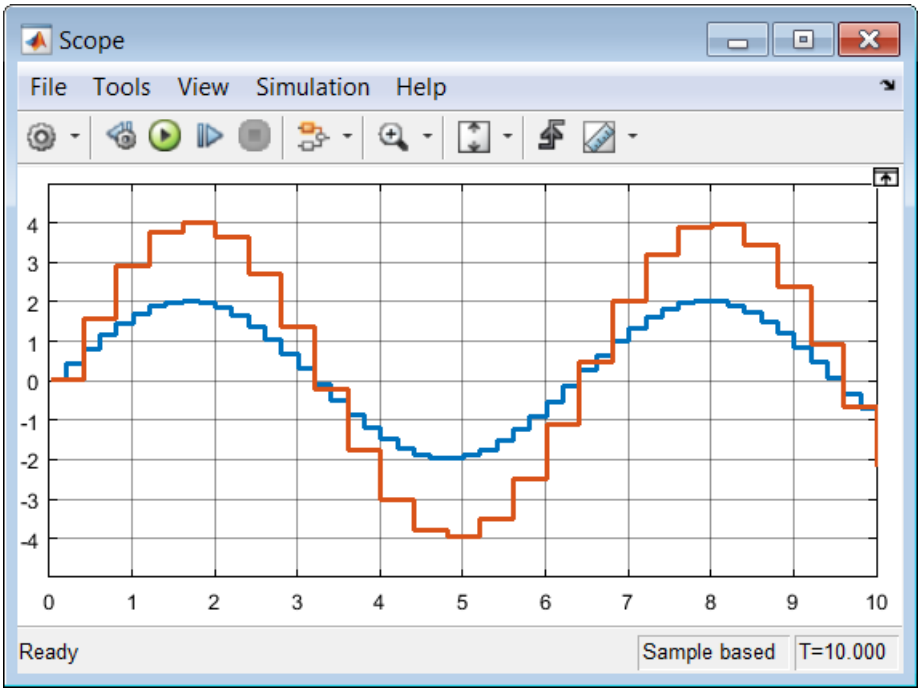
Simulate Rate-Based Model

Simulate the behavior of a rate-based model from the test model.

1 Run a simulation. Some common compile and run-time errors are caused by:

- A periodic event port that is not connected to a function-call initiator with the same specified sample time.
- A scheduled Inport block (**Sample time** parameter set to a value) in the referenced component model that does not specify one of the periodic function-call event port rates (sample times specified in the **Port discrete rates** table).

2 Observe the behavior of the component model. Open the Scope block.


Scope
File Tools View Simulation Help
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8 9 10
Ready
Sample based
T=10.000

Generate Code from Rate-Based Model

Generate code from the rate-based model, not from the model test harness. For scheduled subsystems with different discrete rates, multi-tasking is required and the resulting code has separate entry points.

1. Generate code for the component model. Display the **C Code** tab by selecting the **Apps** tab, and then in the **Apps** section, select **Embedded Coder**  . On the **C Code** tab, select **Generate Code**  .
2. Open the code generation report. On the **C Code** tab, select **Open Latest Report**.

Function: model_component_step

Prototype	**void model_component_step(int_T tid)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 0.2 seconds
Arguments	**#** **Name** **Data Type** **Description** **1** tid int_T TID0
Return value	None
Header file	model_component.h

Function: model_component_step

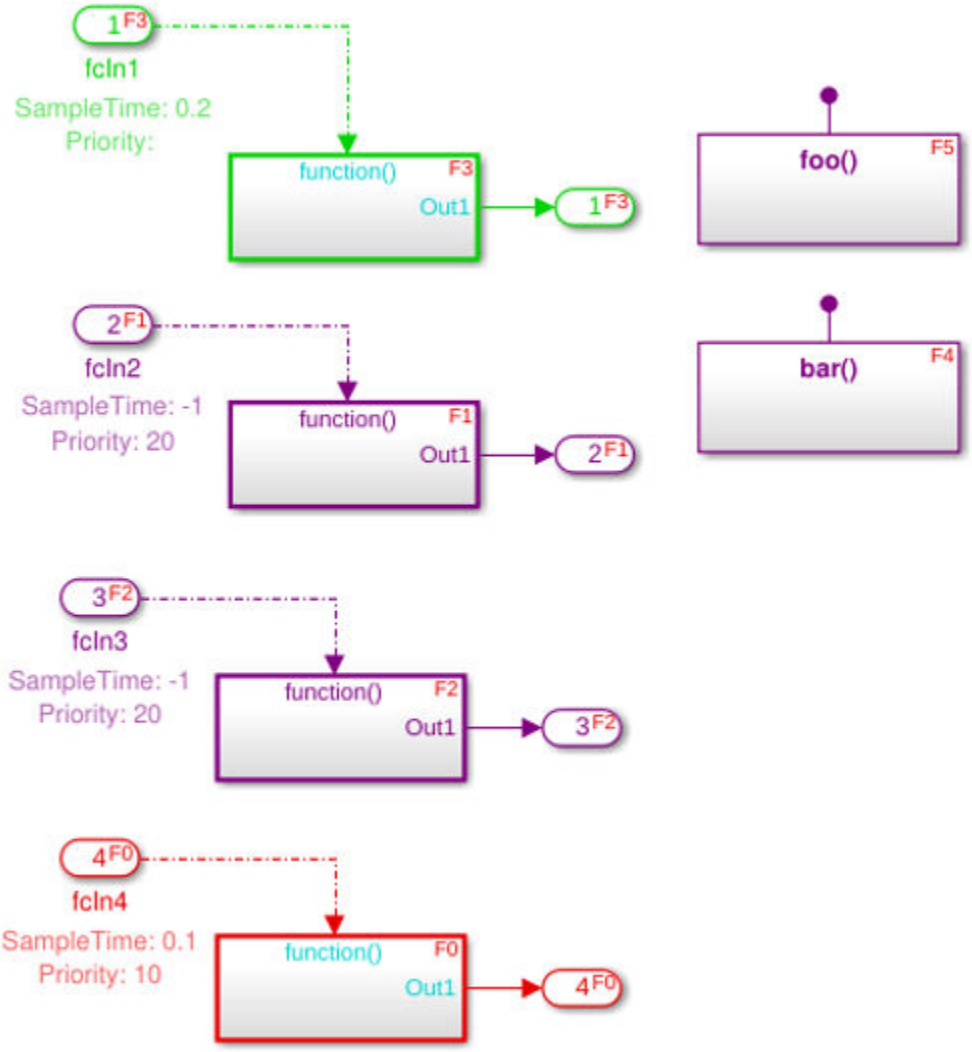
Prototype	**void model_component_step(int_T tid)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 0.4 seconds
Arguments	**#** **Name** **Data Type** **Description** **1** tid int_T TID1
Return value	None
Header file	model_component.h

Sorting Rules for Explicitly Scheduled Model Components

Simulink determines the execution order for model components (subsystems and referenced models).

Export-Function Models

Export-function models include Function-Call Subsystem blocks, function-call Model blocks, Simulink Function blocks at the root level, and S-Function blocks invoked by function-call root Inport blocks.



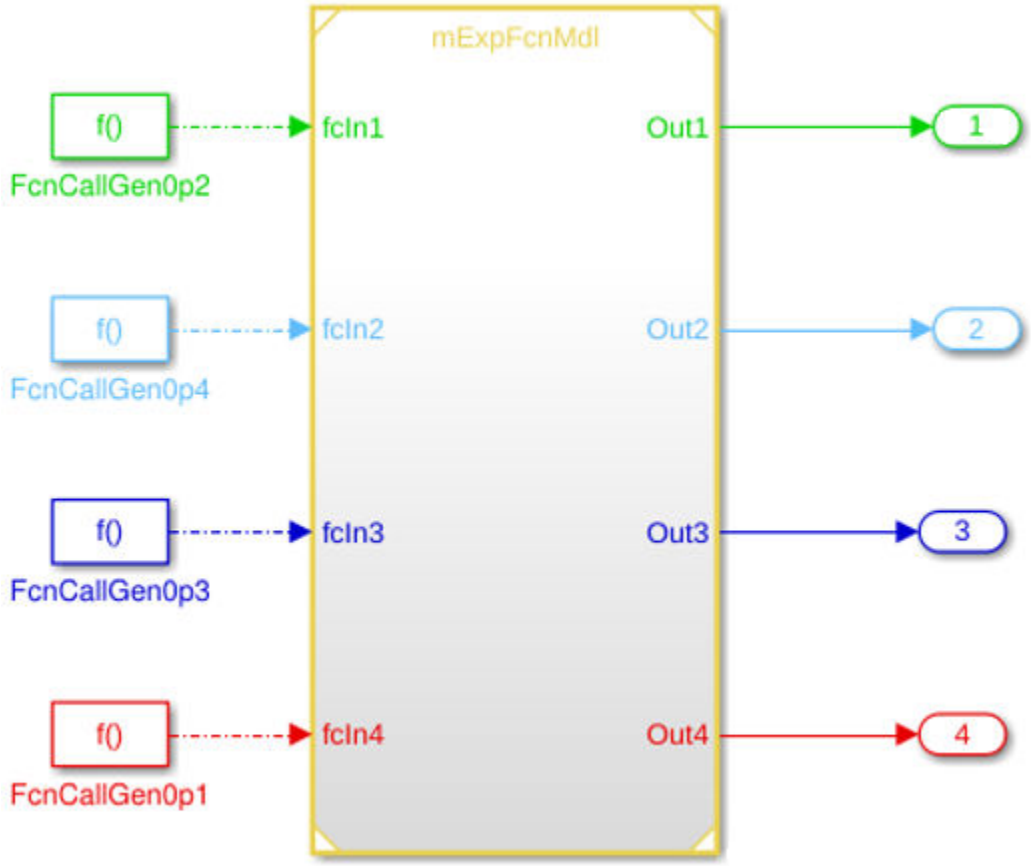
Root function-call Inport blocks are sorted with the following rules:

- First compare block priorities. The block with the highest priority (a small value) is sorted before the others.
- If block priorities are the same, compare their sample times. The block with a faster rate (a smaller sample time value) is sorted before the other.
- If the sample times are the same, compare the input port numbers. The block with the smaller port number is sorted before the other.

Root Simulink Function blocks are sorted after root function-call Inport blocks.

Test Harness for Export Function Models with Strict Scheduling

Reference the export-function model in a test harness and connect ports to Function Generator blocks.



If you select the check box for the configuration parameter **Enable strict scheduling checks for referenced models**, both compile time and run-time checks ensure initiators will invoke function-calls based on the pre-defined scheduling order. Initiators are sorted based on their sample time priorities. For this example, the scheduling order and the sample time priorities do not match. The model mHarness_ExpFcnMdl displays an error.

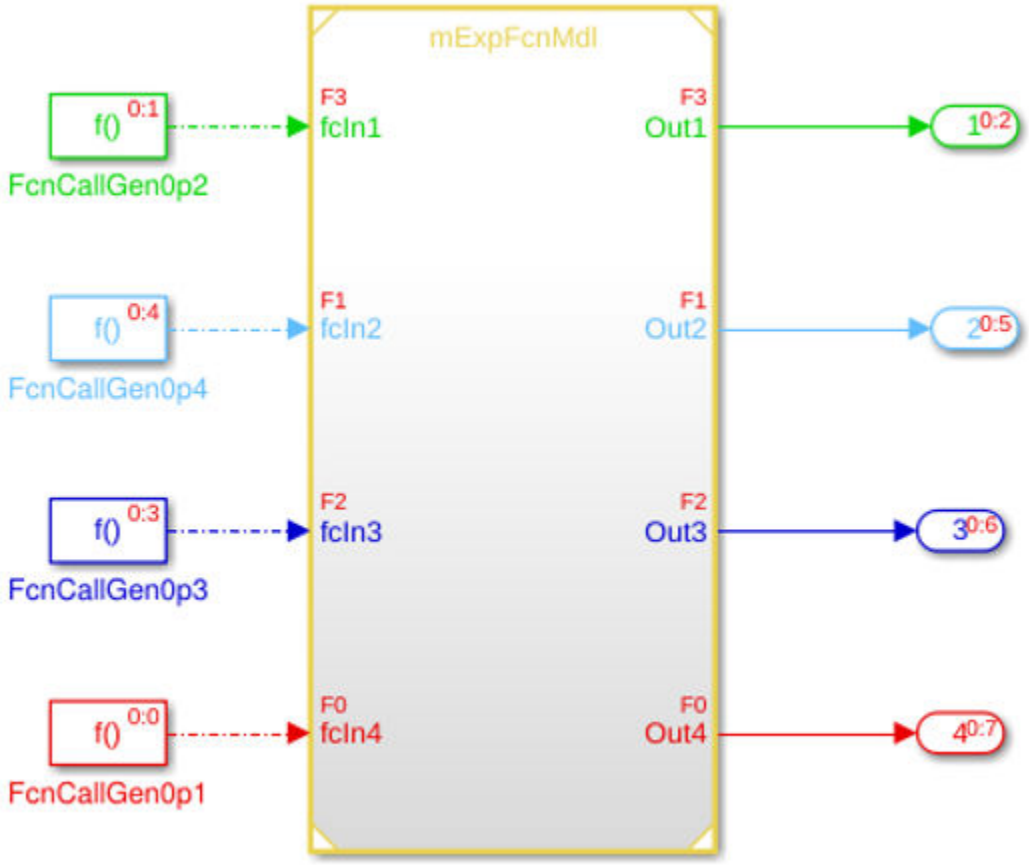
The Model block 'mHarness_ExpFcnMdl/Model' requires that function-call input port 'fcIn2' execute before function-call input port 'fcIn3'. However, this execution order cannot be honored because the task priority of the sample time for function-call input port 'fcIn2' is lower than that of function-call input port 'fcIn3'. The two function-call input ports are driven by the function-call initiator blocks, 'mHarness_ExpFcnMdl/FcnCallGen0p4' and 'mHarness_ExpFcnMdl/FcnCallGen0p3', respectively. Consider using a function-call initiator with faster sample time (higher priority) or higher asynchronous task priority for function-call input port 'fcIn2'.

Suggested Actions

- To disable this error message, clear the parameter 'Enable strict scheduling checks for referenced models' in the Model Referencing page of the Configuration Parameters dialog. Fix
- Alternatively, consider using a Function-Call Split block or a common function-call initiator block such as a Stateflow chart to schedule function-calls for these two input ports in the required order.

Test Harness for Export-Function Models Without Strict Scheduling

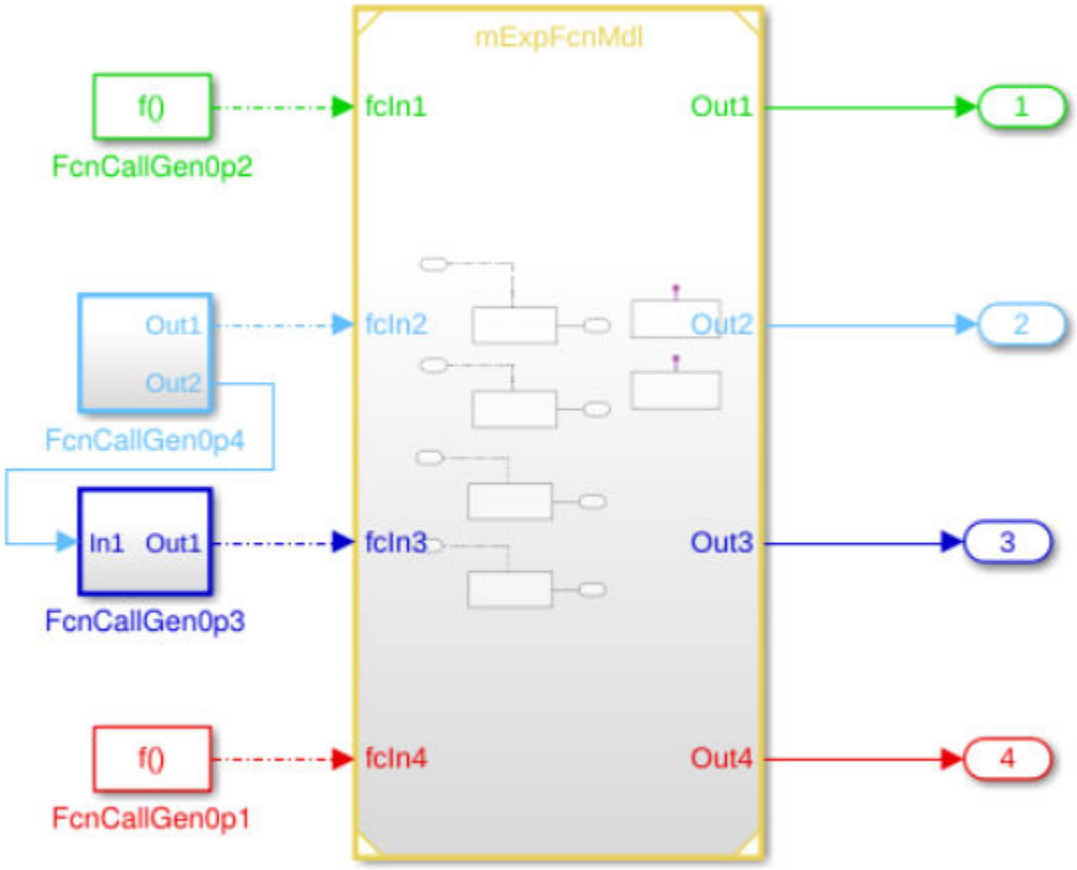
Reference the export-function model in a test harness and connect ports to Function Generator blocks.



If you clear the check box for the configuration parameter **Enable strict scheduling checks for a referenced model** and the test harness model is in signal taking mode. The function-call initiators are sorted based on their sample time priorities. For this example, the execution order is `FcnCallGen0p1` > `FcnCallGen0p2` > `FcnCallGen0p3` > `FcnCallGen0p1`.

Data Dependency Error Caused by Data Sorting Rules

Consider a model where the output from one function-call initiator is the input to another.



The function-call initiator `FcnCallGen0p3` should be executed before `FcnCallGen0p4`. However, because `FcnCallGen0p4` is also a source for `FcnCallGen0p3` a data dependency occurs and Simulink displays an error.

Update Diagram 4
11:24 PM Elapsed: 0.441 sec

The function-call initiators invoking the export-function model referenced by Model block 'mHarness_ExpFcnMdl_Loop/Model' must execute in the order of their sample times or task priorities. In a model with a single task, this implies that 'mHarness_ExpFcnMdl_Loop/FcnCallGen0p3' must execute before 'mHarness_ExpFcnMdl_Loop/FcnCallGen0p4'. However, applying this rule caused a data dependency violation. Consider clearing the 'Configuration Parameters' > 'Solver' > 'Treat each discrete rate as a separate task' option or tracing the data connections between the blocks listed below to resolve the data dependency loop.

Component: Simulink | Category: Model error

Block 'mHarness_ExpFcnMdl_Loop/FcnCallGen0p3' is involved in the loop.

Component: Simulink | Category: Model error

Block 'mHarness_ExpFcnMdl_Loop/FcnCallGen0p4' is involved in the loop.

Component: Simulink | Category: Model error

Block 'mHarness_ExpFcnMdl_Loop/Model' is involved in the loop.

Test Harness for Models with Initialize, Reset, and Terminate Function Blocks

If a Model block references a model that has an initialize, reset, or terminate ports, the function-call initiators connected to these ports have a higher priority than other function-call input ports. For example, export-function models, rate-based models, and JMAAB-B models can have other function-call input ports. Simulink sorts function-call initiators in the follow order:

- Initialize, reset, and then terminate ports.
- If there is more than one reset port, initiators to those reset ports are not sorted. For example, if a model has one initialize port driven by block `A`, two reset ports driven by blocks `B` and `C`, and one terminate port driven by block `D`, then Simulink sorts in the order A, B or C, and then D. B and C are sorted using general sorting rules.

Initiators for Model Block in Test Harness

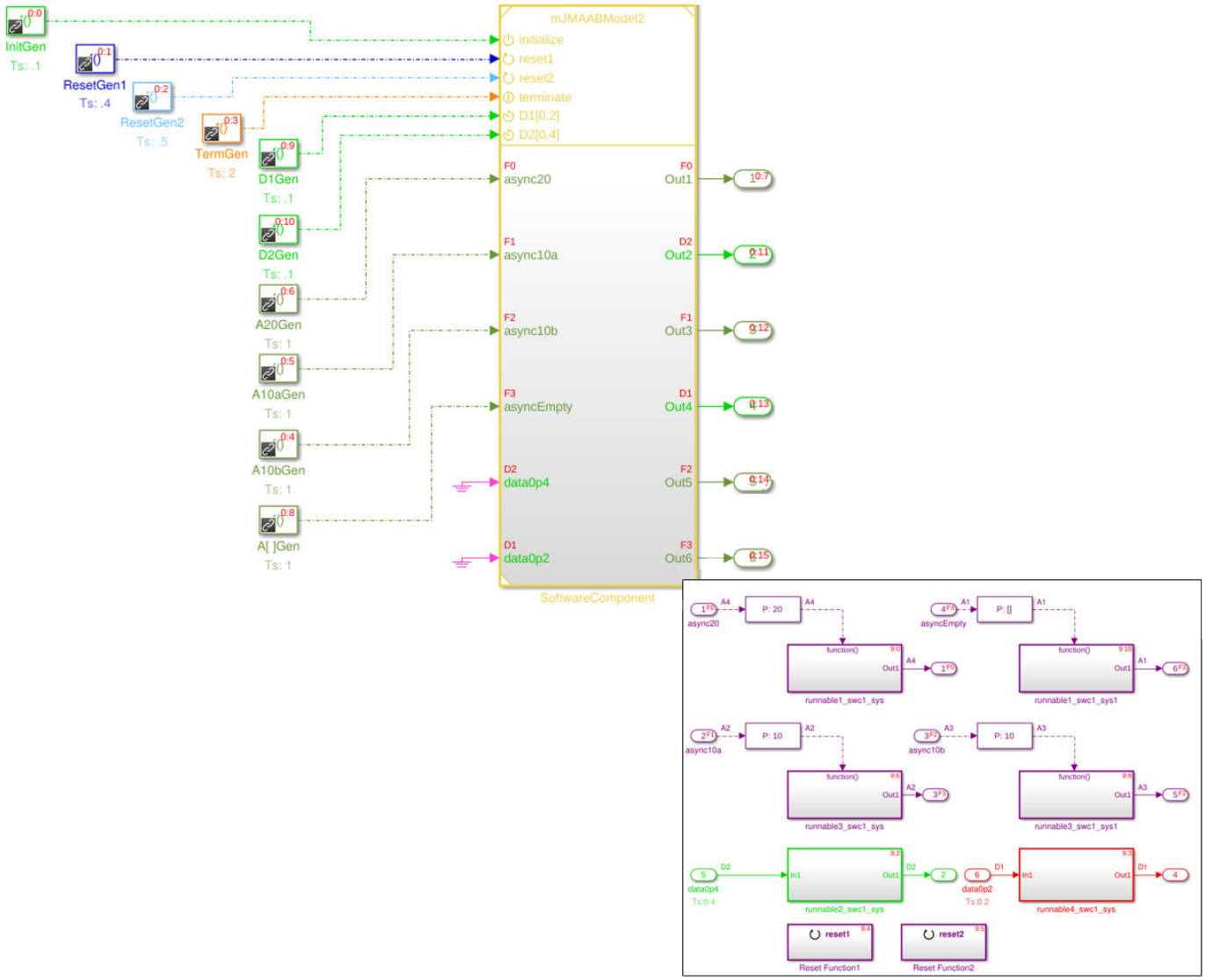
Add function-call event ports to a Model block in a test harness that references a rate-based model or JMAAB-B model by selecting the Model block parameter **Schedule rates**.

In a single tasking model, all discrete rates are in the same task. In a multi-tasking model, discrete rates with the same value execute in the same task. Simulink sorts test harness initiators in the same task in the following order:

- Initialize, reset, and then terminate ports.
- Function-call input ports mapped to asynchronous function-call root Inport blocks if adapted model is a JMAAB-B model. Among those "async" function-call input ports, use the task priorities specified by the Asynchronous Task Specification block connected to the function-call root Inport block inside the referenced model to compare ports. In the follow cases, do not compare ports:
  - For two "async" function-call input ports with the same task priorities.
  - For "async" function-call input ports with an empty (unspecified) task priority
- Periodic function-call event input ports mapped to discrete rates. Use rate monotonic scheduling (RMS) rules to compare.

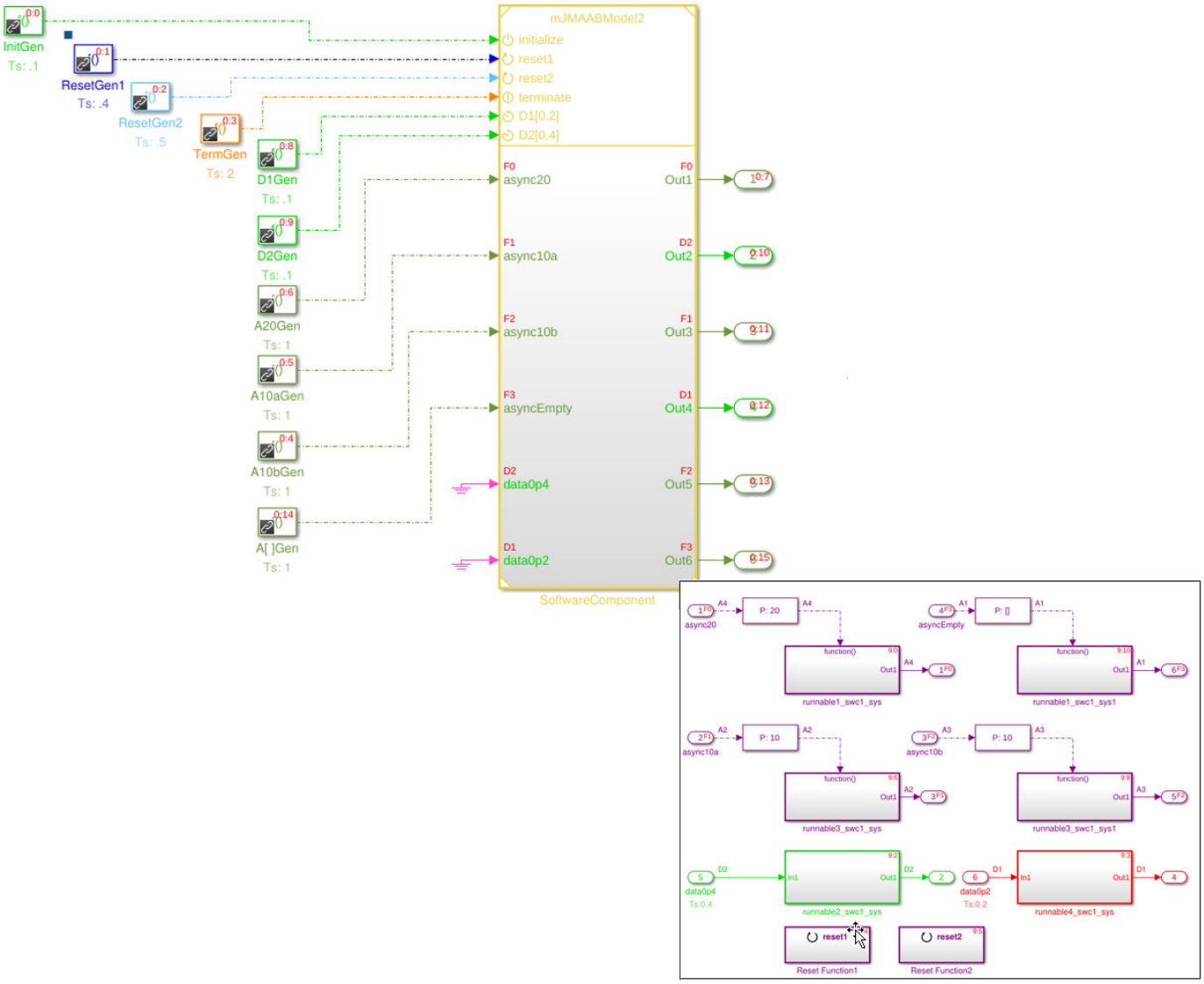
In a single tasking model, all initiators are in the same task:

- `InitGen` > `ResetGen1` or `ResetGen2` > `TermGen` > `A10aGen` or `A10bGen` or `A[]Gen` > `D1Gen` > `D2Gen`
- `A10aGen` or `A10bGen` > `A20Gen`
- Could swap relative ordering of (`ResetGen1`, `ResetGen2`) or (`A10aGen`, `A10bGen`), or (`A[]Gen`, `A20Gen`), etc.



In a multi-tasking model, initiators of the same color are in the same task.

- InitGen > D1Gen > D2Gen
- A10aGen or A10bGen > A20Gen


InitGen
Ts: .1
ResetGen1
Ts: .4
ResetGen2
Ts: .5
TermGen
Ts: 2
D1Gen
Ts: .1
D2Gen
Ts: .1
A20Gen
Ts: 1
A10aGen
Ts: 1
A10bGen
Ts: 1
A[ ]Gen
Ts: 1
mJMAABModel2
initialize
reset1
reset2
terminate
D1[0.2]
D2[0.4]
F0
async20
F1
async10a
F2
async10b
F3
asyncEmpty
D2
data0p4
D1
data0p2
F0
Out1
D2
Out2
F1
Out3
D1
Out4
F2
Out5
F3
Out6
SoftwareComponent
P: 20
P: []
P: 10
P: 10
runnable1_swc1_sys
runnable1_swc1_sys1
runnable3_swc1_sys
runnable3_swc1_sys1
runnable2_swc1_sys
runnable4_swc1_sys
reset1
reset2
Reset Function1
Reset Function2

Conditional Subsystem Output Values When Disabled

Although a conditional subsystem does not execute while it is disabled, the output signal is still available to other blocks. When a conditional subsystem is disabled and you have specified not to inherit initial conditions from an input signal, you can hold the subsystem outputs at their previous values or reset them to their initial conditions.

To specify output values when disabled:

1. Open the dialog box for an Outport block in a conditional subsystem.
2. From the **Source of initial output value** drop-down list, select `Dialog`.
3. From the **Output when disabled** drop-down list, select one of these options:
   - `held` — Maintain the most recent value.
   - `reset` — Use the initial condition when enabled.

   Note If you are connecting the output of a conditionally executed subsystem to a Merge block, set **Output when disabled** to `held` to ensure consistent simulation results.

   If you are using simplified initialization mode, you must select `held` when connecting a conditionally executed subsystem to a Merge block. For more information, see “Underspecified initialization detection”.
4. In the **Initial output** box, enter the initial value.

Note If an Outport block in an Enabled Subsystem resets its output when disabled at a different rate from the execution of the subsystem contents, both the disabled and execution outputs write to the subsystem output. This behavior can cause unexpected results.

See Also

More About

- “Conditionally Executed Subsystems Overview” on page 10-3
- “Conditional Subsystem Initial Output Values” on page 10-37

Simplified Initialization Mode

In this section...

Initialization mode controls how Simulink handles:

- Initialization values for conditionally executed subsystems.
- Initial values for Merge blocks.
- Discrete-Time Integrator blocks.
- Subsystem elapsed time.

The default initialization mode for a model is simplified. This mode uses enhanced processing to improve consistency of simulation results and helps to:

- Attain the same simulation results with the same inputs when using the same blocks in a different model.
- Avoid unexpected changes to simulation results as you modify a model.

When to Use Simplified Initialization

Use simplified initialization mode for models that contain one or more of the following blocks:

- Conditional subsystem blocks.
- Merge blocks. If a root Merge block has an empty matrix (`[]`) for its initial output value, simplified mode uses the default ground value of the output data type.
- Discrete-Time Integrator blocks. Simplified mode always uses the initial value as both the initial and reset value for output from a Discrete-Time Integrator block.

Use simplified mode if your model uses features that require simplified initialization mode, such as:

- Specify a structure to initialize a bus.
- Branch merged signals inside a conditional subsystem.

Set Initialization Mode to Simplified

Simplified mode is the default initialization mode when creating a new Simulink model. If your model is using classic mode, you might need to make changes after you select simplified mode. See “Convert from Classic to Simplified Initialization Mode” on page 10-71.

1. Open the Configuration Parameters dialog box. On the **Modeling** tab and from the **Setup** section, select **Model Settings**  .
2. In the search box, enter **Underspecified initialization detection**.
3. From the drop-down list, select `Simplified`.

See Also

More About

- “Conditionally Executed Subsystems Overview” on page 10-3

Classic Initialization Mode

In this section...

When to Use Classic Initialization

Initialization mode controls how Simulink handles the initialization values for conditionally executed subsystems.

`Classic` mode was the default initialization mode for Simulink models created in R2013b or before. You can continue to use classic mode if:

- The model does not include any modeling elements affected by simplified mode.
- The behavior and requirements of simplified mode do not meet your modeling goals.
- The work involved in converting to simplified mode is greater than the benefits of simplified mode. See “Convert from Classic to Simplified Initialization Mode” on page 10-71.

Set Initialization Mode to Classic

To set classic initialization mode:

1. Open the Configuration Parameters dialog box. On the **Modeling** tab and from the **Setup** section, select **Model Settings**  .
2. In the search box, enter **Underspecified initialization detection**.
3. From the drop-down list, select `Classic`.

Classic Initialization Issues and Limitations

Using classic initialization mode can result in one or more of the following issues. You can address these issues by using simplified mode. The description of each issue includes an example of the behavior in classic mode, the behavior when you use simplified mode, and a summary of the changes you must make to use simplified mode.

- “Identity Transformation Can Change Model Behavior” on page 10-58.

  Conditional subsystems that include identical subsystems can display different initial values before the first execution if both of the following apply:

  - The model uses classic initialization mode.
  - One or more of the identical subsystems outputs to an identity transformation block.
- “Inconsistent Output with Discrete-Time Integrator or S-Function Block” on page 10-61

  Conditional subsystems that use classic initialization mode and whose output connects to a Discrete-Time Integrator block or S-Function block can produce inconsistent output.
- “Execution Order Affecting Merge Block Output” on page 10-63

  The execution order of conditional subsystems that used classic mode initialization, when connected to a Merge block, can affect the output of that Merge block. A change in block execution order can produce unexpected results.
- When you rename the Merge block source subsystem blocks, the initial output of the Merge block can change.

  When two or more subsystems are feeding different initial output values to a Merge block that does not specify its own initial output value, renaming one of the subsystems can affect the initial output of the Merge block in classic initialization mode.
- “Simulink does not provide correct consistency check” on page 10-70

  Simulink does not provide the correct consistency check for settings between two Outport blocks connected through a model reference boundary.

For additional information about the tasks involved to convert a model from classic to simplified mode, see “Convert from Classic to Simplified Initialization Mode” on page 10-71.

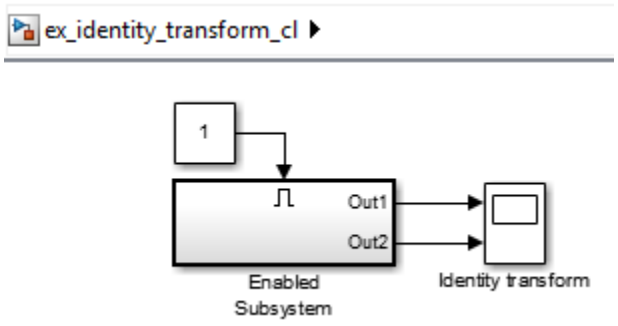
Identity Transformation Can Change Model Behavior

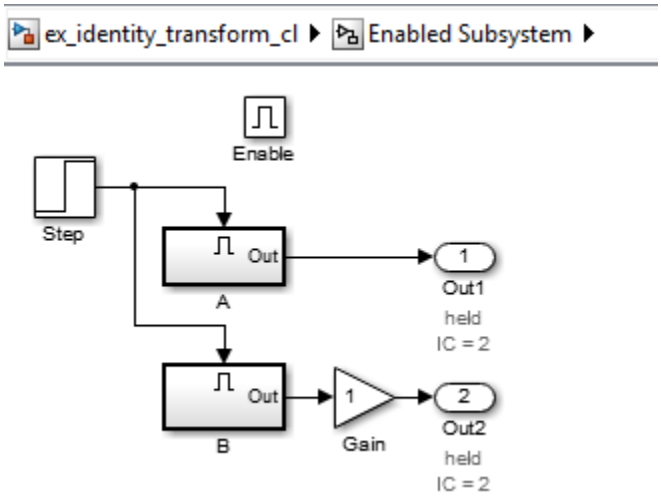
Conditional subsystems that include identical subsystems can display different initial values before the first execution if both of the following apply:

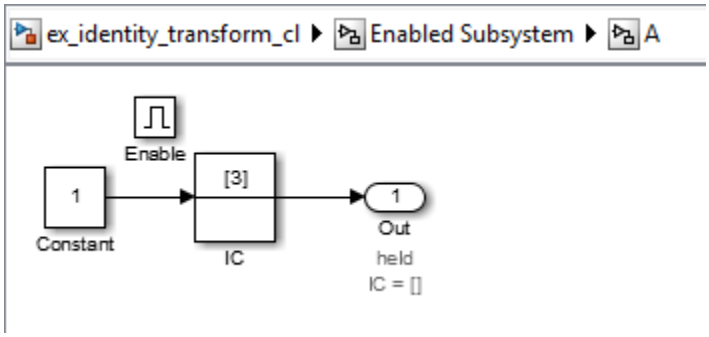
- The model uses classic initialization mode.
- One or more of the identical subsystems outputs to an identity transformation block.

An identity transformation block is a block that does not change the value of its input signal. Examples of identify transform blocks are a Signal Conversion block or a Gain block with a value of 1.

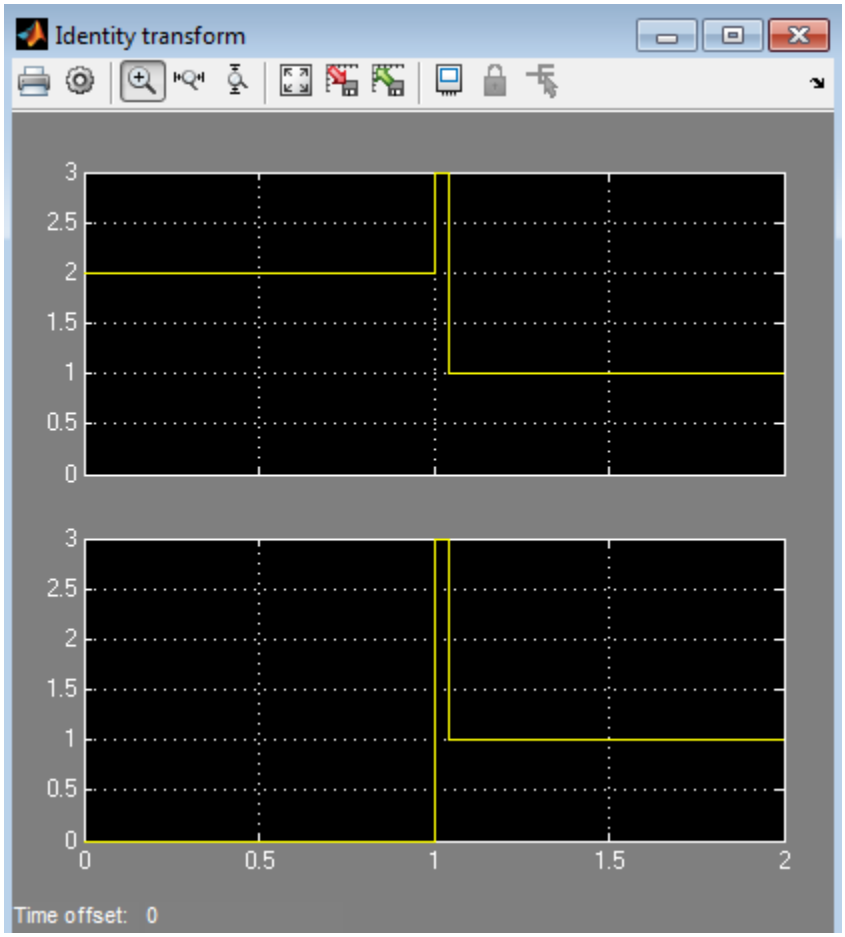
In the `ex_identity_transform_cl` model, subsystems A and B are identical, but B outputs to a Gain block, which in turn outputs to an Outport block.







When you simulate the model, the initial value for `A` (the top signal in the Scope block) is `2`, but the initial value of `B` is `0`, even though the subsystems are identical.

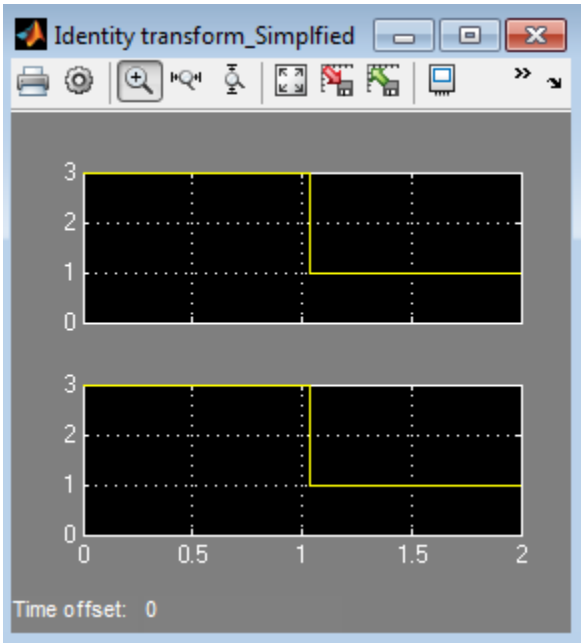


If you update the model to use simplified initialization mode (see `ex_identity_transform_simpl`), the model looks the same. The steps required to convert `ex_identity_transform_cl` to `ex_identity_transform_simpl` are:

1. Set **Underspecified initialization detection** to `Simplified`.
2. For the Outport blocks in subsystems A and B, set the **Source of initial output value** parameter to `Input signal`.

   You can also get the same behavior by setting the **Source of initial output value** parameter to `Dialog` and the **Initial output** parameter to `3`.

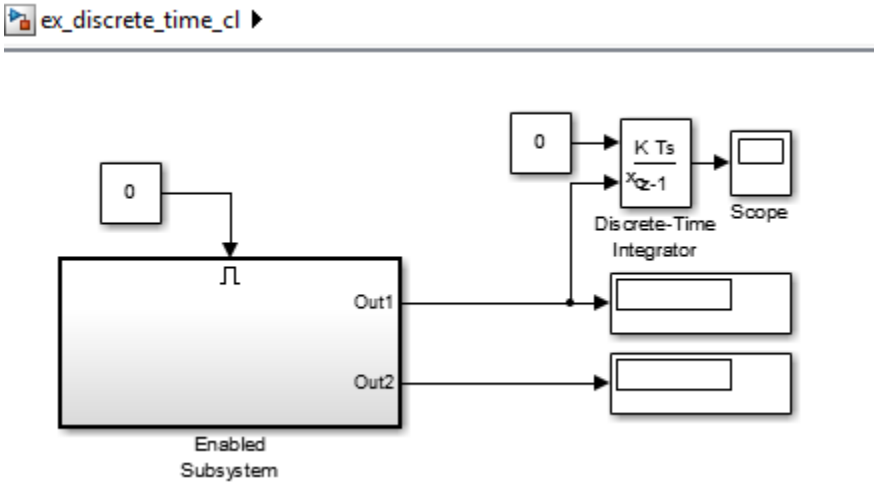
When you simulate the updated model, the connection of an identity transformation does not change the result. The output is consistent in both cases.

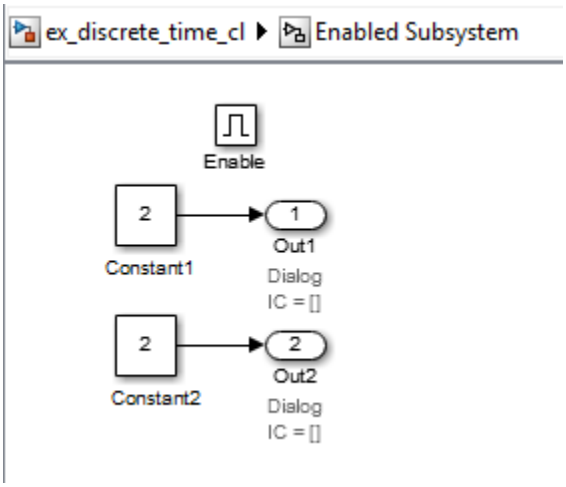


Inconsistent Output with Discrete-Time Integrator or S-Function Block

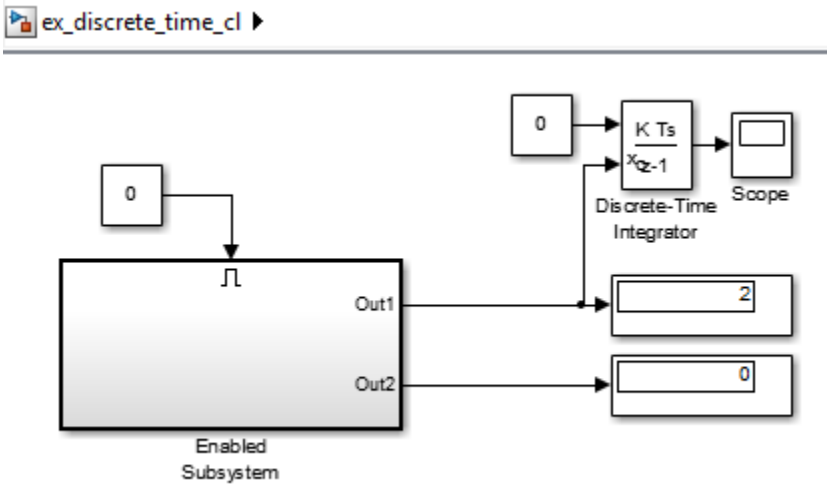
Conditional subsystems that use classic initialization mode and whose output connects to a Discrete-Time Integrator block or S-Function block can produce inconsistent output.

In the `ex_discrete_time_cl` model, the enabled subsystem includes two Constant blocks and outputs to a Discrete-Time Integrator block. The enabled subsystem outputs to two Display blocks.





When you simulate the model, the two display blocks show different values.



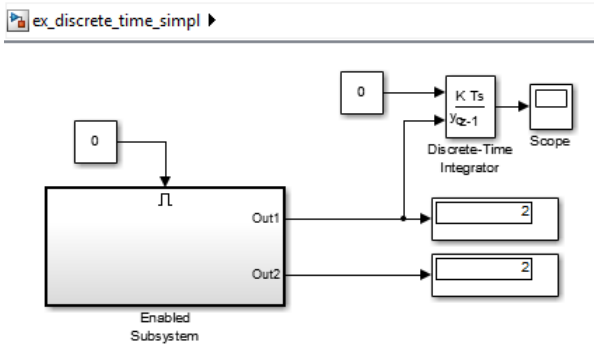
The Constant1 block, which is connected to the Discrete-Time Integrator block, executes, even though the conditional subsystem is disabled. The top Display block shows a value of 2, which is the value of the Constant1 block. The Constant2 block does not execute, so the bottom Display block shows a value of 0.

If you update the model to use simplified initialization mode (see `ex_discrete_time_simpl`), the model looks the same. The updated model corrects the inconsistent output issue by using simplified mode. The steps required to convert `ex_discrete_time_cl` to `ex_discrete_time_simpl` are:

1. Set **Underspecified initialization detection** to `Simplified`.
2. For the Outport blocks Out1 and Out2, set the **Source of initial output value** parameter to `Input signal`. This setting explicitly inherits the initial value, which in this case is 2.

   You can also get the same behavior by setting the **Source of initial output value** parameter to `Dialog` and the **Initial output** parameter to 2.

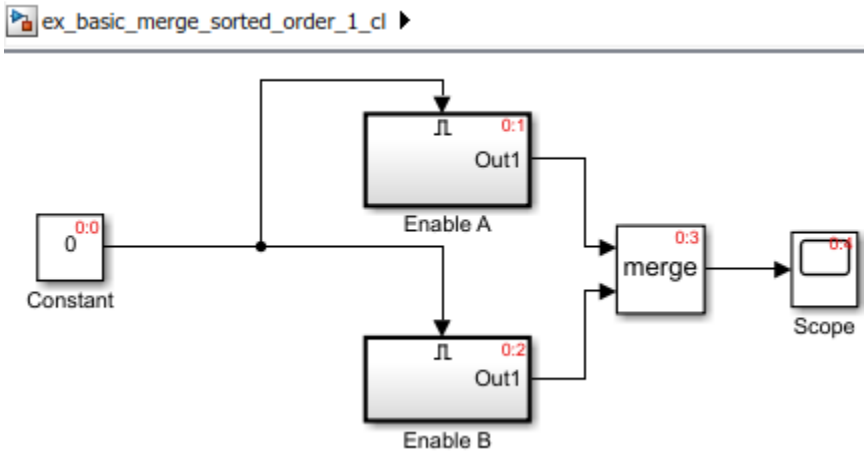
When you simulate the updated model, the Display blocks show the same output. The output value is 2 because both Outport blocks inherit their initial value.

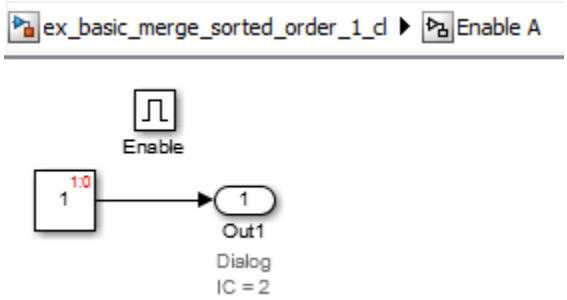


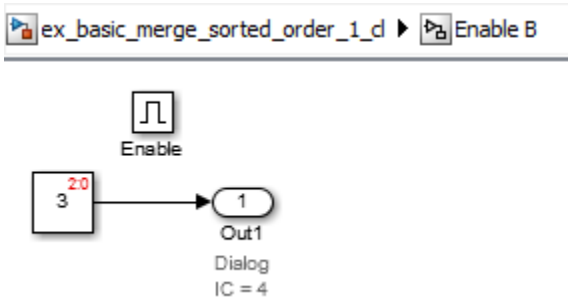
Execution Order Affecting Merge Block Output

The execution order of conditional subsystems that use classic mode initialization, when connected to a Merge block, can affect the output of that Merge block. A change in block execution order can produce unexpected results. The behavior depends on how you set the **Output When Disabled** parameter.

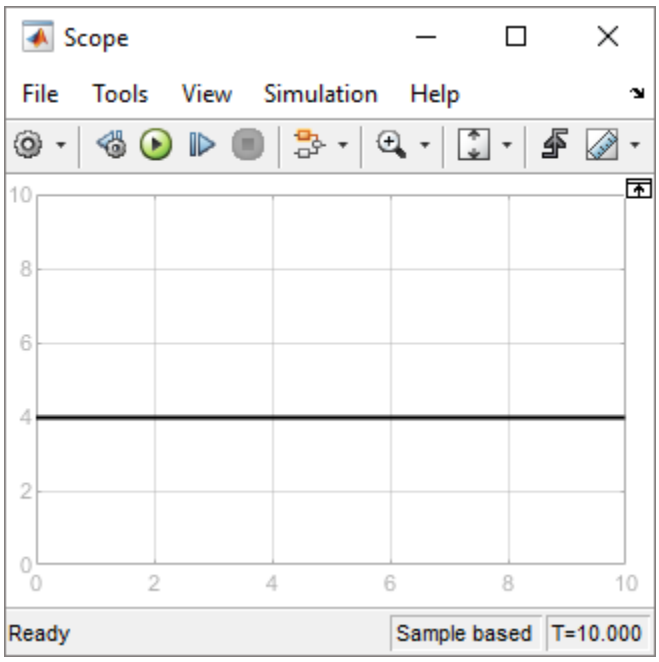
The `ex_basic_merge_sorted_order_1_cl` model has two identical enabled subsystems (Enable A and Enable B) that connect to a Merge block. When you simulate the model, the red numbers show the sorted execution order of the blocks.





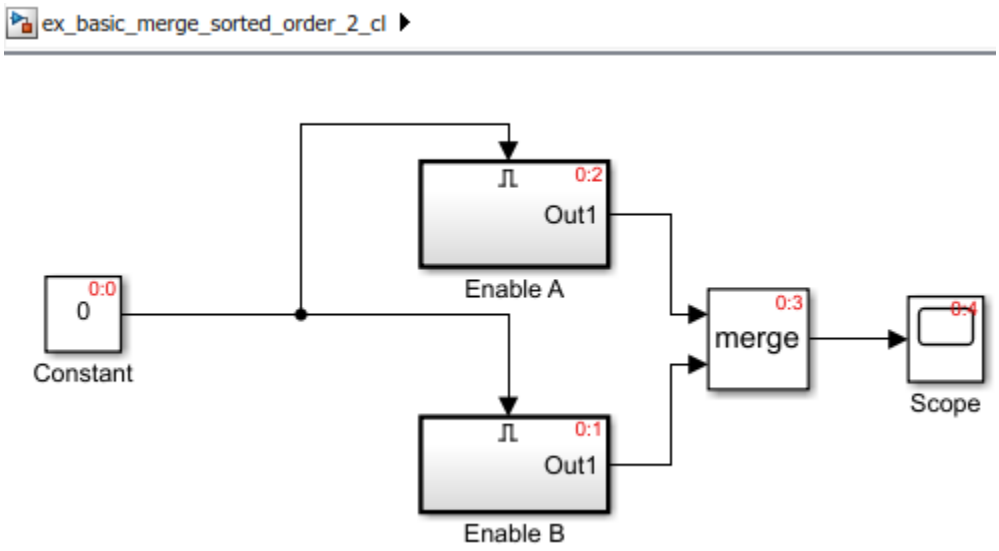


When you simulate the model, the Scope block looks like the following:

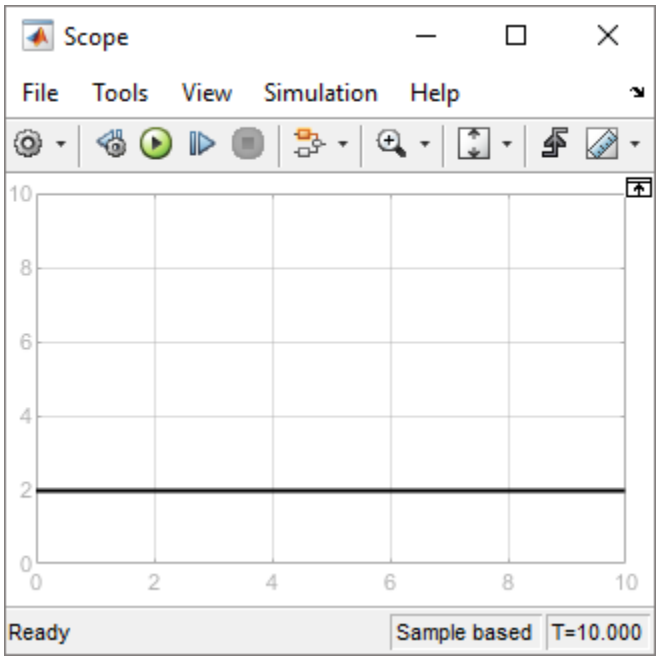


The `ex_basic_merge_sorted_order_2_cl` model is the same as `ex_merge_sorted_1_cl`, except that the block execution order is the reverse of the default execution order. To change the execution order:

1. Open the Properties dialog box for the Enable A subsystem and set the **Priority** parameter to 2.
2. Set the **Priority** of the Enable B subsystem to 1.



When you simulate the model using the different execution order, the Scope block looks like the following:

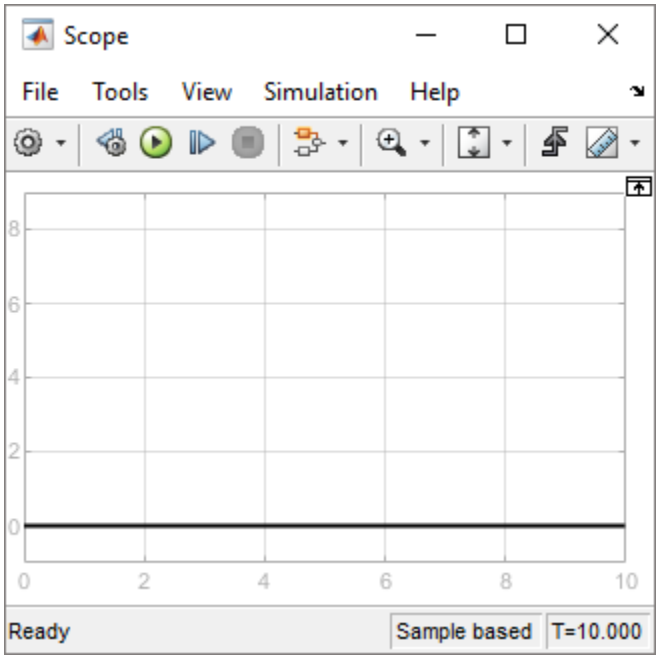


The change in execution order produces different results from identical conditional subsystems.

To update the models to use simplified initialization mode (see `ex_basic_merge_sorted_order_1_simpl` and `ex_basic_merge_sorted_order_2_simpl`):

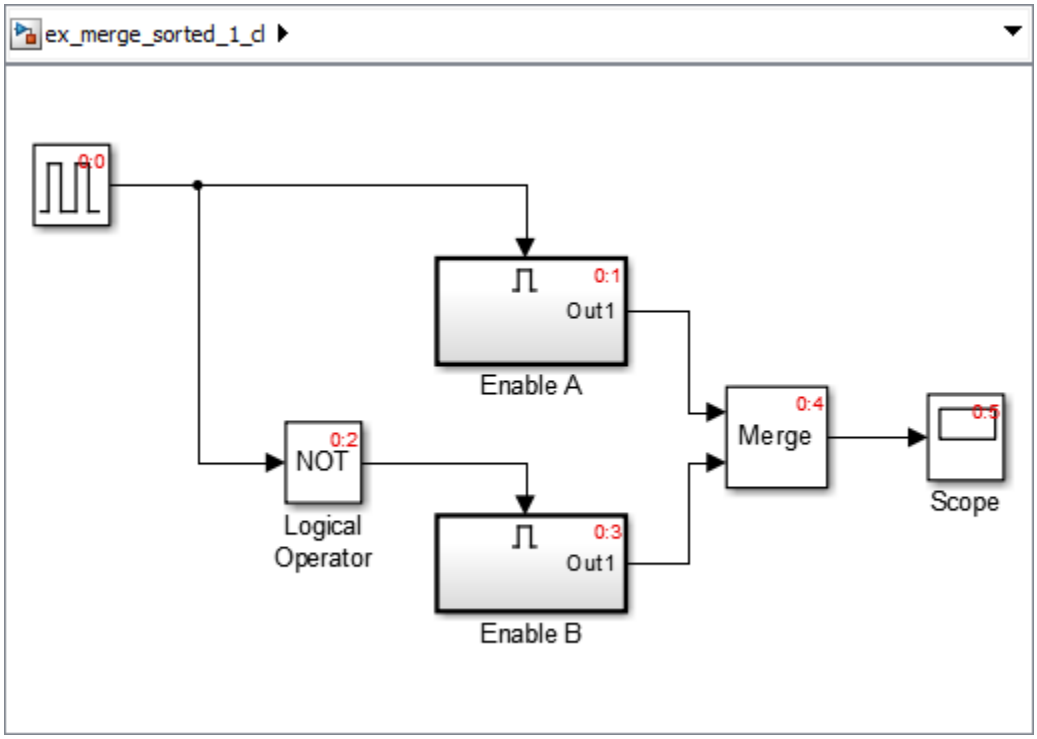
1 Set **Underspecified initialization detection** to `Simplified`.

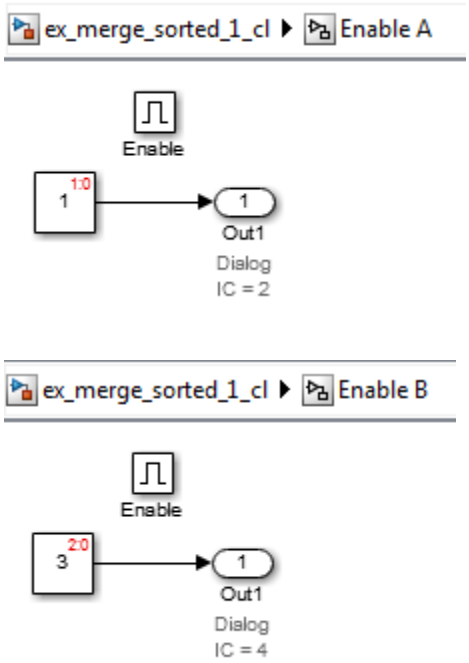
The **Initial Output** parameter of the Merge block is an empty matrix, `[]`, by default. Hence, the initial output value is set to the default initial value for this data type, which is 0. For information on default initial value, see “Initialize Signal Values” on page 75-9. When you simulate each simplified mode model, both models produce the same results.



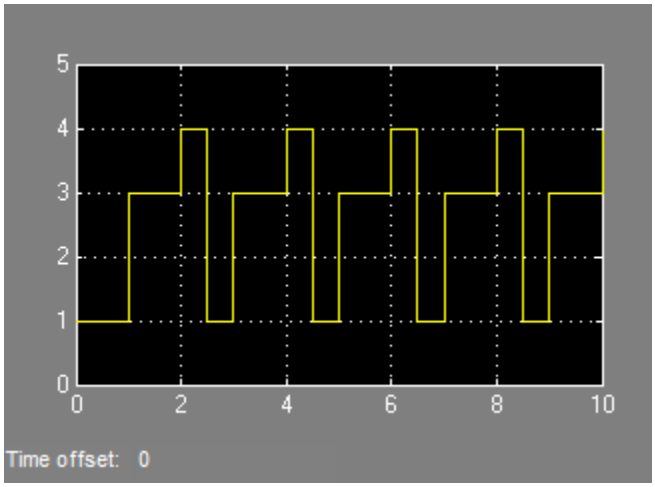
Using Output When Disabled Parameter Set to Reset

The `ex_merge_sorted_1_cl` model has two enabled subsystems (Enable A and Enable B) that connect to a Merge block. When you simulate the model, the red numbers show the sorted execution order of the blocks.



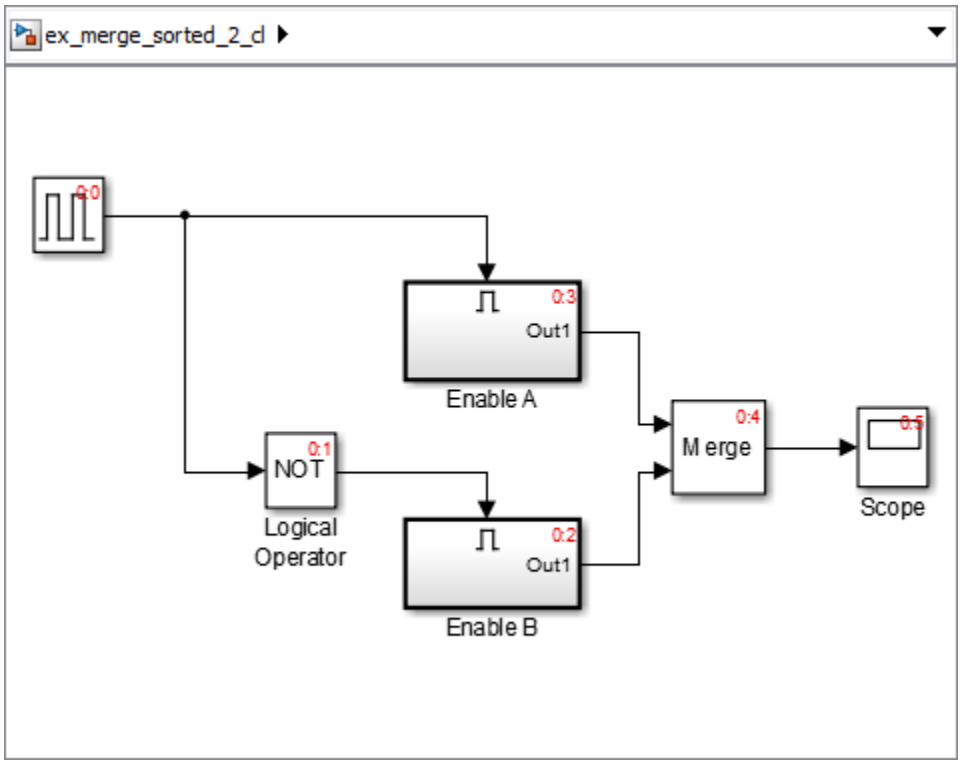


When you simulate the model, the Scope block looks like the following:

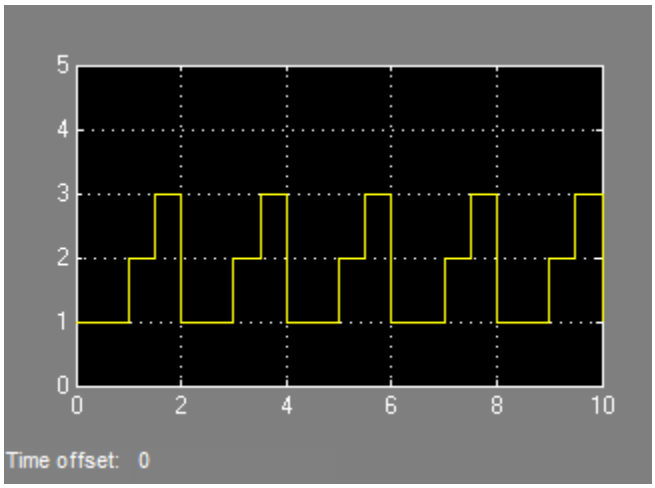


The `ex_merge_sorted_2_cl` model is the same as `ex_merge_sorted_1_cl`, except that the block execution order is the reverse of the default execution order. To change the execution order:

1. Open the Properties dialog box for the Enable A subsystem and set the **Priority** parameter to 2.
2. Set the **Priority** of the Enable B subsystem to 1.



When you simulate the model using the different execution order, the Scope block looks like:

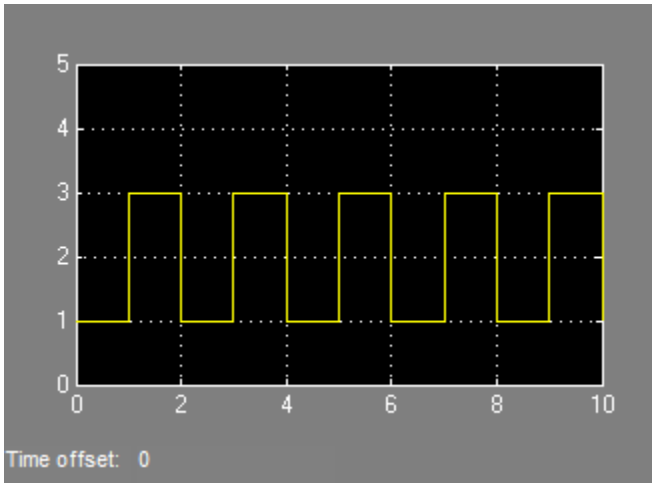


The change in execution order produces different results from identical conditional subsystems.

To update the models to use simplified initialization mode (see `ex_merge_sorted_1_simpl` and `ex_merge_sorted_2_simpl`):

1. Set **Underspecified initialization detection** to `Simplified`.
2. For the Outport blocks in Enable A and Enable B, set the **Output when disabled** parameter to `held`. Simplified mode does not support `reset` for output ports of conditional subsystems driving Merge blocks.

When you simulate each simplified mode model, both models produce the same results.



Tunable Parameters

Many block parameters are tunable. A *tunable parameter* is a parameter whose value can be changed without recompiling the model (see “Model Compilation” on page 3-2 for more information on compiling a model). For example, the gain parameter of the Gain block is tunable. You can alter the block's gain while a simulation is running. If a parameter is not tunable and the simulation is running, the dialog box control that sets the parameter is disabled.

When you change the value of a tunable parameter, the change takes effect at the start of the next time step.

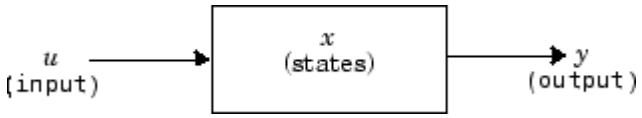
State

Typically the current values of some system, and hence model, outputs are functions of the previous values of temporal variables. Such variables are called states. Computing a model's outputs from a block diagram hence entails saving the value of states at the current time step for use in computing the outputs at a subsequent time step. This task is performed during simulation for models that define states.

Two types of states can occur in a Simulink model: discrete and continuous states. A continuous state changes continuously. Examples of continuous states are the position and speed of a car. A discrete state is an approximation of a continuous state where the state is updated (recomputed) using finite (periodic or aperiodic) intervals. An example of a discrete state would be the position of a car shown on a digital odometer where it is updated every second as opposed to continuously. In the limit, as the discrete state time interval approaches zero, a discrete state becomes equivalent to a continuous state.

Blocks implicitly define a model's states. In particular, a block that needs some or all of its previous outputs to compute its current outputs implicitly defines a set of states that need to be saved between time steps. Such a block is said to have states.

The following is a graphical representation of a block that has states:



Blocks that define continuous states include the following standard Simulink blocks:

- Integrator
- State-Space
- Transfer Fcn
- Variable Transport Delay
- Zero-Pole

The total number of a model's states is the sum of all the states defined by all its blocks. Determining the number of states in a diagram requires parsing the diagram to determine the types of blocks that it contains and then aggregating the number of states defined by each instance of a block type that defines states. This task is performed during the Compilation phase of a simulation.

Simulink does not provide correct consistency check

Simulink does not provide the correct consistency check for settings between two Outport blocks connected through a model reference boundary.

Simulink either throws a false warning or no warning when all of the following conditions are true:

- The option **Underspecified initialization detection** is set to `Classic`.
- The model contains a Model block.
- In the referenced model, a root Outport block is driven directly (or indirectly through virtual blocks) by a conditionally executed subsystem. In this scenario, the Outport block corresponding to the conditionally executed subsystem output is a *source* Outport block.
- In the top model, the output port of the Model block that is driven by the source Outport block, in turn, drives a *destination* Outport block of a conditionally executed subsystem.

If both the source and destination Outport blocks are in the same model, and the settings **Initial output** and **Output when disabled** (if applicable) for both Outport blocks differ, Simulink throws an error. However, in the case described above, Simulink either throws a false warning when the two Outport blocks have the same settings or throws no warning or error when they are different.

Convert from Classic to Simplified Initialization Mode

If you switch the initialization mode from classic to simplified mode, you can encounter several issues that you must fix. For most models, the following approach helps you to address conversion issues more efficiently.

1. Save the existing model and simulation results for the model.
2. Simulate the model and address any warnings.
3. In the Model Advisor, in the Simulink checks section, run the checks in the folder “Migrating to Simplified Initialization Mode Overview”.
4. Address the issues that Model Advisor identifies.
5. Simulate the model to make sure that there are no errors.
6. Rerun the Model Advisor checks in the folder “Migrating to Simplified Initialization Mode Overview” check to confirm that the modified model addresses the issues related to initialization.

For examples of models that have been converted from classic initialization mode to simplified initialization mode, see “Classic Initialization Issues and Limitations” on page 10-57.

Blocks to Consider

Discrete-Time Integrator Blocks

Discrete-Time Integrator block behaves differently in simplified mode than it does in classic mode. The changes for simplified mode promote more robust and consistent model behavior. For details, see “Behavior in Simplified Initialization Mode” in the Discrete-Time Integrator block reference documentation.

Library Blocks

Simulink creates a library assuming that classic mode is in effect. If you use a library block that is affected by simplified mode in a model that uses simplified mode, then use the Model Advisor to identify changes you must make so that the library block works with simplified mode.

See Also

Related Examples

- “Conditionally Executed Subsystems Overview” on page 10-3

Create an Export-Function Model

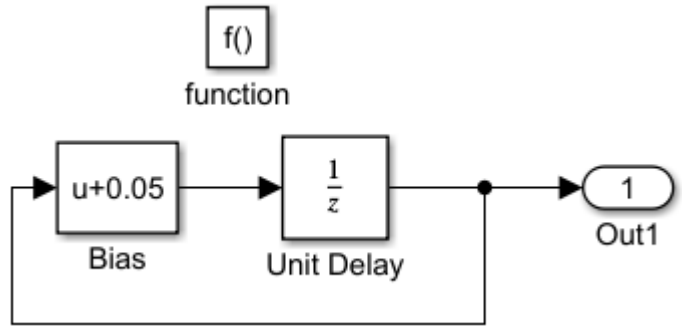
The first step for creating independent functions in the generated code from a Simulink model is to define the functions in the context of an export-function model. See “Export-Function Models Overview” on page 10-97.

To open a completed export-function model, see `ex_export_function_model`.

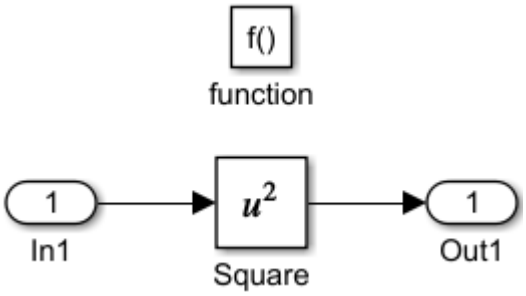
Create Model Algorithms

At the top-level of an export-function model, functions are modeled within Function-Call Subsystem, function-call Model, Simulink Function, and S-Function blocks. This example uses Function-Call Subsystem blocks to model two functions.

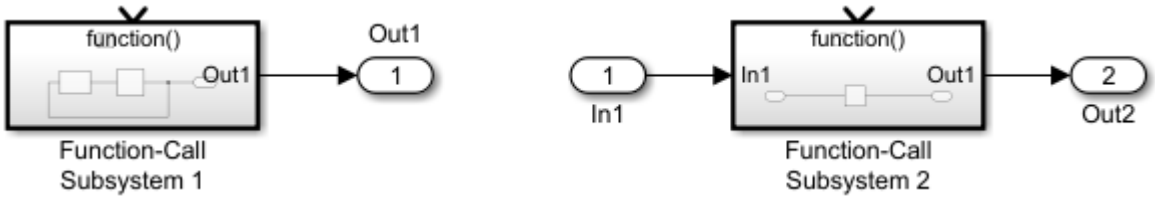
1. Add two Function-Call Subsystem blocks.
2. In the first subsystem, model a unit delay that increments by 0.05 at each time step.



3. In the second subsystem, model the square of an input signal.



4. Add Inport and Outport blocks.



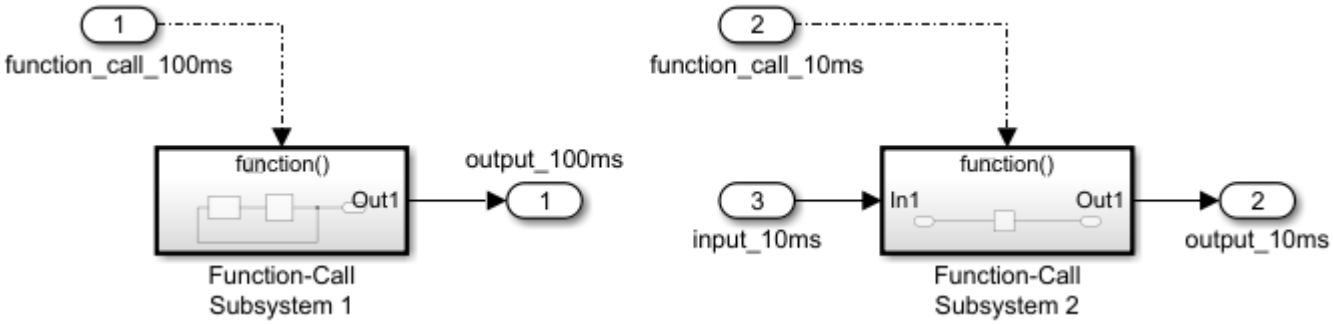
Add Function-Call Inputs

Inport blocks configured to output function-call signals control the execution of Function-Call Subsystem blocks during a simulation. The function-call Inport blocks also create an entry point function in the generated code.

1. Attach Inport blocks to the `function()` input ports on the Function-Call Subsystem blocks.
2. Specify sample times. Open the Inport block dialogs and select the **Signal Attributes** tab. Select the **Output function call** check box. Set **Sample time** for the delay function to `0.1` (100 ms), and the square function to `0.01` (10 ms).

   Setting the sample time is only for simulation testing. Sample time values do not affect the generated code. However, comments are added in the code identifying the rate you should call the functions.

   You could set the sample times to `-1` and allow any function-call rate (periodic or aperiodic) during simulation. By setting sample times, Simulink checks the calling rate with the specified rate and displays an error if there is a mismatch.
3. Rename blocks. The names help to identify signals in the generated code.
4. Update the model (Ctrl-D). Dotted-and-dashed lines identify function-call signals.



Satisfy Export-Function Model Requirements

1. Open the Configuration Parameters dialog box. On the **Modeling** tab and from the **Setup** section, select **Model Settings**.
2. In the Solver pane, set Solver selection **Type** to `Fixed-step`, **Solver** to `auto`, and Fixed-step size to `auto`.
3. In the Code generation pane, set **System target file** to `ert.tlc`. This step requires an Embedded Coder license.

After you create an export-function model, you can test it with simulations. Choose one of the following simulation testing methods: “Test Export-Function Model Simulation Using Input Matrix” on page 10-75, “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79, and “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82.

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86
- “Generate Code for Export-Function Model” on page 10-90

Test Export-Function Model Simulation Using Input Matrix

When function-call sequencing is simple enough to be specified as a model input, simulation using an input matrix is the preferred method for testing an export-function model.

- Create time vectors that specify function-call times.
- Create a matrix that adds input data to the time vectors.
- Run simulation.

To create the model in this example, see “Create an Export-Function Model” on page 10-72.

Create Function-Call Inputs and Data Inputs

Create time-vectors indicating when events occur for root-level function-call Inport blocks and data-vectors for root-level data Inport blocks.

1. For function-call Inport blocks `1` and `2`, create column vectors with time steps of `0.1` and `0.01`. In the MATLAB Command Window, enter

   ```
   t1 = [0:0.1:10]';
   t2 = [0:0.01:10]';
   ```

   - The time vector must be monotonically increasing and of double data type.
   - If the sample time for a function-call Inport block is specified, the values in the corresponding time vector must all be integer multiples of the specified value.
   - To specify multiple function-calls at a given time step, repeat the time value accordingly. For example, to specify three events at t = 0.1 and 2 events at t = 0.5, list 0.1 three times and 0.5 twice in your time vector, `t1 = [0.1 0.1 0.1 0.5 0.5]'`.
   - To use nonperiodic sample times, set the Inport block **Sample time** to `-1` and provide a nonuniform time vector, e.g. `t1 = [0, 0.1, 0.2, 0.4, 0.8]`.
2. Create a matrix with time steps and data values for data Inport block `3`.

   ```
   sine_data = sin(0:0.01:10)';
   d3 = [t2,sine_data];
   ```

   The data input can use any supported format as described in “Forms of Input Data” on page 70-36.

The following table provides additional information for specifying the time vector `t`.

Sample time type for internal Trigger Port block set to:	Root-level Function-Call Inport block with inherited sample time (-1)	Root-level Function-Call Inport block with discrete sample time
`triggered`	Nondecreasing column vector. The function-call subsystem executes at the times specified by the column vector. The sample times can be periodic or aperiodic. If you specify an empty matrix (`[]`), the function-call subsystem does not execute.	Nondecreasing column vector. Each element in the column vector must be an integer multiple of the sample time specified by the Inport block. The function-call subsystem executes at the times specified by the column vector. Alternatively, specify an empty matrix (`[]`) and the function-call subsystem executes once at every sample time specified by the Inport block.
`periodic`	Configuration not allowed.	Empty matrix(`[]`). The function-call subsystem executes at the times specified by the Inport block calling it.

Simulate Export-Function Model

Simulate the export-function model to test and observe its behavior before generating code.

1. Import time and data from the MATLAB workspace.

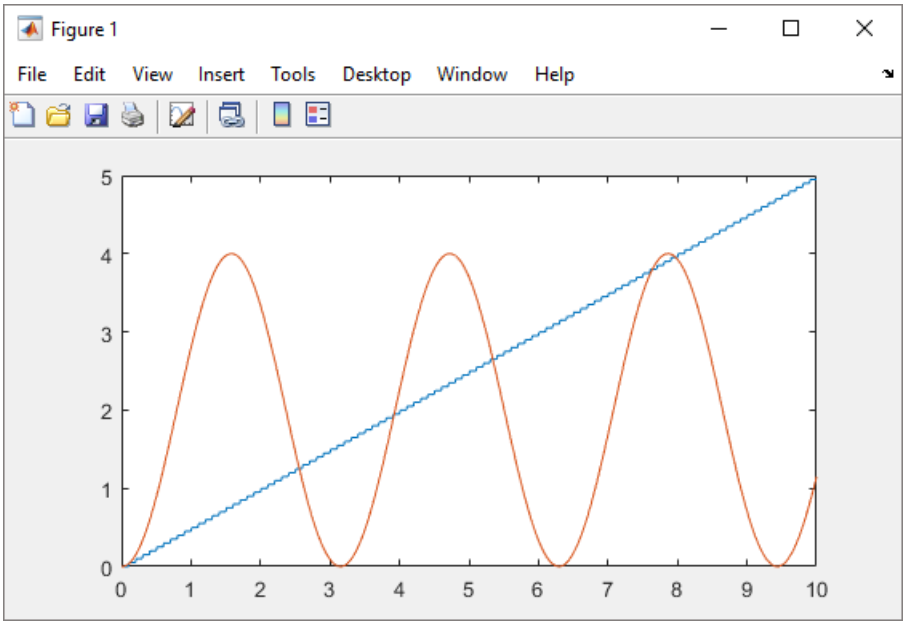
   On the **Modeling** tab, select **Model Settings**  . In the Configuration Parameters dialog box, select the **Data Import/Export** pane and set the **Input** parameter to `t1, t2, d3`.

   `t1` and `t2` are column vectors containing event times for the function-call Inport blocks `1` and `2`. `d3` is a table of input values versus time for the data Inport block `3`.

2. Run simulation.

3. Plot results. In the MATLAB Command Window, enter.

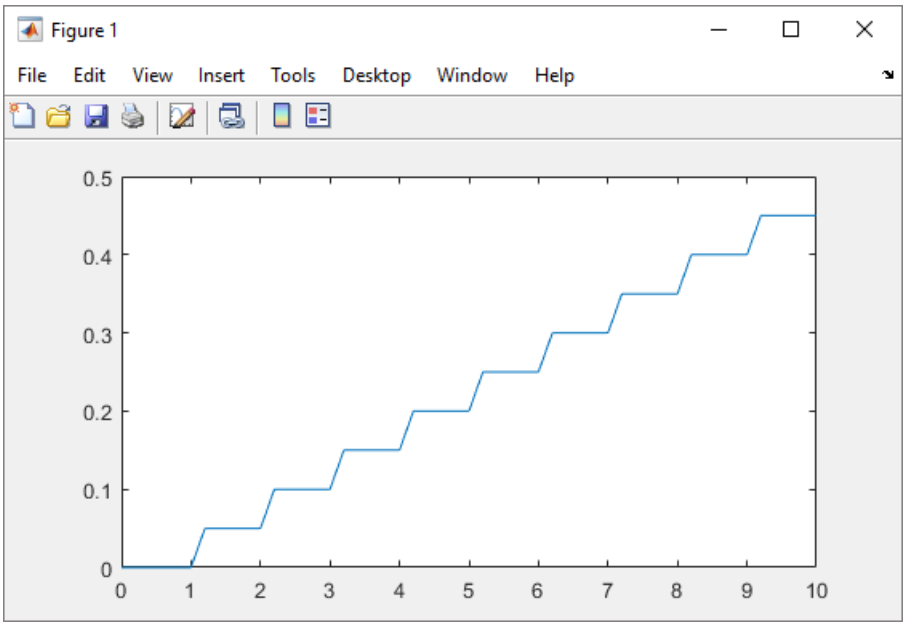
   ```
   plot(yout.time, yout.signals(1).values)
   hold
   plot(yout.time, yout.signals(2).values)
   ```



4 Change `t1` to provide events every `0.5` seconds (`0.5` is an integer multiple of the sample time of `0.1` specified in Inport block `1`).

```
t1 = [0:0.5:10]';
```

5 Rerun simulation.



After you test your model, you can generate code for the functions. See “Generate Code for Export-Function Model” on page 10-90.

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86
- “Generate Code for Export-Function Model” on page 10-90

Test Export-Function Model Simulation Using Function-Call Generators

Use multiple Function-Call Generator blocks with distinct sample times and let Simulink schedule the function-call components for simulation. This strategy is useful when the rate-monotonic scheduling behavior in Simulink is similar to the target OS behavior.

- Create a new Simulink model.
- Add a Model block that references the export-function model.
- Specify function-call events using Function-Call Generator blocks.
- Specify data inputs.
- Run simulation.

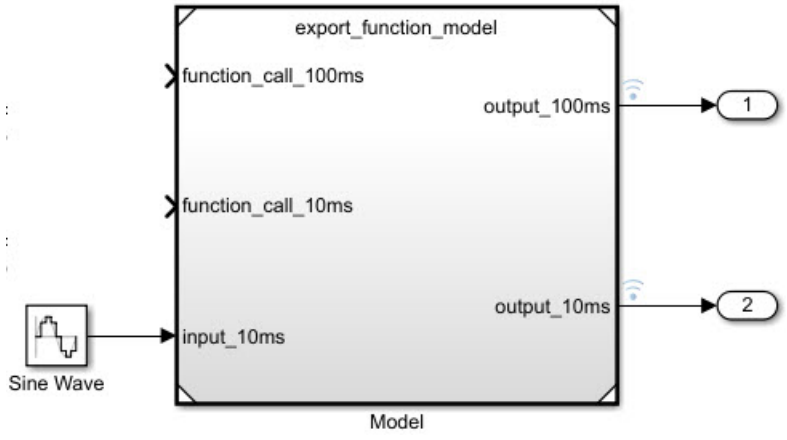
To create the model for this example, see “Create an Export-Function Model” on page 10-72.

To open a completed test model, see `ex_export_function_test_model_with_function_call_generators`.

Create Referenced Export-Function Model

Referencing an export-function model from a Model block allows the addition of function-call events and logging of data signals for testing without changing the model itself.

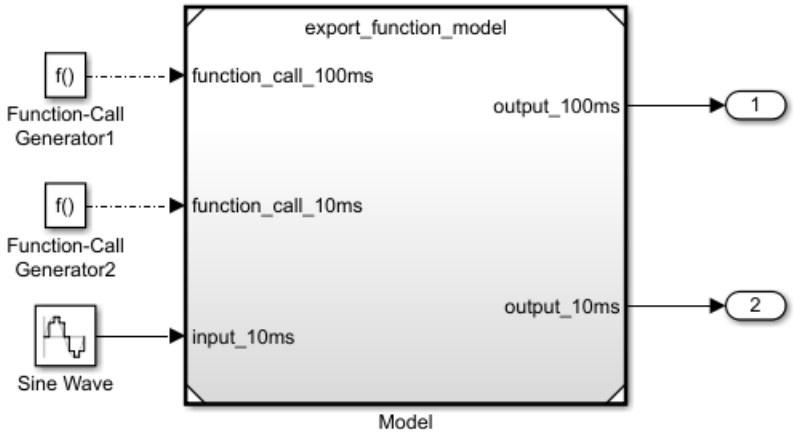
1. Add a Model block to a new Simulink model. In the **Model name** box, enter `export_function_model`.
2. Add Outport blocks to the **output_100ms** and **output_10ms** ports on the Model block.
3. Add a Sine Wave block to provide data input. Set **Amplitude** to `2` and **Sample time** to `0.01`. Connect the block to the **input_10ms** input port on the Model block.



Create Test Model (Harness) for Simulation

You use a Simulink test model only for simulation. After simulation testing, generate code from the export-function model, and then integrate exported function code with an externally coded scheduler.

1. Add a Function-Call Generator block. Set **Sample time** to `0.1`. Connect the block to the **function_call_100ms** input port.
2. Add a second Function-Call Generator block. Set **Sample time** to `0.01`. Connect the block to the **function_call_10ms** input port.



Scheduling Restrictions for Referenced Export-Function Models

If a test model references an export-function model, there are some restrictions to ensure consistency with simulation results.

For the test model:

- You cannot use two Function-Call Generator blocks with the same sample time.
- Function-calls to the input ports on the Model block must follow the execution order of the root-level function-call Inport blocks in the referenced export-function model. Function-Call Generator blocks with smaller sample times execute first.

  If the test model calls the referenced model functions out of order at any time step, Simulink displays an error. For information on sorted execution order, see “Control and Display Execution Order” on page 36-25. To disable this restriction, clear the check box for the configuration parameter **Enable strict scheduling checks for referenced model**.
- You can use a Mux block to connect signals from the Function-Call Generator blocks with different sample times before connecting them to the referenced export-function model. In the Configuration Parameters dialog box, clear the check box for the parameter **Treat each discrete rate as a separate task**.

For the export-function model:

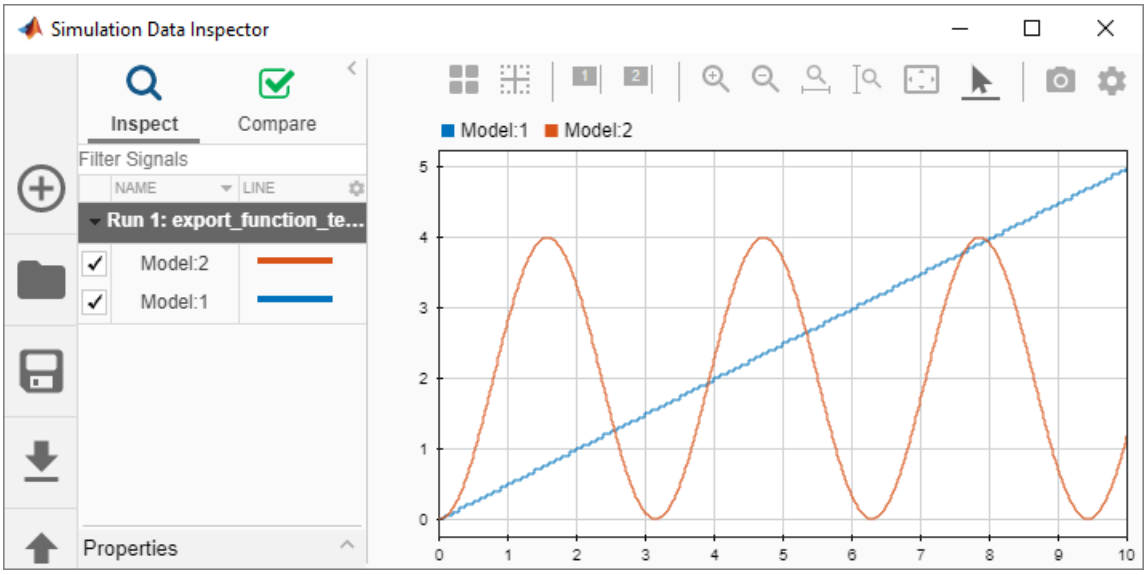
- The sample times for the root-level function-call Inport blocks must be set to inherited (`-1`) or match the sample time of the Function-Call Generator blocks that drive them.

Simulate Export Function Model

Simulate the export-function model to test and observe its behavior before generating code.

Note Simulink does not simulate preempting function-calls.

1. Set configuration parameters for the test model. On the **Modeling** tab and from the **Setup** section, select **Model Settings**. Select the Model Referencing pane. Clear the check box for the configuration parameter **Enable strict scheduling check for referenced models**.
2. Verify the configuration parameters for Solver Type is set to `Fixed-step`, Solver set to `discrete (no continuous states)`, and Fixed-step size (fundamental sample time) set to `auto`.
3. Set up logging of signals. Right-click output port signals and select **Log selected signal**.
4. Run simulation.
5. Open the Simulation Data Inspector by clicking the toolstrip icon .



After you test your model, you can generate code for the functions. See “Generate Code for Export-Function Model” on page 10-90.

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86
- “Generate Code for Export-Function Model” on page 10-90

Test Export-Function Model Simulation Using Stateflow Chart

Use a Stateflow chart to provide a function-call scheduler where you can fully control the scheduling process for periodic (synchronous) or aperiodic (asynchronous) call sequences.

- Create a new Simulink model.
- Add a Model block that references the export-function model.
- Specify function-call Inputs using a Stateflow chart.
- Specify data inputs.
- Run simulation.

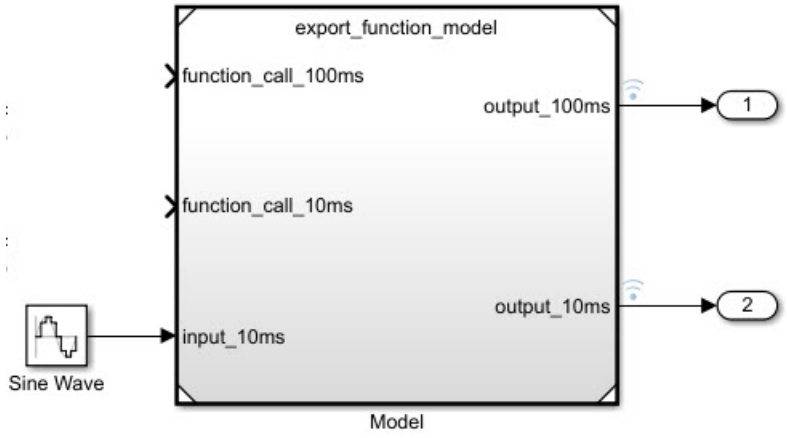
To create the model for this example, see “Create an Export-Function Model” on page 10-72.

To open a completed test model, see `ex_export_function_test_model_with_stateflow_chart`.

Create Referenced Export-Function Model

Referencing an export-function model from a Model block allows the addition of function-calls events from a Stateflow chart and the logging of data signals for testing without changing the model itself.

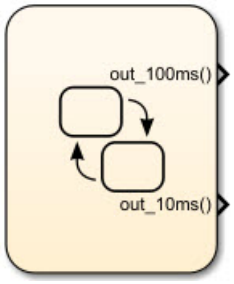
1. Add a Model block to a new Simulink model. In the **Model name** box, enter `export_function_model`.
2. Add Outport blocks to the **output_100ms** and **output_10ms** ports for saving simulation data to MATLAB.
3. Add a Sine Wave block to provide data input. Set **Amplitude** to `2` and **Sample time** to `0.01`. Connect the block to the **input_10ms** input port on the Model block.



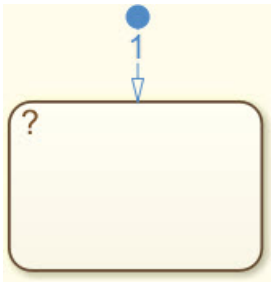
Create Periodic Scheduler Using Stateflow Chart

1. Create a new Stateflow chart. This step requires a Stateflow license.
2. On the **Modeling** tab and from the **Design** section, select **Model Explorer**. In the **Model Hierarchy** pane, select `Chart`.

3 Add function-call events with output ports to the chart. From the menu, select **Add** > **Event**. In the **Name** box, enter `out_100ms`. From the **Scope** list, select `Output to Simulink`. Repeat step to create a function-call event and output port for `out_10ms`.



4 Open the chart by double-clicking the block. Add a State block and a Default transition arrow.

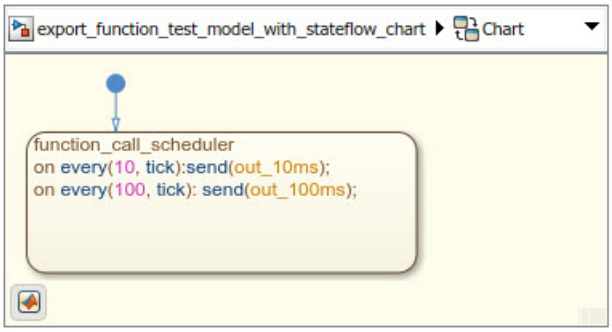


5 Rename the state to `function_call_scheduler`.

6 Add periodic function-calls every 10 and 100 milliseconds. In the state block, enter the following commands.

```
on every(10, tick):send(out_10ms);
on every(100, tick):send(out_100ms);
```

The keyword `tick` is an implicit event that counts the number of simulation steps while `send` is an explicit event that outputs a function-call event to the output ports.

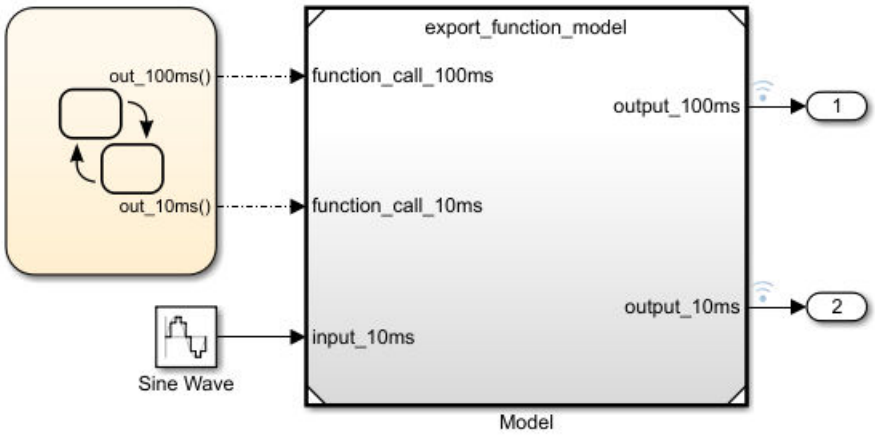


Create Test Model (Harness) for Simulation

You use a Simulink test model only for simulation. After simulation testing, generate code from the export-function model, and then integrate exported function code with an externally coded scheduler.

1 Add a Stateflow chart to your test model.

2 Connect the Stateflow Chart outputs to Model block inputs.



Scheduling Restrictions for Referenced Export-Function Models

If a test model references an export-function model, there are some restrictions to ensure consistency with simulation results.

For the test model:

- Function-calls to the input ports on the Model block must follow the execution order of the root-level function-call Inport blocks in the referenced export-function model.

  If the test model calls the referenced model functions out of order at any time step, Simulink displays an error. For information on sorted execution order, see “Control and Display Execution Order” on page 36-25. To disable this restriction, clear the check box for the configuration parameter **Enable strict scheduling checks for referenced models**.

For the export-function model:

- The sample times for the root-level function-call Inport blocks must be set to inherited (`-1`) or match the sample time of the function-calls from the Stateflow chart that drives them.

Simulate Export Function Model

Simulate the export-function model to test and observe its behavior before generating code.

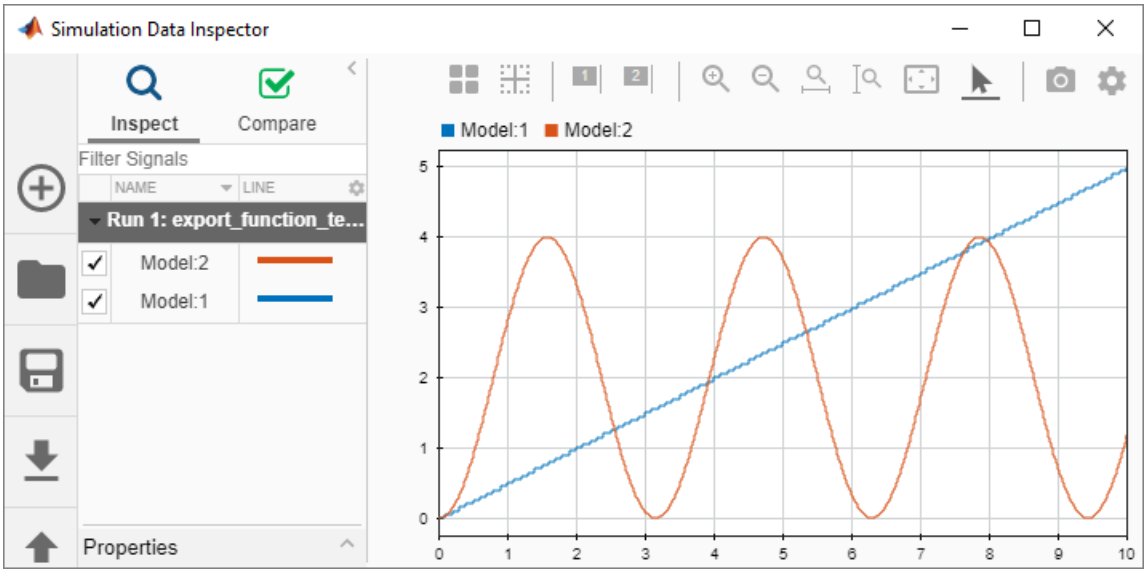
Note When using export-function models in top-model simulations, do not change the enable/disable status of the model during simulation. Enable it at the start of simulation and use function-calls to call it.

---

1 Set configuration parameters for the test model. On the **Modeling** tab and from the **Setup** section, select **Model Settings**. Select the Model Referencing pane. Clear the check box for the configuration parameter **Enable strict scheduling check for referenced models**.

2 Verify the configuration parameters **Solver Type** is set to `Fixed-step`, **Solver** set to `discrete (no continuous states)` and Fixed-step size (fundamental sample time) set to `auto`.

3. Right-click output port signals on the Model block and select **Log selected signal**.
4. Run simulation.
5. Open the Simulation Data Inspector by clicking the icon .



After you test your model, you can generate code for the functions. See “Generate Code for Export-Function Model” on page 10-90.

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86
- “Generate Code for Export-Function Model” on page 10-90

Test Export-Function Model Simulation Using Schedule Editor

Use the Schedule Editor to schedule the function-call components for simulation. This strategy is useful when you want to set the order of execution for function-call components and view data dependencies between components.

- Create a new Simulink model.
- Add a Model block that references the export-function model.
- Specify function-call events using the Schedule Editor.
- Specify data inputs.
- Run a simulation.

To create the model for this example, see “Create an Export-Function Model” on page 10-72.

To open a completed test model, see `ex_export_function_test_model_with_schedule_editor`.

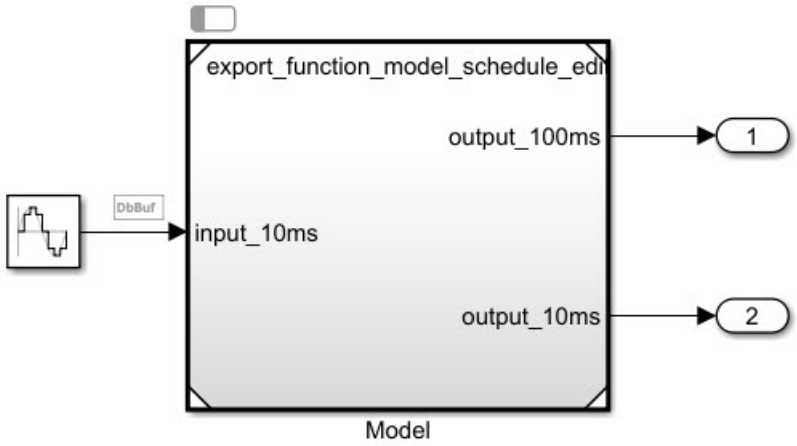
Create Test Model (Harness) for Simulation

A Simulink test model is used only for simulation. After simulation testing, generate code from the export-function model, and then manually integrate exported function code with an externally coded scheduler. Referencing an export-function model from a Model block allows the addition of function-call events and logging of data signals for testing without changing the model itself.

1. Create a new Simulink model.
2. On the **Modeling** tab and from the **Setup** section, select **Model Settings**  .

   In the left pane of the Configuration Parameters dialog box, select Solver. In the right pane, select the Solver details arrow to display additional parameters. Select the check boxes for **Treat each discrete rate as a separate task** and **Automatically handle rate transition for data transfer**.
3. Add a Model block to your model
4. On the **Modeling** tab, select the **Design** section, and then select **Property Inspector**  .

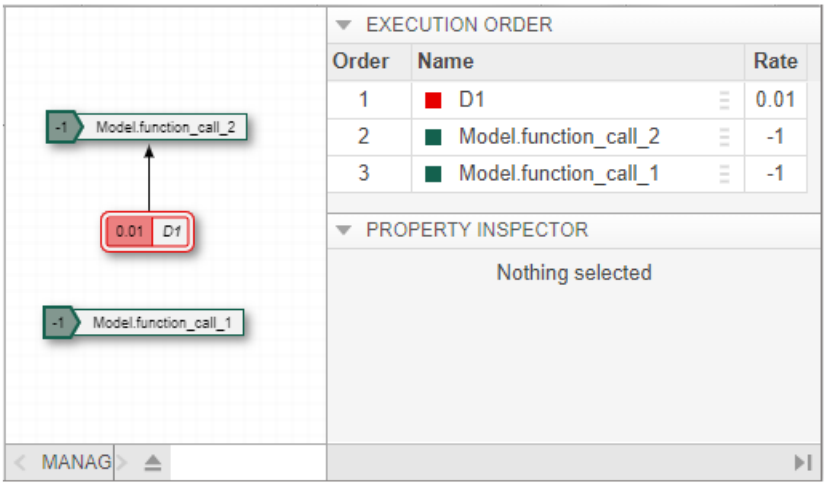
   In the **Model name** box, enter `export_function_model`. Select the **Schedule rates** check box. From the **Schedule rates** drop-down list, select `Schedule Editor`.
5. Add Outport blocks to the **output_100ms** and **output_10ms** ports on the Model block.
6. Add a Sine Wave block to provide data input. Set **Amplitude** to `2` and **Sample time** to `0.01`. Connect the block to the **input_10ms** input port on the Model block.



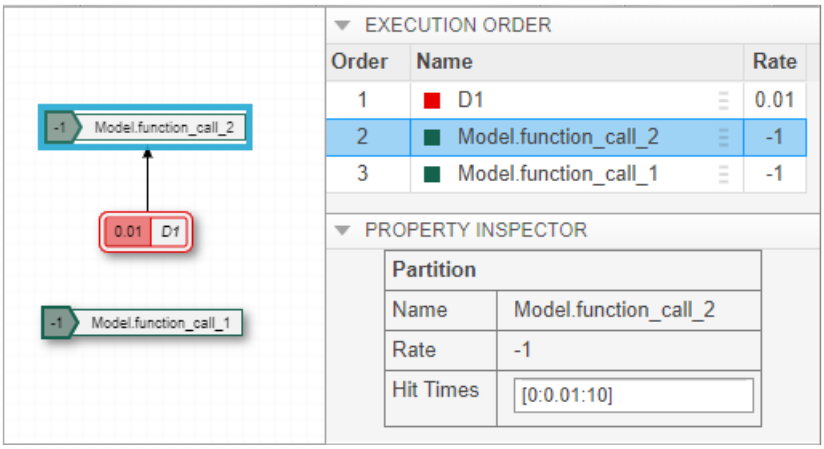
Create Function-Call Events Using the Schedule Editor

Use the Schedule Editor to provide function-call events by defining time-vectors that indicate when events occur for root-level function-call Inport blocks

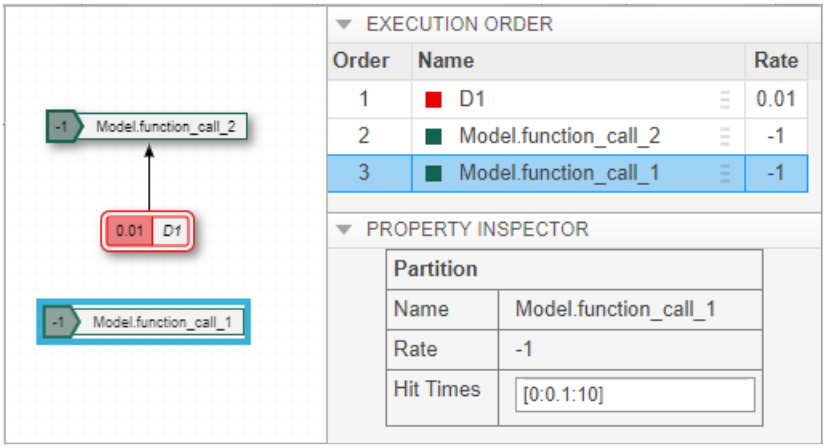
1. Open the Schedule Editor. On the **Modeling** tab and from the **Design** section, select **Schedule Editor**. The Schedule Editor partitions the function-call Inport blocks and names the partitions using the block names.



2. Select the function_call_2 partition. In the **Hit Times** box, enter a matrix with values that begin at 0, periodically increase by 0.01 to 10. You could also test asynchronous behavior by entering a matrix with random values that are multiples of `0.01`.



3 Select the function_call_1 partition. In the **Hit Times** box, enter a matrix with values that begin at 0, increase by 0.1 to 10.



Simulate Export Function Model

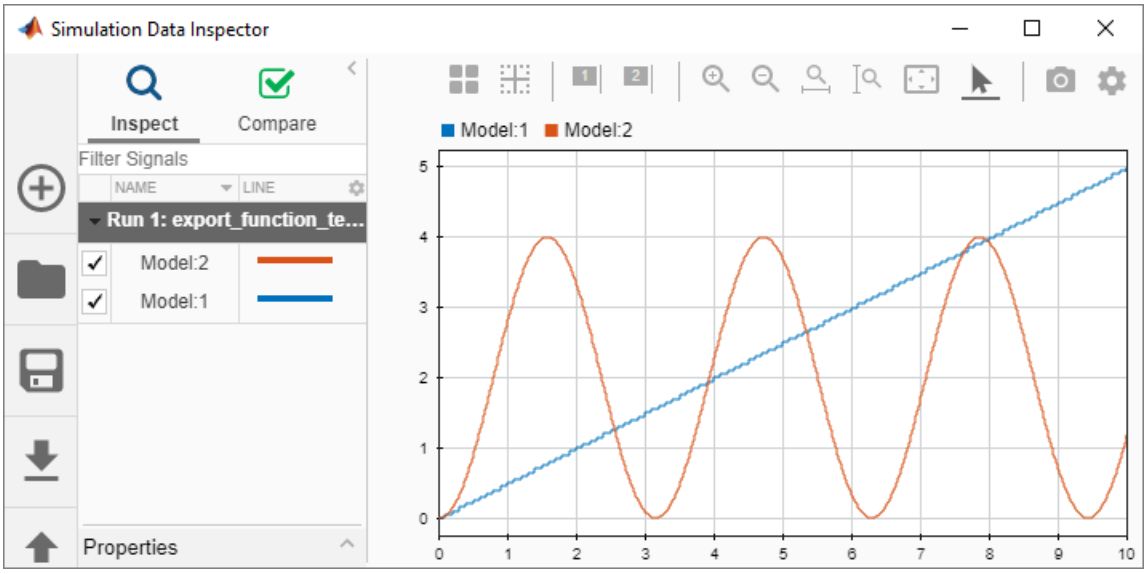
Simulate the export-function model to test and observe its behavior before generating code.

1 Verify the configuration parameters for Solver Type is set to `Fixed-step`, Solver set to `discrete (no continuous states)` or `auto (Automatic solver selected)`, and **Fixed-step size (fundamental sample time)** is set to `auto`.

2 Set up logging of data signals. Right-click output port signals and select the **Log selected signal** check box.

3 On the **Simulation** tab, select the run button .

4 Open the Simulation Data Inspector by clicking the toolstrip icon .



After you test your model, you can generate code for the functions. See “Generate Code for Export-Function Model” on page 10-90.

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Using the Schedule Editor” on page 24-11
- “Export-Function Models Overview” on page 10-97
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Generate Code for Export-Function Model” on page 10-90

Generate Code for Export-Function Model

You generate code for independent functions from an export-function model, not the simulation test model (harness). After generating the function code, you can integrate the functions with a scheduler that you hand-code externally from Simulink.

To create the model for this example, see “Create an Export-Function Model” on page 10-72.

Generate Code for Exported Functions

Generate code from the model that contains the functions.

1. Open an export-function model.
2. On the **Simulation** tab and from the **Prepare** section, select **Model Settings**.

   In the Solver pane, set **Solver Type** to `Fixed-step`. In the Code generation pane, set **System target file** to `ert.tlc`. Requires an Embedded Coder license.
3. Display the **C Code** tab by selecting the **Apps** tab, and then in the **Apps** section, select **Embedded Coder**. On the **C Code** tab, select **Generate Code**. Wait for the code building process to complete.
4. On the **C Code** tab, select **Open Latest Report**.

Entry-Point Functions

Function: function_call_100ms

Prototype	**void function_call_100ms(void)**
Description	Exported function: *<Root>/function_call_100ms*
Timing	Must be called periodically, every 0.1 seconds
Arguments	None
Return value	None
Header file	export_function_model.h

Function: function_call_10ms

Prototype	**void function_call_10ms(void)**
Description	Exported function: *<Root>/function_call_10ms*
Timing	Must be called periodically, every 0.01 seconds
Arguments	None
Return value	None
Header file	export_function_model.h

In the generated code, each root-level function-call Inport block generates a void-void function. The function name is the name of the output signal from the block. If there is no signal name, then the function name is derived from the name of the block. In this example, the function name was derived from the block name.

File: export_function_model.c

```
36 /* Model step function for TID1 */
37 void function_call_10ms(void)          /* Sample time: [0.01s, 0.0s]
38 {
39   /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor
40    *  SubSystem: '<Root>/Function-Call Subsystem 2'
41    */
42   /* Outport: '<Root>/output_10ms' incorporates:
43    *  Inport: '<Root>/input_10ms'
44    *  Math: '<S2>/Square'
45    */
46   export_function_model_Y.output_10ms = export_function_model_U.input
47     export_function_model_U.input_10ms;
48
49   /* End of Outputs for RootInportFunctionCallGenerator: '<Root>/Root
50 }
51
52 /* Model step function for TID2 */
53 void function_call_100ms(void)         /* Sample time: [0.1s, 0.0s] *
54 {
55   /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor
56    *  SubSystem: '<Root>/Function-Call Subsystem 1'
57    */
58   /* Outport: '<Root>/output_100ms' incorporates:
59    *  UnitDelay: '<S1>/Unit Delay'
60    */
61   export_function_model_Y.output_100ms =
62     export_function_model_DW.UnitDelay_DSTATE;
63
64   /* Bias: '<S1>/Bias' incorporates:
65    *  Outport: '<Root>/output_100ms'
66    *  UnitDelay: '<S1>/Unit Delay'
67    */
68   export_function_model_DW.UnitDelay_DSTATE =
69     export_function_model_Y.output_100ms + 0.05;
70
71   /* End of Outputs for RootInportFunctionCallGenerator: '<Root>/Root
72 }
```

See Also

Blocks
Function-Call Subsystem

Related Examples

- “Export-Function Models Overview” on page 10-97
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86

Generate Code for Export-Function Model with Rate-Based Model

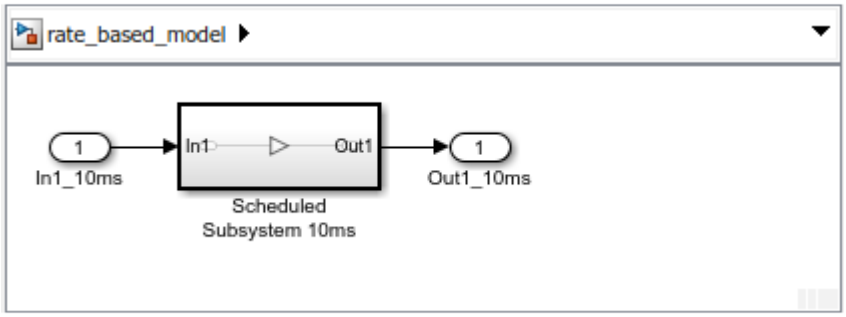
You can generate code from a model that includes both function-call subsystems and scheduled subsystems. Create a rate-based model with scheduled subsystems, and then add the model to an export-function model by reference from a Model block.

To open a completed model, see `ex_export_function_model04`.

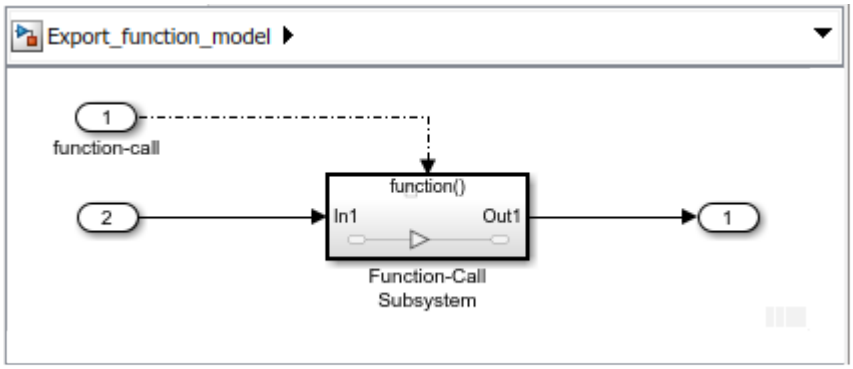
Create Export-Function Model with Scheduled Subsystems

Create a model with function-call and scheduled subsystems.

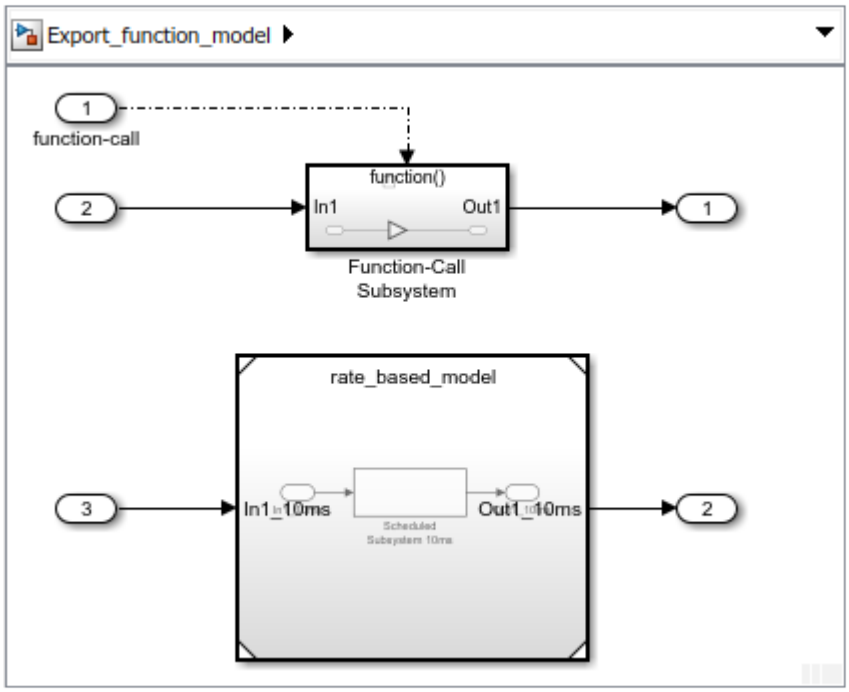
1. Begin by creating a rate-based model with scheduled subsystems. In this example, the **Sample time** for the Scheduled Subsystem is set to `0.01`. See “Create A Rate-Based Model” on page 10-40.



2. Create an export-function model with function-call subsystems. See “Create an Export-Function Model” on page 10-72.



3. Include the rate-based model in the export-function model by reference from a Model block.

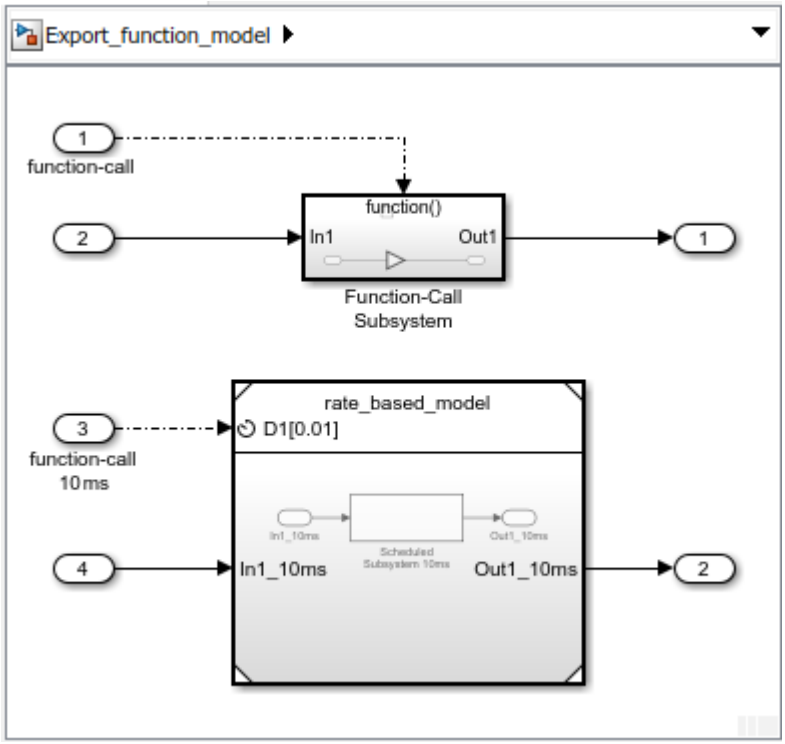


4 Display periodic event ports on the Model block by selecting the **Schedule rates** check box.

   Connect Inport blocks to the periodic event ports.

5 In the Inport block dialog box, select the **Output function-call** parameter check box and specify the Sample time with the same sample time from the scheduled subsystem. In this example, the **Sample time** for the `function-call 10ms` Inport block is set to `0.01`

   .



Generate Code for Exported Functions

Generate code from the model that contains the functions.

1. On the **Simulation** tab and from the **Prepare** section, select **Model Settings**.

   In the Solver pane, set **Solver Type** to `Fixed-step`. In the Code generation pane, set **System target file** to `ert.tlc`. Requires an Embedded Coder license.

2. Display the **C Code** tab by selecting the **Apps** tab, and then in the **Apps** section, select **Embedded Coder**. On the **C Code** tab, select **Generate Code**. Wait for the code building process to complete.

3. On the **C Code** tab, select **Open Latest Report**.

In the generated code, each root-level function-call Inport block generates a void-void function. The function name is the name of the output signal from the block. If there is no signal name, then the function name is derived from the name of the block. In this example, the function name was derived from the block name.

File: Export_function_model.c

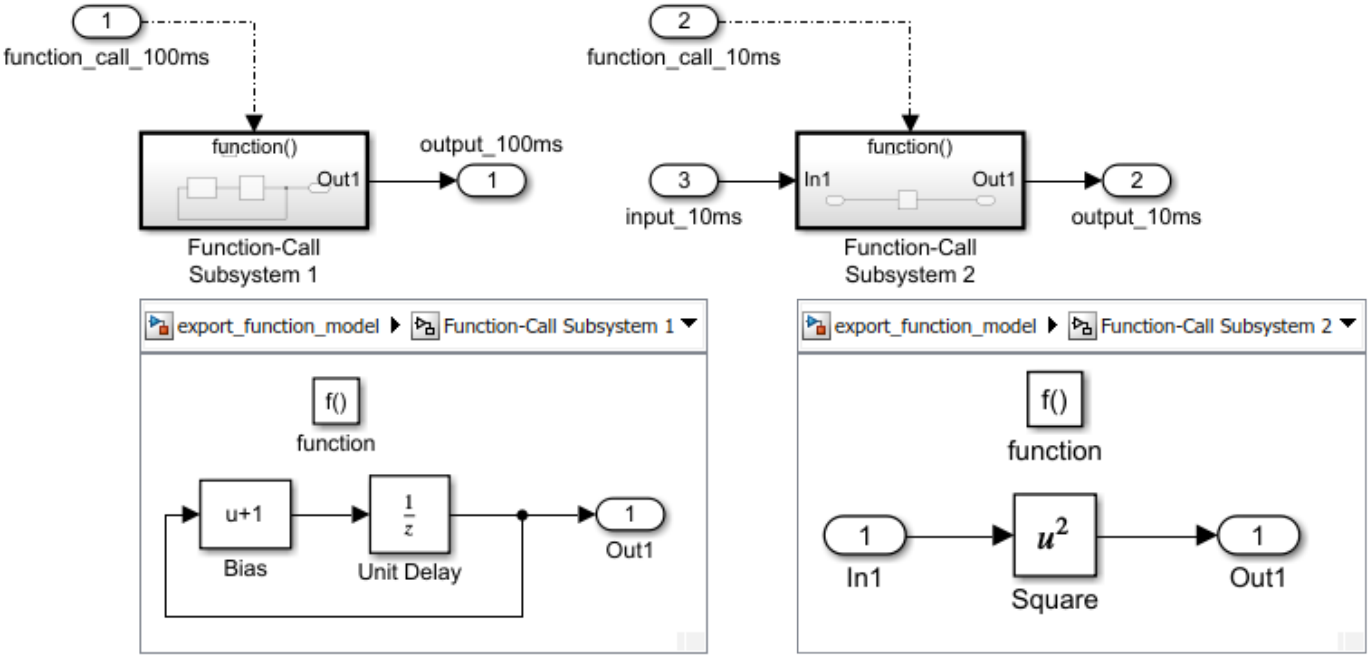
```
36 /* Model step function for TID1 */
37 void function_call_10ms(void)          /* Explicit Task: function_call_10ms */
38 {
39   /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor_function_
40   rate_based_model(&Export_function_model_U.In1, &Export_function_model_Y.Out1)
41
42   /* End of Outputs for RootInportFunctionCallGenerator: '<Root>/RootFcnCall_In
43 }
44
45 /* Output function */
46 void function_call(void)
47 {
48   /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor_function_
49    *  SubSystem: '<Root>/Function-Call Subsystem'
50    */
51   /* Outport: '<Root>/Out2' incorporates:
52    *  Gain: '<S1>/Gain'
53    *  Inport: '<Root>/In4'
54    */
55   Export_function_model_Y.Out2 = 3.0 * Export_function_model_U.In4;
56
57   /* End of Outputs for RootInportFunctionCallGenerator: '<Root>/RootFcnCall_In
58 }
```

Export-Function Models Overview

In this section...

Export-function models are Simulink models that generate code for independent functions that can be integrated with an external environment and scheduler. Functions are defined using Function-Call Subsystem, function-call Model, Simulink Function, and S-Function blocks.

The following export-function model contains two functions defined with Function-Call Subsystem blocks. For a step-by-step procedure to create this model, see “Create an Export-Function Model” on page 10-72.



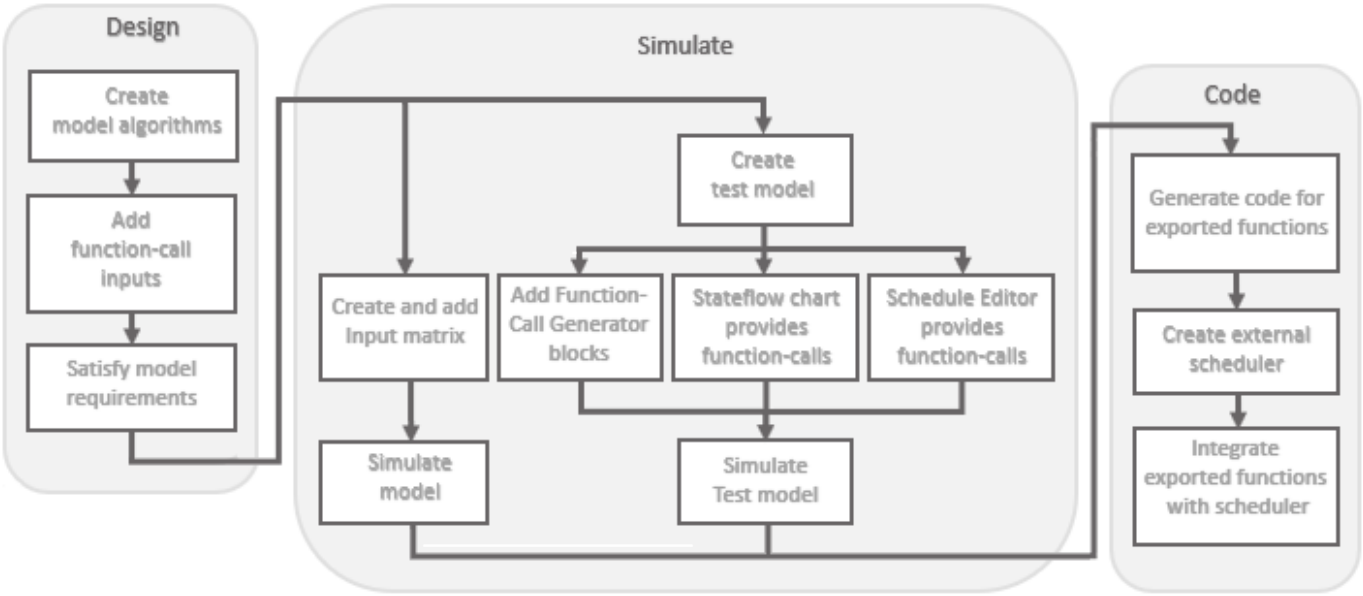
Code generated from this model has two independent functions, one for a delay function and the other for a square function.

```
/*
 * File: export_function_model.c
 * Code generated for Simulink model 'export_function_model'.
 */
```

```
void function_call_100ms(void)         /* Sample time: [0.1s, 0.0s] */
{
  export_function_model_Y.output_100ms =
    export_function_model_DW.UnitDelay_DSTATE;

  export_function_model_DW.UnitDelay_DSTATE =
    export_function_model_Y.output_100ms + 1.0;
}

  void function_call_10ms(void)          /* Sample time: [0.01s, 0.0s] */
{
    export_function_model_Y.output_10ms = export_function_model_U.input_10ms *
    export_function_model_U.input_10ms;
}
```

Workflows for Export-Function Models

Four common processes for creating export-function models differ in how you simulate and test your model before generating code for the functions.



When function-call sequencing is simple enough to be specified as a model input, simulation using an input matrix is the preferred method for testing an export-function model. See “Test Export-Function Model Simulation Using Input Matrix” on page 10-75.

When function-call sequencing is too complicated to specify with an input matrix, create a test model (harness) to mimic the target environment behavior. Use this test model to provide function-call inputs to the export-function model. See “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79, “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86 and “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82.

Allowed Blocks

At the top-level, an export-function model is limited to the following blocks:

- Inport

- Outport
- Goto
- From
- Function-Call Subsystem
- Function-call Model
- Function-Call Split
- Simulink Function
- Initialize Function
- Reset Function
- Terminate Function
- Data Store Memory
- Bus Creator
- Bus Selector
- Mux
- Demux
- Merge
- Signal Specification
- S-Function

Requirements for Export Function Models

For an export-function model to successfully generate function code, the following requirements must be met.

Model Configuration Parameters:

- Solver **Type** set to `Fixed-step`.
- **Solver** set to `auto` or `discrete`.
- Code Generation **System target file** set to `ert.tlc`. Selecting `ert.tlc` requires an Embedded Coder license.
- For function-call Model blocks, **Periodic sample time constraint** for the referenced model set to `Ensure sample time independent`.

Root-level function-call Inport blocks:

- **Output function call** check box selected.
- Cannot receive a signal from an Asynchronous Task Specification block.

Root-level data Inport and Outport blocks cannot connect to virtual bus data signals.

Root-level Function-Call Subsystem blocks and function-call Model blocks:

- All internal blocks within the block must support code generation.
- If the Trigger block **Sample time type** is set to:
  - `triggered`, internal blocks must have **Sample time** set to `-1`.

- `periodic`, the root-level function-call Inport block must have its **Sample time** set to a specified discrete time and all internal blocks must have **Sample time** set to `-1` or the specified discrete time.

Sample Time for Function-Call Subsystems

In an export-function model, you can specify sample time for root-level function-call Inport blocks and the Trigger block inside a root-level Function-Call Subsystem block or function-call Model block. The following table shows how to specify these sample times.

Trigger block **Sample time type**	Trigger block **Sample time**	Inport block **Sample time**	Function-call rate during simulation
`Triggered`	Not specified, parameter is inactive.	-1 (inherited)	For simulation, the function-call initiator connected to the Inport block sets the rate of simulation.
		Specified discrete time	Function-call initiator, in test model, connected to Inport block must have a sample time equal to the specified discrete time for the Inport block. For simulation, component executes at the specified discrete rate. If a function-call source uses a different sample, Simulink displays an error message.
`Periodic` Periodic function-call run-time checks apply if the export-function model is referenced from a Model block.	-1 (inherited) or the specified discrete time for the Inport block.	-1 (inherited)	This configuration is not allowed. Simulink displays an error message.
		Specified discrete time.	For simulation, component executes at the specified discrete sample time. If a function-call source uses a different sample time, Simulink displays an error message.

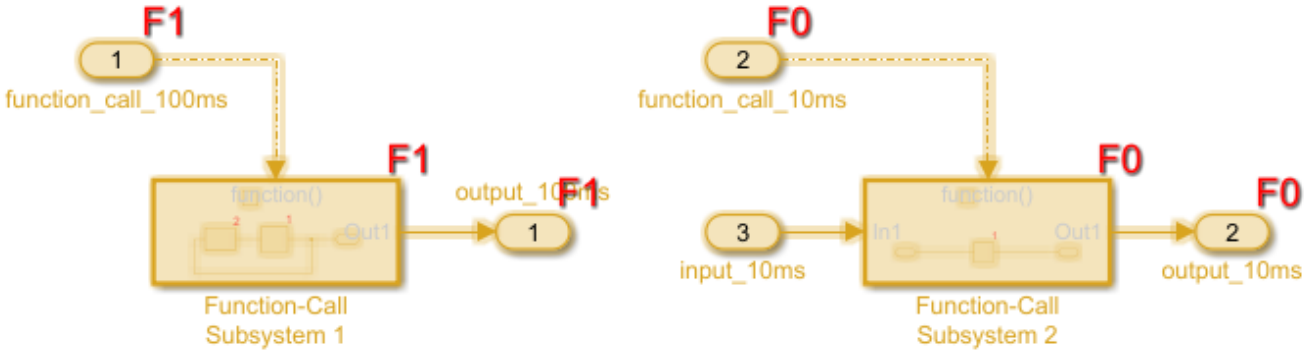
Execution Order for Root-Level Function-Call Inport Blocks

By specifying sample time and priority for function-call Inport blocks you can control the execution order of Function-Call Subsystems and function-call Models during simulation. Alternatively, you can

use the Schedule Editor or a Stateflow chart to test scheduling. See “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86 and “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82.

1. Specify sample time for simulation execution. Right-click a function-call Inport block, then select **Block parameters**.
2. Select the Signal Attributes tab. In the **Sample time** box, enter a discrete time.
3. Specify the block priority for simulation. Right-click a function-call Inport block, then select **Properties**.
4. In the **Priority** box, enter a priority value.
5. Display block execution order for simulation. On the **Debug** tab, select **Information Overlays**  , then from the drop-down dialog, select **Execution Order**. This display has no impact on the generated code.

In the following export-function model, Function-Call Subsystem 2 with **Sample time** for Inport block 2 set to `0.01` (10 ms) runs before Function-Call Subsystem 1 with **Sample time** for Inport block 1 set to `0.1` (100 ms).



Determine Relative Execution Order

Simulink compares function-call Inport block properties to determine their relative execution order using the following rules:

1. Priority – higher priority (lower number) executes first
2. Sample time – smaller sample time executes first
3. Port number – smaller port number executes first

When two blocks have different values for the **Priority** parameter, the block with the higher priority executes first. If the **Priority** parameter is equal, the block with the faster rate (smaller sample time) executes first. If **Priority** and sample time are the same for both of the blocks, the block with the lower port number executes first.

Note When the simulation mode of the top model is Accelerator or Rapid Accelerator, Simulink does not perform run-time simulation checks for the execution order of root-level function-call Inport blocks inside referenced export-function models.

Suppose that an export function model has five root-level function-call Inport blocks, A to E, with block properties as shown in the table. To determine their relative execution order, Simulink compares their **Priority** parameters, sample times (if distinct and non-inherited), and port number.

Root-level function-call Inport block	**A**	**B**	**C**	**D**	**E**
Priority	10	30	40	40	30
Sample Time	–1	0.2	0.1	0.1	–1
Port Number	5	4	3	2	1

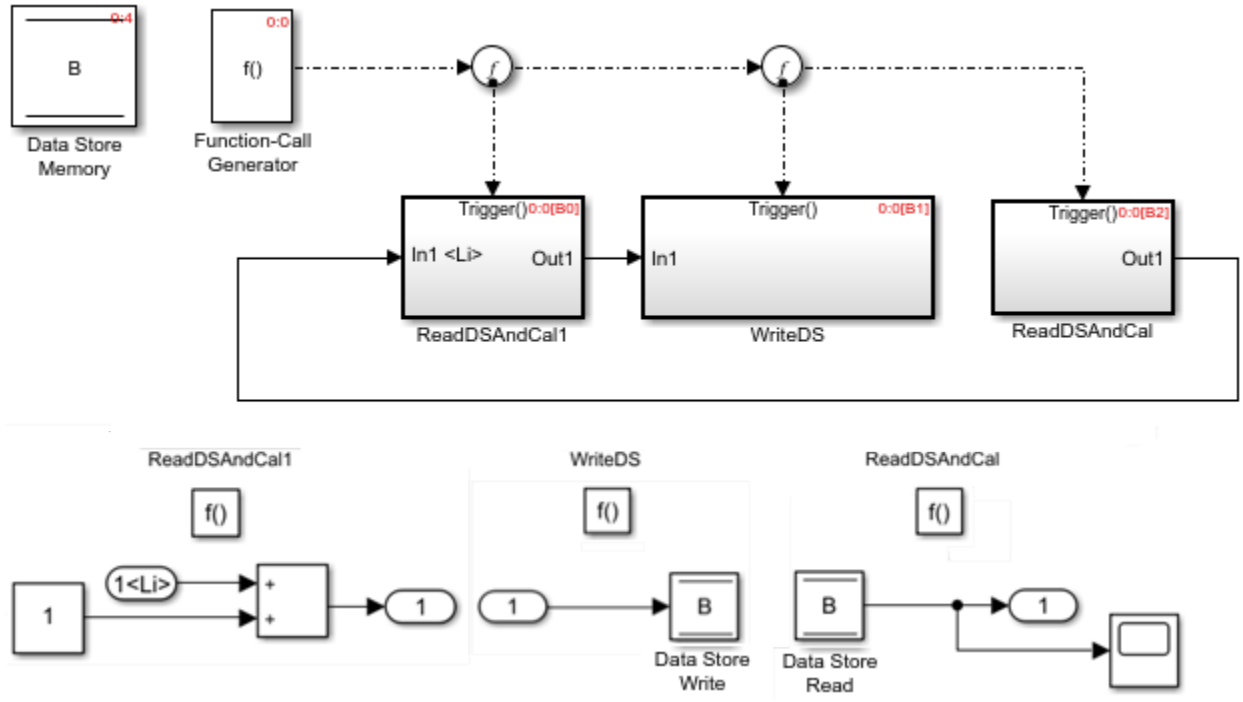
- Block A has the highest priority of all five blocks. A executes first.
- B and E execute after A but before C and D. Since B and E have the same priority, Simulink compares their sample times to determine execution order. E has a sample time of `-1` (inherited), which is smaller than `0.2`, the sample time of B. E executes before B.
- C and D have the same priority and the same distinct, non-inherited sample times. The port number for D (2) is smaller than C (3), D executes before C.

The relative execution order for these function-call Inport blocks is A, E, B, D, and C.

Latched Input Data for Function-Call Subsystems

You can latch input data for Inport blocks within Function-Call Subsystem blocks to ensure data integrity. To latch input data, select the **Latch input for feedback signals of function-call subsystem outputs** check box.

In the following model, input data for the Inport block in the function-call subsystem `ReadDSAndCal1` is latched (indicated by `<Li>`) and cannot change during the execution of the subsystem. The Data Store Read and Data Store Write blocks are called within each function-call subsystem. The first and second function-calls write data and the last function-call reads data to the Data Store Memory block.

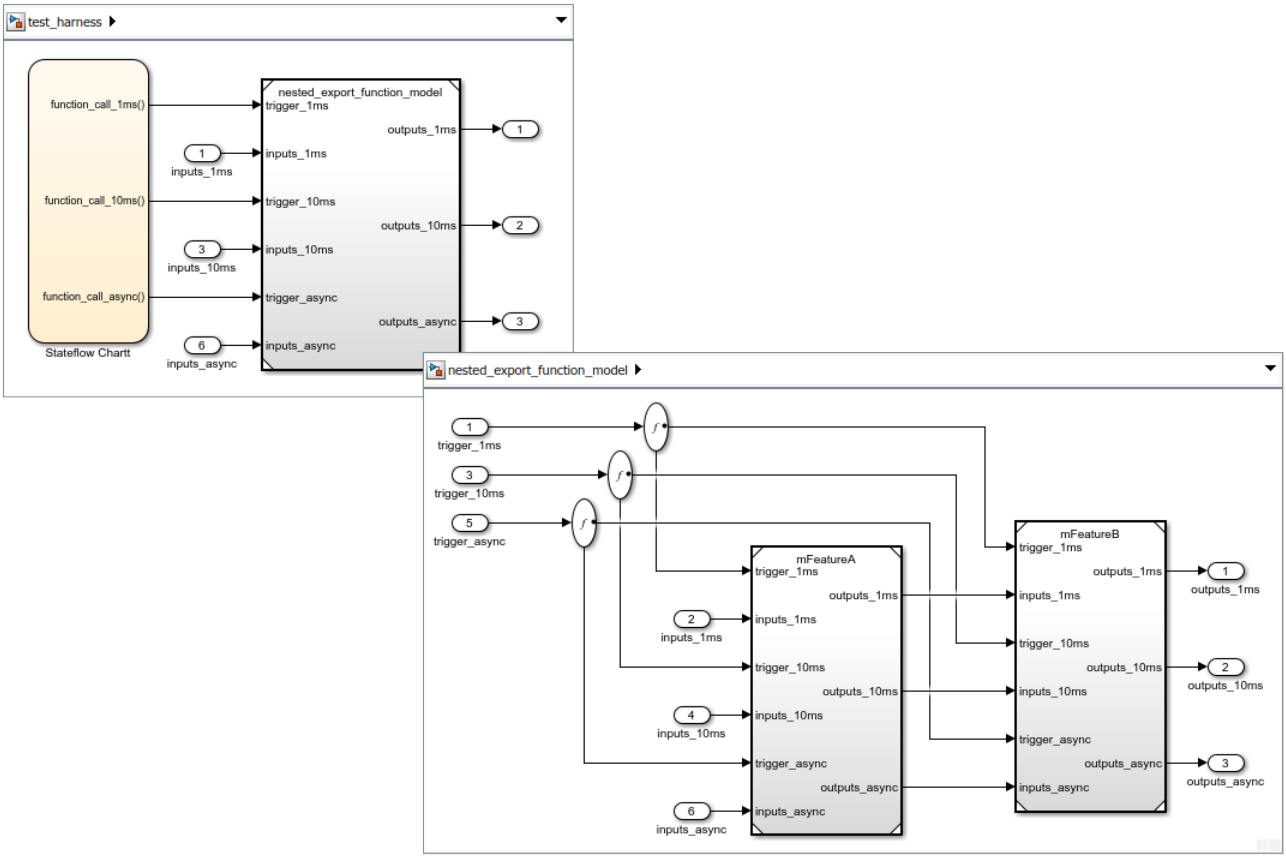


Note The root-level data Inport block connected to an internal Inport block is also latched if all of the blocks connected to the root-level block are latched. For more information, see “Latch input for feedback signals of function-call subsystem outputs”.

Note Data transfer signals are unprotected in the generated code by default. Use custom storage classes to prevent data corruption in these signals due to preemption of the current task in the target environment.

Nested Export-Function Models

Nested export-function models provide an additional layer of organization. The following model has two referenced export-function models that are referenced from a Model block.

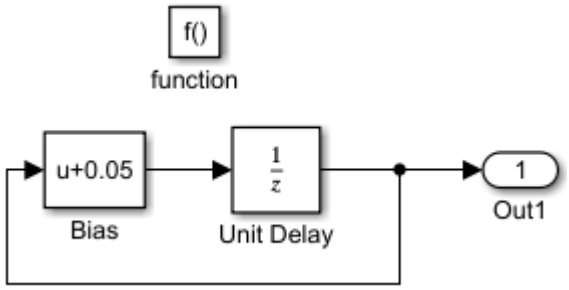


Note An export-function model cannot contain a referenced model with asynchronous function-call inputs, but can contain function-call subsystems and function-call models. A model with asynchronous function-call inputs can contain an export-function model, a function-call subsystem, or a function-call model.

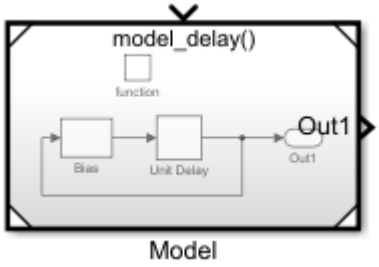
Export-Function Model with a Multi-Instanced Function-Call Model

You can use Function-Call Subsystem blocks or function-call Model blocks within an export-function model. If you use a function-call Model block, you can also create multiple instances of the model.

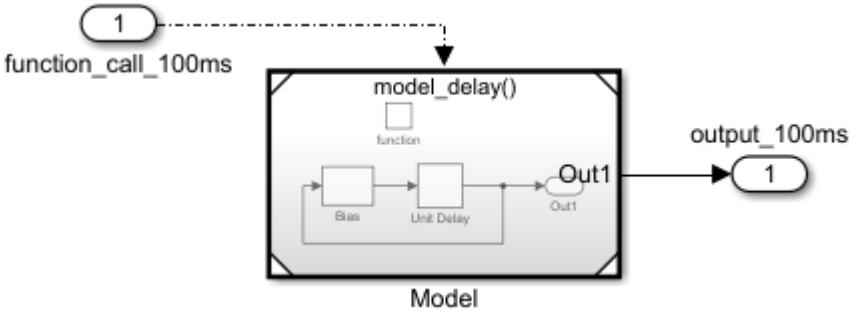
1 Define your algorithm with a model that contains a Trigger block. Set **Trigger type** to `function-call`.



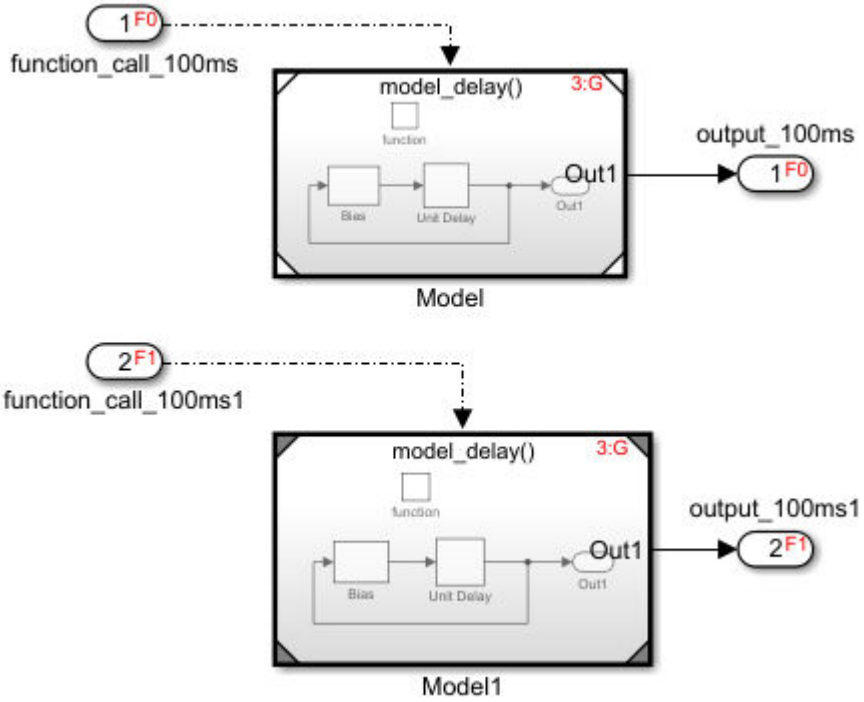
2 Reference the model from a Model block. The result is a function-call model.



3 Connect a function-call Inport block and select the **Output function call** check box. Add signal Inport and Outport blocks. Update the model (Ctrl-D). The result is an export-function model with a function-call model.



4 Copy the referenced model and port blocks to create a second instance of the model. Execution order shows the first instance runs before the second instance.



Export-Function Models and Models with Asynchronous Function-Call Inputs

An export-function models capability is available for models with asynchronous function-call input ports. You use these models primarily in the Simulink environment where the Simulink scheduler calls the functions.

Comparison Characteristic	Export-Function Models	Models with Asynchronous Function-Call Inputs
Definition	These models have root-level function-call Inport blocks that are not connected to an Asynchronous Task Specification block. These Inport blocks trigger function-call subsystems or function-call models (Model block with Trigger block).	These models have root-level function-call Inport blocks connected to Asynchronous Task Specification blocks. These Inport blocks trigger function-call subsystems or function-call models.
Root-level blocks	Only blocks executing in a function-call context are allowed at the root level.	Blocks executing in a non-function-call context are also allowed.
Data transfer	Use data transfer indicators to interpret simulation results. Data transfer in export-function models is not protected by default in generated code. For more details, see “Latched Input Data for Function-Call Subsystems” on page 10-102.	Use Rate Transition blocks to protect data transferred between function-call subsystems running at different rates. For more information, see Rate Transition.
Simulation support	These models support standalone simulation and test model simulation in all simulation modes.	These models support test model simulation in all simulation modes and standalone simulation in Normal, Accelerator, and Rapid Accelerator modes.
Code generation support	Top-model and standalone code generation are supported.	Top-model and standalone code generation are supported.

See Also

Blocks
Function-Call Subsystem | Model | Trigger

Related Examples

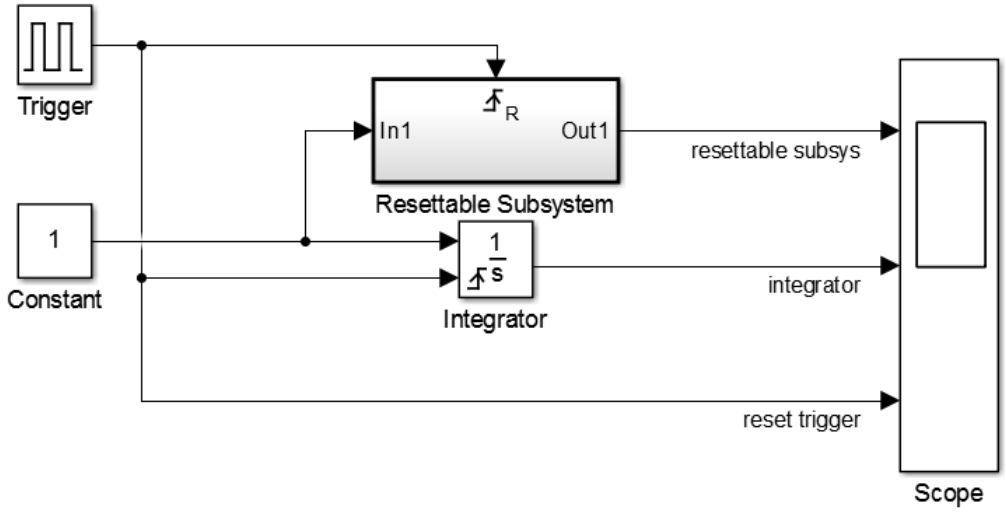
- “Create an Export-Function Model” on page 10-72
- “Test Export-Function Model Simulation Using Input Matrix” on page 10-75
- “Test Export-Function Model Simulation Using Function-Call Generators” on page 10-79
- “Test Export-Function Model Simulation Using Stateflow Chart” on page 10-82
- “Test Export-Function Model Simulation Using Schedule Editor” on page 10-86
- “Generate Code for Export-Function Model” on page 10-90

Use Resettable Subsystems

In this section...

Behavior of Resettable Subsystems

Use resettable subsystems when you want to conditionally reset the states of all blocks within a subsystem to their initial condition. A resettable subsystem executes at every time step but conditionally resets the states of blocks within it when a trigger signal occurs at the reset port. This behavior is similar to the reset behavior of blocks with reset ports, except that a resettable subsystem resets the states of all blocks inside it.

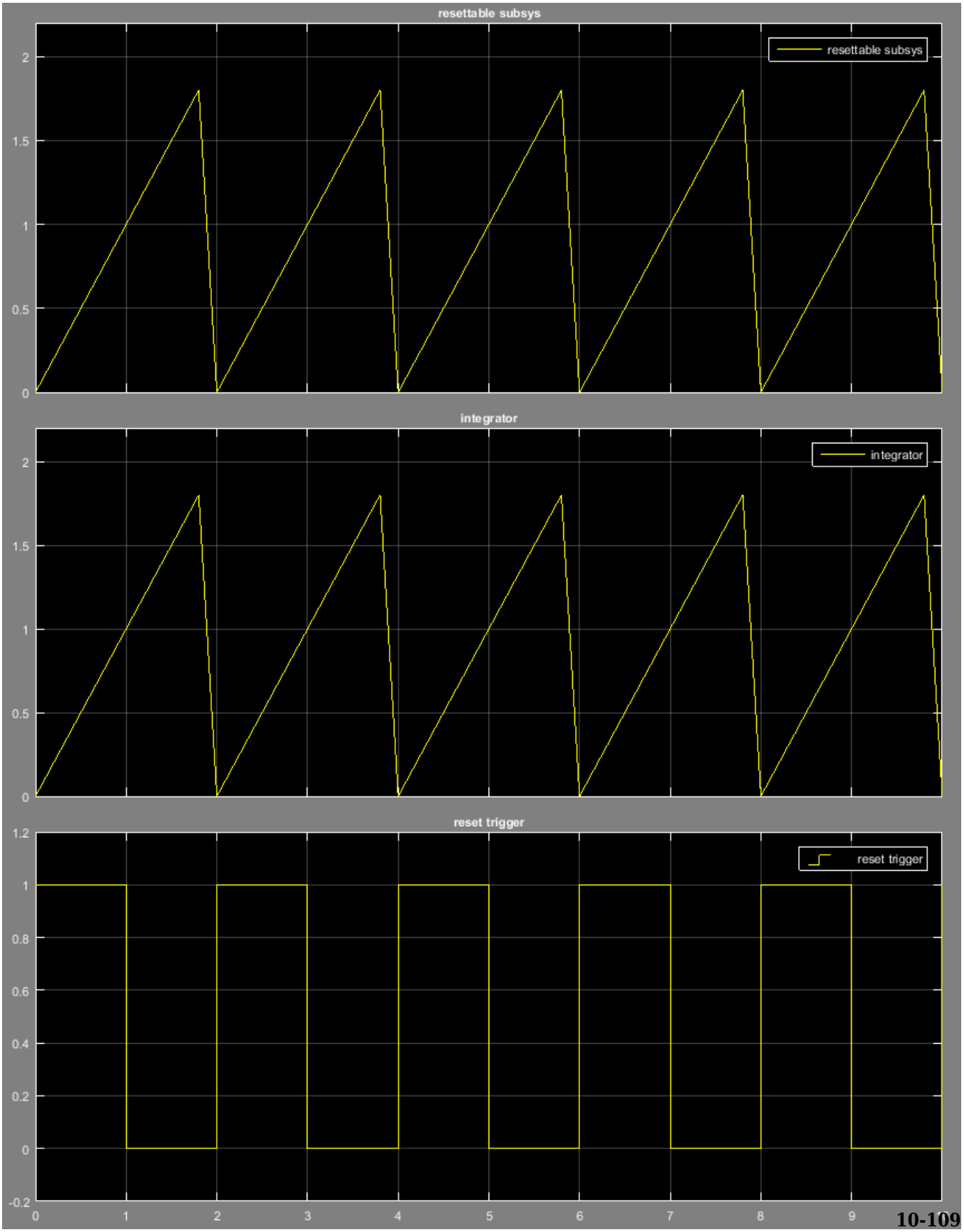


Using resettable subsystems over other methods of resetting states of your block or subsystem has these advantages:

- When you want to reset the states of multiple blocks in a subsystem, displaying and connecting the reset port of each block is cumbersome and makes the block diagram hard to read. Instead, place all the blocks in a resettable subsystem and configure the Reset block in the subsystem.
- Some blocks, such as the Discrete State-Space block, have states but do not have reset ports. You cannot reset these blocks individually, and you must reset the subsystem they are inside. In such cases, it is useful to place these blocks in a resettable subsystem.
- You can also reset blocks in enabled subsystems by setting the **States when enabling** parameter on the enable port to `reset`. However, for this behavior, you must disable the subsystem and then reenable it at a later time step. To reset your block states at the same time step, use resettable subsystems. For more information, see “Comparison of Resettable Subsystems and Enabled Subsystems” on page 10-110.

All blocks in a resettable subsystem must have the same sample time, and they execute at every sample time hit of the subsystem. Resettable subsystems and the model use a common clock.

This model shows that the behavior of block reset ports and resettable subsystems is the same. A resettable subsystem enables you to reset the states of all blocks inside it. The resettable subsystem contains an integrator block that is configured similar to the root-level Integrator block, but the block does not have a reset port. The subsystem resets the states of the integrator block inside it in the same manner as the reset port of the Integrator block. You can see this behavior by running the model and viewing the output in the scope.

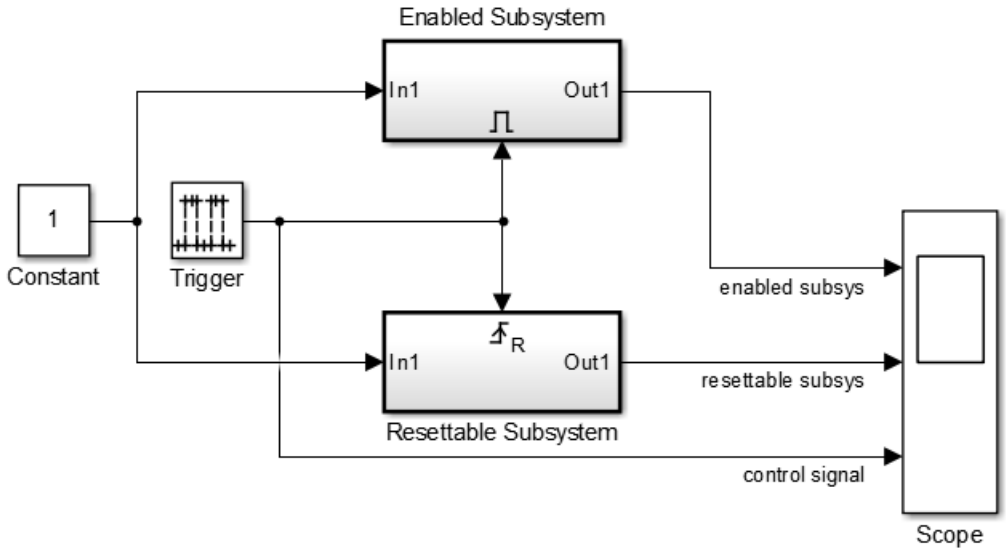

resettable subsys
resettable subsys
2
1.5
1
0.5
0
integrator
integrator
2
1.5
1
0.5
0
reset trigger
reset trigger
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
1
2
3
4
5
6
7
8
9

Comparison of Resettable Subsystems and Enabled Subsystems

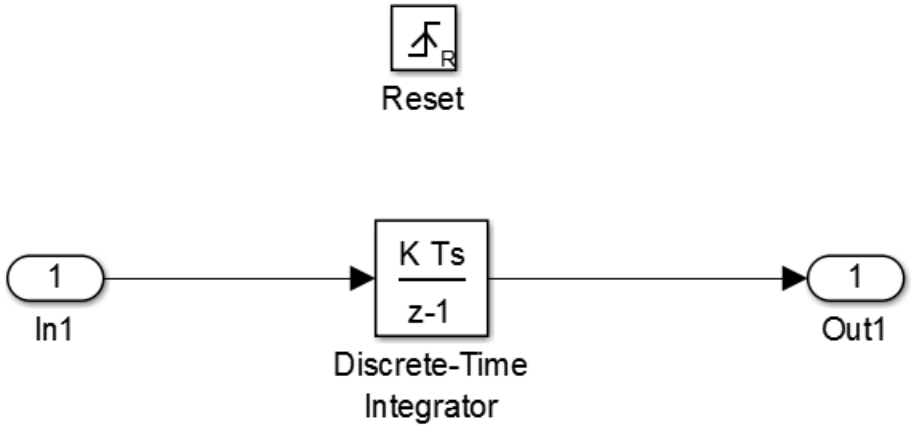
If you set **States when enabling** for the Enable block to `reset`, the enabled subsystem resets the states of all blocks in the subsystem. However, you must disable the subsystem for at least one time step and then reenable it for the states to reset.

In contrast, resettable subsystems always execute and reset the states of their blocks instantaneously.

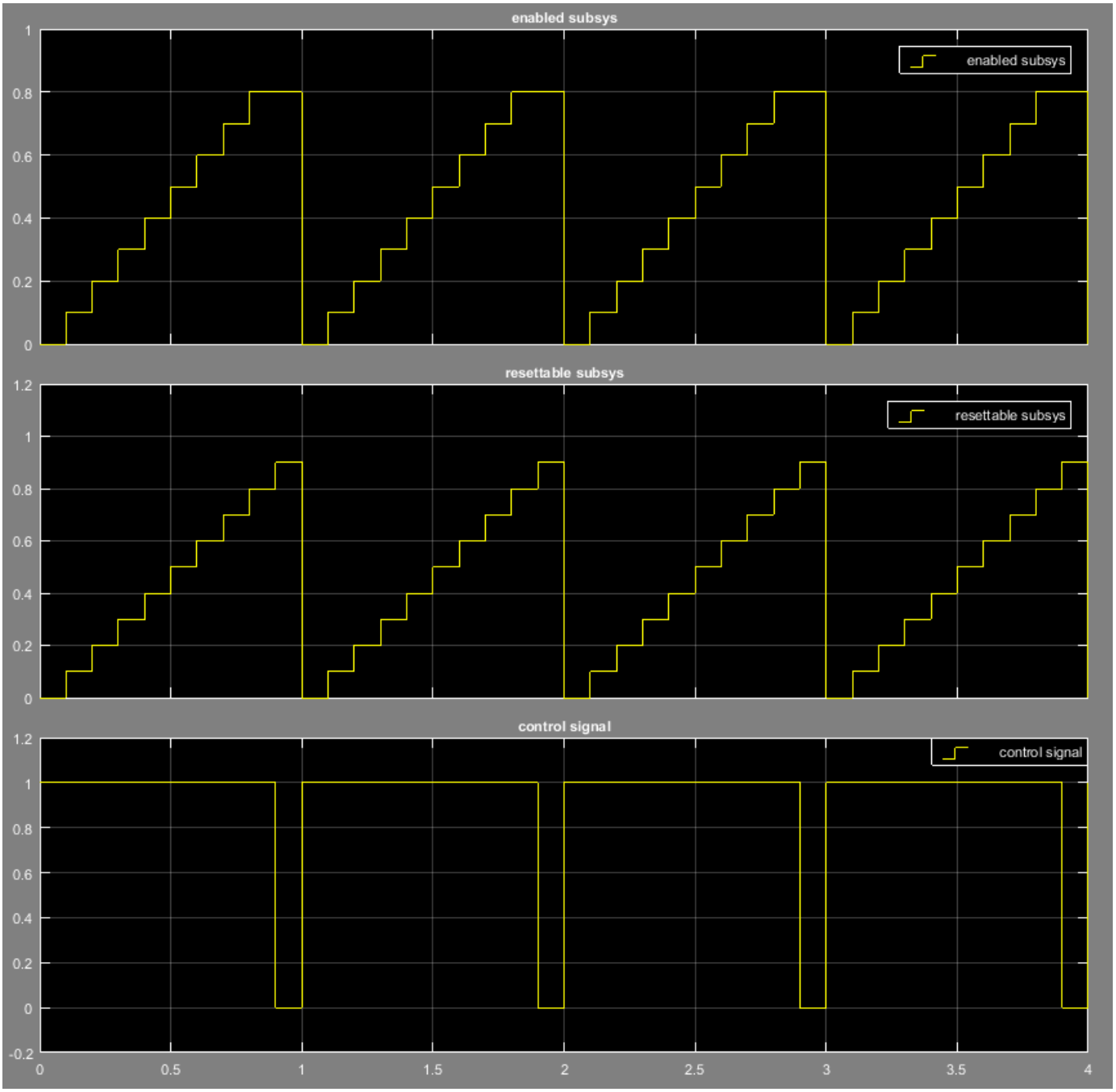
This model shows the difference in the execution behavior of these subsystems. It contains an enabled subsystem and a resettable subsystem whose control ports are connected to pulse generator. The resettable subsystem is set to reset on the rising edge of the control signal, and the enabled subsystem has the **States when enabling** parameter set to `reset` in the enable port.



The subsystems contain identical Discrete-Time Integrator blocks, whose input is the Constant block at the root level of the model. The figure shows the contents of the resettable subsystem.



The figure shows the simulation output.



When the control signal is 0, the enabled subsystem is disabled and the integrator does not change its output while the resettable subsystem is executing. The rising edge of the control signal triggers the reset port of the resettable subsystem and enables the enabled subsystem. Both subsystems reset their states at this time step.

Notice that the enabled subsystem must be disabled for at least one time step before its states can be reset. The resettable subsystem does not have this limitation.

Model Examples

- “Resettable Subsystems”
- “Discrete and Continuous Resettable Subsystems”

See Also

Blocks
Enabled Subsystem | Resettable Subsystem

Related Examples

- “Conditionally Executed Subsystems Overview” on page 10-3
- “Using Enabled Subsystems” on page 10-10

Simulink Functions Overview

In this section...

What Are Simulink Functions?

A Simulink function is a computational unit that calculates a set of outputs when provided with a set of inputs. The function header uses a notation similar to programming languages such as MATLAB and C++. You can define and implement a Simulink function in several ways:

- **Simulink Function block** — Function defined using Simulink blocks within a Simulink Function block.
- **Exported Stateflow graphical function** — Function defined with state transitions within a Stateflow chart, and then exported to a Simulink model.
- **Exported Stateflow MATLAB function** — Function defined with MATLAB language statements within a Stateflow chart, and then exported to a Simulink model.
- **S-function** — Function defined using an S-function block. For an example with an S-function, open `sfcndemo_simulinkfunction_getset`.

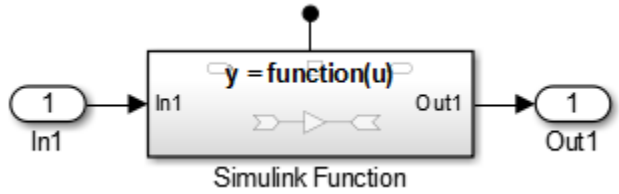
What Are Simulink Function Callers?

A Simulink function caller invokes the execution of a Simulink function from anywhere in a model or chart hierarchy.

- **Function Caller block** — Call a function defined in Simulink or exported from Stateflow. See Function Caller block reference.
- **Stateflow chart transition** — In a Stateflow chart, call a function defined in Simulink or exported from Stateflow.
- **MATLAB Function block** — Call a function from a MATLAB language script.
- **S-Function block** — Call a function using system methods. See `ssDeclareFunctionCaller` and `ssCallSimulinkFunction`.
- **MATLAB System block** — Call a function using a System Object and the MATLAB language.

Connect 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 port blocks to connect and communicate between two Simulink Function blocks or connect to root Inport and Outport blocks that represent external I/O.

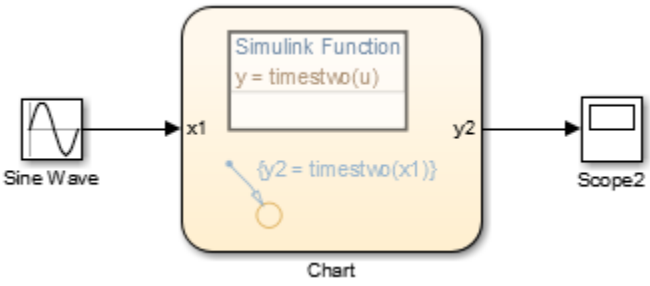


You can also connect the Outport blocks to sink blocks that include logging (To File, To Workspace) and viewing (Scope, Display) blocks. However, these blocks execute last after all other blocks.

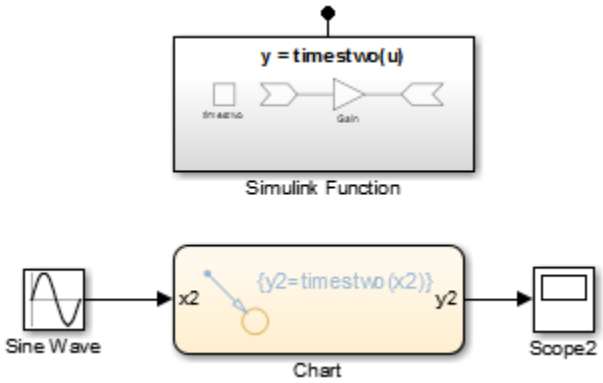
A Simulink Function block can output a function-call event to an Outport block.

Reusable Logic with Functions

Use functions when you need reusable logic across a model hierarchy. Consider an example where a Simulink Function with reusable logic is defined in a Stateflow chart.



You can move the reusable logic from inside the Stateflow chart to a Simulink Function block. The logic is then reusable by function callers in Simulink subsystems (Subsystem and Model blocks) and in Stateflow charts at any level in the model hierarchy.



The result is added flexibility for structuring your model for reuse.

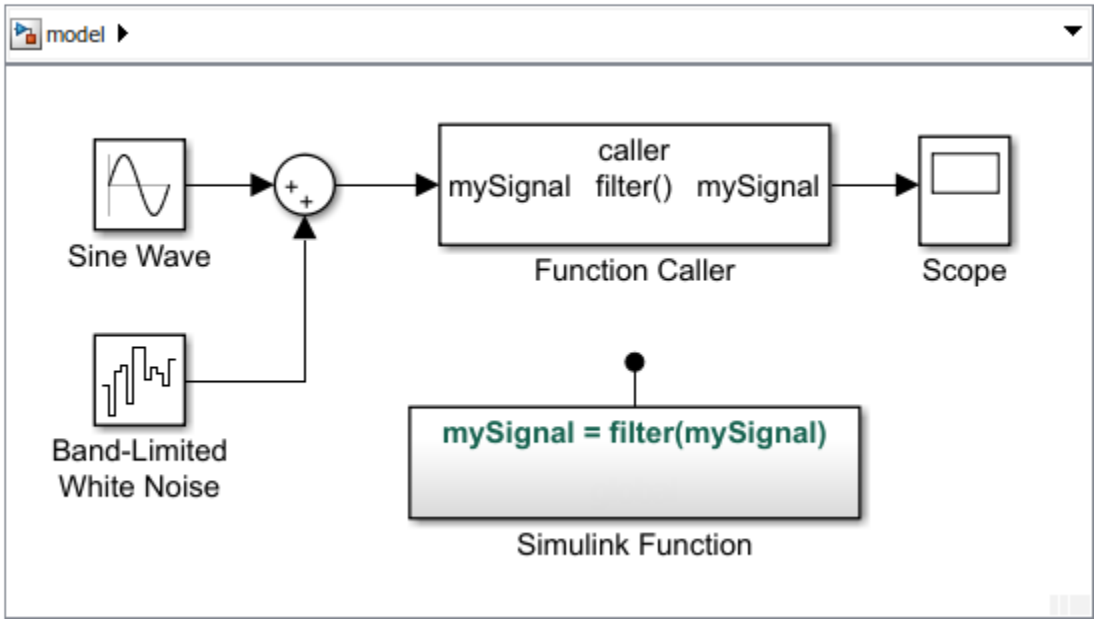
Note Input and output argument names (x2, y2) for calling a function from a Stateflow chart do not have to match the argument names in the function prototype (u, y) of a Simulink Function block.

Input/Output Argument Behavior

The function prototype for a Simulink Function block can have identical input and output arguments. For example, a function that filters noise could input a signal and then return the signal after filtering.

```
mySignal = filter(mySignal)
```

You can call the function with a Function Caller block and add noise to a test signal to verify the function algorithm.

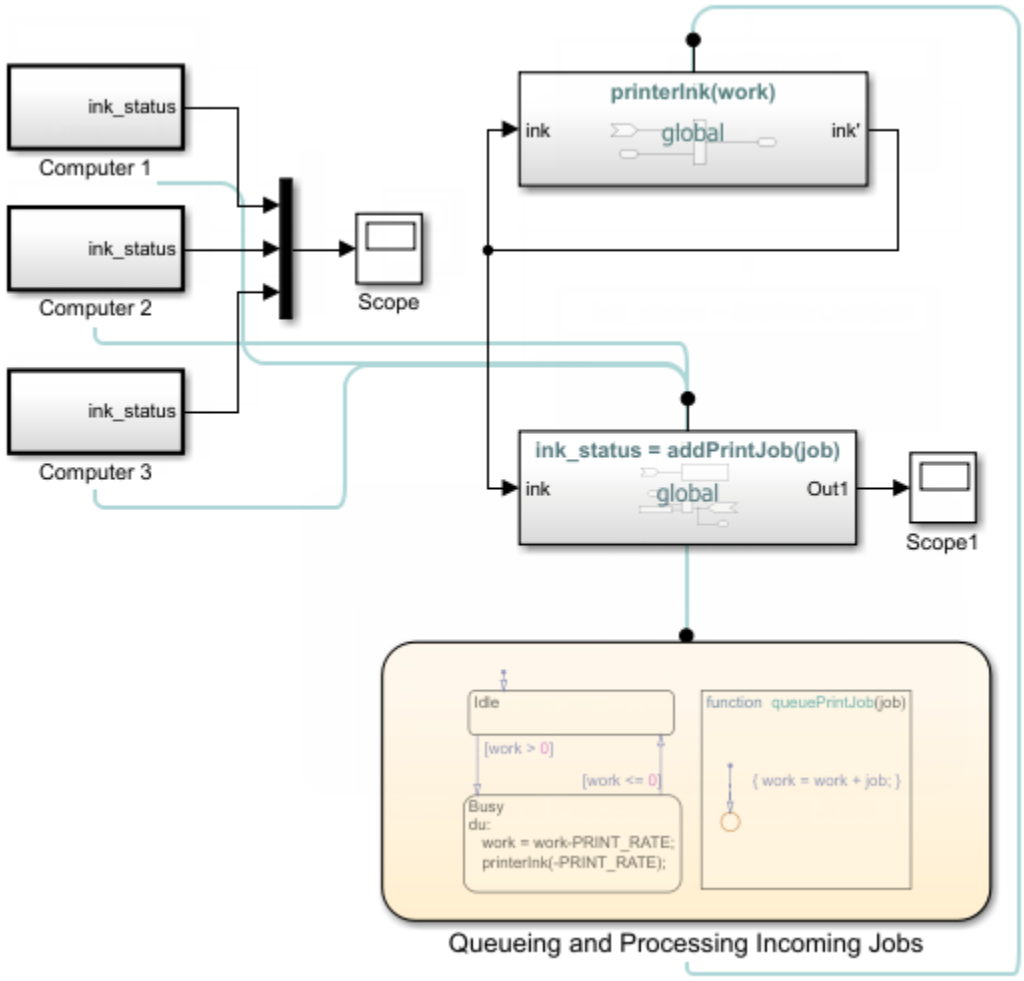


When generating code for this model, the input argument for the Simulink Function block passes a pointer to the signal, not a copy of the signal value.

```
void filter(real_T *rtuy_mySignal)
{
        . . .

        *rtuy_mySignal = model_P.DiscreteFilter_NumCoef * DiscreteFilter_tmp;
    }
```

Shared Resources with Functions

Use functions when you model a shared resource, such as a printer. The model `slexPrinterExample` uses Simulink Function blocks as a common interface between multiple computers and a single Stateflow chart that models a printer process.



How a Function Caller Identifies a Function

The function interface uses MATLAB syntax to define the name of a function and its input and output arguments. The model hierarchy can contain only one function definition with the identified function name. Simulink verifies that:

- The arguments in the **Function prototype** parameter for a Function Caller block matches the arguments specified in the function. For example, a function with two input arguments and one output argument appears as:

  ```
  y = MyFunction(u1, u2)
  ```

- The data type, dimension, and complexity of the arguments must agree. For a Function Caller block, you can set the **Input argument specifications** and **Output argument specifications** parameters, but usually you do not need to specify these parameters manually. Simulink derives the specification from the function.

  The only case where you must specify the argument parameters is when the Function Caller block cannot find the function in the model or in any child model it references. This situation can happen when the Function Caller block and called function are in separate models that are referenced by a common parent model. See “Simulink Function Blocks in Referenced Models” on page 10-140 and “Argument Specification for Simulink Function Blocks” on page 10-136.

Reasons to Use a Simulink Function Block

Function-Call Subsystem blocks with direct signal connections for triggering provide better signal traceability than Simulink Function blocks, but Simulink Function blocks have other advantages.

- **Eliminate routing of signal lines.** The Function Caller block allows you to execute functions defined with a Simulink Function block without a connecting signal line. In addition, functions and their callers can reside in different models or subsystems. This approach eliminates signal routing problems through a hierarchical model structure and allows greater reuse of model components.
- **Use multiple callers to the same function.** Multiple Function Caller blocks or Stateflow charts can call the same function. If the function contains state (e.g., a Unit Delay block), the state is shared between the different callers.
- **Separate function interface from function definition.** Functions separate their interface (input and output arguments) from their implementation. Therefore, you can define a function using a Simulink Function block, an exported graphical function from Stateflow, or an exported MATLAB function from Stateflow. The caller does not need to know how or where the function was implemented.

Choose a Simulink Function or Reusable Subsystem

A consideration for using a Simulink Function block or a Subsystem block has to do with shared state between function calls. A Simulink Function block has shared state while a Subsystem block, even if specified as a reusable function, does not.

- For a Simulink Function block, when one block has multiple callers, code is always generated for one function. If the Simulink Function block contains blocks with state (for example, Delay or Memory), the state is persistent and shared between function callers. In this case, the order of calls is an important consideration.
- For a Subsystem block, when a block has multiple instances and is configured as a reusable function, code is usually generated for one function as an optimization. If the Subsystem block contains blocks with state, code is still generated for one function, but a different state variable is passed to the function. State is not shared between the instances.

When Not to Use a Simulink Function Block

Simulink Function blocks allow you to implement functions graphically, but sometimes using a Simulink Function block is not the best solution.

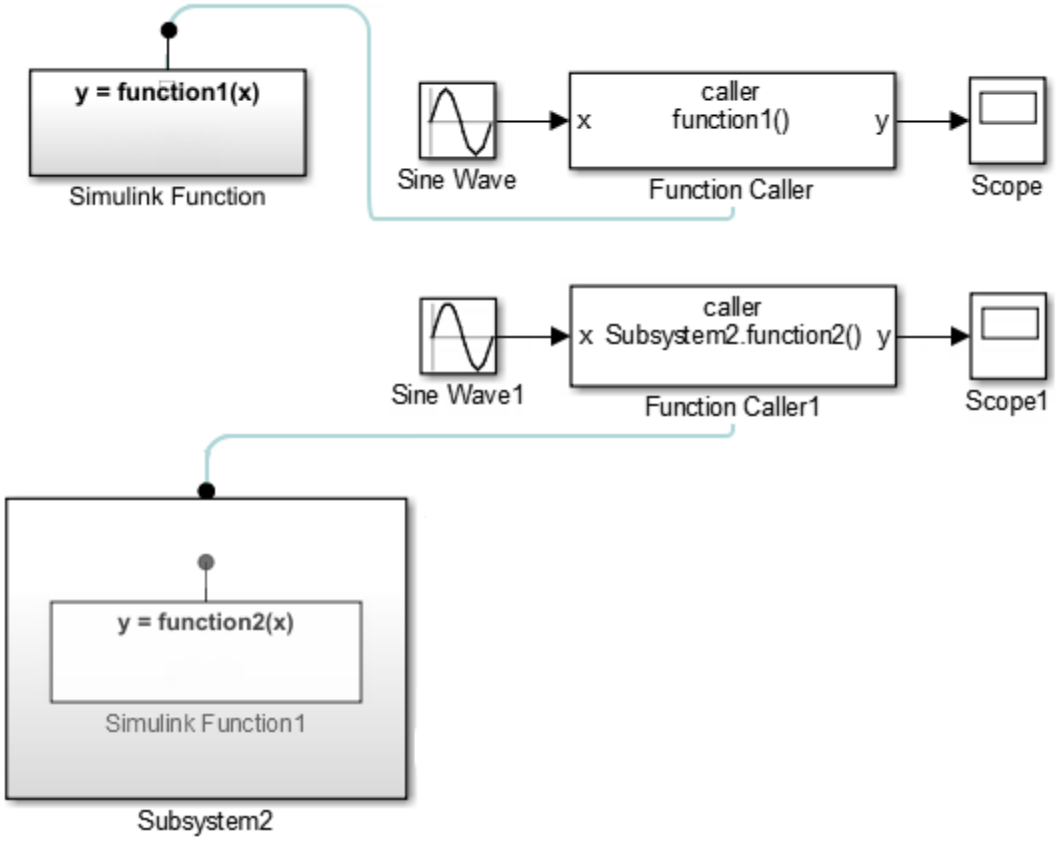
For example, when modeling a PID controller or a digital filter and you have to model the equations defining the dynamic system. Use an S-Function, Subsystem, or Model block to implement systems of equations, but do not use a Simulink Function block, because these conditions can occur:

- **Persistence of state between function calls.** If a Simulink Function block contains any blocks with state (for example, Unit Delay or Memory), then their state values are persistent between calls to the function. If there are multiple calls to that function, the state values are also persistent between the calls originating from different callers.
- **Inheriting continuous sample time.** A Simulink Function block cannot inherit a continuous sample time. Therefore, do not use this block in systems that use continuous sample times to model continuous system equations.

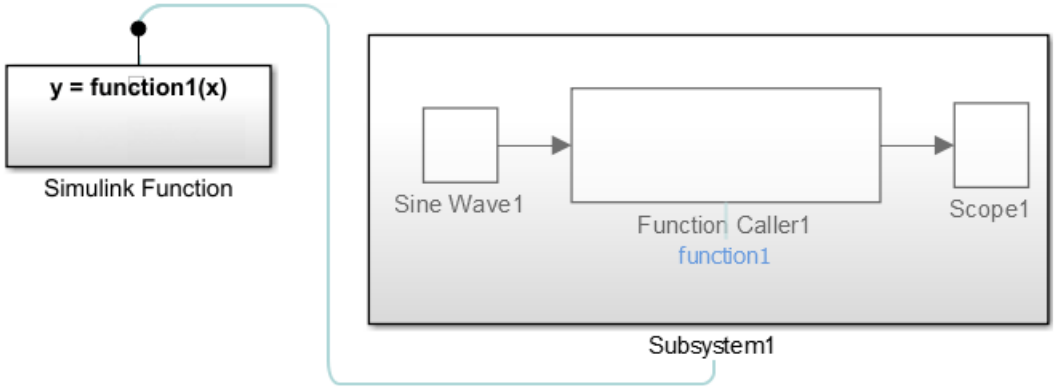
Tracing Simulink Functions

Visually display connections between a Simulink function and their callers with lines that connect callers to functions:

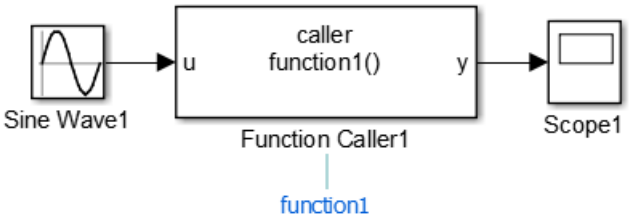
- Turning on/off tracing lines — On the **Debug** tab, select **Information Overlays**  . From the drop-down box, select **Function Connectors**  .
- Direction of tracing lines — Lines connected at the bottom of a block are from a function caller. Lines connected at the top of a block are to a Simulink function or a subsystem containing the function.



- Navigation to functions — A function caller can be within a subsystem.



Navigate from a caller in a subsystem to a function by first opening the subsystem, and then clicking a link to the function.

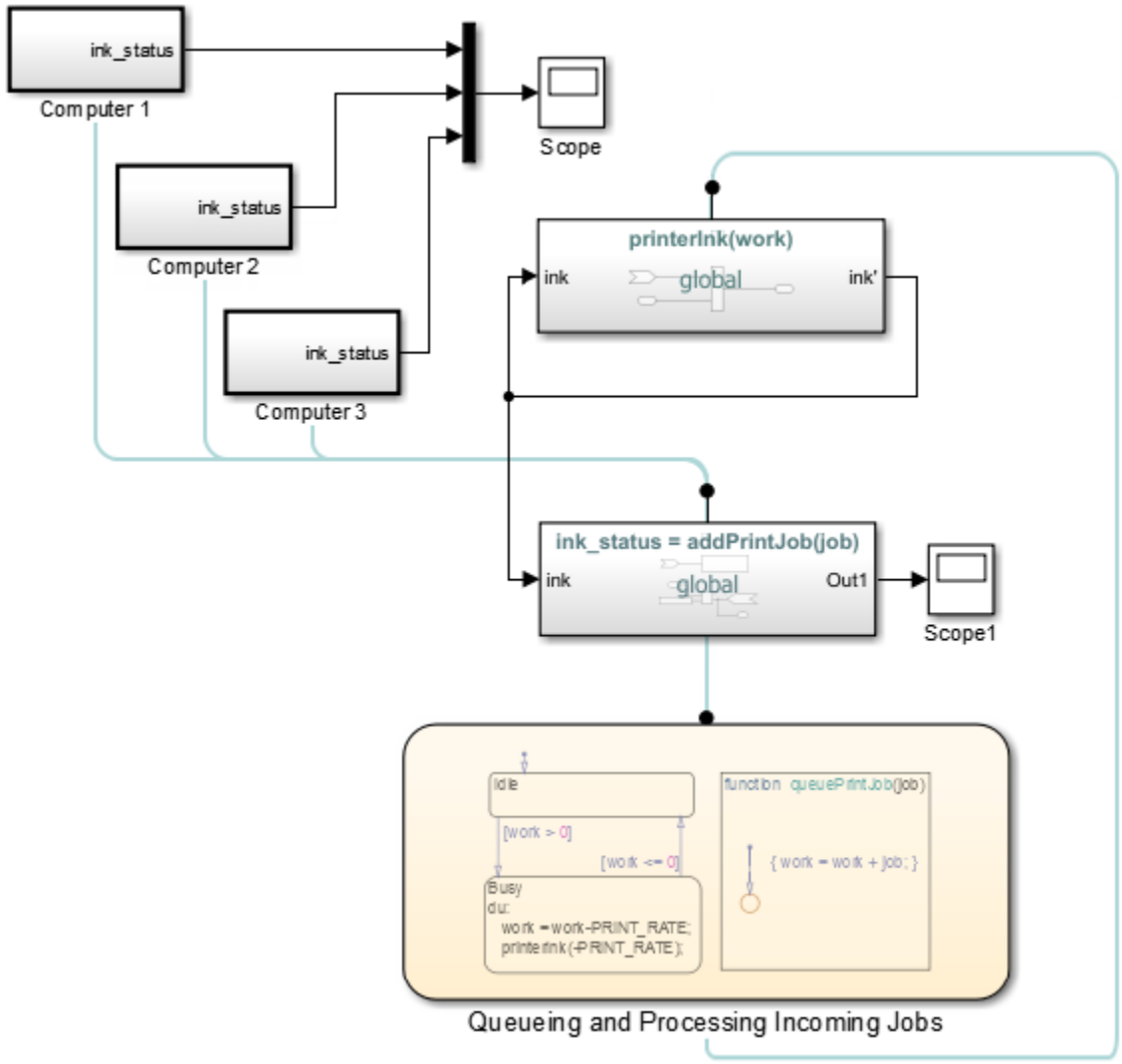


If the function is at the root level of a model, the function opens. If the function is within a subsystem, the subsystem containing the function opens.

Monitor Ink Status on a Shared Printer Using Simulink Functions

After selecting **Function Connectors**, the model `slexPrinterExample` shows the relationships between callers and functions.

In this example, the Function Caller in the Simulink Function block `addPrintJob`, calls the exported Stateflow function `queuePrintJob`. The subchart `Busy` calls the Simulink Function block `printerInk`. Tracing lines are drawn into and out of the Stateflow chart.



See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Scoped Simulink Function Blocks in Models” on page 10-157
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions

In this section...

Simulink functions have an interface with input and output arguments similar to programming languages. You can create the function definition for a Simulink function using:

- Simulink blocks within a Simulink Function block
- Stateflow state transitions in a graphical function exported from a Stateflow chart.
- MATLAB code in a MATLAB function exported from a Stateflow chart.

The following sections show how to create a Simulink function for the function `y = timestwo(x)`. The function multiplies a value (`x`) from a caller by 2, and then sends the calculated value (`y`) back to the caller. To call the function, see “Simulink function callers: Function Caller block, MATLAB Function block, Stateflow chart” on page 10-128.

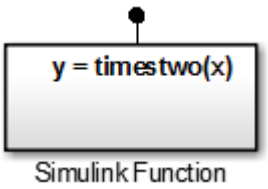
To open completed model with Simulink functions and function callers, see `ex_simulink_functions_and_function_callers`.

Create a Simulink function using a Simulink Function Block

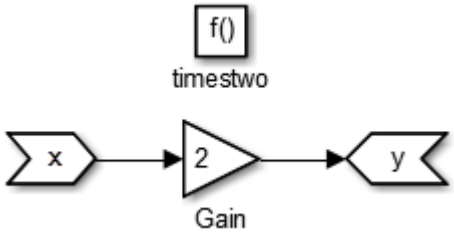
Set up a Simulink Function block to receive data through an input argument from a function caller, and then pass a calculated value back through an output argument.

1. Add a Simulink Function block to your model.
2. On the block face, enter the function prototype.

   ```
   y = timestwo(x)
   ```

3. Double-click the block to open the subsystem defining the function algorithm.
4. Add a Gain block and set the **Gain** parameter to 2.



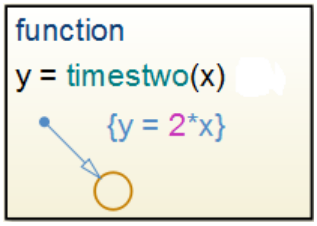
Create a Simulink function using an exported graphical function from a Stateflow chart

Set up a graphical function in a Stateflow chart to receive data through an input argument from a function caller and pass the calculated value back through an output argument. Set chart parameters to export the function to a Simulink model.

Define a graphical function In a Stateflow chart

Create a graphical function in a Stateflow chart. Define the function interface and function definition.

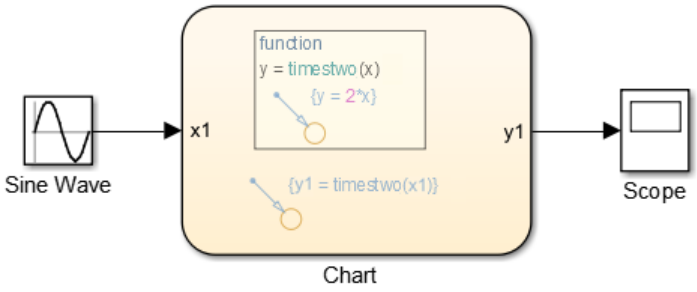
1. Add a Stateflow Chart to your Simulink model. Double-click on the Simulink block diagram. In the search box, enter `chart`, and then from the search results, select `Chart`.
2. Double-click to open the chart.
3. Add a graphical function. From the left-side toolbar, click and drag the graphical function icon onto the chart.
4. Define the function interface. In the function box, replace the ? with the function interface `y = timestwo(x)`.
5. Define the function algorithm. Click the transition arrow and replace the ? with`{y = 2*x}`.



Test the graphical function

Test the graphical function within the Stateflow chart before exporting to a Simulink model.

1. Add a default transition in the chart for testing the function. From the left-side toolbar, click and drag a default transition arrow onto the chart.
2. Double-click the arrow and replace the ? with `{y1 = timestwo(x1)}`.
3. Add an input port to the chart. Open the Model Explorer. In the left pane, select `Chart`. From the menu, select **Add** > **Data**. Set **Name** to `x1` and **Scope** to `Input`.
4. Add an output port to the chart. From the menu, select **Add** > **Data**. Set **Name** to `y1` and **Scope** to `Output`.
5. Add a Sine Wave block to provide test data for the input and a Scope block to view results from the output..



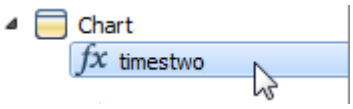
6 Run a simulation.

Set argument parameters for a graphical function

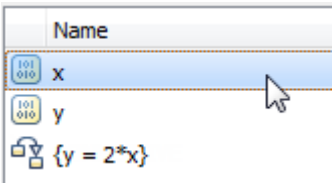
Specify the size, complexity, and type of the function input and output arguments. A chart can export only functions with fully specified prototypes.

1 Open the Model Explorer. On the **Modeling** tab and from the **Design** section, select **Model Explorer**.

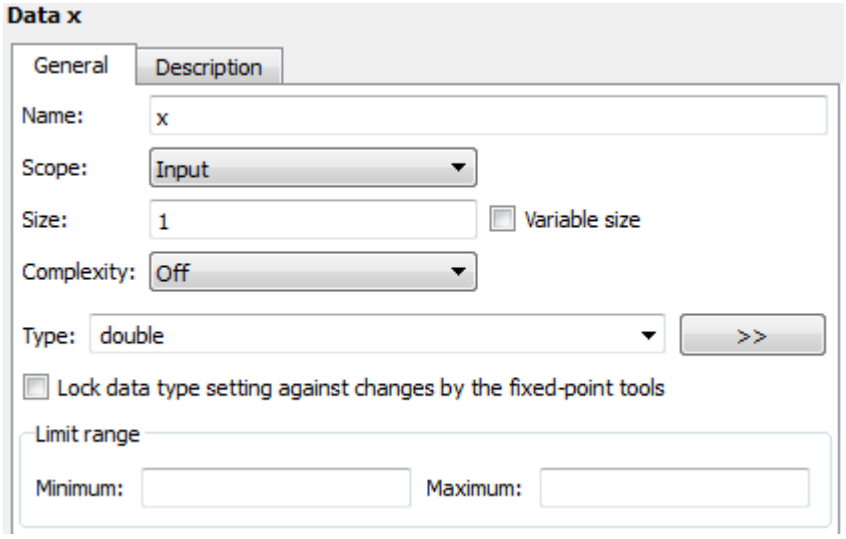
2 In the left pane, select the graphical function.



3 From the **Column View** list in the middle pane, select `Stateflow`. Select the filter icon, and then from the toggle list, select **All Stateflow Objects**. From the center pane table, select an input or output argument.



4 In the right pane, set Size to 1 (scalar), Set **Complexity** to `Off` (real number), and set **Type** to `double`.



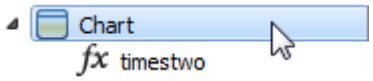
5 Repeat steps 2 and 3 with the output function argument `y`.

Set export function parameters for a graphical function

Set parameters to export a graphical function to a Simulink model from a Stateflow chart during a simulation.

1 Open the Model Explorer.

2 In the left pane, select the chart containing the graphical function.



3 In the property dialog box on the right side, select the **Export Chart Level Functions** check box, click the **Apply** button, and then select the **Treat Exported Functions as Globally Visible** check box.

If you are calling the exported graphical function from another Stateflow chart (not the chart that exported the graphical function), you do not need to select the **Treat Exported Functions as Globally Visible** check box.

Create a Simulink function using an exported MATLAB function from a Stateflow chart

Set up a MATLAB function in a Stateflow chart to receive data through an input argument from a function caller and then pass a calculated value back through an output argument. Set chart parameters to export the function to a Simulink model.

Define a MATLAB function in a Stateflow chart

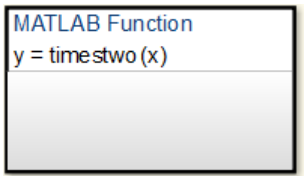
Create a MATLAB function in a Stateflow chart. Define the function interface and function definition.

1 Add a Stateflow Chart to your Simulink model. Double-click on the block diagram. In the search box, enter `chart`, and then from the search results, select `Chart`.

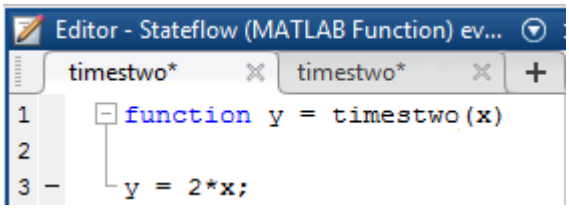
2 Open the chart.

3 Add a MATLAB function. From the left-side toolbar, click and drag the graphical function icon onto the chart.

4 Define the function interface. In the function box, replace the ? with the function interface `y = timestwo(x)`.



5 Double-click the function box to open the MATLAB code editor. Define the function algorithm with the MATLAB code.



Test the MATLAB function

Test the MATLAB function within a Stateflow chart before exporting to a Simulink model.

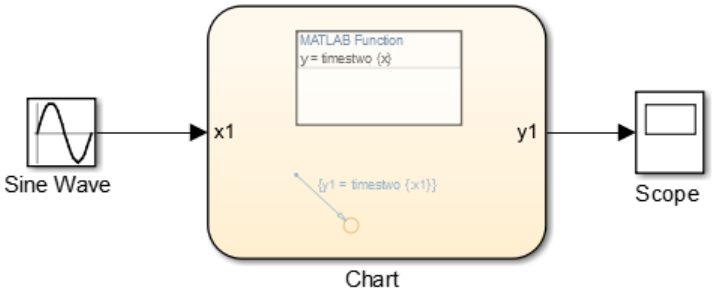
1 Add a default transition in the chart for testing the function. From the left-side toolbar, click and drag a default transition arrow onto the chart.

2 Double-click the arrow and replace the ? with `{y1 = timestwo(x1)}`.

3 Add an input port to the chart. Open the Model Explorer. In the left pane, select `Chart`. From the menu, select **Add** > **Data**. Set **Name** to `x1` and **Scope** to `Input`.

4 Add an output port to the chart. From the menu, select **Add** > **Data**. Set **Name** to `y1` and **Scope** to `Output`.

5 Add a Sine Wave block to provide test data for the input and a Scope block to view results from the output..

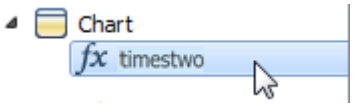


6 Run a simulation.

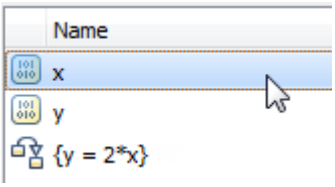
Set the argument parameters for a MATLAB function

Specify the size, complexity, and type of the function input and output arguments. A chart can export only functions with fully specified prototypes.

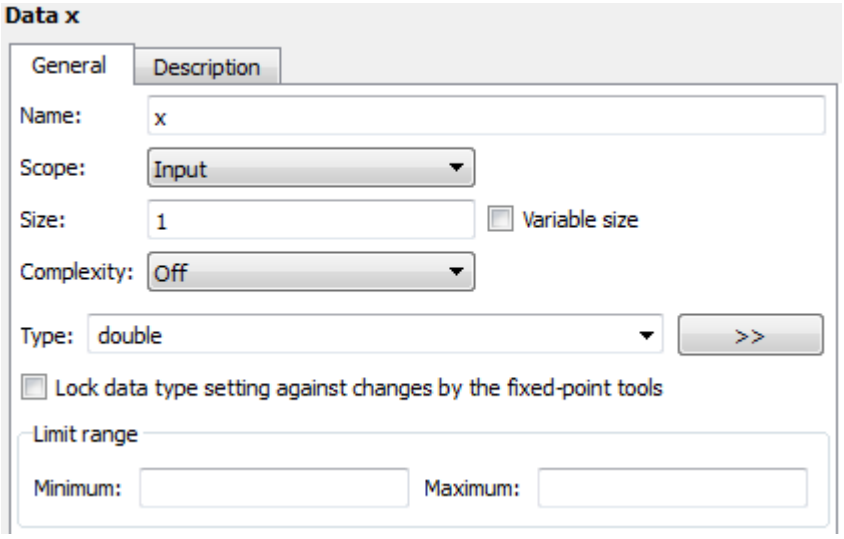
1. Open the Model Explorer. On the **Modeling** tab and from the **Design** section, select **Model Explorer**.
2. In the left pane, select the MATLAB function.



3. From the **Column View** list in the middle pane, select `Stateflow`. Select the filter icon, and then from the toggle list, select **All Stateflow Objects**. From the center pane table, select an input or output argument.



4. In the right pane, set **Size** to 1 (scalar), Set **Complexity** to `Off` (real number), and set **Type** to `double`.

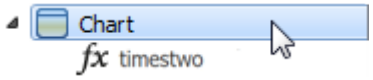


5. Repeat steps 2 and 3 with the output function argument `y`.

Set export function parameters for a MATLAB function

Set parameters to export a MATLAB function from a Stateflow chart during a simulation.

1. Open the Model Explorer.
2. From the left pane, select the chart containing the MATLAB function.



3 In the property dialog box on the right side, select the **Export Chart Level Functions** check box, click the **Apply** button, and then select the **Treat Exported Functions as Globally Visible** check box.

   If you are calling the exported MATLAB function from another Stateflow chart (not the chart that exported the MATLAB function), you do not need to select the **Treat Exported Functions as Globally Visible** check box.

See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Scoped Simulink Function Blocks in Models” on page 10-157
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Simulink function callers: Function Caller block, MATLAB Function block, Stateflow chart

In this section...

Simulink functions have an interface with input and output arguments similar to programming languages. Simulink function callers send data through input arguments to Simulink functions, execute the function, and then receive data back from the function through output arguments. You can call a Simulink function using:

- Function Caller blocks
- MATLAB Function blocks
- Stateflow charts

The following sections show how to call a Simulink function. The function `y = timestwo(x)` multiplies a value (`x`) from a caller by `2`, and then sends the calculated value (`y`) back to the caller. To create the functions, see “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121.

To open completed model with Simulink functions and function callers, see `ex_simulink_functions_and_function_callers`.

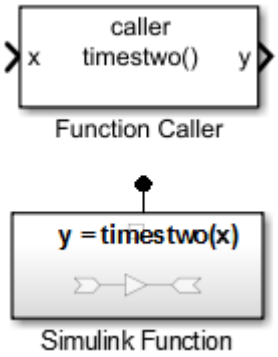
Use a Function Caller block to call a Simulink Function block

Set up a Function Caller block to send data through an input argument to a Simulink Function block, and receive data back from the function through an output argument.

1. Add a Function Caller block to your model.
2. Open the Function Caller dialog box. In the **Function prototype** box, enter `y = timestwo(x)`. This function prototype creates an input port `x` and output port `y` on the Function Caller block.

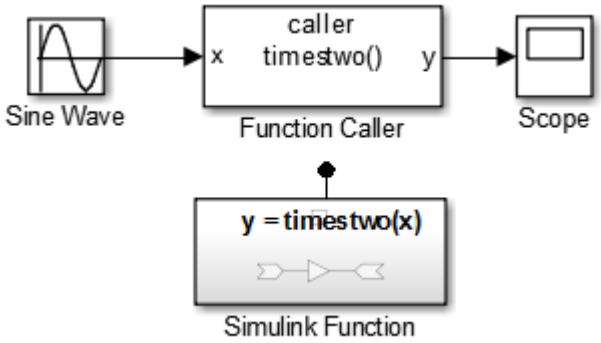
   Note Typing in a blank text box displays a list of previously created function prototypes that match the text you are typing.
3. Add and setup a Simulink Function block as described in “Create a Simulink function using a Simulink Function Block” on page 10-121.

   Note The function and argument names for the Simulink Function block and the **Function prototype** for the Function Caller block must match exactly.

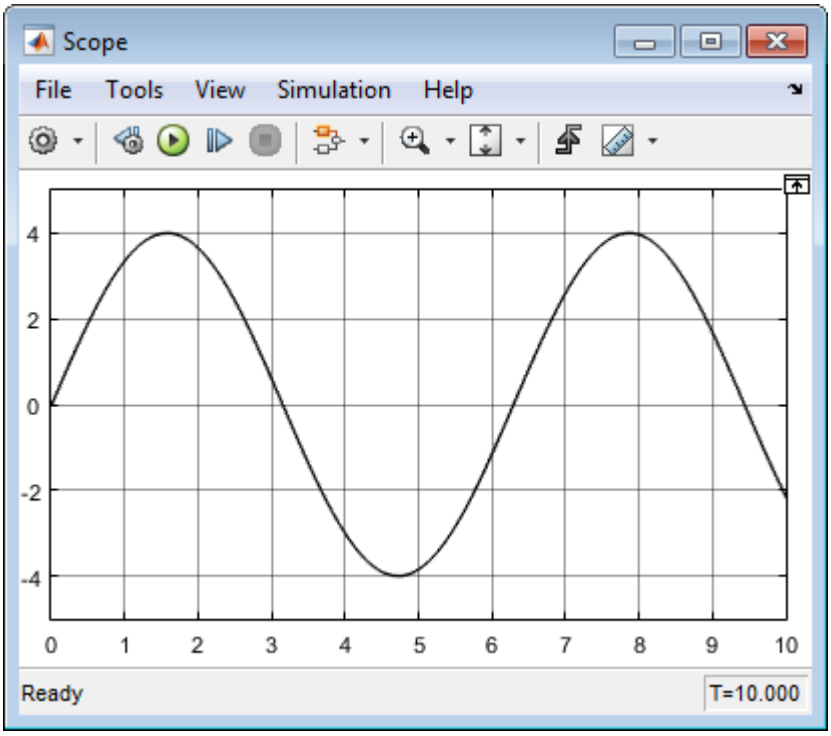


Test the function call

1 Add a Sine Wave block to provide test data for the input and a Scope block to view results from the output.



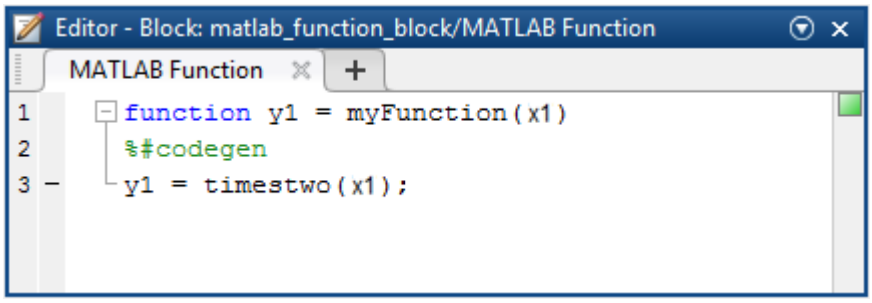
2 Run a simulation. The input sine wave with an amplitude of 2 is doubled.



Use a MATLAB Function block to call a Simulink Function block

Set up a MATLAB Function block to send data through an input argument to a Simulink Function block, and receive data back from the function through an output argument.

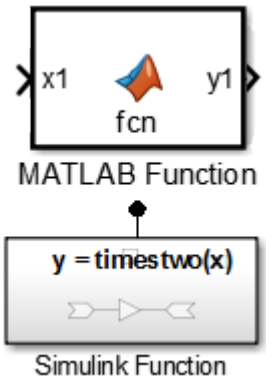
1. Add a MATLAB Function block to your model.
2. Double-click the block, which opens the MATLAB editor. Enter the function call `y1 = timestwo(x1)`.



Note The argument names for the function you define in the MATLAB Function block do not have to match the argument names for the function that you define with a Simulink Function block. For a Function Caller block that calls a Simulink Function block, argument names must match.

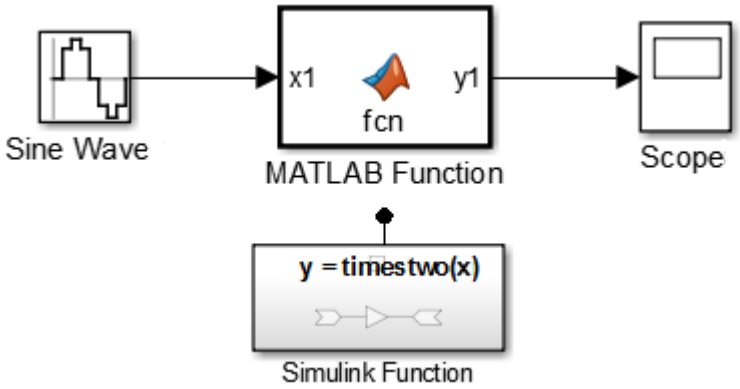
Note MATLAB Function blocks only support discrete and fixed-in-minor sample times.

3 Add and setup a Simulink Function block as described in “Create a Simulink function using a Simulink Function Block” on page 10-121.



Test the function call

1 Add a Sine Wave block to provide test data for the input and a Scope block to view results from the output.



2 For the Sine Wave block, set the **Sample time** to `0.01`. For the model, open the Configuration Parameters dialog box to the solver pane. Set **Type** to `Fixed-step` and **Fixed-step size** to `0.01`.

3 Run a simulation.

Use a Stateflow chart to call a Simulink Function block

Set up a Stateflow chart to send data through an input argument to a Simulink Function block, and receive data back from the function through an output argument.

1 Add a Stateflow Chart to your Simulink model. Double-click on the Simulink block diagram. In the search box, enter `chart`, and then from the search results, select `Chart`.

2 Double-click the chart to open it.

3 From the left-side toolbar, click and drag the default transition icon onto the chart.

4 Add an input port to the chart. Open the Model Explorer. In the left pane, select `Chart`. From the menu, select **Add** > **Data**. Set **Name** to `x1` and **Scope** to `Input`.

Note The argument names for the function you define in the Stateflow chart do not have to match the argument names for the function that you define with a Simulink Function block. For a Function Caller block that calls a Simulink Function block, argument names must match.

5 Add an output port to the chart. From the menu, select **Add** > **Data**. Set **Name** to `y1` and **Scope** to `Output`.

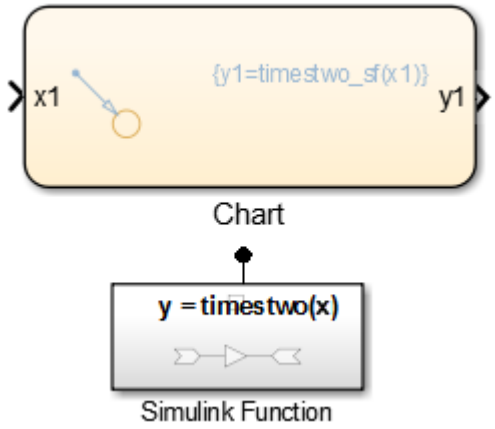
6 Add a Sine Wave block and connect signal output to the chart input port. Add a Scope block and connect input to the chart output port.

7 Edit transition code to call a function. For example, to call the Simulink Function block, enter:

```
{y1=timestwo_sf(x1);}
```

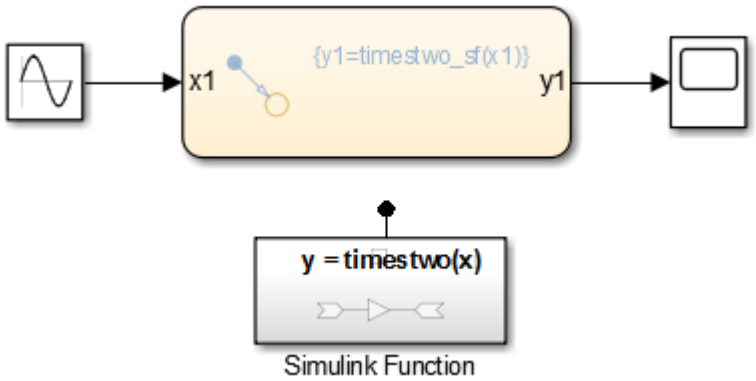
Note Input signals to a Stateflow chart can be either continuous or discrete.

8 Add and setup a Simulink Function block as described in “Create a Simulink function using a Simulink Function Block” on page 10-121.



Test the function call

1 Add a Sine Wave block to provide test data for the input and a Scope block to view results from the output.

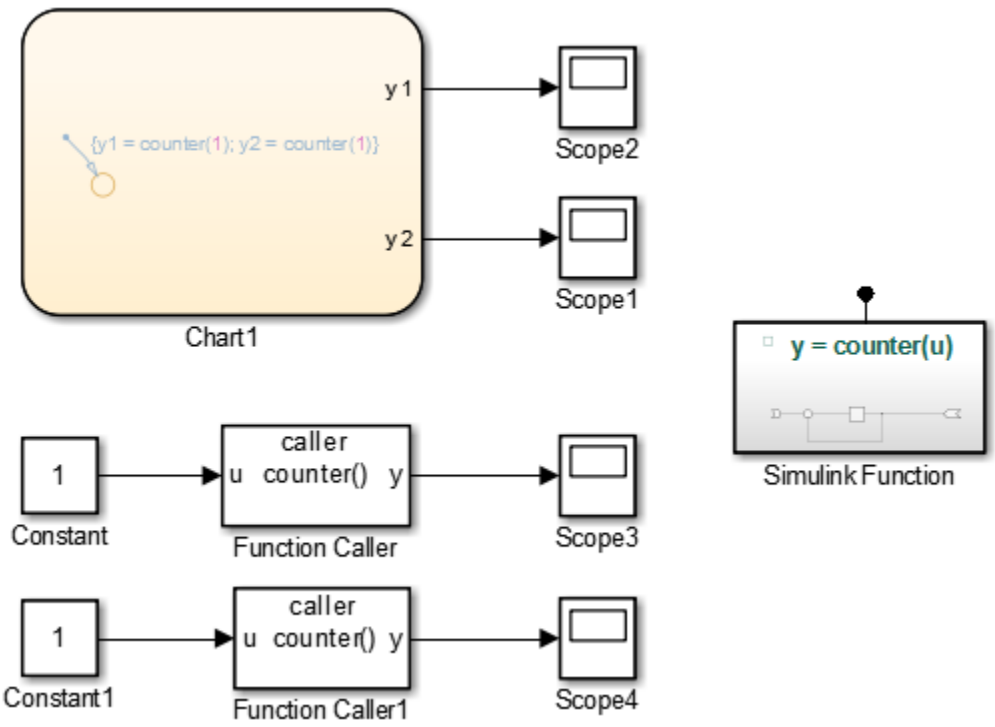


2 For the Sine Wave block, set the **Sample time** to `0.01`. For the model, open the Configuration Parameters dialog box to the solver pane. Set **Type** to `Fixed-step` and **Fixed-step size** to `0.01`.

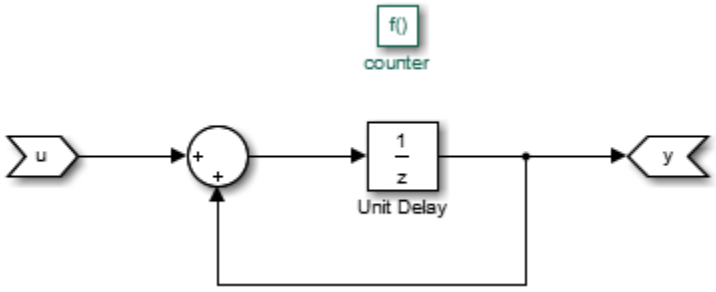
3 Run a simulation.

Call a Simulink Function block from multiple sites

If you call a Simulink Function block from multiple sites, all call sites share the state of the function. For example, suppose that you have a Stateflow chart with two calls and two Function Caller blocks with calls to the same function.



A function defined with a Simulink Function block is a counter that increments by 1 each time it is called with an input of 1.

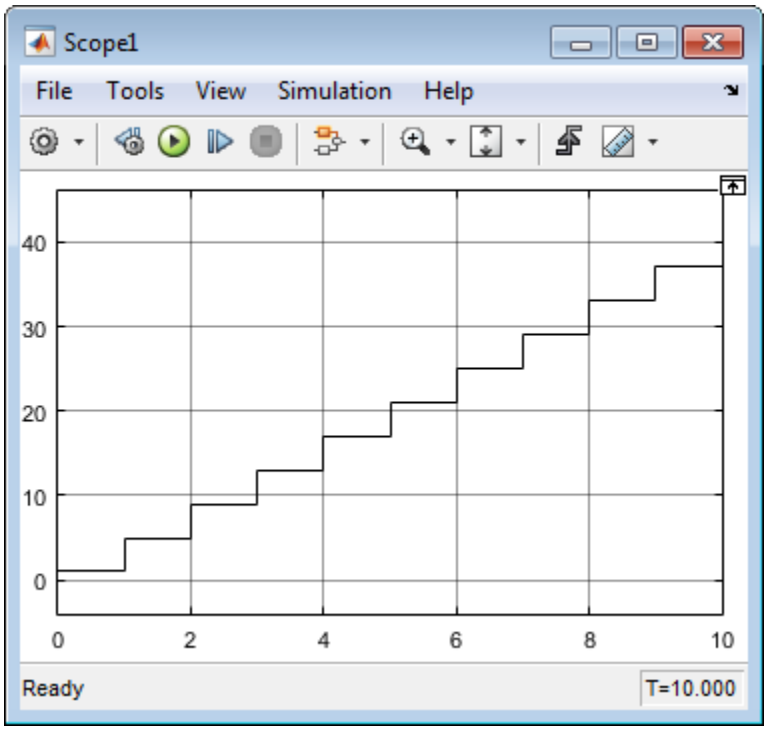


The Unit Delay block has state because the block value is persistent between calls from the two Function Caller blocks and the Stateflow chart. Conceptually, you can think of this function being implemented in MATLAB code:

```
function y = counter(u)
persistent state;
if isempty(state)
      state = 0;
end
y = state;
state = state + u;
```

Simulink initializes the state value of the Unit Delay block at the beginning of a simulation. After that, each time the function is called, the state value is updated.

In this example, the output observed in Scope1 increments by 4 at each time step. Scope2, Scope3, and Scope4 show a similar behavior. The only difference is a shift in the observed signal due to the execution sequence of the function calls.



Diagnostic Settings With Multiple Callers

For multiple callers that share a function and have different sample time rates, data integrity and consistency of real-time code might be a problem. Consider controlling the severity of diagnostics.

Select a **Fixed-step** solver. Set the **Treat each discrete rate as a separate task** to:

- Clear (single–tasking), and then set **Single task rate transition** parameter to `none` (default), `warning`, or `error`.
- Select (multi-tasking), and then set **Multitask rate transition** parameter to `error` (default) or `warning`.

See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink Function Blocks in Referenced Models” on page 10-140

Argument Specification for Simulink Function Blocks

In this section...

When a Simulink Function block is within the scope of a Function Caller block, you do not have to specify the parameters. In such a case, the Function Caller block can determine the input and output argument specifications.

You specify arguments when a Simulink Function block is outside the scope of a Function Caller block. A Simulink Function block is considered to be out of scope of a Function Caller block when the two blocks are in separate models referenced by a common parent model.

Example Argument Specifications for Data Types

This table lists 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 scalars.
`boolean`	`boolean(true)`,`boolean(false)`	Boolean, 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.

Simulink Function Block Data Type	Function Caller Block Expression	Description
`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.

Input Argument Specification for 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.

A bus input to a Function Caller block must be a non-virtual bus using a bus object.

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_MATLABStruct.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.DataType = 'Bus: myBus';
   myBus_parameter.Value = myBus_MATLABStruct;
   ```

5. For the Function Caller block dialog box, set the **Input argument specification** parameter to `myBus_parameter`.
6. For the Argument In block dialog box of the Simulink Function block, set the **Data type** parameter to `Bus: myBus`.

Input Argument Specification for 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`.

Input Argument 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. For the Function Caller block dialog box, set the **Input argument specification** parameter to `myAlias_parameter`.
5. For the Argument In block dialog box within a Simulink Function block, set the **Data type** parameter to `myAlias`.

See Also

Blocks
Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Scoped Simulink Function Blocks in Models” on page 10-157
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Simulink Function Blocks in Referenced Models

In this section...

You can place Simulink Function blocks and function callers (such as Function Caller blocks and Stateflow charts) in a referenced model, but doing so requires some special considerations:

- The referenced model must follow export-function model rules. See “Export-Function Models Overview” on page 10-97.
- Sometimes, you must explicitly define the argument data types for a Function Caller block.

These examples show four relationships between Function Caller blocks, Simulink Function blocks, and referenced models.

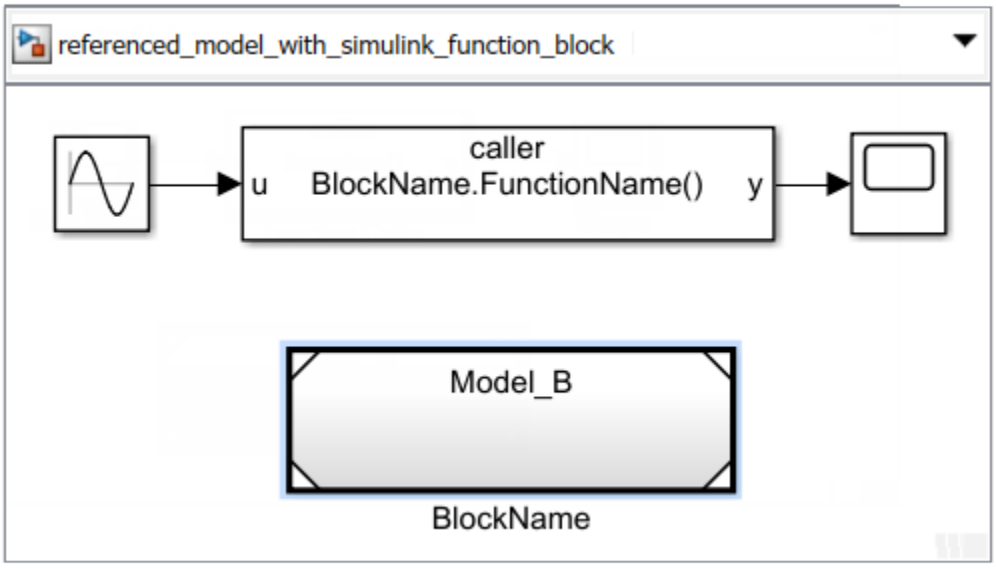
Simulink Function Block in Referenced Model

In this example, the parent model contains a Function Caller block, and the referenced model, `Model_B`, contains a Simulink Function block. `Model_B` must follow export-function model rules.

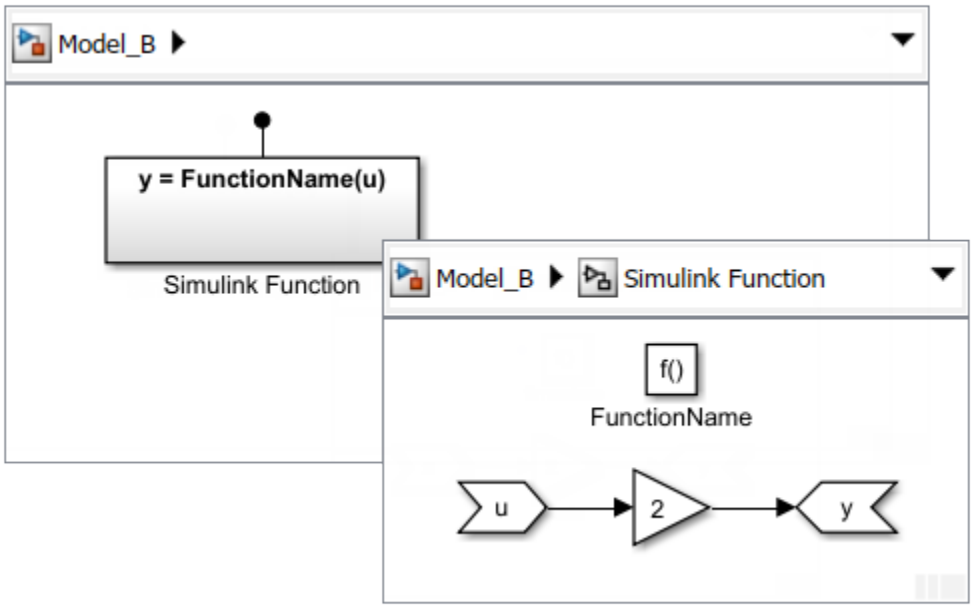
The Function Caller block can determine the argument data types of the function. In the Function Caller block, you do not need to define the **Input argument specification** and **Output argument specification** parameters.

But since, by default, the Simulink Function block is scoped to the model, you must qualify a call to the function name with the Model block name.

To open a completed model, see `ex_referenced_model_with_simulink_function_block`.



`Model_B` contains a Simulink Function block that defines a function for multiplying the input by 2. Because this model contains only a Simulink Function block, it satisfies export-function model rules. See “Export-Function Models Overview” on page 10-97.



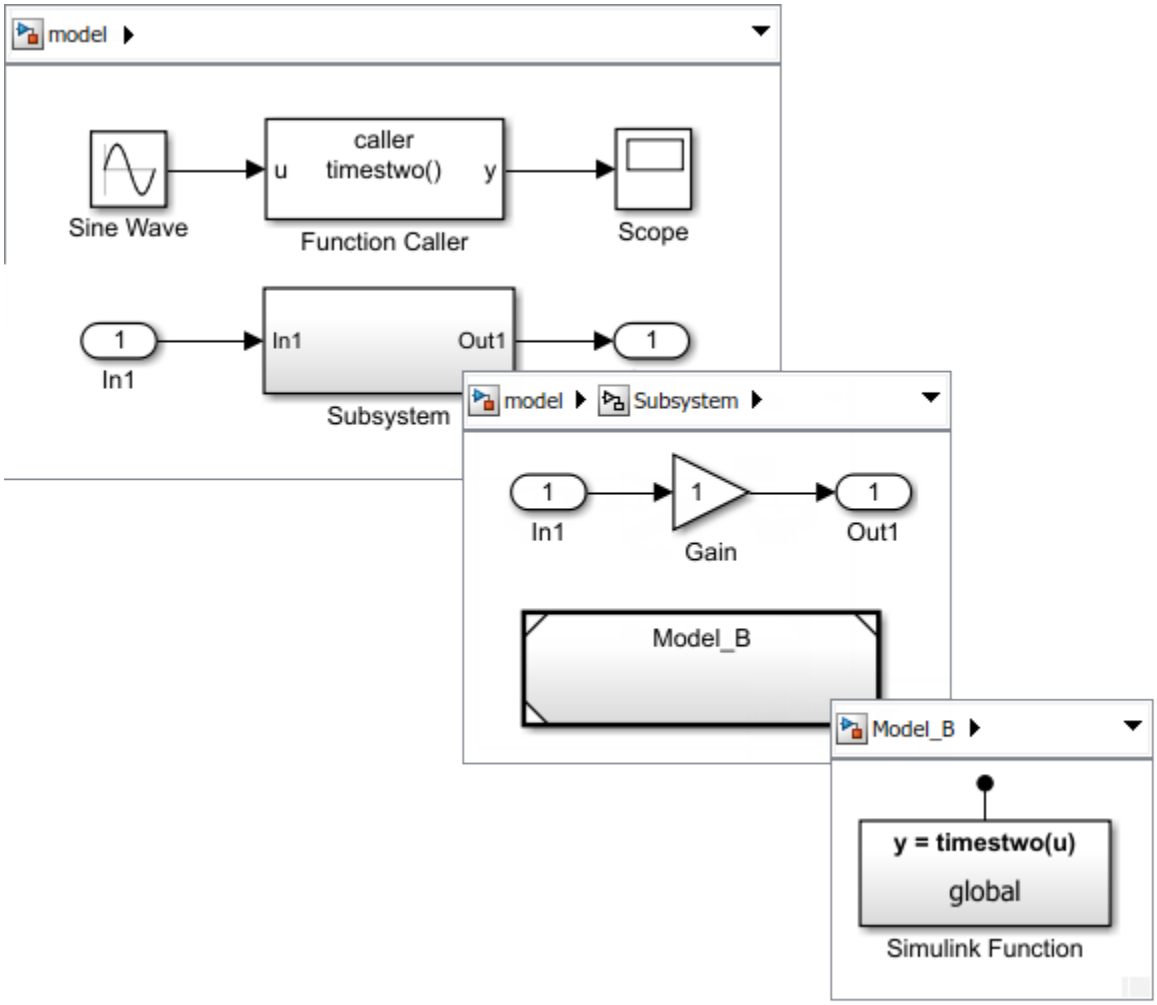
For `Model_B`, set the Configuration Parameters for the solver to satisfy export-function model rules:

- **Type**: `Fixed-step`.
- **Solver**: `discrete (no continuous states)`.

Simulink Function Block in Referenced Model Placed in Subsystem

Once the Simulink Function block is identified as global by setting **Function visibility** to `global`, there are no limitations on where this referenced model can be placed.

For example, you could place `Model_B` with a Simulink Function block in a Subsystem block.

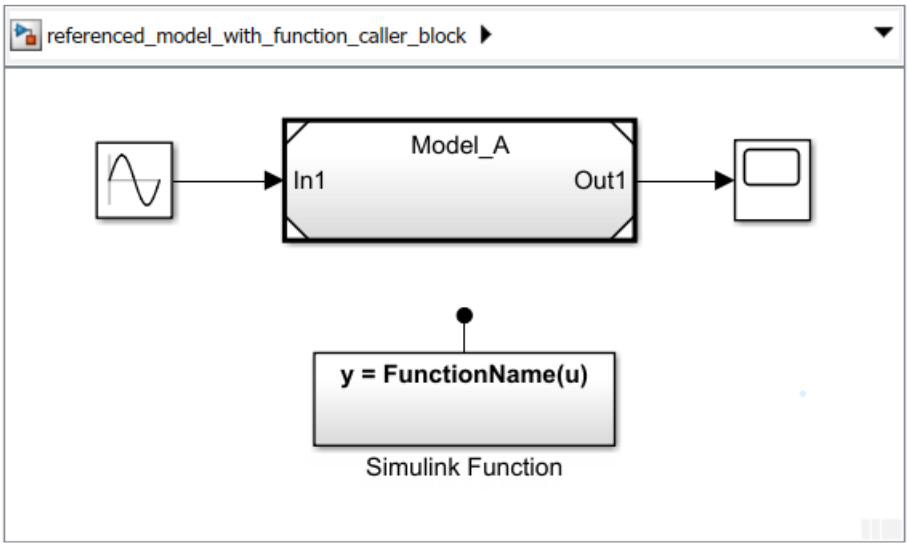


Function Caller Block in Referenced Model

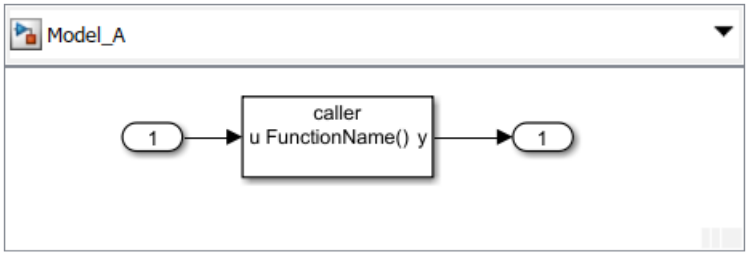
In this example, the parent model contains a Simulink Function block, and a referenced model, `Model_A`, contains a Function Caller block. If you want to use this modeling pattering, the **Function visibility** parameter for the Trigger port block in the Simulink Function block must be set to `global`.

For the parent model, set the solver type to `Variable-step` or `Fixed-step`.

To open a completed model, see `ex_referenced_model_with_function_caller_block`.



`Model_A` contains a Function Caller block.



Since the Function Caller block cannot find the function in `Model_A`, you must set the **Function visibility** parameter for the Trigger block to `global` and specify the Function Caller block argument parameters:

- **Input argument specification**: Specify to match the Simulink Function block input argument data types, for example, `double(1.0)`.

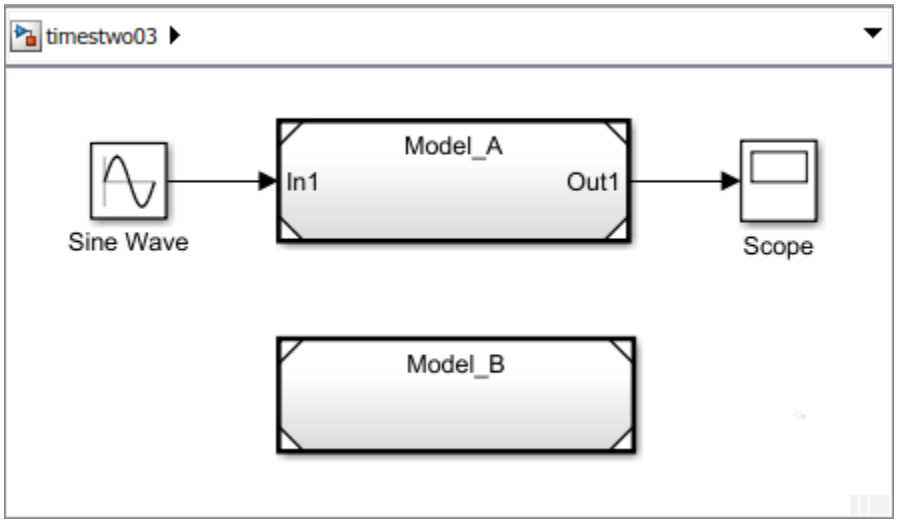
  Specify the argument specification for a Simulink Function block with the **Data type** parameter in the Input Argument and Output Argument blocks.
- **Output argument specification**: Specify to match the Simulink Function block output argument data types, for example, `double(1.0)`.

Function and Function Caller Blocks in Separate Referenced Models

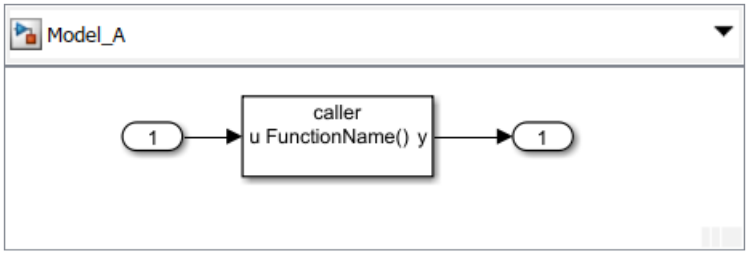
In this example, the parent model contains two referenced models. `Model_A` is a referenced model with a Function Caller block. `Model_B` is a referenced model with a scoped Simulink Function block. Only `Model_B` with a Simulink Function block must follow export-function rules.

For `Model_A`, provide the argument specification as you do for the referenced model in “Function Caller Block in Referenced Model” on page 10-142. For `Model_B`, specify parameters as you do for the referenced model in “Simulink Function Block in Referenced Model” on page 10-140.

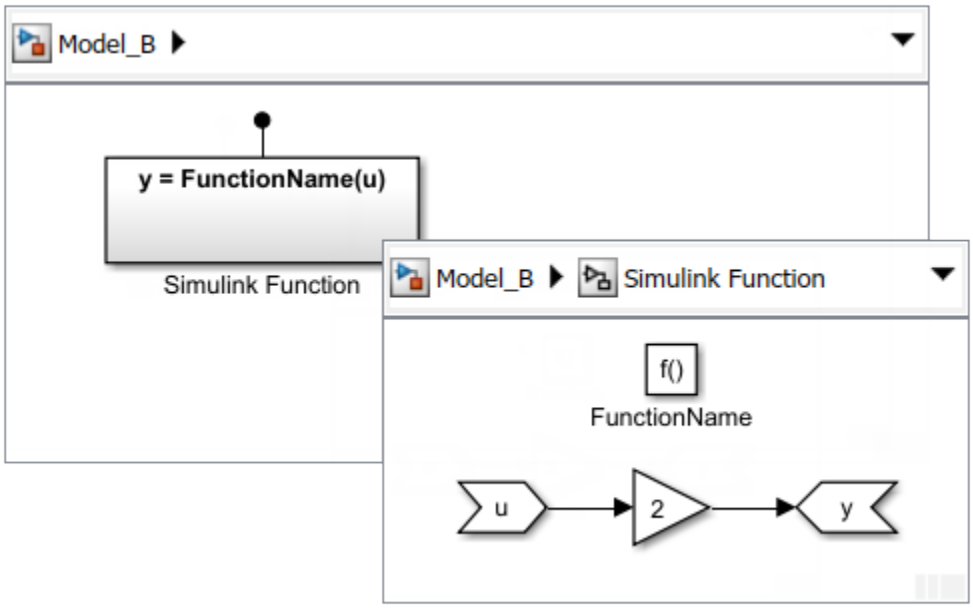
To open a completed model, see `ex_referenced_model_with_simulink_function_and_function_caller`.



`Model_A` contains a Function Caller block. If the function is set to `global`, define the Input and Output Argument Specification parameters. If the function is set to `scoped`, provide the file name, not the block name, of the model where the function is expected to be resolved to as `y = Model_B.FunctionName(u)`.



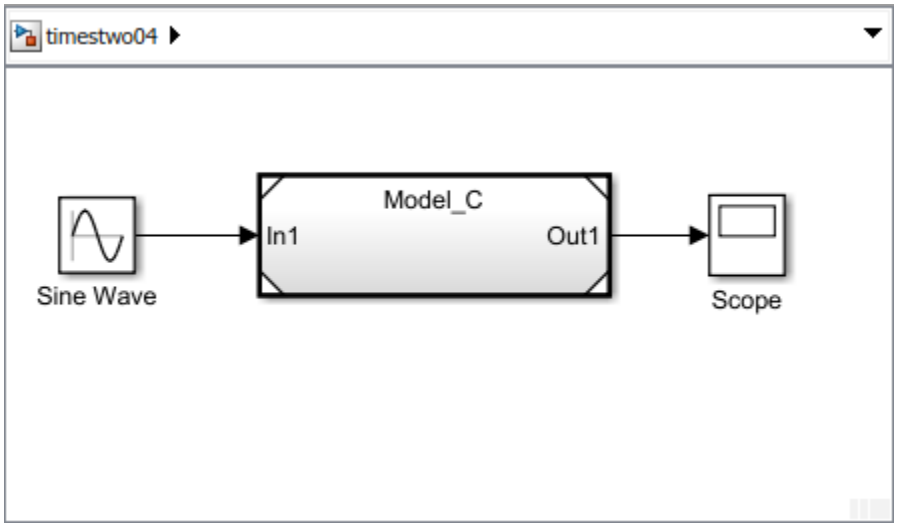
`Model_B` contains a Simulink Function block that defines a function for multiplying the input by 2. Because this model contains only a Simulink Function block, it satisfies export-function model rules. See “Export-Function Models Overview” on page 10-97.



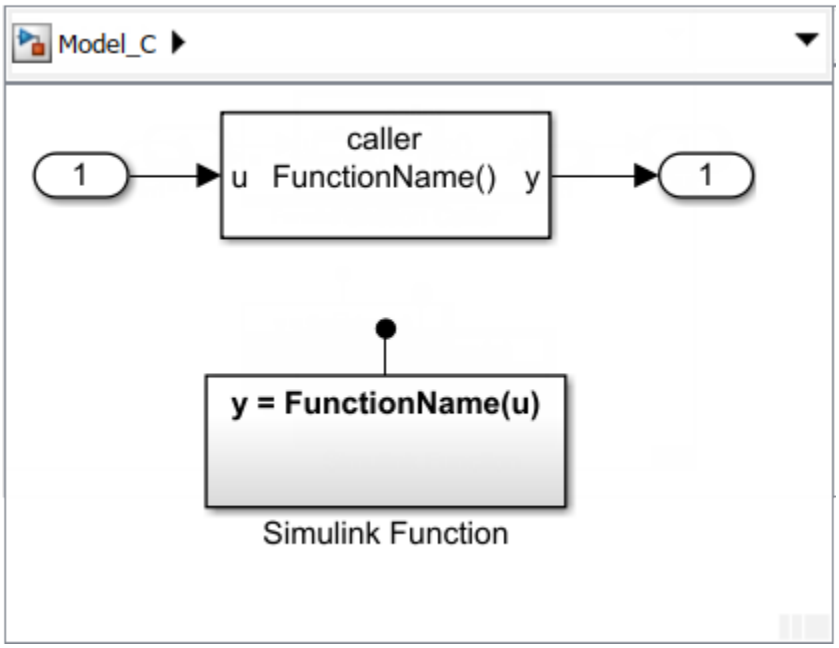
Function and Function Caller in the Same Model

In this example, the parent model contains one referenced model, `Model_C`, with both a Function Caller block and a scoped Simulink Function block.

- If the there is only one instance of `Model_C`, and the Configuration Parameter **Total number of instances allowed per top model** is set to `One`, the parent model simulates without error. Since `Model_C` does not export the function, it does not need to follow export-function rules.
- If the Configuration Parameter **Total number of instances allowed per top model** is set to `Multiple`, `Model_C` exports the function regardless if there is a single instance or multiple instances of the model. In both cases, Simulink displays an error. The model needs to follow export-function rules, but it does not because a Function Caller block is not allowed at the top-level of an export-function model.



`Model_C` contains both a Function Caller block and a scoped Simulink Function block. If you want to use this modeling pattern, only one instance of `Model_C` is allowed in the parent model.



See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Scoped Simulink Function Blocks in Models” on page 10-157
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Scoped and Global Simulink Function Blocks Overview

Defining the visibility of functions can help you to avoid name space conflicts when integrating your referenced models. A Simulink Function block defines the visibility of its function in relationship to the subsystem or model containing the block as either scoped or global. By default, Simulink Function blocks are scoped.

- *Function Visibility.* A scoped function is visible in its hierarchy. A function caller located at the same level as the function, or one or more levels below can refer to the function. A global function is visible across a model hierarchy. This means that a function caller located anywhere in the current model or in the parent model hierarchy can refer to the function.
- *Function accessibility* is determined by the visibility of a function and the location of the function caller relative to the Simulink Function block. For function callers one hierarchical level above the function, qualify the function name with the virtual subsystem block name or model block name.
- *Function exporting* refers to functions exported from models. A function with global visibility, placed anywhere in an export function model, is exported to the top level of a model hierarchy in addition to the model interface. A function with scoped visibility at the root level of an export function model is exported to the model interface. In both these cases, you can access the exported function outside of the model.

Use the **Function visibility** parameter for the Trigger block within a Simulink Function block to set the function visibility to either `scoped` or `global`.

Summary of Simulink Function Block Visibility and Access

	Function visibility	Function accessibility	Function exporting
Virtual Subsystem	`scoped` Function name does not have to be unique	function caller inside hierarchy or at parent level. function caller inside Subsystem block hierarchy – unqualified, `fcn()`. See Resolve to a Function Hierarchically in "Scoped Simulink Function Blocks in Subsystems" on page 10-150. function caller at parent level – qualified with subsystem block name, `subsystem.fcn()`. See Resolve to a Function by Qualification in "Scoped Simulink Function Blocks in Subsystems" on page 10-150.	Does not apply
	`global` Function name must be unique	function caller at any level of hierarchy down or up. function caller at any level of hierarchy – unqualified, `fcn()`	Function at any level of model exported to the global name space of the top-level model
Atomic Subsystem	`scoped` Function name does not have to be unique	function caller only inside hierarchy function caller inside Subsystem block hierarchy – unqualified, `fcn()`. See Resolve to a Function Hierarchically in "Scoped Simulink Function Blocks in Subsystems" on page 10-150. function caller at parent level – not allowed	Does not apply
	`global` visibility not allowed	function call not allowed	Does not apply

	Function visibility	Function accessibility	Function exporting
Model	`scoped` Function name does not have to be unique	function caller inside hierarchy or at parent level. function caller inside Subsystem block hierarchy – unqualified, `fcn()`. See Resolve to a Function Hierarchically in "Scoped Simulink Function Blocks in Models" on page 10-157. function caller at parent level - qualified with Model block name, `model_block.fcn()`. See Resolve to a Function by Qualification in "Scoped Simulink Function Blocks in Models" on page 10-157.	Function at the root level of a model exported to the model interface
	`global` Function name must be unique	function caller at any level of hierarchy down or up. function caller at any level of hierarchy – unqualified, `fcn()`	Function at any level of model exported to the global name space of the top-level model

See Also

Blocks
Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- "Simulink Functions Overview" on page 10-113
- "Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions" on page 10-121
- "Simulink Function Blocks in Referenced Models" on page 10-140
- "Scoped Simulink Function Blocks in Subsystems" on page 10-150
- "Scoped Simulink Function Blocks in Models" on page 10-157
- "Diagnostics Using a Client-Server Architecture" on page 10-164

Scoped Simulink Function Blocks in Subsystems

The scope of a Simulink function is defined in its parent subsystem within the context of a model. If you place a function in any Subsystem block, access to the function from outside the model is prohibited by default. In both cases, the Trigger block **Function visibility** parameter is set to `scoped`. The Simulink Function block can be located:

- In a virtual subsystem — Call the function from within the containing Subsystem block hierarchy without qualifying the function name, or call the function from outside the subsystem by qualifying the function name with the subsystem block name.
- In an atomic or non-virtual subsystem — Call the function from within the containing Subsystem block hierarchy without qualifying the function name. Accessing the function from outside of the subsystem is prohibited.

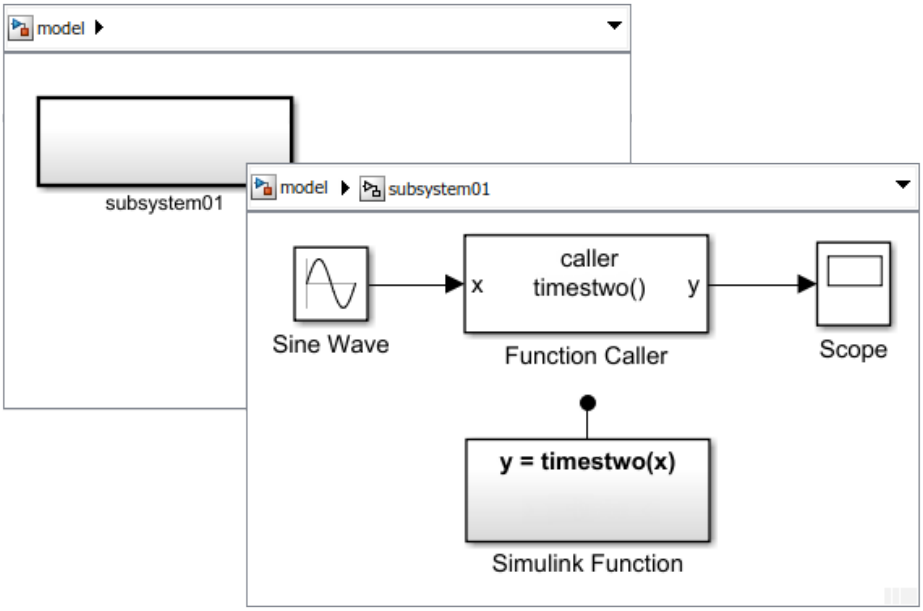
Resolve to a Function Hierarchically

Placing a scoped Simulink Function block within any Subsystem block (virtual or atomic) limits access to the function and removes the function name from the global name space. When a function caller resolves to a function hierarchically, it looks for the function using the following rules:

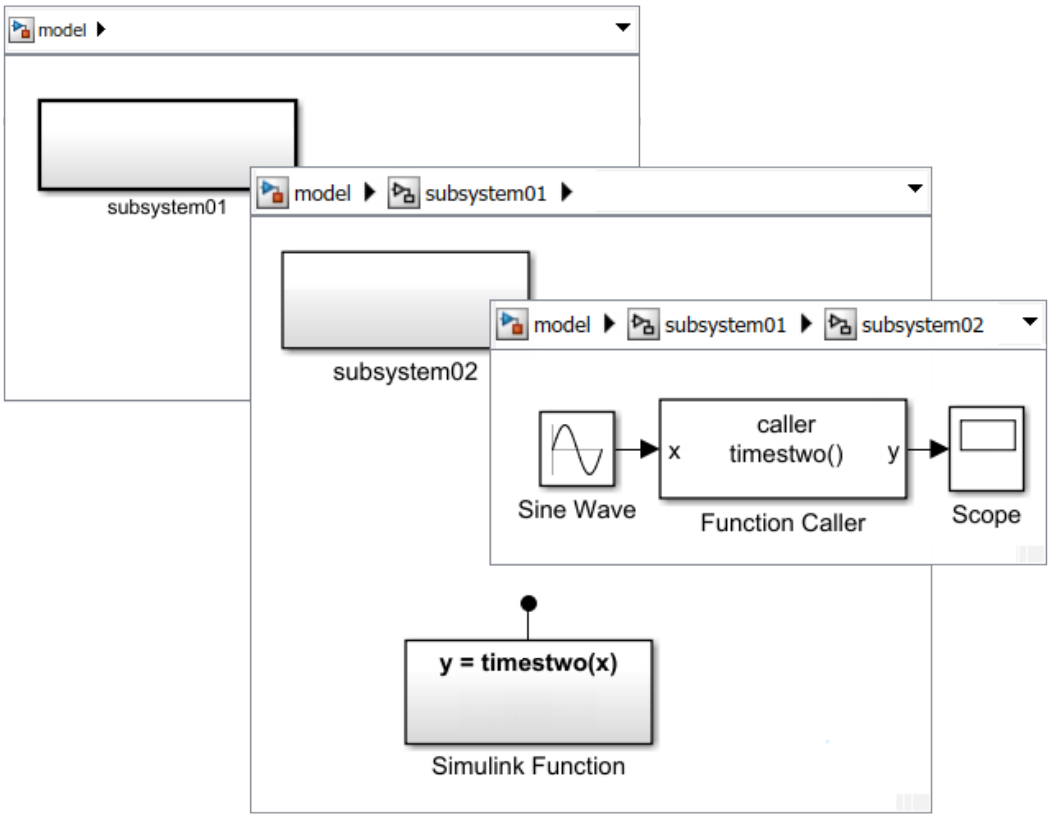
- Resolution Rule 1: Is the scoped Simulink Function block in the current Subsystem block with the function caller?
- Resolution Rule 2. If the scoped function is not in the current subsystem, is the scoped Simulink Function block in a parent or grandparent Subsystem block one or more levels above the function caller, or in a parent Model?

If a function caller resolves to a function hierarchically, you can call the function without qualifying the function name:

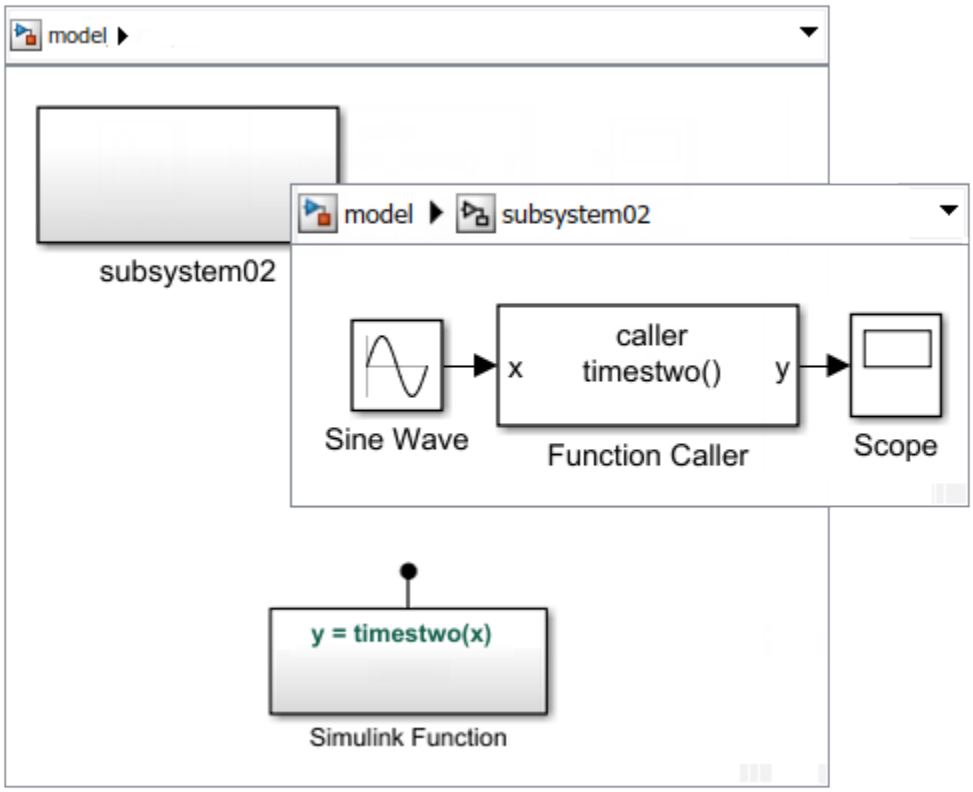
- Function caller located at the same hierarchic level as the function. In this case, the function caller finds the scoped function in the current subsystem (Resolution Rule 1).



- Function caller located in a Subsystem block one or more Subsystem block levels below the hierarchic level of the Simulink Function block. The function caller hierarchy cannot include a Model block since the function caller cannot cross model reference boundaries. In this case, the function caller didn't find the scoped function in the current subsystem, but it found the function in the parent subsystem (Resolution Rule 2).



In this case, the function caller didn't find the scoped function in the current subsystem, but it found the function in the parent model (Resolution Rule 2).



- You can also call a Simulink Function block in a Subsystem block without qualification from a MATLAB Function block or a Stateflow chart within the block.

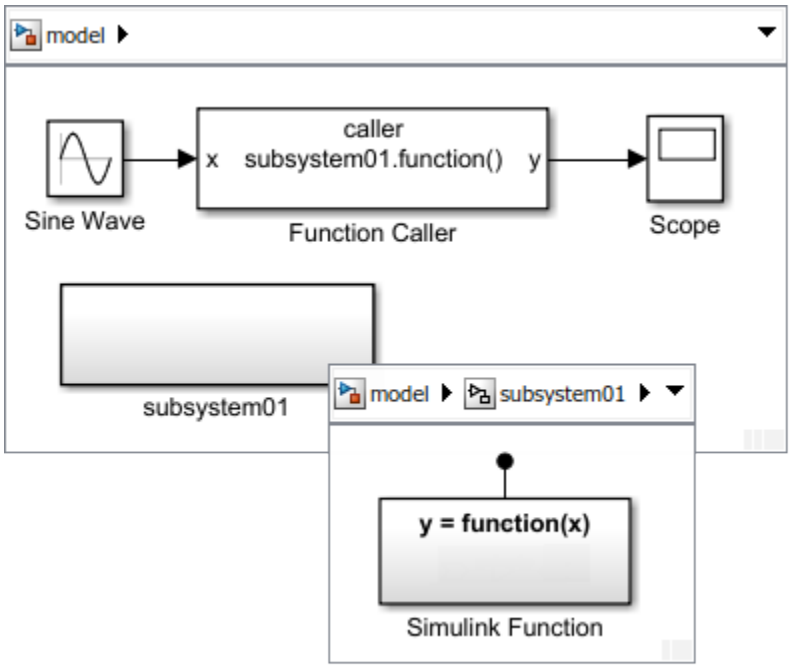
Resolve to a Function by Qualification

When you place a Simulink Function block in a virtual Subsystem block, the function name is not visible outside of the subsystem. However, you can call the function by qualifying the function name with the Subsystem block name. When a function caller resolves to a qualified function hierarchically, it looks for the virtual Subsystem block containing the function using the following rules:

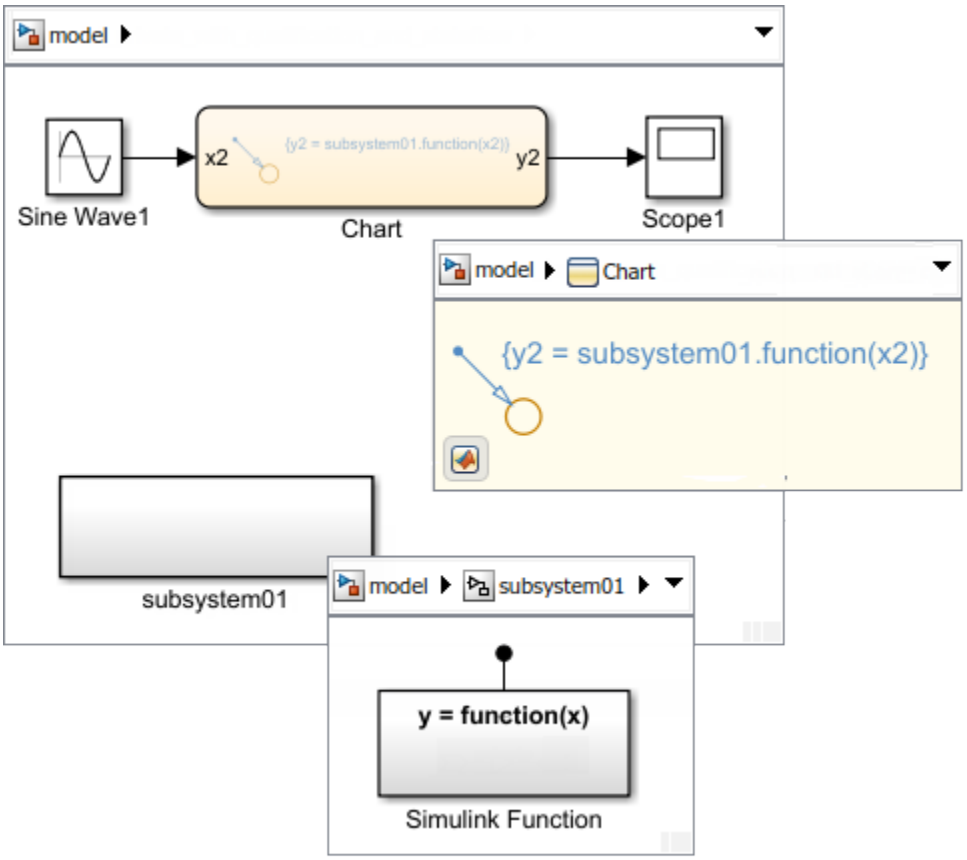
- Resolution Rule 1: Is the virtual Subsystem block in the current component with the function caller? A component can be a Subsystem block or Model.
- Resolution Rule 2. If the virtual Subsystem block is not in the current component, is the virtual Subsystem block in a parent or grandparent component one or more levels above the function caller?

If a function caller resolves to a virtual Subsystem block with a scoped function, you can call the function by qualifying the function name:

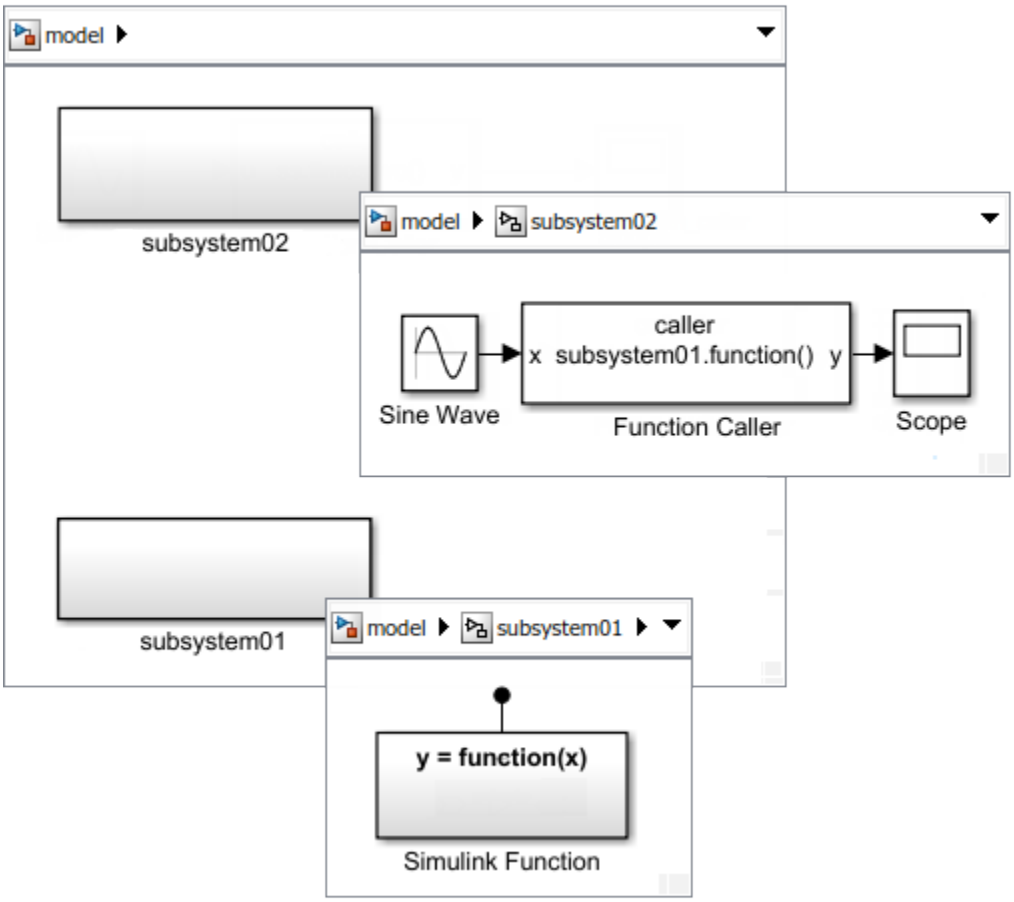
- Function caller located outside of the subsystem one hierarchic level above the function. In this case, the function caller finds the Subsystem block with the scoped function in the current model (Resolution Rule 1).



- Calling the function from a Stateflow chart outside the subsystem one hierarchic level above the function. In this case, the function caller finds the Subsystem block with the scoped function in the current model (Resolution Rule 1).

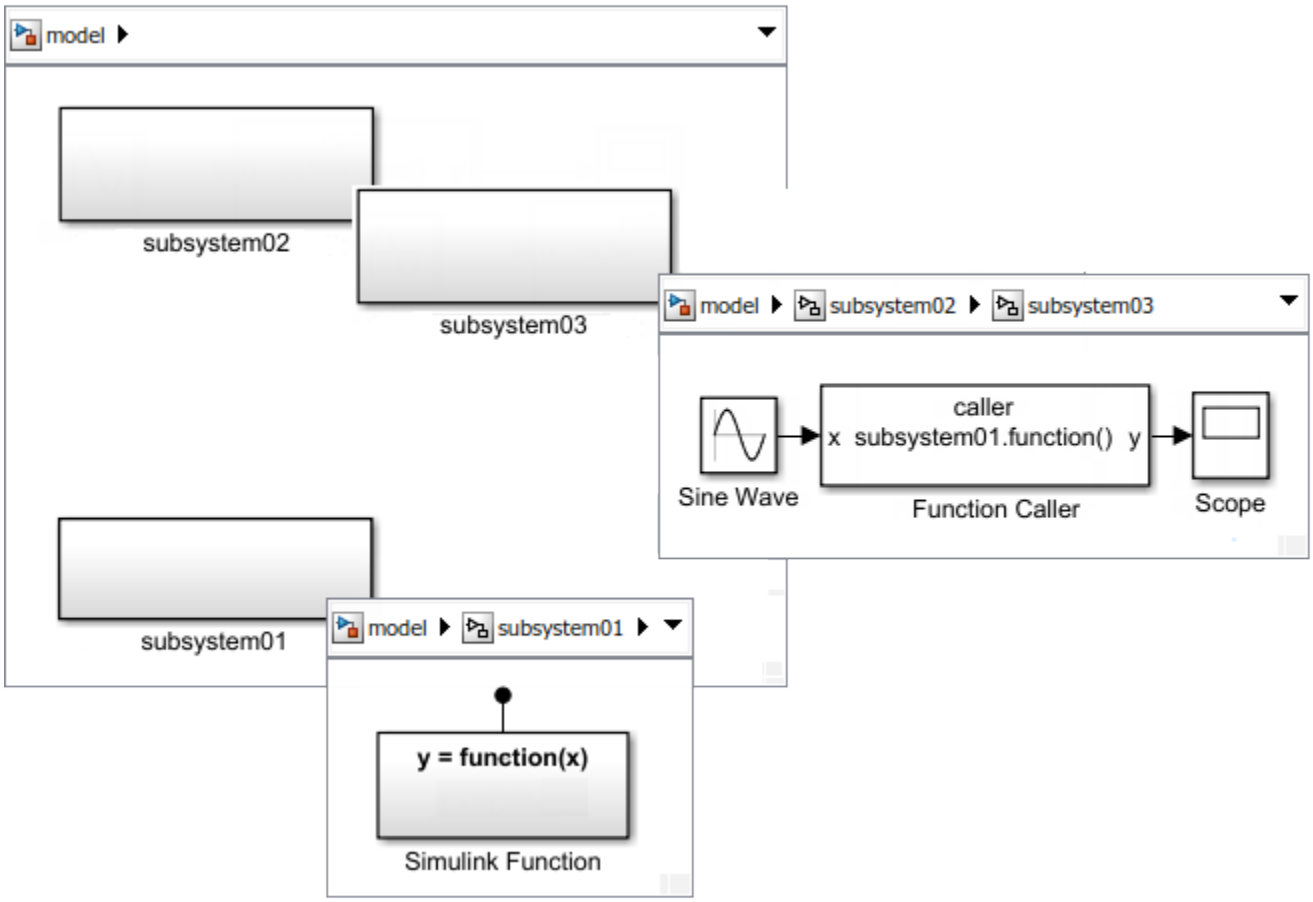


- Function caller is in another subsystem at the same hierarchic level as the function. In this case, the function caller didn't find the Subsystem block with the scoped function in the current subsystem, but it found the Subsystem block in the parent model (Resolution Rule 2).



- Function caller is in another subsystem one or more subsystem levels below the hierarchic level of the function. In this case, the function caller didn't find the Subsystem block with the scoped function in the current subsystem, but it found the Subsystem block in the grandparent model (Resolution Rule 2).

  The function caller hierarchy cannot include a Model block since the function caller cannot cross model reference boundaries.



See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Models” on page 10-157
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Scoped Simulink Function Blocks in Models

The scope of a Simulink function is defined in the context of a model. If you place a Simulink Function block in a model at the root level, the function is scoped to the model by default. The Trigger block **Function visibility** parameter is set to `scoped`. Access the function with a function caller located:

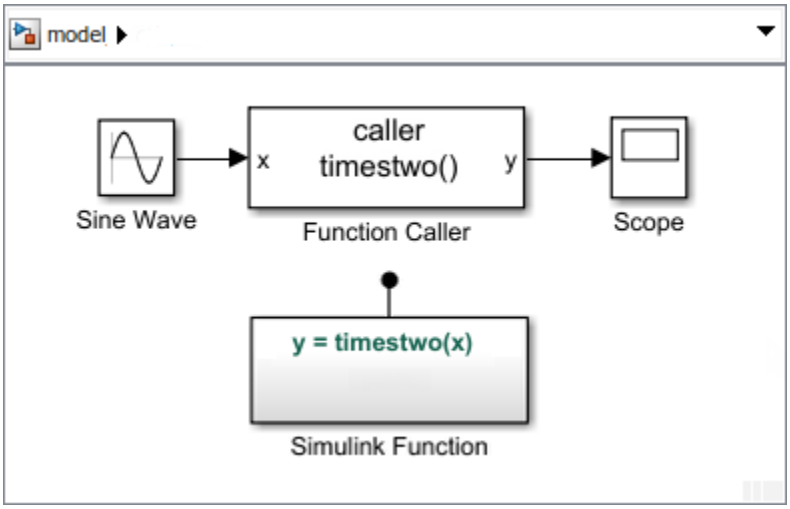
- Within the model hierarchy containing the function. Call the function without qualifying the function name.
- Outside the model. Call the function by qualifying the function name with the model block instance name, not the model file name.

Setting **Function visibility** for a Simulink Function block to `global` allows you to access the function from anywhere in the model or a parent model. As a result, models with a Simulink Function block set to global cannot be multi-instanced because function names must be unique.

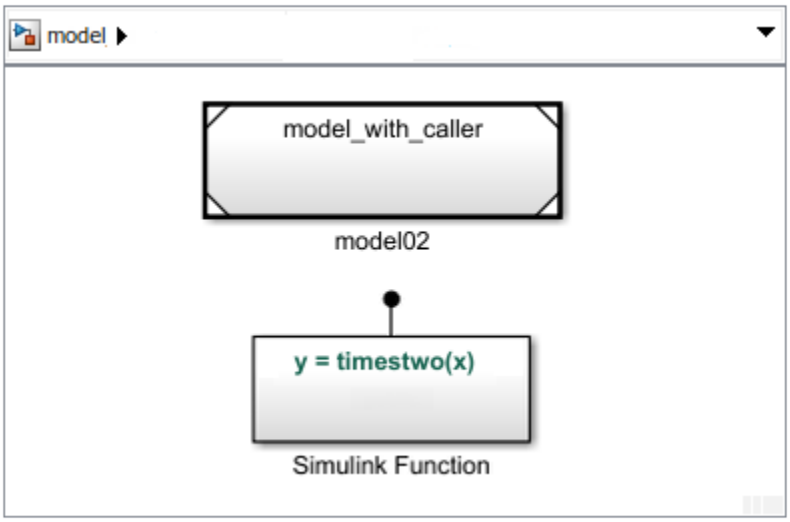
Resolve to a Function Hierarchically

Placing a scoped Simulink Function block within a model at the root level limits access to the function and removes the function name from the global name space.

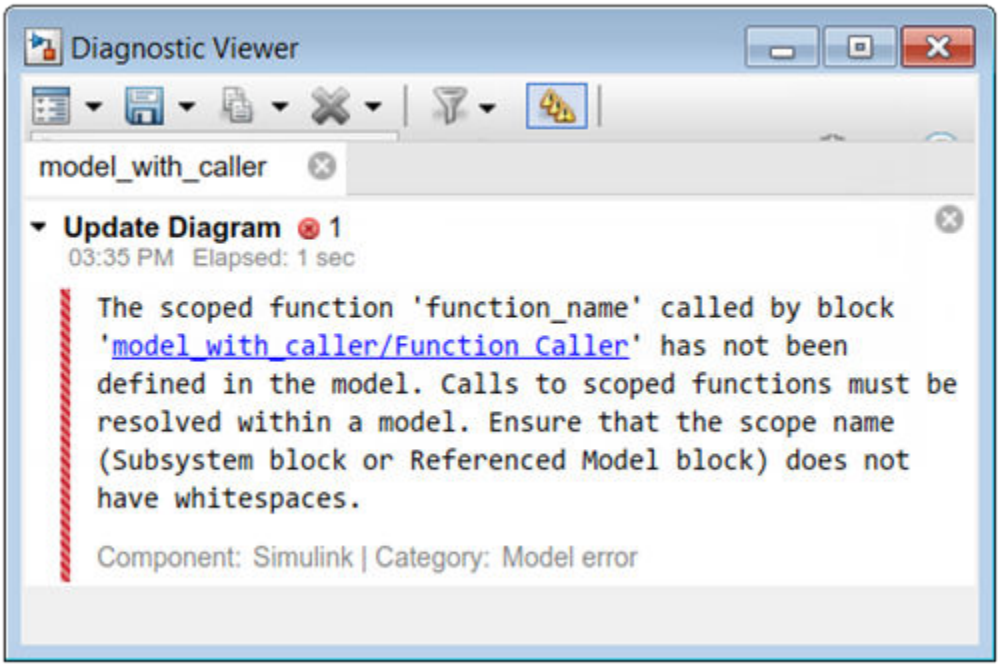
- Function caller located in the current model.



- You cannot place a function caller inside a Model block and the Simulink Function block in the parent model,



If you place a function caller inside a Model block, Simulink displays an error. This error occurs because the model containing the caller does not know the name of the function. Function calls cannot cross model reference boundaries.



Resolve to a Function by Qualification

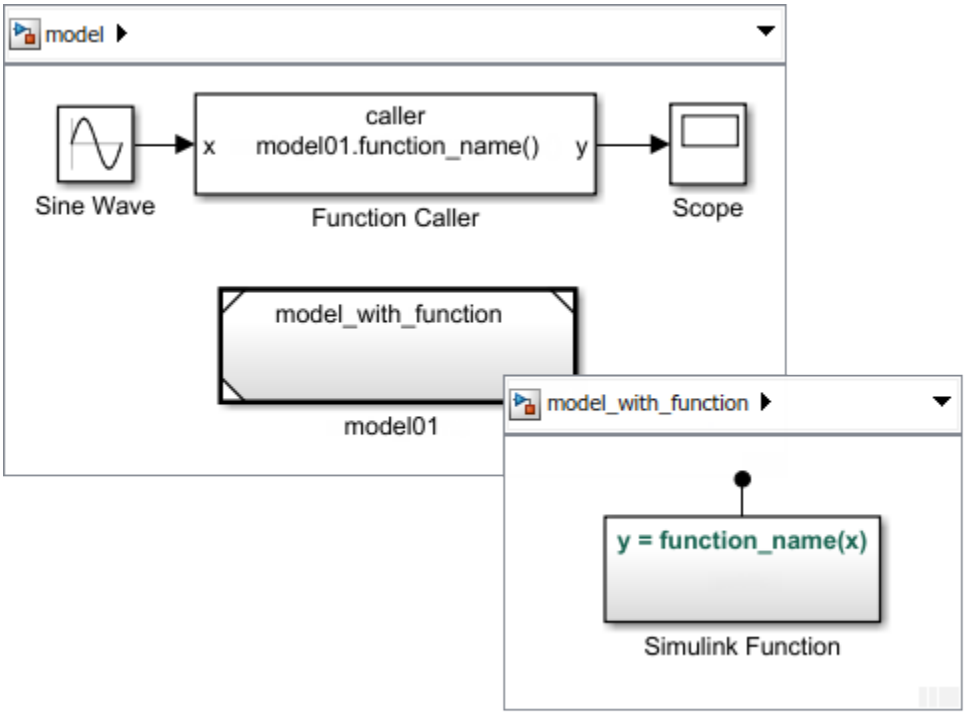
When you place a Simulink Function block in a Model block, the function name is not accessible outside the model. However, you can call a function by qualifying the function name with the Model block name. When a function caller resolves to a qualified function hierarchically, it looks for the Model block containing the function using the following rules:

- Resolution Rule 1: Is the Model block in the current model with the function caller?

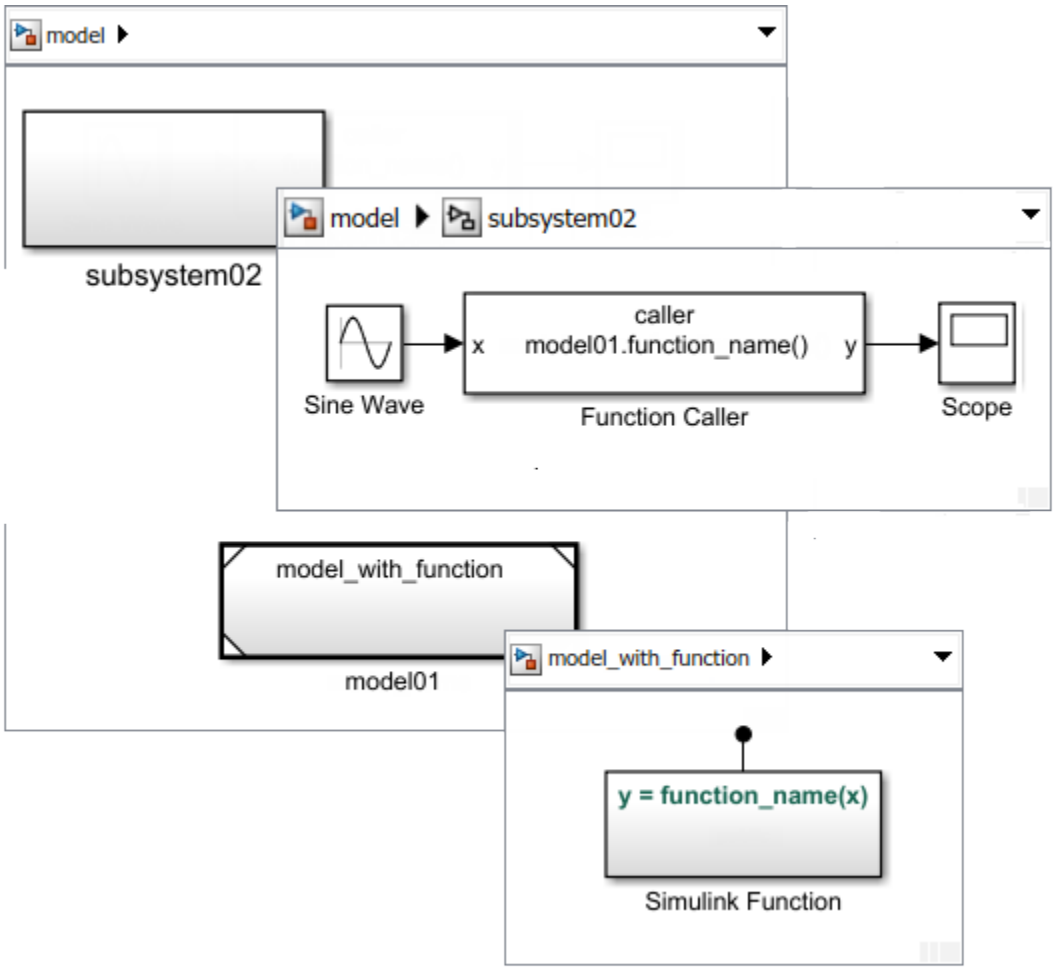
- Resolution Rule 2: If the Model block is not in the current model or subsystem, is the Model block in a parent or grandparent model one or more levels above the function caller?

If a function caller resolves to a Model block with a scoped function, you can call the function by qualifying the function name:

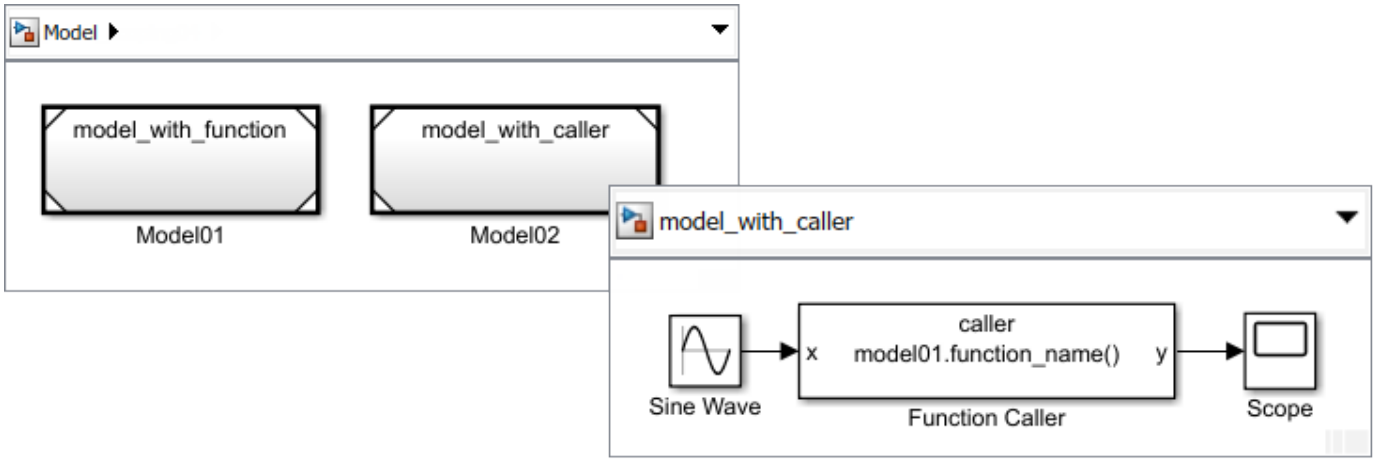
- Function caller located outside of the Model block one hierarchic level above the function. In this case, the function caller finds the Model block with the scoped function in the current model (Resolution Rule 1).



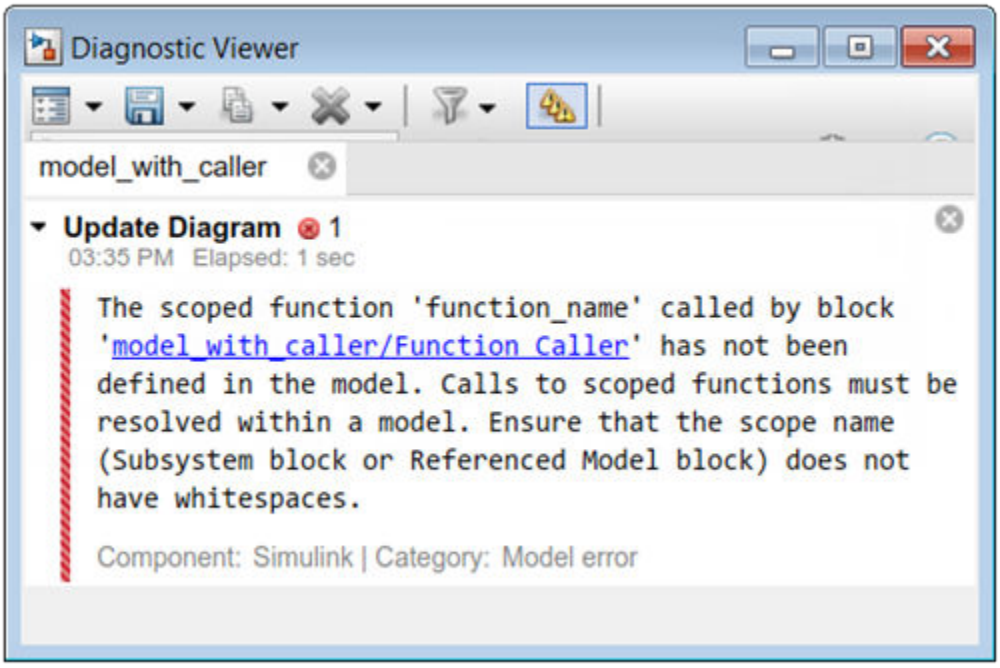
- Function caller in a subsystem at the same hierarchic level as the function. In this case, the function caller didn't find the Model block in the current subsystem, but it found the Model block in the parent model (Resolution Rule 2).



- You cannot place a Simulink Function block in one Model block and the function caller in another Model block.



If you place a Simulink Function block in a referenced model and a function caller in another referenced model, Simulink displays an error. This error occurs because the qualified function name using the Model block name is not visible to the model containing the caller.

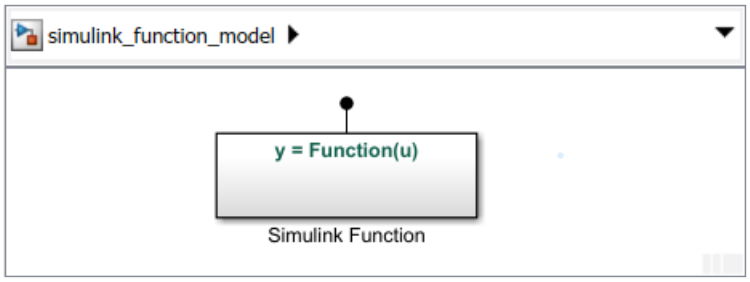


If you want to access the function using this modeling pattern, see the section Function Caller Block in Referenced Model and Function and the section Function Caller in Separate Models in the topic “Simulink Function Blocks in Referenced Models” on page 10-140.

Multi-Instance Modeling with Simulink Functions

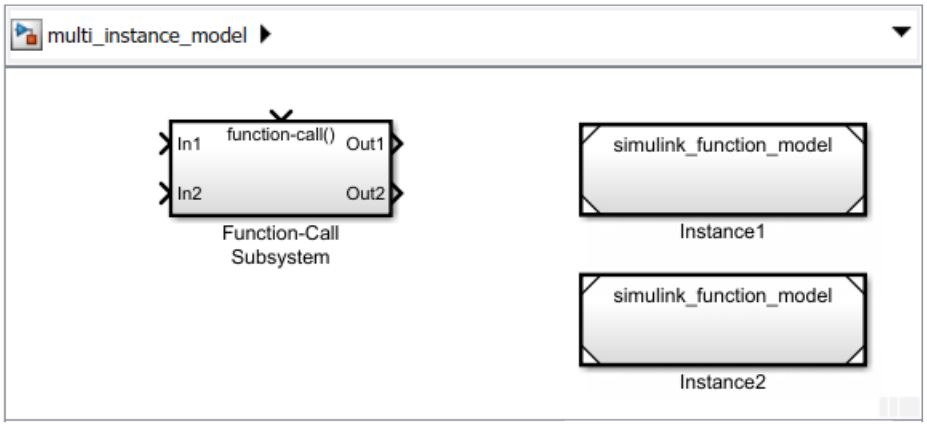
Setting **Function visibility** for a Simulink Function block to `scoped` encapsulates the function within the model, allowing you to multi-instance the model. Adding the model instance name to the function name creates a qualified function name that is unique within the parent model.

1 Create a model containing Simulink Function blocks.

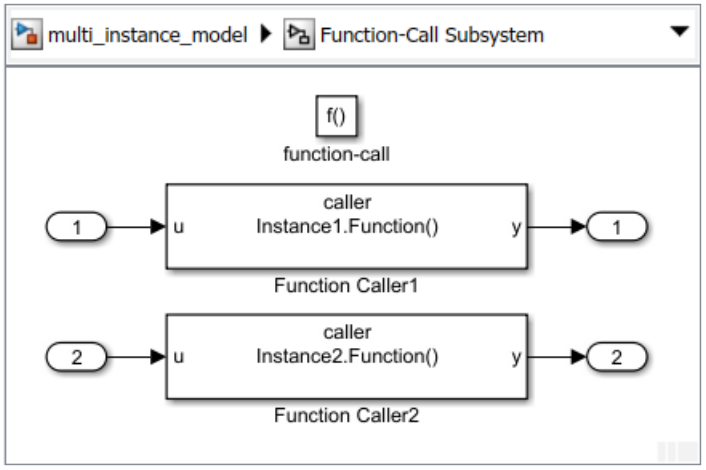


By default, the **Function visibility** parameter for the Trigger block within the Simulink Function block is set to `scoped`.

2 Reference the model with Simulink functions from multiple Model blocks. Add a Function-Call Subsystem block to schedule calls to the functions.

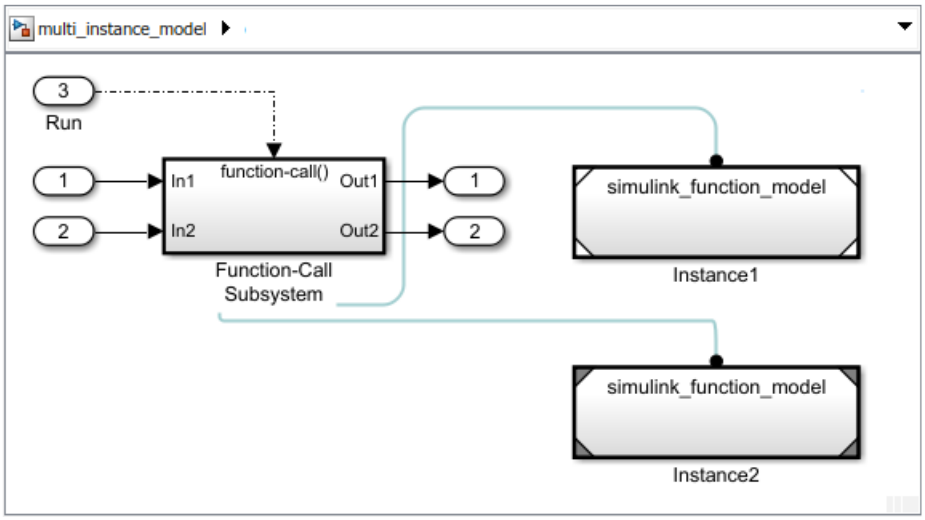


3 Add Function Caller blocks to the Function-Call Subsystem block. Access the function in separate model instances by qualifying the function name with the block name.



4 On the **Debug** tab, select **Information Overlays**  . From the drop-down box, select **Function Connectors**  .

Tracing lines are drawn to help you navigate from a function caller to the function.



For a model using Simulink Function blocks with multiple instances, see “Modeling Reusable Components Using Multiply Instanced Simulink Functions”.

See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Diagnostics Using a Client-Server Architecture” on page 10-164

Diagnostics Using a Client-Server Architecture

In this section...

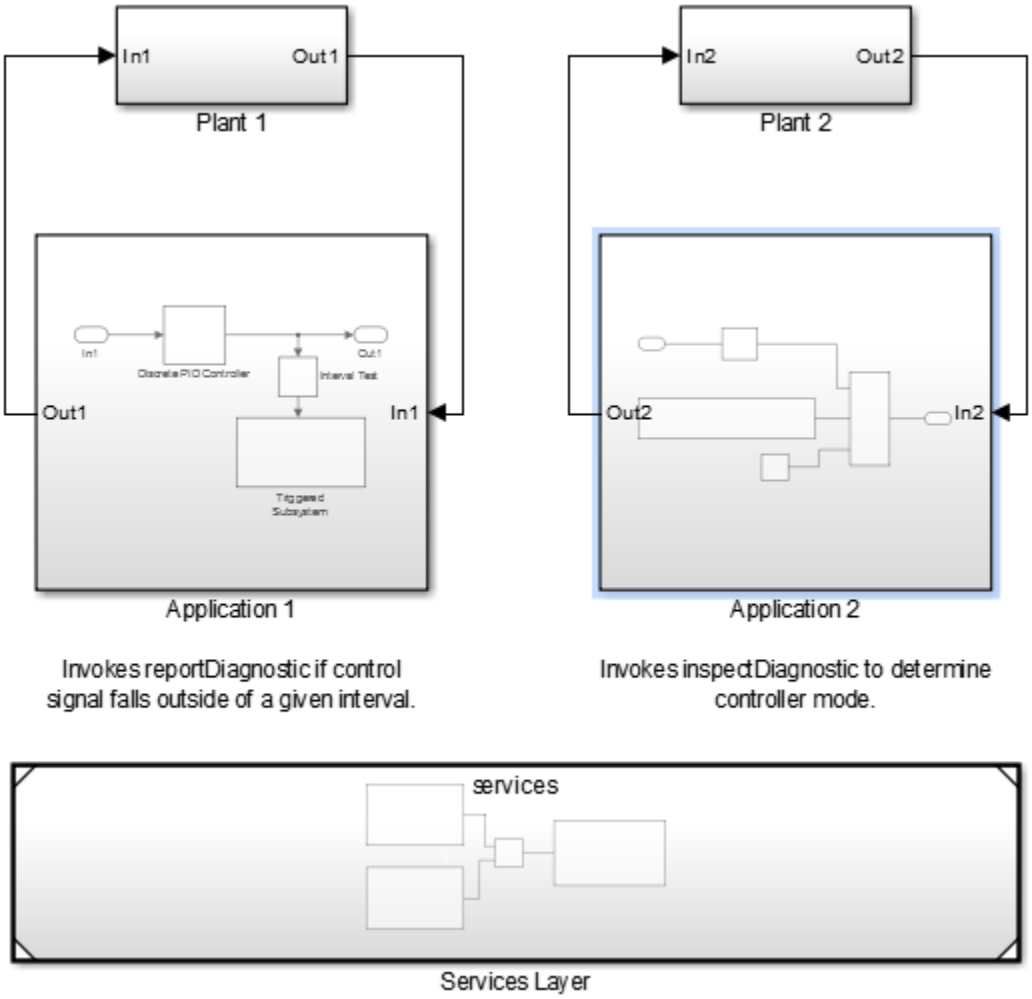
Diagnostic Messaging with Simulink Functions

Use Simulink functions when you define a diagnostic service where callers pass an error code. The service tracks error codes for all errors that occur. One way to implement this service is to use an indexed Data Store Memory block. A diagnostic monitoring system can then periodically check for the occurrence of specific errors and modify system behavior accordingly.

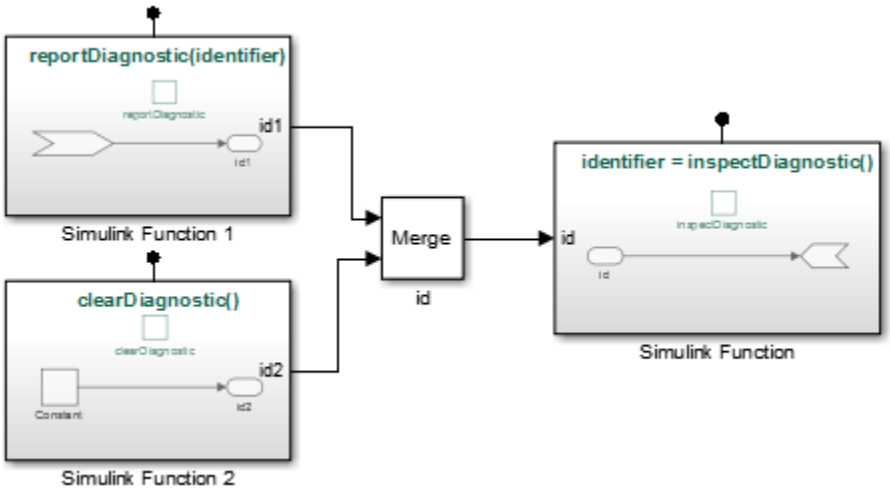
Client-Server Architecture

You can use Simulink Function blocks and Function Caller blocks to model client-server architectures. Uses for this architecture include memory storage and diagnostics.

As an example, create a model of a simple distributed system consisting of multiple control applications (clients), each of which can report diagnostics throughout execution. Since client-server architectures are typically constructed in layers, add a service layer to model the diagnostic interface.



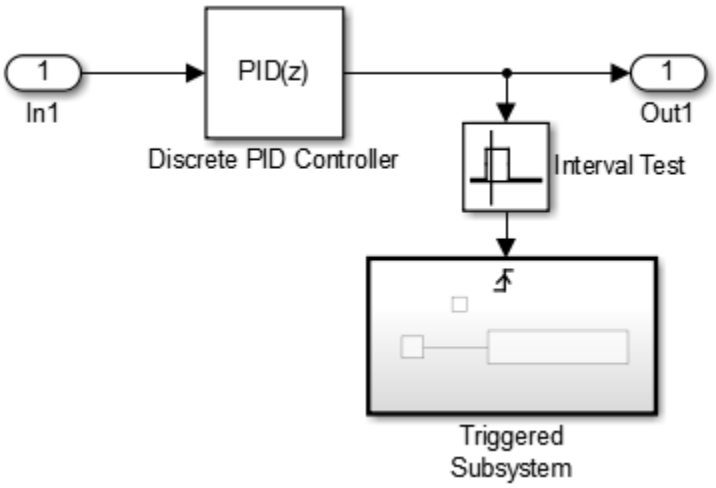
The services (servers), modeled using Simulink Function blocks, are in a separate model. Add the service model to your system model as a referenced model.



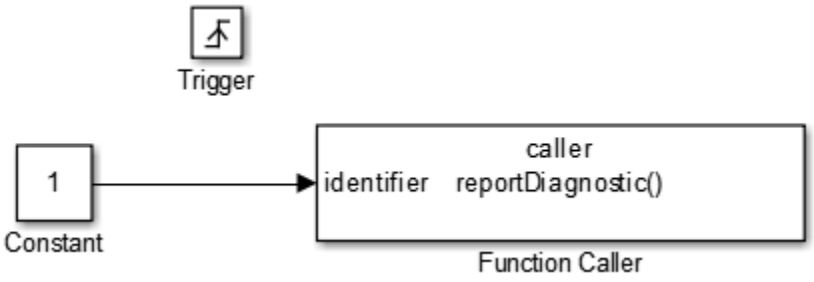
The control applications (clients) interact with the diagnostic interface using Function Caller blocks.

Modifier Pattern

Application 1 reports a diagnostic condition by invoking the `reportDiagnostic` interface within the service layer. The application calls this function while passing in a diagnostic identifier.



The interval test determines when to create a diagnostic identifier.

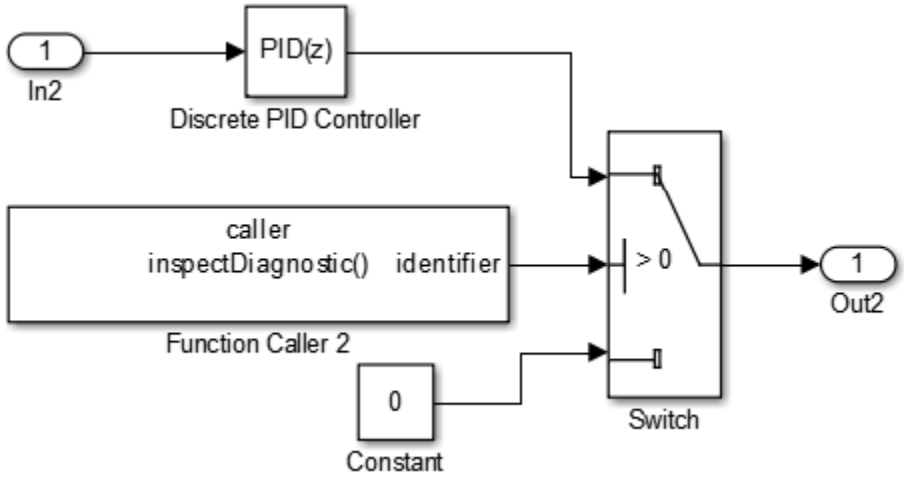


The implementation of the function (Simulink Function 1) tracks the passed-in identifier by transferring the value to a graphical output of the function. A graphical output is a server-side signal that is not part of the server interface but facilitates communication between service functions through function arguments. The value of graphical outputs is held between function invocations.

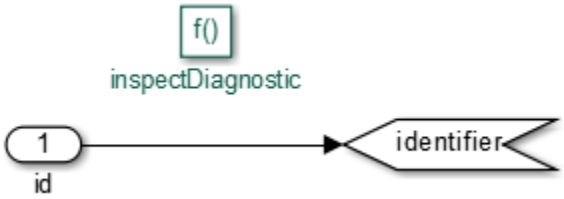
The `reportDiagnostic` function is an example of a modifier pattern. This pattern helps to communication of data from the caller to the function and later computations based on that data.

Observer Pattern

Application 2 invokes the `inspectDiagnostic` interface within the service layer to inspect whether diagnostics were reported.



The implementation of the function (Simulink Function) uses a graphical input (`id`) to observe the last reported diagnostic and transfer this value as an output argument (`identifier`) to the caller. A graphical input is a server-side signal that is not part of the server interface.



The `inspectDiagnostic` function is an example of an observer pattern. This pattern helps to communication of data from the function to the caller.

See Also

Blocks

Argument Inport | Argument Outport | Function Caller | MATLAB Function | Simulink Function

Related Examples

- “Simulink Functions Overview” on page 10-113
- “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121
- “Simulink Function Blocks in Referenced Models” on page 10-140
- “Scoped and Global Simulink Function Blocks Overview” on page 10-147
- “Scoped Simulink Function Blocks in Subsystems” on page 10-150
- “Scoped Simulink Function Blocks in Models” on page 10-157

Using Initialize, Reset, and Terminate Functions

In this section...

Some blocks maintain state information that they use during a simulation. For example, the Unit Delay block uses the current state of the block to calculate the output signal value for the next simulation time step.

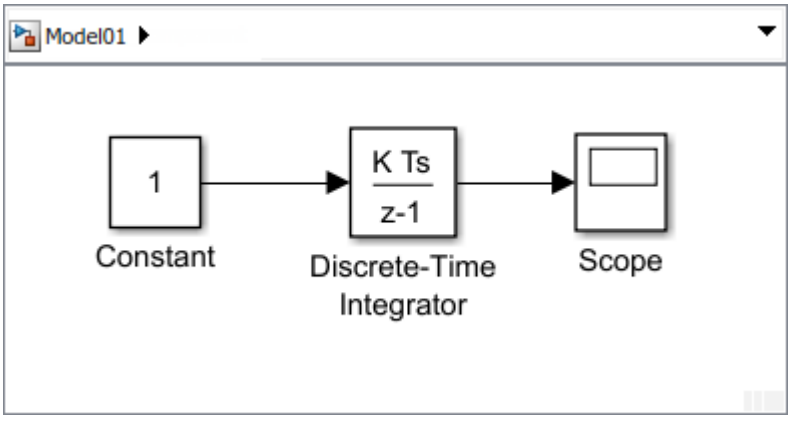
Subsystem blocks have default initialize and termination routines. You can add custom routines to the default routines using Initialize Function and Terminate Function blocks to change or read block states. These function blocks contain:

- Event Listener blocks that execute the combined routines when receiving an initialize or terminate function-call event.
- State Writer blocks to initialize the block state and State Reader blocks to read the state.

Create Model Component with State

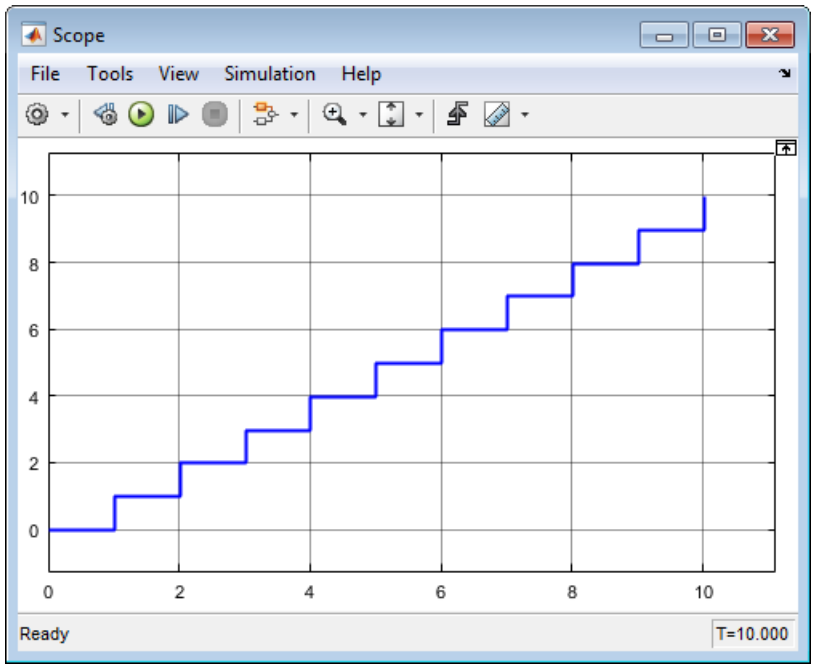
You can define model algorithms using Simulink blocks. In this example, a single Discrete-Time Integrator block defines the algorithm for integrating an input signal.

1. Open a new Simulink model. Save this model with the name `Model01`.
2. Add a Discrete-Time Integrator block. Verify the default parameter values are `1.0` for **Gain value**, `0` for **Initial condition**, `State (most efficient)` for **Initial condition setting**, and `-1` for **Sample time**.
3. Connect a Constant block to the input of the Discrete-Time Integrator block to model an input signal. Connect a Scope block to the output signal.



4. Open the Configuration Parameters dialog box. Set the simulation parameters for the Solver **Type** to `Fixed-step`, **Solver** to `auto`, and **Fixed-step size** to `1`.

5 Open the Scope block, and then run simulation. The output signal increases by 1 at each time step.



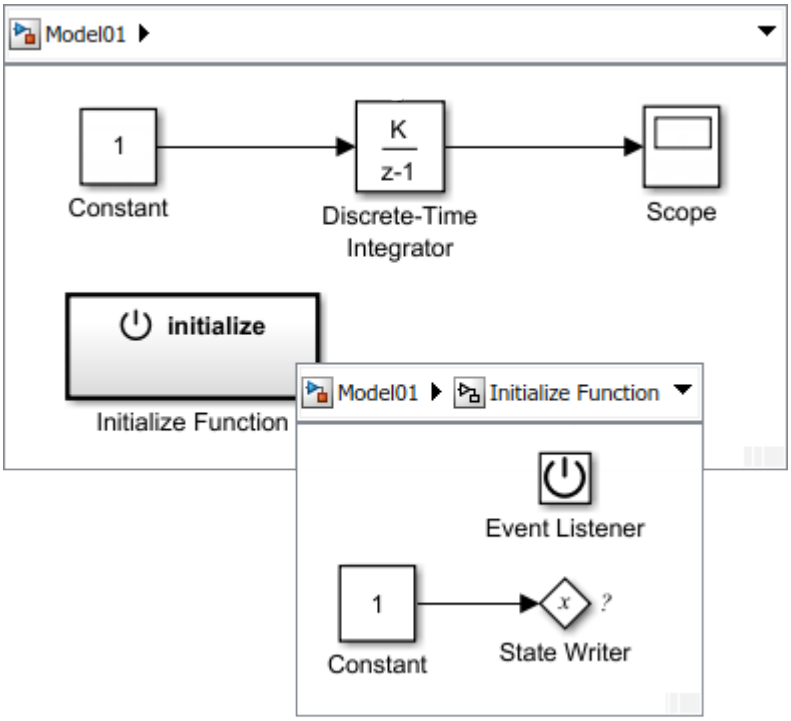
Initialize Block State

Some model algorithms contain states that you can initialize. For example, with an algorithm that reads a sensor value, you can perform a computation to set the initial sensor state.

At the beginning of a simulation, initialize the state of a block using a State Writer block. To control when initialization occurs, use an Initialize Function block that includes the State Writer block.

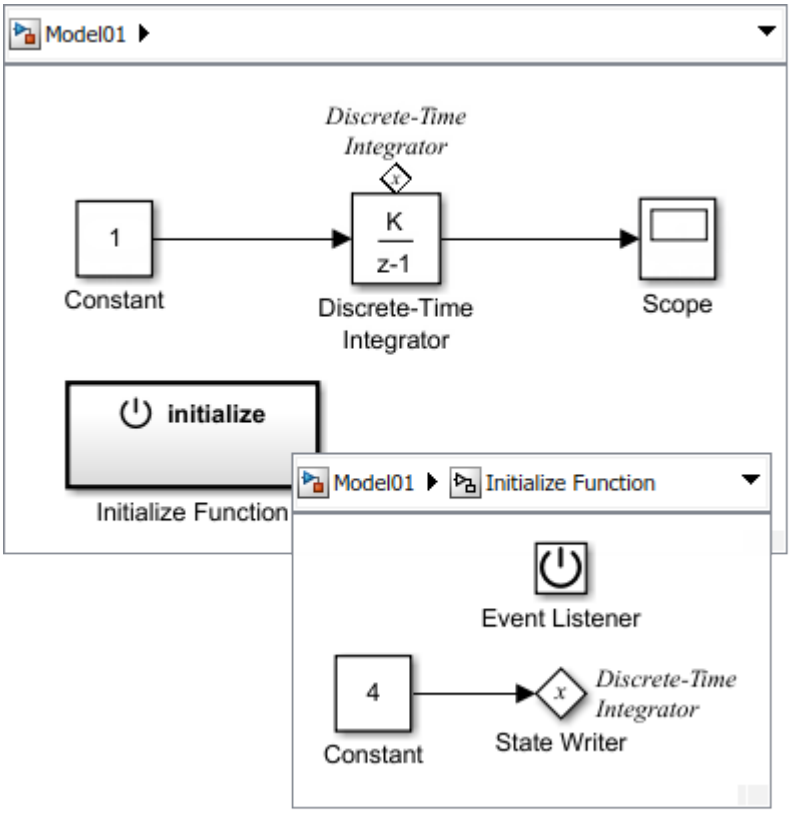
1 Add an Initialize Function block.

By default, the Initialize Function block includes an Event Listener block with the **Event type** parameter set to `Initialize`. The block also includes a State Writer block, and a Constant block as a placeholder for the source of the initial state value.



2 Model initial conditions. In this example, set the **Constant value** parameter for the Constant block to 4.

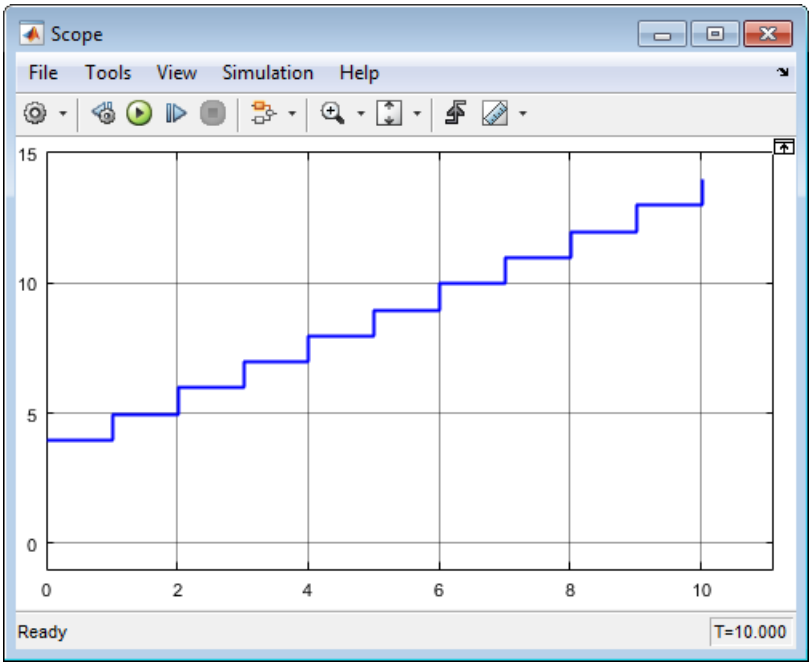
3 Connect the state writer with the state owner. Open the State Writer dialog box. Expand the State Owner Selector Tree, select `Discrete-Time Integrator`, and then click **Apply**.



The State Writer block displays the name of the state owner block. The state owner block displays a tag indicating a link to a State Writer block. If you click the label above the tag, a list opens with a link for navigating to the State Writer block.

4 Run simulation to confirm that your model simulates without errors.

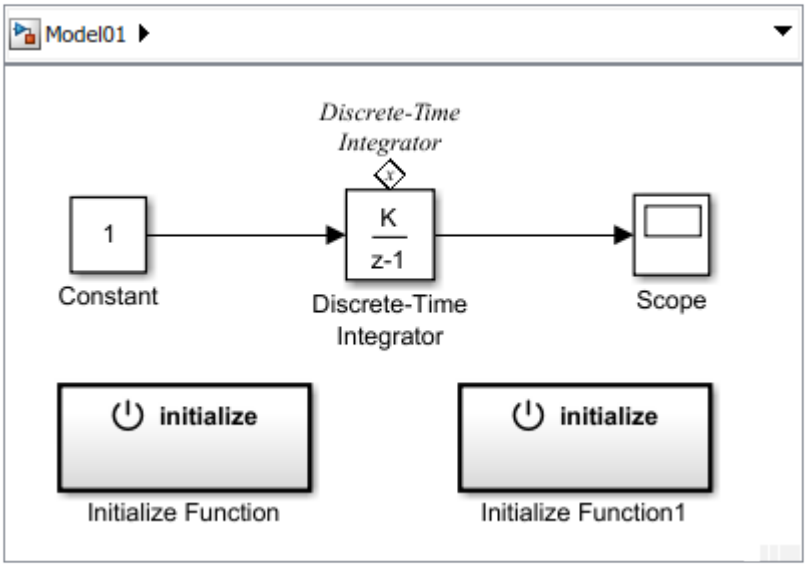
The Initialize Function block executes at the beginning of a simulation. The output signal starts with and initial value of 4 and then increases by 1 until the end of the simulation.



Reset Block State

During a simulation, you can reset the state of a block using a State Writer block. To control when reset occurs, use an Initialize Function block that you reconfigure to a Reset Function block.

1 Add an Initialize Function block.

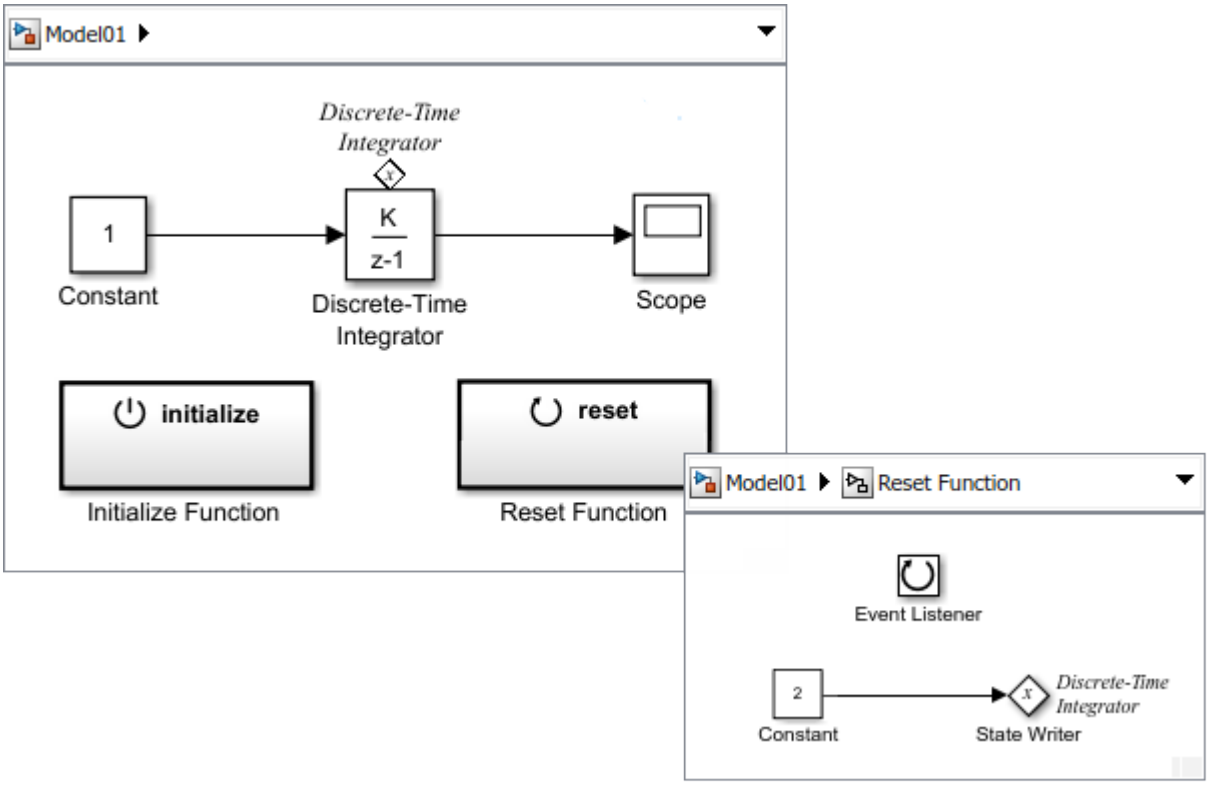


2 Open the new Initialize Function block.

3 Configure block for reset. Open the Block Parameter dialog box for the Event Listener block. From the **Event type** drop-down list, select `Reset`. In the **Event name** box, enter an event name. For example, enter `reset`. Close the dialog box.

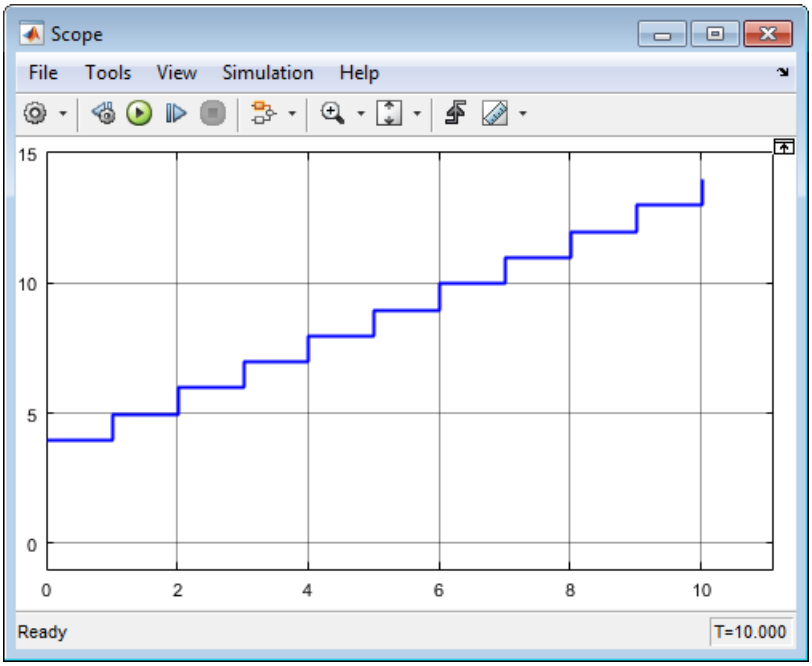
4. Model reset conditions. In this example, set the **Constant value** parameter for the Constant block to 2.
5. Connect state writer with the state owner. Open the State Writer dialog box. Expand the State Owner Selector Tree, select `Discrete-Time Integrator`, and then click **Apply**.
6. Navigate to the top level of `Model01`. Rename the block from `Initialize Function1` to `Reset Function`.

   After updating your model, the event name for the Reset Function block is displayed on the face of the block.



   If you click above the tag, a list opens with a link for navigating to the State Writer blocks located in the Initialize Function block and the Reset Function block.
7. Run a simulation to confirm that your model simulates without errors.

   The Reset Function block does not execute during the simulation. It needs an function-call event signal.



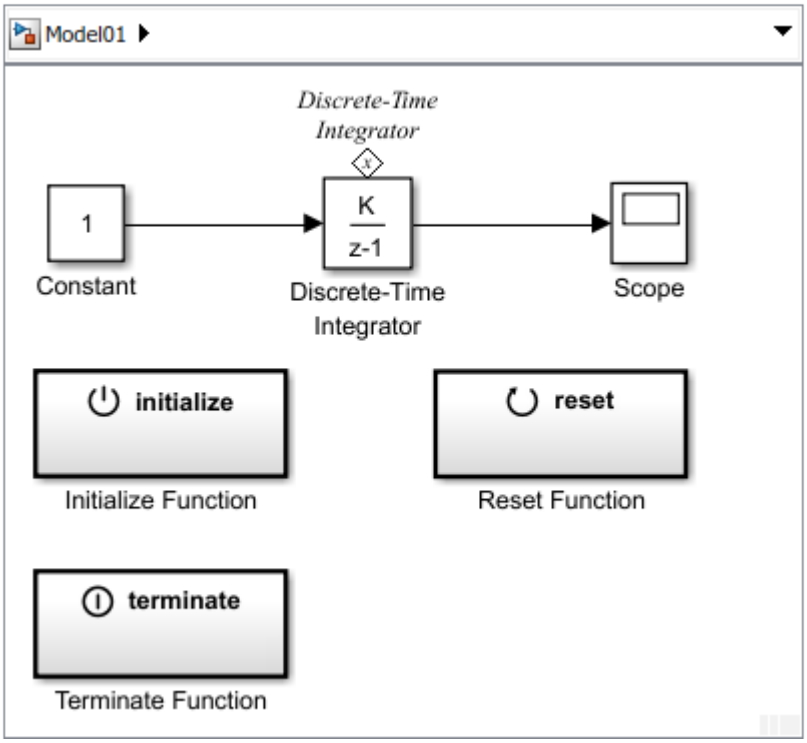
To create an function-call event signal for the Reset Function block, see “Create Test Harness to Generate Function Calls” on page 10-180.

Read and Save Block State

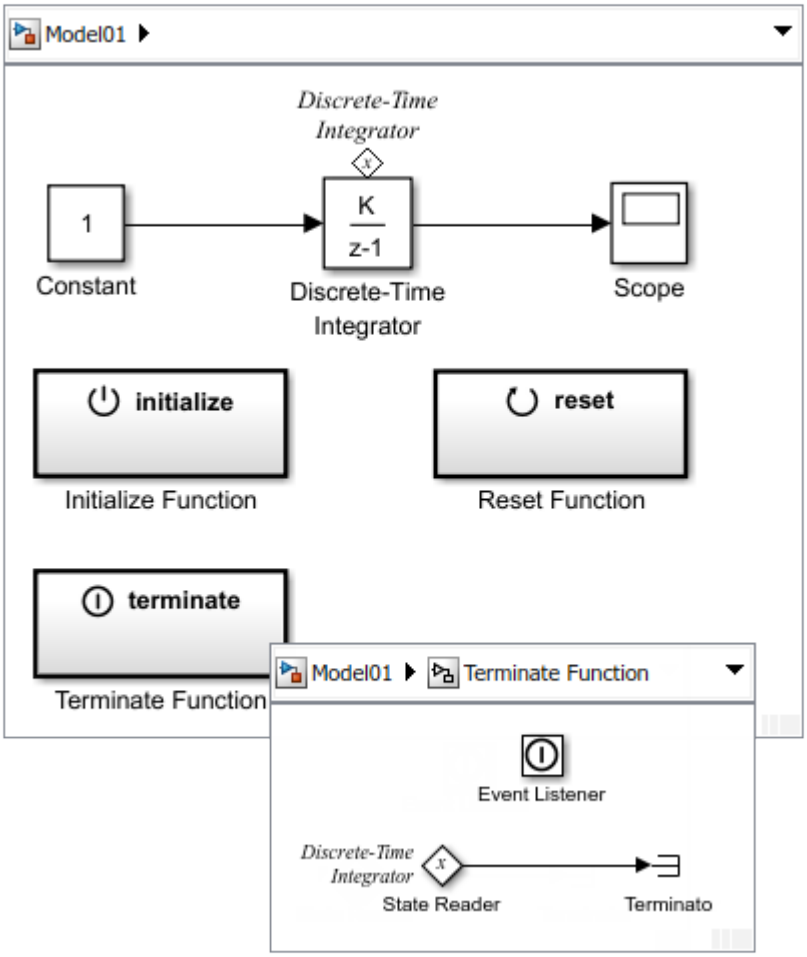
At the end of a simulation, you can read the state of a block, and save that state.

1 Add a Terminate Function block.

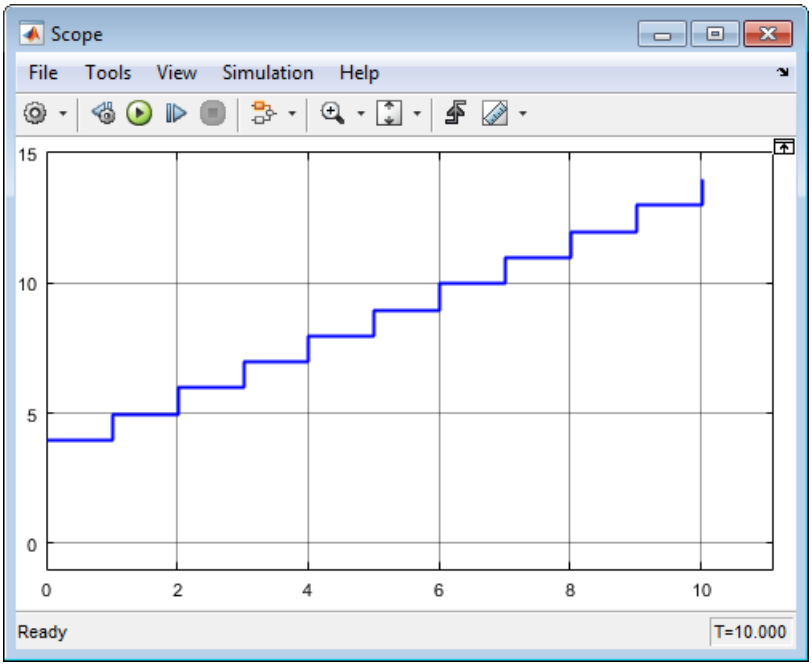
By default, the Terminate Function block includes an Event Listener block with the parameter **Event type** set to `Terminate`. The block also includes a State Reader block, and a Terminator block as a placeholder for saving the state value.



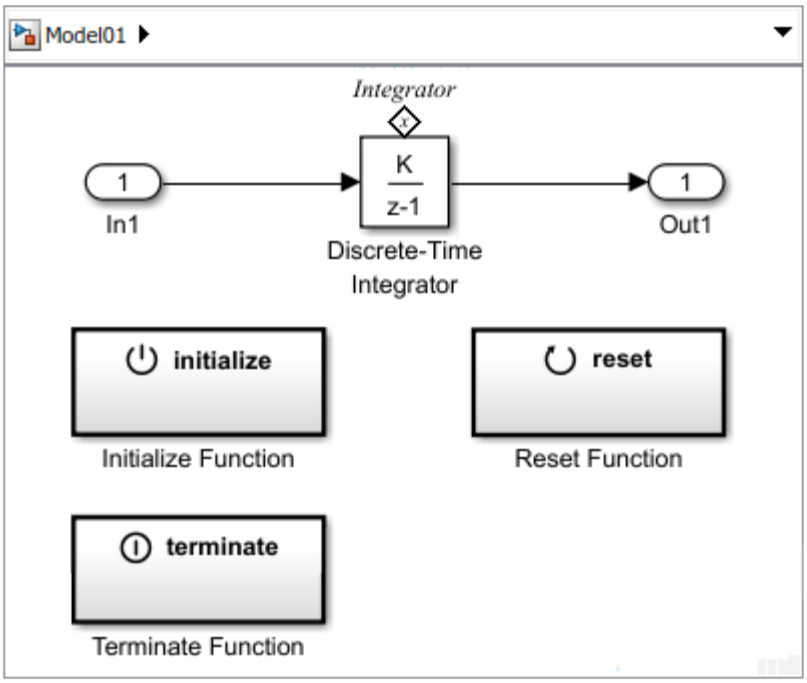
2 Connect the state reader with the state owner. Open the State Reader dialog box. From the State Owner Selector Tree, select `Discrete-Time Integrator`, and then click **Apply**.



3 Run a simulation to confirm that your model simulates without errors. The Terminate Function block executes at the end of a simulation.



4 Delete the blocks that you added for testing. Replace the Constant block with an Inport block and the Scope block with an Outport block.



Prepare Model Component for Testing

Make the following changes to avoid simulation errors when the component model is placed in an export-function model for simulation testing.

1. Open the Block Parameters dialog box for the Discrete-Time Integrator block. Set **Integrator method** to `Accumulation:Forward Euler`.
2. Open the Model Configuration Parameters dialog box. Confirm the solver **Type** is set to `Fixed-step` and **Solver** is set to `auto`. Change the **Fixed-step size** from `1` to `auto`.

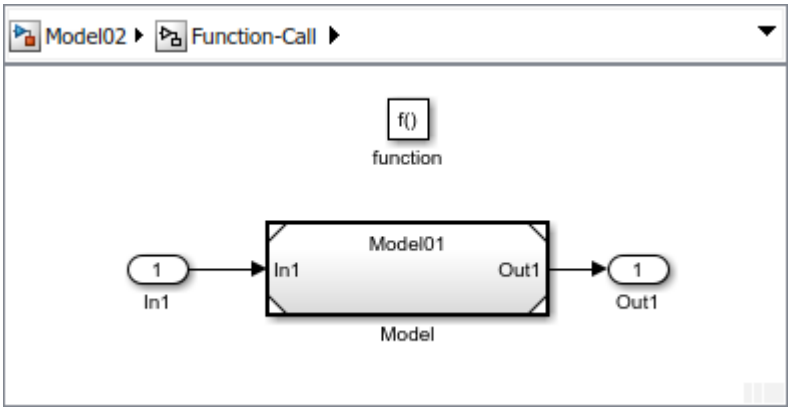
   This change avoids a simulation error caused by having multiple sample times in a Function-Call Subsystem.

Create an Export-Function Model

Placing a model component in a test harness for testing the initialize, reset, and terminate functions requires the model to follow export-function rules. See “Export-Function Models Overview” on page 10-97 and “Create Test Harness to Generate Function Calls” on page 10-180.

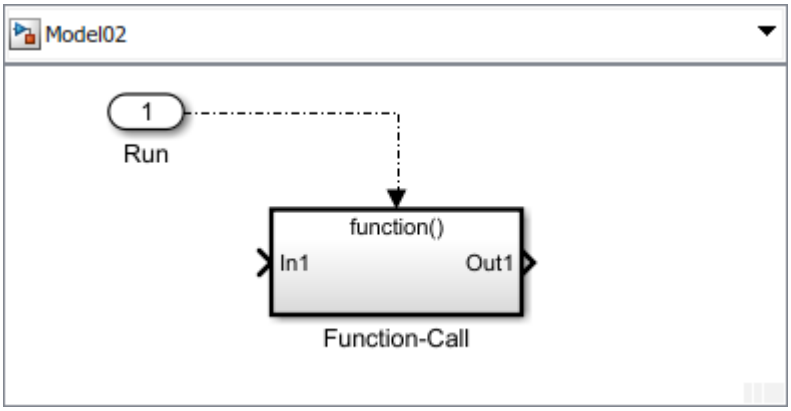
To create an export-function model, place the model component in a Function-Call Subsystem block using a Model block. Connect input and output ports from the model to the subsystem input and output ports.

1. Create a Simulink model. Save this model with the name `Model02`.
2. Open the Configuration Parameters dialog box. Set the simulation parameter for the Solver **Type** to `Fixed-step`. Confirm **Solver** is set to `auto` and **Fixed-step size** is set to `auto`.
3. Add a Function-Call Subsystem block. Open the subsystem by double-clicking the block.
4. Add a Model block to the subsystem and set **Model name** to `Model01`. Add Inport and Outport blocks.



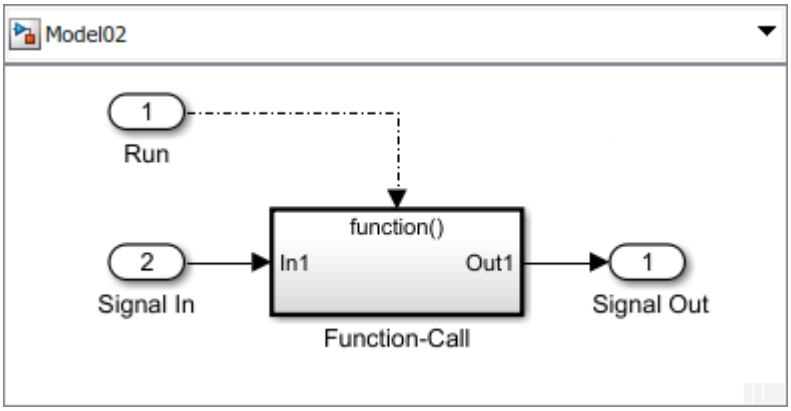
5. Navigate to the top level of the model.
6. Add an Inport block. This block is the control signal for executing the subsystem. Change the block name to `Run` and connect it to the `function()` port.

   Open the Inport block dialog box and on the Signal Attributes tab, select the **Output function call** check box.



7 Add a second Inport block and rename it to `Signal In`. Connect it to the `In1` port of the subsystem. This block is the signal for the integration algorithm.

   Add an Outport block, rename it to `Signal Out`, and then connect it to the `Out1` port of the subsystem. This block is the integrated signal.



8 Open the Configuration Parameters dialog box. On the Model Referencing pane, set the **Total number of instances allowed per top model** to one.

9 Update your model and confirm that there are no errors by pressing **Ctrl-D**.

The next step is create a test harness. See “Create Test Harness to Generate Function Calls” on page 10-180.

See Also

Blocks

Event Listener | Initialize Function | Parameter Writer | Reset Function | State Reader | State Writer | Terminate Function

Related Examples

- “Create Test Harness to Generate Function Calls” on page 10-180
- “Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)
- Initialize and Terminate Functions video

Create Test Harness to Generate Function Calls

In this section...

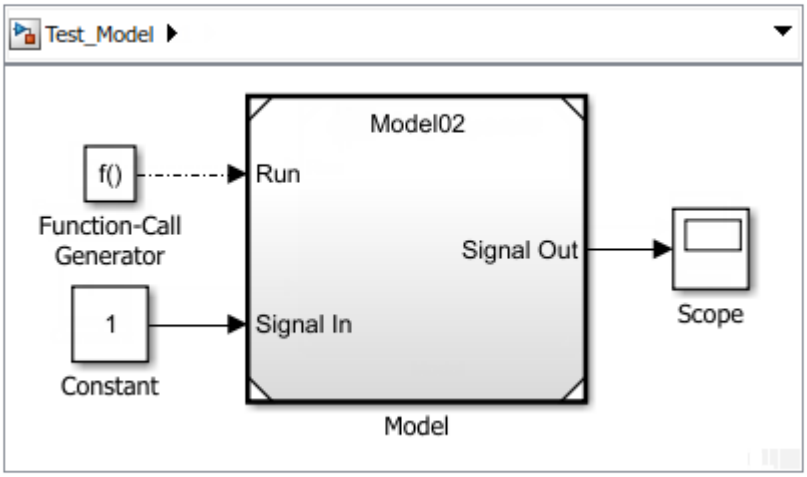
After you create a model component to initialize, reset, and terminate the state of blocks (see “Using Initialize, Reset, and Terminate Functions” on page 10-168), you can place the model in a simulation test harness. A test harness is a Simulink model that you use to develop, test, and debug a model component.

To create the test harness, reference the export-function model containing the model component in a new model, and then add a Stateflow chart to model a function-call event scheduler.

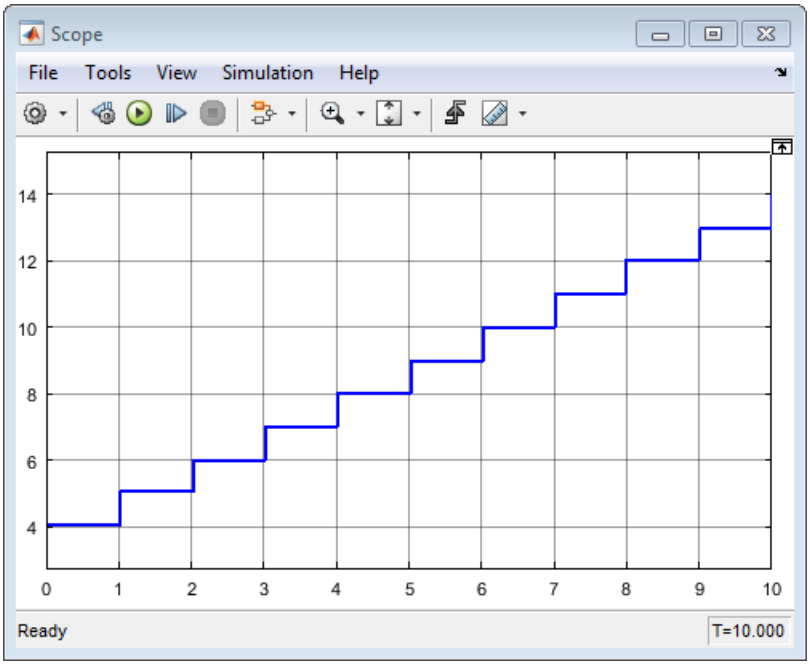
Reference the Export-Function Model

The export-function model contains the model component for testing. To create the export function model, see “Create an Export-Function Model” on page 10-178.

1. Create a Simulink model. Save this model with the name `Test_Model`.
2. Set configuration parameters for solver **Type** to `Fixed-step`, Solver to `auto`, and **Fixed-step size** to `1`.
3. Add a Model block. Open the Block Parameters dialog box. In the **Model name** text box, enter the name of your export-function model. In this example, enter `Model02`.
4. Test the referenced model component by connecting a Function-Call Generator block to the `Run` port. Connect a Constant block to the `Signal In` port and a Scope block to the `Signal Out` port.

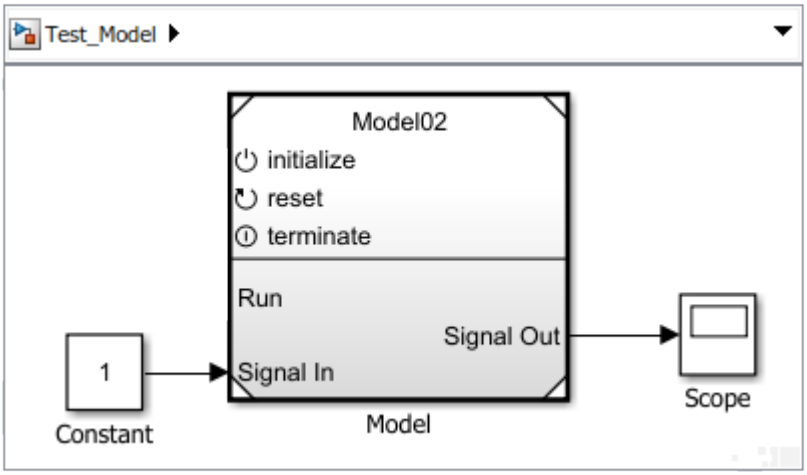


5. Run simulation to verify your model simulates correctly from the parent model. When the model is simulated without function-call event ports, the Initialize Function block executes at the beginning of a simulation and the Terminate Function block executes at the end of the simulation.



6 Expose function-call input ports on the model block. Right-click the Model block and select **Block Parameters**. In the Block Parameters dialog box, select the **Show model initialize port**, **Show model reset port**, and **Show model terminate port** check boxes.

7 Delete the Function-Call Generator block and update the model by pressing **Ctr-D**.



When you activate the initialize function-call input port on a Model block, the model has to receive an initialize function call on the `initialize` port before it can execute. The reception of a function call triggers the execution of the default model initialize routine, and then the execution of the Initialize Function block contents.

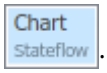
The reception of a function call on the `Reset` port triggers the execution of the Reset Function block contents.

The reception of a function call on the `Terminate` port triggers the execution of the Terminate Function block contents, and then the execution of the default model terminate routine. The

model then stops running. To execute the model again, you have to reinitialize the model by sending an function-call event to the `initialize` port.

Model an Event Scheduler

Use a Stateflow chart to model an event schedule and generate the initialize and terminate function call signals.

1. Add a Stateflow chart. Click the model diagram and start typing `Chart`. From the search list, select 

.
2. Open the chart and add two state blocks, one above the other.
3. Add a default transition and connect it to the top state block. Edit the label:

   ```
   {step = 0}
   ```
4. Add a transition from the top block to the bottom block. Edit the label:

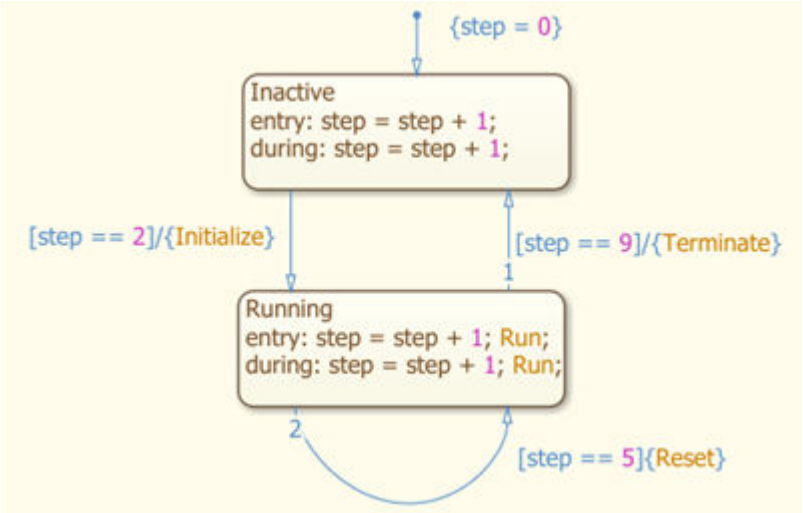
   ```
   [step == 2]/{Initialize}
   ```
5. Add a transition from the bottom block and back to the bottom block. Edit the label:

   ```
   [step == 5]/{Reset}
   ```
6. Add a transition from the bottom block to top block. Edit the label:

   ```
   [step == 9]/{Terminate}
   ```
7. Edit the content of the top block:

   ```
   Inactive
   entry: step = step + 1;
   during: step = step + 1;
   ```
8. Edit the content of the bottom block:

   ```
   Running
   entry: step = step + 1; Run;
   during: step = step + 1; Run;
   ```



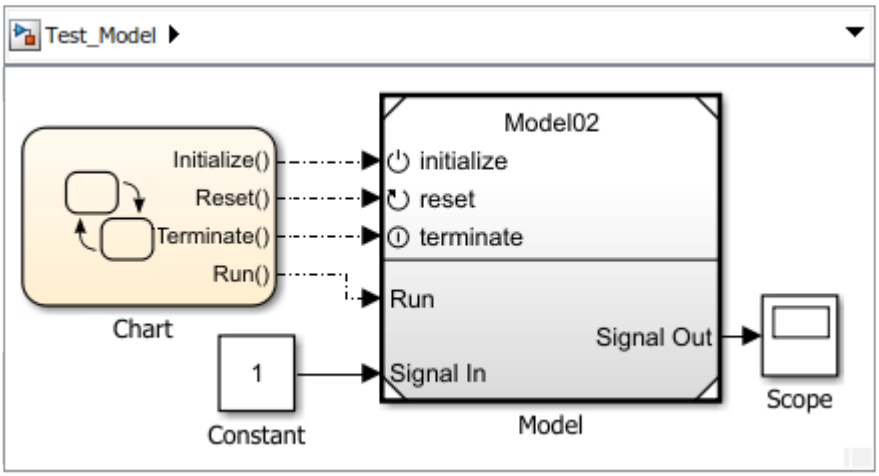
Connect Chart to Test Model

Create function-call output ports on the chart to control and run the model component.

1. Open Model Explorer. On the **Modeling** tab and from the **Design** section, select **Model Workspace**  ..
2. Create index variable. From the menu, select **Add** > **Data**. In the Data dialog box, enter `Step` for the **Name**.
3. Create function-call output ports. For each function-call event you create, select **Add** > **Event** and in the Event dialog box, enter, and select the following values.

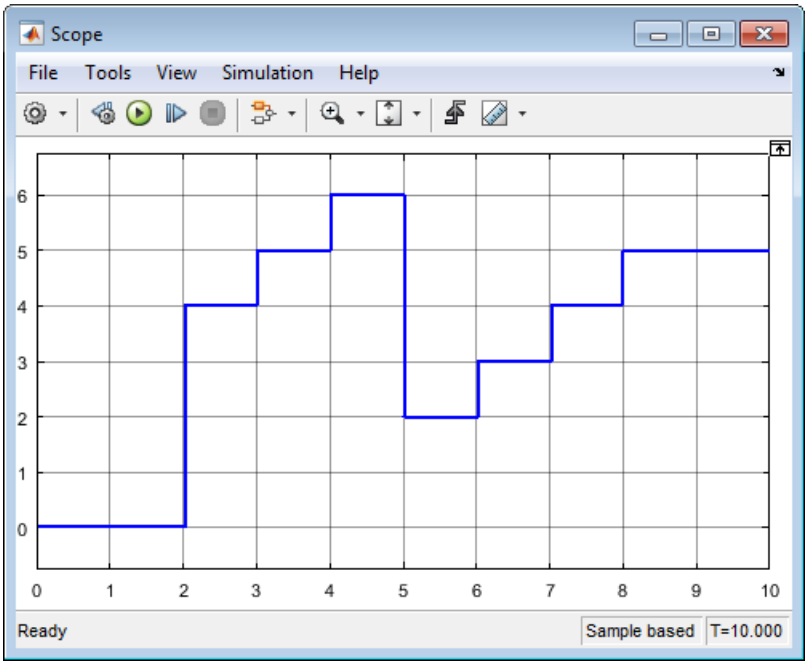
Enter in Event Text Box	Set Scope	Set Trigger
`Initialize`	`Output to Simulink`	`Function call`
`Reset`	`Output to Simulink`	`Function call`
`Terminate`	`Output to Simulink`	`Function call`
`Run`	`Output to Simulink`	`Function call`

4. Navigate to the top level of the model. Connect the `Initialize`, `Reset`, `Terminate`, and `Run` ports on the chart to the `initialize`, `reset`, `terminate`, and `Run` input ports on the Model block.



5. Run simulation.

   The model cannot execute until the second time step, when the block state is initialized to 4. At the fifth time step, a reset function call to the reset port triggers the Reset Function block to execute. At the ninth time step, the subsystem stops executing, and the block state remains constant.



If the model receives a function call to run before an initialize function call, a simulation error occurs.

See Also

Blocks

Event Listener | Initialize Function | Parameter Writer | Reset Function | State Reader | State Writer | Terminate Function

Related Examples

- “Using Initialize, Reset, and Terminate Functions” on page 10-168
- Initialize and Reset Parameter Values on page 10-185
- Initialize and Terminate Functions video

Initialize and Reset Parameter Values

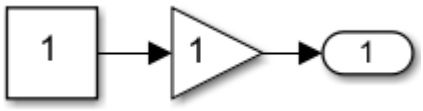
During a simulation, you can respond to an event such as reading an environment sensor value, and then update an algorithm with a new parameter value.

The Initialize Function and Reset Function blocks can respond to events while the Parameter Writer block can change parameter values. The Parameter Writer block does not write directly to block parameter values. Instead, it changes block parameter values in a referenced model by writing to instance parameters belonging to the Model block.

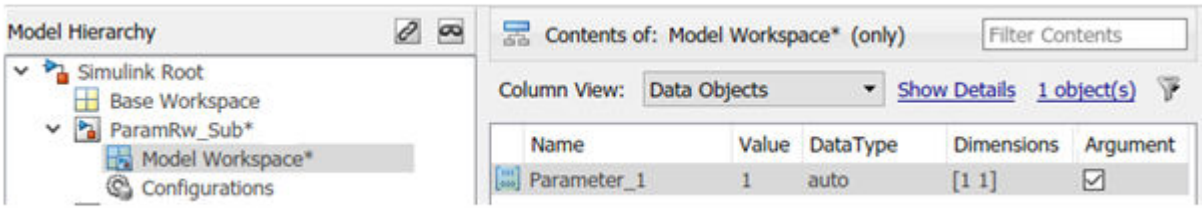
Using the Parameter Writer Block

The following example shows how to use the Parameter Writer block to change a parameter value for a Gain block.

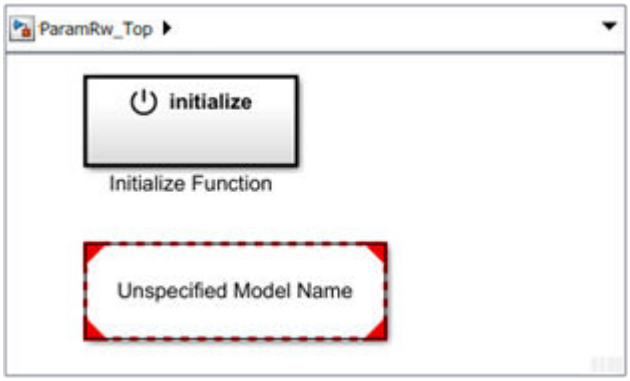
1. Create a model with a writable parameter, that is a block parameter you can define with a model parameter. In the example, add a Constant, Gain, and Outport block to a new model. Connect blocks. Save the model with the name `ParamRw_Sub`.



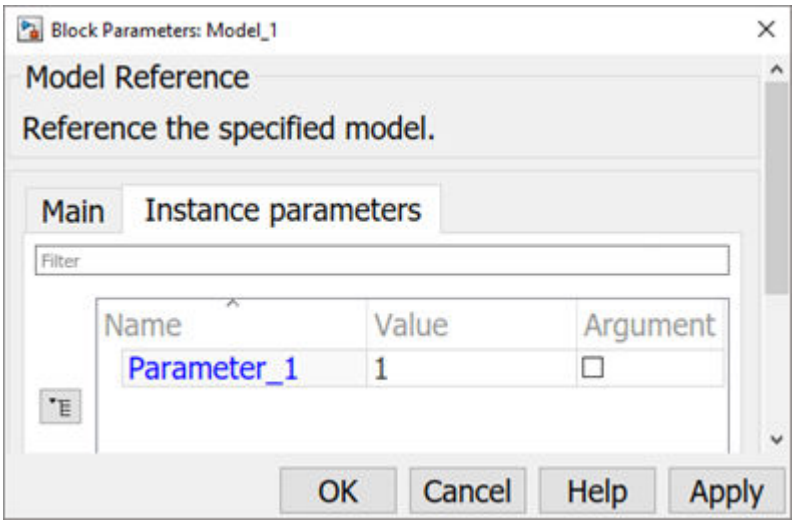
2. Add a Simulink parameter to the Model Workspace. On the **Modeling** tab and from the **Design** section, select **Model Workspace**  . From the Model Explorer menu, select **Add** > **Simulink Parameter**. Set **Name** to `Parameter_1` and **Value** to `1`. Select the **Argument** check box. Click **Apply**.



3. Open the Gain block parameter dialog. Set **Gain** to `Parameter_1`.
4. Create a model that initializes the parameter. Add an Initialize Function and Model block to a new model. Save the model with the name `ParamRw_Top`.

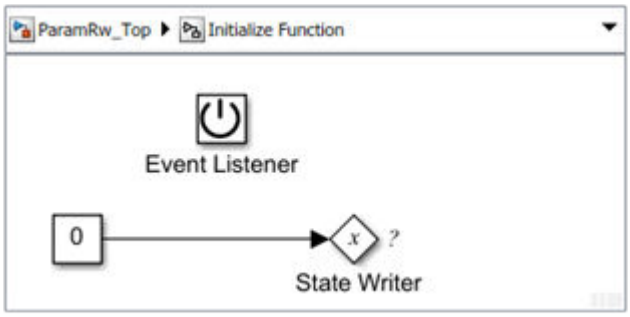


5 Rename the Model block to `Model_1`. Open the Model block parameter dialog box. In the **Model name** box, enter `ParamRw_Sub`. Select the Instance parameters tab. Set the **Value** for `Parameter_1` to `1`. This is the default value the model uses before the Parameter Writer block updates this parameter with a new value.

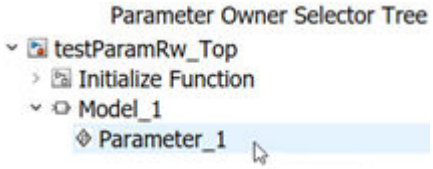


If the Model block is at a lower hierarchical level than the Parameter Writer block, select the **Argument** check box to promote the parameter to a higher level and make it visible to the Parameter Writer block.

6 Double-click the Initialize Function block. The block is preconfigured with a State Writer block.

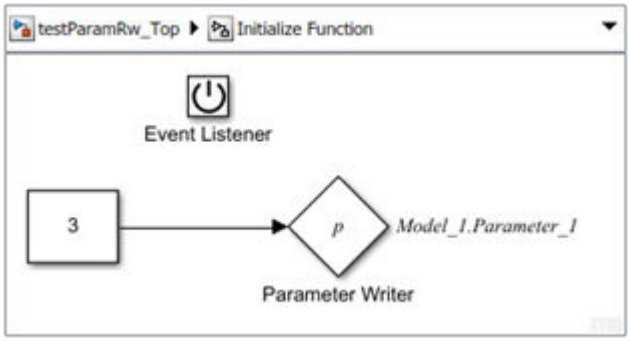


7 Replace the State Writer block with a Parameter Writer block. Open the Parameter Writer block parameter dialog box. From the **Parameter Owner Selector Tree**, select `Parameter_1`.

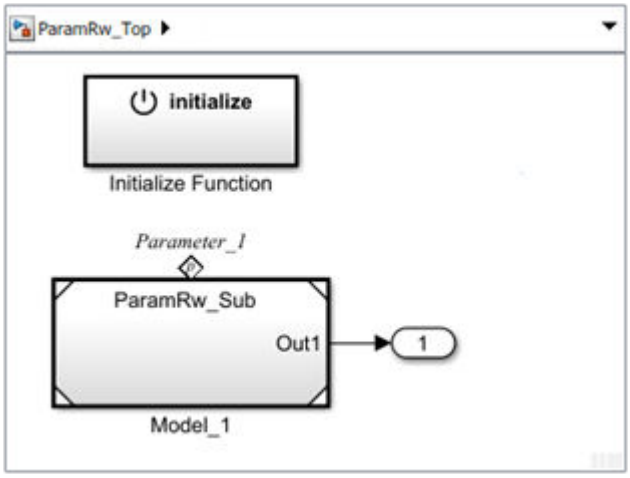


8 Open the Constant block parameter dialog box. Set **Constant value** to 3. This is the value for setting the gain with the Parameter Writer block.

9 Click **OK** to close the dialog. The Parameter Writer block displays a label indicating it writes to the model instance parameter `Parameter_1` for `Model_1`.



10 Click the `ParamRw_Top` tab. The Model block displays a badge  indicating a value is written to Parameter_1.



See Also

Blocks

Event Listener | Initialize Function | Parameter Writer | Reset Function | State Reader | State Writer | Terminate Function

Related Examples

- “Using Initialize, Reset, and Terminate Functions” on page 10-168
- “Create Test Harness to Generate Function Calls” on page 10-180
- “Initialize, Reset, and Terminate Function Limitations” on page 10-188
- Initialize and Terminate Functions video

Initialize, Reset, and Terminate Function Limitations

In this section...

Unsupported Blocks

Initialize Function, Reset Function, and Terminate Function blocks do not support:

- Model blocks
- Custom code blocks
- Stateflow charts
- Resettable Subsystem blocks
- Blocks with state, for example, Unit Delay blocks
- Blocks with absolute time, for example, Clock blocks
- MATLAB System blocks
- MATLAB Function blocks which use persistent or global data. However, MATLAB Function blocks without persistent or global data are supported.

Initialize Function, Reset Function, and Terminate Function blocks cannot call Simulink Function blocks with:

- Input or output ports
- An Initialize Function, Reset Function, or Terminate Function block
- Unsupported blocks

Unsupported Features

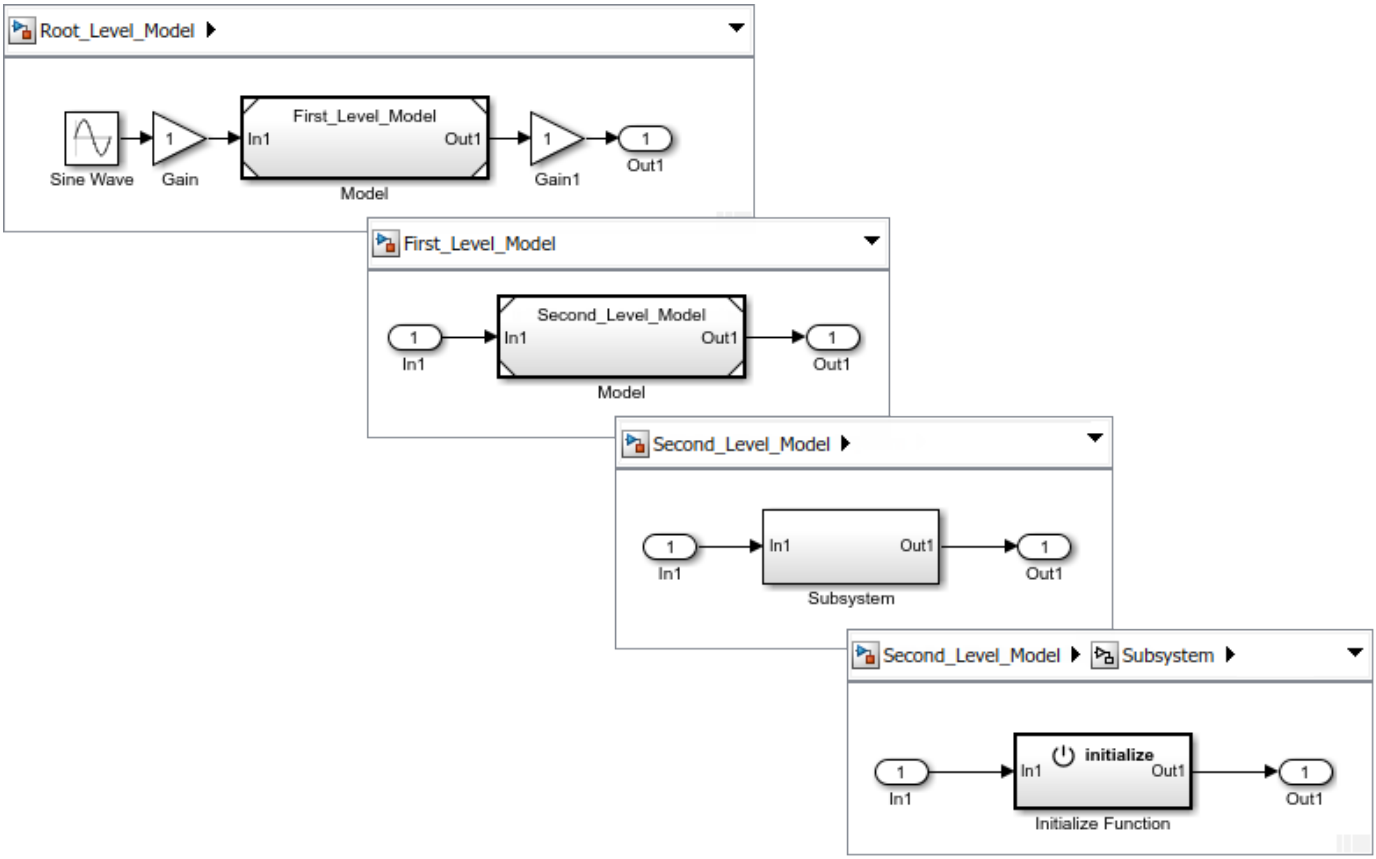
Initialize Function, Reset Function, and Terminate Function blocks do not support:

- Using variable-size signals

Component I/O Blocks

The input and output ports of a model component containing Initialize Function, Reset Function, or Terminate Function blocks must connect to root Inport and Outport blocks without intervening blocks.

In this example, an Initialize Function block is placed in a Subsystem block. The model containing the Subsystem block is referenced from a model that is referenced from the root level model. Only the top-level model with the **Show model initialize port** parameter selected can have blocks between the input and output ports.



See Also

Blocks

Event Listener | Initialize Function | Parameter Writer | Reset Function | State Reader | State Writer | Terminate Function

Related Examples

- “Using Initialize, Reset, and Terminate Functions” on page 10-168
- “Create Test Harness to Generate Function Calls” on page 10-180

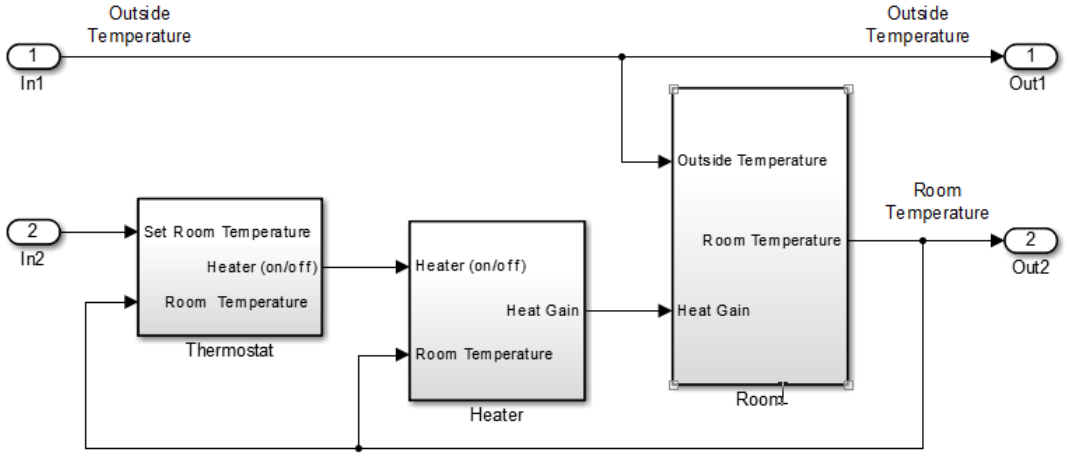
Model A House Heating System

In this section...

This tutorial shows how to model and simulate a dynamic system using Simulink software. The model is for a heating system that includes a heater (plant model), controlled by a thermostat (controller model), to heat a room (environment model) to a set temperature. While this is simple model, the processes for creating model structure and algorithm design are the same processes you will use for more complex models.

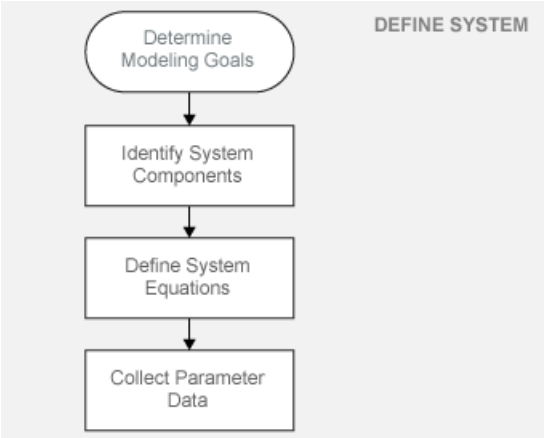
To review a completed model, in the MATLAB Command Window, enter

```
open_system(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_modeling'))
```



Define a House Heating System

Modeling begins with completion of tasks that are outside of the Simulink software environment. Define model requirements and derive mathematical equations. Collect data for model parameters and output signal data measurements to validate simulation results.



Determine Modeling Goals

Before designing a model, consider your goals and requirements. The goals for modeling the house heating system are:

- Observe how the changing outdoor temperature affects the indoor temperature.
- Examine the effect of changing parameters on the indoor temperature.

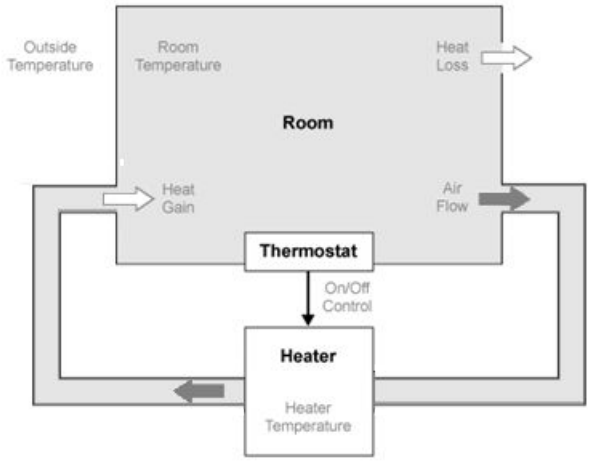
Identify System Components

Once you understand your modeling requirements, you can begin to identify the components of the system.

The house heating system in this tutorial defines a heating system and its relationship to a room. It includes:

- Thermal characteristics of a house
- Thermal characteristics of a heater
- A thermostat to control the heater
- Outdoor environment
- Indoor environment

The thermostat monitors the room temperature regularly and turns the heater on or off, depending on the difference between the set temperature and the room temperature.



The model for this system includes three components: heater, thermostat, and room.

Define System Equations

Three time-dependent variables define the heat exchange in the room:

- Temperature of the room (T_{room})
- Heat gain: Thermal energy transferred from the heater (Q_{gain}) to the room
- Heat loss: Thermal energy transferred from the room (Q_{loss}) to the outdoor environment

A differential equation defines the relationship between these variables, but since heat transfer is defined in terms of changing temperature, only room temperature is a state variable.

Rate of Heat Gain Equation

The temperature of the air in the heater is constant at T_{heater} and the room temperature is T_{room}. Thermal energy gain to the room is by convection of heated air from the heater, with a heat capacity of c_{air}. Heat gain for a mass of air in the heater, $m_{heaterair}$, is proportional to the temperature difference between the heater and the room:

$$Q_{gain} = m_{heaterair}c_{air}(T_{heater} - T_{room}).$$

The rate of thermal energy gain from the heater is

$$\frac{dQ_{gain}}{dt} = \frac{dm_{heaterair}}{dt}c_{air}(T_{heater} - T_{room}).$$

A fan takes room air, and passes it through the heater and back to the room. A constant amount of air, $M_{heaterair}$, flows through the heater per unit time, and replacing $dm_{heaterair}/dt$with that constant simplifies the equation to

$$\frac{dQ_{gain}}{dt} = M_{heaterair}c_{air}(T_{heater} - T_{room}).$$

Rate of Heat Loss Equation

Thermal energy loss from the room is by conduction through the walls and windows, and is proportional to the temperature difference between the room temperature and the outside temperature:

$$Q_{loss} = \frac{kA(T_{room} - T_{outside})t}{D}.$$

The rate of thermal energy loss is

$$\frac{dQ_{loss}}{dt} = \frac{kA(T_{room} - T_{outside})}{D}.$$

Replacing kA/D with $1/R$ where R is the thermal resistance simplifies the equation to

$$\frac{dQ_{loss}}{dt} = \frac{(T_{room} - T_{outside})}{R}.$$

Changing Room Temperature Equation

Define the rate of temperature change in the room by subtracting the rate of heat loss from the rate of heat gain:

$$\frac{dT_{room}}{dt} = \frac{1}{m_{roomair}c_{air}}\left(\frac{dQ_{gain}}{dt} - \frac{dQ_{loss}}{dt}\right).$$

Collect Parameter Data

Most of the parameter values needed for the house heating model are published in standard property tables. The flow rate for the heater is from a manufacturer data sheet.

List the variables and coefficients from your equations and check for dimensional consistency between the units. Since the unit of time for the model is hours, convert published values for the thermal property of materials from units of seconds to hours.

Equation Variables and Constants

You can use the constant names and values in this table when building the model.

Equation Variable or Coefficient	Description	Units
A	Area of wall or window surface`A_wall = 914`, `A_window = 6`	square meter
D	Depth of wall or window `D_wall = 0.2`, `D_window 0.01`	meter
Q	Thermal energy transferred	joule
dQ/dt	Rate of thermal energy transferred	joule/hour

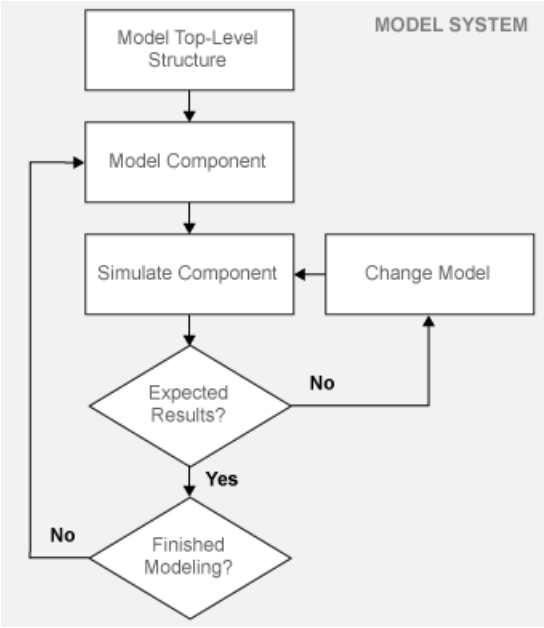
Equation Variable or Coefficient	Description	Units
k	Thermal conductivity; property of a material to conduct heat transfer `k_fiberglass = 136.8`, `k_glass = 2808`	joule/meter· hour· degree
r	Thermal resistivity; property of a material to resist heat transfer `r = 1/k`	meter· hour· degree/joule

R	Thermal resistance $R = D/kA = (T_1 - T_2)Q$
	
	`R_wall = 1.599e-6`, `R_window = 5.935e-7`
	
	`R_equivalent = (R_wall * R_window)/(R_wall + R_window) = 4.329e-7`
		hour· degree/joule

m	Mass of air in the room or heater `m_room_air = 1470` The mass of the heater `m_heater_air` is not needed for this model.	kilogram
dm/dt	Rate of air mass passing through the heater	kilogram/hour
M	Constant rate of air mass passing through the heater `M_heater_air = 3600`	kilogram/hour
c	Specific heat capacity `c_air = 1005.4`	joule/kilogram· degree
T_{heater}	Constant air temperature from heater `T_heater = 50`	degree Celsius
T_{room}	Initial air temperature of room `T_roomIC = 20`	degree Celsius

Model House Heating System

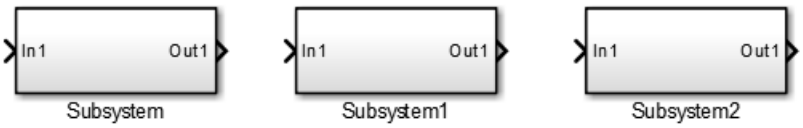
Model the top-level structure with components that including interfaces for passing data between individual components. Your model should be organized into a hierarchical structure that corresponds to the components of the system.



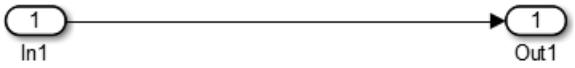
Model Top-Level Structure

At the top level of the house heating model, use Subsystem blocks to organize your model and create the structure. The model includes the subsystems Thermostat, Heater, and Room.

1. Open a new Simulink model: “Open New Model”.
2. Open the Library Browser: “Open Simulink Library Browser”
3. Add Subsystem blocks. Drag three Subsystem blocks from the Ports & Subsystems library to the new model in the Simulink Editor.



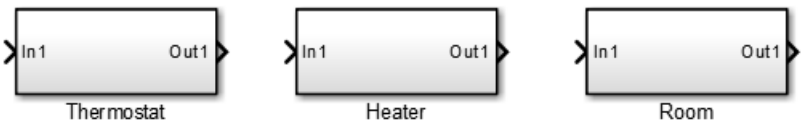
4. Open a Subsystem block. Double-click the block.



   Each new Subsystem block contains one Inport (In1) and one Outport (Out1) block. These blocks define the signal interface with the next higher level in a model hierarchy.

   Each Inport block creates an input port on the Subsystem block, and each Outport block creates an output port. Add more blocks for additional input and output signals.

5. On the Simulink Toolstrip, click the **Navigate Up To Parent** button to return to the top level. Rename the Subsystem blocks as shown. Double-click a block name and type the new name.



For each component, model the equations, define parameters, prepare the subsystem for simulation, and simulate to verify its behavior.

Model Heater Component

Let's start by modeling the heater system component. The heater model:

- Takes the current temperature from the room and a control signal from the thermostat as inputs
- Calculates the heat gain from the heater
- Outputs the heat gain when the control signal is on

To model the heater subsystem, model the rate of heat gain equation with Simulink blocks:

$$\frac{dQ_{gain}}{dt} = M_{heaterair} c_{air} (T_{heater} - T_{room}) .$$

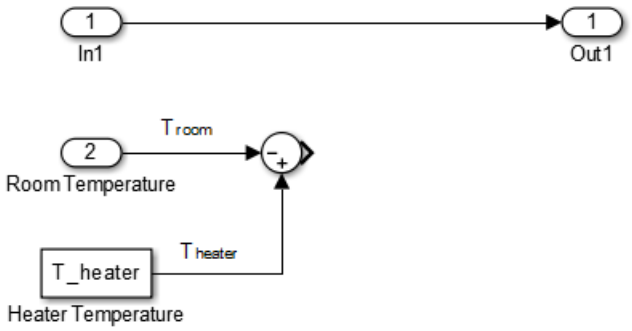
Subtract Room Air Temperature from Heater Air Temperature

The temperature difference is the current room temperature subtracted from the constant temperature of the heater (`T_heater`).

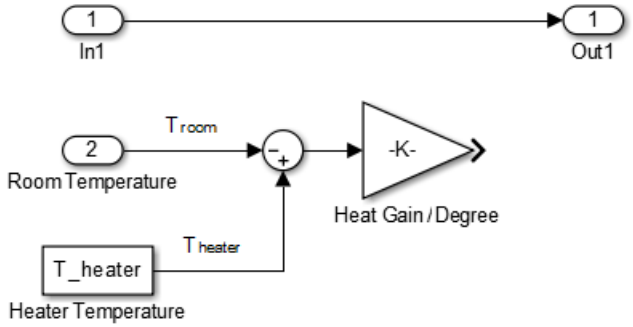
1. Open the Heater subsystem.
2. Click the model and type Sum to display a list of blocks with Sum in the name. Click the Sum block on the list. When prompted for a list of signs, type `|-+` to place - and + input ports on the block, and press **Enter**.

   The vertical bar (`|`) changes the position of input ports by inserting spaces between the ports. A vertical bar at the beginning of the sign list, places a space at the top of the block and shifts the ports counter clockwise.
3. Add a Constant block to model the constant air temperature from the heater. Set the block **Constant value** parameter to `T_heater`. You will define the value of `T_heater` in the Model Workspace.

   If the block displays `-C-`, resize the block to display the variable name.
4. Add a second Inport block to take the room temperature signal from another part of your model.
5. Add a Gain block to the Heater subsystem. Set the **Gain** parameter to `M_heater_air*c_air`. You will define the values of these variables in the Model Workspace.
6. Connect the output of the Sum block to the input of the Gain block.
7. Add labels to the signal lines to help trace model components to the equations and model requirements. Double-click above a signal line and enter a label.



8 Rename the blocks and connect them as shown in the figure.



Model a Heater Switch

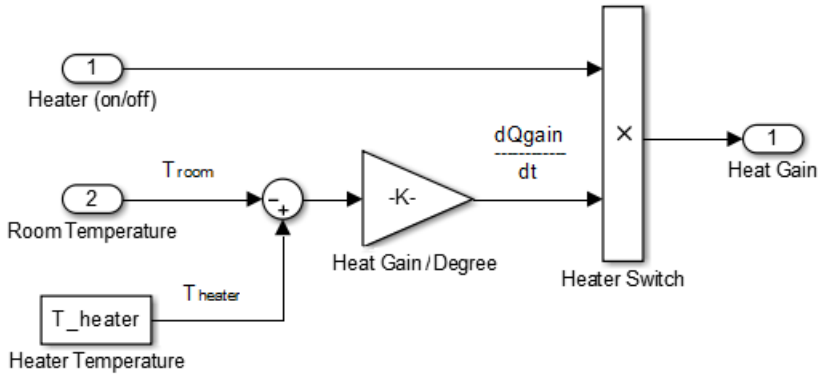
The thermostat sends an on/off signal equal to 1 (on) or 0 (off) to the heater. Because the input signal is binary, you can use a Product block to model a switch.

1 Remove the connection between the In1 and Out1 blocks. Select the line and press **Delete**.

2 Add a Product block. Resize the block vertically to align the block in your diagram. Connect the In1 block to the first block input and the block output to the Out1 block. Rename the blocks as shown.

3 Connect the output from the Gain block to the second input. Move all the connected blocks together. Draw a selection box around the blocks you want to move, and then drag them to the new location.

4 Rename blocks and add labels to signals as shown in the figure.

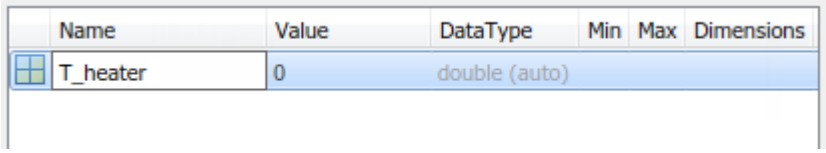


The Inport and Outport blocks create ports that connect this subsystem to other subsystems in your model.

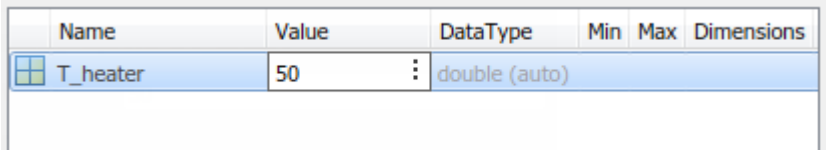
Define Heater Model Parameters

You can define parameters in the MATLAB Workspace and then enter their names in the block parameter dialog boxes. However, a more robust method is to use the Simulink Model Workspace because variable values are saved with the model.

1. In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Model Workspace**.
2. In Model Explorer, select **Add** > **MATLAB Variable**. In the middle pane, click the new variable `Var` and enter the variable name for a block parameter. For this example, enter `T_heater`.



3. Click the `0` value and enter the value for this variable. For this example, enter `50` degrees.



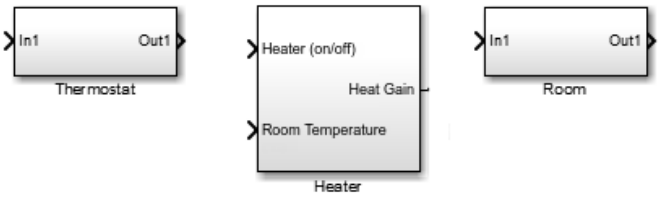
4. Using the same approach, add the variable `M_heater_air` with a value of `3600` kilogram/hour and `c_air` with a value of `1005.4` joule/kilogram· degree.

Prepare Heater Model for Simulation

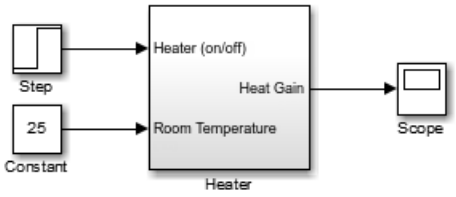
Set up the heater model for simulation. Think about the expected behavior and how you can test that behavior with a simulation. When the thermostat output is 1 (on), and assuming constant room temperature of 25, the expected output from the gain is (50 – 25) x 3600 × 1005.3 = 9.05×10^7. Verify this output by running the model with these inputs:

- Heater on/off signal that changes from `0` to `1` after the 4^{th} hour
- Room temperature constant at `25`

1. From the Heater subsystem, click the **Navigate Up To Parent** button to navigate to the top level of your model. You can resize the Heater block as shown in the figure.



   Notice the Heater block has a second input port and that each port corresponds to an Inport block or Outport block in the subsystem.
2. Add a Constant block to represent the room temperature, and set the value to 25 (degrees Celsius). Add a Step block for a temporary Heater (on/off) signal. Set **Step time** to 4.
3. Add a Scope block and connect it to the Heat Gain output.

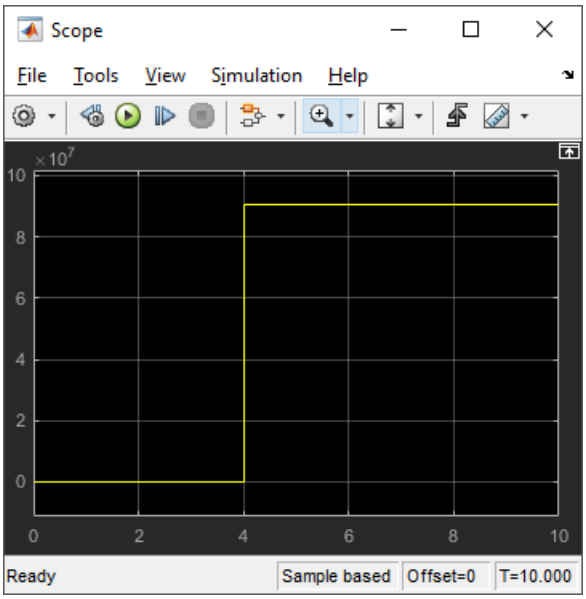


Simulate Heater Model and Evaluate Results

Use the default simulation settings to validate your model design.

1. Double-click the Scope block to open it.
2. Simulate the model. Click the **Run** button .

   As the simulation runs, the Scope plots the results.
3. View the scope trace.



4. Determine if this result is what you expected.

   When the heater on/off signal flips from 0 to 1 at 4 hours, the heater outputs 9.05×10^7 joule/hour. The simulation validates the expected behavior.
5. Remove Constant, Step, and Scope blocks you added for testing the Heater component.

Model Thermostat Component

You can model a thermostat without using system equations. Requirements for this component:

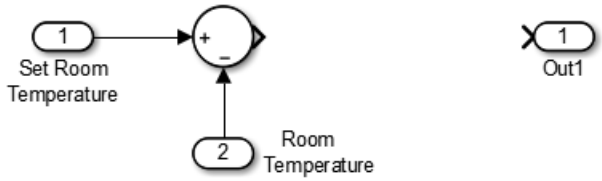
- When the room temperature is below the set temperature, heater is on and the control signal equals 1. When the room temperature is above the set temperature, the control signal equals 0.
- To avoid repeated switching around the set temperature, the thermostat allows a hysteresis of 2 degrees Celsius around the temperature set point. If the thermostat is on, the room temperature must increase 2 degrees above the set temperature before turning off. If the thermostat is off, the room temperature must drop 2 degrees below the set temperature before turning on.

This component models the operation of a thermostat, determining when the heating system is on or off. It contains only one Relay block but logically represents the thermostat in the model.

Subtract Set Room Temperature from Room Temperature

If the set room temperature is warmer than the room temperature, the thermostat model sends an "on" signal to the heater model. To determine if this is the case, begin by subtracting the room temperature from the set temperature.

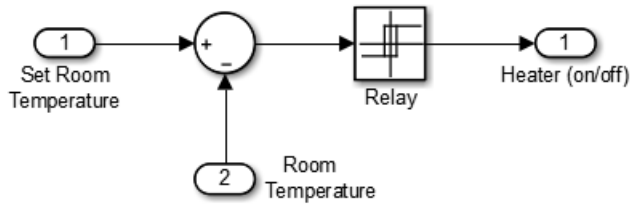
1. Open the Thermostat subsystem. Add a Sum block. Set the parameter **List of signs** to `|+-`.
2. Connect the Inport block to the + input of the Sum block. The Inport block sets the room temperature.
3. Add a second Inport block and connect it to the – input of the Sum block. This second Inport block is the current room temperature from the room subsystem. Move the output port to the top of the block. Right-click the block and select **Rotate & Flip > Counterclockwise**. If you want, you can reshape the block as shown in the figure by dragging the handles.
4. Rename the blocks as shown.



Model Thermostat Signal

Model the signal from the thermostat with a hysteresis value of 2 degrees Celsius.

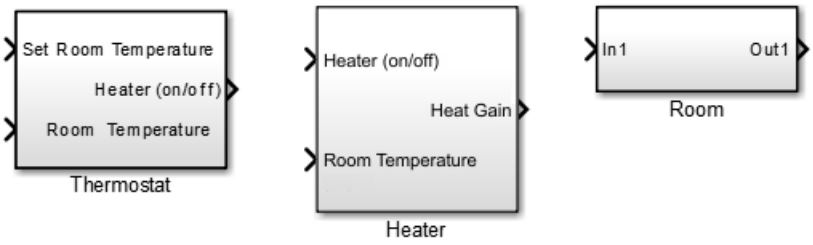
1. In the Thermostat subsystem, add a Relay block. Set the **Switch on point** parameter to 2, and the **Switch off point** parameter to `-2`.
2. Connect and rename the blocks as shown in the figure.



Prepare Thermostat Model for Simulation

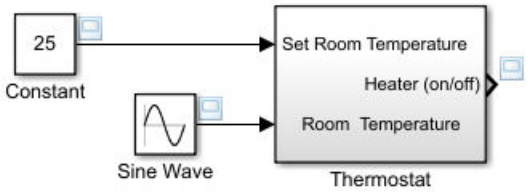
Prepare the Thermostat subsystem for simulation. Think about the expected behavior of the thermostat and how you can test that behavior with a simulation. When the room temperature rises above the thermostat setting by 2 degrees, the thermostat output is 0. When the room temperature moves below the thermostat setting by 2 degrees, the thermostat output is 1.

1. From the Thermostat subsystem, click the **Navigate Up To Parent** button to navigate to the top level of your model. Resize the Thermostat block as shown in the figure.



Notice the Thermostat subsystem now has a second input port. Each input port corresponds to an Inport block in the subsystem.

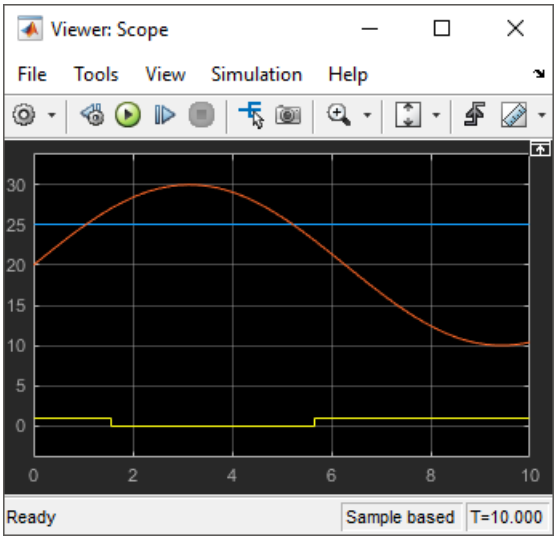
2. Add a Constant block for the set temperature. Set the **Constant** parameter to `25` (degrees Celsius).
3. Add a Sine Wave block to represent the changing room temperature. Set the **Amplitude** parameter to `10`, the **Bias** to `20`, and the **Frequency** to `0.5`. These parameters give a variation above and below the temperature set point of 25.
4. Create and connect Scope Viewer at the Heater port. See “Add Signal Viewer”.
5. Connect the two input signals to the Scope Viewer.



Simulate Thermostat Model and Evaluate Results

Use the default simulation settings to validate your model design.

1. Simulate the model. As the simulation runs, the Scope Viewer plots the results.
2. Open the Scope to view the scope trace.



3 Determine if this result is what you expected.

Initially the room temperature is below the set temperature and the relay is on. When the room temperature reaches the set temperature, the relay continues to be on until the room temperature increases by 2 more degrees. Simulation validates the expected behavior.

Model Room Component

Inputs to the room component are heat flow from the heater component and the external air temperature. The room component uses these inputs to compute heat loss through the walls, heat loss through the windows, and the current room temperature.

To design the room subsystem, use the Rate of Heat Loss equation and the Changing Room Temperature Equation.

Model Changing Room Temperature

The rate of temperature change in the room (dTroom/dt) is defined by the equation

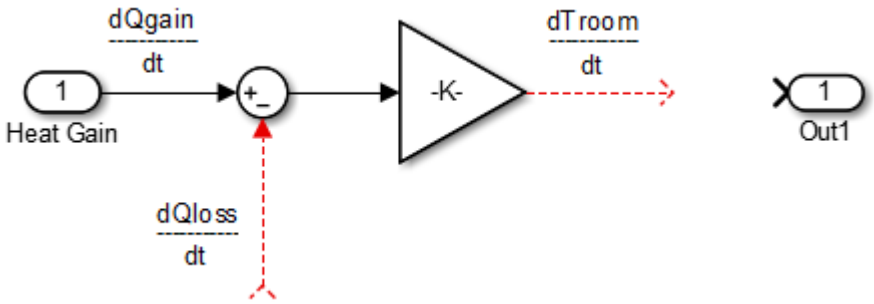
$$\frac{dT_{room}}{dt} = \frac{1}{m_{roomair}c_{air}}\left(\frac{dQ_{gain}}{dt} - \frac{dQ_{loss}}{dt}\right).$$

The term `dQgain/dt` is a signal from the Heater subsystem.

1 Open the Room subsystem block. In the Room subsystem, add a Sum block. Set the **List of signs** parameter to `|+-`.

2 Connect In1 to the + input. The input is the heat gain (`dQgain/dt`) from the heater component. The – input connects to the heat loss (`dQloss/dt`) from the room.

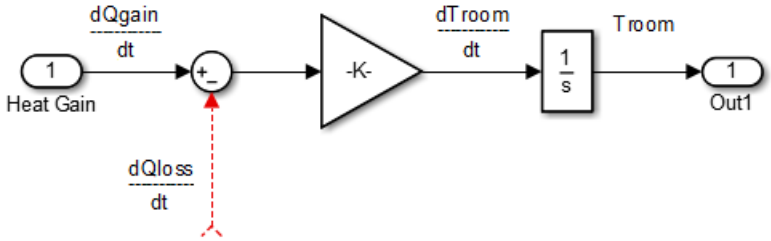
3 Add a Gain block. Set the **Gain** parameter to `1/(m_room_air*c_air)`. Connect the output of the Sum block to the input of the Gain block. Label signals as shown in the figure. Dotted signal lines are signals you will connect later.



Model Room Temperature

The output of the Gain block is the change in room temperature (`dTroom/dt`). To get the current room temperature (`Troom`), integrate the signal.

1. Add an Integrator block. Set the **Initial condition** parameter to `Troom_IC`.
2. Connect the output of the Integrator block to Out1 as shown.

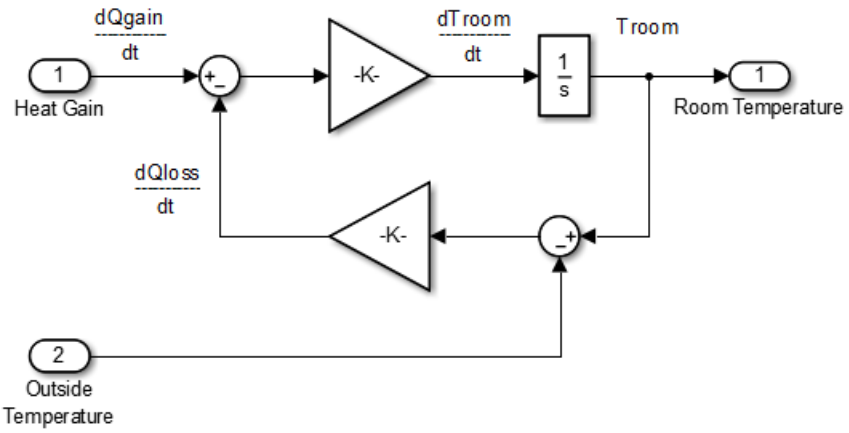


Model Heat Loss Through Walls and Windows

This equation is the rate of thermal energy loss through the walls and windows:

$$\frac{dQ_{loss}}{dt} = \frac{(T_{room} - T_{outside})}{R}.$$

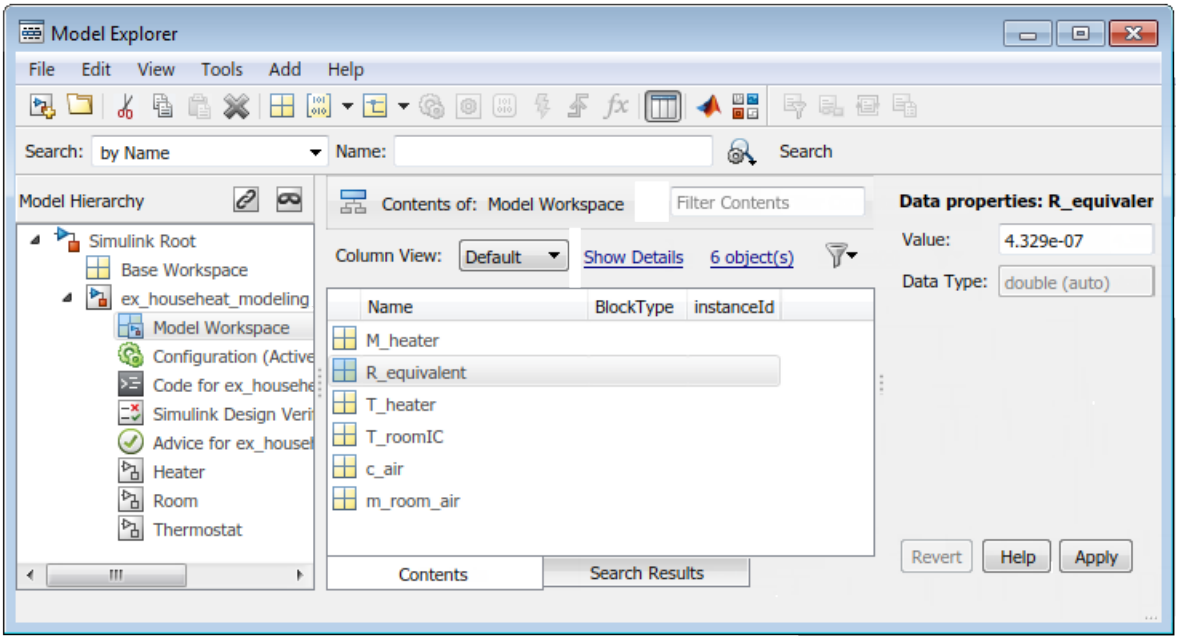
1. In the Room subsystem, add a Sum block. Set the **List of signs** parameter to `|+-`. Right-click the block and select **Rotate & Flip** > **Flip Block**.
2. Connect `Troom` to the Sum block. Click the signal line for `Troom` and the `+` input on the Sum block.
3. Add another Inport block and connect it to the – input of the Sum block. Rename it to Outside Temperature.
4. Add another Gain block. Set the **Gain** parameter to `1/R_equivalent`. Right-click the block and select **Rotate & Flip** > **Flip Block**.
5. Connect the blocks as shown in the figure.



Define Room Model Parameters

You can define parameters in the MATLAB Workspace and then enter their names in the block parameter dialog boxes. However, a more robust method is to use the Simulink Model Workspace, which saves parameter values with the model.

1. In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Model Workspace**.
2. In the Model Explorer, select **Add** > **MATLAB Variable**.
3. In the middle pane, click the new variable `Var` and enter the name `m_room_air`. In the right pane, enter the value `1470` (kilograms).
4. Add the variables `T_roomIC` = `20` (degrees Celsius) and `R_equivalent` = `4.329e-7` (hour·degree/joule).

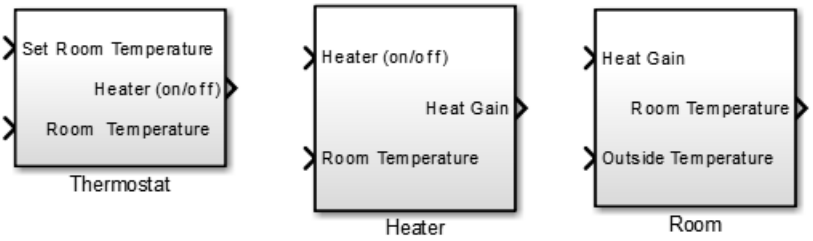


Prepare Room Model for Simulation

Prepare the Room subsystem for simulation. Think about the expected behavior and how you can test that behavior with a simulation. When the heater is off (Heat Gain = 0) and the initial temperature of

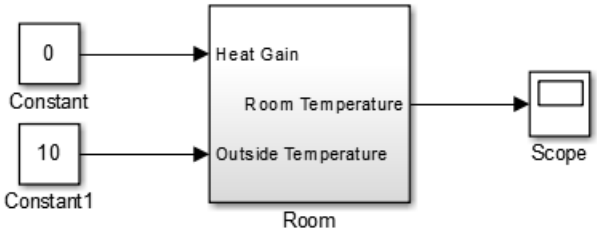
the room (20) is above the outside temperature (10), heat loss should continue until the room temperature is equal to the outside temperature.

1. From the Room subsystem, click the **Navigate Up To Parent** button to navigate to the top level of your model. Resize the Room block as shown in the figure.



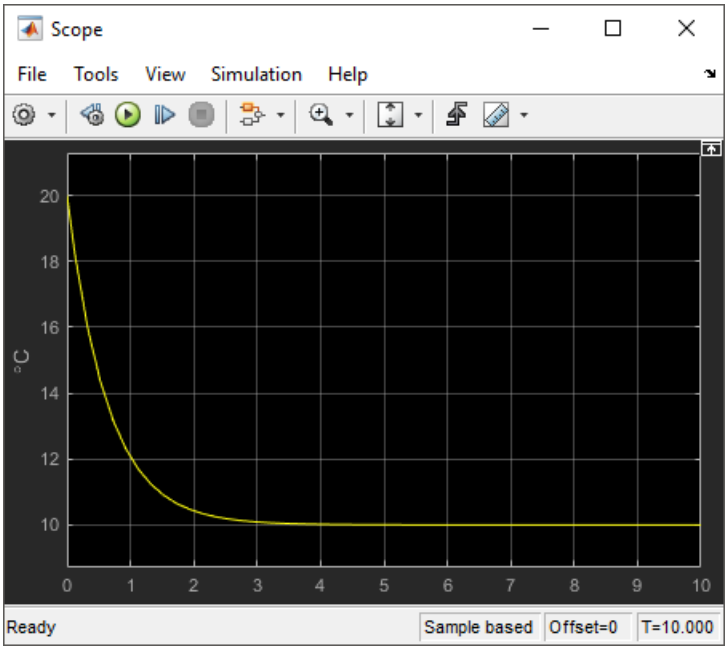
   The Room block now has a second input port. Each input port corresponds to an Inport block in the subsystem.

2. Add a Constant block and connect it to the Heat Gain input. Set the **Constant value** parameter to `0` (degrees Celsius) to mean that the heater is turned off.
3. Add another Constant block and connect it to the Outside Temperature input. Set the **Constant value** parameter to `10` (degrees Celsius).
4. Add and connect a Scope block to view the changing room temperature.



Simulate Room Model and Evaluate Results

1. In the toolstrip, set the **Stop Time** to `20`.
2. Simulate the model.
3. Open the Scope and click the **Autoscale** button to view the scope trace.



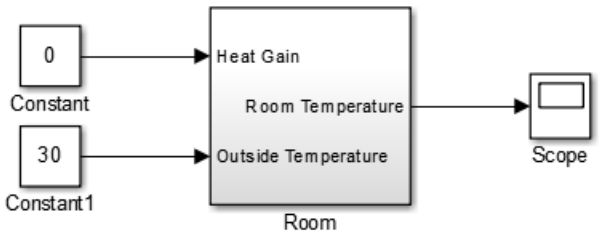
4 Determine if this result is what you expected.

The room temperature starts at the initial room temperature set in the Integrator block. Because the heat gain is 0, the signal decays to the outside temperature (10). The simulation validates the expected behavior.

Prepare Room Model for Second Simulation

Set the constant outside temperature to a value above the initial room temperature (20).

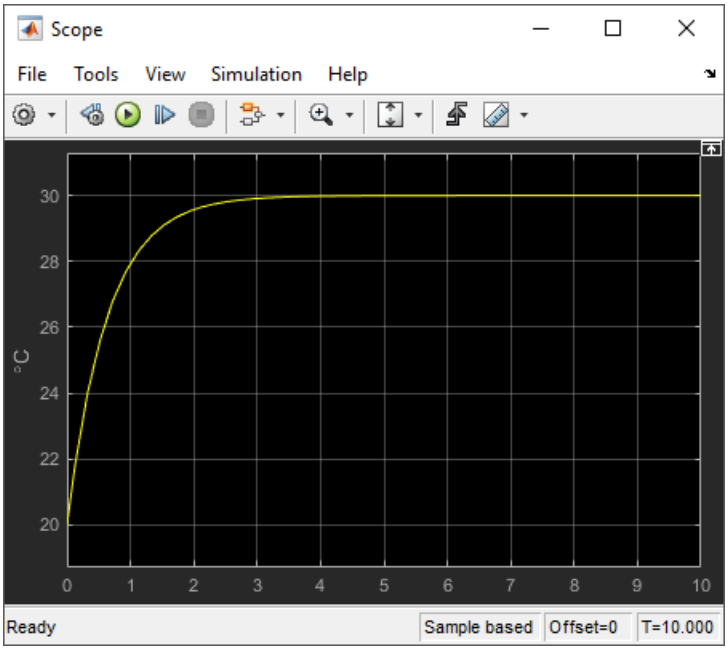
In the Constant block that is connected to the Outside Temperature input, set **Constant value** to `30` (degrees Celsius).



Simulate Model and Evaluate Results

1 Simulate the model.

2 Open the Scope and click the **Autoscale** button to view the scope trace.



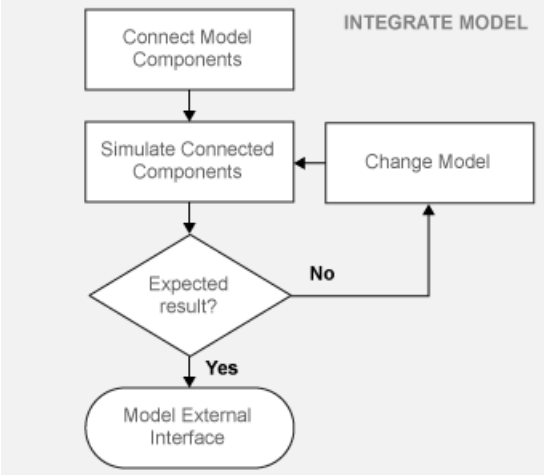
3 Determine if this result is what you expected.

Room temperature starts at the initially set temperature of 20, but with the heater off (heat gain = 0) the room temperature rises to the outside temperature — a behavior that the model did not explicitly specify, and might be considered unexpected.

The equation that models the heat loss also models the heat gain when the outside temperature is above the inside room temperature. While the model did not explicitly specify this behavior when the heater is turned off, the result makes sense physically.

Integrate a House Heating Model

Connect model components, add realistic input, and then simulate the model behavior over time to validate the model design.

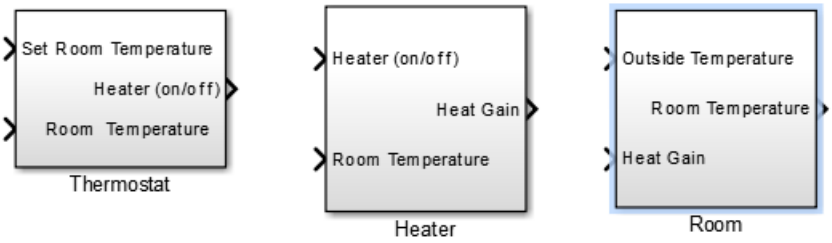


Integrate Heater and Thermostat Components

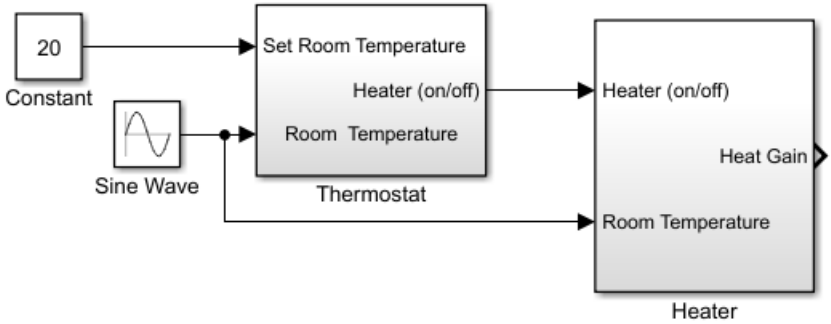
To simulate the heater and thermostat subsystems without the Room subsystem, you need a signal for the changing room temperature. Use a Constant block to set the thermostat temperature and a Sine Wave block for a realistic outside temperature signal.

Prepare Model for Simulation

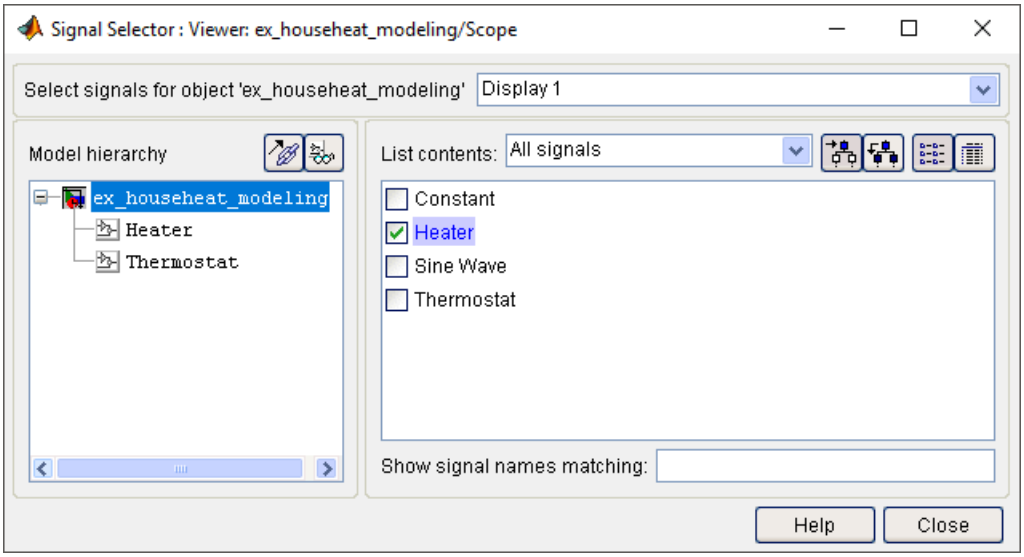
1. Open your model with the completed subsystems. Remove any blocks you added to test the separate components.
2. Open the Room subsystem. Double-click the Inport block labeled Heat Gain. In the Inport block dialog box, set **Port number** to `2`. The Heat Gain port moves to the bottom of the Room subsystem.



3. Connect the Heater (on/off) signal from the Thermostat subsystem output to the Heater subsystem input.
4. Add a Constant block to set the thermostat room temperature. Set **Constant** value to `20` (degrees Celsius).
5. Add a Sine Wave block to represent the changing room temperature. Set the parameters **Amplitude** to 10 (degrees Celsius), **Bias** to 15, and **Frequency** to `0.5`.
6. Connect the blocks as shown in the figure.



7. Add a Scope viewer and add the output signals from Heater, Constant, and Sine Wave blocks. See “Add Signal Viewer”.
8. On the Scope viewer window, in the **Configuration Properties** button click the arrow and then click **Layout** icon. Select two boxes. A second empty graph appears below the first.
9. From the toolbar, click the **Signal Selector** button. Select `Display 1`. Select the **Heater** check box.



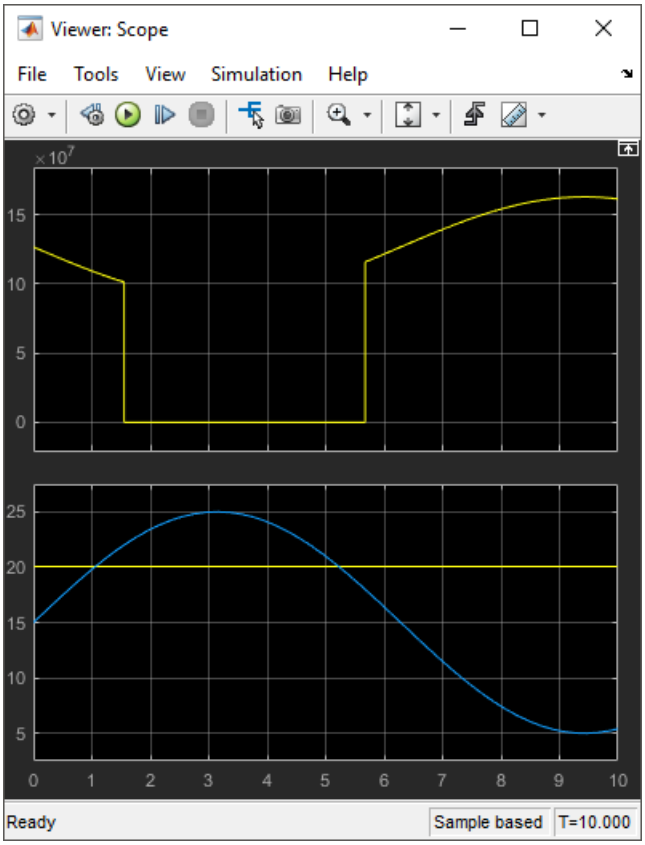
10 Select `Display 2`. Select the **Constant** and **Sine Wave**check boxes.

Simulate Model and Evaluate Results

Simulate the model using the default stop time of 10.

1 Simulate the model.

2 Open the Scope Viewer and view the simulation results. The top graph is the heater gain while the lower graph shows the changing room temperature modeled with a sine wave.



3 Determine if this result is what you expected.

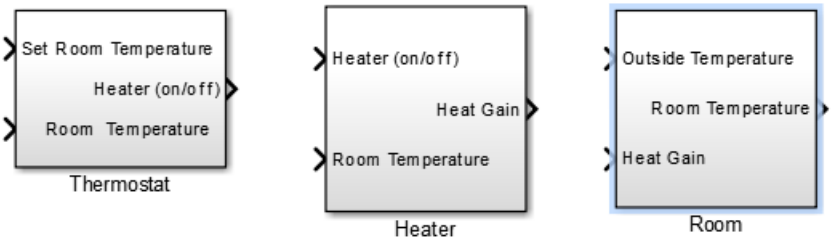
From about 0 to 1.5 hours, the heater is turned on. Heat gain is not constant but changes because heat gain is a function of the difference between the heater air temperature and the room air temperature. From 1.5 to 5.6 hours, the heater is turned off and the heat gain (top graph) is zero. The simulation confirms the expected behavior.

Integrate Room Component

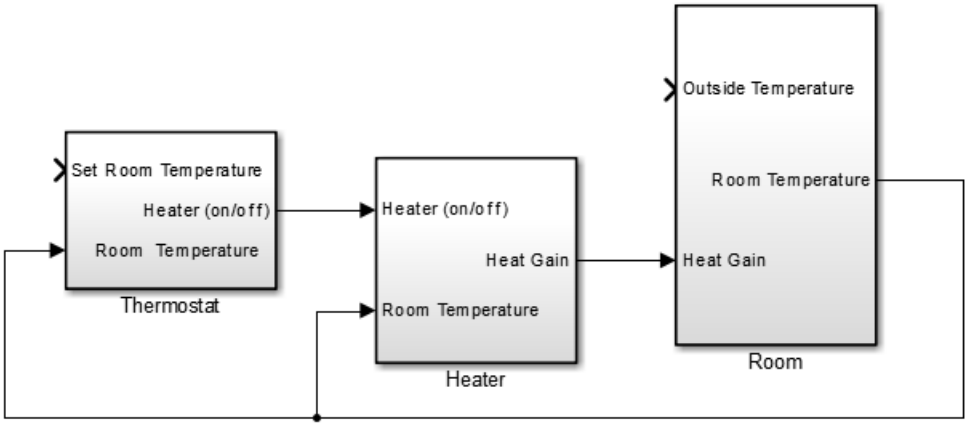
To simulate the Heater and Thermostat subsystems with the Room subsystem, you need a signal for the changing outside temperature. Simulating the model allows you to observe how the thermostat setting and outdoor temperature affect the indoor temperature.

Prepare Model for Simulation

1 Open your model with completed subsystems. Remove any blocks you added to test the separate components.

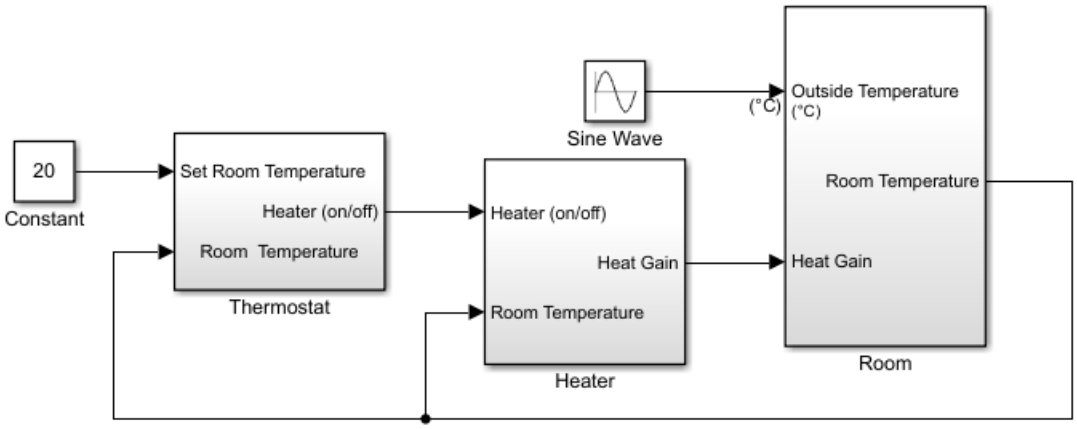


2 Connect the subsystems as shown.



3 Add a Constant block for setting the room temperature. Set **Constant value** parameter to `20` (degrees Celsius).

4 Add a Sine Wave block to represent the changing outside temperature. Set **Amplitude** to `5`, **Bias** to `12`, **Frequency** to `2*pi/24`, and **Phase** to `180`.



5 Add a Scope Viewer block to view simulation results.

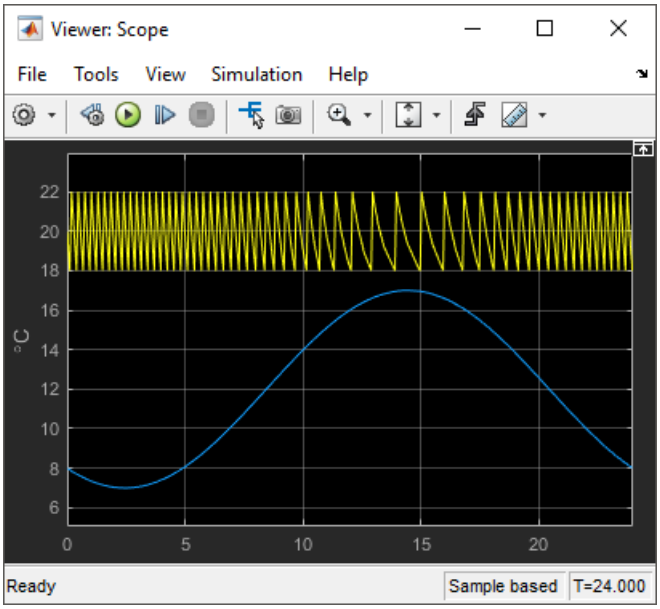
6 In the Signal Viewer, click the **Signal Selector** button. In the Signal Selector dialog box and in the left pane, select the top model hierarchy. In the right pane, select the Room and Sine Wave signals.

Simulate Model and Evaluate Results

1 Set the simulation stop time to `24` (hours) to represent a day.

2 Simulate the model.

3 Open the Scope Viewer and view results.



4 Determine if the simulation result matches your expectation.

When the outside temperature is below the set room temperature, the room temperature fluctuates 2 degrees above and below the set temperature. Since the thermostat subsystem includes a 2 degree hysteresis, this simulation result is expected.

5 You can compare your results with an example model. In the MATLAB Command Window, enter

```
open_system(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_modeling_prepared'))
```

or click `ex_househeat_modeling_prepared.slx`.

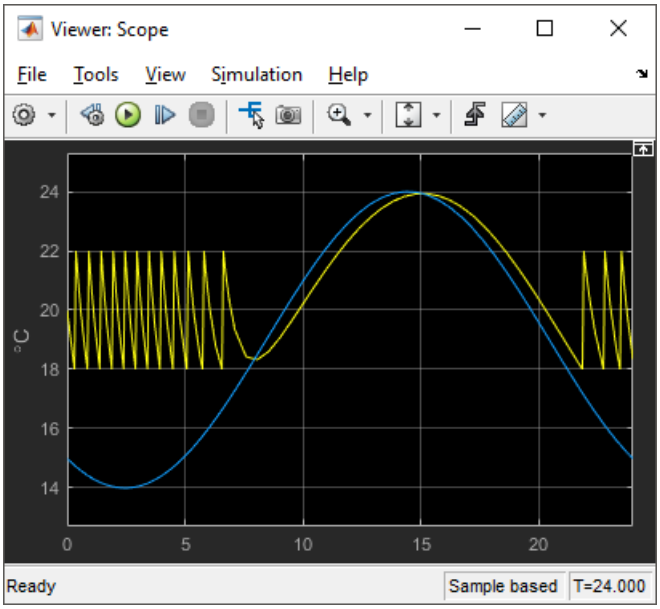
Refine Model Parameters

With Simulink models, you can interactively change model parameters and then observe changes in the behavior of your model. This approach allows you to evaluate your model quickly and validate your design.

Change the outside temperature in the Sine Wave block so that upper values are above the set temperature.

1 In the Sine Wave dialog box, set **Amplitude** to 5 and **Bias** to 19. These settings show what happens when outside temperature is higher than inside temperature.

2 Simulate the model and view the results.



3 Determine if the results match your expectations.

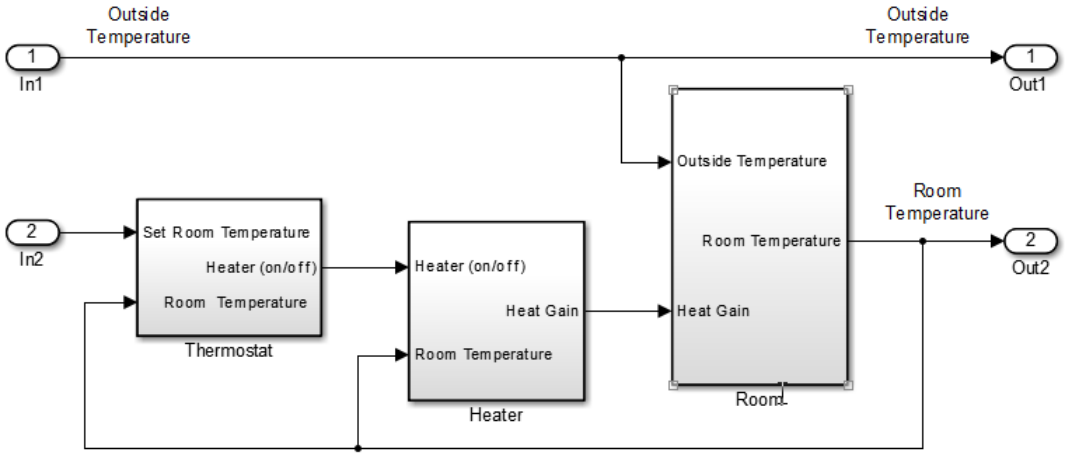
When the outside temperature is above the set temperature, the room temperature follows the outside temperature with a slight delay. In this case, heat loss works in the reverse direction - and represents the loss of heat from the outside environment into the room.

Model External Interface

Model the external interface for further testing and possible use in a larger model. In Simulink, you model the external interface using Inport and Outport blocks.

1 Add Inport blocks to read data from the outside temperature and thermostat set temperature into your model.

2 Add Outport blocks to connect the outside temperature and room temperature to a larger model or to visualize results.

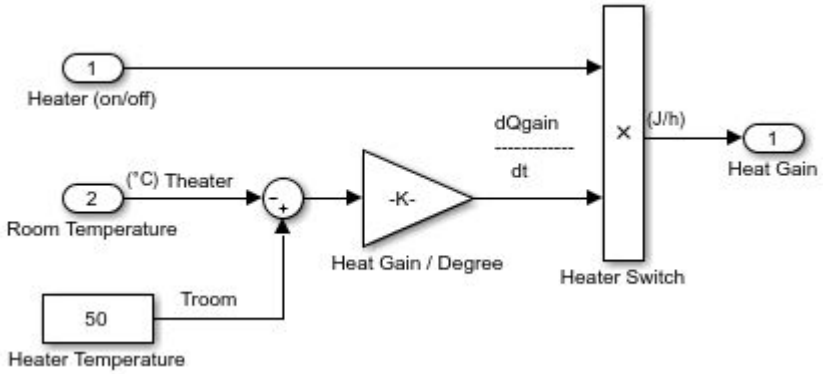


Specify Physical Units

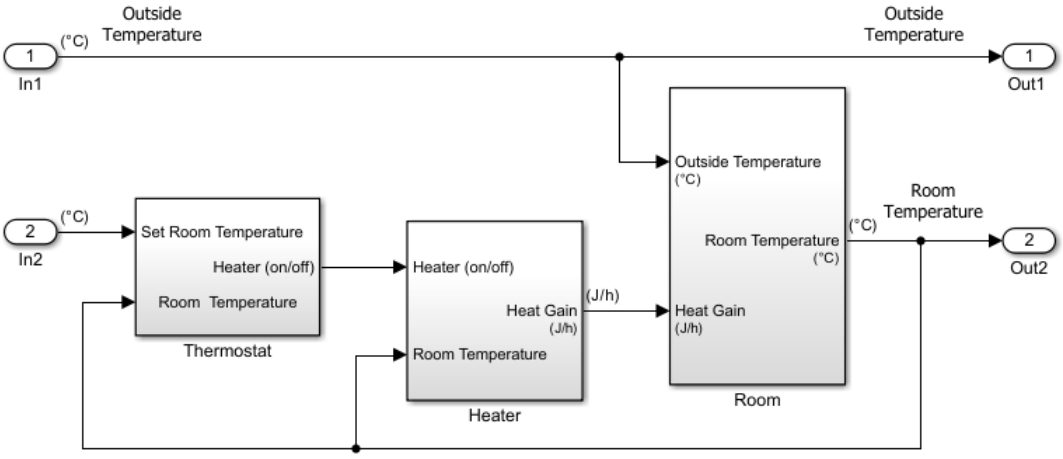
By specifying physical units for model signals, you ensure the consistency of calculations across model components. In Simulink, you specify signal units through Inport and Outport blocks.

1. Double-click the `In1` block to open the Block Parameters dialog box. Select the **Signal Attributes** tab.
2. In the **Unit** box, start typing `degree`. From the list of symbols and names, select `°C degree_Celsius`.

   For the remaining temperature Inport and Outport blocks, set the **Unit** parameter to `°C degree_Celsius`.
3. Display units on block ports. On the **Debug** tab, select **Information Overlays > Units**.
4. Double-click the Heater Subsystem block. Double-click the Heat Gain Outport block to open the Block Parameters dialog box. Select the **Signal Attributes** tab.



5. In the **Unit** box, start typing `joule/hour`. From the list of symbols and names, select `joule/h joule/hour`.
6. Update the model. Press **Ctrl+D**.



Your next step is to verify the correctness of the model by comparing simulations with real system data.

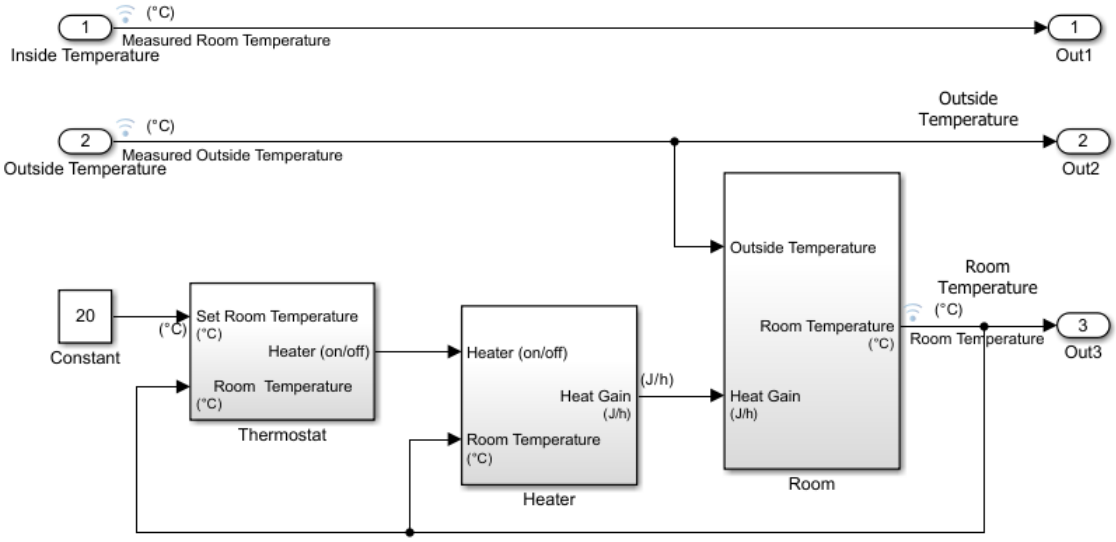
Prepare for Simulation

After initial simulations, you can use the results to improve the model to match model behavior to measured data. After you prepare the model for simulation, you can use an interface to input measured system data and set room temperature.

To load the finished example model, in the MATLAB Command Window, enter

```
copyfile(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_measured_data.mat'))
```

```
open_system(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_simulation_prepared'))
```



Verify that a simulation represents the behavior of the system you modeled. Begin by experimentally measuring physical characteristics of the system that have comparable signals in your model:

- Collect data from physical system
- Prepare model for simulation

Collect and Plot System Data

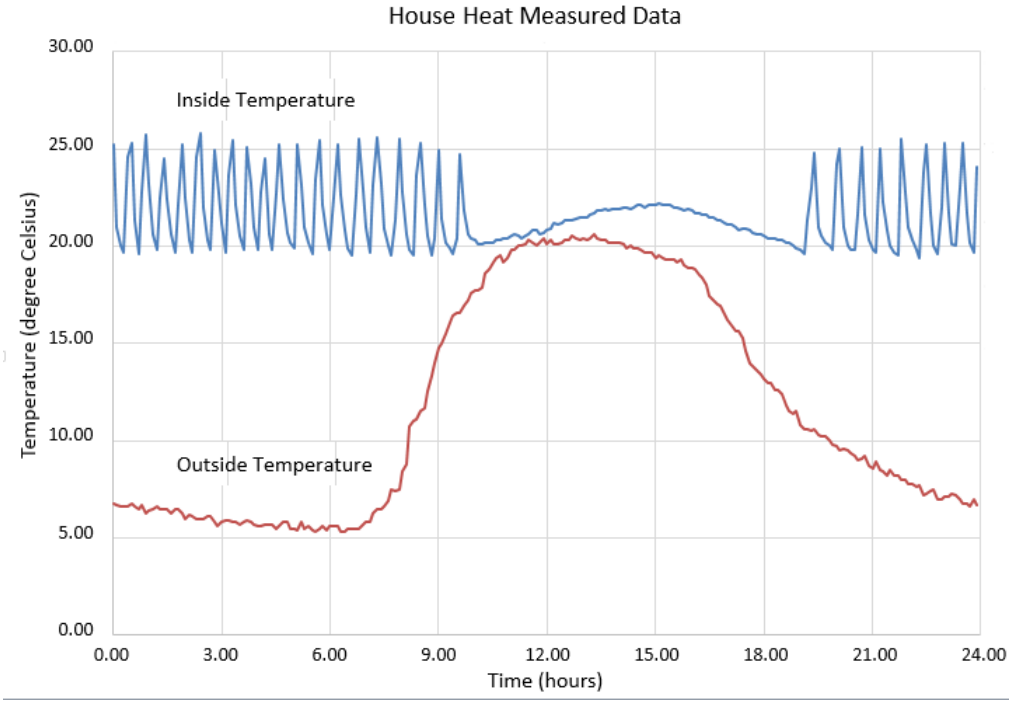
Measure the dynamic characteristics from an actual house heating system. You will use the measured data with model simulations to verify the behavior and accuracy of your model.

1. Measure the outside and inside temperatures of a house every 6 minutes for 24 hours.
2. Enter the measured data into a Microsoft Excel worksheet or open an example spreadsheet. In the MATLAB Command Window, enter

   ```
   winopen(fullfile(matlabroot,...
   'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_measured_data.xls'))
   ```

	A	B	C
1	Time (hours)	Inside Temperature	Outside Temperature
2	0.00	25.20	6.60
3	0.10	21.00	6.50
4	0.20	20.00	6.60
5	0.30	19.70	6.60
6	0.40	24.60	6.60
7	0.50	25.30	6.40
8	0.60	21.30	6.60
9	0.70	19.80	7.00
10	0.80	22.80	6.70

3 Review a plot of the measured data. The inside temperature data shows temperature spikes when the hot air heater turns on. This pattern is typical for a hot air heating system.

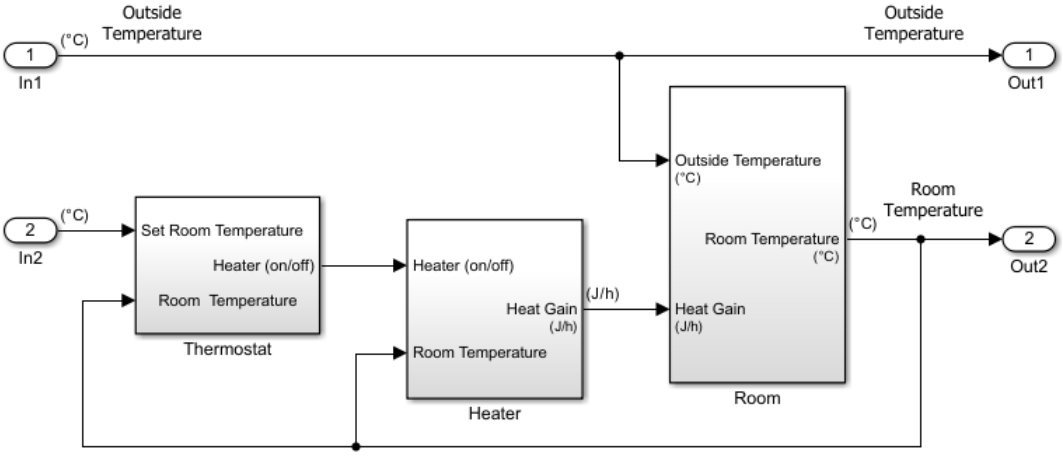


Prepare Model for Simulation

Prepare a model for simulation by adding an external interface for data input and input control signals.

1 Use the model you created in the tutorial Model House Heating System or open the example model. In the MATLAB Command Window, enter

```
open_system(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_modeling'))
```

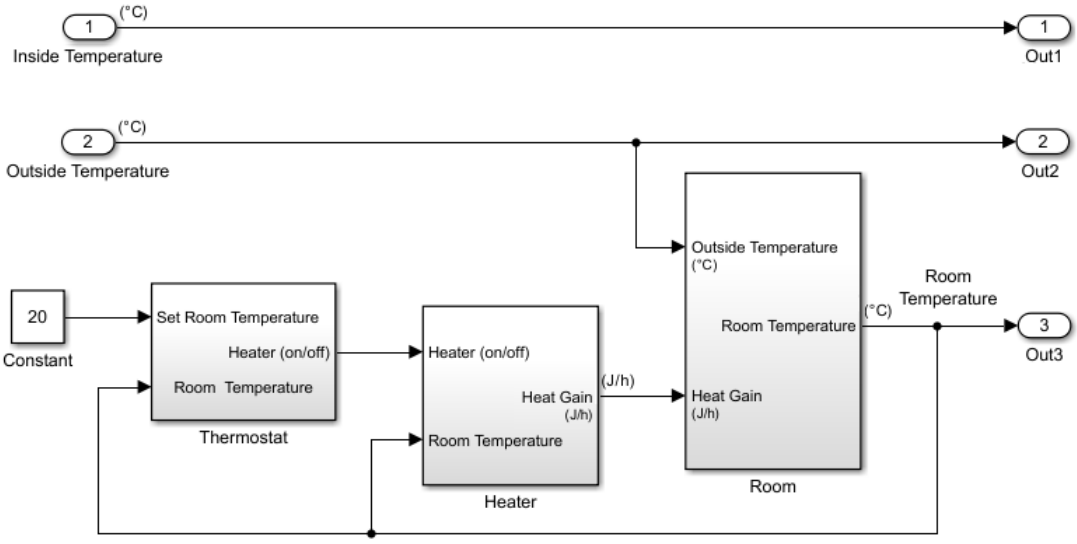


2 Replace the Inport block `In2` with a Constant block and set the **Constant** parameter to `20`. The Constant block sets the thermostat temperature.

3 Add an Inport block. Set **Port number** to `1`. This action also sets the **Port number** of the outside temperature signal to `2`.

4 Rename the first Inport block to `Inside Temperature`. Rename the second Inport block to `Outside Temperature`.

5 Add an Outport block and connect it to the first Inport block (Inside Temperature). The Outport blocks are needed for saving (logging) the signals. Set **Port number** to `1`.



Run and Evaluate Simulation

Verify the accuracy of the model and optimize parameters. Some parameters to consider for optimization are heater hysteresis, temperature offset, and the resistance of the house to heat loss. Follow these steps to verify your model:

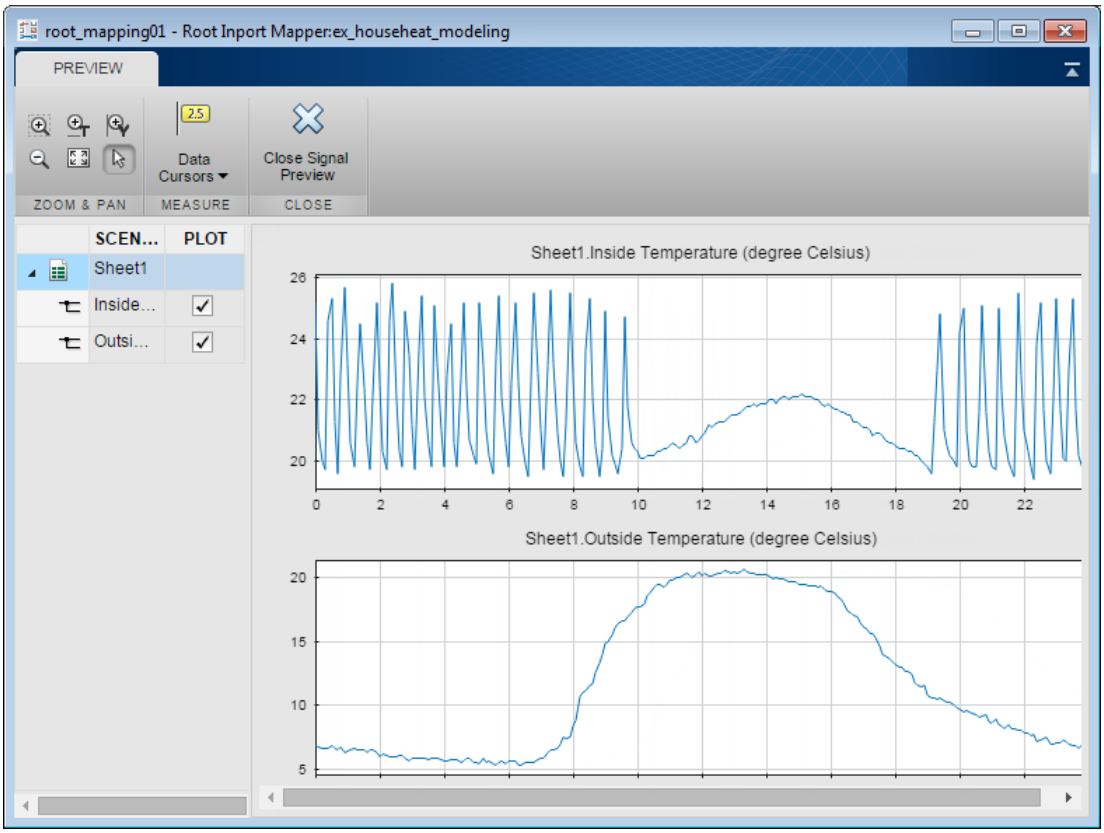
- Import data
- Run simulation

- Evaluate simulation result
- Change model parameters
- Rerun simulation

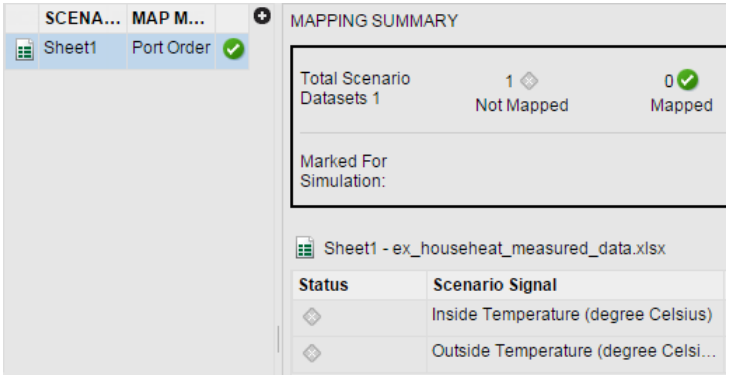
Import Data with Root Inport Mapping

You can use the Root Inport Mapper tool to bring measured signal data from an Excel spreadsheet into a Simulink model.

1. Open any Inport block. Click the **Connect Input** button to open the Root Inport Mapper.
2. On the toolstrip, click **From Spreadsheet**.
3. In the From Spreadsheet dialog box, click the browse button. Browse to and select the file *`matlabroot`*`\help\toolbox\simulink\examples\ex_househeat_measured_data.xls`. Click **Open**. Click **OK** to import the spreadsheet.
4. From the **Signals** drop-down list, select **Preview Signals**.
5. On the left side, expand the tree view of `Sheet1`. Select the Inside Temperature and Outside Temperature check boxes.

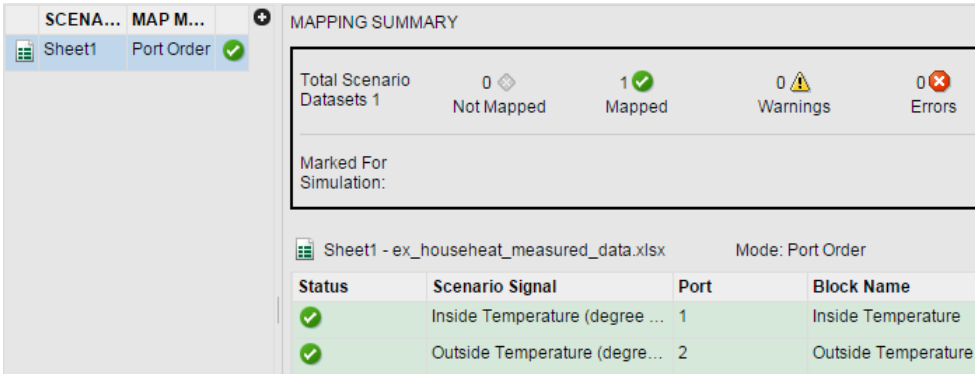


6. Click **Close Signal Preview**.
7. On the left side, select `Sheet1`. The **Scenario Signal** column shows the two signals from the Excel spreadsheet and an icon indicating the signals are unmapped.

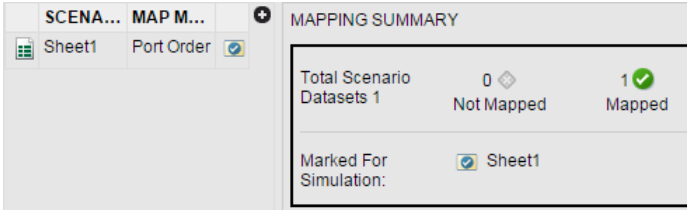


8 On the toolstrip, select the **Port Order** option. From the **Options** drop-down list, select the **Update Model** check box.

9 From the **Map to Model** dropdown list, select `Map Unconnected`. The mapping summary shows the signals from the Excel spreadsheet mapped to the Input port blocks.



10 Click **Mark for Simulation**. The mapping summary shows `Sheet1` is marked for simulation and a `Dataset` object is created in the MATLAB Workspace.



11 Save the signal data in a MAT-file. In the MATLAB Command Window, type

```
save('ex_househeat_measured_data.mat', 'Sheet1')
```

Configure Model to Load Signal Data

Signal data mapped to input ports is located in a MATLAB workspace variable. With each new MATLAB session, you have to manually reload the data or let the model preload function do it for you.

1 From the Simulink Editor, on the **Modeling** tab, select **Model Settings** > **Model Properties** to open the Model Properties dialog box.

2 Select the **Callbacks** tab.

3 In the **Model callbacks** section, select `PreLoadFcn`.
4 In the **Model pre-load function** box, enter

```
load('ex_househeat_measured_data.mat')
```

5 Click **OK**.

Configure Model to Save Simulation Results

Configure your model to save (log) signal data during a simulation. You can then view logged signals from a simulation using the Simulation Data Inspector.

1 In the model, on the **Modeling** tab, click **Model Settings**.
2 In the left pane, select **Data Import/Export**.
3 In the right pane, clear the **Time** and **Output** check boxes.
4 Select the **Signal logging** check box.
5 Select the **Record logged workspace data in Simulation Data Inspector** check box.
6 Click **OK**.

Select Signals to Save

Identify signals to display in the Simulation Data Inspector, name the signals if they are unnamed, and set the logging parameters.

1 Right-click the Inside Temperature signal line and select **Properties**.
2 In the **Signal name** box, enter `Measured Room Temperature`. Select the **Log signal data** check box. A logging badge appears above the signal line.
3 Name and select logging for these signals.

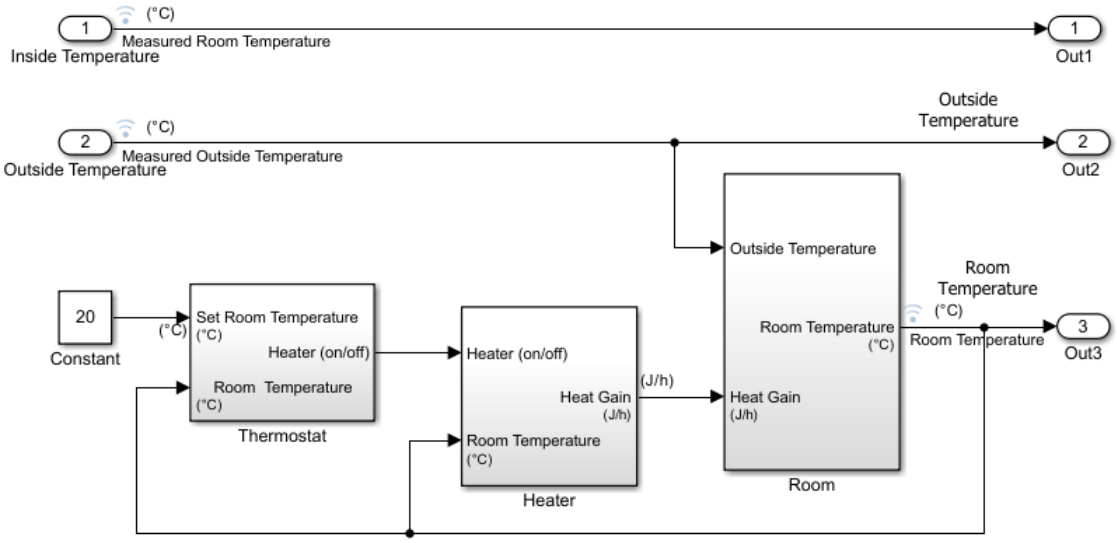
Location of signal	Signal name
Outside Temperature from output port 2.	`Measured Outside Temperature`
Room Temperature from Room subsystem output port	`Room Temperature`

Run Simulation

After importing data and enabling logging of data for the signals, you can run a simulation.

1 Use the model you prepared for simulation or open the example model. In the MATLAB Command Window, enter

```
open_system(fullfile(matlabroot,...
'help', 'toolbox', 'simulink', 'examples', 'ex_househeat_simulation_prepared'))
```



2 On the Simulink Toolstrip, set **Stop Time** to 24 (hours).

3 Click the **Run** button .

The model simulation runs from `0.0` to `24.0` hours using the outside temperature data from the root import block as input.

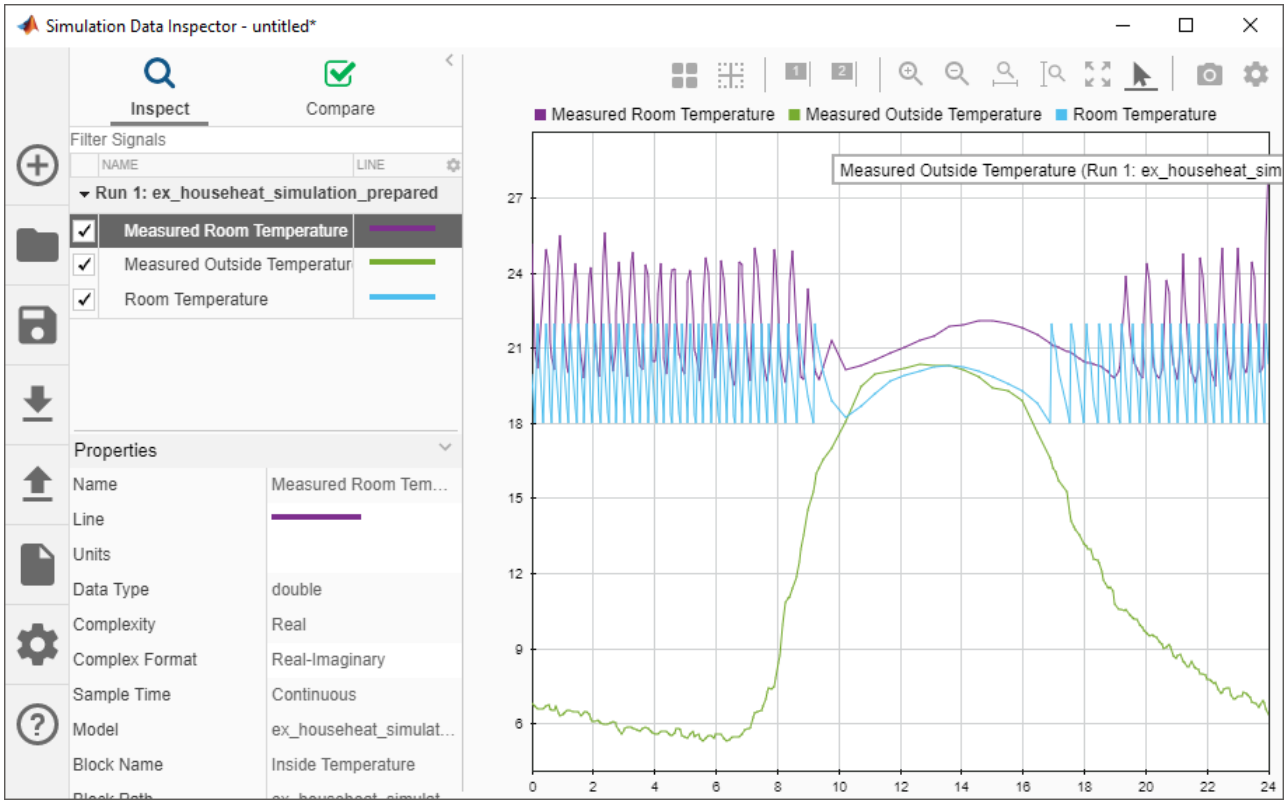
Compare Simulation Results with Measured System Data

Use the Simulation Data Inspector to compare the simulated output signals with measured data.

1 On the Simulink Toolstrip, click the **Simulation Data Inspector** button .

A separate run appears in the **Runs** pane each time you simulate the model.

2 Select all the signal check boxes. The graph show the plot of each signal you select.



The top signal is the Measured Room Temperature. The middle signal is the Measured Outside Temperature. The bottom signal is the simulated Room Temperature.

Determine Changes to Model

One obvious change to the model is the hysteresis of the thermostat. The simulated room temperature oscillates 18–22 degrees around the temperature set point of 20 degrees. The measured room temperature oscillates 20–25 degrees with the same set point.

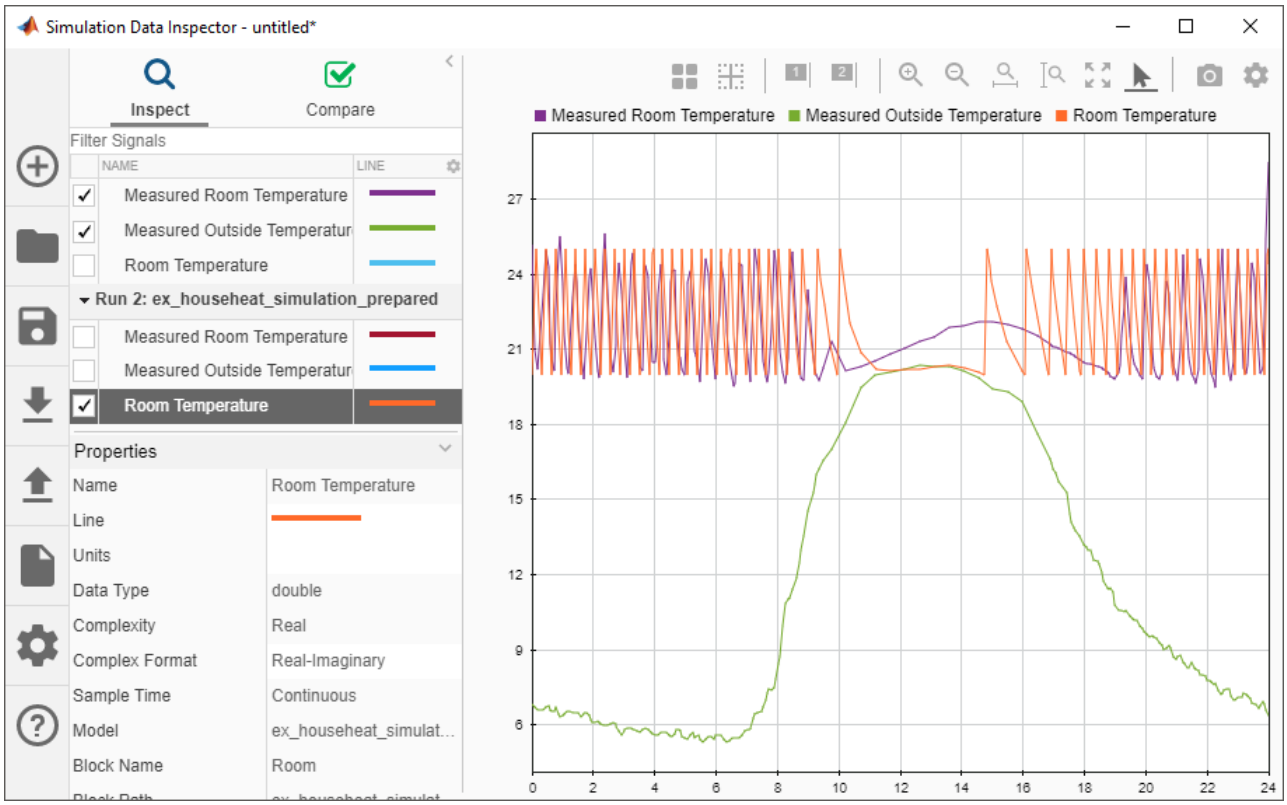
1. Open the Relay block in the Thermostat subsystem.
2. Change **Switch on point** from `2` to `0` because the difference between the room temperature and set point is 0.
3. Change **Switch off point** from `-2` to `-5`. When the room temperature is 5 degrees above the set point, you want to turn off the heater. The set point is 5 degrees below the room temperature.

Compare Results Between Simulations

Use the Simulation Data Inspector to compare differences between two simulations that use different model parameters. This comparison shows how changes improve the accuracy of your model.

1. Simulate the model.
2. Open the Simulation Data Inspector.
3. Expand the list of logged signals by selecting the arrow to the left of the run. For **Run1**, select the `Measured Outside Temperature` and `Measured Room Temperature` check boxes. For **Run2**, select the `Room Temperature` check box.

4 Review the signals. The minimum and maximum values for the simulated room temperature now match the measured room temperature values.



Messages in Simulink

Simulink Messages Overview

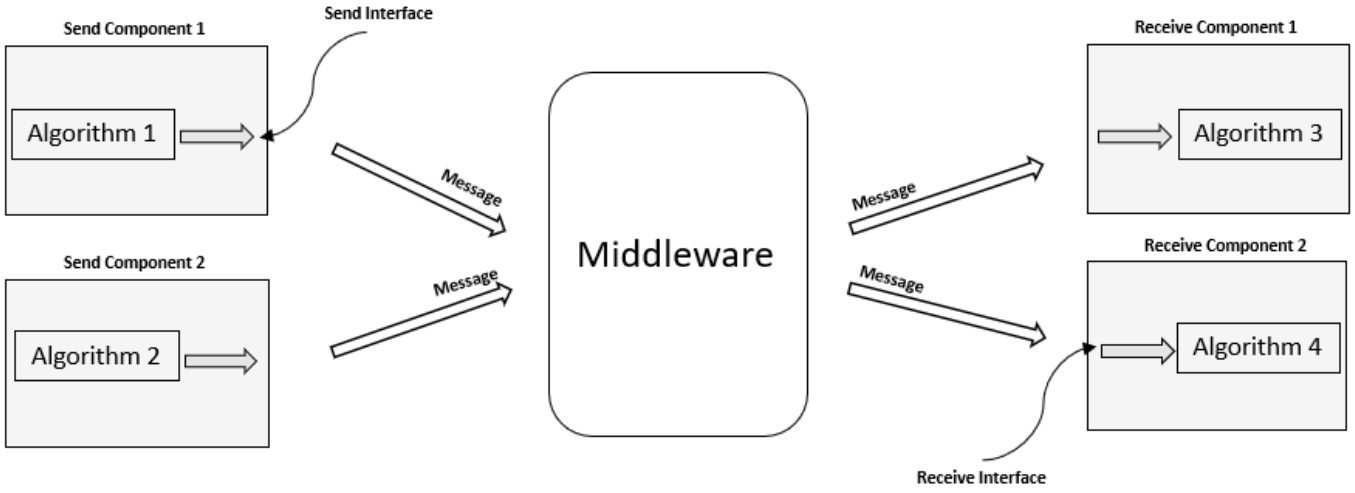
Message-based communication is necessary in various applications, such as control system architectures in which centralized architectures are replaced with distributed architectures due to the complexity of the systems. In a distributed architecture, multiple components of the system communicate via a shared network.

A distributed architecture has these three elements:

- Component — Represents partitions of a design that performs a set of functionalities or algorithms with defined I/O interfaces. Generally, components generate events and data asynchronously.
- Interface — Provides a shared boundary through which components of the system communicate. To provide asynchronous communication, messages are useful modeling artifacts that combine events with related data.
- Middleware — Provides the services needed by the components to support asynchronous communication across the shared network.

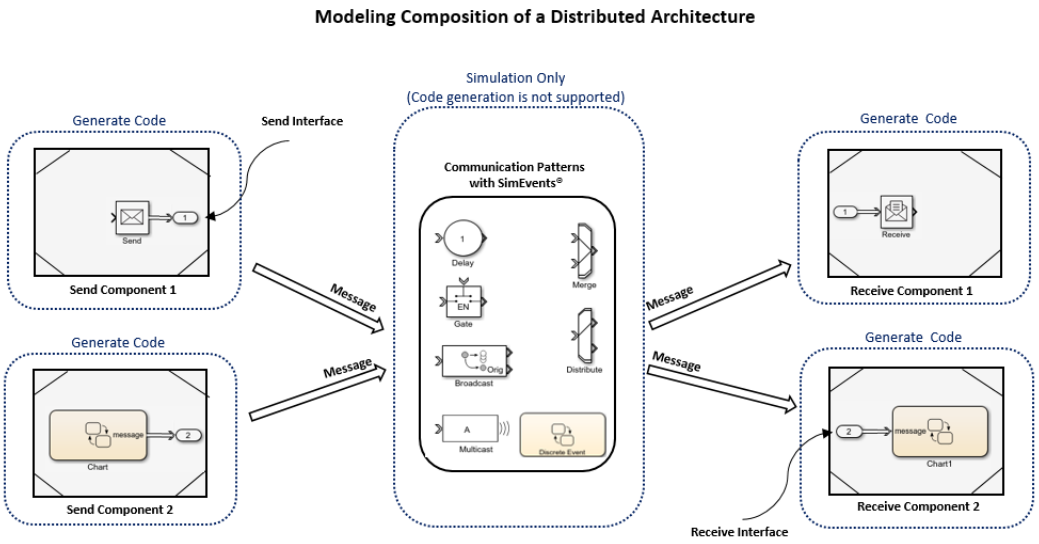
Below is an illustration that shows the composition of a distributed architecture and its elements.

Composition of a Distributed Architecture



When modeling such an architecture, you typically model components that are clearly identifiable, reusable, and deployable. To achieve asynchronous event-based communication between components, use message send and receive interfaces. Model the middleware to facilitate the network topology that represents the connectivity of components, such as one-to-many, many-to-one, or many-to-many based on the number of message sending and receiving components. For an example, see “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29.

To learn how to model a distributed architecture, using Simulink, SimEvents, and Stateflow, see the illustration below. The illustration includes two message sending and two message receiving components that are created as referenced models. Model components with send and receive interfaces using Simulink Send and Receive blocks. If your send and receive interfaces involve states or require decision logic, use a Stateflow chart.



After you model your components and interfaces:

- Simulate the behavior of your distributed architecture by modeling the middleware using SimEvents. Using the blocks from the SimEvents library, you can model custom routing and communication patterns, such as merging, delaying, distributing, and broadcasting messages, and investigate the effects of middleware on your communication network.
- Generate code for your components, including the interface, and connect to your middleware or an operating system communication API.

Model Message Send and Receive Interfaces and Generate Code

Let us start by understanding how message blocks work. To create a model that uses messages, use Send blocks to convert data and send messages and Receive blocks to receive and convert messages to data. For a simple example that shows how Send and Receive blocks work, see “Animate and Understand Sending and Receiving Messages” on page 11-5.

Use Send and Receive blocks to model message send and receive interfaces for your components. For a simple example that shows the basics of creating send and receive interfaces, see “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20. To learn how to generate code for the same model, see “Generate C++ Messages to Communicate Between Simulink Components” (Embedded Coder).

You can further modify send and receive interfaces for custom behavior. For example, you can synchronize when a receive interface executes to when data is available. For more information, see “Model a Message Receive Interface that Runs on Message Availability” on page 11-24.

After modeling, generate code for your send and receive interfaces and connect them to the middleware or an operating system communication API. For an example that generates code for a top model and allows your application to communicate in a distributed system that uses an external message protocol service (for example, DDS, ROS, SOMEIP, or POSIX messages), see “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder).

Simulate Middleware Effects on a Distributed Architecture

Use Queue blocks to store, sort and queue messages. The Queue block allows you to specify message storage capacity and the overwriting and sorting policies for message transitions. For a simple example that shows how a Queue block works, see “Use a Queue Block to Manage Messages” on page 11-10.

You can also use SimEvents to model and simulate middleware effects on your communication network. Use the blocks provided by the SimEvents library to model message routing, peer-to-peer communication, wireless communication, packet loss, and channel delays. For more information about SimEvents, see “Discrete-Event Simulation in Simulink Models” (SimEvents).

For basic communication patterns that can be modeled by SimEvents, see “Modeling Message Communication Patterns with SimEvents” on page 11-27. You can use combinations of these patterns to create more complex communication behavior. For an example of a system with multiple message sending and receiving components and an ideal shared channel with delay, see “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29. To see a model with shared wireless channel with channel failure and packet loss, see “Model Wireless Message Communication with Packet Loss and Channel Failure” on page 11-35.

To see an example that shows how to model more complex network behavior, such as an Ethernet communication network with CSMA/CD protocol, see “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 11-45.

Note SimEvents blocks do not support code generation.

See Also

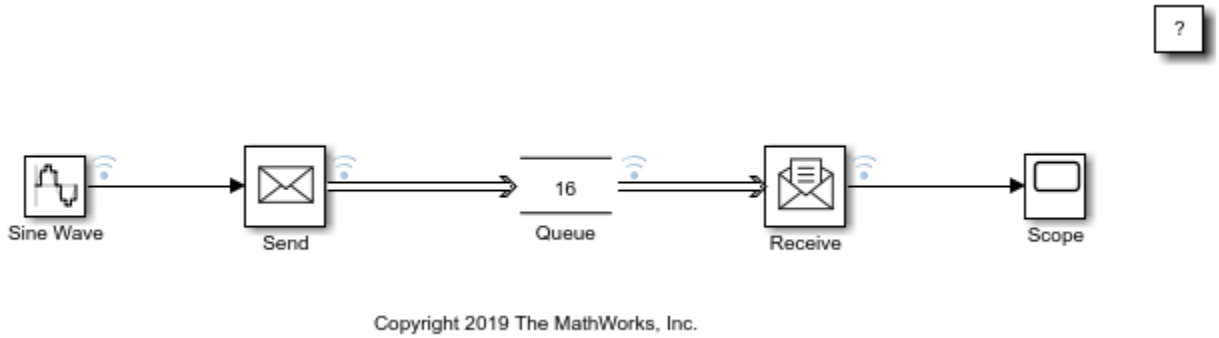
Queue | Receive | Send | Sequence Viewer | Sine Wave

More About

- “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20
- “Model a Message Receive Interface that Runs on Message Availability” on page 11-24
- “Generate C Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder)
- “Use Handwritten Code to Integrate C++ Messages with POSIX” (Embedded Coder)
- “Discrete-Event Simulation in Simulink Models” (SimEvents)

Animate and Understand Sending and Receiving Messages

This example shows how to send, receive, and visualize messages. The example also shows how to use Simulation Data Inspector, Animation, and Storage Inspector to understand how messages flow in your model.



Model Description

The SimpleMessagesModel contains these blocks:

- Sine Wave — The signal source. The **Sample time** parameter of the block is set to `0.1`.
- Send — Converts data signals and send messages. The specified value for the **Sample time** parameter of the Sine Wave block determines the rate at which the Send block sends messages. Therefore, the Send block sends one message every `0.1` simulation time.
- Queue — Stores messages. Observe the message line between the Send block and the Queue block. The default capacity of the queue is `16`, which means the Queue block can store at most `16` messages. The default message sorting behavior is `LIFO`, which means incoming messages are sorted based on last-in-first-out policy. By default, the **Overwrite the oldest element if queue is full** check box is selected. When the queue is full, an incoming message overwrites the oldest message in the queue. For more information about using the Queue block, see “Use a Queue Block to Manage Messages” on page 11-10.
- Receive — Receives messages and converts them to signal data. In the block, **Sample time** parameter is set to `0.1`. The Receive block receives a message every 0.1 simulation time.
- Scope — Visualizes messages received by the Receive block.

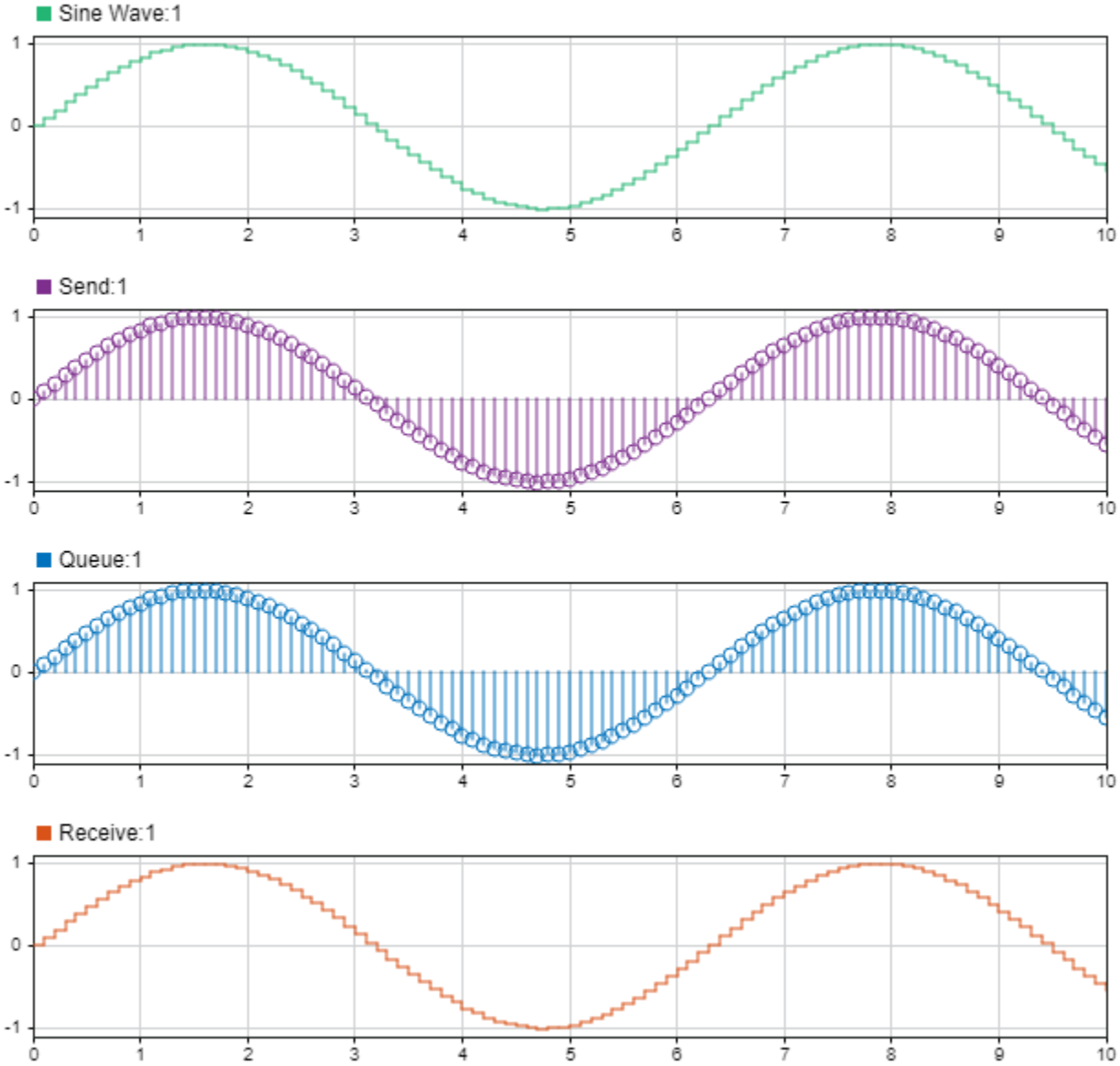
Simulate the Model and Review Results

In the model, data logging is enabled for the signal and message lines among Sine Wave, Send, Queue, Receive, and Scope blocks.

Simulate the SimpleMessagesModel and observe from the Simulation Data Inspector that:

- The Sine Wave block generates the sine wave signal (green).
- Every `0.1` simulation time, Send block converts the value of the signal to a message and sends it to the Queue block. Simulation Data inspector displays messages as stem plots. Observe the Simulation Data Inspector displaying sent messages (purple).

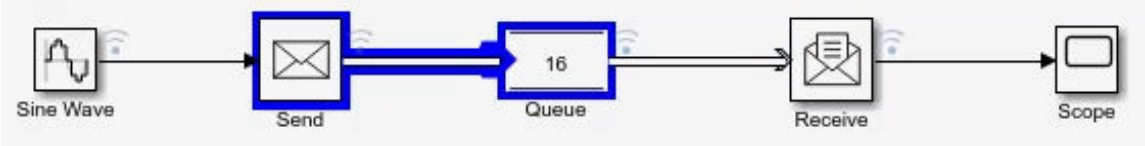
- The Queue block sends messages to the Receive block (blue).
- Receive block output is the reconstructed sine wave signal (orange).



Use Animation and Storage Inspector

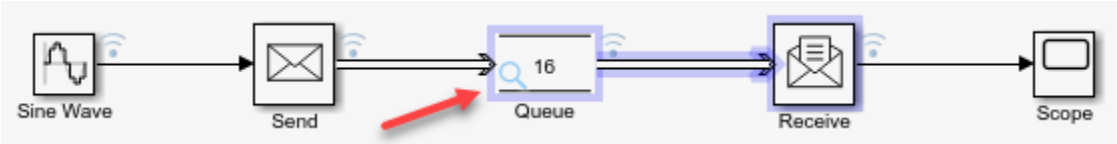
You can observe the message flow between model components by using **Animation** to animate the model. You can specify the speed of the animation as `Slow`, `Medium`, or `Fast`. A fourth option `None` disables the model animation.

In the model window right-click and select **Animation Speed**, and set its speed to `Slow`. Simulate the model again. Observe the highlighted message lines representing the message flow between the blocks.

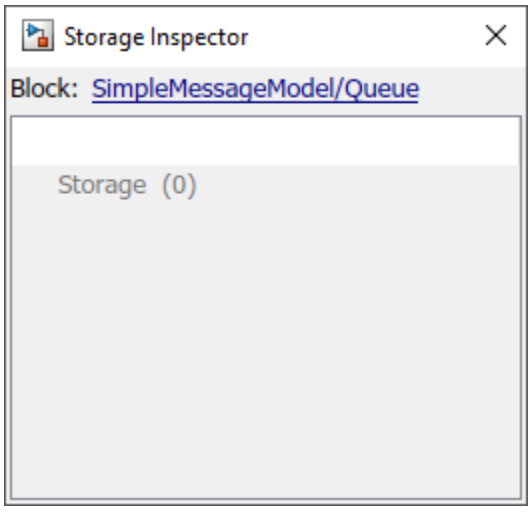


Animation highlights message lines and Simulink function calls based on event rather than time passed. In one simulation time, more than one message flow or Simulink function call event can be highlighted.

Pause the animation. In the Simulink Toolstrip, in the **Debug** tab, click **Pause**. Observe that the last highlighted message line is highlighted in violet.



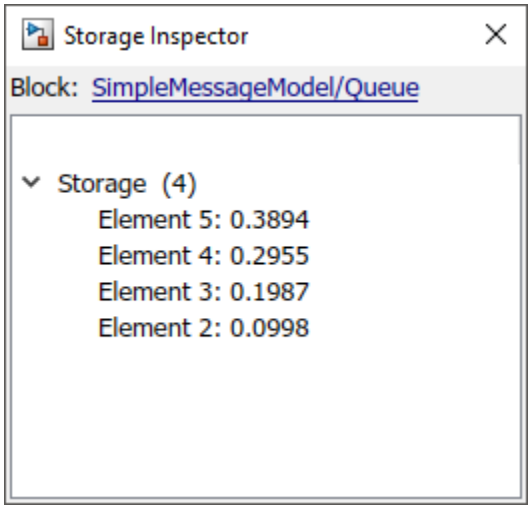
The Storage Inspector allows you to visualize the details of the stored messages in Queue block. When you pause the simulation, the Storage Inspector magnifying glass icon appears on the Queue block. To open the Storage Inspector, click the magnifying glass.



The Storage Inspector does not show any stored messages because messages arriving at the Queue block are simultaneously received by the Receive block at the same simulation time.

To create a scenario with stored messages, stop the simulation and change the **Sample time** parameter of the Receive block to `0.5`. Now the Send block sends one message every `0.1` simulation time, but the Receive block receives messages every `0.5` simulation time. This causes a backlog of messages that are stored in the Queue block.

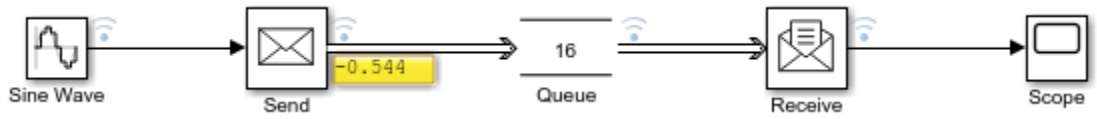
Simulate the model again with animation speed set to `Slow`, and pause the simulation to check the status in the Storage Inspector. An entity in the Storage Inspector represents stored elements in the Queue block, in this case, messages. **Storage Inspector** lists the messages stored in the Queue block with their ID and data value carried by each message.



Display Message Payload as Port Value Labels

To display message data as port value labels, right-click the message line emerging from a block and select **Show Value Label of Selected Port**.

In the example below, the port values are displayed for the message line connecting the Send block to the Queue block.



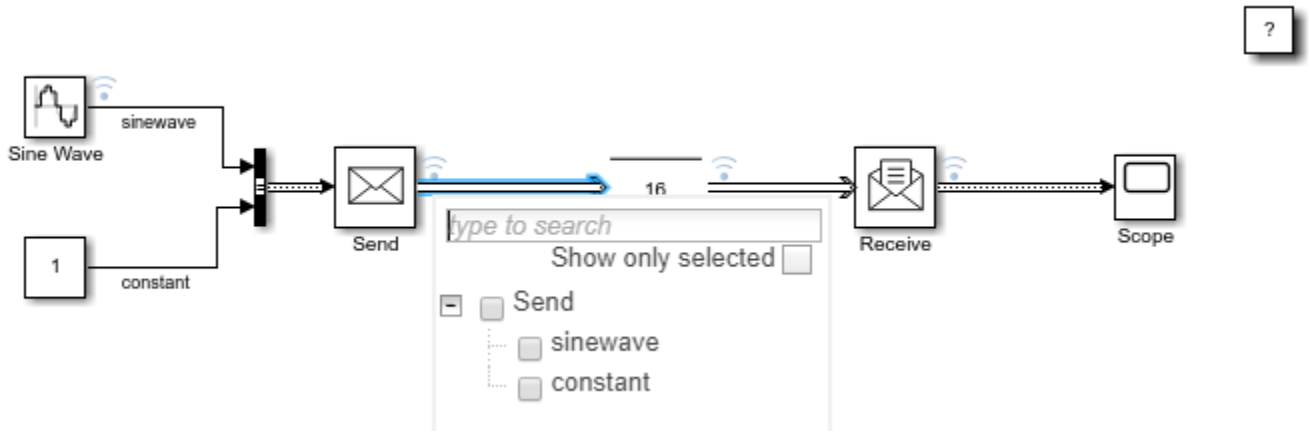
If a message carries data as a bus object, you can also select the elements to be displayed as the port values.

The `SimpleMessagesPortValueModel` is a variation of the `SimpleMessagesModel` where the input to the Send block comes from a Bus Creator block. In this example, a bus object `Data_message` is created with two elements, `sinewave` and `constant`.

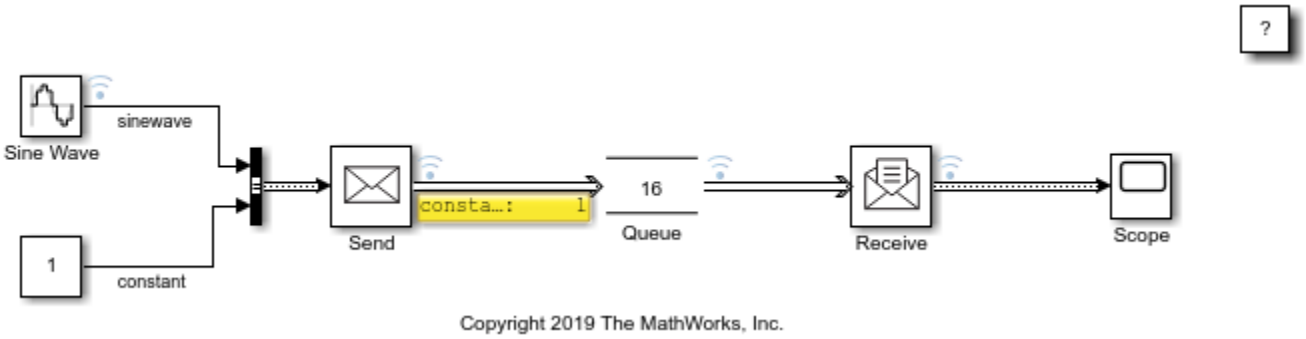
To open this model, enter:

```
open_system('SimpleMessagesPortValueModel');
```

Right-click the message line emerging from the Send block, click `Show Value Label of Selected Port`, and select constant.



Only the values of the constant signal are displayed as port values.



If the message transmission stops on a message line, the port value display keeps displaying the last message payload value.

See Also

Hit Crossing Probe | Queue | Receive | Send | Sequence Viewer | Sine Wave

More About

- “Display Port Values for Debugging” on page 36-16
- “Simulink Messages Overview” on page 11-2
- “Use a Queue Block to Manage Messages” on page 11-10
- “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20
- “Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” on page 11-58

Use a Queue Block to Manage Messages

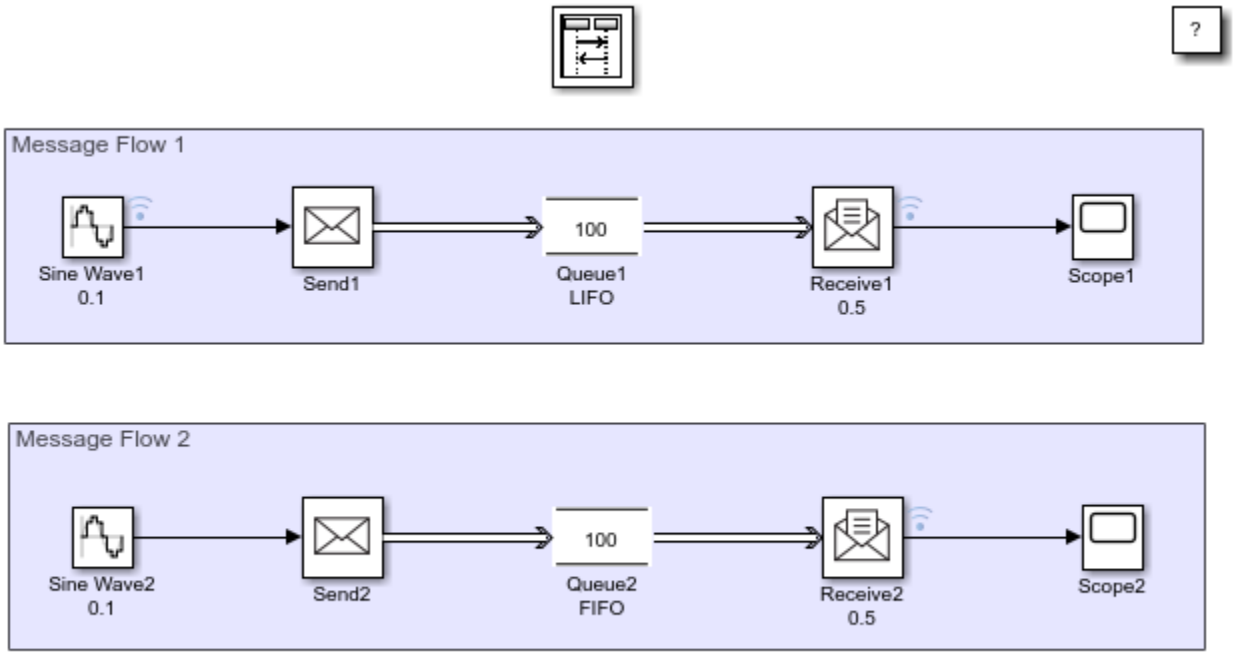
This example shows how to use a Queue block to store and queue messages. The Queue block allows you to specify message storage capacity, overwriting policy, and sorting policy during message transitions.

Manipulate Message Transitions Using Queue Sorting Policies

The Queue block supports three message sorting policies:

- Last-in-first-out (FIFO) — The newest message in the storage departs first.
- First-in-first-out (FIFO) — The oldest message in the storage departs first.
- Priority — Messages are sorted based on their priority. The priority queue can be used only when the **Overwrite the oldest element if queue is full** check box is cleared.

This example uses a simple message-based communication model that is introduced in “Animate and Understand Sending and Receiving Messages” on page 11-5. The model includes Message Flow 1 and Message Flow 2 in which messages flow from Sine Wave1 to Scope1 and Sine Wave2 to Scope2.



Model Description

In Message Flow 1 and Message Flow 2:

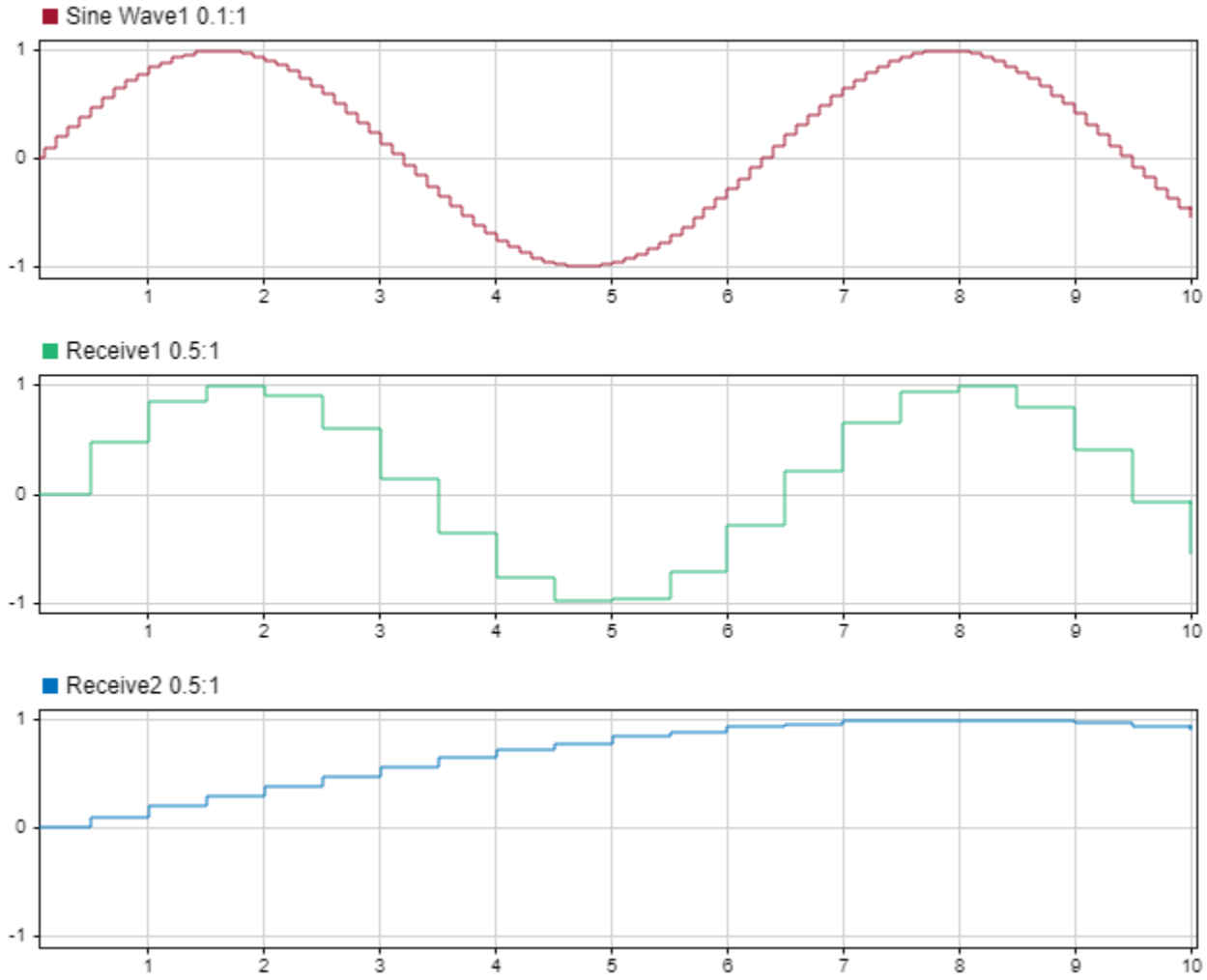
- The **Sample time** parameter of Sine Wave1 and Sine Wave2 are set to `0.1`. They are identical sine wave sources.
- The **Sample time** parameter of Receive1 and Receive2 are set to `0.5`. They are identical Receive blocks.

- The capacity of the Queue1 and Queue2 are `100`. Both queues have the same capacity.
- The sorting policy of the Queue1 is `LIFO` and Queue2 is `FIFO`, which is displayed under the block labels. The only difference between two message flow scenarios is the queue sorting policy.
- The signals among Sine Wave1, Receive1, and Receive2 are logged.

Simulate the Model and Review the Results

Simulate the model and observe the results in the Simulation Data Inspector.

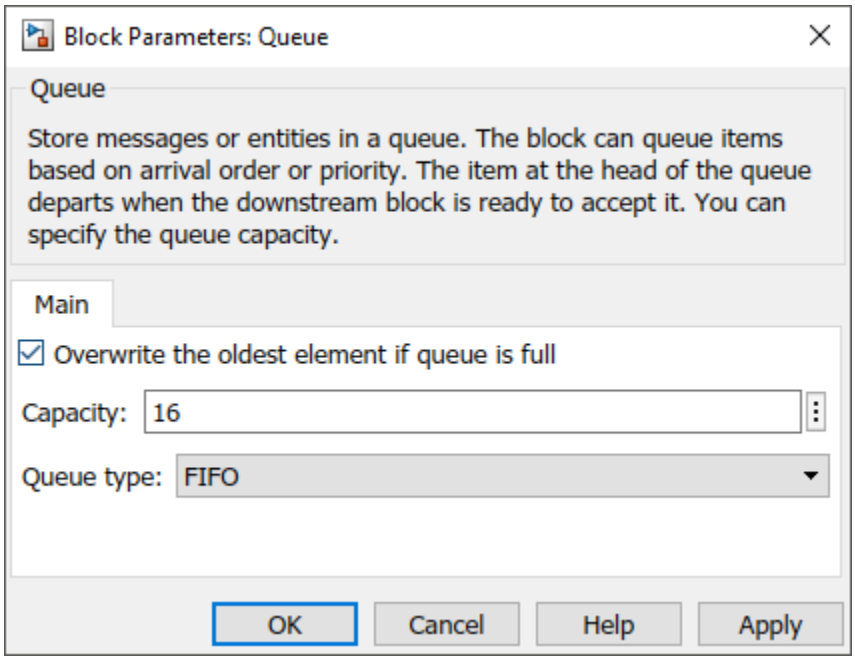
Observe the output from Sine Wave1. The output from SineWave2 is not displayed because it is identical to SineWave1. Also observe that the signal from Receive1 is the representation of the sine wave but with longer intervals between samples. However, the signal from Receive2 is the first part of the sine wave reaching to the positive peak because the sorting policy of the Queue block is set to `FIFO` and its capacity is large enough to store messages.



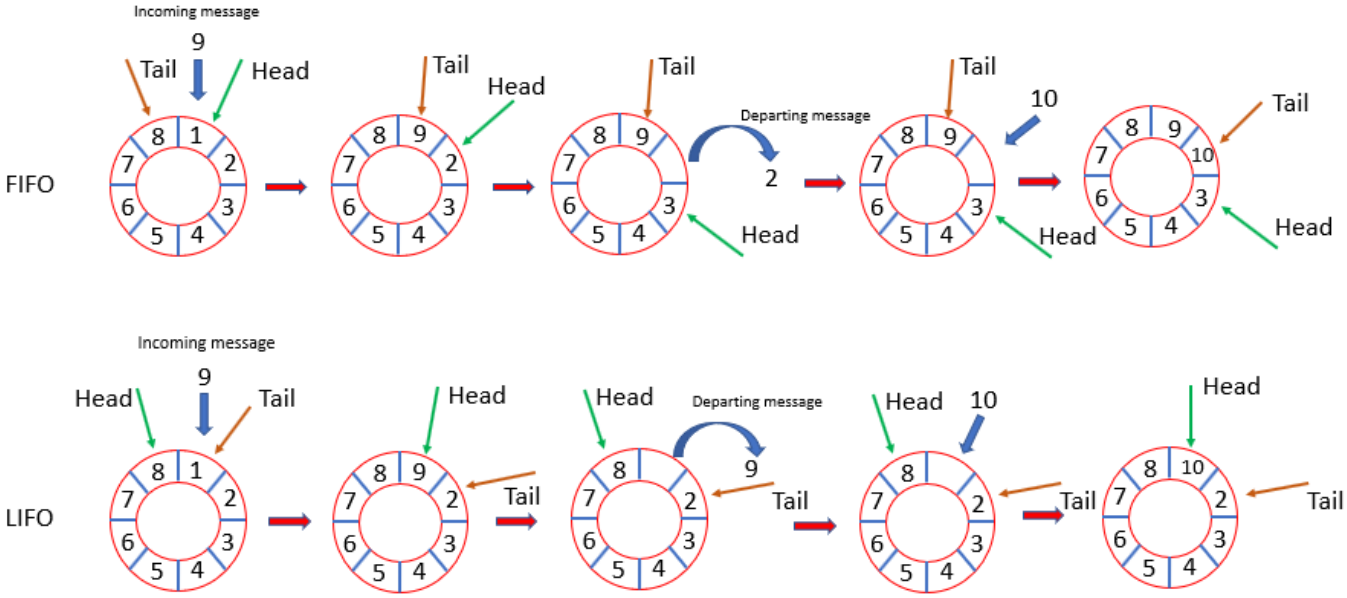
Queue Block Message Overwriting Policies

You can also specify the Queue block message overwriting policy when the queue is full:

- By default, the **Overwrite the oldest element if queue is full** check box is selected. The block is set to always accept an incoming message, overwriting the oldest message in the storage. In this case, the block overwrites the oldest message, but the message departing the block is determined by the queue sorting policy.



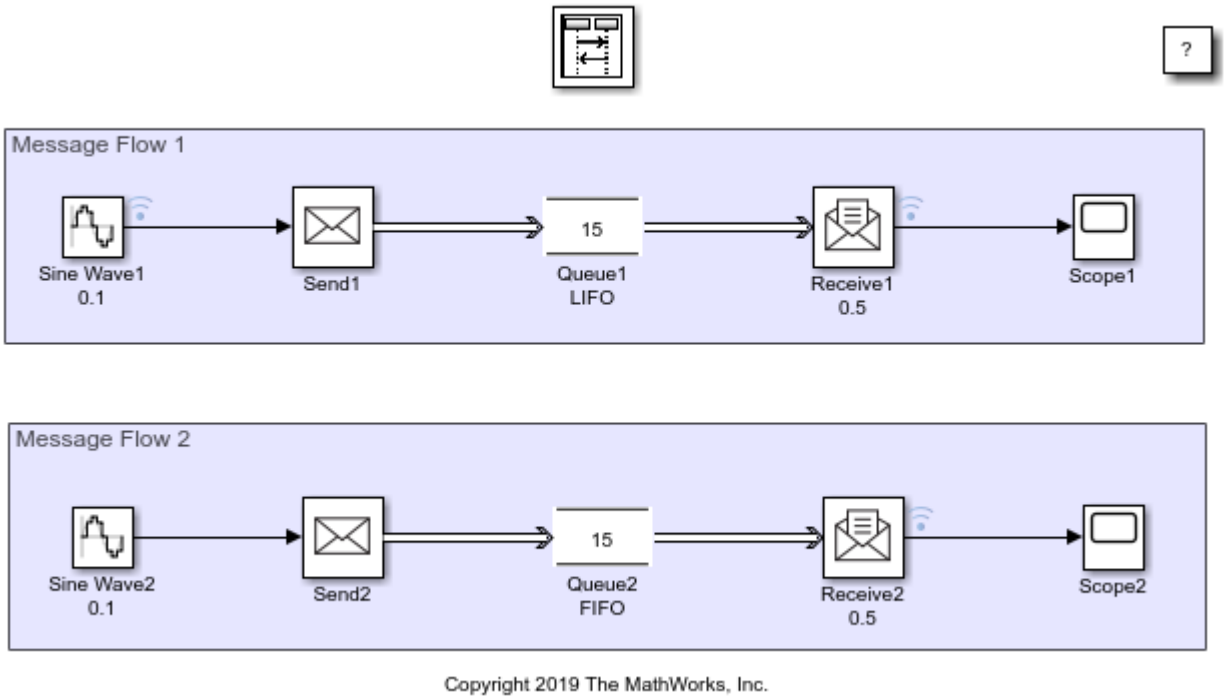
In this example of queue behavior, messages are enumerated based on their arrival time at the FIFO and LIFO queues with capacity `8`. A new message with number `9` arrives. In FIFO and LIFO cases, message `9` replaces message `1` because it is the oldest element in the queue. However, observe the change of head and tail of the queue during this process. In the FIFO queue, message `2` departs first because it is the oldest message after message `1` is replaced. In the LIFO queue, message `9` departs first because it is the last message that arrives at the storage. After the message departure, a new message `10` arrives. The queue accepts the new messages to its empty storage bin.



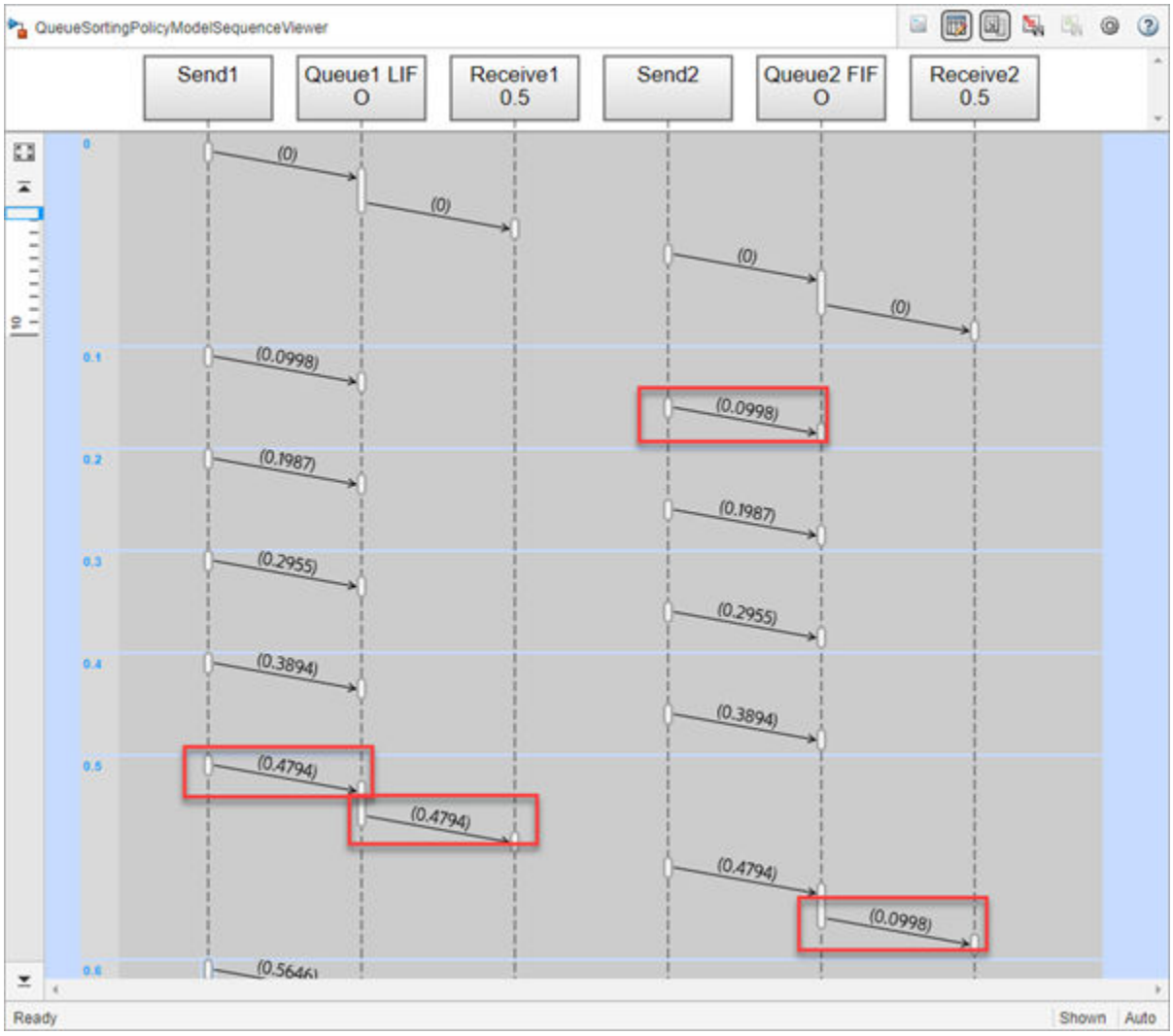
- If the **Overwrite the oldest element if queue is full** check box is cleared, the icon of the Queue block changes and the block does not accept new messages when the queue is full. This is a blocking queue behavior.

Queue Block with Overwriting Policy Enabled

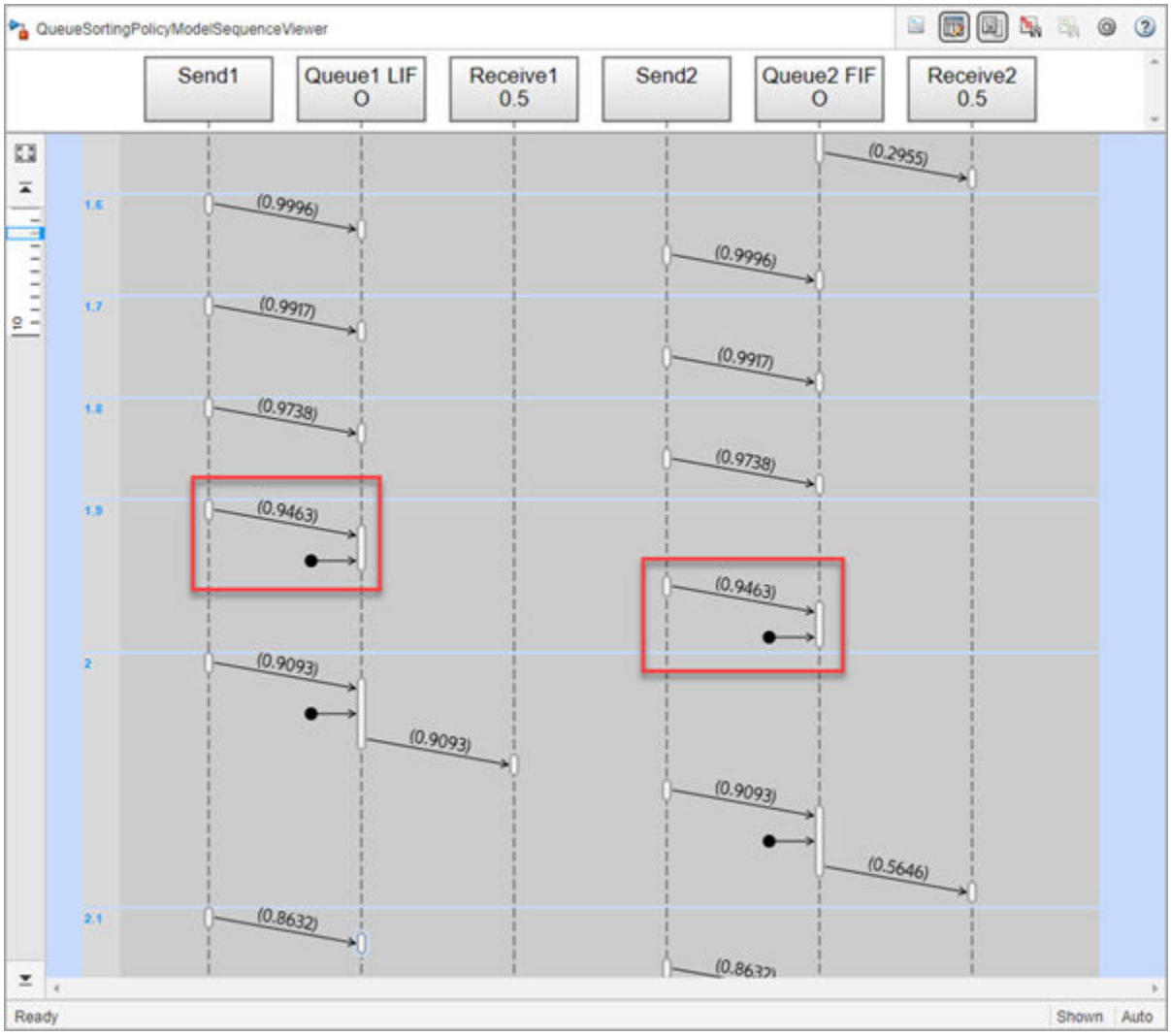
In the QueueSortingPoliciesModelSequenceViewer model, the capacities of the Queue blocks in Message Flow 1 and Message Flow 2 are changed to 15. Decreasing the capacity causes the incoming messages to overwrite the existing ones in the storage.



Simulate the model and open the Sequence Viewer block. In the Sequence Viewer block, scroll up or click the **Go to first event** icon on the left. Observe the messages departing the block based on FIFO and LIFO policies.

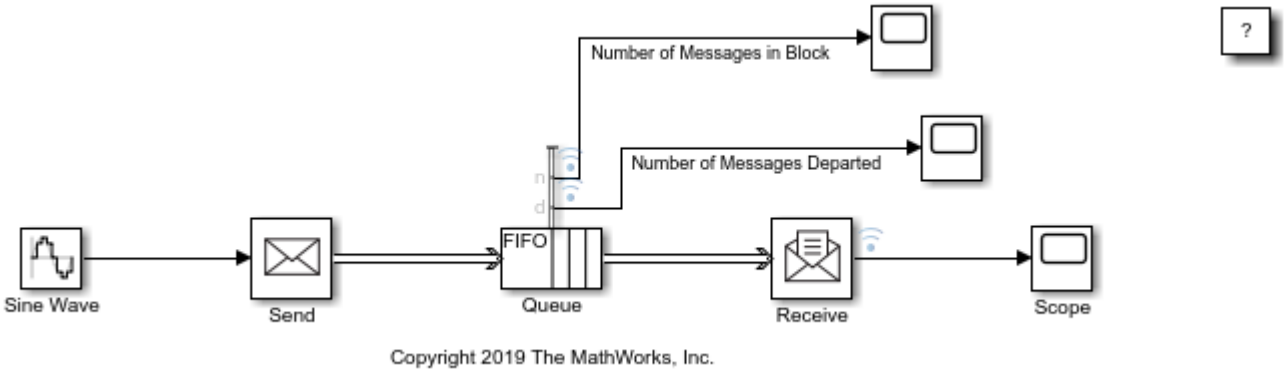


Both Queue blocks have limited capacity. When their capacity is full, a new incoming message overwrites the oldest existing message in the queue. Scroll down and observe that the Sequence Viewer block displays the messages that overwrite the existing ones.



An Example of a Blocking Queue Behavior

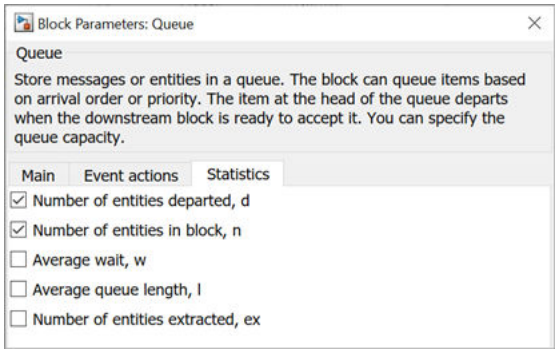
Open the QueueOverWritingPolicyModel to inspect the blocking queue behavior.



In the Receive block, set the **Sample time** parameter to `0.5`.

Observe that in the Queue block:

- The **Overwrite the oldest element if queue is full** check box is cleared. Observe the block icon change.
- The **Capacity** parameter is set to `16`.
- In the **Statistics** tab, the **Number of entities departed, d** and **Number of entities in block, n** check boxes are selected.



Simulate the Model and Review Results

Simulate the model. Observe the warning displayed in the Diagnostic Viewer. Messages sent by the Send block were dropped during simulation.

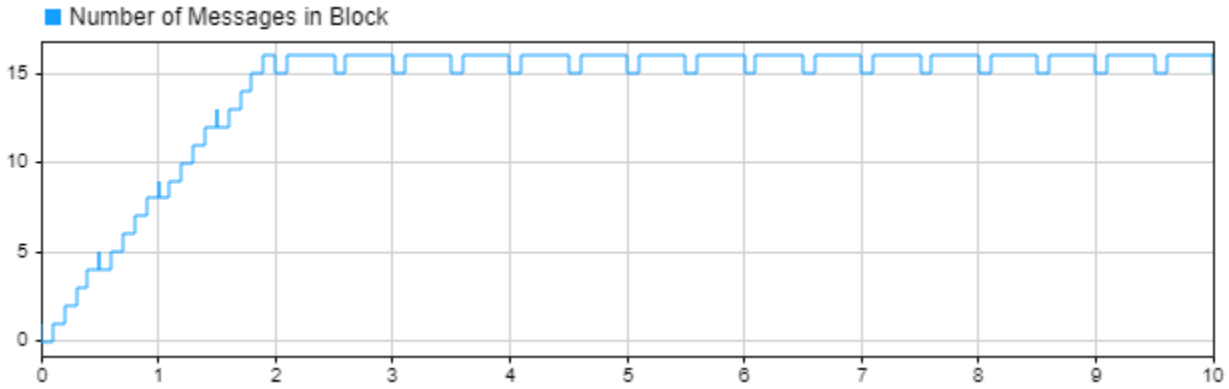
```
Unable to send message from output port 1 of block 'QueueOverWritingPolicyModel/Send' because the
output port is blocked. 65 messages were dropped during this simulation. To prevent messages from
being dropped, increase the capacity of message storage connected to this output port.

Component: Simulink | Category: Block warning
```

The Queue block blocks the messages when the **Overwrite the oldest element if queue is full** check box is cleared. You can increase the capacity of the Queue block to prevent message dropping.

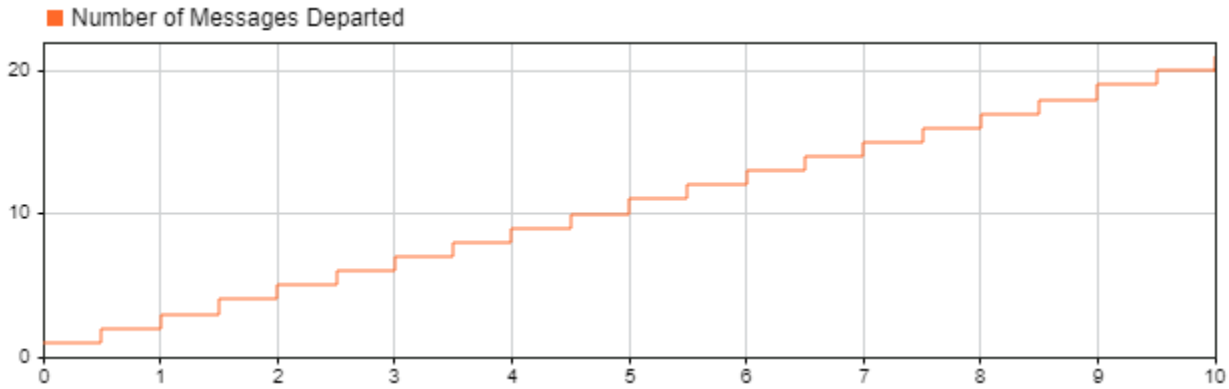
Use Statistics to Analyze Queue Behavior

When the **Overwrite the oldest element if queue is full** check box is cleared the **Statistics** tab is enabled. Use **Statistics** tab to enable output ports and observe the Queue block statistics. Statistics are not supported for code generation.

Observe the signals labeled Number of Messages in Block and Number of Messages Departed. Up to simulation time `2`, there are `16` messages in the storage, which is the queue capacity. After that, Number of Messages in Block takes values `15` and `16` because messages depart every `0.5` simulation time and a new message arrives.



At simulation time `10`, a total of `21` messages departs the block.



Use Event Actions

You can also use **Event actions**, when the **Overwrite the oldest element if queue is full** check box is cleared. Event actions are not supported for code generation. For more information, see "Events and Event Actions" (SimEvents).

Use **Event actions** to specify the behavior of the message in certain events. For instance, the **Entry** and the **Exit** actions are called just after the message entry and just before message exit. The **Blocked** action is called after a message is blocked.

For more information, see "Model a Message Receive Interface that Runs on Message Availability" on page 11-24.

You can also model more complex communication policies by using blocks from the SimEvents® library, which requires a SimEvents® license.

See Also

Hit Crossing Probe | Queue | Receive | Send | Sequence Viewer | Sine Wave

More About

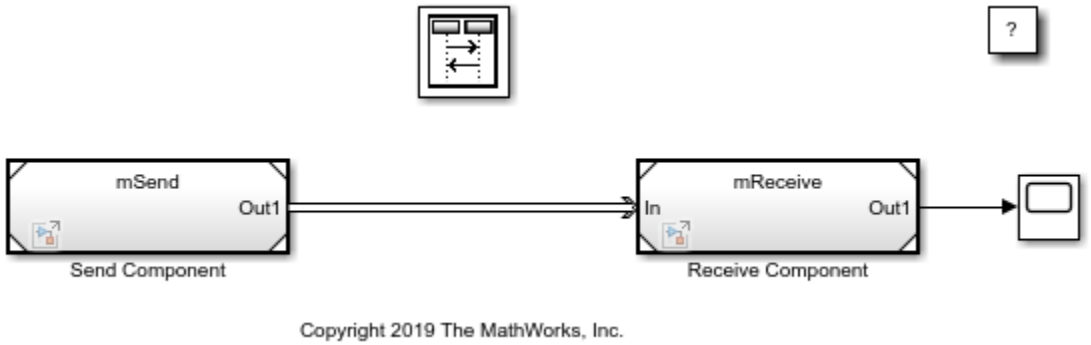
- "Simulink Messages Overview" on page 11-2
- "Animate and Understand Sending and Receiving Messages" on page 11-5

- “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20
- “Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” on page 11-58

Establish Message Send and Receive Interfaces Between Software Components

This example shows how to model message-based communication between software components that run in different applications. The example also shows how to prepare model components for C++ code generation.

In this example, the message-based communication is constructed between two components. Send component sends data and receive component consumes data. In this scenario, after send component sends messages, they are stored in a queue. Receive component pulls a message based on the logic it represents.



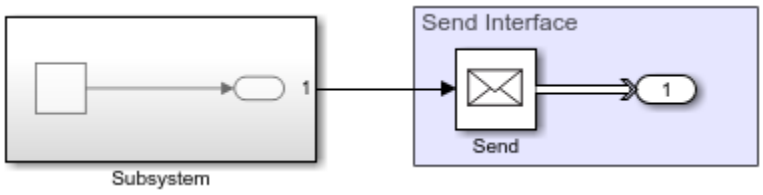
Two Model blocks, labeled Send Component and Receive Component, represent the components connected by a message line. Message-based communication is achieved using a Send block and a Receive block that are connected to the root-level Outport and Inport blocks.

For more information about generating C or C++ code for the model, see “Generate C++ Messages to Communicate Between Simulink Components” (Embedded Coder) and “Generate C Messages to Communicate Between Simulink Components” (Embedded Coder).

You can also generate C++ code for each component, and the code contains necessary software interfaces that are sufficient for you to connect with an operating system or message middleware. For more information, see “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder).

Send Component

The algorithm in the Send Component can contain logic of any complexity. In the example, a simple Sine Wave block is used in a subsystem as the signal source. The **Sample time** parameter of the block is set to `0.1`.



To prepare the Send Component for message-based communication, a Send block is connected to the root-level Outport block. The Send block converts data signals and send messages.

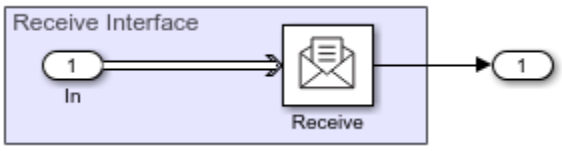
To prepare the Send Component for code generation, in the Model Configuration Parameters:

1. In the **Solver** pane, in the **Solver selection** section, the **Type** is set to `Fixed-step`.
2. The **Fixed-step size** is set to `0.1`.
3. In the **Code Generation** pane, the **System target file** is set to `ert.tlc` and **Language** to C++.
4. The model is saved as `mSend`.

Receive Component

In the Receive Component, a Scope block is used to represent the algorithm that receives messages.

To prepare the Receive Component, the Inport block is connected to a Receive block. The Receive block receives messages and converts them to signal data. By default, the **Sample time** parameter of the Receive block is `-1`.



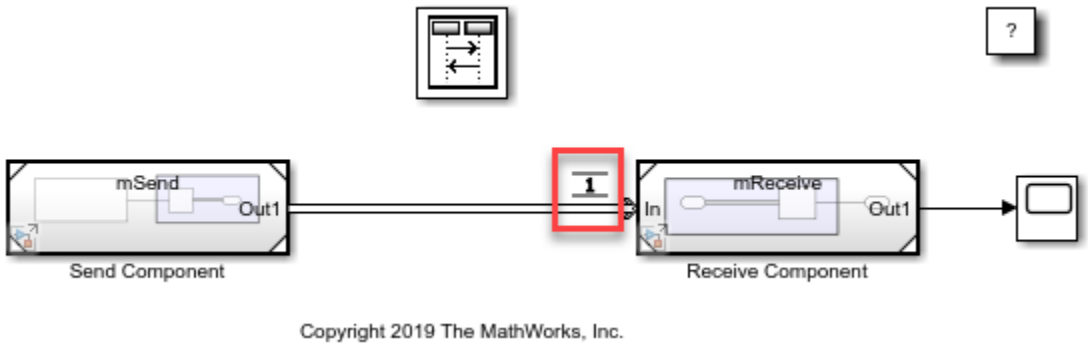
To prepare the Receive Component for code generation, in the Model Configuration Parameters:

1. In the **Solver** pane, in the **Solver selection** section, the **Type** is set to `Fixed-step`.
2. The **Fixed-step size** is set to `0.1`.
3. In the **Code Generation** pane, the **System target file** is set to `ert.tlc` and **Language** to C++.
4. The model is saved as `mReceive`.

Visualize Message Transitions Between Components Using the Sequence Viewer Block

This is a composition model with Send and Receive components. The Model blocks, Send Component and Receive Component, refer to models `mSend` and `mReceive`, respectively.

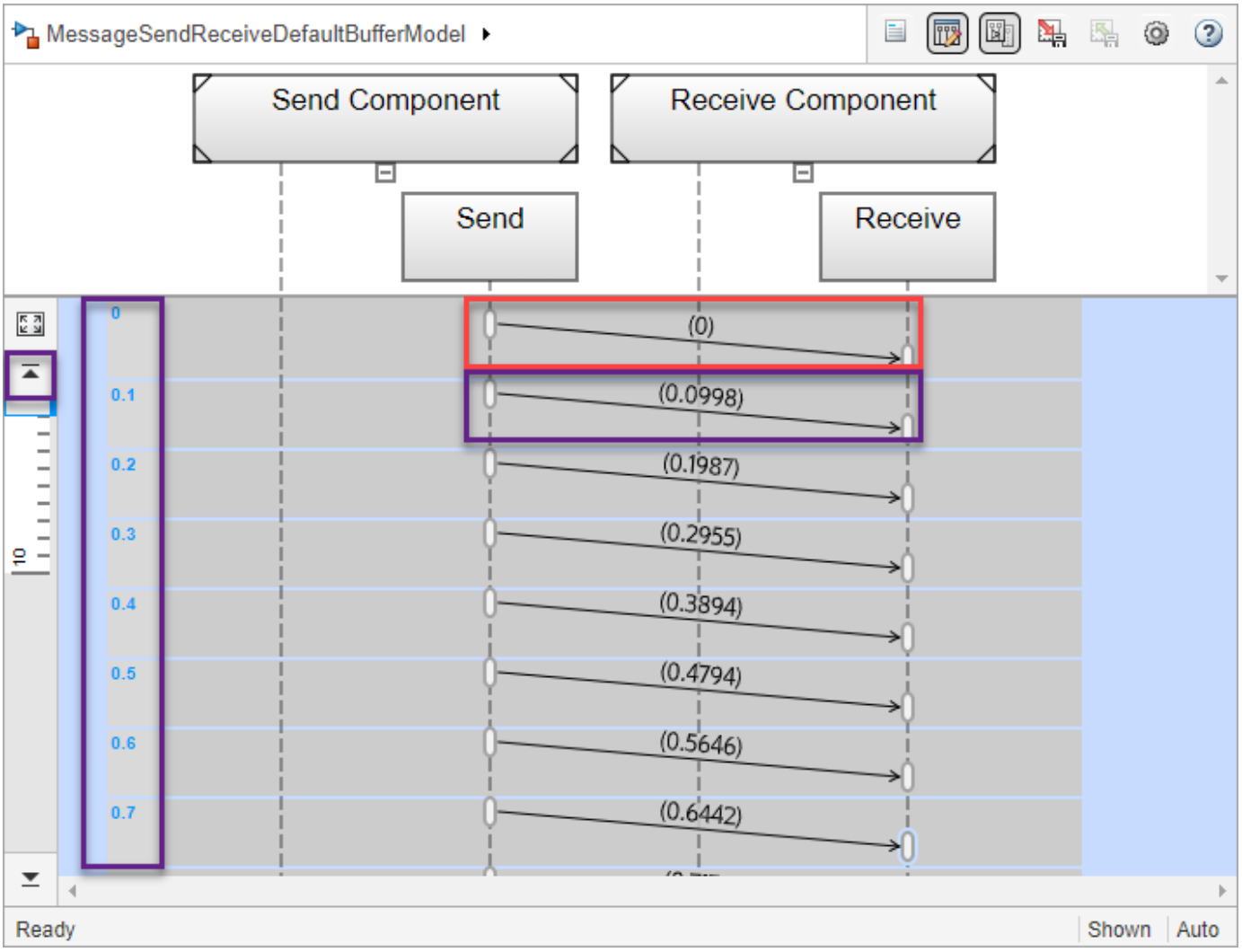
Simulate the model. Observe the queue inserted by default. An icon above the message line represents the default queue. The capacity of the default queue is `1` and the message overwriting policy is enabled. You can customize the queue by using a Queue block between components. For more information, see “Use a Queue Block to Manage Messages” on page 11-10.



Open the Sequence Viewer block. The block allows you to visualize message transition events and the data that the messages carry.

The Sequence Viewer block window shows the simulation time in the left vertical bar. Each time grid row contains events that occur at the same simulation time. Each message transition event is represented by an arrow that is labeled with the message data value. For more information about the Sequence Viewer block, see “Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” on page 11-58.

In the Sequence Viewer block, scroll up or click **Go to first event** icon on the left. Observe that at time zero the Send block sends a message with data value `0` to the Receive block, and at time `0.1` the block sends another message with data value `0.0998`. The block sends a message in every `0.1` simulation time.



See Also

Queue | Receive | Send | Sequence Viewer | Sine Wave

More About

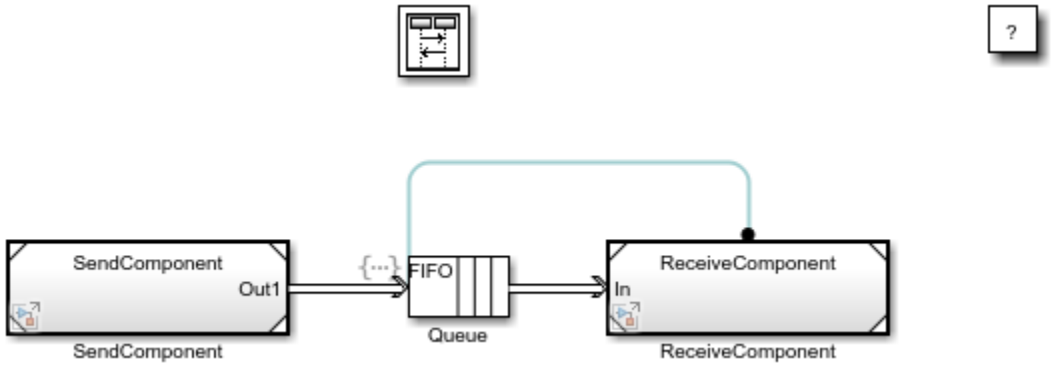
- “Simulink Messages Overview” on page 11-2
- “Model a Message Receive Interface that Runs on Message Availability” on page 11-24
- “Generate C Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder)
- “Use Handwritten Code to Integrate C++ Messages with POSIX” (Embedded Coder)
- “Animate and Understand Sending and Receiving Messages” on page 11-5

Model a Message Receive Interface that Runs on Message Availability

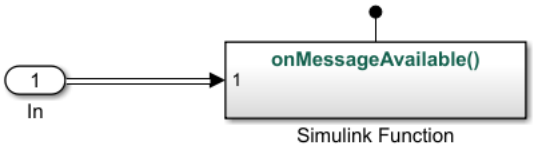
This example shows how to synchronize execution with data availability in message-based systems. We show how to model this behavior using events and actions. This example builds on another example, “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20, where two software components communicate using messages.

As with the previous example, the code generation process should apply to the SendComponent and the ReceiveComponent. The Message Storage models the middleware and data notification events provided by the middleware.

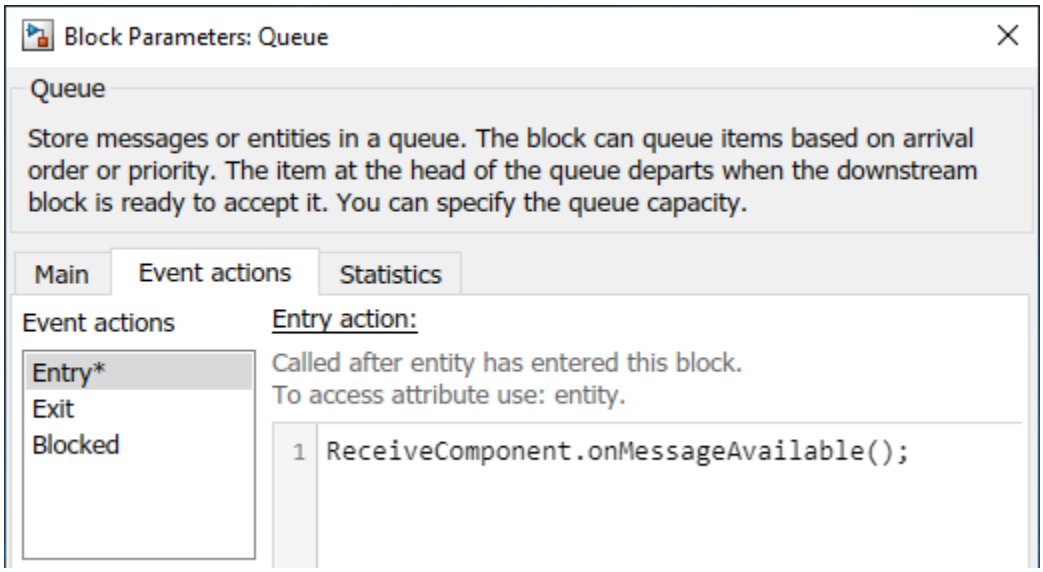
Below is the composition model with a SendComponent and a ReceiveComponent created by using two Model blocks. These two Model blocks are connected through a Queue block with message lines.



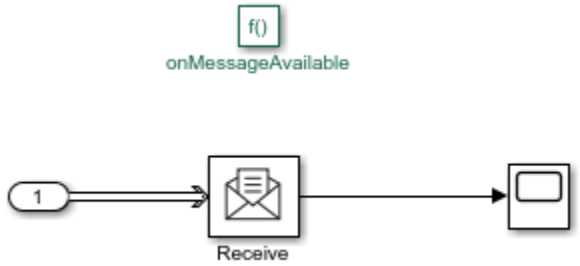
When a message arrives at the Queue block, its arrival is communicated to the ReceiveComponent by using Queue entry event action. This activates the Simulink Function block inside the ReceiveComponent to accept a new message for processing.



To achieve this behavior, in the Queue block, in the **Event action** tab, in the **Entry** field, the block calls the Simulink Function `onMessageAvailable()`. See “Event Action Languages and Random Number Generation” (SimEvents) for more information on Event Actions.



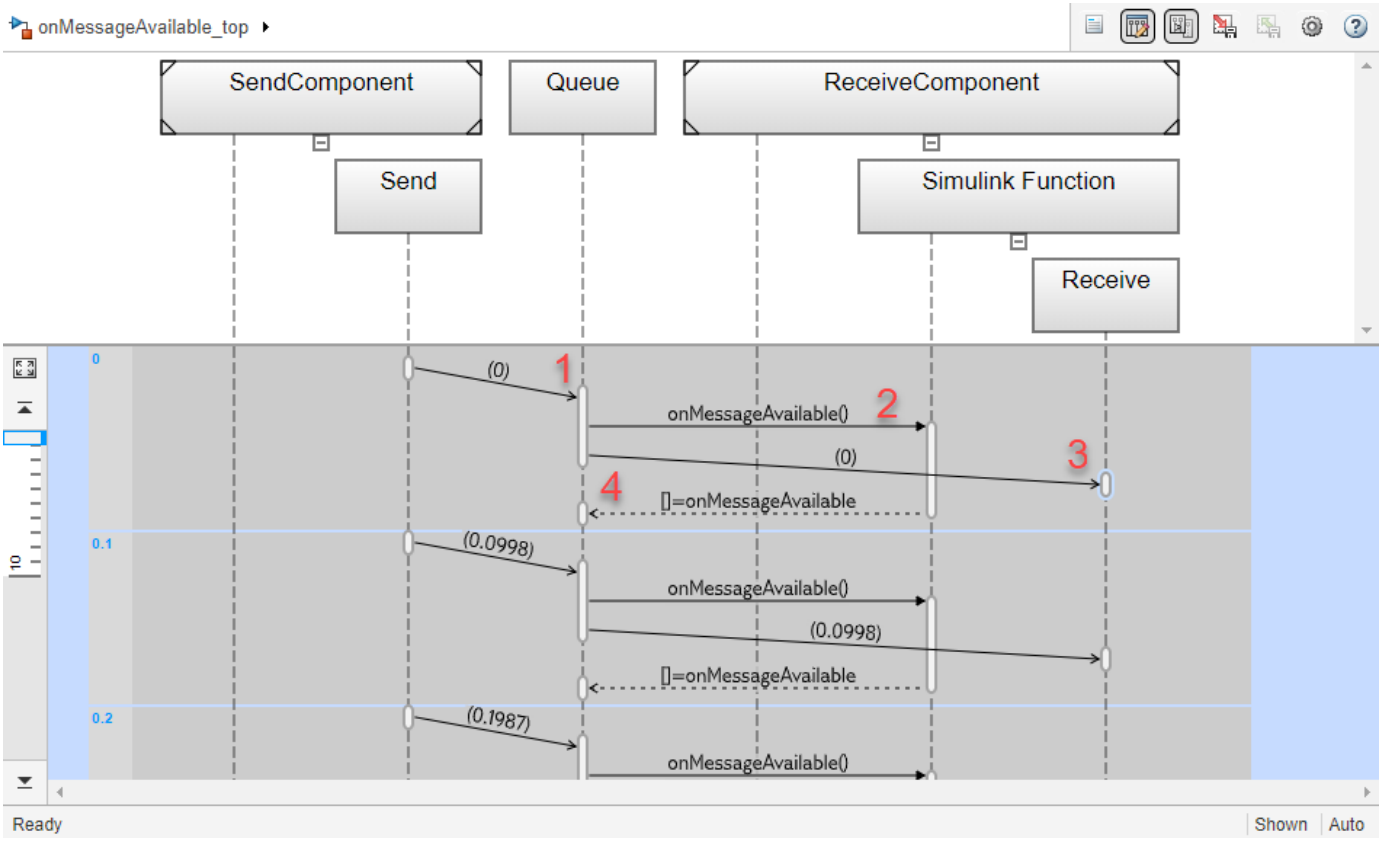
When the The Simulink Function block is activated, it accepts a message and converts it to a signal.



As a result, messages sent from the SendComponent trigger execution within the ReceiveComponent. To observe this behavior, use the Sequence Viewer block.

For example, observe the simulation time `0`:

1. The arrow from the Send block to the Queue block represents that a message is sent with a data value of `0`.
2. The arrow from the Queue block to the Simulink Function block indicates a call to the `onMessageAvailable()` function.
3. An arrow from the Queue block illustrates that the message with data value `0` is received by the Receive block within this function call.
4. The horizontal, dashed arrow from the Simulink Function block to the Queue block indicates the return of function `onMessageAvailable()`.



For more information about the Sequence Viewer block, see “Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” on page 11-58.

See Also

Queue | Receive | Send | Sine Wave

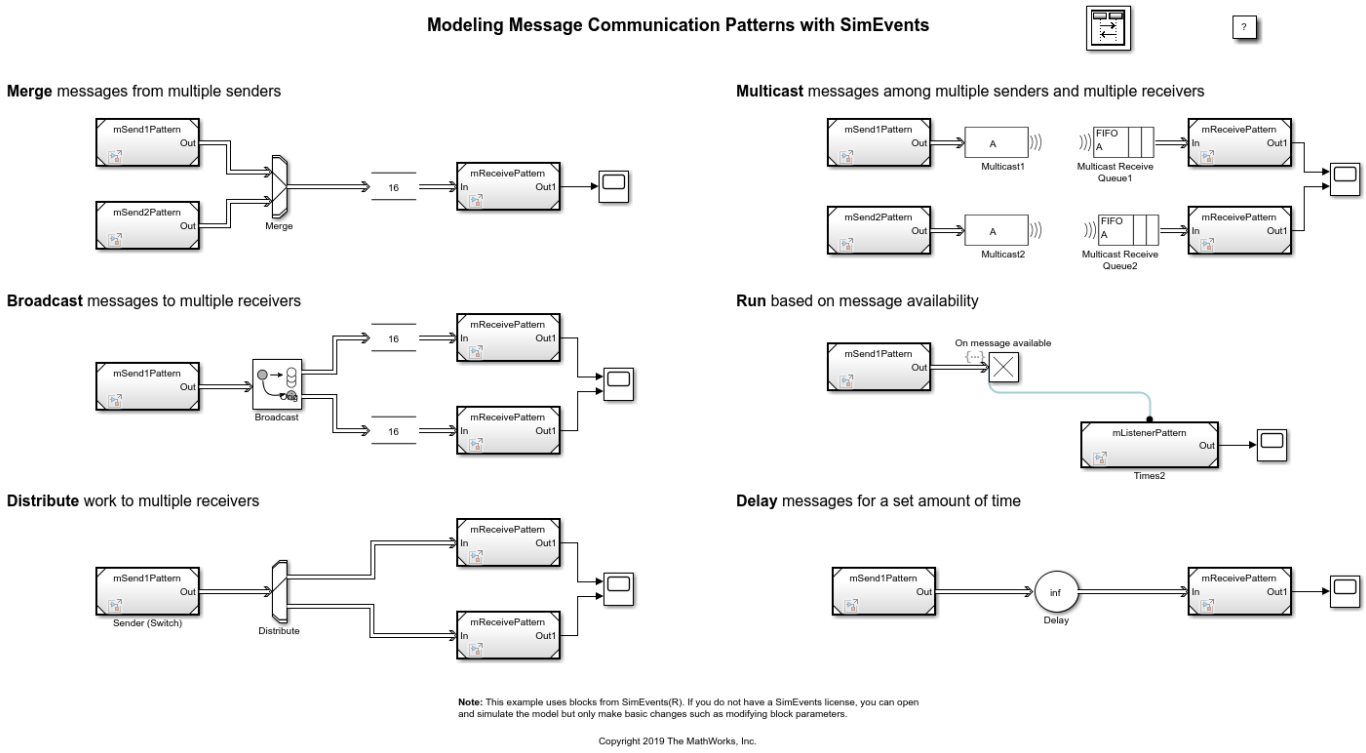
More About

- “Simulink Messages Overview” on page 11-2
- “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20
- “Generate C Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink Components” (Embedded Coder)
- “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder)
- “Use Handwritten Code to Integrate C++ Messages with POSIX” (Embedded Coder)

Modeling Message Communication Patterns with SimEvents

This example shows how to create common communication patterns using SimEvents®. In message-based communication models, you can use SimEvents® to model and simulate middleware, and investigate the effects of communication and the environment on your distributed architecture.

The systems in this example represent common communication patterns created by using SimEvents® blocks that can be used to simulate various network types, such as cabled or wireless communication, and channel behavior such as failure, or packet loss.



The communication patterns involve:

- Merging messages from multiple senders.
- Broadcasting messages to multiple receivers.
- Distributing work to multiple receivers.
- Multicasting messages among multiple senders and multiple receivers.
- Running a component based on message availability and data.
- Delaying messages for a set amount of time.

To create more complex networks and channel behavior, use combinations of these simple patterns.

By using these patterns, you can model:

- *N* -to- *n* communication with multiple senders and receivers with an ideal channel with communication delay. For an example, see “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29.

- *N* -to- *n* communication with channel failure and packet loss. For an example, see “Model Wireless Message Communication with Packet Loss and Channel Failure” on page 11-35.
- An *N* -to- *n* Ethernet communication network with an inter-component communication protocol. For an example, see “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 11-45.

See Also

Queue | Receive | Send | Sine Wave

More About

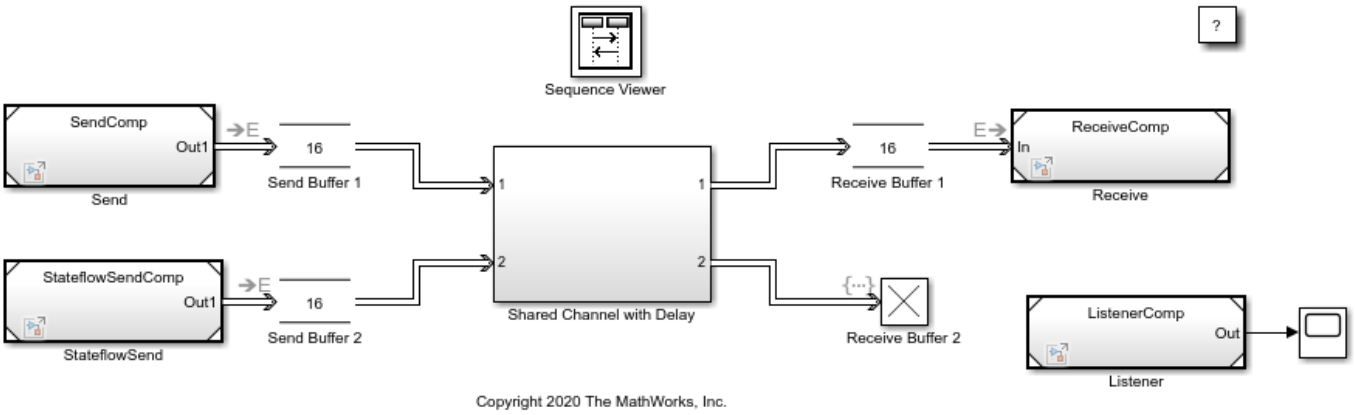
- “Simulink Messages Overview” on page 11-2
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29
- “Model Wireless Message Communication with Packet Loss and Channel Failure” on page 11-35
- “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 11-45

Build a Shared Communication Channel with Multiple Senders and Receivers

This example shows how to model communication through a shared channel with multiple senders and receivers by using Simulink® messages, SimEvents®, and Stateflow®.

For an overview about messages, see “Simulink Messages Overview” on page 11-2.

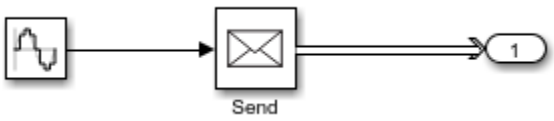
In this model, there are two software components that send messages and two components that receive messages. The shared channel transmits messages with an added delay. SimEvents® blocks are used to create custom communication behavior by merging the message lines, and copying and delaying messages. A Stateflow® chart is used in a send component to send messages based on a decision logic.



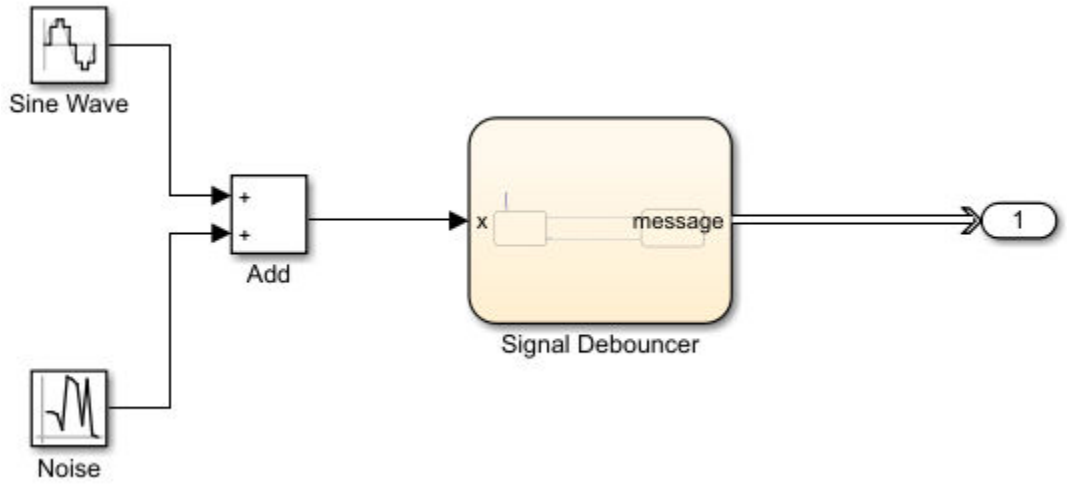
Create Components to Send Messages

In the model, there are two software components that output messages, Send and StateflowSend.

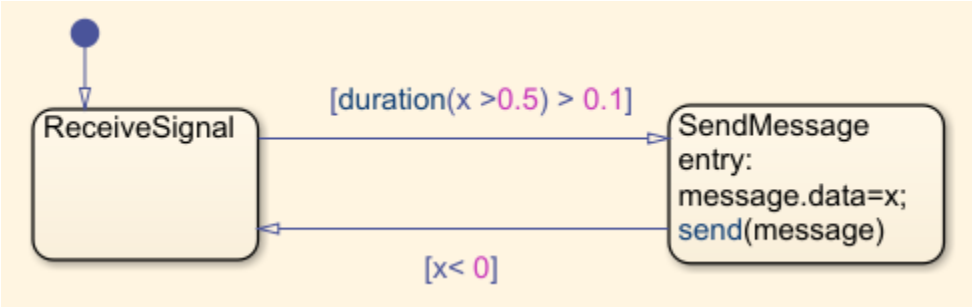
In the Send component, the Sine Wave block is the signal source. The block generates a sine wave signal with an amplitude of `1`. The block's sample time is `0.1`. The Send block converts the signal to a message that carries the signal value as data. The Send component sends messages to Send Buffer 1.



In the StateflowSend component, another Sine Wave block generates a sine wave signal and a Noise block injects noise into the signal. The Noise block outputs a signal whose values are generated from a Gaussian distribution with mean of `0` and variance of `1`. The sample time of the block is `0.1`.



The Stateflow® chart represents a simple logic that filters the signal and decides whether to send messages. If the value of the signal is greater than `0.5` for a duration greater than `0.1`, then the chart sends a message that carries the signal value. If the signal value is below `0`, then the chart transitions to the `ReceiveSignal` state. The StateflowSend component sends messages to Send Buffer 2.



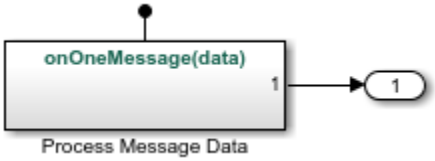
For more information about creating message interfaces, see “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20.

Create Components to Receive Messages

In the model, there are two software components that receive messages, Receive and Listener.

In the Receive component, a Receive block receives messages and converts the message data to signal values.

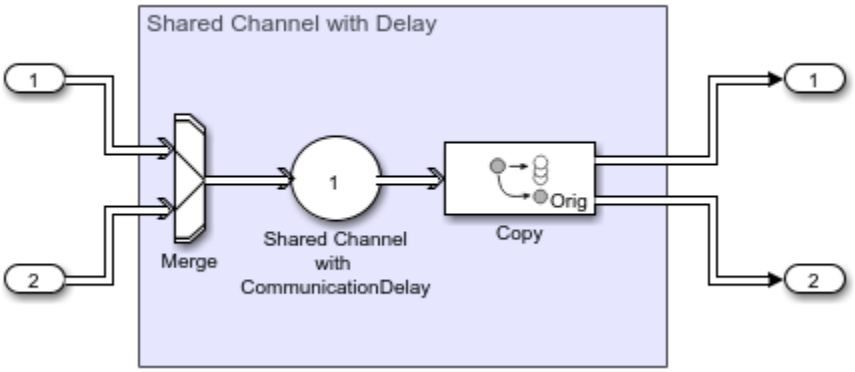
In the Listener component, there is a Simulink Function block. The block displays the function, onOneMessage(data), on the block face.



When a message arrives at Receive Buffer 2, the Listener block is notified and it takes the argument `data`, which is the value from the message data, as the input signal. In the block, `data` values are multiplied by 2. The block outputs the new data value.

Routing Messages using SimEvents®

In the shared channel, the message paths originating from the two message-sending components are merged to represent a shared communication channel.



A SimEvents® Entity Input Switch block merges the message lines. In the block:

- **Number of input ports** specifies the number of message lines to be merged. The parameter value is 2 for two message paths.
- **Active port selection** specifies how to select the active port for message departure. If you select `All`, all of the messages arriving at the block are able to depart the block from the output port. If you select `Switch`, you can specify the logic that selects the active port for message departure. For this example, the parameter is set to `All`.

A SimEvents® Entity Server block is used to represent message transmission delay in the shared channel. In the block:

- **Capacity** is set to `1`, which specifies how many messages can be processed at a time.

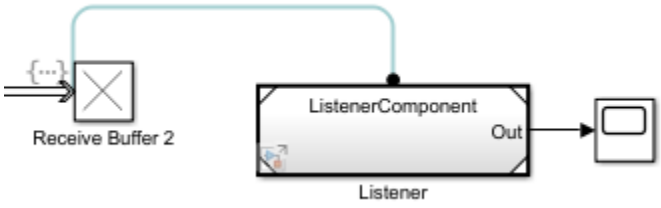
- **Service time value** is set to `1`, which specifies how long it takes to process a message

A SimEvents® Entity Replicator block is used to generate identical copies of messages. In the block:

- **Replicas depart from** specifies if the copies leave the block from separate output ports or the same output port as the original messages. The parameter is set to `Separate output ports`.
- **Number of replicas** is set to `1`, which specifies the number of copies generated for each message.
- **Hold original entity until all replicas depart** holds the original message in the block until all of its copies depart the block.

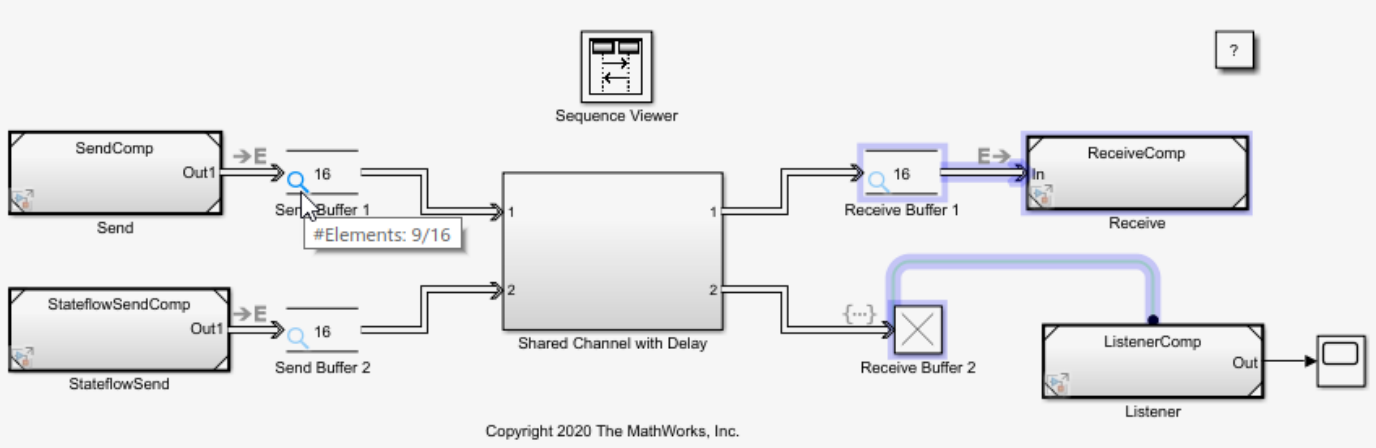
A SimEvents® Entity Terminator block is used to model Receive Buffer 2. In the block:

- Under the **Event actions** tab, in the **Entry action** field, you can specify MATLAB code that performs calculations or Simulink® function calls that are invoked when the message enters the block. In this example, `onOneMessage(entity)` is used to notify the Simulink Function block in the Listener component. To visualize the function call, under **Debug** tab, select **Information Overlays** and then **Function Connectors**.



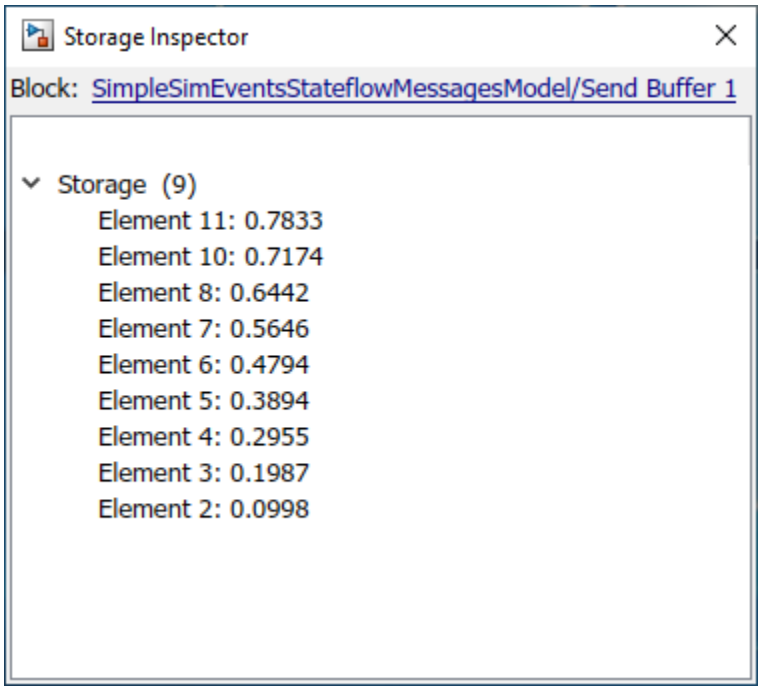
Simulate the Model and Review Results

Simulate the model. Observe that the animation highlights the messages flowing through the model. You can turn off the animation by right-clicking on the model canvas and setting **Animation Speed** to `None`.



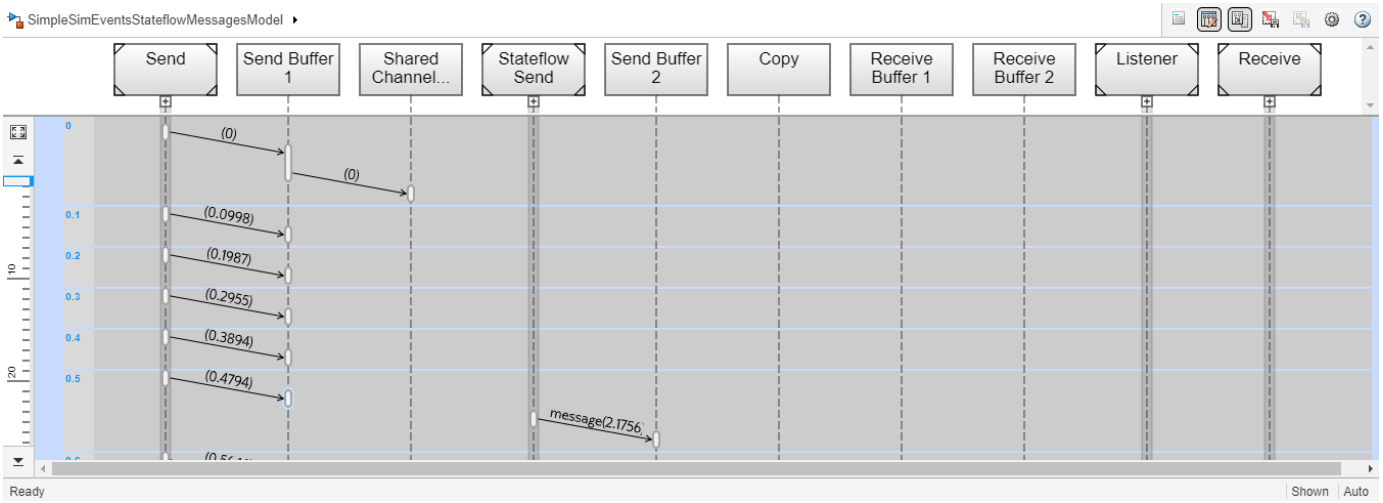
When you pause the animation, a magnifying glass appears on the blocks that store messages. If you point to the magnifying glass, you see the number of messages stored in the block.

To observe which messages are stored in the block, click the magnifying glass to open the Storage Inspector. For instance, the graphic below illustrates the messages stored in Send Buffer 1.



Turn the animation off and open the Sequence Viewer block to observe the Simulink Function calls and the flow of messages in the model.

For instance, observe the simulation time `0`, during which a message carrying value `0` is sent from the Send component to Send Buffer 1. From simulation time `0.1` to `0.5`, the Send component keeps sending messages to Send Buffer 1 with different data values. At time `0.5`, the StateflowSend component sends a message to Send Buffer 2. For more information about using the Sequence Viewer block, see “Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” on page 11-58.



See Also

Entity Input Switch (SimEvents) | Entity Server (SimEvents) | Entity Replicator (SimEvents) | Entity Terminator (SimEvents) | Queue | Receive | Send | Sine Wave

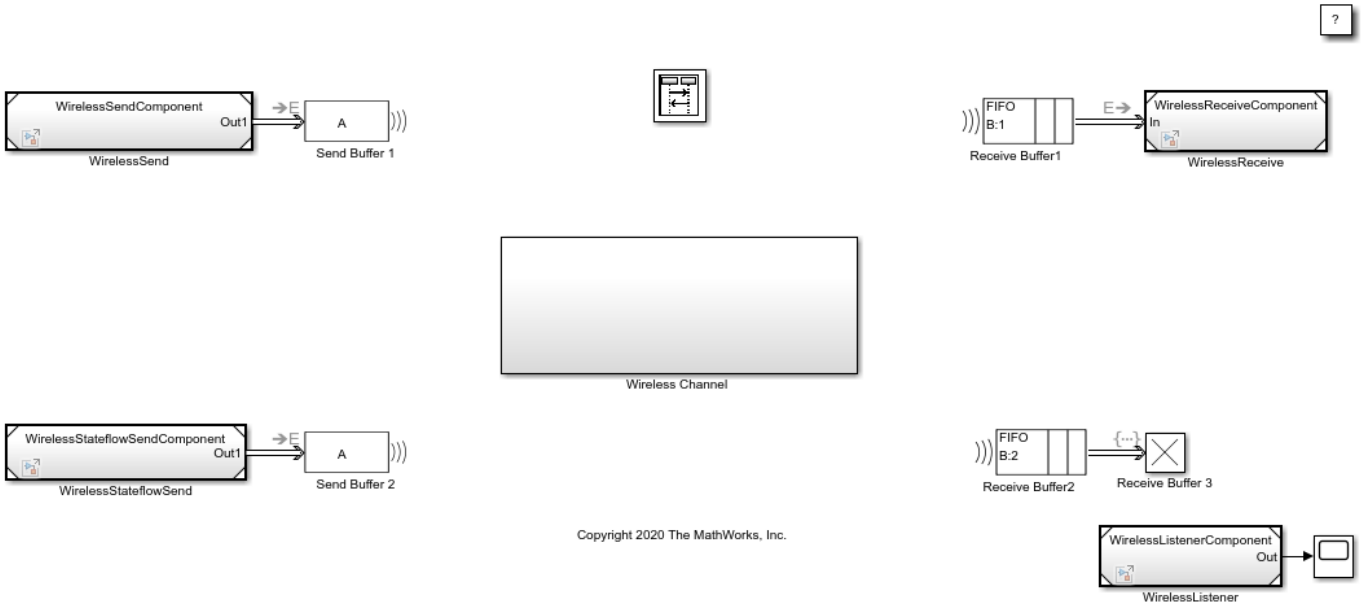
More About

- “Simulink Messages Overview” on page 11-2
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Model Wireless Message Communication with Packet Loss and Channel Failure” on page 11-35
- “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 11-45

Model Wireless Message Communication with Packet Loss and Channel Failure

This example shows how to model wireless message communication with packet loss and channel failure by using Simulink® messages, Stateflow®, and SimEvents®.

In this model, there are two components that send messages and two components that receive messages. The messages are transmitted using a shared wireless channel with a transmission delay. A Stateflow® chart models message-sending logic in a wireless component and SimEvents® blocks model wireless message transmission, channel failure, and packet loss.



For an overview about messages, see “Simulink Messages Overview” on page 11-2.

Create Components to Send and Receive Messages

In the model, there are two software components that output messages, WirelessSend and WirelessStateflowSend.

In the WirelessSend component, the Sine Wave block is the signal source. The Sine Wave block generates a sine wave with an amplitude of `1`. The block **Sample time** is set to `0.1`. The Send block converts the signal to a message that carries the data of the signal value. The WirelessSendComponent is connected to Send Buffer 1.

In the WirelessStateflowSend component, another Sine Wave block generates a sine wave signal and a Noise block is used to inject noise into the signal. The Noise block outputs a signal whose values are generated from a Gaussian distribution with mean of `0` and variance of `1`. The Stateflow® chart represents a simple logic that filters a signal and decides whether to send messages. The StateflowSend component sends messages to Send Buffer 2.

In the model, there are two software components that receive messages, WirelessReceive and WirelessListener.

In the WirelessReceive component, a Receive block receives messages and converts message data to signal values. The component is connected to Receive Buffer 1.

In the WirelessListener component, there is a Simulink Function block that runs the `onOneMessage(data)` function. When a message arrives at Receive Buffer 3, the Simulink Function block takes the argument `data`, which is the value from message data, as the input signal. In the block, the `data` values are multiplied by `2`. The block outputs the new data value.

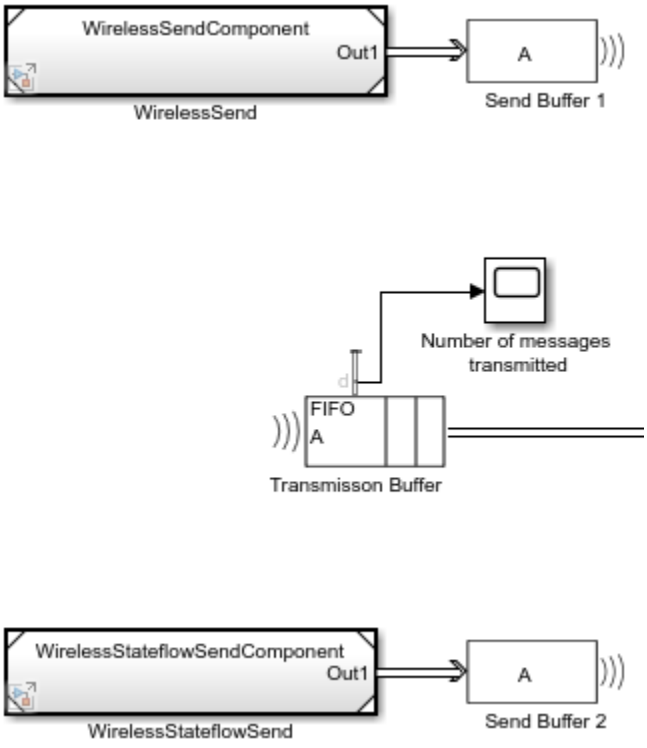
To learn more about creating these components, see “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29.

Model Wireless Message Communication Using Multicasting

The WirelessSend and WirelessStateflowSend components send messages to Send Buffer 1 and Send Buffer 2, which are SimEvents® Entity Multicast blocks that can wirelessly transmit messages. The Transmission Buffer block is a SimEvents® multicast receive queue that can receive messages sent by Send Buffer 1 and Send Buffer 2.

To achieve this wireless communication between Send Buffer 1, Send Buffer 2, and the Transmission Buffer block that is inside the Wireless Channel block:

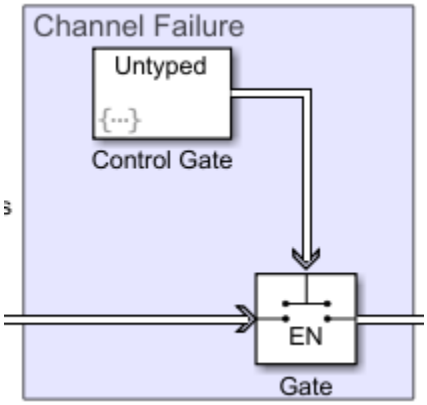
1. In the Send Buffer 1 and Send Buffer 2 blocks, set the **Multicast tag** parameter to `A`.
2. In the Transmission Buffer block, set the **Multicast tag** parameter to `A`.



The **Multicast tag** parameter defines from which Entity Multicast blocks the messages are received.

Model Channel Failure

A SimEvents® Entity Gate block is used to model channel failure. The block has two input ports. One input port is for incoming messages from Transmission Buffer. The second input port is a control port to decide when to open the gate.

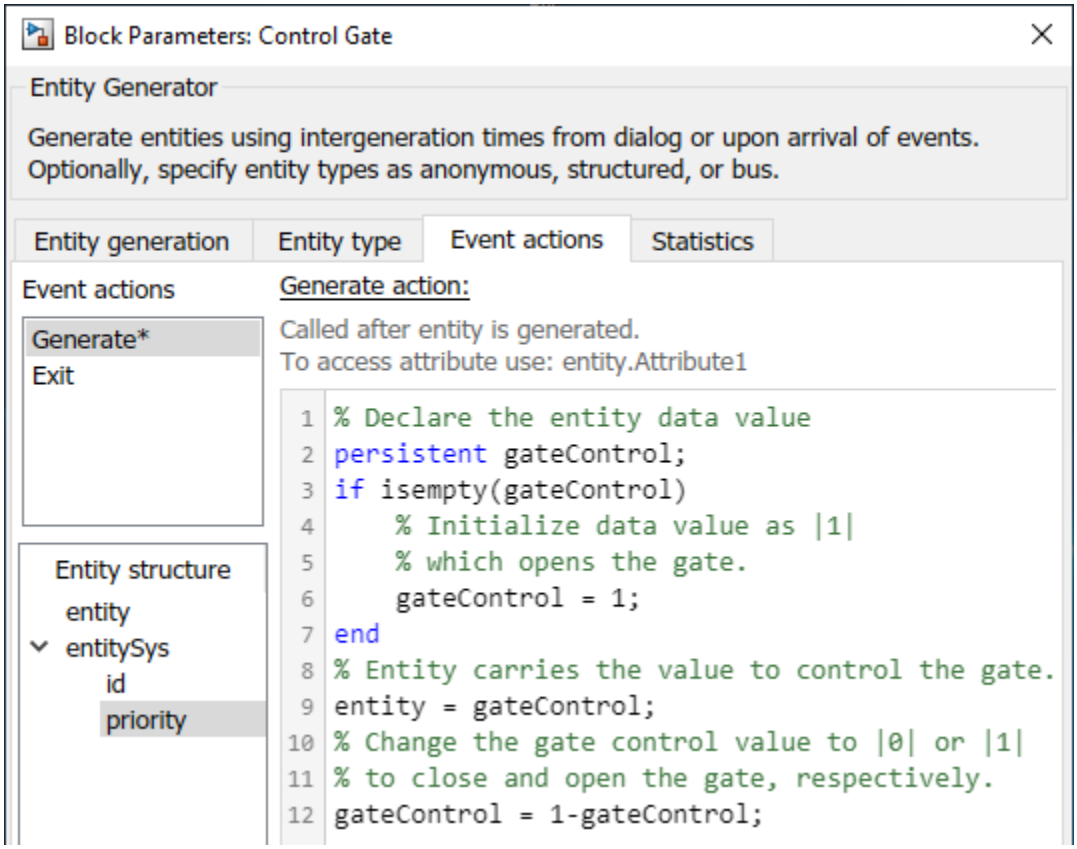


Set the Gate block's **Operating mode** parameter to `Enable gate`. In this mode:

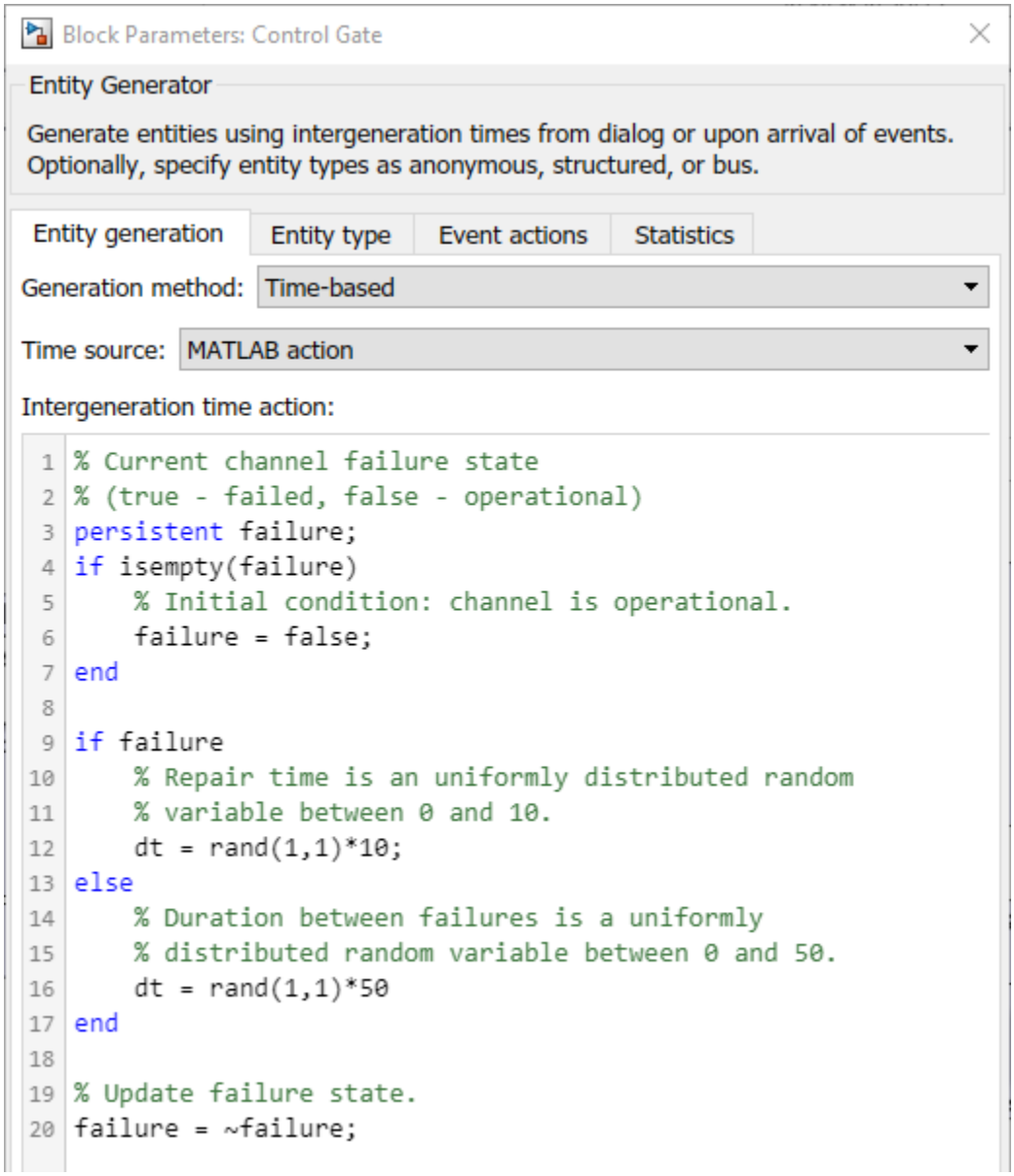
- The block opens the gate and permits messages to advance when it receives an entity carrying a value that is greater than `0` from its control port. This represents an operational channel.
- The block closes the gate and blocks messages passing if an entity carries data whose value is less than or equal to `0`. This represents a channel failure.

To control the Gate block, you can use the SimEvents® Entity Generator block, which is labeled Control Gate in this example, to generate entities carrying different data values.

In the Control Gate block, in **Event actions**, in the **Generate action** field, the code below is used to generate entities to open and close the Gate block. Initially, entity data is `1` and the gate is open and the channel is in operational state. When a new entity is generated, its value changes to `0`, which closes the gate. Every generated entity changes the gate's status, from open to closed or from closed to open.



In the Control Gate block, in the **Intergeneration time action** field, the code below is used to represent the operational and failed state of the channel. The code initializes the channel as operational. `dt` is the entity intergeneration time and is used to change the status of channel because each generated entity changes the status of the Gate block.

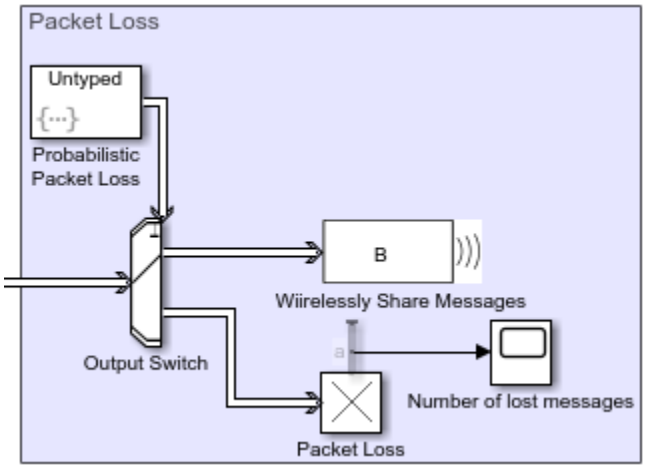


In the code, the repair time is generated from a uniform distribution that takes values between `0` and `10`. The time interval between the failures is generated from another uniform distribution that takes values between `0` and `50`.

Model Packet Loss

To model the packet loss, a SimEvents® Entity Output Switch block is used.

The block has two input ports. One input port accepts messages. The other input port accepts entities that determine the output port selection. If the entity is set to `1`, the block selects output port `1` to forward messages to the Wirelessly Share Messages block. If the entity is set to `2`, the block selects output port `2`, which is connected to an Entity Terminator block that represents packet loss.



In the Output Switch block:

- The **Number of output ports** is set to 2.
- To determine which output is selected, the **Switching criterion** is set to `From control port` and **Initial port selection** is set to 1.

To model a `0.1` probability of packet loss, in the Probabilistic Packet Loss block, select the **Event actions** tab, and in the **Generate action** field includes this code:

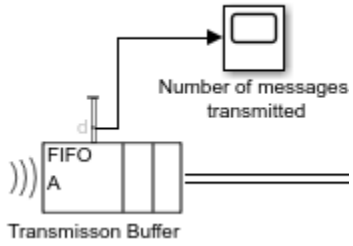
```
persistent rngInit;
if isempty(rngInit)
    seed = 12345;
    rng(seed);
    rngInit = true;
end

% Pattern: Uniform distribution
% m: Minimum, M: Maximum
m = 0; M = 1;
x = m + (M - m) * rand;

% x is generated from uniform distribution and
% takes values between |0| and |1|.
if x > 0.1
   % Entity carries data |1| and this forces Output switch to select
   % output |1| to forward entities to receive components.
   entity  = 1;
else
   % Entity carries data |2| and this forces Output switch to select
   % output |2| and this represents a packet loss.
    entity = 1;
end
```

This means that entities entering the control port have a `0.9` probability of being set to `1`, which makes the block output messages to the Wirelessly Share Messages block.

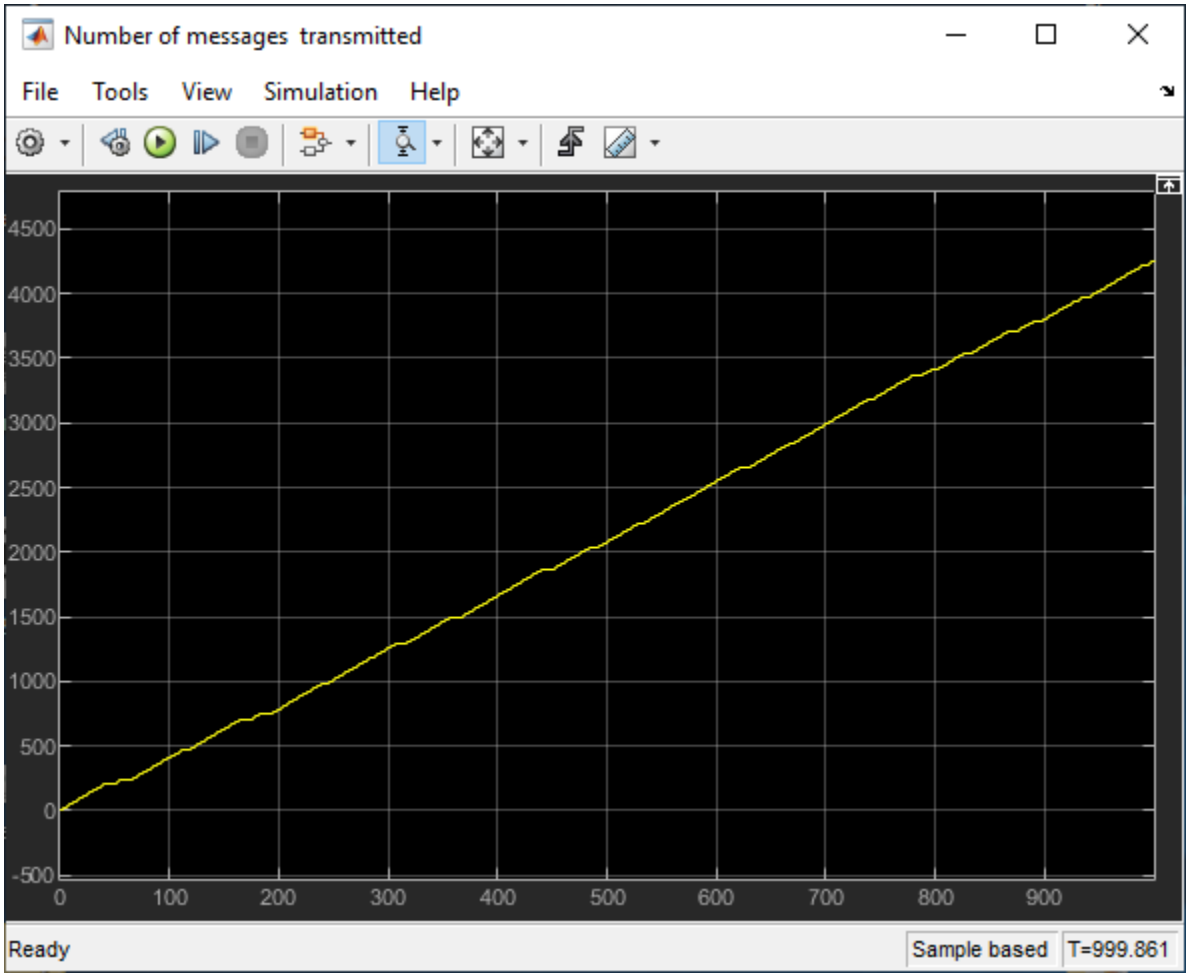
Simulate the Model and Review results

Simulate the model.

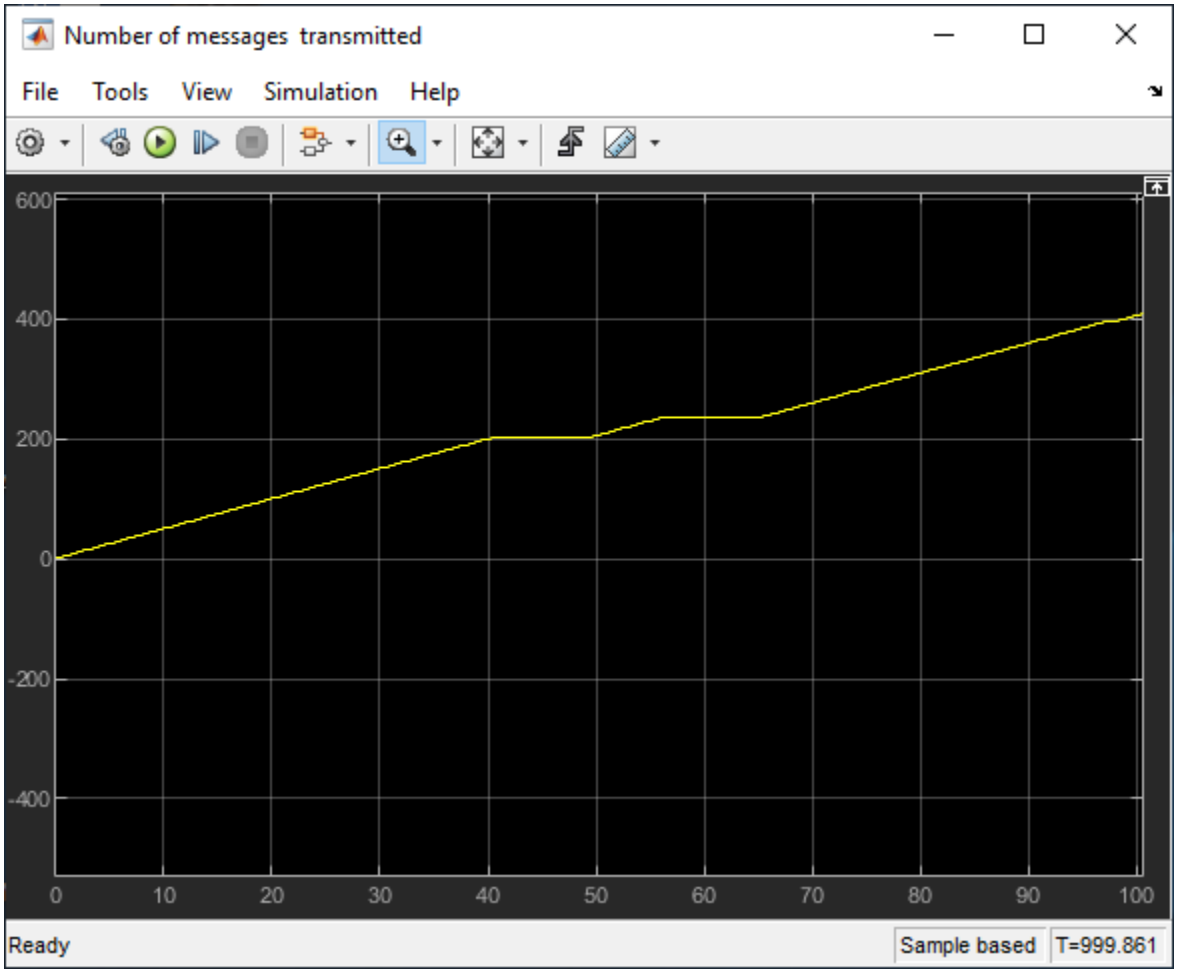
- Open the Scope block connected top the Transmission Buffer block. The block displays the total number of messages transmitted through the shared channel.



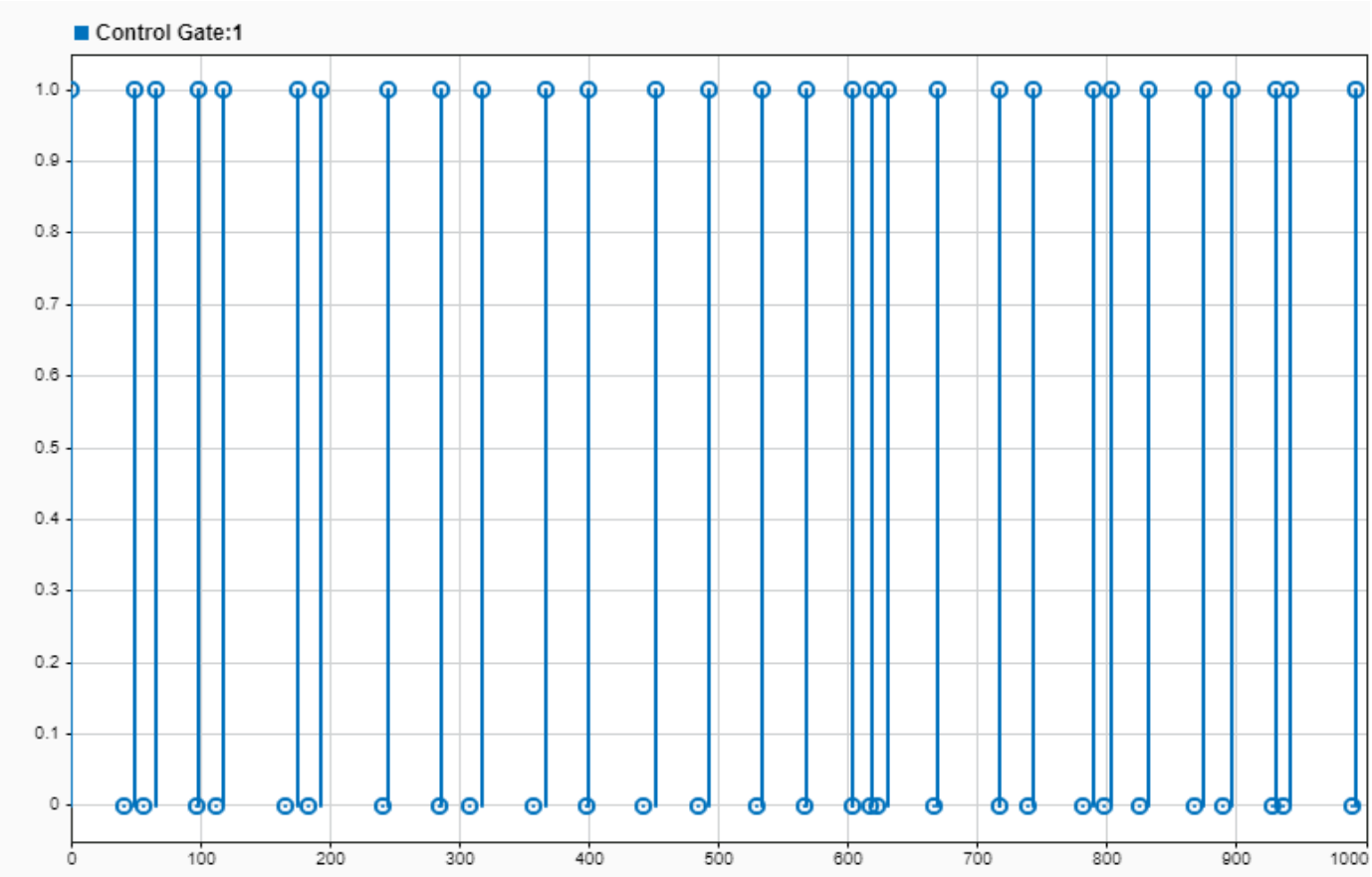
4255 messages are transmitted through the channel.



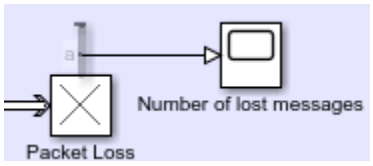
The plot also displays the channel failures. For example, zoom into the first `100` seconds. Observer that the channel failure occurs between `40` and `49` during which message transmission is blocked.



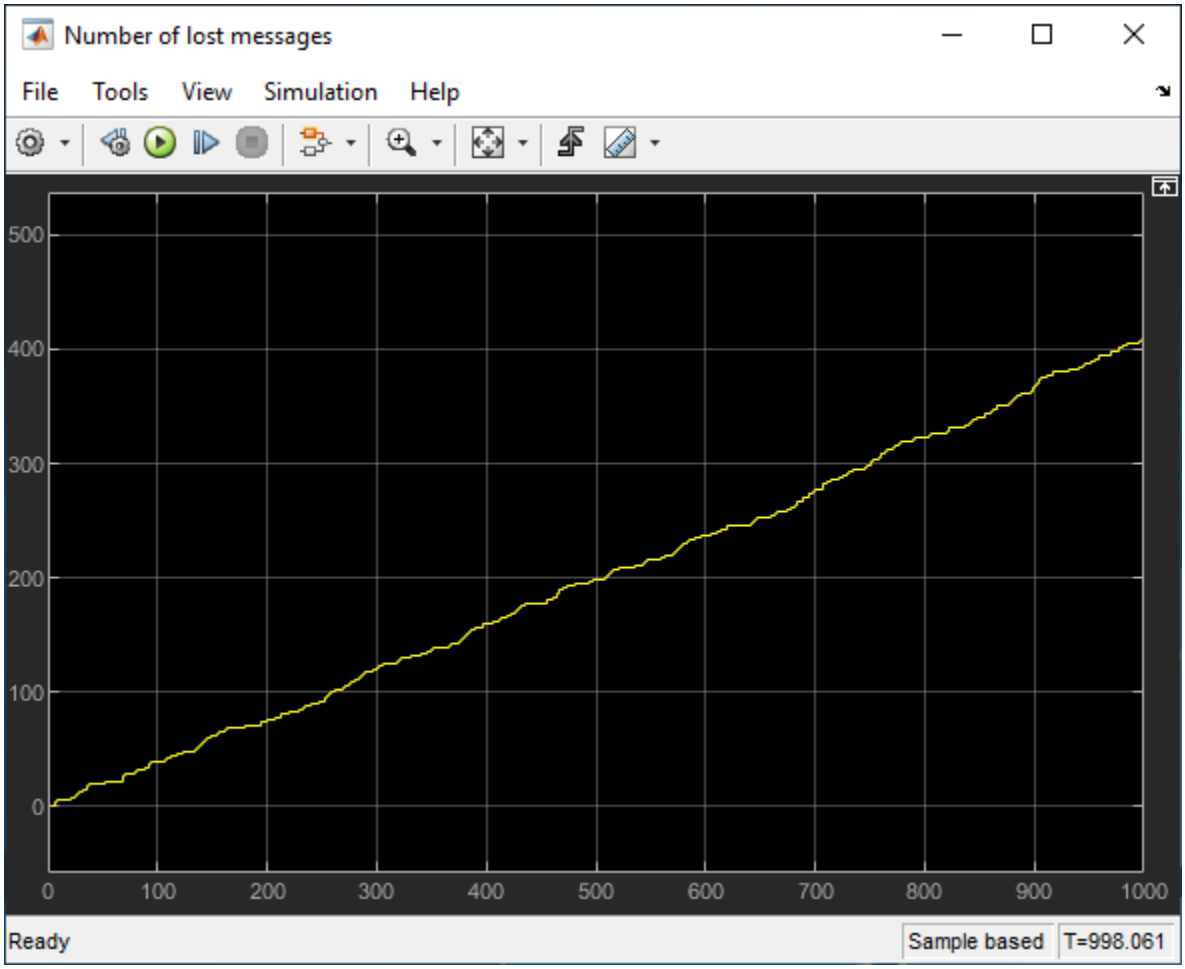
Open the Data Inspector to visualize the entities that control the Gate. Entity data changes from `1` to `0` for each generated entity.



To see the number of lost messages, open the Scope block connected to the Packet Loss block.



`409` messages are lost during transmission. This is `9.6` percent of the messages.



See Also

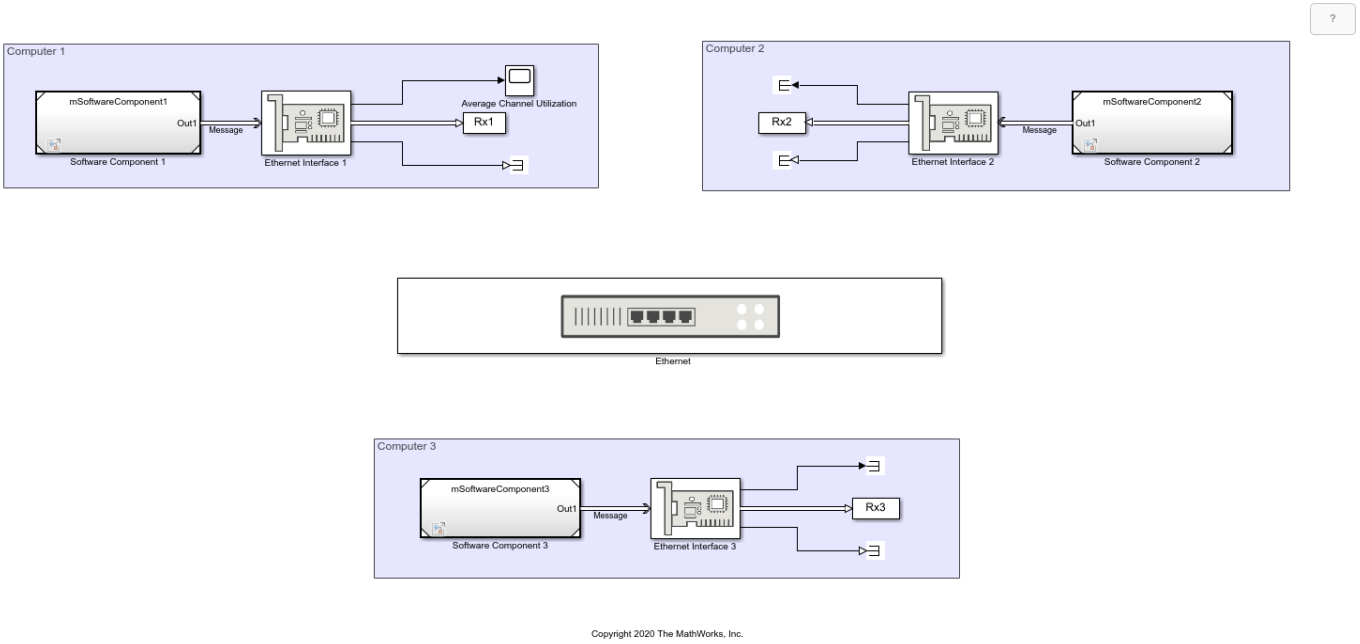
Entity Terminator (SimEvents) | Entity Output Switch (SimEvents) | Entity Gate (SimEvents) | Entity Multicast (SimEvents) | Queue | Receive | Send | Sine Wave

More About

- “Simulink Messages Overview” on page 11-2
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29
- “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 11-45

Model an Ethernet Communication Network with CSMA/CD Protocol

This example shows how to model an Ethernet communication network with CSMA/CD protocol using Simulink® messages and SimEvents®. In the example, there are three computers that communicate through an Ethernet communication network. Each computer has a software component that generates data and an Ethernet interface for communication. Each computer attempts to send the data to another computer with a unique MAC address. An Ethernet interface controls a computer's interaction with the network by using a CSMA/CD communication protocol. The protocol is used to respond to collisions that occur when multiple computers send data simultaneously. The Ethernet component represents the network and the connection between the computers.

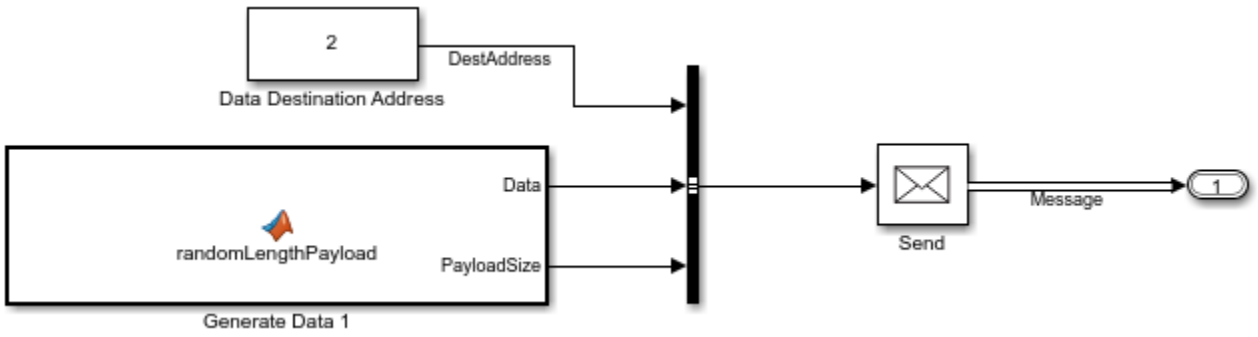


Software Components

In the model, each software component generates data (payload) and combines the data, its size, and its destination into a message. Then, the message is sent to the Ethernet interface for communication.

In each Software Component subsystem:

- A MATLAB Function block generates data with a size between `46` and `1500` bytes [1].
- A Constant block assigns destination addresses to data.
- A Bus Creator block converts the `Data`, `PayloadSize`, and `DestAddress` signals to a nonvirtual bus object called `dataPacket`.
- A Send block converts `dataPacket` to a message.
- An Outport block sends the message to the Ethernet interface for communication.

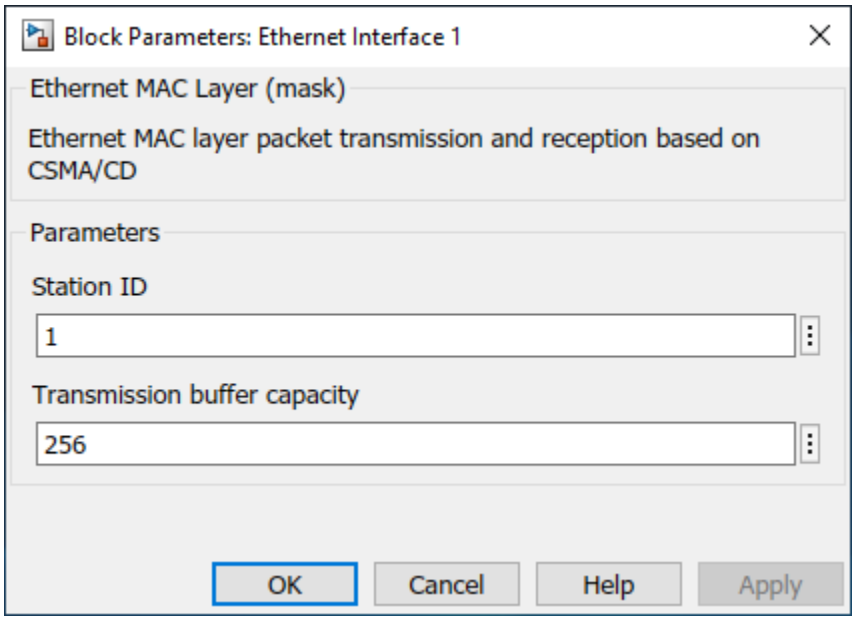


Each computer generates data with a different rate. You can change the data generation rate from the MATLAB Function block's sample time.

To learn the basics of creating message send and receive interfaces, see “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20.

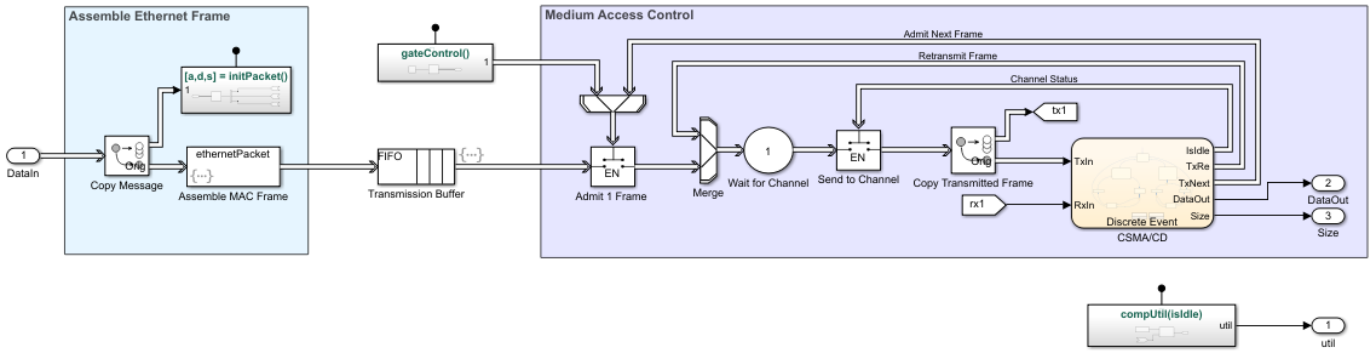
Ethernet Interface

Double-click Ethernet Interface 1. Observe that you can specify the **Station ID** and **Transmission buffer capacity**.



The Ethernet Interface subsystems have three main parts:

1. Assemble Ethernet Frame — Converts an incoming message to an Ethernet (MAC) frame.
2. Transmission Buffer — Stores Ethernet frames for transmission.
3. Medium Access Control — Implements a CSMA/CD protocol for packet transmission [2].

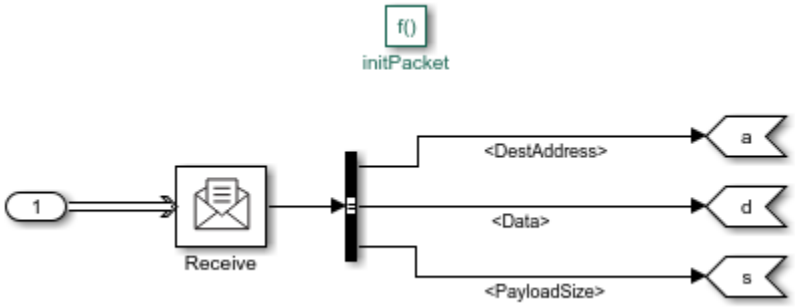


Assemble Ethernet Frame

The Assemble Ethernet Frame blocks convert messages to Ethernet frames by attaching Ethernet-specific attributes to the message [1].

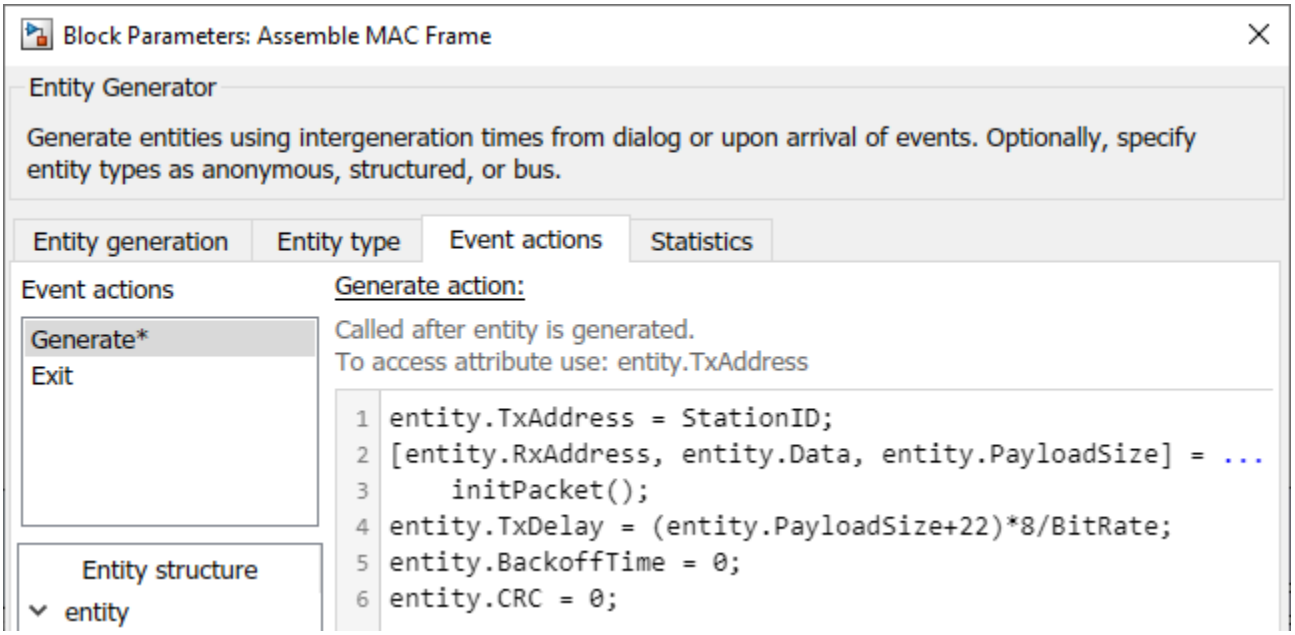
In the packet assembly process:

- A SimEvents® Entity Replicator block labeled Copy Message copies an incoming message. The original message is forwarded to a SimEvents® Entity Generator block labeled Assemble MAC Frame. Because the Entity Generator block **Generation method** parameter is set to `Event-based`, it immediately produces an entity when the original message arrives at the block. A copy of the message is forwarded to a Simulink Function block with the `initPacket()` function. The terms *message* and *entity* are used interchangeably between Simulink® and SimEvents®.
- The Simulink Function block transfers the data, its size, and its destination address to the Assemble MAC Frame block for frame assembly.



- The Assemble MAC Frame block generates the Ethernet frames that carry both Ethernet-specific attributes and values transferred from the Simulink Function block.

Assemble MAC Frame block calls the `initPacket()` function as an action that is invoked by each frame generation event.



These are the attributes of the generated Ethernet frame:

- `entity.TxAddress` is `StationID`.
- `entity.RxAddress`, `entity.Data`, and `entity.PayloadSize` are assigned the values from the Simulink Function block.
- `entity.TxDelay` is the transmission delay. It is defined by the payload size and the bitrate. The `Bitrate` parameter is specified by an initialization function in the Model Properties.
- `entity.CRC` is the cyclic redundancy check for error detection.

Transmission Buffer

The transmission buffer stores entities before transmission by using a first-in-first-out (FIFO) policy. The buffer is modeled by a Queue block.

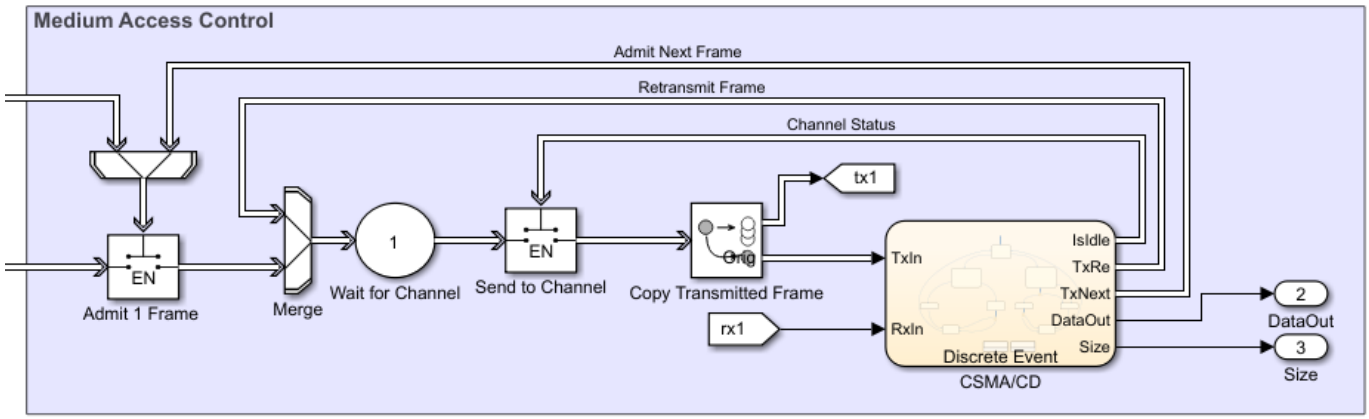
The capacity of the queue is determined by the **Transmission buffer capacity** parameter.

Medium Access Control

The Medium Access Control blocks are modeled by using six SimEvents® blocks.

- An Entity Gate block labeled Admit 1 Frame is configured as an enabled gate with two input ports. One input port allows frames from the Transmission Buffer block. The other input port is called the control port, which accepts messages from the CSMA/CD block. The block allows one frame to advance when it receives a message with a positive value from CSMA/CD block.
- An Entity Input Switch block labeled Merge merges two paths. One input port accepts new frames admitted by the Admit 1 frame block and the other input port accepts frames for retransmission that are sent by the CSMA/CD block.
- An Entity Server block labeled Wait for Channel models the back off time of a frame before its retransmission through the channel.

- Another Entity Gate block labeled Send to Channel opens the gate to accept frames when the channel is idle. The channel status is communicated by the CSMA/CD chart.
- An Entity Replicator block labeled Copy Transmitted Frame generates a copy of the frame. One frame is forwarded to the Ethernet network, and the other is forwarded to the CSMA/CD chart.
- A Discrete-Event Chart block labeled CSMA/CD represents the state machine that models the CSMA/CD protocol.



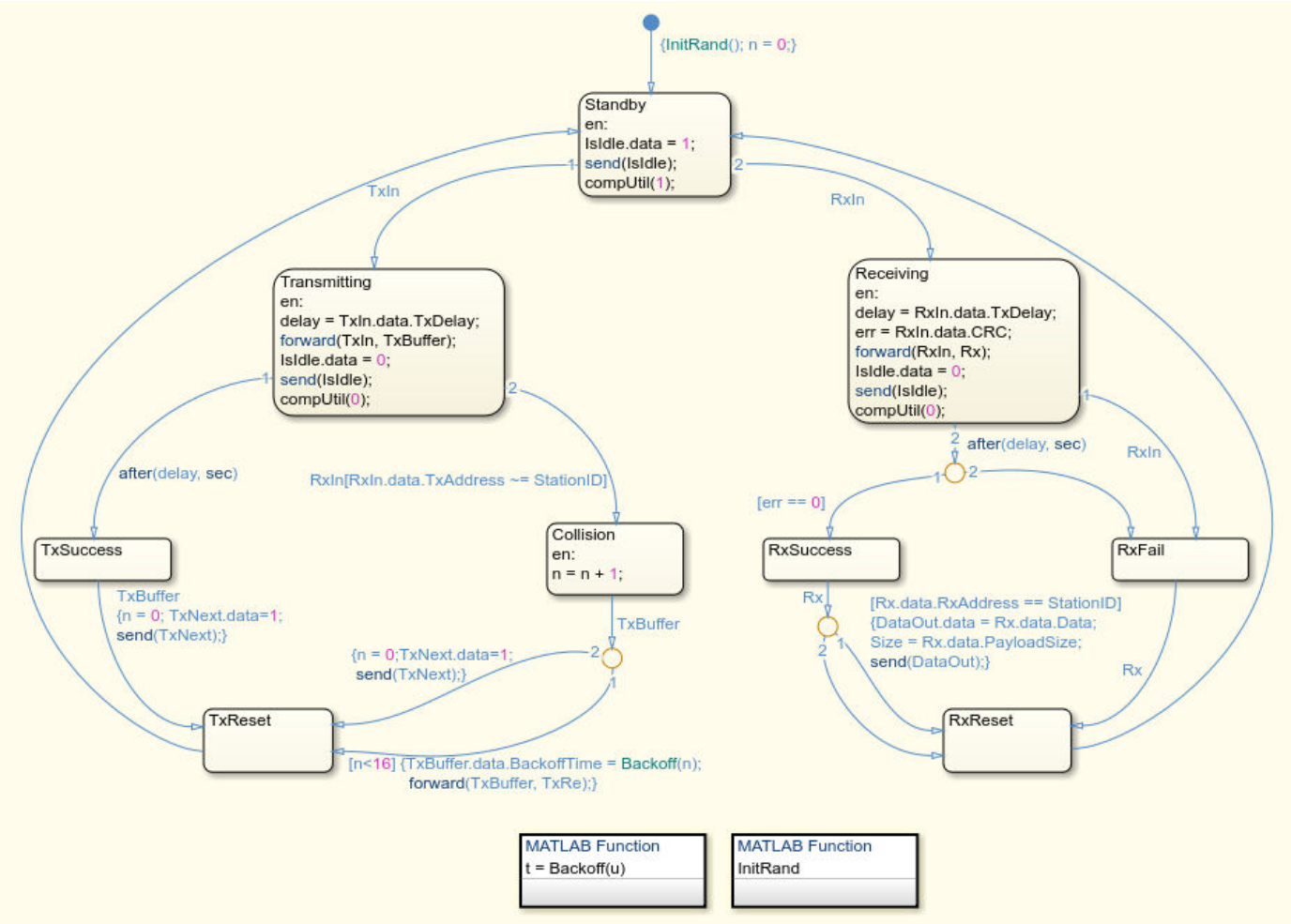
CSMA/CD Protocol

The CSMA/CD protocol [2] is modeled by a Discrete-Event Chart block that has two inputs:

- `TxIn` — Copy of the transmitted frame.
- `RxIn` — Received frame from the Ethernet network.

The chart has five outputs:

- `IsIdle` — Opens the Send to Channel gate to accept frames when the value is `1`, and closes the gate when the value is `0`.
- `TxRe` — Retransmitted frame that is forwarded to the Merge block if there is a collision detected during its transmission.
- `TxNext` — Opens the Admit 1 Frame gate to accept new frames when the value is `1`.
- `DataOut` — Received data.
- `Size` — Size of the received data.



Transmitting and Receiving Messages

The block is initially in the `Standby` state and the channel is idle.

If the block is transmitting, after a delay, the block attempts to transmit the message and `Isle.data` is set to `0` to declare that the channel is in use.

If the transmission is successful, the block sets `TxNext.data` to `1` to allow a new message into the channel and resets to the `Standby` state.

If there is a collision, the block resends the message after delaying it for a random back off time. *n* is the counter for retransmissions. The block retransmits a message a maximum of `16` times. If all of the retransmission attempts are unsuccessful, then the block terminates the message and allows the entry of a new message. Then it resets to `StandBy`.

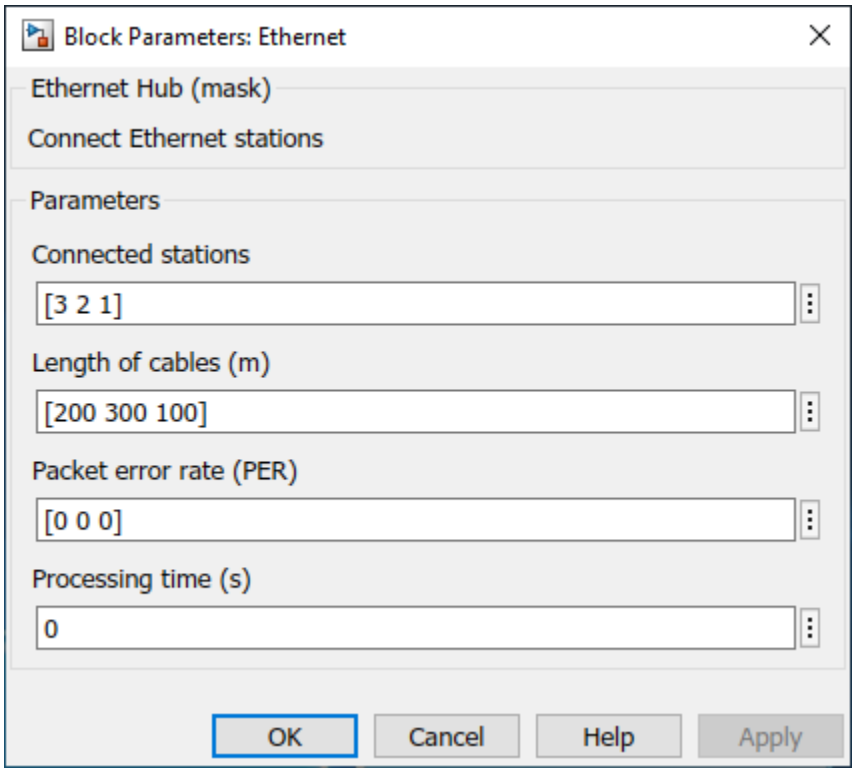
Similarly, the block can receive messages from other computers. If there is no error, the messages are successfully received and the block outputs the received data and its size.

Ethernet Hub

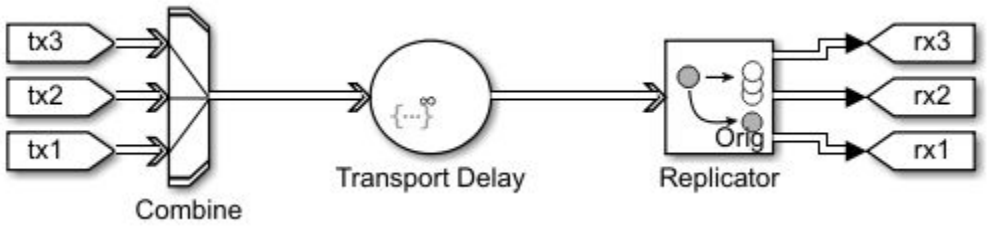
The Ethernet component represents the communication network and the cabled connections of the computers to the network.

Double-click the Ethernet block to see its parameters.

- **Connected stations** — These values are assigned to `Stations`, which is a vector with the station IDs as elements.
- **Length of cables (m)** — These values are assigned to `CableLength` and represent the length of the cables, in meters, for each computer connected to the hub.
- **Packet error rate (PER)** — These values are assigned to `PER` and represent the rate of error in message transmission for each computer.
- **Processing time (s)** — These values are assigned to `ProcessingTime` and it represents the channel transmission delay.

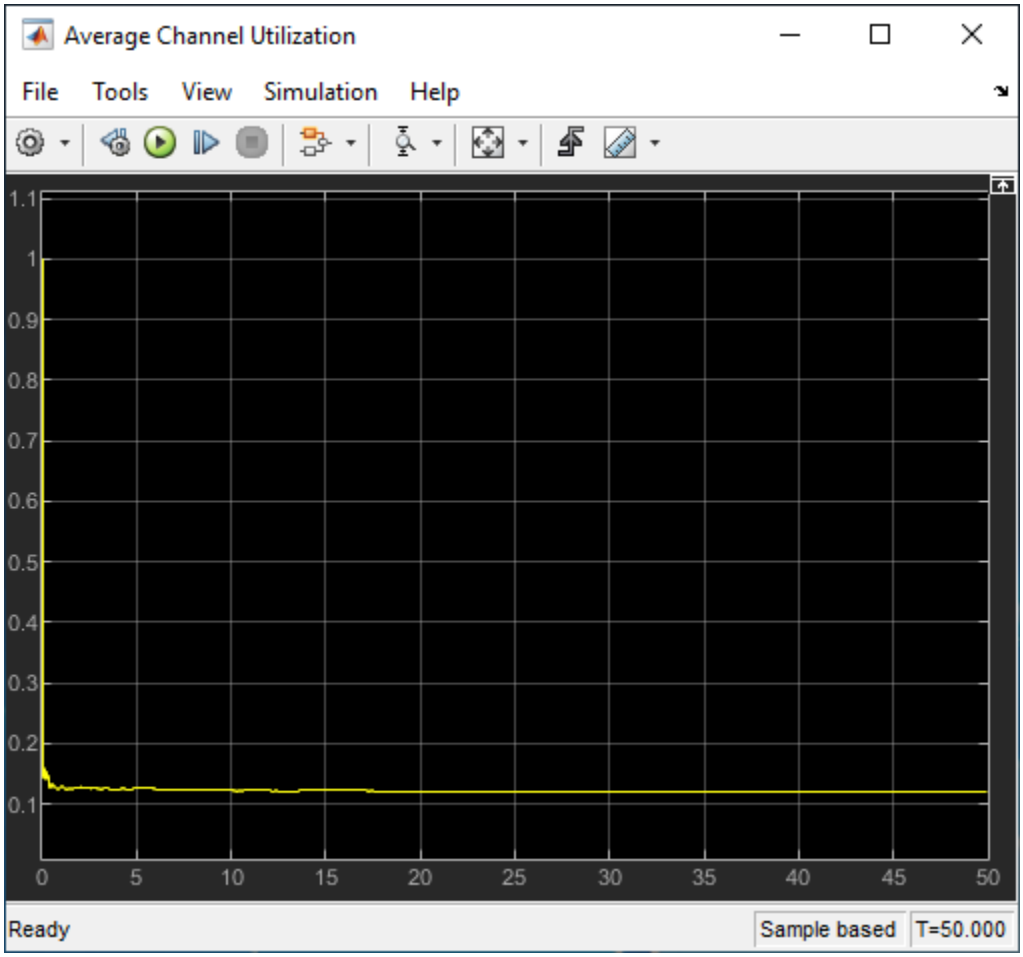


Three SimEvents® blocks are used to model the Ethernet network. The three computer connections are merged by using an Entity Input Switch block. An Entity Server block is used to model the channel transmission delay based on the cable length. An Entity Replicator block copies the transmitted message and forwards it to the three computers.

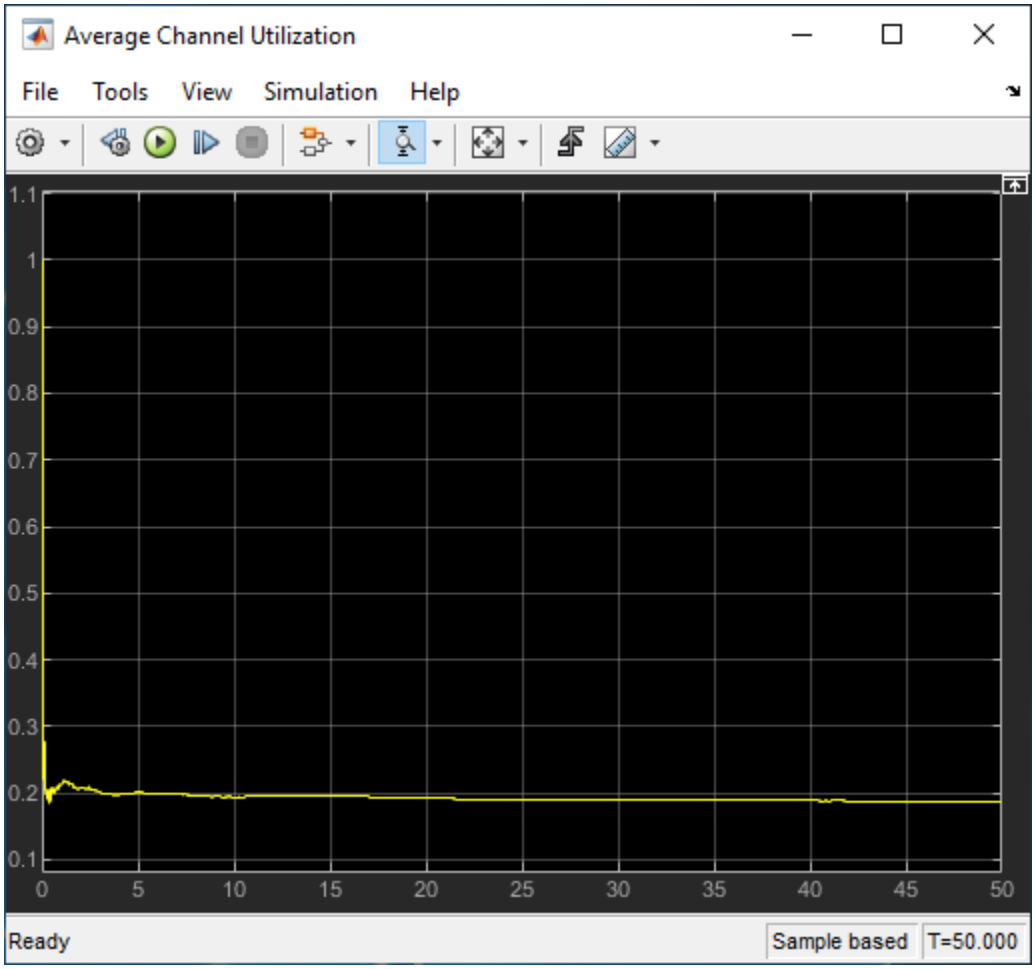


Simulate the Model and Review the Results

Simulate the model and open the Scope block that displays the average channel utilization. The channel utilization converges to approximately `0.12`.



Open Software Component 1 as a top model and change the data generation rate by setting the **Sample time** of the Generate Data 1 block to `0.01`. Run the simulation again and observe that the channel utilization increases to `0.2`.

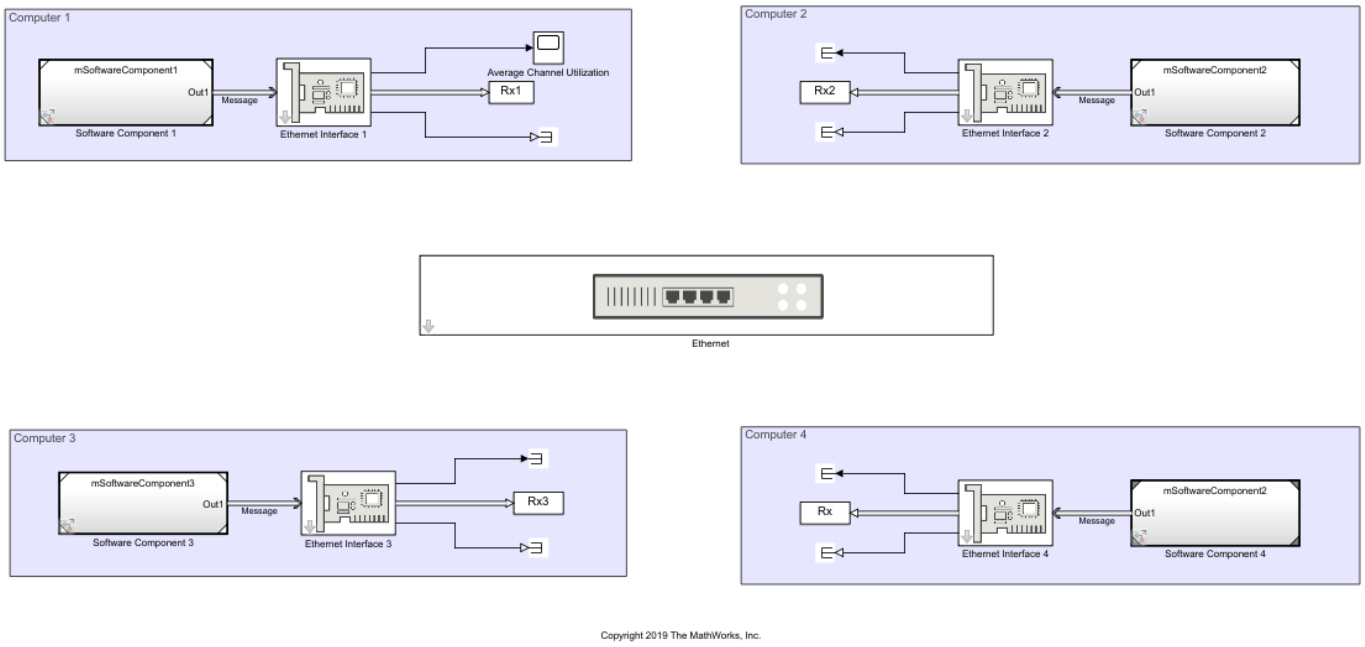


Connect New Computers to the Network

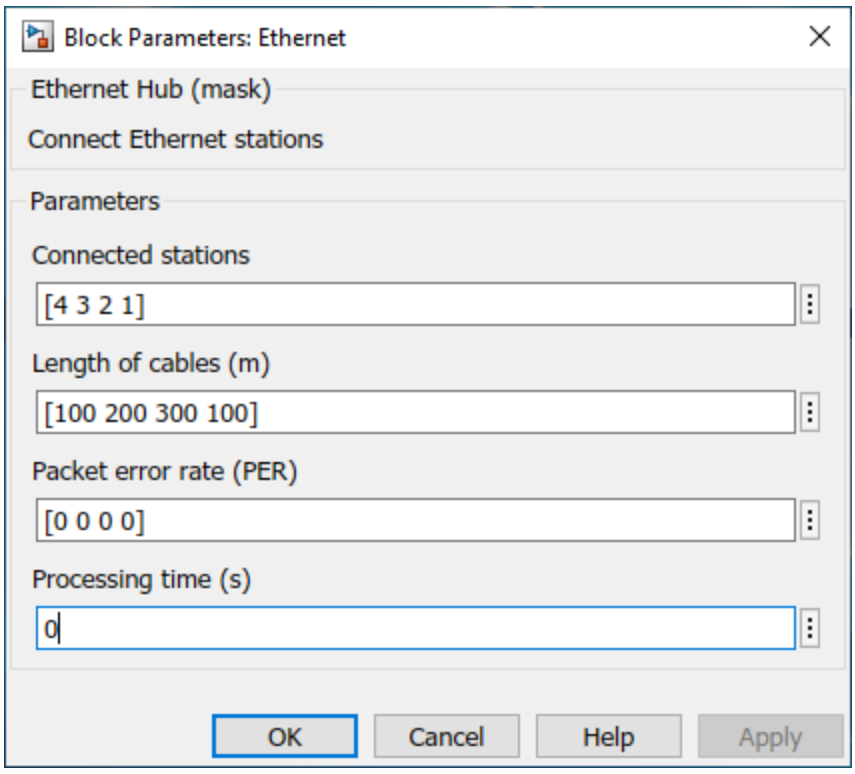
You can connect more computers to the network.

To add a new computer to the network:

- Copy an existing computer and assign a new ID by double-clicking the Ethernet Interface block. In this example, new computer has ID 4.



- Double-click the Ethernet block and add a station ID, cable length, and packet error rate for the new computer.



References

1 Ethernet frame - Wikipedia (https://en.wikipedia.org/wiki/Ethernet_frame)

2 Carrier-sense multiple access with collision detection - Wikipedia (https://en.wikipedia.org/wiki/Carrier-sense_multiple_access_with_collision_detection)

See Also

Entity Input Switch (SimEvents) | Entity Replicator (SimEvents) | Discrete-Event Chart (SimEvents) | Entity Generator (SimEvents) | Entity Gate (SimEvents) | Queue | Receive | Send

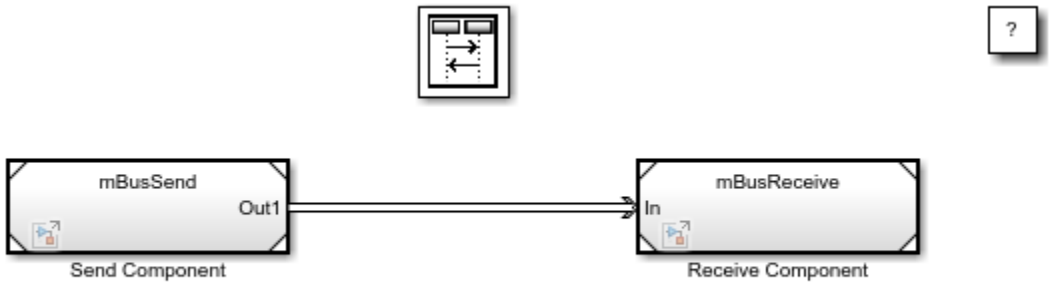
More About

- “Simulink Messages Overview” on page 11-2
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 11-29

Send and Receive Messages Carrying Bus Data

This example shows how to send and receive messages carrying bus data between model components.

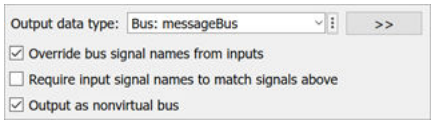
In the model, there are two components, Send Component and Receive Component that send and receive messages, respectively. This example builds on another example, “Establish Message Send and Receive Interfaces Between Software Components” on page 11-20, where two software components communicate using messages. The model is modified to send messages carrying bus data.



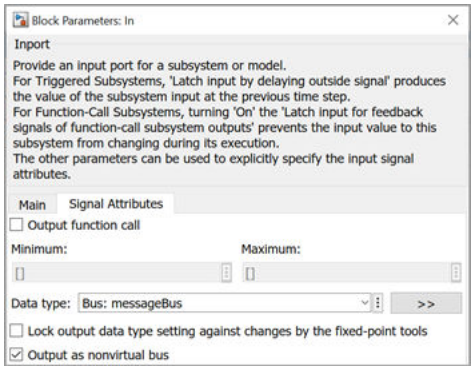
Prepare the Model for Messages with Bus Data

The following steps are used to prepare the model for messages carrying bus data type.

- To create messages carrying non-virtual bus, in the Send Component, in the Bus Creator block, set the **Output data type** to `messageBus` and select the **Output as nonvirtual bus** checkbox.



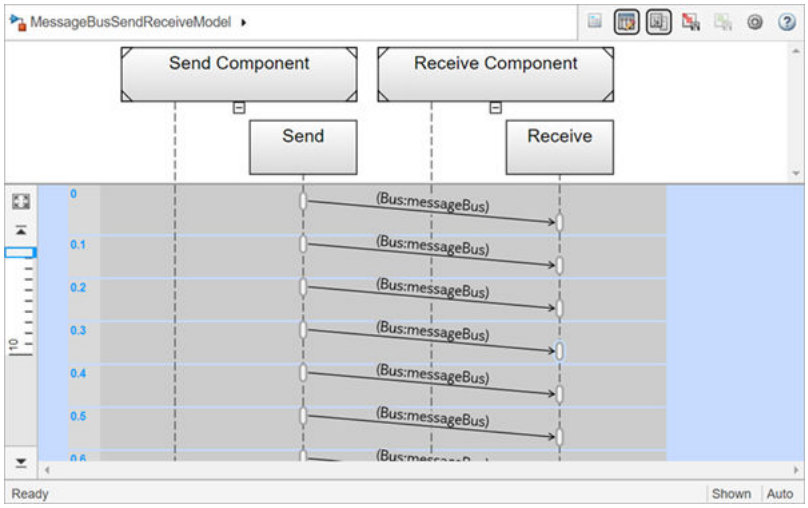
- To send messages carrying bus data, in the Send Component, in the Outport block, under the **Signal Attributes** tab, set the **Data type** to `messageBus` and select the select the **Output as nonvirtual bus** checkbox.



- To receive messages carrying bus data, in the Receive Component, in the Inport block, under the **Signal Attributes** tab, set the **Data type** to `messageBus` and select the select the **Output as nonvirtual bus** checkbox.

Simulate the Model and Visualize the Results

Simulate the model and open the Sequence Viewer block. Observe the transmission of messages carrying bus data.



Use the Sequence Viewer Block to Visualize Messages, Events, and Entities

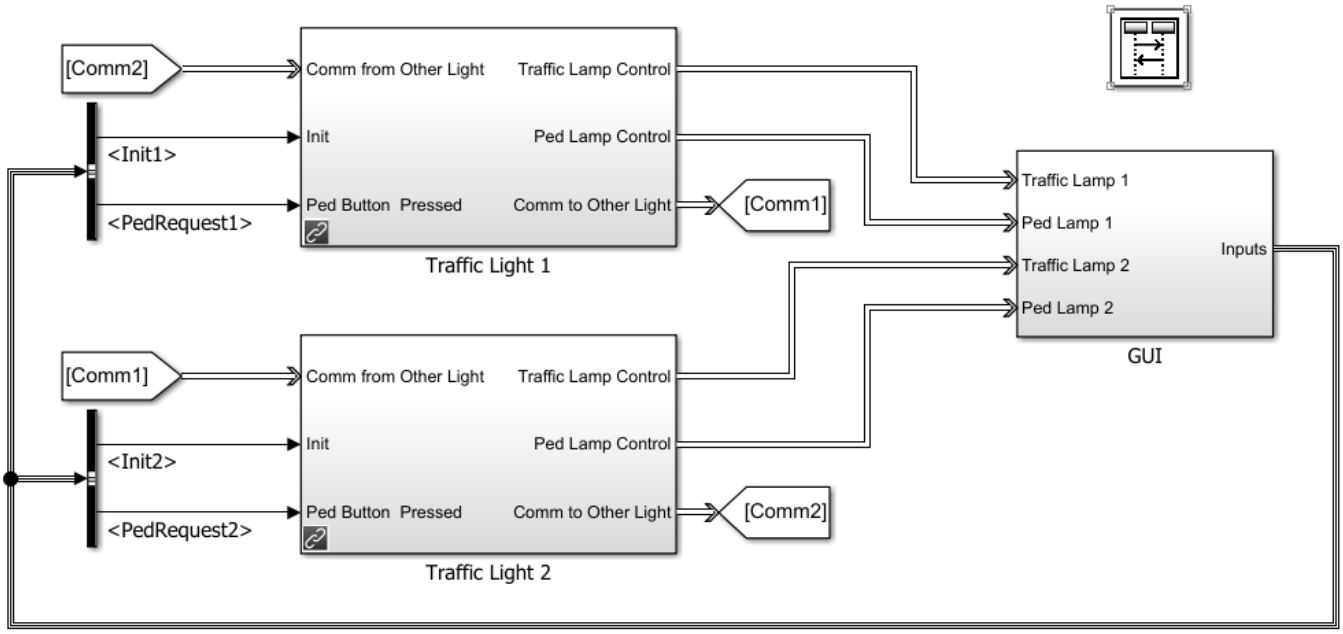
To see the interchange of messages and events between Stateflow charts in Simulink models and the movement of entities between SimEvents blocks, add a Sequence Viewer block to your Simulink model.

In the Sequence Viewer block, you can view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Sequence Viewer window shows messages as they are created, sent, forwarded, received, and destroyed at different times during model execution. The Sequence Viewer window also displays state activity, transitions, and function calls to Stateflow graphical functions, Simulink functions, and MATLAB functions.

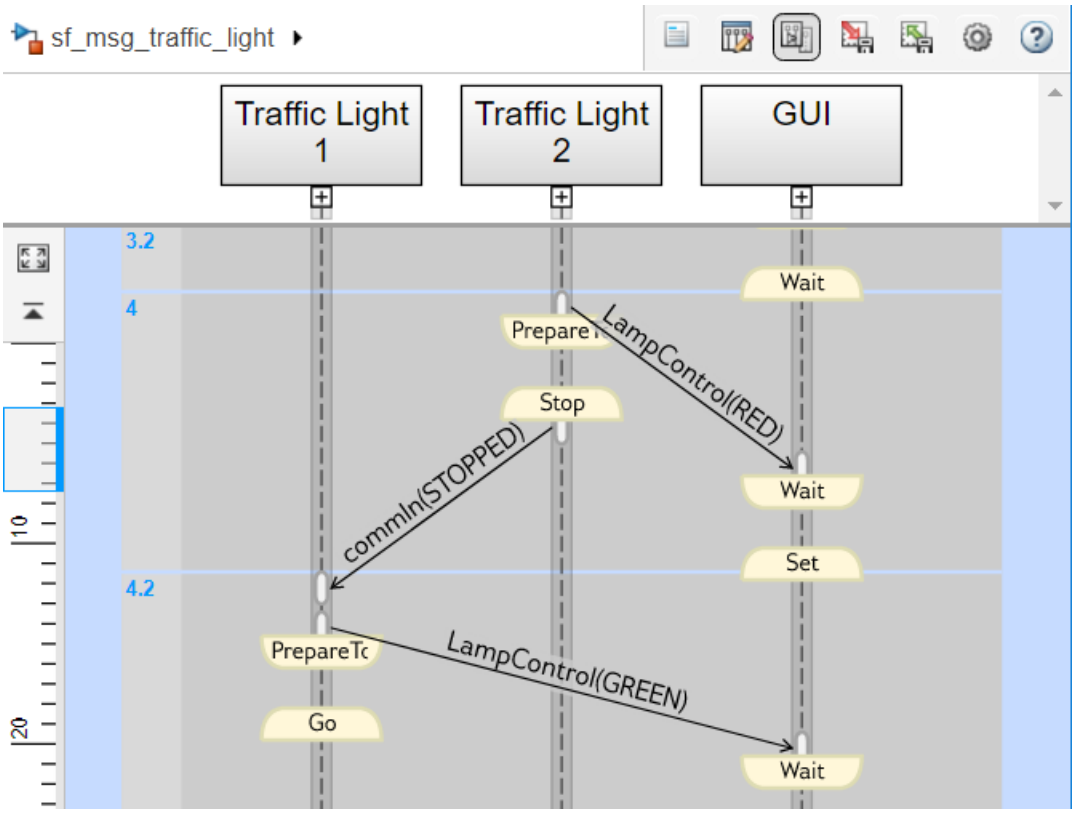
With the Sequence Viewer block, you can visualize the movement of entities between blocks when simulating SimEvents models. All SimEvents blocks that can store entities appear as lifelines in the Sequence Viewer window. Entities moving between these blocks appear as lines with arrows. You can view calls to Simulink Function blocks and to MATLAB Function blocks.

You can add a Sequence Viewer block to the top level of a model or any subsystem. If you place a Sequence Viewer block in a subsystem that does not have messages, events, or state activity, the Sequence Viewer window informs you that there is nothing to display.

For instance, suppose that you simulate the Stateflow example `sf_msg_traffic_light`.



This model has three Simulink subsystems: Traffic Light 1, Traffic Light 2, and GUI. The Stateflow charts in these subsystems exchange data by sending messages. As messages pass through the system, you can view them in the Sequence Viewer window. The Sequence Viewer window represents each block in the model as a vertical lifeline with simulation time progressing downward.



Components of the Sequence Viewer Window

Navigation Toolbar

At the top of the Sequence Viewer window, a navigation toolbar displays the model hierarchy path. Using the toolbar buttons, you can:

- Show or hide the Property Inspector.
- Select an automatic or manual layout.
- Show or hide inactive lifelines.
- Save Sequence Viewer block settings.
- Restore Sequence Viewer block settings.
- Configure Sequence Viewer block parameters.
- Access the Sequence Viewer block documentation.

Property Inspector

In the Property Inspector, you can choose filters to show or hide:

- Events

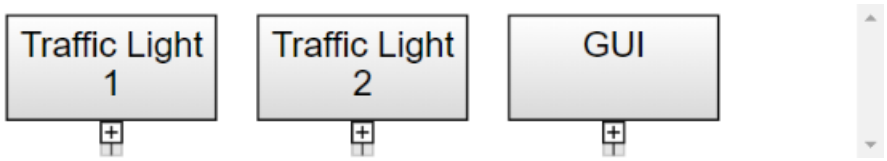
- Messages
- Function Calls
- State Changes and Transitions

Header Pane

The header pane below the Sequence Viewer toolbar shows lifeline headers containing the names of the corresponding blocks in a model.

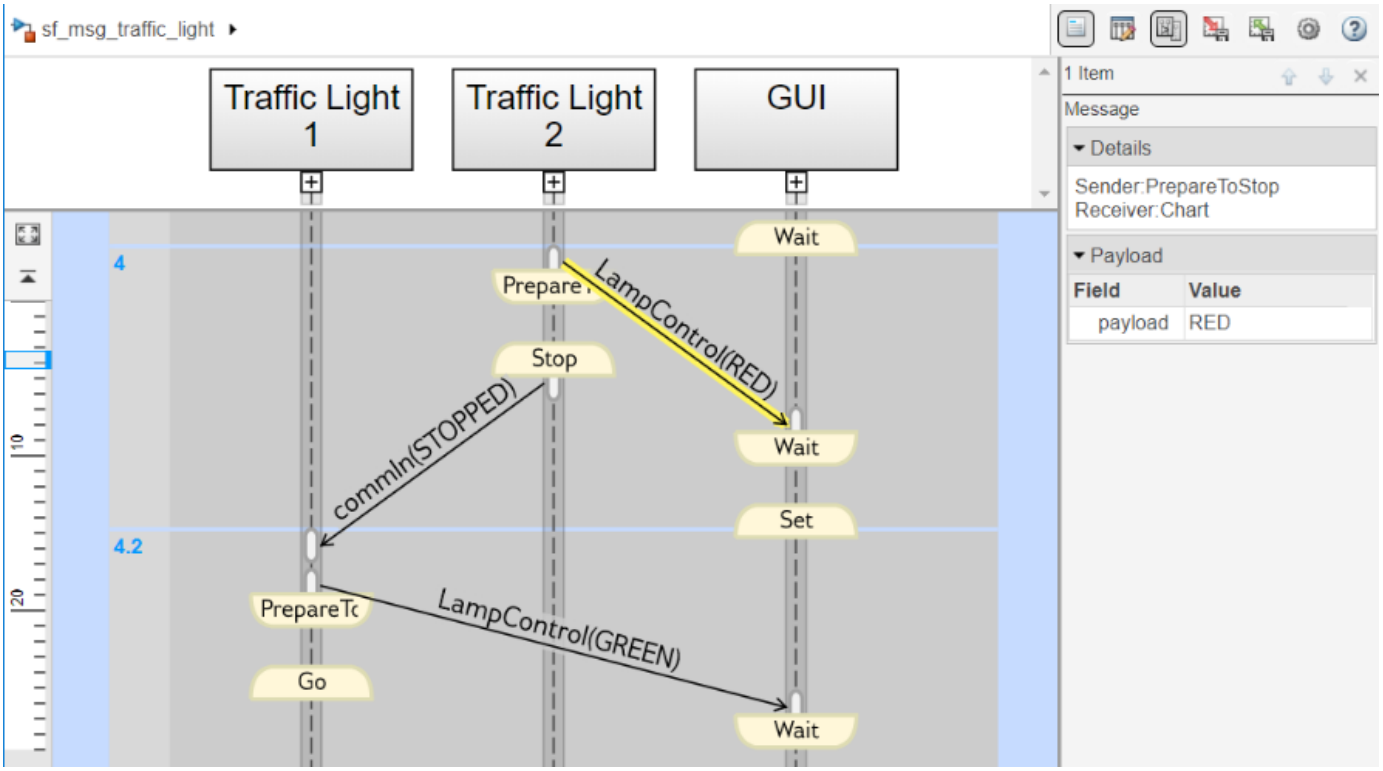
- Gray rectangular headers correspond to subsystems.
- White rectangular headers correspond to masked subsystems.
- Yellow headers with rounded corners correspond to Stateflow charts.

To open a block in the model, click the name in the corresponding lifeline header. To show or hide a lifeline, double-click the corresponding header. To resize a lifeline header, click and drag its right-hand side. To fit all lifeline headers in the Sequence Viewer window, press the space bar.



Message Pane

Below the header pane is the message pane. The message pane displays messages, events, and function calls between lifelines as arrows from the sender to the receiver. To display sender, receiver, and payload information in the Property Inspector, click the arrow corresponding to the message, event, or function call.



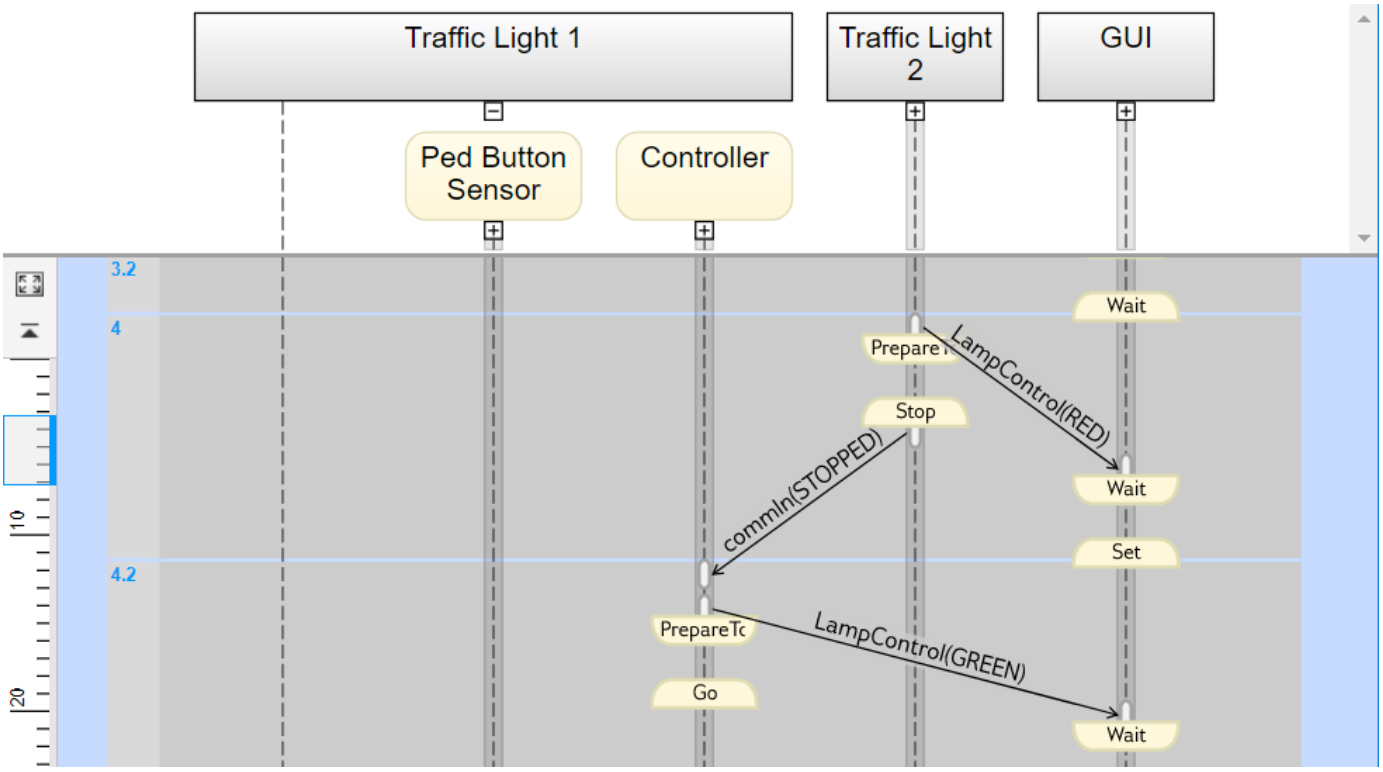
Navigate the Lifeline Hierarchy

In the Sequence Viewer window, the hierarchy of lifelines corresponds to the model hierarchy. When you pause or stop the model, you can expand or contract lifelines and change the root of focus for the viewer.

Expand a Parent Lifeline

In the message pane, a thick, gray lifeline indicates that you can expand the lifeline to see its children. To show the children of a lifeline, click the expander icon ⊞ below the header or double-click the parent lifeline.

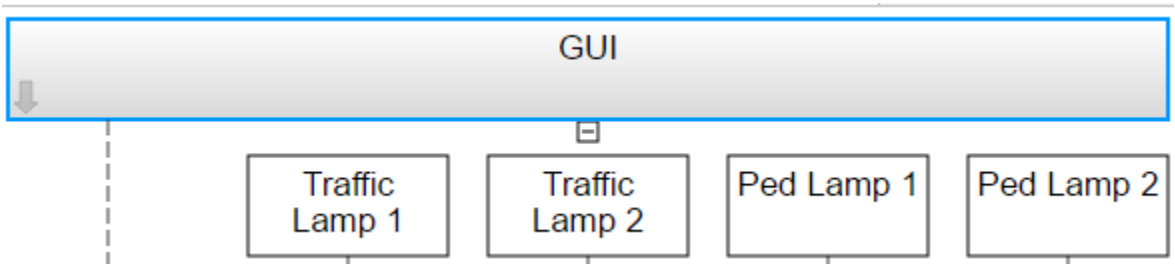
For example, expanding the lifeline for the Traffic Light 1 block reveals two new lifelines corresponding to the Stateflow charts Ped Button Sensor and Controller.



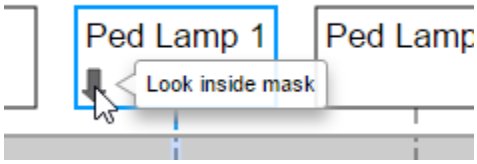
Expand a Masked Subsystem Lifeline

The Sequence Viewer window displays masked subsystems as white blocks. To show the children of a masked subsystem, point over the bottom left corner of the lifeline header and click the arrow.

For example, the GUI subsystem contains four masked subsystems: Traffic Lamp 1,Traffic Lamp 2, Ped Lamp 1, and Ped Lamp 2.

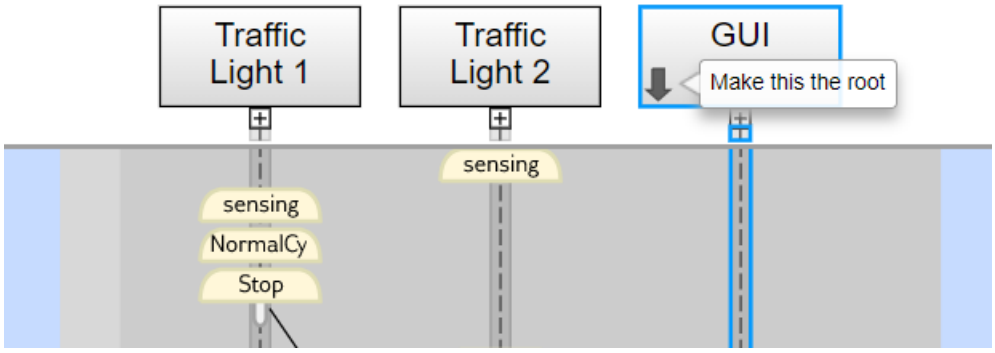


You can display the child lifelines in these masked subsystems by clicking the arrow in the parent lifeline header.

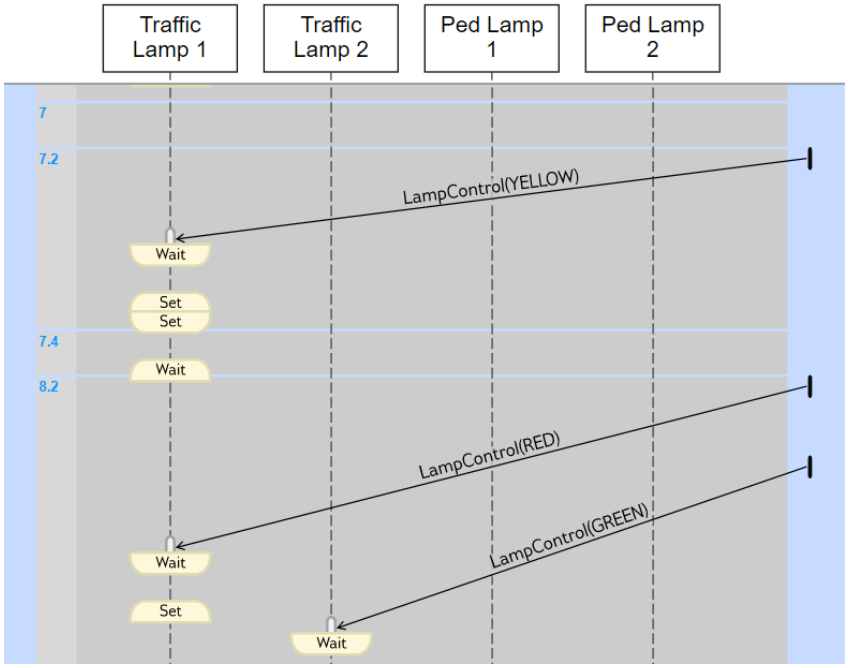


Change Root of Focus

To make a lifeline the root of focus for the viewer, point over the bottom left corner of the lifeline header and click the arrow. Alternatively, you can use the navigation toolbar at the top of the Sequence Viewer window to move the current root up and down the lifeline hierarchy. To move the current root up one level, press the **Esc** key.



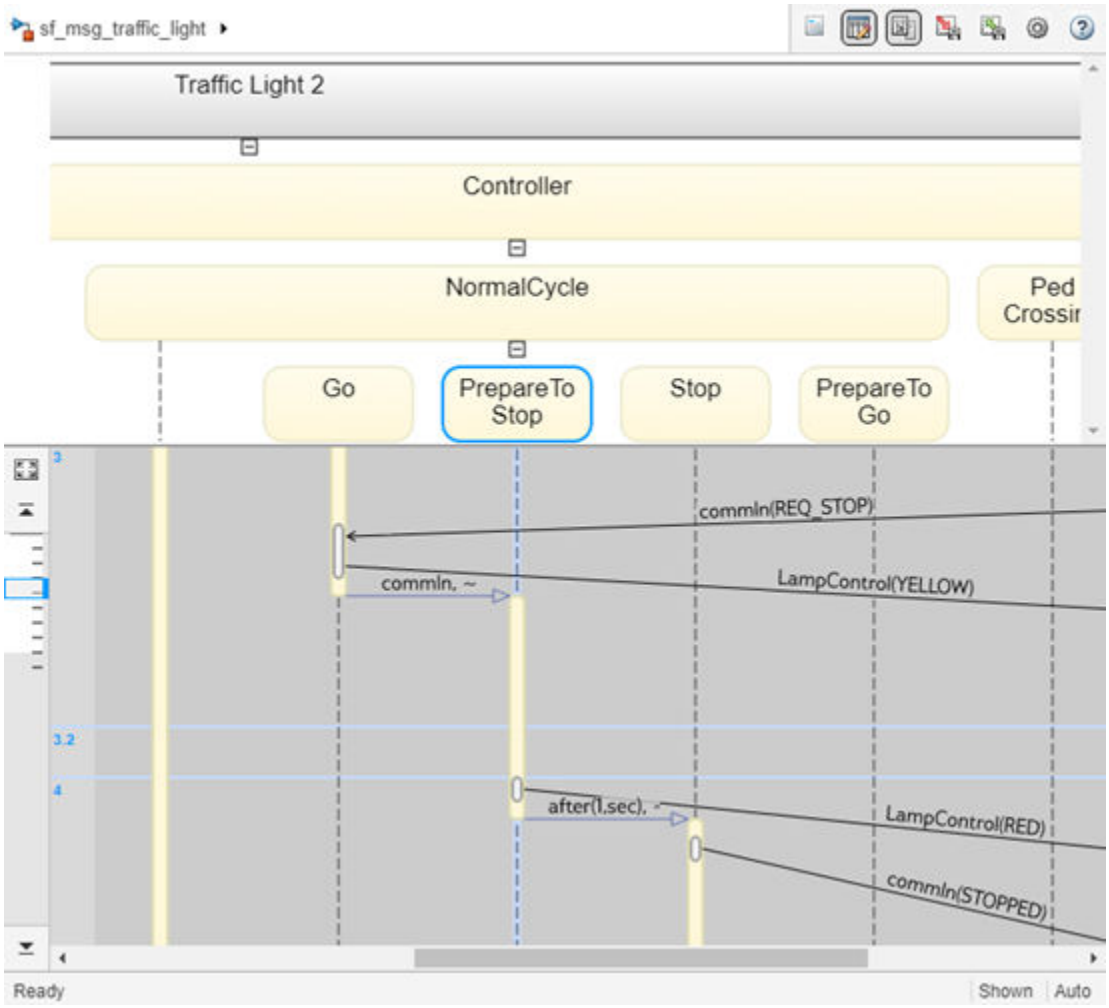
The Sequence Viewer window displays the current root lifeline path and shows its child lifelines. Any external events and messages are displayed as entering or exiting through vertical slots in the diagram gutter. When you point to a slot in the diagram gutter, a tooltip displays the name of the sending or receiving block.



View State Activity and Transitions

To see state activity and transitions in the Sequence Viewer window, expand the state hierarchy until you have reached the lowest child state. Vertical yellow bars show which state is active. Blue horizontal arrows denote the transitions between states.

In this example, you can see a transition from `Go` to `PrepareToStop` followed, after 1 second, by a transition to `Stop`.



To display the start state, end state, and full transition label in the Property Inspector, click the arrow corresponding to the transition.

To display information about the interactions that occur while a state is active, click the yellow bar corresponding to the state. In the Property Inspector, use the **Search Up** and **Search Down** buttons to move through the transitions, messages, events, and function calls that take place while the state is active.

View Function Calls

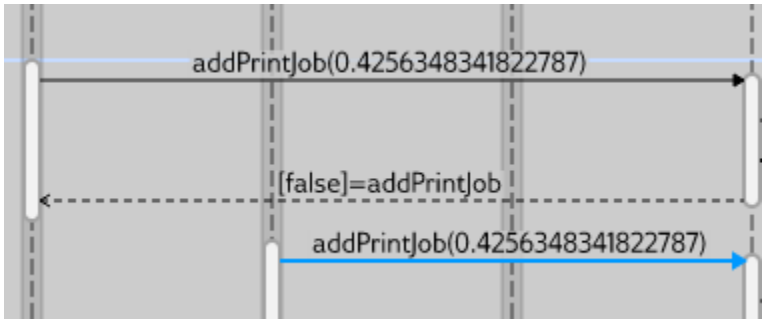
The Sequence Viewer block displays function calls and replies. This table lists the type of support for each type of function call.

Function Call Type	Support
Calls to Simulink Function blocks	Fully supported

Function Call Type	Support
Calls to Stateflow graphical or Stateflow MATLAB functions	• Scoped — Select the **Export chart level functions** chart option. Use the *chartName*.*functionName* dot notation. • Global — Select the **Treat exported functions as globally visible** chart option. You do not need the dot notation.
Calls to function-call subsystems	Not displayed in the Sequence Viewer window

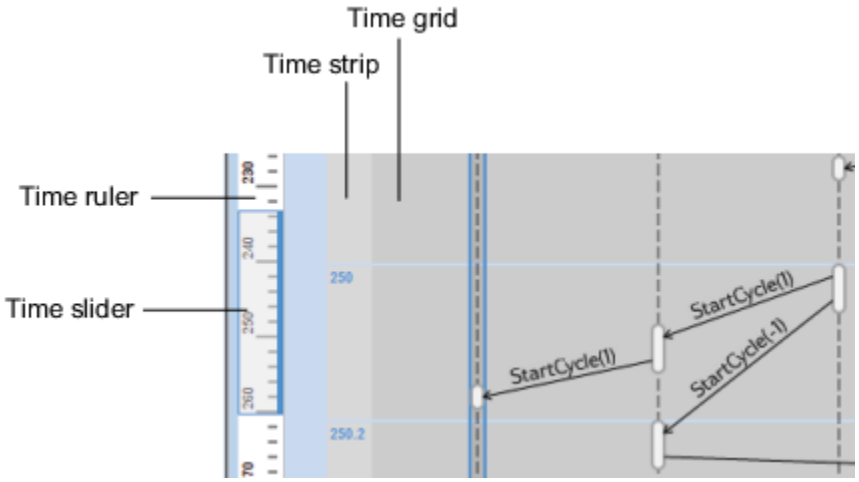
The Sequence Viewer window displays function calls as solid arrows labeled with the format *function_name*(*argument_list*). Replies to function calls are displayed as dashed arrows labeled with the format [*argument_list*]=*function_name*.

For example, in the model `slexPrinterExample`, a subsystem calls the Simulink Function block `addPrinterJob`. The function block replies with an output value of `false`.



Simulation Time in the Sequence Viewer Window

The Sequence Viewer window shows events vertically, ordered in time. Multiple events in Simulink can happen at the same time. Conversely, there can be long periods of time during simulation with no events. As a consequence, the Sequence Viewer window shows time by using a combination of linear and nonlinear displays. The time ruler shows linear simulation time. The time grid shows time in a nonlinear fashion. Each time grid row, bordered by two blue lines, contains events that occur at the same simulation time. The time strip provides the times of the events in that grid row.



To show events in a specific simulation time range, use the scroll wheel or drag the time slider up and down the time ruler. To navigate to the beginning or end of the simulation, click the **Go to first event** or **Go to last event** buttons. To see the entire simulation duration on the time ruler, click the **Fit to view** button  .

When using a variable step solver, you can adjust the precision of the time ruler. In the Model Explorer, on the **Main** tab of the Sequence Viewer Block Parameters pane, adjust the value of the **Time Precision for Variable Step** field.

Redisplay of Information in the Sequence Viewer Window

The Sequence Viewer block saves the order and states of lifelines between simulation runs. When you close and reopen the Sequence Viewer window, it preserves the last open lifeline state. To save a particular viewer state, click the **Save Settings** button  in the toolbar. Saving the model then saves that state information across sessions. To load the saved settings, click the **Restore Settings** button  .

You can modify the **Time Precision for Variable Step** and **History** parameters only between simulations. You can access the buttons in the toolbar before simulation or when the simulation is paused. During a simulation, the buttons in the toolbar are disabled.

See Also

Sequence Viewer

More About

- “Synchronize Model Components by Broadcasting Events” (Stateflow)
- “Communicate with Stateflow Charts by Sending Messages” (Stateflow)
- “Model a Distributed Traffic Control System by Using Messages” (Stateflow)

Modeling Variant Systems

What Are Variants and When to Use Them

In this section...

What Are Variants?

In Simulink, you can use the variant blocks to create a single model that caters to multiple variant requirements. Such models have a fixed common structure and a finite set of variable components. The variable components are activated depending on the variant choice that you select. Thus, the resultant active model is a combination of the fixed structure and the variable components based on the variant choice.

The use of variant blocks in a model helps in reusability of the model for different conditional expressions called variant choices. This approach helps you to meet diverse customer requirements based on application, cost, or operational considerations.

You can use these variant blocks depending on the model design:

- Variant Subsystem: For hierarchical model structure. The block is a template with two Subsystem blocks to use as variant systems. You can add Subsystem blocks, as well as Model blocks, for variants.
- Variant Model: For hierarchical model structure. The block is a template with two Model blocks to use as variant systems. You can add Model blocks, as well as Subsystem blocks, for variants.
- Inline Variants: For flat model structure.
  - Variant Source
  - Variant Sink
  - Manual Variant Source
  - Manual Variant Sink

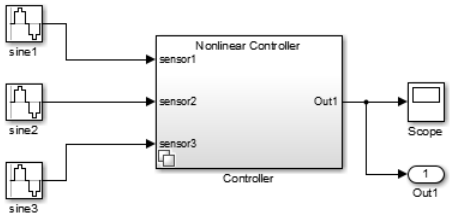
Note Sample time for single input Variant Source / Variant Sink blocks can differ with multiple input Variant Source / Variant Sink blocks. For more information on sample time, see “What Is Sample Time?” on page 7-2

Use of a Variant Subsystem block provides these advantages:

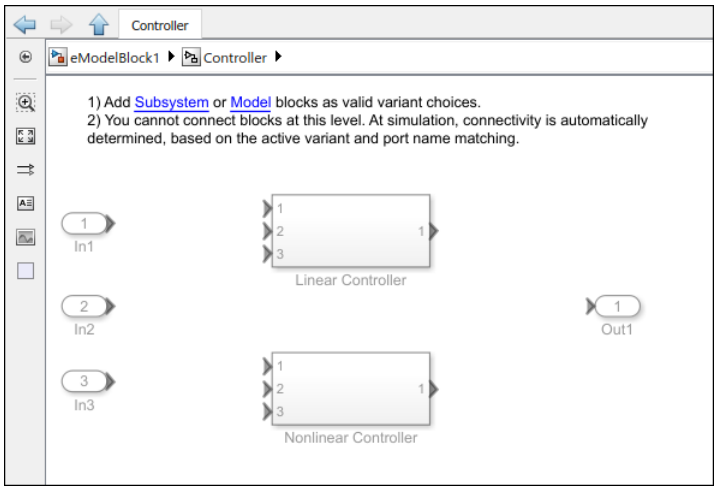
- Allows you to mix Model and Subsystem blocks as variant systems
- Supports flexible I/O, so that all variants do not need to have the same number of input and output ports

To convert a Model block that contains variant models to a Variant Subsystem block that contains Model blocks that reference the variant models, right-click the Model block and select **Subsystems & Model Reference > Convert to > Variant Subsystem**. Alternatively, you can use the `Simulink.VariantManager.convertToVariant` function. Specify the Model block name or block handle. The converted model produces the same results as the original model.

If you want to simulate a model that represents an automobile with several configurations. These configurations, although similar in several aspects, can differ in properties such as fuel consumption, engine size, or emission standard. Instead of designing multiple models that together represent all possible configurations, you can use variants to model only the varying configurations. This approach keeps the common components fixed.



This model contains two variant choices inside a single Variant Subsystem block. Variant choices are two or more configurations of a component in your model.



Advantages of Using Variants

Using variants in Model-Based Design provides several advantages:

- Variants provide you a way to design one model for many systems.
- You can rapidly prototype design possibilities as variants without having to comment out sections of your model.
- Variants help you develop modular design platforms that facilitate reuse and customization. This approach improves workflow speed by reducing complexity.
- If a model component has several alternative configurations, you can efficiently explore these varying alternatives without altering the fixed, unvarying components.

- You can use different variant configurations for simulation or code generation from the same model.
- You can simulate every design possibility in a combinatorial fashion for a given test suite.
- If you are working with large-scale designs, you can distribute the process of testing these designs on a cluster of multicore computers. Alternatively, you can map different test suites to design alternatives for efficiently managing design-specific tests.
- You can generate a reduced model with a subset of configuration from a master model with many variants.

When to Use Variants

Variants help you specify multiple implementations of a model in a single, unified block diagram. Here are three scenarios where you can use variants:

- Models that represent multiple simulation, code generation, or testing workflows.

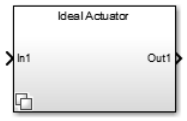


Figure 1. Model of an ideal actuator for basic simulation.

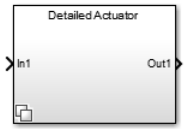


Figure 2. Detailed actuator for simulating customizations.

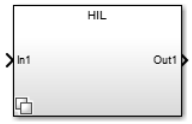
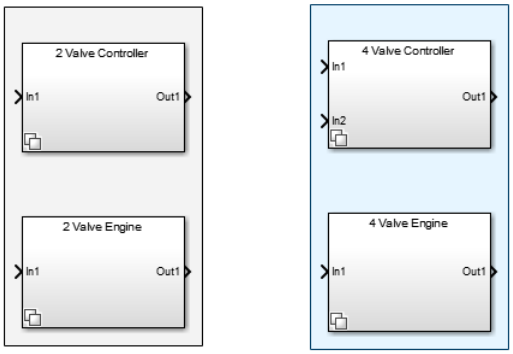


Figure 3. Model that is configured for Hardware-in-the-Loop simulation.

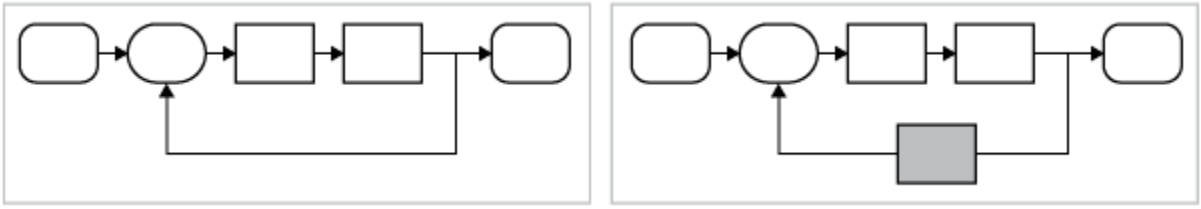
- Models that contain multiple design choices at the component level.



Subsystem blocks representing variant choices can have input and output ports that differ in number from the input and output ports in the parent Variant Subsystem block. See “Mapping Inports and Outports of Variant Choices” on page 12-5.

- Models that are mostly similar but have slight variations, such as in cases where you want to separate a test model from a debugging model.

  The test model on the left has a fixed design. On the right, the same test model includes a variant that is introduced for debugging purposes.



Simulink selects the active variant during update diagram time and during code compile time.

Options for Representing Variants in Simulink

You can represent one or more variants as variant choices inside these blocks.

	Variant Source and Variant Sink blocks	Variant Subsystem and Variant Model blocks
Variant choice representation	Number of ports	Subsystem or Model block
Allows choice hierarchy	No	Yes
Mismatched number of input and output ports among variant choices	Simulink disables inactive ports	Simulink disables inactive ports
Option to specify default variant	Yes	Yes
Supports control ports	No	Yes
Can be saved as standalone file	No	No
Supports physical modeling connection ports	No	Partially
Comment choice (%)	No	No

In addition, you can represent variant choices using Variant Source and Variant Sink block. These blocks enable the propagation of variant conditions throughout the model and can propagate conditions through model reference hierarchy.

You can create variants at several levels inside your model hierarchy.

Mapping Inports and Outports of Variant Choices

A Variant Subsystem block is a container of variants choices that are represented as Subsystem or Model blocks. The inputs that the Variant Subsystem block receives from upstream models components map to the input and output ports of the variant choices.

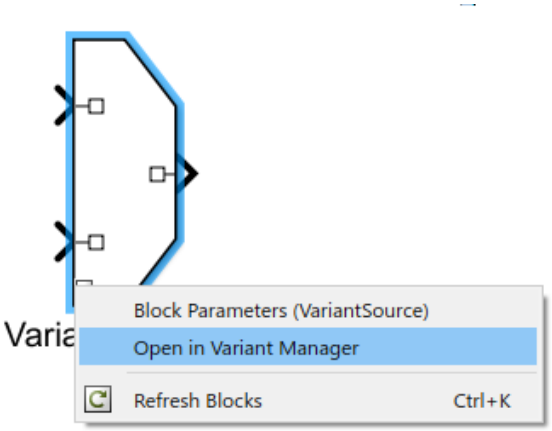
Subsystem and Model blocks representing variant choices can have input and output ports that differ in number from the input and output ports in the parent Variant Subsystem block. However, the following conditions must be satisfied:

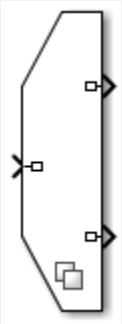
- The names of the inports of a variant choice are a subset of the inport names used by the parent variant subsystem.
- The names of the output ports of a variant choice are a subset of the output port names used by the parent variant subsystem.
- If variant choices have control port, the name of data input port must match with control port name.

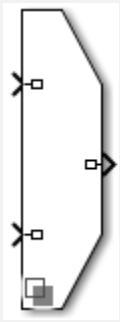
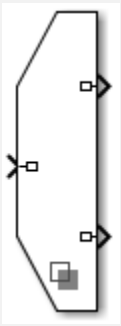
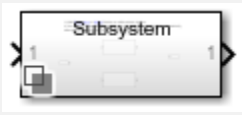
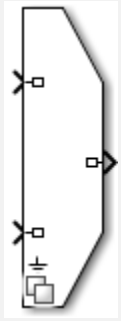
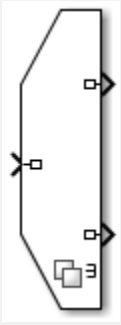
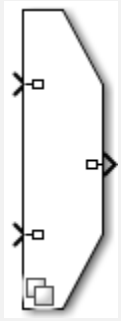
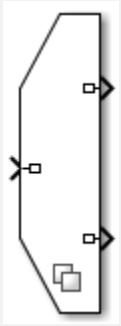
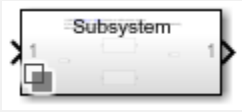
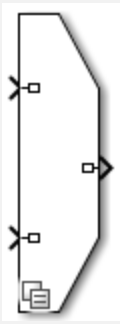
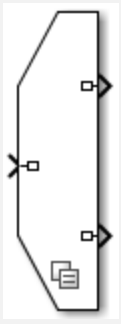
During simulation, Simulink disables the inactive ports in a Variant Subsystem block.

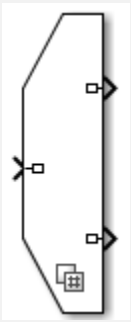
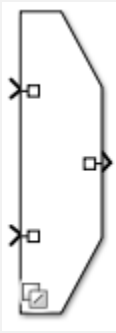
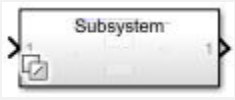
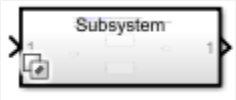
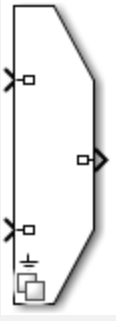
Variant Badges

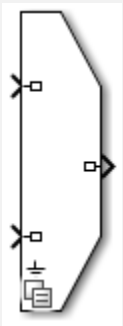
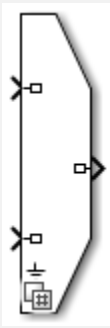
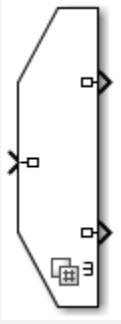
Each Variant block has a badge associated with it. The color and icon of a Variant badge indicate the status of the Variant block. It also provides quick access to few Variant commands. You can right-click the Variant badge to access these commands.

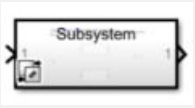


Variant Badge	Variant Source	Variant Sink	Variant Subsystem
Default Variant badge when no option is selected.			

Variant Badge	Variant Source	Variant Sink	Variant Subsystem
Variant block with `Label` selected as **Variant control mode** and an active variant choice is selected from **Label mode active choice** option.			
Variant block with **Allow zero active variant controls** option selected.			
Variant block with `expression` selected as **Variant control mode** and `update diagram` selected as **Variant activation time**.			
Variant block with `expression` selected as **Variant control mode** and `update diagram and analyze all choices` selected as **Variant activation time**.			

Variant Badge	Variant Source	Variant Sink	Variant Subsystem
Variant block with `expression` selected as **Variant control mode** and `code compile` selected as **Variant activation time**.			
Variant block with `sim codegen switching` selected as **Variant control mode** and `update diagram` selected as **Variant activation time**.			
Variant block with `sim codegen switching` selected as **Variant control mode** and `update diagram and analyze all choices` selected as **Variant activation time**.			
Variant block with **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	
Variant block with `update diagram` selected as **Variant activation time** and **Allow zero active variant controls** option selected.			

Variant Badge	Variant Source	Variant Sink	Variant Subsystem
Variant block with `update diagram and analyze all choices` selected as **Variant activation time** and **Allow zero active variant controls** option selected.			
Variant block with `code compile` selected as **Variant activation time** and **Allow zero active variant controls** option selected.			
Variant block with `update diagram` selected as **Variant activation time** and **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	
Variant block with `update diagram and analyze all choices` selected as **Variant activation time** and **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	
Variant block with `code compile` selected as **Variant activation time** and **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	

Variant Badge	Variant Source	Variant Sink	Variant Subsystem
Variant block with **Allow zero active variant controls** and **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	
Variant block with `sim codegen switching` selected as **Variant control mode** and **Propagate conditions outside of variant subsystem** option selected.	Not applicable	Not applicable	

Comment Out and Comment Through

Consider when you want to simulate a Simulink model by excluding some of its blocks from simulation and without physically removing the blocks from the model. The **Comment Out** and **Comment Through** commands in Simulink provide you with an option to exclude blocks from simulation. Depending on your modeling requirement, you can use these options:

- **Comment Out**: Excludes the selected block from simulation. The signals are terminated and grounded.
- **Comment Through**: Excludes the selected block from simulation. The signals are passed through. To comment through a block, the number of input ports and the output ports for the block must be same.

To access the **Comment Out** or the **Comment Through** options, right-click the block and in the context menu either select **Comment Out** or **Comment Through** based on your modeling requirement.

Alternatively, you can also select the block and press **Ctrl+Shift+X** to comment out or press **Ctrl +Shift+Y** to comment through.

You can use `get_param` and `set_param` commands to view or change the commented state of a block programmatically. For example,

- `get_param(gcb,'commented'); % To view the commented state of the block`
- `set_param(gcb,'commented','on'); % To comment out a block`
- `set_param(gcb,'commented','through'); % To comment through a block`
- `set_param(gcb,'commented','off'); % To uncomment a block`

When you comment out a block, the signal names at the output port of the block are ignored. To include such signals during simulation, the signal name must be added at the input port of the block.

Comment Out and **Comment Through** are not supported with these blocks: Inport, Outport, Duplicate Port, Connection ports, Argument Inport, Argument Outport, Data Store Memory, Signal Generator, Goto Tag Visibility, For, and While blocks.

See Also

Related Examples

- “Define, Configure, and Activate Variants” on page 12-42
- “Create and Validate Variant Configurations” on page 12-69
- “Create Variant Controls Programmatically” on page 12-40
- “Working with Variant Choices” on page 12-21
- “Transform Model to Variant System” (Simulink Check)
- “Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model” (Simulink Check)

More About

- “Introduction to Variant Controls” on page 12-24
- “Create a Simple Variant Model” on page 12-36
- Variant System Design
- “Commenting Stateflow Objects in a Chart” (Stateflow)

Variant Terminology

Simulink variant terminology helps you to understand various parameters and terms.

Variant Terminology	Description
Variant Subsystem, Variant Model	Contains one or more choices where each choice is a Subsystem or Model block.
Variant Source	Provides variation on the source of a signal.
Variant Sink	Provides variation on the sink (destination) of a signal.
Variant Model	Variant Subsystem block containing Model block as variant choices.
Active choice on page 12-22	Variant choice associated with a variant control that evaluates to `true`.
Variant control expression on page 12-25	Boolean expression or a `Simulink.Variant` object containing a boolean expression or `(default)`. Used as Variant control mode.
Variant control label on page 12-25	String that is not evaluated and the choice used in simulation is determined by the **Label mode active choice** parameter. Used as Variant control mode.
Variant control variable	MATLAB variable, `Simulink.Variant` object, or a `Simulink.Parameter` object.
Variant object	Container of variant control expression.
Variant Manager	Central tool that allows you to manage various variation points that are modeled using variant blocks in a system model.
Variant Reducer on page 12-77	Reduces variant models to simplified, standalone model depending on the selected variant configurations.

Command Line Parameters

Variant Subsystem Parameters

Parameter name	Description
`Variant`	Used to check if the subsystem is a Variant Subsystem block. Returns `on` if the subsystem is a Variant Subsystem block, else it returns `off`. Example: `get_param(gcb, 'Variant')` This is a read-only parameter.
`VariantControl`	Subsystem block and Model block parameter, which applies to a choice block of a Variant Subsystem block and returns the variant control for the choice block. • `get_param` behavior: Returns variant controls for choice block of a Variant Subsystem block. Example: `get_param(gcb, 'VariantControl')` • `set_param` behavior: Sets variant control for current block. Example: `set_param(gcb, 'VariantControl', 'A==1')`
`VariantControlMode`	Specifies the mode for modeling Variant blocks, which can be either `Expression` or `Label`. • `get_param` behavior: Returns the mode set for modeling Variant blocks. Example: `get_param(gcb, 'VariantControlMode')` • `set_param` behavior: Sets the mode for modeling Variant blocks. Example: `set_param(gcb, 'VariantControlMode', 'Label')`

Parameter name	Description
`LabelModeActiveChoice`	Returns the variant control label of the selected choice for the Variant Subsystem block when `VariantControlMode` is specified as `Label` . If `VariantControlMode` is specified as `Expression`, this parameter returns empty (`''`). • `get_param` behavior: Returns the variant control label of the selected choice for the Variant Subsystem when `VariantControlMode` is set to `Label`. If `VariantControlMode` is set to `Expression`, this parameter returns empty (`''`). • `set_param` behavior: When `VariantControlMode` is specified as `Expression`, `set_param` makes the label selected as the active choice. When `VariantControlMode` is `Label`, `set_param` switches between specified labels.
`TreatAsGroupedWhenPropagatingVariantConditions on page 12-132`	Returns `on` if the Subsystem is treated as a group when propagating variant conditions else returns `off`. • `get_param` behavior: Indicates if the subsystem is treated as a group when propagating Variant conditions or not by returning `on` or `off`. Example: `get_param(gcb, 'TreatAsGroupedWhenPropagatingVariantConditions')` • `set_param` behavior: Enables or disables treating subsystem as a group when propagating variant conditions. Example: `set_param(gcb, 'TreatAsGroupedWhenPropagatingVariantConditions', 'on')`

Parameter name	Description
`GeneratePreprocessorConditionals`	Indicates if all the choices are to be analyzed and preprocessor conditionals to be generated by returning `on` or `off`. • `get_param` behavior: Indicates if all the choices are to be analyzed and preprocessor conditionals be generated by returning `on` or `off`. Example: `get_param(gcb, 'GeneratePreprocessorConditionals')` • `set_param` behavior: Enables or disables analyzing all the choices and generating preprocessor conditionals. Example: `set_param(gcb, 'GeneratePreprocessorConditionals', 'on')`
`CompiledActiveChoiceControl`	Returns the variant control corresponding to the active choice of the Variant Subsystem block and returns empty `(' ')` when no choice is active. When the block is commented or is inside a commented subsystem, this parameter returns empty`(' ')`. Example: `get_param(gcb, 'CompiledActiveChoiceControl')`. This is a read-only parameter.
`CompiledActiveChoiceBlock`	Returns the full block path name of the active Variant Subsystem block choice and returns empty `(' ')` when no choice is active. When the block is commented or is inside a commented Subsystem, the value is returned as empty`(' ')`. Example: `get_param(gcb, 'CompiledActiveChoiceBlock')`. This is a read-only parameter.
`CompiledVariantInfo`	Indicates if a block is active during simulation and if it is part of generated code. Example: `get_param(gcb, 'CompiledVariantInfo')`. This is a read-only parameter.

Parameter name	Description
`PropagateVariantConditions`	Indicates if conditions on ports inside the Variant Subsystem block are to be propagated outside the block. • `get_param` behavior: Indicates if conditions on ports inside the Variant Subsystem block are to be propagated outside the block. • `set_param` behavior: Enables or disables propagating conditions outside Variant Subsystem block. Example: `set_param(gcb, 'PropagateVariantConditions','on')`
`AllowZeroVariantControls`	Indicates if the Variant Subsystem block is allowed to have no active choices. • `get_param` behavior: Indicates if the Variant Subsystem block is allowed to have no active choices. • `set_param` behavior: Enables or disables having active choices in Variant Subsystem block. Example: `set_param(gcb, 'AllowZeroVariantControls','on')`

Variant Source and Variant Sink Parameters

Parameter name	Description
`VariantControls`	Returns a `1-by-N` cell array of variant control expressions corresponding to each of the `N` ports of the Variant Source or Variant Sink blocks. • `get_param` behavior: Returns a cell array of variant control expressions corresponding to each ports of the Variant Source or Variant Sink blocks. Example: `get_param(gcb, 'VariantControls')` • `set_param` behavior: Sets the cell array of Variant control expressions corresponding to each of ports of Variant Source or Variant Sink blocks. Example: `set_param(gcb, 'VariantControls', ('{A==1}, '4'))`
`VariantControlMode`	Specifies the mode for modeling variant blocks, which can be either `Expression` or `Label`. • `get_param` behavior: Returns the mode set for modeling Variant blocks. Example: `get_param(gcb, 'VariantControlMode')` • `set_param` behavior: Sets the mode for modeling Variant blocks. Example: `set_param(gcb, 'VariantControlMode', 'Label')`

Parameter name	Description
`LabelModeActiveChoice`	Returns the variant control label of the selected choice for Variant Source or Variant Sink block when `VariantControlMode` is specified as `Label`. If `VariantControlMode` is specified as `Expression`, this parameter returns empty (`''`). • `get_param` behavior: Returns the variant control label of the selected choice for the Variant Subsystem when `VariantControlMode` is set to `Label`. If `VariantControlMode` is set to `Expression`, this parameter returns empty (`''`). • `set_param` behavior: When `VariantControlMode` is specified as `Expression`, `set_param` makes the label selected as the active choice. When `VariantControlMode` is `Label`, `set_param` switches between specified labels.
`GeneratePreprocessorConditionals`	Indicates if all the choices are to be analyzed and preprocessor conditionals to be generated by returning `on` or `off`. • `get_param` behavior: Indicates if all the choices are to be analyzed and preprocessor conditionals to be generated by returning `on` or `off`. Example: `get_param(gcb, 'GeneratePreprocessorConditionals')` • `set_param` behavior: Enables or disables analyzing all the choices and generating preprocessor conditionals. Example: `set_param(gcb, 'GeneratePreprocessorConditionals', 'on')`

Parameter name	Description
`ShowConditionOnBlock`	Indicates if the `VariantControlExpression` is to be displayed on the block by returning `on` or `off`. • `get_param` behavior: Indicates if the `VariantControlExpression` is to be displayed on the block or not. • `set_param` behavior: Enables or disables the displaying of `VariantControlExpression` on the block. Example: `set_param(gcb, 'ShowConditionOnBlock','on')`
`AllowZeroVariantControls`	Indicates if the block is allowed to have no active ports by returning `on` or `off`. • `get_param` behavior: Indicates if the Variant Source or Variant Sink block is allowed to have no active choices. • `set_param` behavior: Enables or disables having active choices in Variant Source or Variant Sink block. Example: `set_param(gcb, 'AllowZeroVariantControls','on')`
`CompiledActiveVariantControl`	Returns the variant control corresponding to the active port from the last compilation instance. If no port is active, returns empty `(' ')`. If the block is commented or inside a commented Subsystem or inside an inactive choice of a Variant Subsystem block, the value is not computed and returns empty `(' ')`. Example: `get_param(gcb, 'CompiledActiveVariantControl')` This is a read-only parameter.

Parameter name	Description
`CompiledActiveVariantPort`	Returns the "index" of the active port from the last compilation instance or returns `-1` when no port is active. If the block is commented or inside a commented Subsystem or inside an inactive choice of a Variant Subsystem block (with generate preprocessor conditionals Off), the value is not computed, and returns empty `(' ')`. Example: `get_param(gcb, 'CompiledActiveVariantPort')` This is a read-only parameter.

See Also

Related Examples

- Variant Subsystem, Variant Model
- Variant Source
- Variant Sink

More About

- Variant System Design

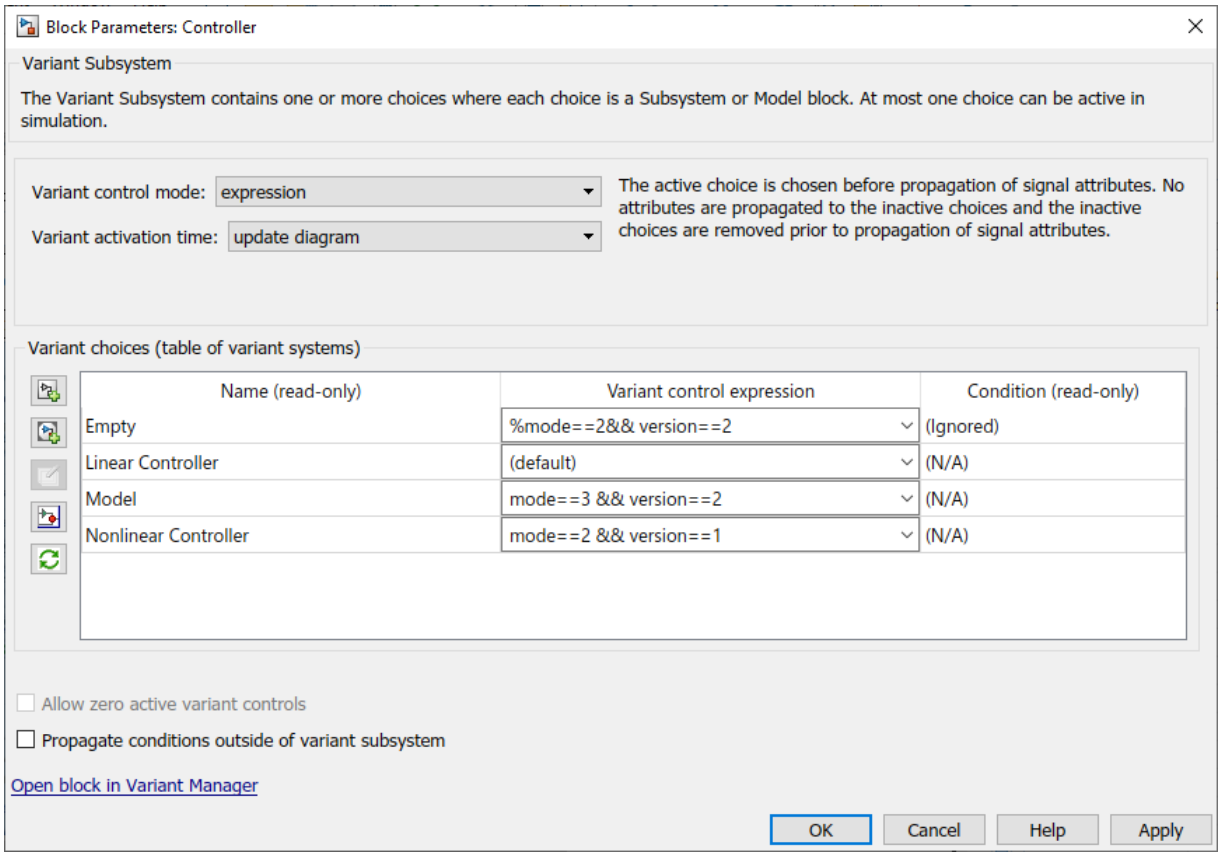
Working with Variant Choices

In this section...

Each variant choice in your model is associated with a conditional expression called variant control. The way you specify your variant controls determines the active variant choice. **Variant control mode** parameter available in the block parameters dialog box allows you to select `Expression` or `Label` mode for modeling Variant blocks.

This image shows the block parameters dialog box of a Variant Subsystem block that contains four variant choices:

- The first choice is commented out by adding the `%` symbol before the variant control.
- The second choice is the `(default)` and is activated when no variant control evaluates to `true`.
- The third choice is activated when the expression `mode==3 && version==2` evaluates to `true`.
- The fourth choice is activated when the expression `mode==2 && version==1` evaluates to `true`.



Default Variant Choice

You can specify at most one variant choice as the default for the model. As shown in the image above, the `Linear Controller` subsystem is defined as the default variant choice. During model compilation, if Simulink finds that no variant control evaluates to `true`, it uses the default choice.

In the dialog box, select the variant choice and change its **Variant control** property to `(default)`.

Active Variant Choice

While each variant choice is associated with a variant control, 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. At most one variant choice can be active. The active variant cannot be changed once model is compiled.

In this example, you can activate either the `Model` variant choice or the `Nonlinear Controller` variant choice by specifying appropriate values for `mode` and `version`.

Value of `mode`	Value of `version`	Active variant choice
2	1	`Nonlinear Controller`
3	2	`Model`

You can specify the values of `mode` and `version` at the MATLAB Command Window.

Inactive Variant Choice

When a variant control activates one variant choice, Simulink considers the other variant choices to be inactive. Simulink ignores inactive variant choices during simulation. However, Simulink continues to execute block callbacks inside the inactive variant choices.

The color of inactive choices fades by default. You can choose to disable the fading effect by using the **Variant Fading** option. The **Variant Fading** option is available in the **Information Overlays** menu on the **Debug** tab of the Simulink Editor. You can use `get_param` and `set_param` commands to view or change the fading state of inactive choices programmatically. For example,

- `get_param('bdroot,'VariantFading') % To view the fading state of inactive choices`
- `set_param('bdroot,'VariantFading','on') % To turn on the fading effect of inactive choices`

Empty Variant Choice

When you are prototyping variant choices, you can create empty Subsystem blocks with no inputs or outputs inside the Variant Subsystem block. The empty subsystem recreates the situation in which that subsystem is inactive without the need for completely modeling the variant choice.

For an empty variant choice, you can either specify a variant activation condition or comment out the variant condition by placing a `%` symbol before the condition.

If this variant choice is active during simulation, Simulink ignores the empty variant choice. However, Simulink continues to execute block callbacks inside the empty variant choices.

Open Active Variant

When you open a model, variant blocks display the name of the variant that was active the last time that you saved your model. Use the **Variant** menu to open the active variant. Right-click the block and select **Variant** > **Open**. Then select the active variant.

Use this command to find the current active choice:

```
get_param(gcb,'CompiledActiveChoiceControl')
```

Use this command to find the path to the current active choice:

```
get_param(gcb,'CompiledActiveChoiceBlock')
```

Note

- The `CompiledActiveChoiceBlock` parameter is supported only for the Variant Subsystem block.
- Active variant cannot be changed once the model is compiled.

See Also

Related Examples

- “Define, Configure, and Activate Variants” on page 12-42

More About

- “Introduction to Variant Controls” on page 12-24
- “Create a Simple Variant Model” on page 12-36
- Variant System Design

Introduction to Variant Controls

In this section...

The components of a Simulink model that contain Variants are activated or deactivated based on the variant choice that you select.

Each variant choice in your model is associated with a conditional expression called variant control. Variant controls determine which variant choice is active. By changing the value of a variant control, you can switch the active variant choice.

While each variant choice is associated with a variant control, only one variant control can evaluate to true. When a variant control evaluates to true, Simulink activates the variant choice that corresponds to that variant control.

A variant control is a Boolean expression that activates a specific variant choice when it evaluates to `true`.

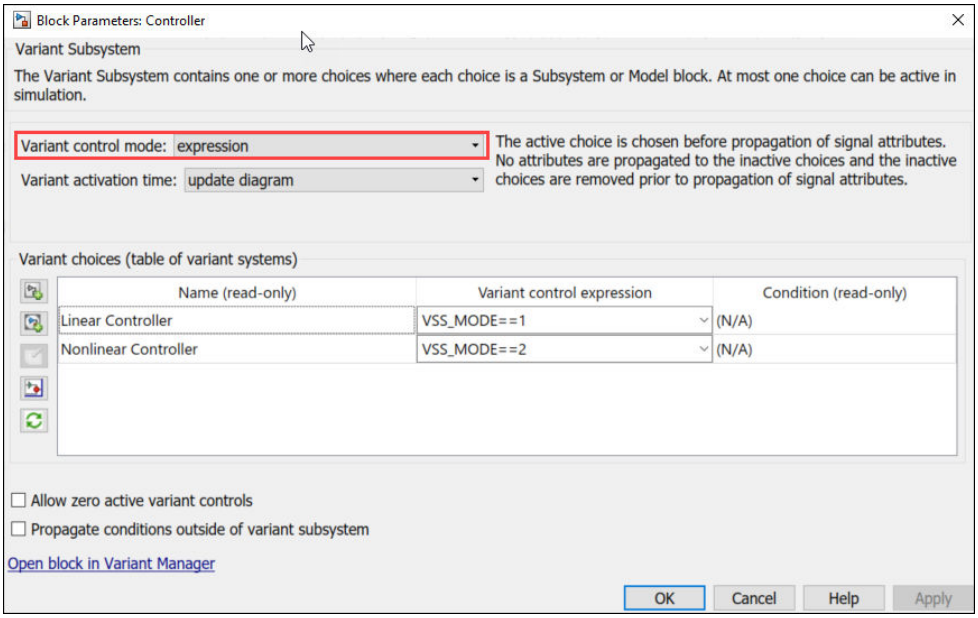
Note You can specify variant controls in the MATLAB global workspace, mask workspace, or a data dictionary.

You can specify variant controls as `Simulink.Variant` objects, MATLAB expressions (including structures) or as expressions that contain one or more of these operands and operators.

Variant control mode

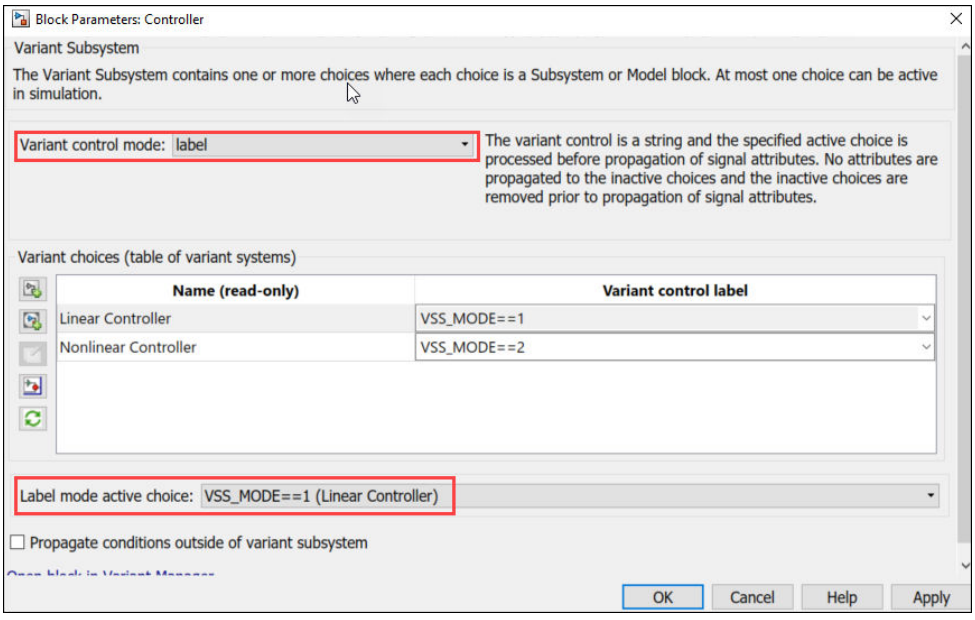
Variant control mode parameter available in the block parameters dialog box allows you to select `Expression` or `Label` or `sim codegen switching` mode for modeling Variant blocks.

- `Expression:` Specifies the active Variant based on the evaluation of the Variant conditions.



Variant control mode: Expression

- `Label:` Specifies the name based Variant controls (Label mode active choice). In `Label` mode, Variant control need not be created in the global workspace. Alternatively, you can select the Label mode active choice from the command line. For example, `set_param(block,'LabelModeActiveChoice', 'Choice_1')`.

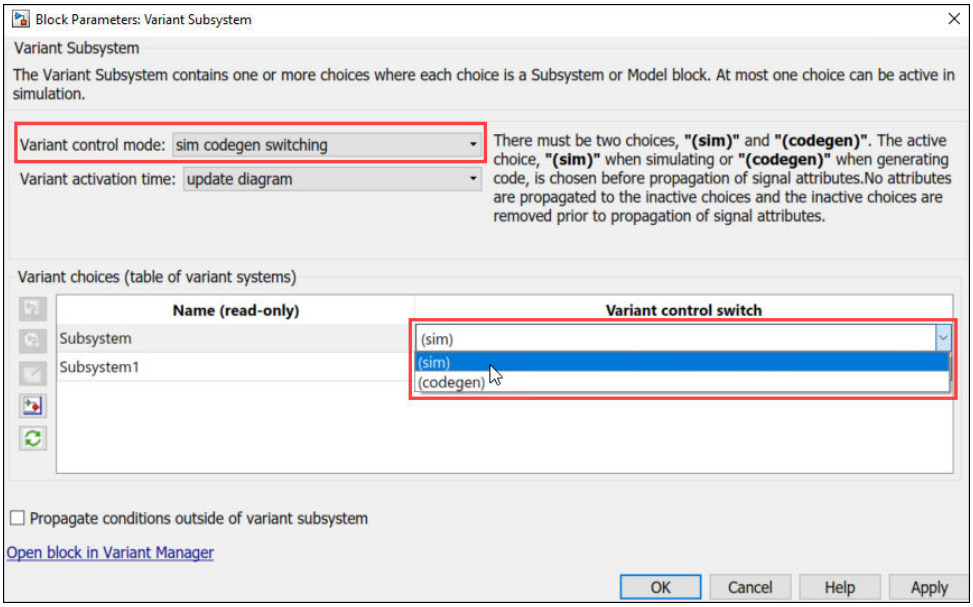


Variant control mode: Label

- `sim codegen switching:` enables automatic variant switching for simulation and code-generation workflows in variant blocks. This feature is convenient to switch between simulation and code-generation modes.

When you simulate (Normal , Accelerator, Rapid Accelerator) a model, then Simulink automatically chooses the `sim` branch as the active choice. Similarly, when you do a Software-in-the-loop (SIL) simulation, Processor-In-Loop (PIL) simulation or generate code or use external mode, Simulink automatically chooses the `codegen` branch.

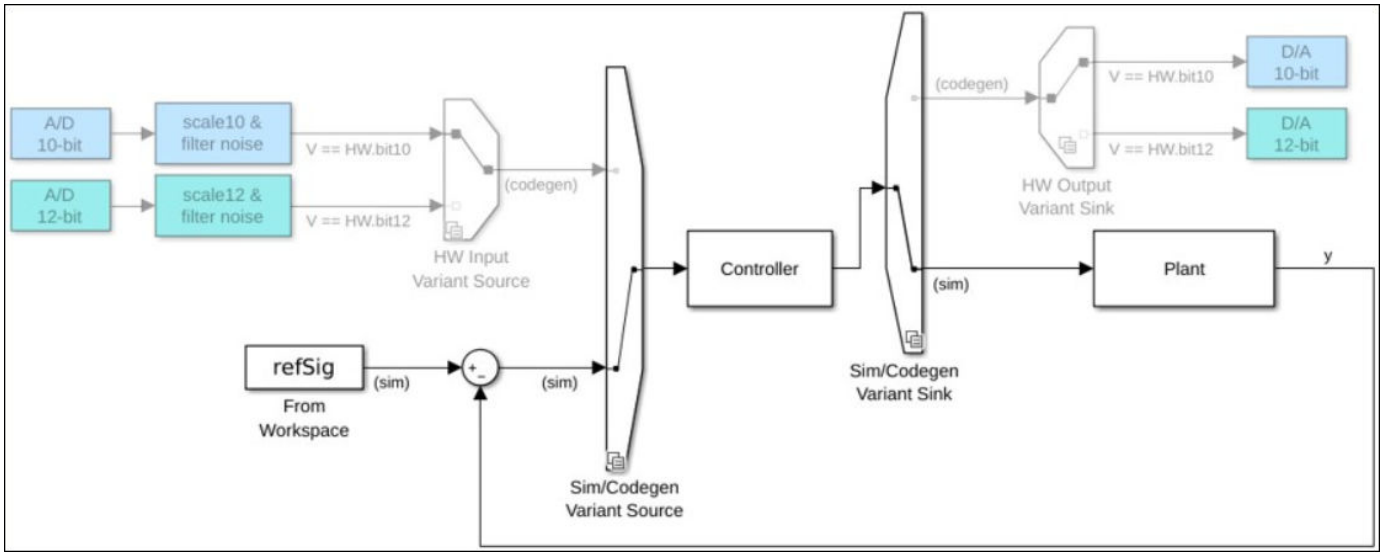
Note If a variant block has **Variant control mode** set to `label` or `expression`, then using `sim` or `codgen` for its choice condition is not supported.



Variant control mode: sim codegen switching

Note In the **Variant activation time** drop-down list, you can choose `update diagram` or `update diagram analyze all choices`. For data signals, `update diagram analyze all choices` ensures that the signal attributes (data types, dimensions, etc.) between both choices are consistent.

Here is a sample screen showing a variant block switched to sim choice.



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 bitwise operators

For more information, see “Operators and Operands in Variant Condition Expressions” on page 12-33.

When you compile the model, Simulink determines that a variant choice is active if its variant control evaluates to `true`. The evaluation of active variant happens in the early stages of compilation and the active variant cannot be changed once model is compiled.

Known Limitations

- Simulink variant objects within structures are not allowed.
- Simulink parameters within structures are not allowed.

Approaches for Specifying Variant Controls

You can use many approaches for switching between variant choices—from options to use while prototyping to options required for generating code from your model.

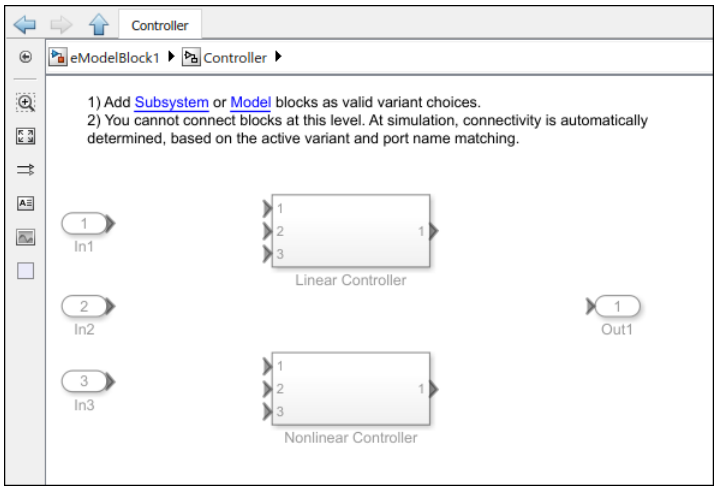
Specification	Purpose	Example
Scalar variable	Rapid prototyping	`A == 1`
`Simulink.Variant` object	Reuse variant conditions	`LinearController = Simulink.Variant('FUEL==2 && EMIS==1');`
`Simulink.Parameter` object or MATLAB variables	Generate preprocessor conditionals for code generation	`mode == 1`, where `mode` can be `Simulink.Parameter` object or MATLAB variables
Enumerated type	Improved code readability because condition values are represented as meaningful names instead of integers	`LEVEL == Level.Advanced`

You can find control variables using the function `Simulink.VariantManager.findVariantControlVars`.

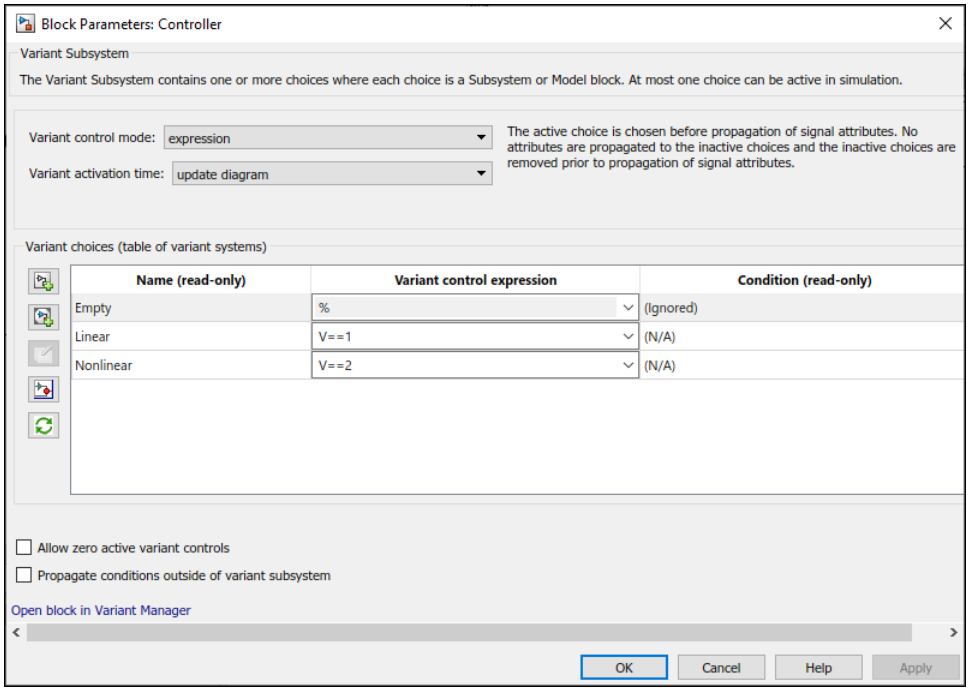
Scalar Variables for Rapid Prototyping

Scalar MATLAB variables allow you to rapidly prototype variant choices when you are still building your model. They help you focus more on building your variant choices than on developing the expressions that activate those choices.

Consider a model that contains two variant choices, each represented by a Variant Subsystem block.



You can specify variant controls in their simplest form as scalar variables in the block parameters dialog box of the Variant Subsystem block.

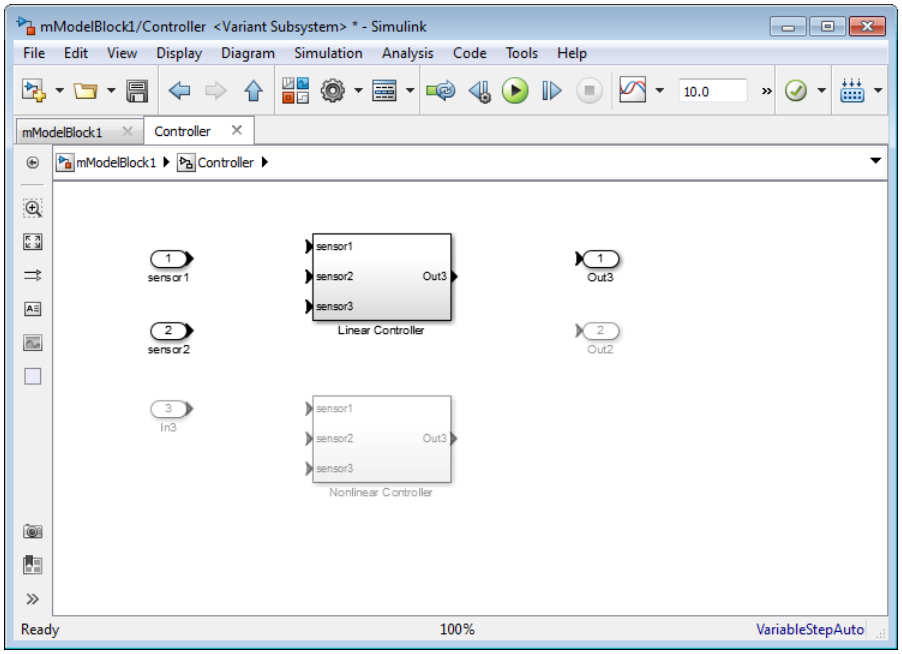


The **Condition** field for both the `Linear Controller` and `Nonlinear Controller` are `N/A`, because the variant control itself is the condition.

You can activate one of the variant choices by defining a scalar variable `V` and setting its value to `1` at the MATLAB Command Window.

```
V = 1;
```

This condition activates the `Linear Controller` variant choice. Variant controls are ignored when `%` symbol is used. Similarly, if variant control is empty, the choice is ignored.



Similarly, if you change the value of `V` to `2`, Simulink activates the `Nonlinear Controller` variant choice.

Simulink.Variant Objects for Variant Condition Reuse

After identifying the variant choices that your model requires, you can construct complex variant conditions to control the activation of your variant choices. Define variant conditions as `Simulink.Variant` objects.

`Simulink.Variant` objects enable you to reuse common variant conditions across models and help you encapsulate complex variant condition expressions.

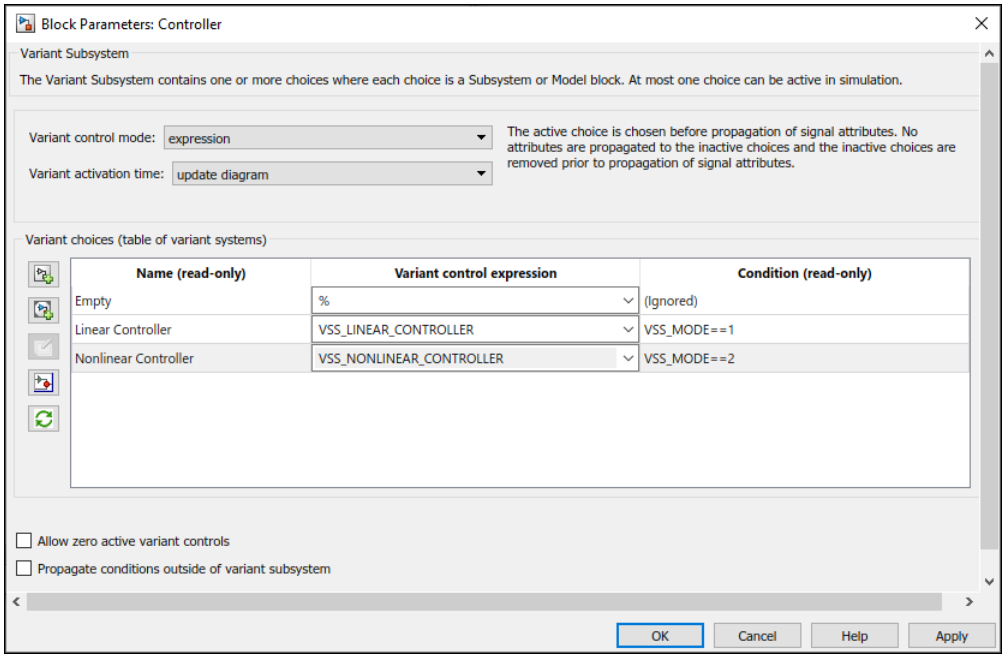
Consider an example where variant controls are already defined in the global workspace.

```
V=1;
V=2;
```

You can convert these controls into condition expressions encapsulated as `Simulink.Variant` objects.

```
LinearController=Simulink.Variant('V==1');
NonLinearController=Simulink.Variant('V==2');
```

You can then specify these `Simulink.Variant` objects as the variant controls in the block parameters dialog box of the Variant Subsystem block.



The **Condition** field now reflects the encapsulated variant condition. Using this approach, you can develop complex variant condition expressions that are reusable.

Simulink.Parameter Objects or MATLAB variables for Code Generation

If you intend to generate code for a model containing variant choices, specify variant control variables as MATLAB variables or `Simulink.Parameter` objects. `Simulink.Parameter` objects allow you to specify other attributes (such as data type) that are required for generating code.

```
VSSMODE = Simulink.Parameter;
VSSMODE.Value = 1;
VSSMODE.DataType = 'int32';
VSSMODE.CoderInfo.StorageClass = 'Custom';
VSSMODE.CoderInfo.CustomStorageClass = 'ImportedDefine';
VSSMODE.CoderInfo.CustomAttributes.HeaderFile =...
'rtwdemo_importedmacros.h';
```

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- Your own storage class that defines data as a macro

You can also convert a scalar variant control variable into a `Simulink.Parameter` object. See “Convert Variant Control Variables into Simulink.Parameter Objects” on page 12-49.

Enumerated Types for Improving Code Readability

Use enumerated types to give meaningful names to integers used as variant control values.

1 In the MATLAB Editor, define the classes that map enumerated values to meaningful names.

```
classdef sldemo_mrv_CONTROLLER_TYPE < Simulink.IntEnumType
        enumeration
        NONLINEAR (1)
        SECOND_ORDER (2)
        end
end
```

2 Define `Simulink.Variant` objects for these classes in the global workspace.

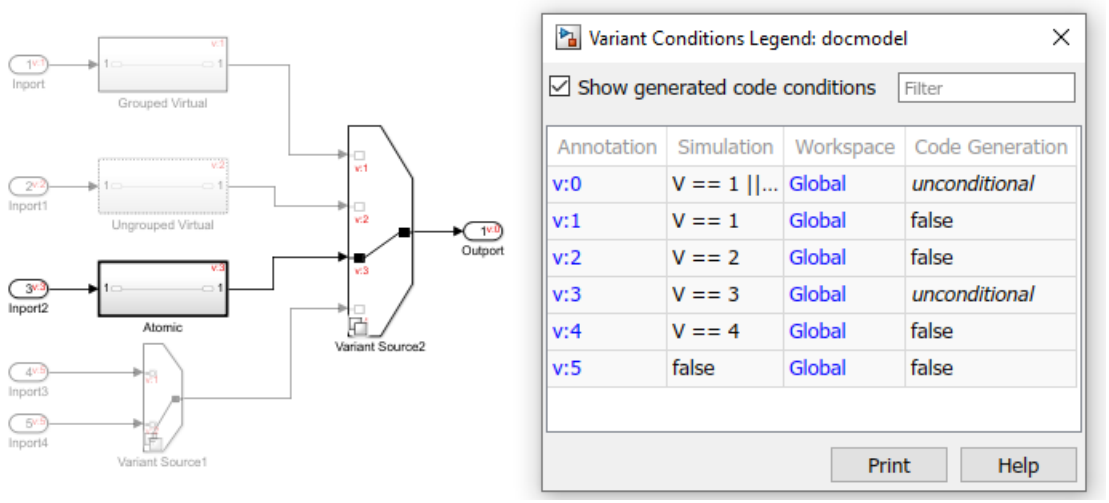
```
VE_NONLINEAR_CONTROLLER = Simulink.Variant...
('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.NONLINEAR')
VE_SECOND_ORDER_CONTROLLER =Simulink.Variant...
('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.SECOND_ORDER')
VE_PROTOTYPE =Simulink.Variant...
('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PROTOTYPE')
VE_PRODUCTION =Simulink.Variant...
('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PRODUCTION')
```

Using enumerated types simplifies the generated code because it contains the names of the values rather than integers.

Viewing Variant Conditions

The Variant Condition Legend helps you visualize the variant conditions associated with a model. To view the Variant Condition Legend, on the **Debug** tab, select **Information Overlays > Variant Legend**.

Note If **Variant Legend** is not available, on the **Debug** tab, select **Information Overlays > Variant Conditions**.



By default, the Variant Condition Legend displays the variant condition annotation, the variant condition during simulation, and the source of the variant condition variables. To view the variant condition in the generated code, select the **Show generated code conditions** option in the Variant Condition Legend window.

In the Variant Condition Legend, the variant conditions on the blocks are annotated as `v:c`, where `v` is the variant semantic indicator and `c` represents the variant condition index. You can click through the hyperlinked variant annotations to observe which parts of the model the condition corresponds to.

When you hover over a block that has a variant condition, the tooltip displays the variant annotation and the related variant condition for the block. To view the variant condition annotation tooltip, the **Variant Condition** option must be selected.

In the legend, the source of the variant condition variables are also displayed. The variables can originate from a mask, model, or a base workspace. The variables originating from different mask workspaces can have the same name and have different values. To observe the source of the variables, click the hyperlinked workspaces.

To view the **Variant Condition Legend** programmatically, use the `Simulink.VariantManager.VariantLegend` function in the MATLAB command window.

Operators and Operands in Variant Condition Expressions

Simulink evaluates condition expressions within variant controls to determine the active variant choice. You can include the following operands in a condition expression:

- Scalar variables
- `Simulink.Parameter` objects that are not structures and that have data types other than `Simulink.Bus` objects
- Enumerated types
- Parentheses for grouping

Variant condition expressions can contain MATLAB operators, provided the expression evaluates to a boolean value. In these examples, `A` and `B` are expressions that evaluate to an integer, and `x` is a constant integer literal.

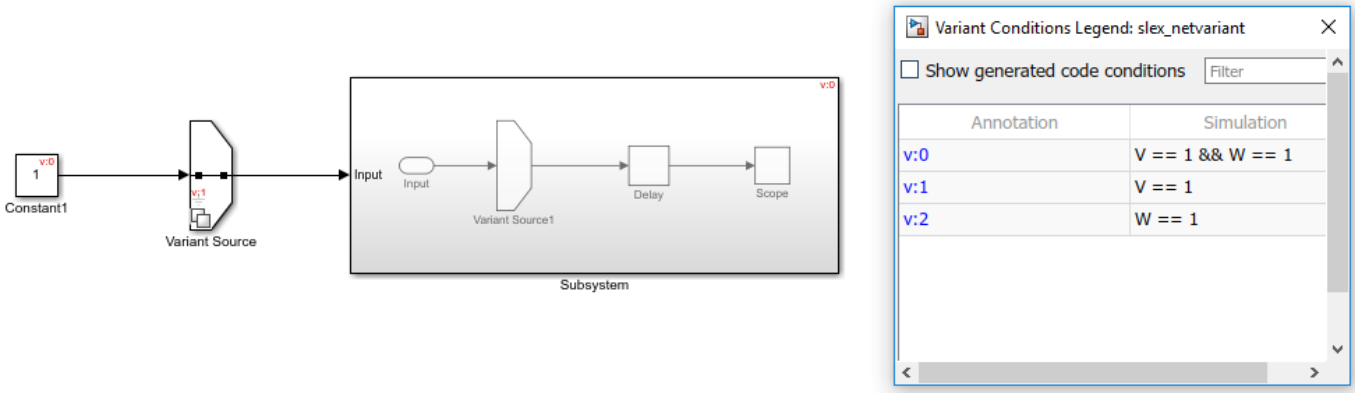
MATLAB Expressions That Support Generation of Preprocessor Conditionals	Equivalent Expression in C Preprocessor Conditional
Arithmetic	
• `A + B` • `+A`	• `A + B` • `A`
• `A - B` • `-A`	• `A - B` • `-A`
`A * B`	`A * B`
`idivide(A,B)`	`A / B` If the value of the second operand (`B`) is `0`, the behavior is undefined.
`rem(A,B)`	`A % B` If the value of the second operand (`B`) is `0`, the behavior is undefined.
Relational	

MATLAB Expressions That Support Generation of Preprocessor Conditionals	Equivalent Expression in C Preprocessor Conditional
`A == B`	`A == B`
`A ~= B`	`A != B`
`A < B`	`A < B`
`A > B`	`A > B`
`A <= B`	`A <= B`
`A >= B`	`A >= B`
Logical	
`~A`	`!A`, where `A` is not an integer
`A && B`	`A && B`
`A || B`	`A || B`
Bitwise (`A` and `B` cannot both be constant integer literals)	
`bitand(A,B)`	`A & B`
`bitor(A,B)`	`A | B`
`bitxor(A,B)`	`A ^ B`
`bitcmp(A)`	`~A`
`bitshift(A,x)`	`A << x`
`bitshift(A,-x)`	`A >> x`

Net Variant Condition

The net variant condition is the total of the local condition and its ancestral condition.

Consider this model `slex_netvariant` with two Single-input Single-Output (SISO) Variant Source blocks, `Variant Source` and `Variant Source1` with variant conditions as `V==1` and `W==1`, respectively.



When you simulate this model, the `Variant Source1` block and other blocks within the Subsystem block will have a local condition `W==1` propagated from the `Variant Source1` block. The ancestral condition `V==1` is propagated from the `Variant Source` block onto the Subsystem block. Therefore,

the net variant condition on the `Variant Source1` block and other blocks within the Subsystem block will be `V==1 && W==1`.

See Also

Related Examples

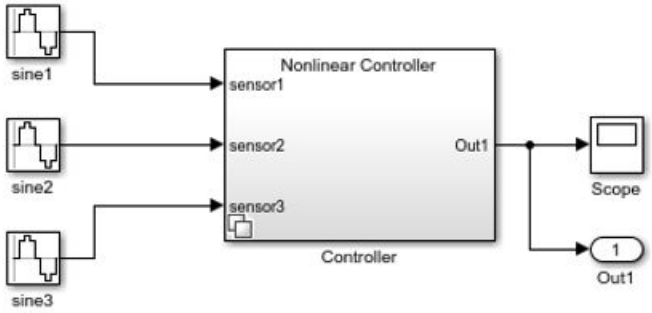
- “Define, Configure, and Activate Variants” on page 12-42
- “Create and Validate Variant Configurations” on page 12-69
- “Create Variant Controls Programmatically” on page 12-40
- “Working with Variant Choices” on page 12-21

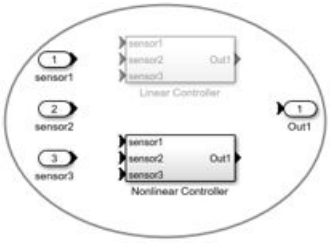
More About

- “Create a Simple Variant Model” on page 12-36
- Variant System Design

Create a Simple Variant Model

1. Create a model that contains variant blocks. For example, see “Variant Subsystems” on page 12-136 that contains a Variant Subsystem block (`Controller`).





Components of 'Controller' block

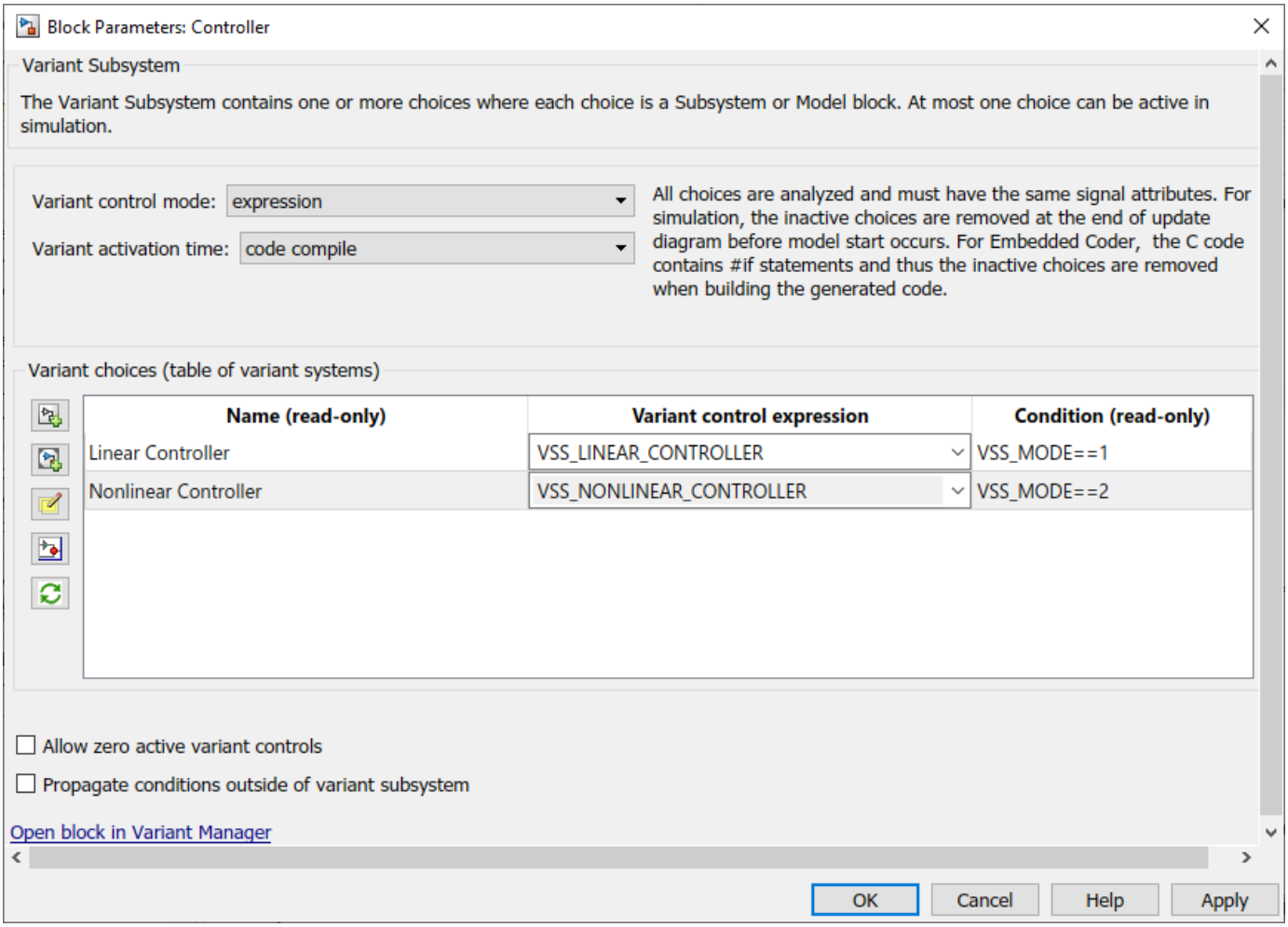
2. Define variant control variables that determine the condition under which a variant choice is active.

   a. Right-click the variant block and click **Block Parameters**. The Block Parameters dialog box for the variant block opens.

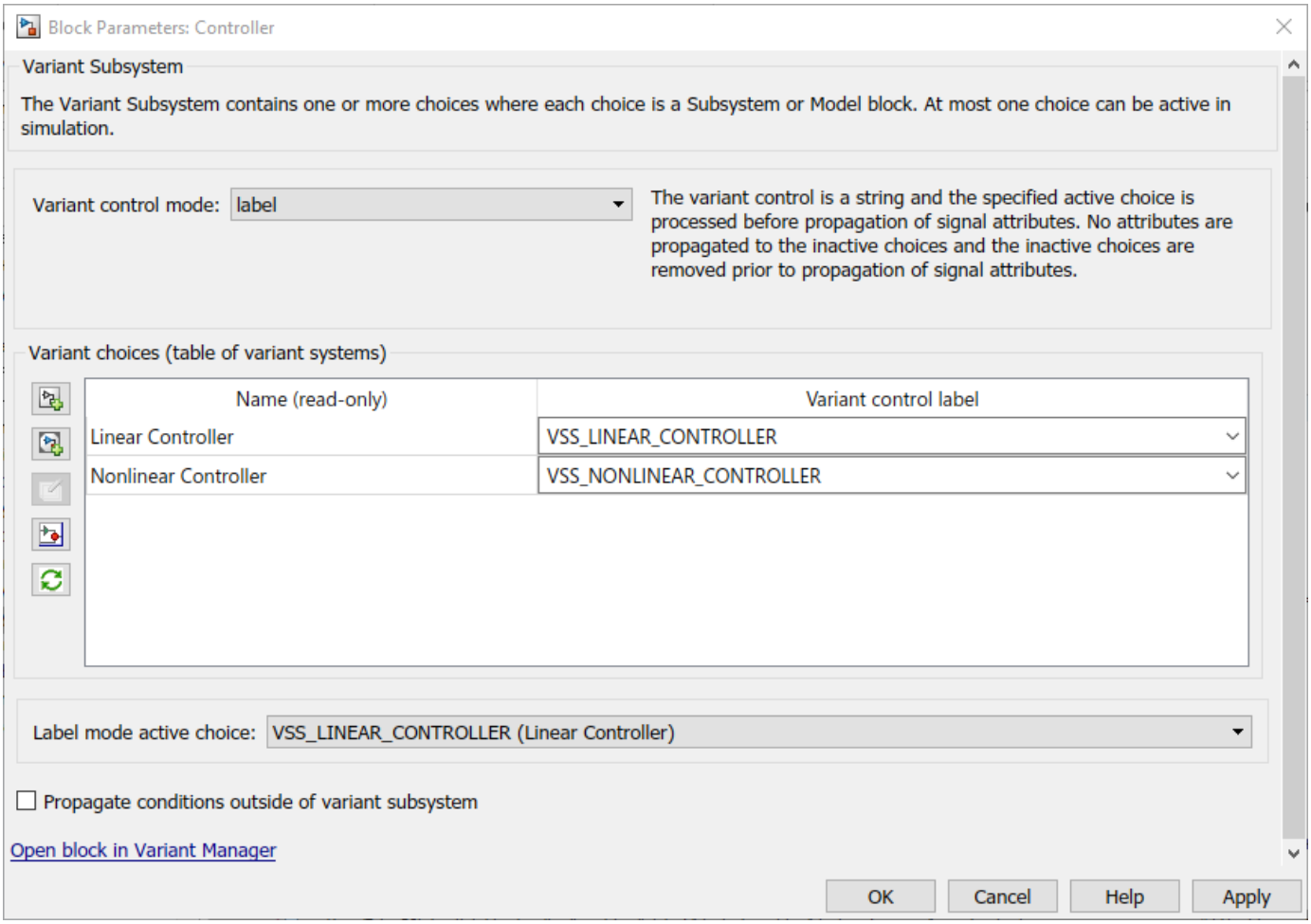
   b. To choose the active Variant based on the evaluation of the Variant conditions, use the `Expression` mode else select `Label` mode. When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. In `Label` mode, Variant control need not be created in the global workspace. You can select an active Variant choice from **Label mode active choice** options.

   c. Use the options available on the Block Parameter dialog box to add variant controls and its corresponding variant condition.

      A sample screenshot for `Expression` mode:

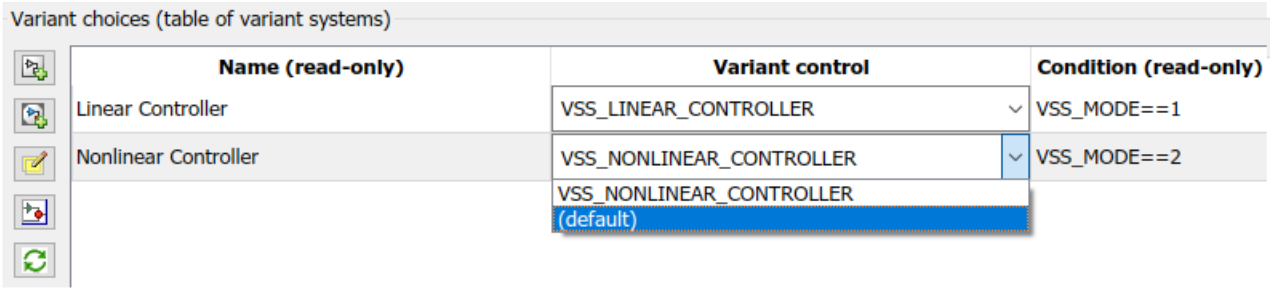


A sample screenshot for `Label` mode:



Note The variables used to specify the variant control and variant condition must be defined in the global workspace, model workspace, mask workspace or data dictionary for the model.

3 Specify a default variant condition to be used when there is no active variant choice. Use the **Variant control** dropdown menu to specify the default.



4 To activate a variant choice, type the variant choice in MATLAB command window. For example, type `VSS_MODE = 2`.

5 To simulate the model, on the toolstrip, click **Run**. The model simulates for the specified active choice.

6. Modify the active choice and simulate the model again, if necessary.
7. Generate code for the variants model with preprocessor conditionals.

   Note You must have an Embedded Coder license to generate code.

   a. In the Block Parameters dialog box, from the **Variant activation time** list, select `code compile`.
   b. Open the Subsystem Block Parameters dialog boxes. Select the **Treat as atomic unit** parameter.
   c. In the **Code Generation** section of Configuration Parameters dialog box, specify the **System target file** as `ert.tlc`.
   d. In Model Explorer, define the variables used to specify the variant choice as a MATLAB variable or as a `Simulink.Parameter`. The data type of the `Simulink.Parameter` can be of type `Integer`, `Boolean`, or `Enumerated` and the storage class can be either `importedDefine(Custom)`, `Define(Custom)`, or `CompilerFlag`.
8. For the variants that are defined in the global workspace, export the control variables to a MAT-file. For example, type the following in the MATLAB command window:

   a. `save <MAT-File Name> <Variable Name>`
   b. `PostLoadCallback > load <MAT-File Name>`

Note To update or refresh active models that contain Variant Subsystem blocks, on the **Modeling** tab, click **Update Model** (**Ctrl + D**) in Simulink.

See Also

Related Examples

- “Define, Configure, and Activate Variants” on page 12-42
- “Create and Validate Variant Configurations” on page 12-69
- “Create Variant Controls Programmatically” on page 12-40
- “Represent Subsystem and Variant Models in Generated Code” (Embedded Coder)
- “Export Workspace Variables” on page 67-116

More About

- Variant System Design

Create Variant Controls Programmatically

In this section...

Create and Export Variant Controls

Create control variables, define variant conditions, and export control variables.

1. Create control variables in the global workspace or a data dictionary.

   ```
   FUEL=2;
   EMIS=1;
   ```

2. Use the control variables to define the control condition using a `Simulink.Variant` object.

   ```
   LinearController=Simulink.Variant('FUEL==2 && EMIS==1');
   ```

   Note Before each simulation, define `Simulink.Variant` objects representing the variant conditions.

3. If you saved the variables in the global workspace, select the control variables to export. Right-click and click **Save As** to specify the name of a MAT-file.

Reuse Variant Conditions

If you want to reuse common variant conditions across models, specify variant control conditions using `Simulink.Variant` objects.

Reuse `Simulink.Variant` objects to change the model hierarchy dynamically to reflect variant conditions by changing the values of the control variables that define the condition expression.

The example model `AutoSSVar` shows the use of `Simulink.Variant` objects to define variant control conditions.

Note You must use `Simulink.Variant` objects to define variant control conditions for AUTOSAR workflows.

Enumerated Types as Variant Controls

Use enumerated types to give meaningful names to integers used as variant control values.

1. In the MATLAB Editor, define the classes that map enumerated values to meaningful names.

   ```
   classdef sldemo_mrv_CONTROLLER_TYPE < Simulink.IntEnumType
           enumeration
           NONLINEAR (1)
           SECOND_ORDER (2)
   ```

```
        end
end

classdef sldemo_mrv_BUILD_TYPE < Simulink.IntEnumType
        enumeration
        PROTOTYPE (1)
        PRODUCTION (2)
        end
end
```

2 Define `Simulink.Variant` objects for these classes in the global workspace.

```
VE_NONLINEAR_CONTROLLER = Simulink.Variant...
('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.NONLINEAR')
VE_SECOND_ORDER_CONTROLLER =Simulink.Variant...
('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.SECOND_ORDER')
VE_PROTOTYPE =Simulink.Variant...
('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PROTOTYPE')
VE_PRODUCTION =Simulink.Variant...
('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PRODUCTION')
```

Using enumerated types simplifies the generated code because it contains the names of the values rather than integers.

See Also

Related Examples

- “Generate Preprocessor Conditionals for Variant Systems” (Embedded Coder)
- “Create and Validate Variant Configurations” on page 12-69

More About

- “Approaches for Specifying Variant Controls” on page 12-27
- Variant System Design

Define, Configure, and Activate Variants

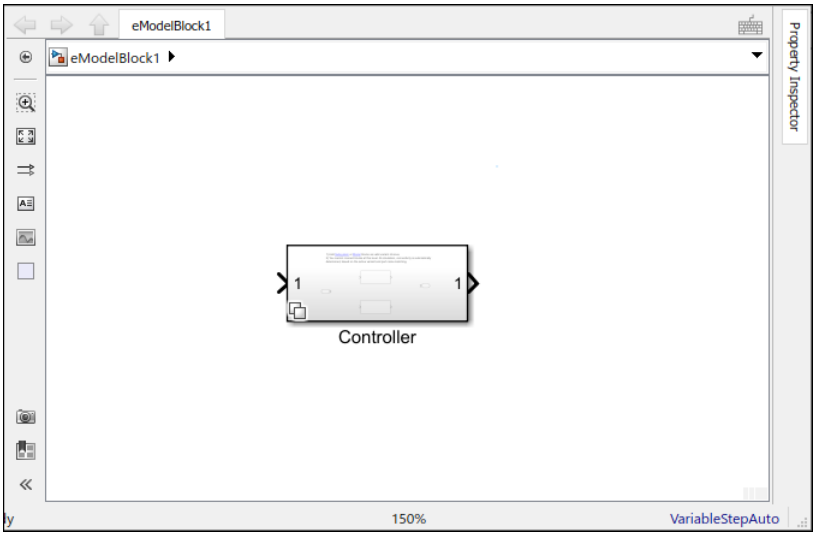
In this section...

Represent Variant Choices

Variant choices are two or more configurations of a component in your model. This example shows how to represent variant choices inside a Variant Subsystem block in your model. For other ways to represent design variants, see “Options for Representing Variants in Simulink” on page 12-5.

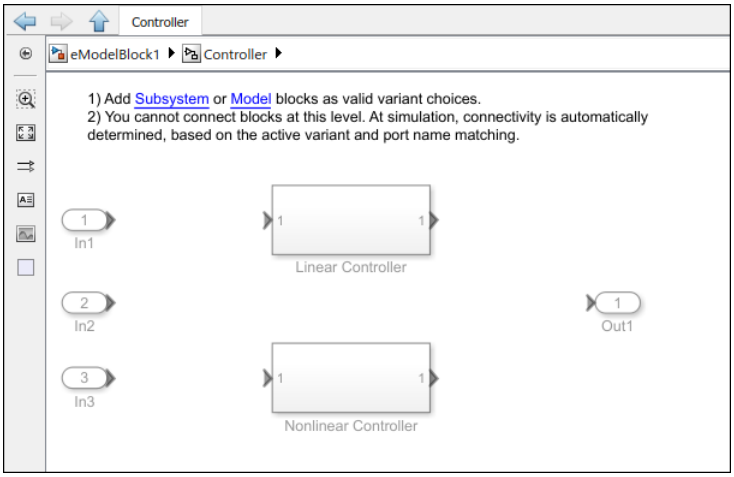
1 Add a Variant Subsystem block to your model and name it.

This block serves as the container for the variant choices.



2 Double-click the Variant Subsystem block. Add Inport and Outport blocks so that they match the inputs into and outputs from this block.

Note You can add only Inport, Outport, Subsystem, and Model blocks inside a Variant Subsystem block. You can pass control signals through data ports.

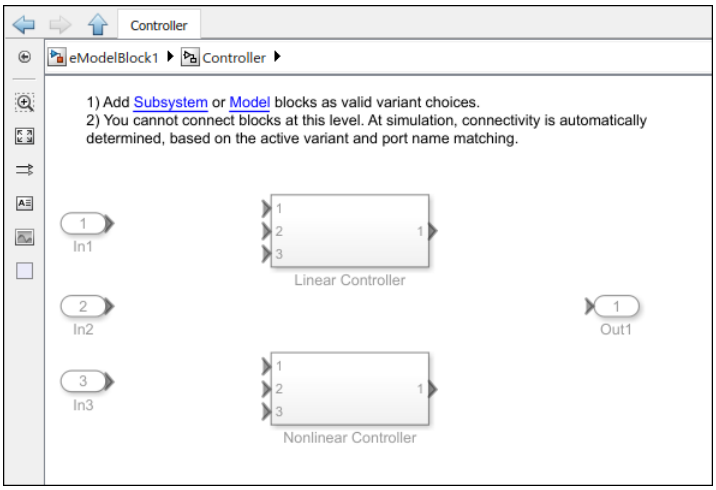


3 Right-click the badge on the Variant Subsystem block and select **Block Parameters (Subsystem)**.

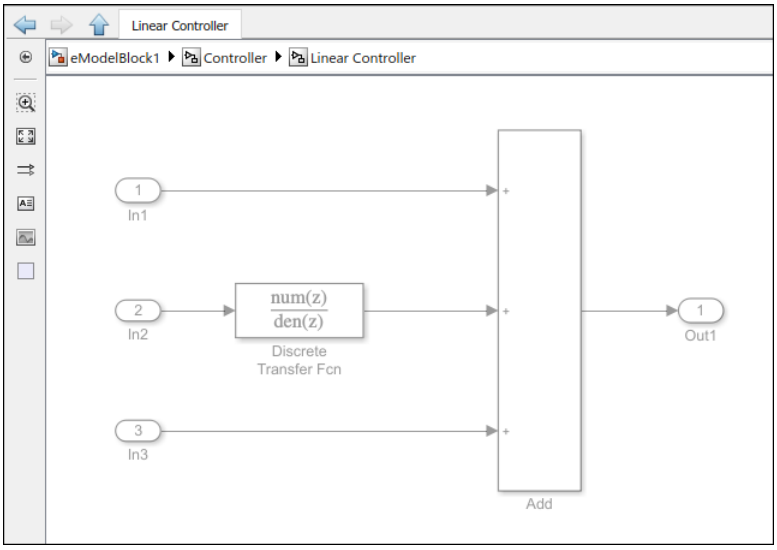
4 In the block parameters dialog box, click the  button for each variant subsystem choice you want to add.

Simulink creates empty Subsystem blocks inside the Variant Subsystem block. The new blocks have the same number of input and output ports as the containing Variant Subsystem block.

Tip (If your variant choices have different numbers of input and output ports, see “Mapping Inports and Outports of Variant Choices” on page 12-5.)

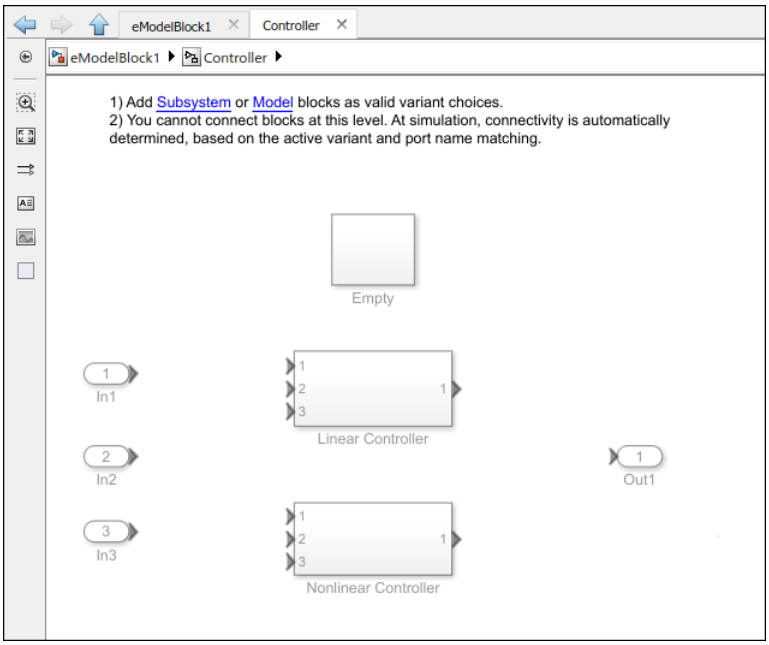


5 Open each Subsystem block and create the model that represents a variant choice.



6 When you are prototyping variants, you can create empty Subsystem blocks with no inputs or outputs inside the Variant Subsystem block. The empty subsystem recreates the situation in which a subsystem is inactive without the need for completely modeling the variant. For an empty variant choice, either specify a variant activation condition or comment out the variant condition by placing a % symbol before the condition.

If the empty variant choice is active during compilation, Simulink ignores it.

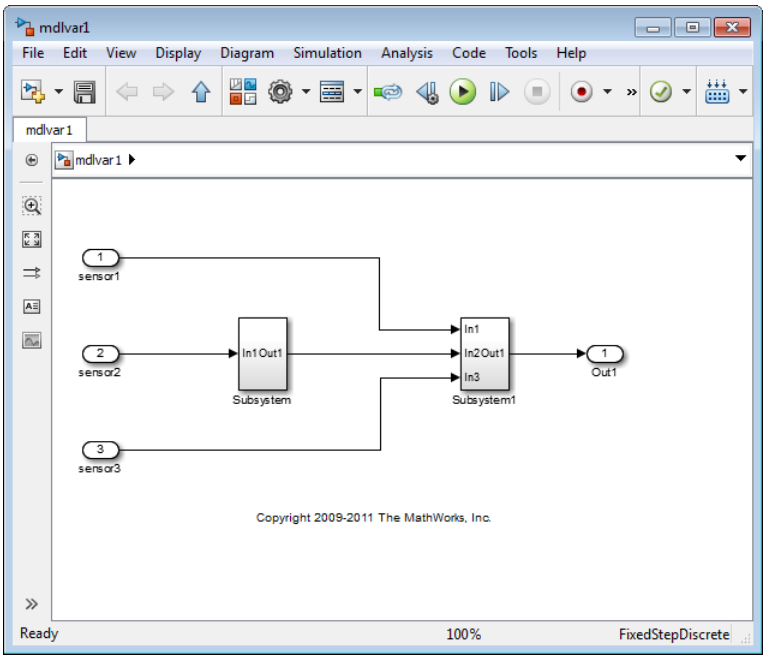


Include Simulink Model as Variant Choice

You can include a Simulink model as a variant choice inside a Variant Subsystem block.

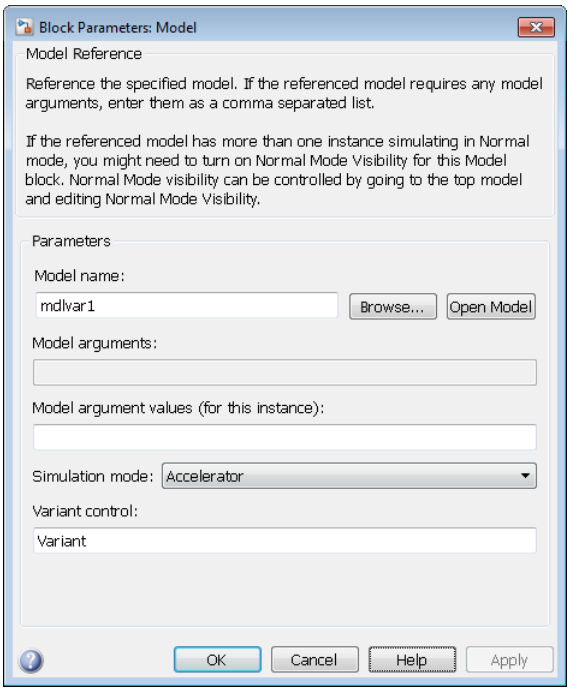
1 Create a model that you want to include as a variant choice. Make sure that it has the same number of input and output ports as the containing Variant Subsystem block.

Note If your model has different numbers of input and output ports, see “Mapping Inports and Outports of Variant Choices” on page 12-5.



2. In your model, right-click the Variant Subsystem block that contains variant choices and select **Block Parameters (Subsystem)**.
3. In the block parameters dialog box, click the  button to add a Model block as variant choice.

   Simulink creates an unresolved Model block in the Variant Subsystem block.
4. Double-click the unresolved Model block. In the **Model name** box, enter the name of the model you want to use as a model variant choice and click **OK**.



Configure Variant Controls

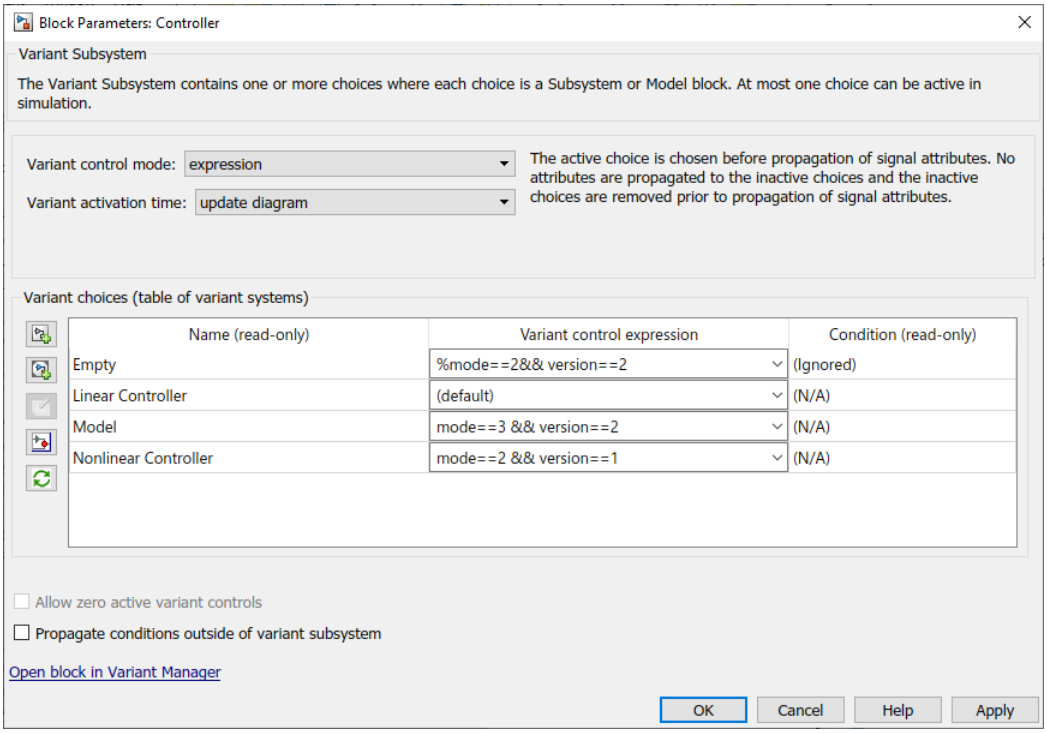
You can specify the conditions for activating a variant choice using variant controls. You can also specify at most one variant choice as the default.

1. At the MATLAB command prompt, specify the control variables that create an activation condition when combined.

   ```
   mode = 3;
   version = 2;
   ```

2. Right-click the Variant Subsystem block that is the container for variant choices in your model and select **Block Parameters (Subsystem)**.
3. In the block parameters dialog box, in the **Variant control** column, select `(default)` next to one of the choices.

   Simulink verifies that only one variant choice is active for simulation. If **Allow zero active variant controls** is selected, you can have zero variant choice. When the control condition does not activate a variant, Simulink uses the default variant for simulation and code generation.



4 Specify a variant condition each of the other choices. If you are using an empty variant choice, specify a variant condition for the choice. You can also comment out an existing activation condition by prefixing it with a `%` symbol.

5 Click **Apply**; otherwise, your changes are not saved.

Convert to Variants

In the Simulink Editor, you can convert these blocks to a Variant Subsystem block:

- Subsystem block
- Model block
- Variant Model block (for models created in versions earlier than R2017b)
- Conditionally executed subsystems

To do so, right-click the block, then in the context menu, click **Subsystem & Model Reference > Convert to > Variant Subsystem**.

You can also convert these block to Variant Subsystem block programmatically. To do so, use any of these syntaxes:

- `Simulink.VariantManager.convertToVariant(gcb)`
- `Simulink.VariantManager.convertToVariant(gcbh)`

For example,

```
open_system('sldemo_variant_subsystems');
Simulink.VariantManager.convertToVariant('sldemo_variant_subsystems/Controller');
```

If you convert variant models to variant subsystem, note that the behavior of the Model block parameter **Generate preprocessor conditionals** is different than the Variant Subsystem block

parameter **Variant activation time**. For variant models, enabling the parameter causes simulation and update diagram to compile the active variant only. For variant subsystem, enabling the parameter compiles all the variants, which can make simulation and updates slower.

Converting variant models to variant subsystems can require that you update scripts that use the `Variants` command-line parameter.

See Also

Related Examples

- “Create and Validate Variant Configurations” on page 12-69

More About

- “Introduction to Variant Controls” on page 12-24
- “Approaches for Specifying Variant Controls” on page 12-27
- Variant System Design

Prepare Variant-Containing Model for Code Generation

In this section...

Using Embedded Coder, you can generate code from Simulink models containing one or more variant choices. The generated code contains preprocessor conditionals that control the activation of each variant choice.

Note Simulink supports using multi-instance referenced models with variant Simulink Functions for code generation.

For information on using `STF_make_rtw_hook` file to customize build process, see “Customize Build Process with STF_make_rtw_hook File” (Simulink Coder)

Convert Variant Control Variables into Simulink.Parameter Objects

MATLAB variables allow you to rapidly prototype variant control expressions when you are building your model and generate preprocessor conditionals for code generation. However, if you want to specify other code generation attributes (such as data type), you can convert MATLAB variables into `Simulink.Parameter` objects.

1. Specify the model in which you want to replace MATLAB variant control variables with `Simulink.Parameter` objects.

```
model = 'my_model_containing_variant_choices';
open_system(model);
```

2. Get the variables that are referenced in variant control expressions.

```
vars = Simulink.VariantManager.findVariantControlVars(model)

vars =

4x1 struct array with fields:

    Name
    Value
    Exists
    Source
    SourceType
```

3. Create an external header file for specifying variant control values so that the variable definitions are imported when the code runs.

```
headerFileName = [model '_importedDefines.h'];
headerPreamble = strrep(upper(headerFileName),'.','_');

fid = fopen(headerFileName,'w+');
fidErr = (fid == -1);
if (fidErr)
    fprintf('There was an error creating header file %s:...
\n', headerFileName);
```

```
else
    fprintf('+++ Creating header file ''%s'' with variant control...
        variable definitions.\n\n', headerFileName);
    fprintf(fid, '#ifndef %s\n', headerPreamble);
    fprintf(fid, '#define %s\n', headerPreamble);
end
```

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes.

- `Define` or `ImportedDefine` with header bfile specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- Your own storage class that defines data as a macro

4 Loop through all the MATLAB variables to convert them into `Simulink.Parameter` objects.

```
count = 0;
for countVars = 1:length(vars)
    var = vars(countVars).Name;
    val = vars(countVars).Value;
    if isa(val, 'Simulink.Parameter')
        % Do nothing
        continue;
    end
    count = count+1;

% Create and configure Simulink.Parameter objects
% corresponding to the control variable names.
% Specify the storage class as Define (Custom).
    newVal = Simulink.Parameter(val);
    newVal.DataType = 'int16';
    newVal.CoderInfo.StorageClass = 'Custom';
    newVal.CoderInfo.CustomStorageClass = 'Define (Custom)';
    newVal.CoderInfo.CustomAttributes.HeaderFile = headerFileName;

        Simulink.data.assigninGlobal(model, var, newVal);

        if ~fidErr
            fprintf(fid, '#endif\n');
            fclose(fid);
        end
end
```

Note The header file can be empty for the `Define` storage class.

Configure Model for Generating Preprocessor Conditionals

If you represent variant choices inside a Variant Subsystem block or a Variant Model block, code generated for each variant choice is enclosed within C preprocessor conditionals `#if`, `#else`, `#elif`, and `#endif`.

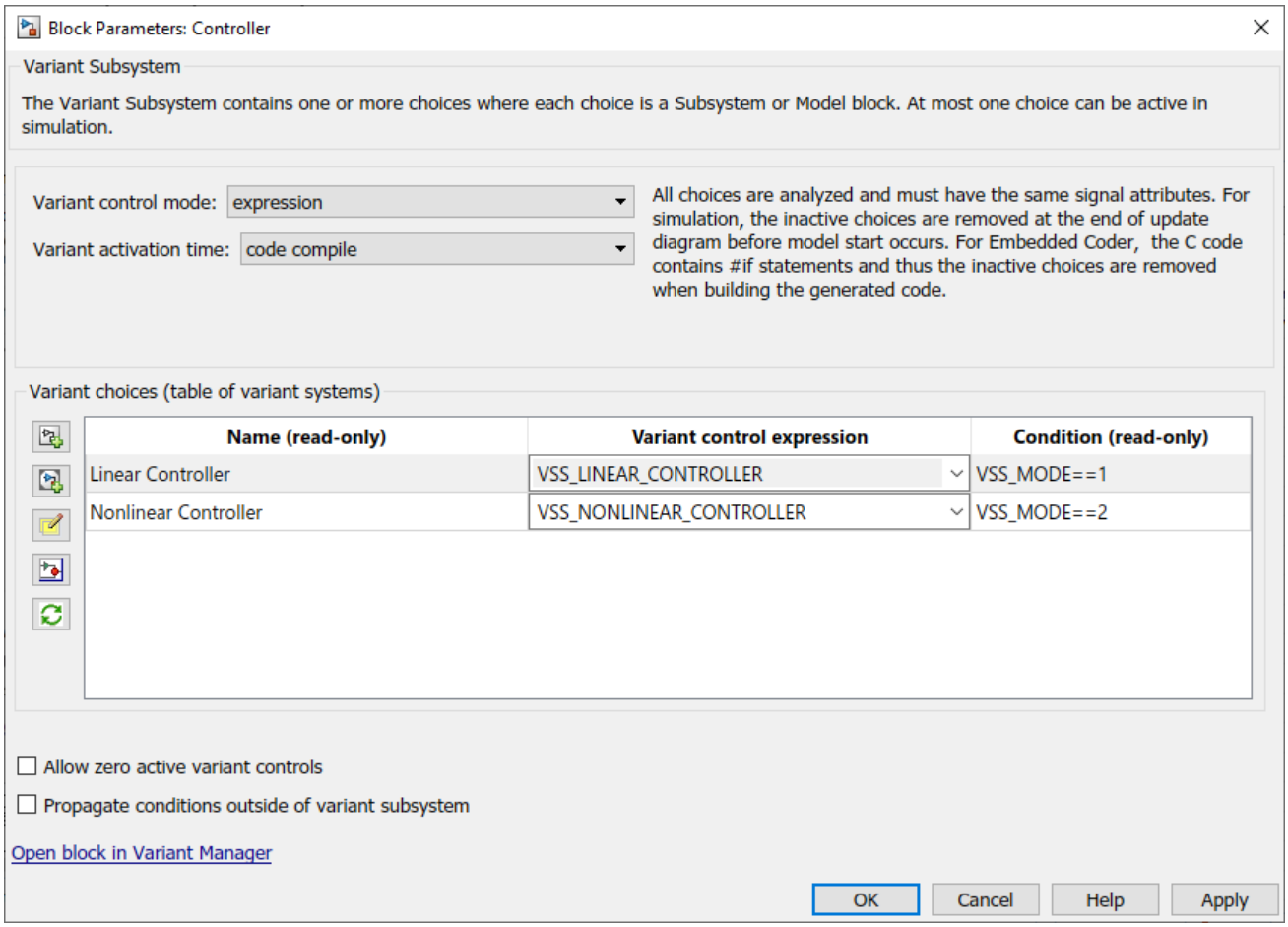
If you represent variant choices using a Variant Source block or a Variant Sink block, code generated for each variant choice is enclosed within C preprocessor conditionals `#if` and `#endif`.

Therefore, the active variant is selected at compile time and the preprocessor conditionals determine which sections of the code to execute.

Note You must have an Embedded Coder® license to generate code.

1. In the **Modeling** tab of the Simulink toolstrip, click **Model Settings**.
2. Select the **Code Generation** pane, and set **System target file** to `ert.tlc`.
3. In the **Report** pane, select **Create code generation report**.

   Note In the **Code Placement** pane, if `Compact` option is selected from **File packaging format** drop-down list, `model_types.h` file is not generated and contents of `model_types.h` file are moved to `model.h` file.
4. Select the **Code Generation** pane, and clear **Ignore custom storage classes** and **Apply**.
5. In your model, right-click the block containing the variant choices (Variant Subsystem, Variant Source, Variant Sink, or Variant Model block) and select **Block Parameters**.
6. Ensure that `Expression` (default option) is selected for **Variant control mode** parameter .
7. From the **Variant activation time** list, select `code compile`.



Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness for all variant choices.

8 Build the model.

See Also

Related Examples

- “Define, Configure, and Activate Variants” on page 12-42
- “Create and Validate Variant Configurations” on page 12-69
- “Create Variant Controls Programmatically” on page 12-40
- “Working with Variant Choices” on page 12-21

More About

- “Code Generation for Variant Blocks” (Embedded Coder)
- “Represent Subsystem and Variant Models in Generated Code” (Embedded Coder)
- “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
- “Model AUTOSAR Variants” (AUTOSAR Blockset)
- Variant System Design

Visualize Variant Implementations in a Single Layer

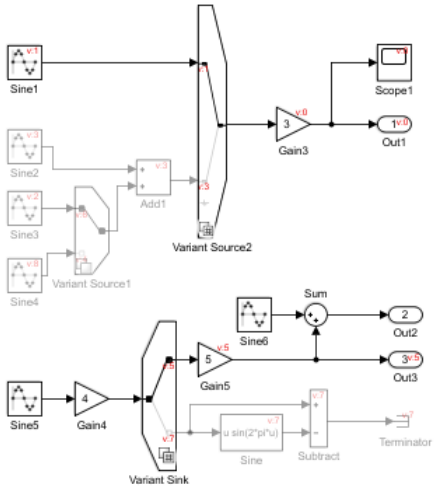
Simulink provides two blocks that you can use to propagate conditions throughout the model and visualize all possible implementations of variant choices in a model. These blocks are called Variant Source and Variant Sink.

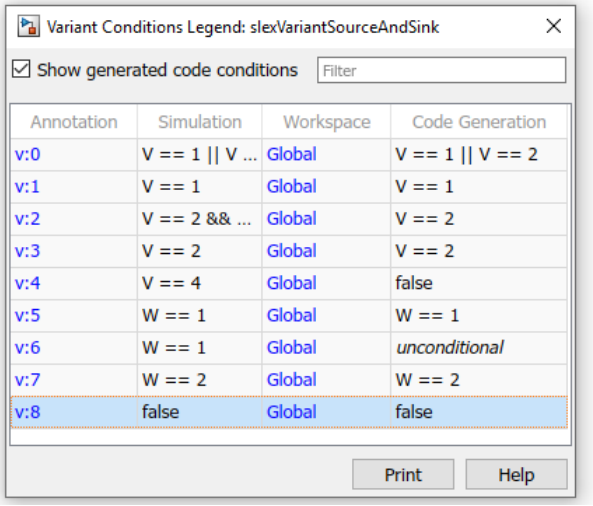
When you compile the model, Simulink determines which variant control evaluates to `true`. Simulink then deactivates blocks that are not tied to the variant control being `true` and visualizes the active connections.

How Variant Sources and Sinks Work

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. The active choice is connected directly to the output port of the Variant Source and the inactive choices are eliminated during simulation.

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. The active choice is connected directly to the input port of the Variant Sink, and the inactive choices are eliminated during simulation.





Connect one or more blocks to the input port of the Variant Source block or the output port of the Variant Sink block. Then, you define variant controls for each variant choice entering the Variant Source block and exiting the Variant Sink block. For more information, see “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60.

Advantages of Using Variant Sources and Sinks

Using Variant Source and Variant Sink blocks in Model-Based Design provides these advantages:

- The blocks enable the propagation of variant conditions throughout the model and allow you to visualize variant choices in a single layer of your model.
- By visualizing all possible implementations of variant choices, you can improve the readability of your model.

- During model compilation, Simulink eliminates inactive blocks throughout the model, improving the runtime performance of your model.
- Variant sources and sinks provide variant component interfaces that you can use to quickly model variant choices.

Limitations of Using Variant Sources and Sinks

- Variant Source and Variant Sink blocks work with time-based, function-call, or action signals. You cannot use SimEvents, Simscape Multibody, or other non-time-based signals with these blocks.
- The code generation variant report does not contain Variant Source and Variant Sink blocks.

See Also

Related Examples

- “Define and Configure Variant Sources and Sinks” on page 12-55

More About

- “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60
- Variant System Design

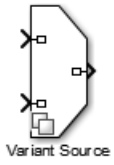
Define and Configure Variant Sources and Sinks

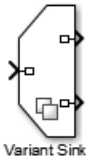
Simulink provides two blocks that you can use to visualize all possible implementations of variant choices in a model graphically. These blocks are called Variant Source and Variant Sink.

When you compile the model, Simulink determines which variant control evaluates to `true`. The active variant determination happens early stages of compilation. Simulink then deactivates blocks that are not tied to the variant control being `true` and visualizes the active connections.

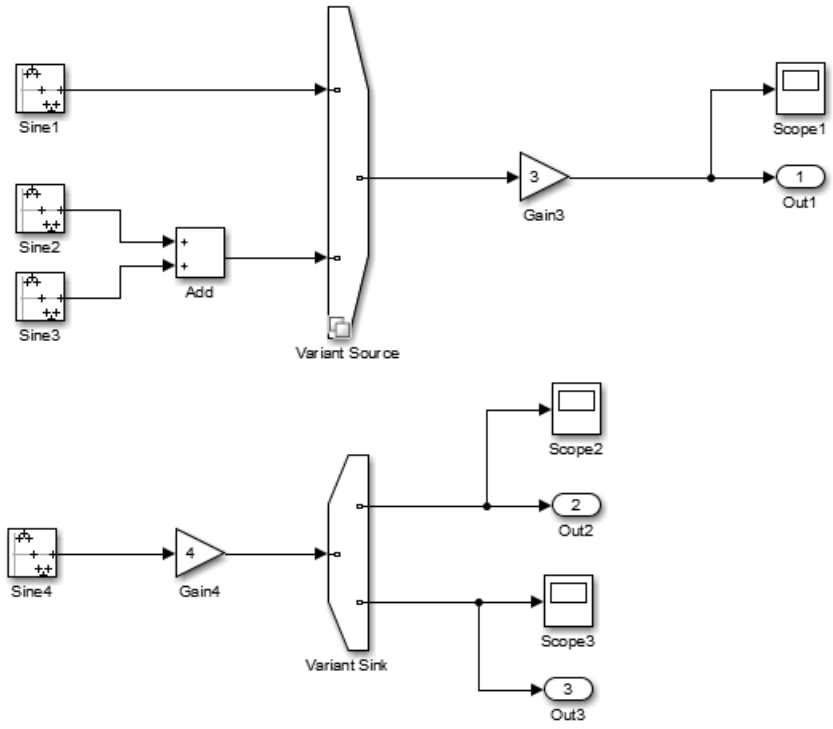
1. Add Variant Source and Variant Sink blocks to your model.

   These blocks enable ports that activate variant choices.





2. Using blocks from the Simulink Library Browser, create sources and sinks that represent variant choices. Connect choices to the input and output ports of the Variant Source and Variant Sink blocks.



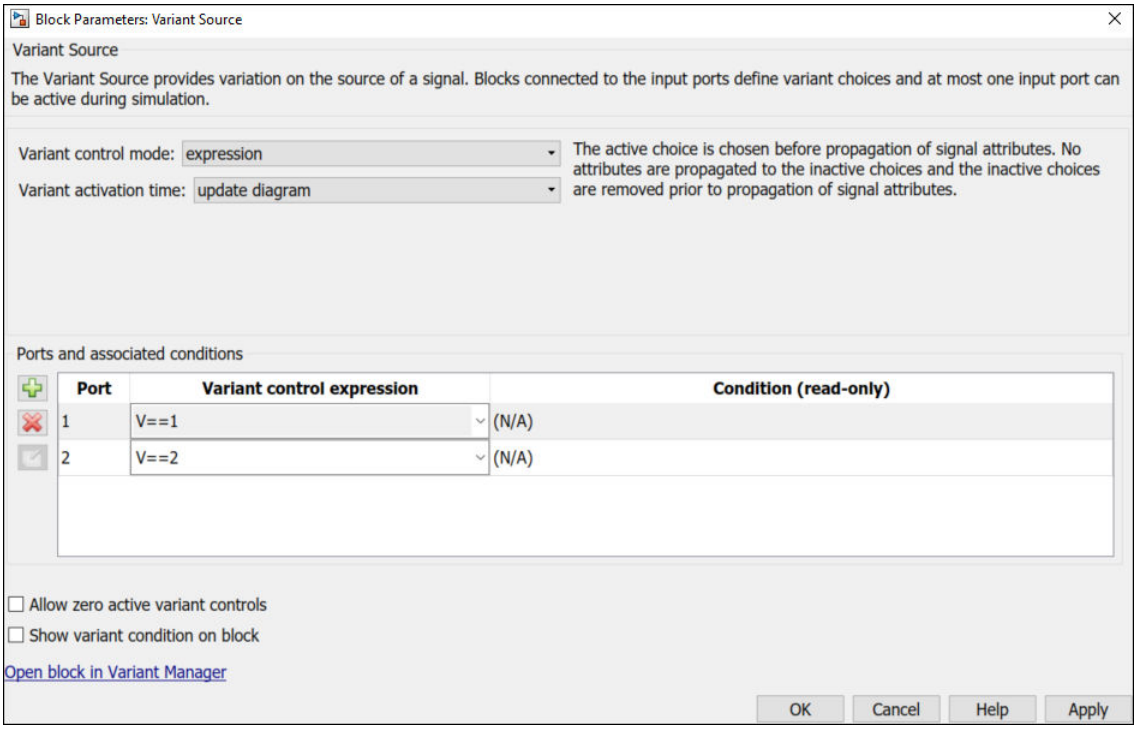
3 At the MATLAB command prompt, specify the control variable that creates an activation condition for the variant source.

```
V = Simulink.Parameter(1);
```

4 Right-click the Variant Source block and select **Block Parameters (VariantSource)**.

5 In the block parameters dialog box, in the **Variant control** column, type `V==1` next to one of the choices and `V==2` next to the other. Click **Apply**; otherwise, your changes are not saved.

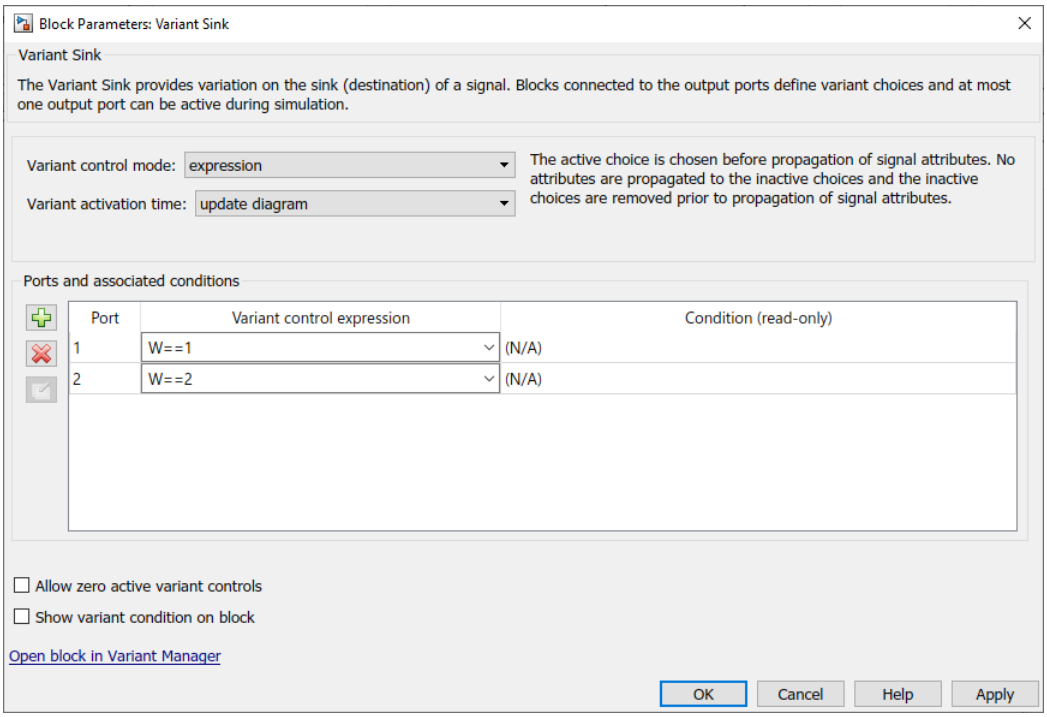
Simulink verifies that only one variant is active for simulation. When the control condition does not activate a variant, Simulink uses the default variant for simulation.



6 At the MATLAB command prompt, specify the control variable that creates an activation condition for the variant sink.

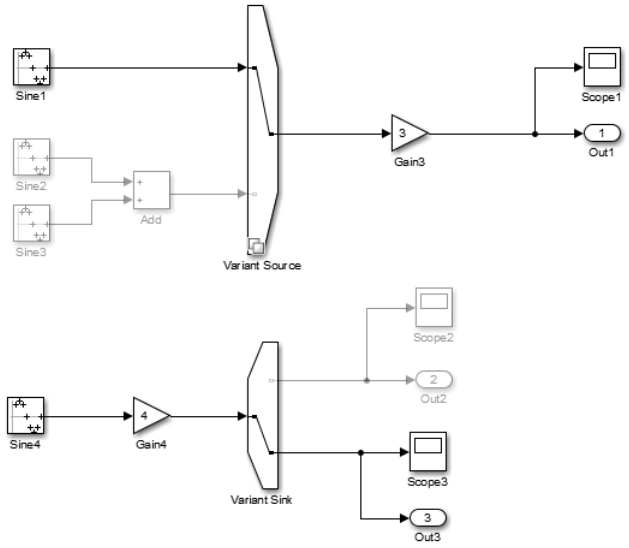
```
W = Simulink.Parameter(2);
```

7 Double-click the Variant Sink. In the block parameters dialog box, in the **Variant control** column, type `W==1` next to one of the choices and `W==2` next to the other.

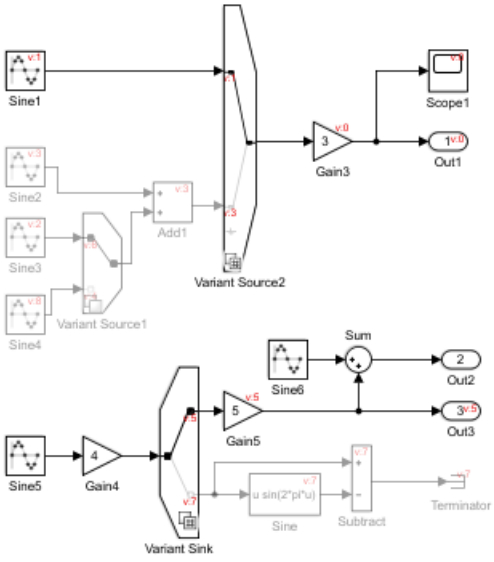


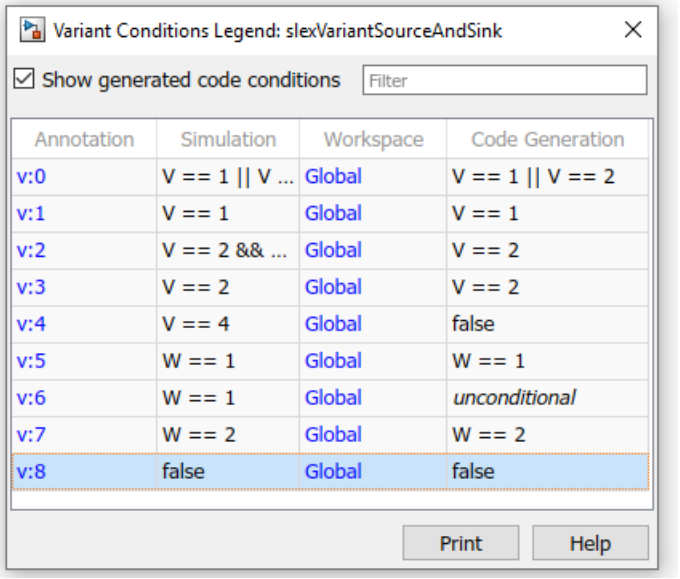
8 Click **Apply**; otherwise, your changes are not saved.

9 Simulate the model. Simulink propagates the variant conditions to identify which model components to activate.

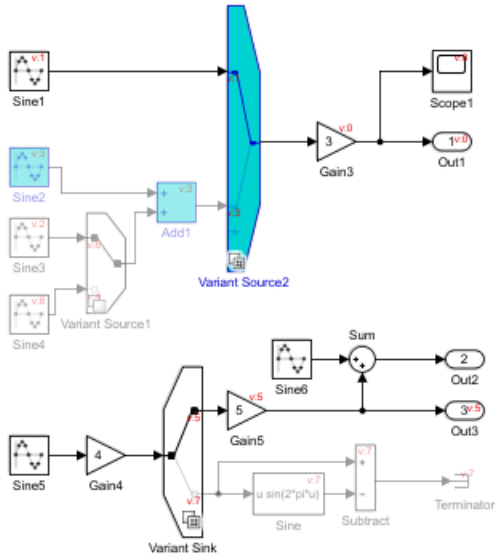


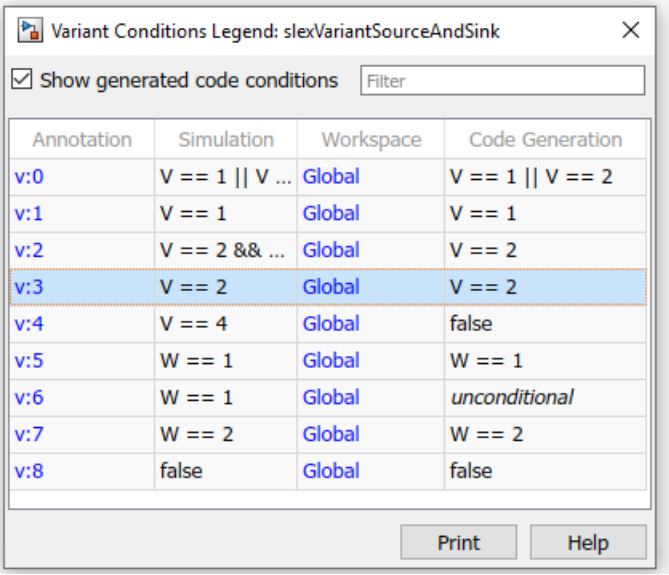
10 You can visualize the conditions that activate each variant choice. In the **Debug** tab of toolstrip, select **Information Overlays > Variant Conditions**.





11 In the Variant Condition Legend dialog box, click through the hyperlinked variant condition annotations to observe which parts of the model each condition activates.





See Also

Related Examples

- “Visualize Variant Implementations in a Single Layer” on page 12-53
- “Working with Variant Choices” on page 12-21

More About

- “Introduction to Variant Controls” on page 12-24
- “Create a Simple Variant Model” on page 12-36
- Variant System Design

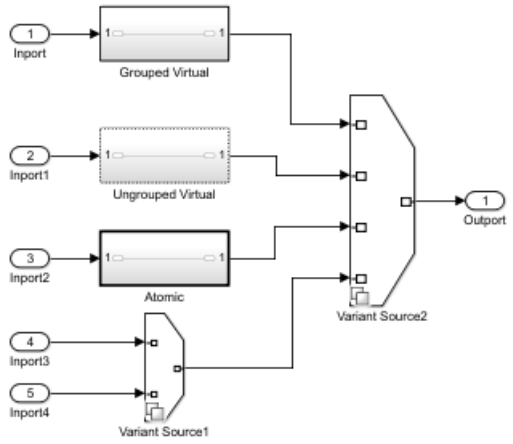
Variant Condition Propagation with Variant Sources and Sinks

When you specify variant conditions in models containing Variant Source and Variant Sink blocks, Simulink propagates these conditions to determine which components of the model are active during simulation.

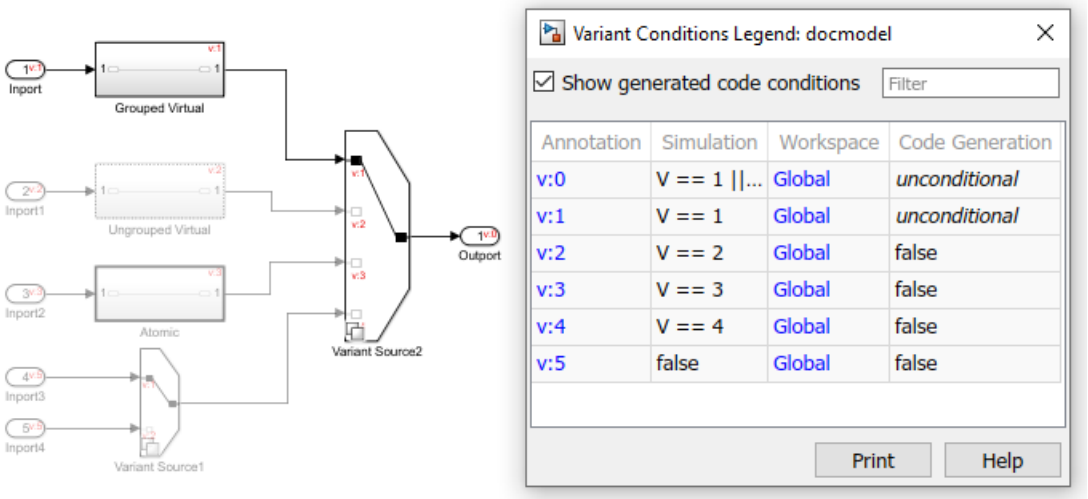
View Variant Condition Annotations

When you create a model that contains Variant Source or Variant Sink blocks, you can visualize the conditions that activate each variant choice. Simulink annotates these blocks in the model with their corresponding variant conditions.

Consider this model containing multiple variant choices feeding into Variant Source blocks. A specific variant condition activates each variant choice.

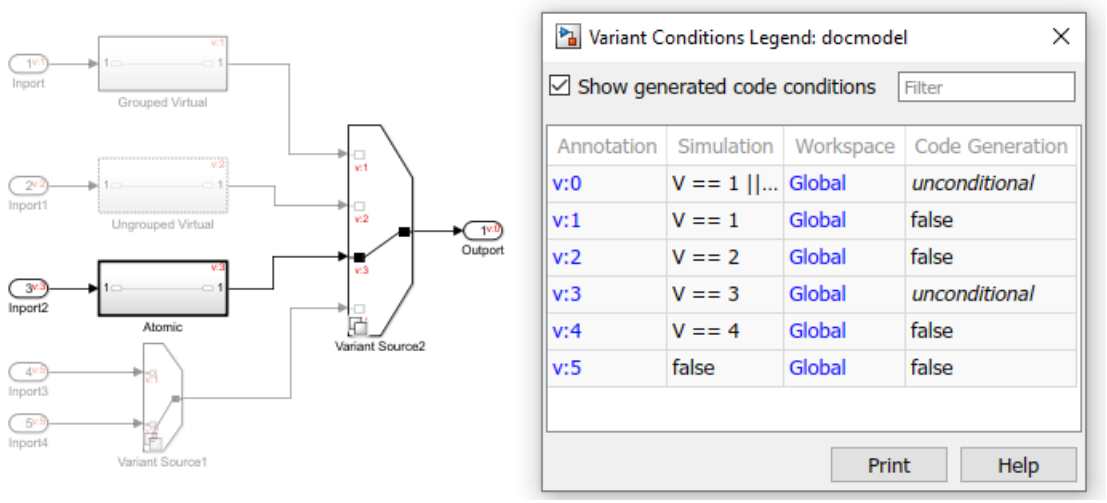


To visualize the variant conditions, on the **Debug** tab of toolstrip, select **Information Overlays** > **Variant Conditions**.



The **Variant Condition Legend** dialog box appears. Variant conditions on blocks are annotated as `v:C`, where `v` is the variant semantic indicator and `C` represents the variant condition index. The dialog box also shows the expression associated with each condition.

In the **Variant Condition Legend** dialog box, you can click through the hyperlinked variant annotations to observe which parts of the model each condition activates. For example, if you click `v:3`, Simulink highlights the parts of the model that are activated when the condition `V==3` evaluates to `true`.



Variant condition annotations have these properties:

- There are no annotations on unconditional blocks. Therefore, the `Out` block is not annotated.
- To reduce clutter, the legend only displays the final computed conditions. For example, if you enter a variant condition in a Variant Source block, that condition appears in the annotations only when you apply your changes.
- The conditions in the legend are sorted during display.
- In the legend, a condition is set to `false` if Simulink assesses that the blocks associated with that condition are never active.

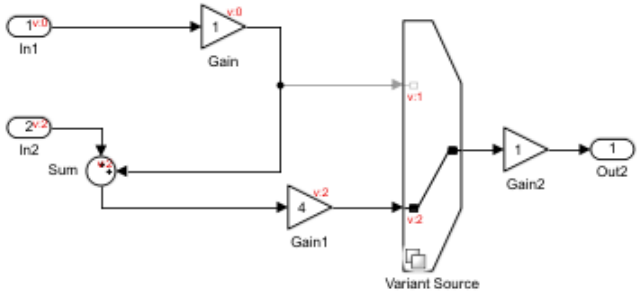
  For example, the `Inport4` block is connected to the `Variant Source1` block, whose condition is `V==1`. `Variant Source1` is connected to the `Variant Source2` block, which activates `Variant Source1` only when `V==4`. Therefore, `Inport4` can only be active when `V==1 && V==4`, a condition that is always `false`.
- In the legend, the `(default)` keyword is displayed as negated condition.

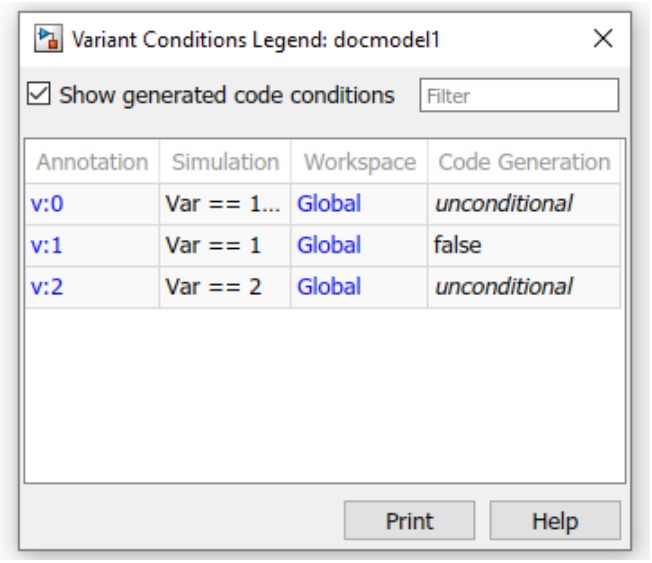
How Variant Condition Propagation Works

When you compile a model containing Variant Source or Variant Sink blocks, Simulink determines which variant control evaluates to `true`. The active variant determination happens in the early stage of compilation. Simulink then deactivates blocks that are not tied to the variant control being `true` and visualizes the active connections.

Cross-Coupling of Inputs and Outputs

In this model, two inputs feed the Variant Source block. The first input is active when `Var == 1`, and it branches into the second input before connecting to the Variant Source block. The second input is the default variant choice.



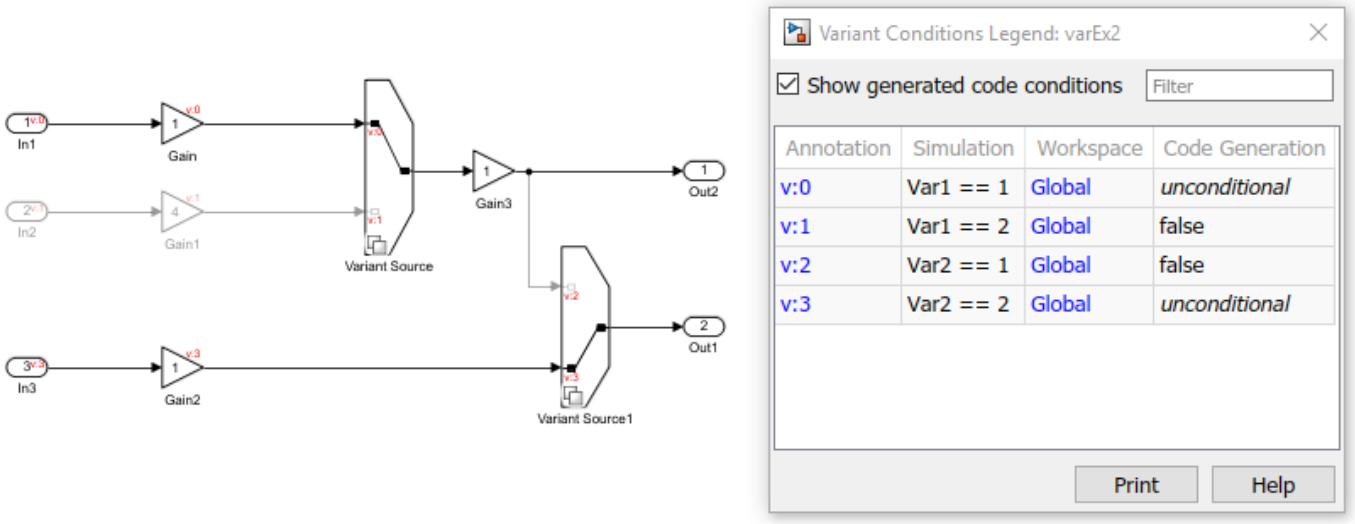


During simulation, this model exhibits three modes of operation:

- When `Var == 1`, the first input is active and its branch into the second input is inactive.
- When `Var == 1 || Var == 2`, the second input is active and the branch of the first input is active.
- When `Var == 2`, the second input is active and the output is active.

Cascading Blocks and Compounding Conditions

In this model, two Variant Source blocks, each fed by two input ports, are connected in a cascading manner. The inputs to `Variant Source` are active when `Var1 == 1` or `Var1 == 2`. The output of `Variant Source` branches into one of the inputs of `Variant Source1`. The inputs to`Variant Source1` are active when `Var2 == 1` or `Var2 == 2`.



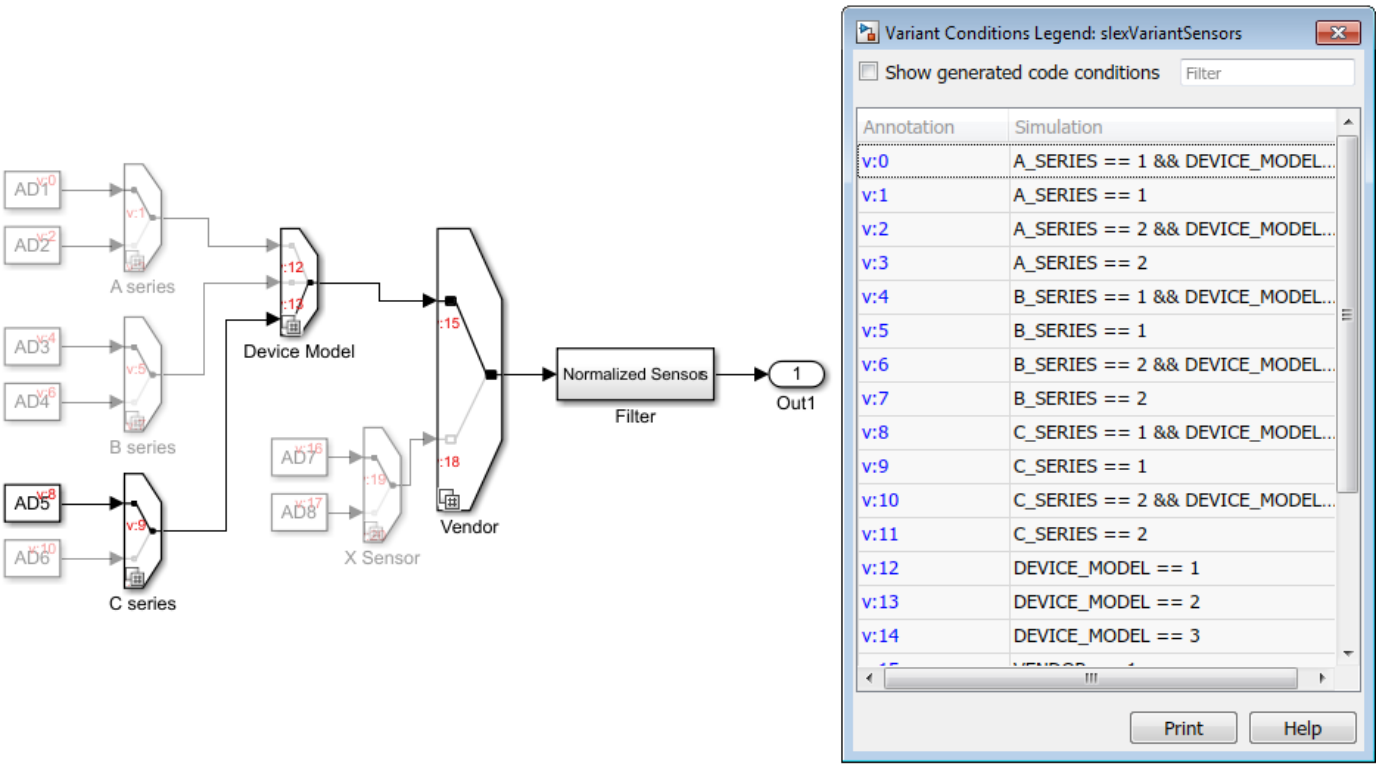
During simulation, this model exhibits eight modes of operation:

- When `Var1 == 1 && Var2 == 1`, the first inputs of `Variant Source` and `Variant Source` are active.
- When `Var1 == 1 && Var2 == 2`, the first input of `Variant Source` and the second input of `Variant Source1` are active.
- When `Var1 == 2 && Var2 == 1`, the second input of `Variant Source` and the first input of `Variant Source1` are active.
- When `Var1 == 2 && Var2 == 2`, the second inputs of `Variant Source` and `Variant Source1` are active.
- When `Var1 == 1 && Var2 != (1,2)`, only the first input of `Variant Source` is active.
- When `Var1 == 2 && Var2 != (1,2)`, only the second input of `Variant Source` is active.
- When `Var1 != (1,2) && Var2 == 1`, none of the inputs or outputs is active.
- When `Var1 != (1,2) && Var2 == 2`, only the second input of `Variant Source1` is active.
- When `Var1 != (1,2) && Var2 != (1,2)`, none of the inputs or outputs is active.

Hierarchical Nesting of Sources or Sinks

In this model, multiple Variant Source blocks are used to create hierarchical nesting of variant choices. Choices are first grouped by series: `A Series`, `B Series`, and `C Series`. A combination of one or more series is provided as input for a device model. The resulting device model is provided as input to the vendor by including or excluding a sensor selection.

Simulink propagates complex variant control conditions to determine which model components are active during compilation.



For more information, see Variant Sensors.

Condition Propagation with Subsystems

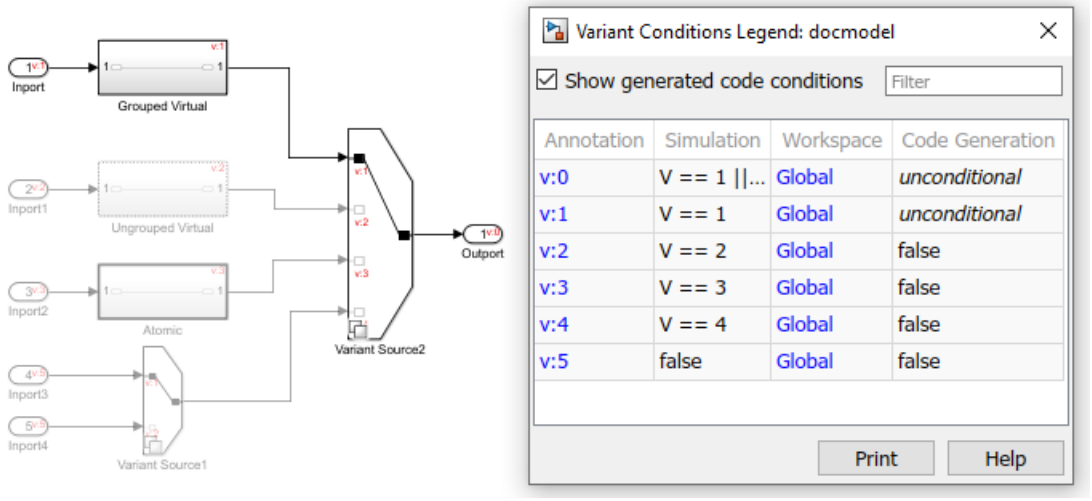
A subsystem can either be a virtual (grouped or ungrouped) or an atomic subsystem depending on the selections made in its **Block Parameters** dialog box. For,

- Grouped Virtual: Select the **Treat as grouped when propagating variant conditions** check box. A grouped virtual subsystem has a continuous line.
- Ungrouped Virtual: Clear the **Treat as grouped when propagating variant conditions** check box. An ungrouped virtual subsystem has a dotted line.
- Atomic: Select the **Treat as atomic unit** check box. An atomic virtual subsystem has a solid line.

Simulink propagates variant conditions differently to these Subsystem types.

In this model, three types of subsystems are provided as input to the block `Variant Source2`.

- The grouped virtual subsystem is activated when `V == 1`. Simulink propagates the variant activation condition to all the blocks in the subsystem.
- The ungrouped virtual subsystem is activated when `V == 2`. Simulink propagates the variant activation condition to the blocks that were available in the subsystem while marking the subsystem virtual.
- The atomic subsystem is activated when `V == 3`. Simulink does not propagate the variant activation condition into this subsystem.



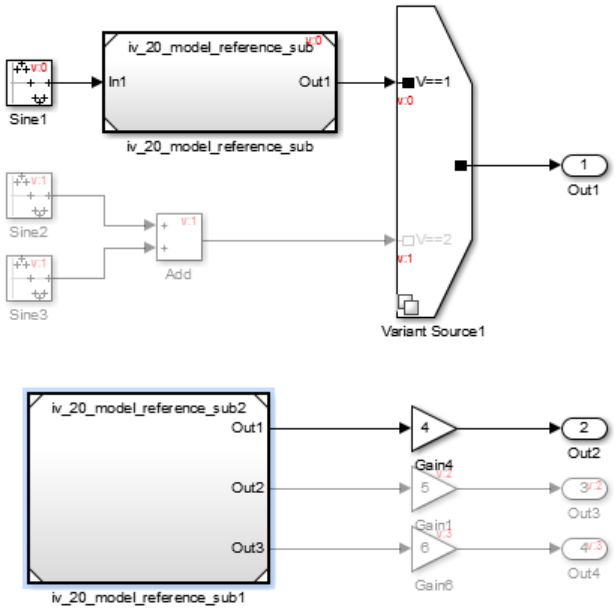
For more information, see “Propagating Variant Conditions to Subsystems” on page 12-132.

Condition Propagation with Other Simulink Blocks

Variant Condition Propagation with Model Block

Simulink compiles referenced models before propagating variant conditions. A variant condition can activate or deactivate a Model block, but variant conditions cannot propagate into the referenced model. A Model block can propagate variant conditions from its interface (input, output, or control port), if that variant condition originates at a port inside the model.

In this example, variant condition `V==1` activates the Model block `iv_20_model_reference_sub`. However, the condition does not propagate into the model referenced by the block. Model block `iv_20_model_reference_sub2` propagates the same variant condition from its output port.

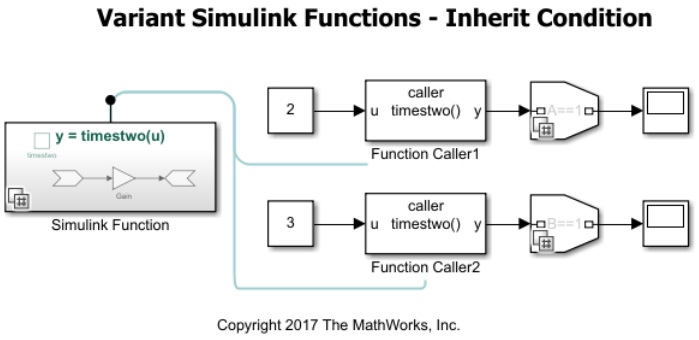


Variant Condition Propagation with Simulink Function block

Argument Inport and Argument Outport blocks interfacing with Simulink Function blocks cannot be connected to Variant Source or Variant Sink blocks. One variant condition must control the entire Simulink Function.

Consider the model slexVariantSimulinkFunctionInherit.

In this example, the function-call port block within the Simulink Function block has the **Enable variant condition** option selected. The `(inherit)` keyword is used to specify the value for the **Variant control** parameter. As a result, the Simulink Function block inherits the variant condition from the corresponding Function Caller blocks in the model. The **Generate preprocessor conditionals** parameter value is also inherited.



Note Use the Configure C Step Function Interface dialog box to customize the generated C entry-point step function interface for a model. If input and output ports share an argument name and have propagated variant conditions, this level of interface control is not supported.

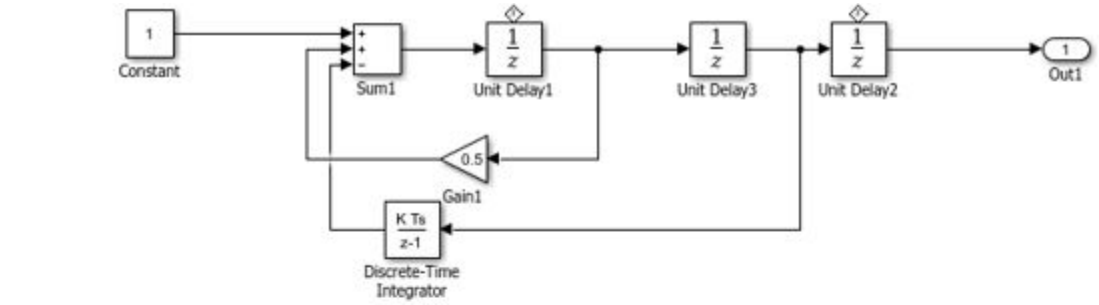
Variant Condition Propagation with Initialize, Reset, and Terminate Blocks

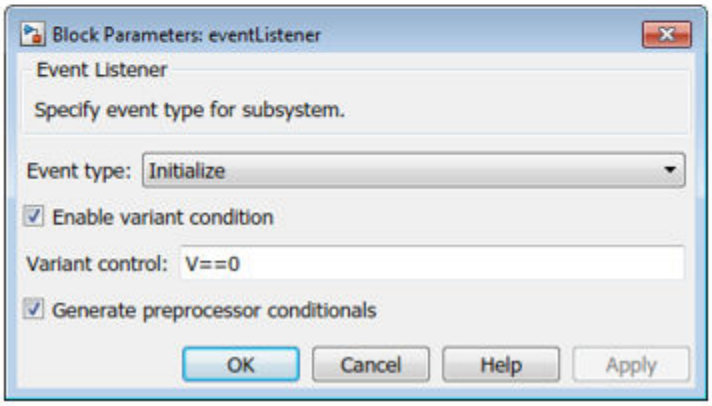
The Initialize, Reset, and Terminate function blocks are pre-configured subsystem blocks that execute during model initialize, reset, and terminate events. Similar to a Simulink Function block these blocks support variant condition propagation. You can propagate Variant conditions in Model blocks that have Reset Event ports. This results in optimized existence of blocks connected to the Reset Event ports. Models with inactive Variant Reset Event functions (in referenced models) also supports Variant condition propagation.

Note

- Initialize and Terminate event ports are always unconditional because they control both the model default and block-specific initialize and terminate events of the referenced model. If you define an Initialize / Terminate function block in the referenced model, it corresponds to an explicit initialize / terminate event.
- If you enable variants and define a variant condition on the Initialize / Terminate function block in the referenced model, the variant condition will not contribute to the model reference block instance’s variant condition.

In this example, the `Event Listener` block within the `Init`, `Reset`, and `Term` blocks have the **Enable variant condition** option selected. The **Variant control** parameter of the `Event Listener` block is specified as `V==0`. If you change the value of `V` to any value other than `0`, the `Init`, `Reset`, and `Term` blocks become inactive.





Variant Condition Propagation with Subsystem Block

A variant condition can activate or deactivate a Subsystem block, but variant conditions cannot propagate into the subsystem. A Subsystem block can propagate variant conditions from its output port if that variant condition originates at a port inside the subsystem.

For more information, see “Propagating Variant Conditions to Subsystems” on page 12-132.

Variant Condition Propagation with Bus

A Variant Source block can accept either virtual or nonvirtual bus inputs. When generating code with preprocessor conditionals, the bus types and hierarchies of all bus inputs must be the same.

However, all elements of a Mux, Demux, or a Vector Concatenate block signal must have the same variant condition.

For more information, see “Variant Condition Propagation with Bus” on page 12-104.

Known Limitations

- Variant condition propagation from Simulink Function inside Stateflow block is not supported.
- When you simulate an Inline variants model with Simscape blocks, the Simscape blocks become unconditional.
- C++ code generation is not supported for models that contain propagated variant conditions.
- Variant condition propagation is not supported with root bus element ports.

See Also

Related Examples

- “Define and Configure Variant Sources and Sinks” on page 12-55

More About

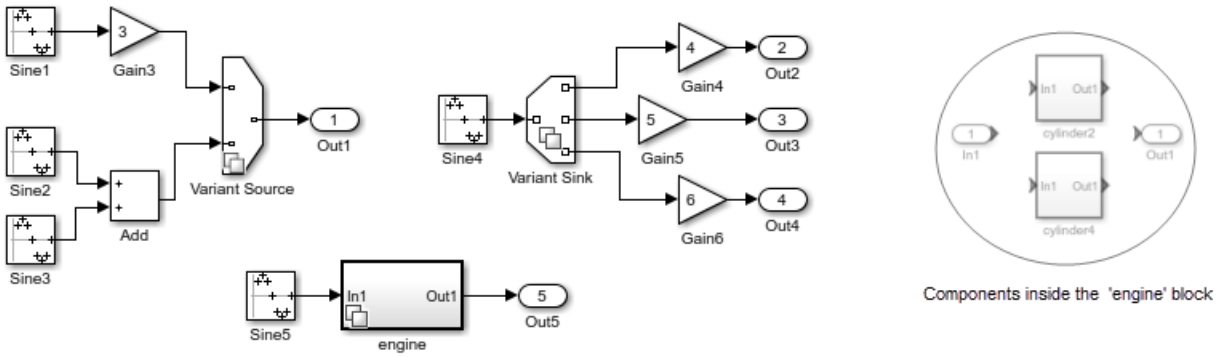
- “Visualize Variant Implementations in a Single Layer” on page 12-53
- Variant System Design

Create and Validate Variant Configurations

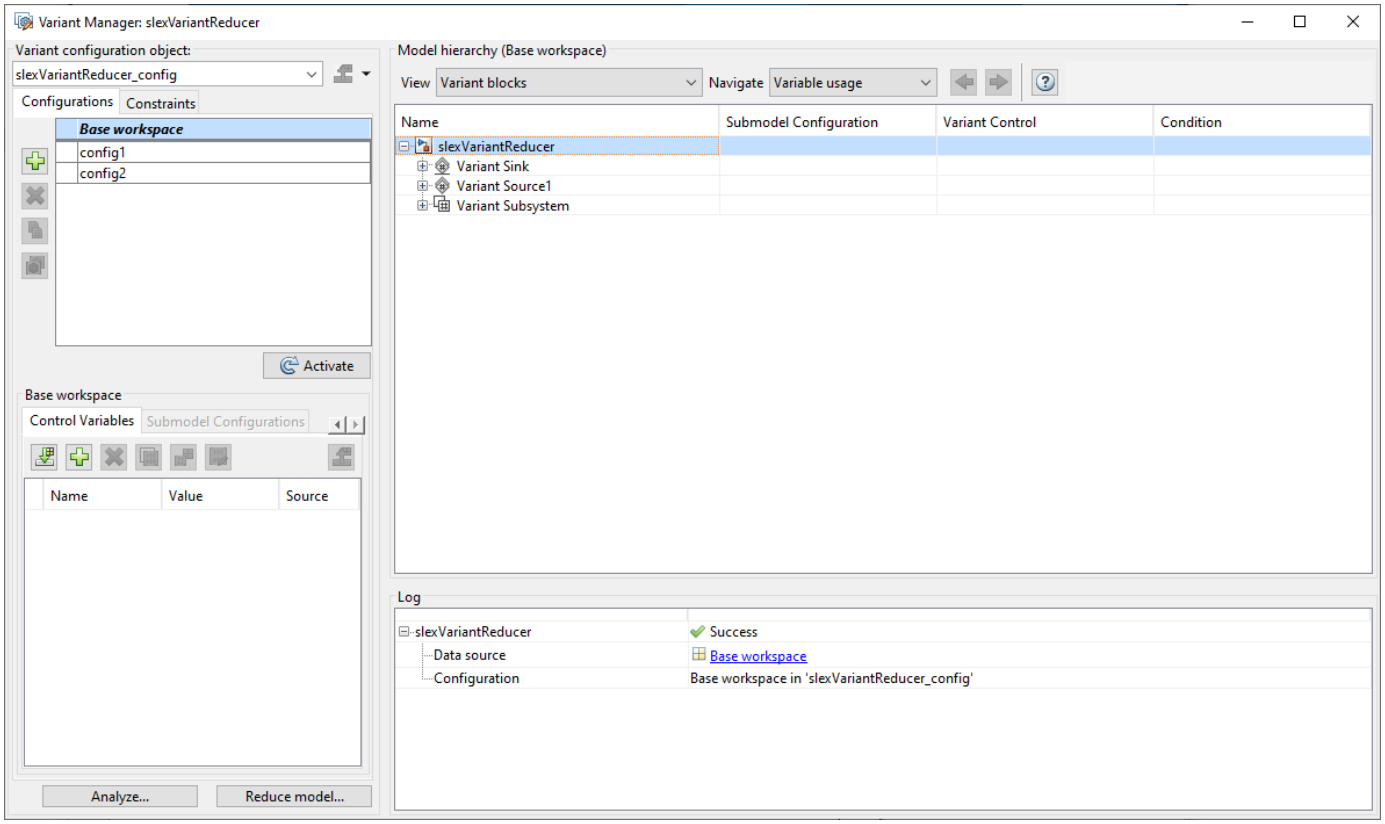
This example shows how to create and validate variant configurations for a model.

Step 1: Open Variant Manager

1 Open the model in which you want to create variant configurations. For example, consider a model containing a Variant Source, Variant Sink, and a Variant Subsystem block.



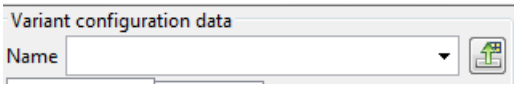
2 Right-click the variant badge and select **Open in Variant Manager**.



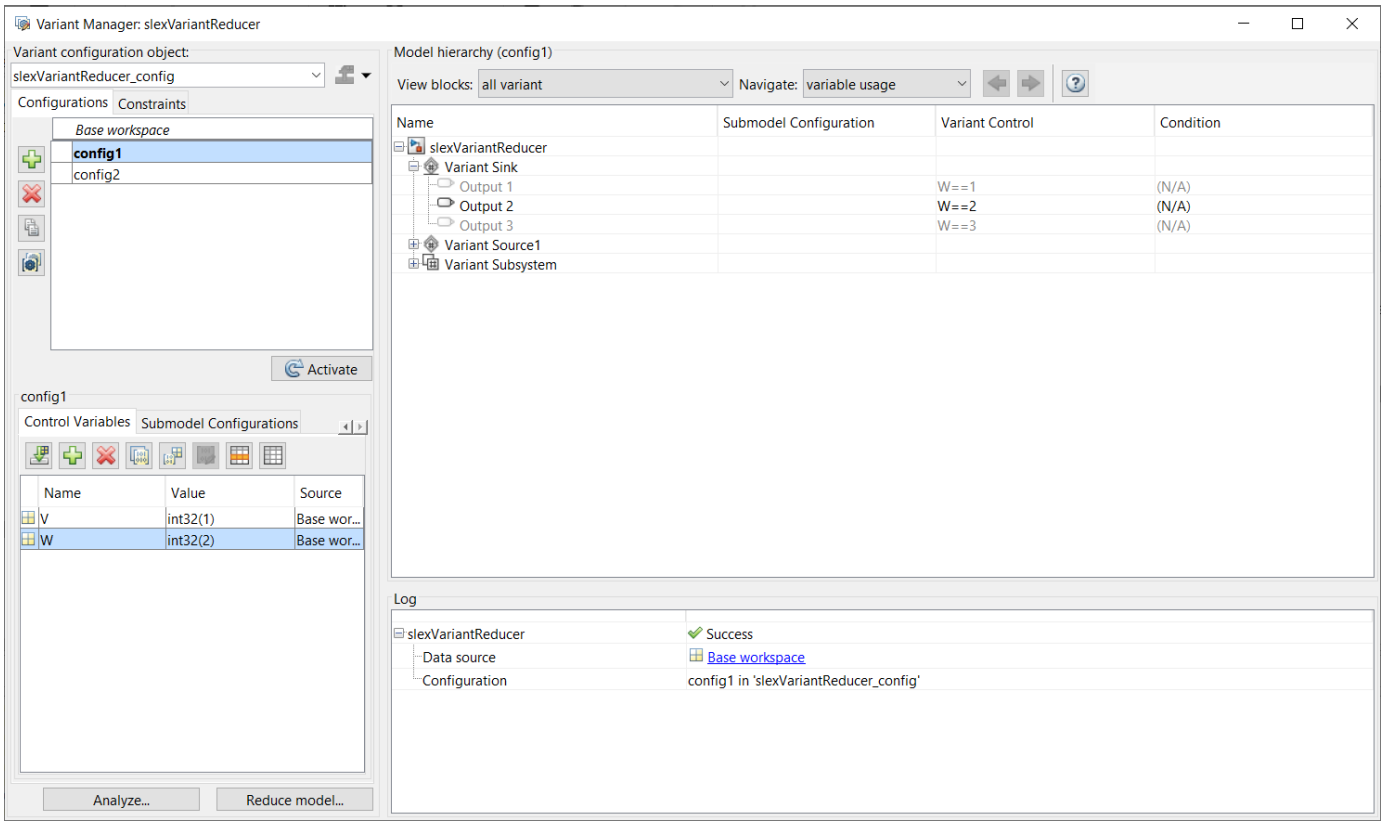
Step 2: Define Variant Configuration

You can use the **Variant configuration data** pane to define and store a variant configuration. For detailed information on each pane, see “Variant Manager Overview”.

1. In the variant configuration data pane, click the **Configurations** tab.
2. Click  . A variant configuration is added. Type a name for this configuration in the **Name** column.
3. Click the **Control Variables** tab in the controls section of the **Configurations** pane.
4. To import the control variables for the variant configuration from the global workspace, click  or  to add new control variables to the model.
5. Type a name for the variant configuration in the **Name** box available at the top-left of Variant Manager.



6. To export the variant configuration information to the global workspace, click  . The variant configuration for the model is now created.



Step 3: Activate and Validate Variant Configuration

To activate a variant configuration, select a configuration from the **Configurations** section and click the **Activate** button. The **Activate** button validates and applies the selected configuration on the model.

Note To reduce a model based on a variant condition, click **Reduce model**. For more information, see “Reduce Models Containing Variant Blocks” on page 12-77

See Also

Related Examples

- “Import Control Variables to Variant Configuration” on page 12-72
- “Define Constraints” on page 12-75

More About

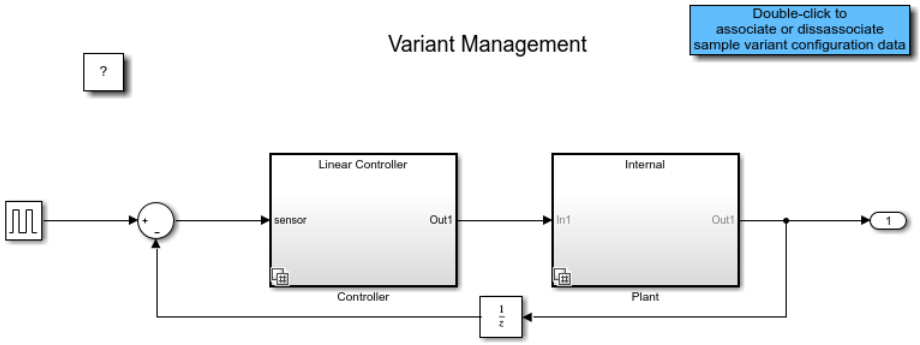
- “Approaches for Specifying Variant Controls” on page 12-27
- Variant System Design

Import Control Variables to Variant Configuration

This example shows how to import control variables to a variant configuration and associate a configuration with a referenced model.

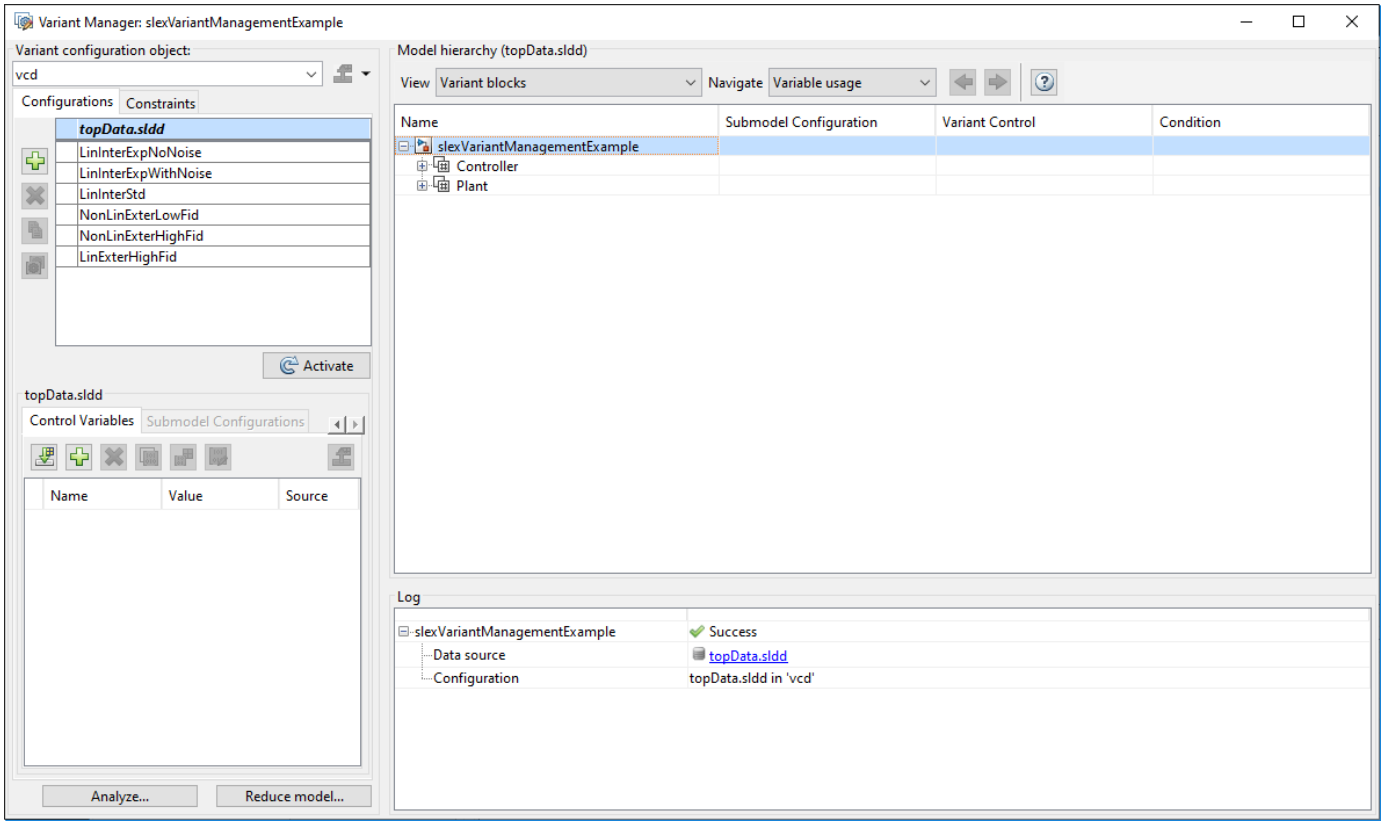
Step 1: Open Variant Manager

1 Open slexVariantManagement, which contains the variant configurations.



2 Double-click the blue block at the top to associate variant configuration data with the model.

3 In the **Modeling** tab of toolstrip, open **Design** section, click **Variant Manager**.

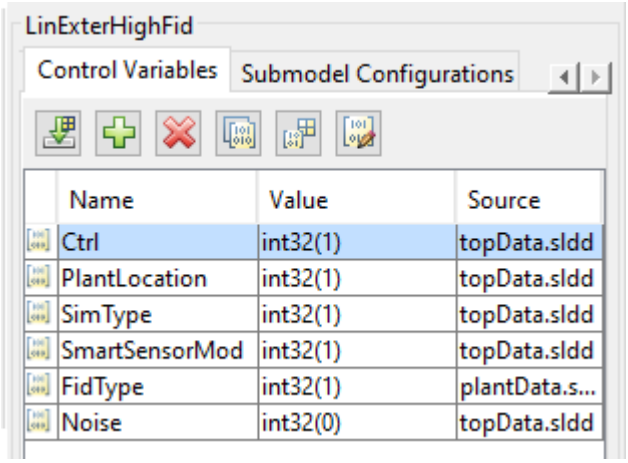


Variant configuration data `vcd` is associated with the model.

Step 2: Import Variant Configuration

1. In the Variant Manager, in the **Configurations** tab, select `LinExterHighFid`.
2. In the **Control Variables** tab, click  .

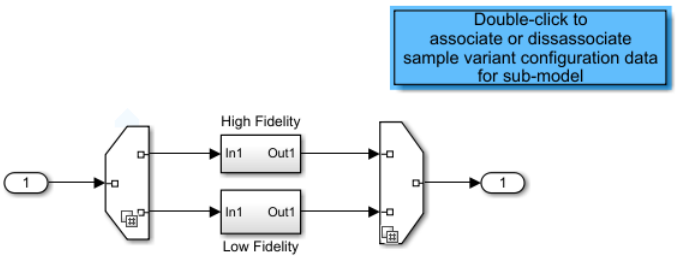
   The variables are imported.



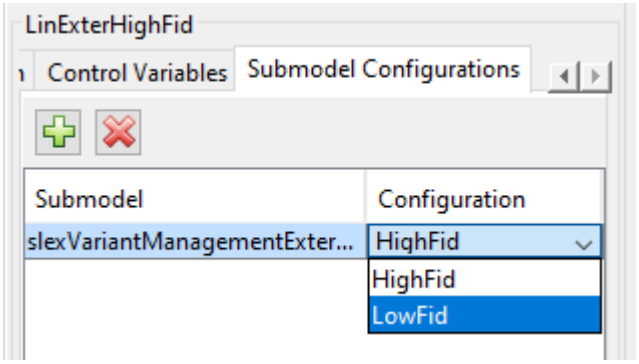
Note You can specify variant controls in the MATLAB global workspace or a data dictionary.

Step 3: View Referenced Model Configuration

1. Open the referenced model slexVariantManagementExternalPlantMdlRef.



2. Double-click the blue block at the top to associate variant configuration data with the referenced model.
3. In the Variant Manager that is opened from `slexVariantManagement`, select `slexVariantManagementExternalPlantMdlRef` under the **Submodel Configurations** tab.
4. Select the `LowFid` configuration from the dropdown menu.



5 To validate the model using the `LinExterHighFid` variant configuration, select `LinExterHighFid` from the **Configurations** list and click **Activate**.

Simulink validates the new configuration against the model and returns the validation results.

See Also

Related Examples

- “Define, Configure, and Activate Variants” on page 12-42

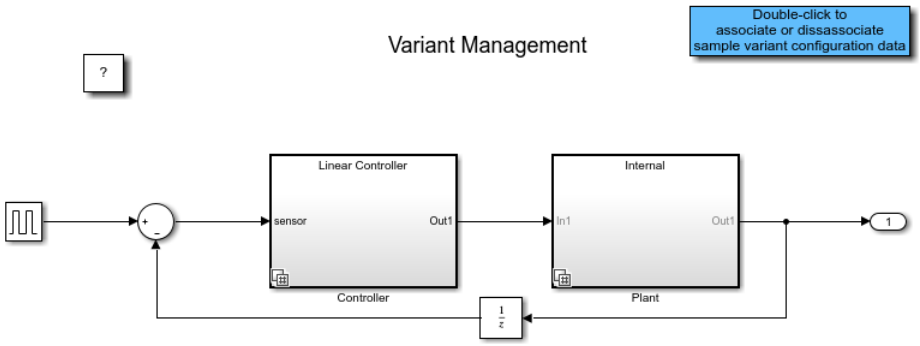
More About

- “Introduction to Variant Controls” on page 12-24
- Variant System Design

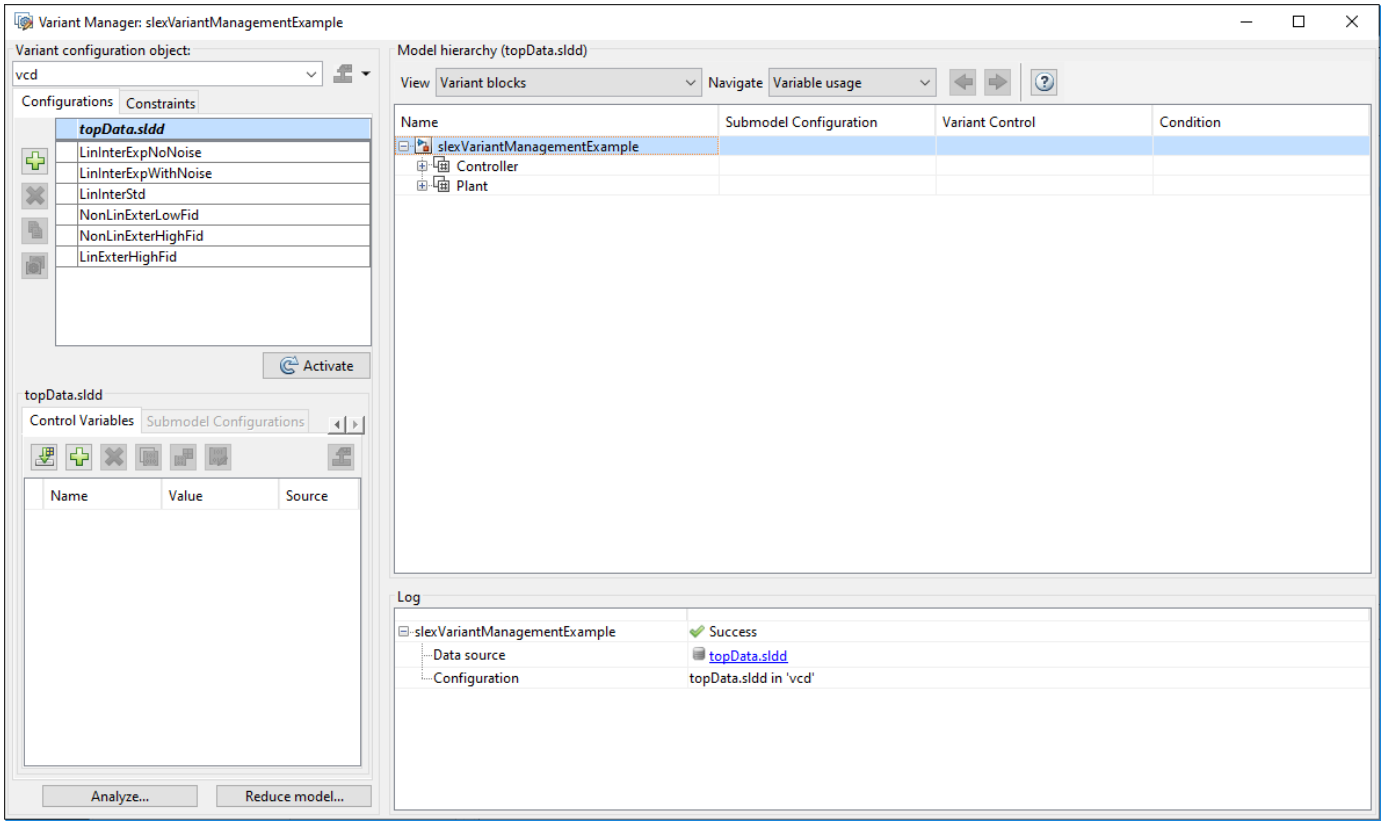
Define Constraints

This example shows how to define model-wide constraints that must evaluate to true for a variant configuration to become active.

1 Open slexVariantManagement.

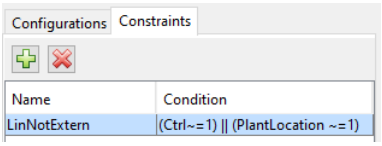


2 In the **Modeling** tab of toolstrip, open **Design** section, click **Variant Manager**.



3 In the Variant Manager, in the **Constraints** tab, click  .

4 Enter `LinNotExtern` as the **Name** and `(Ctrl~=1) || (PlantLocation ~=1)` as the **Condition** for the constraint.



This constraint activates variants that do not use the Linear Controller and External Plant Controller configurations.

5 To activate and validate the constraint, click the **Activate** button.

See Also

Related Examples

- “Create and Validate Variant Configurations” on page 12-69
- “Import Control Variables to Variant Configuration” on page 12-72

More About

- “Create Variant Controls Programmatically” on page 12-40
- “Approaches for Specifying Variant Controls” on page 12-27
- Variant System Design

Reduce Models Containing Variant Blocks

Note You require a Simulink Design Verifier license to reduce your model.

A variant model can contain multiple variable structures and a single fixed structure. The combination of a variable structure and the fixed structure to create a model depends on different combinations of the variant choices that you select. Each combination of the variant choices can be stored as a variant configuration.

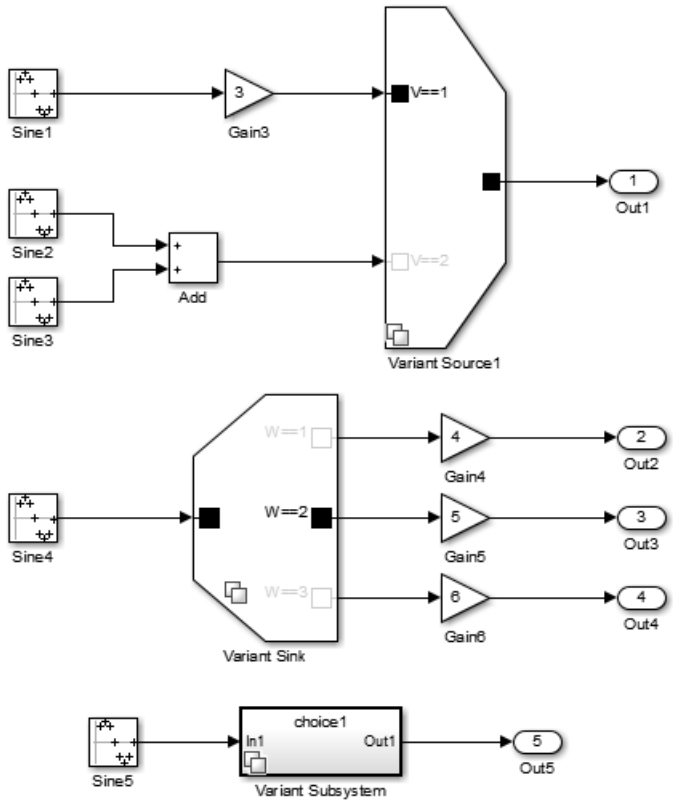
Variant models can be reduced to simplified, standalone model depending on the selected variant configurations. Additionally, all related files and variable dependencies are also reduced. These reduced artifacts are packaged into a user-specified output folder.

Note Variant model containing a Variant Connector block cannot be reduced.

Consider the model Variant Reducer. The model contains a Variant Source block, a Variant Sink block, and a Variant Subsystem block with these variant choices:

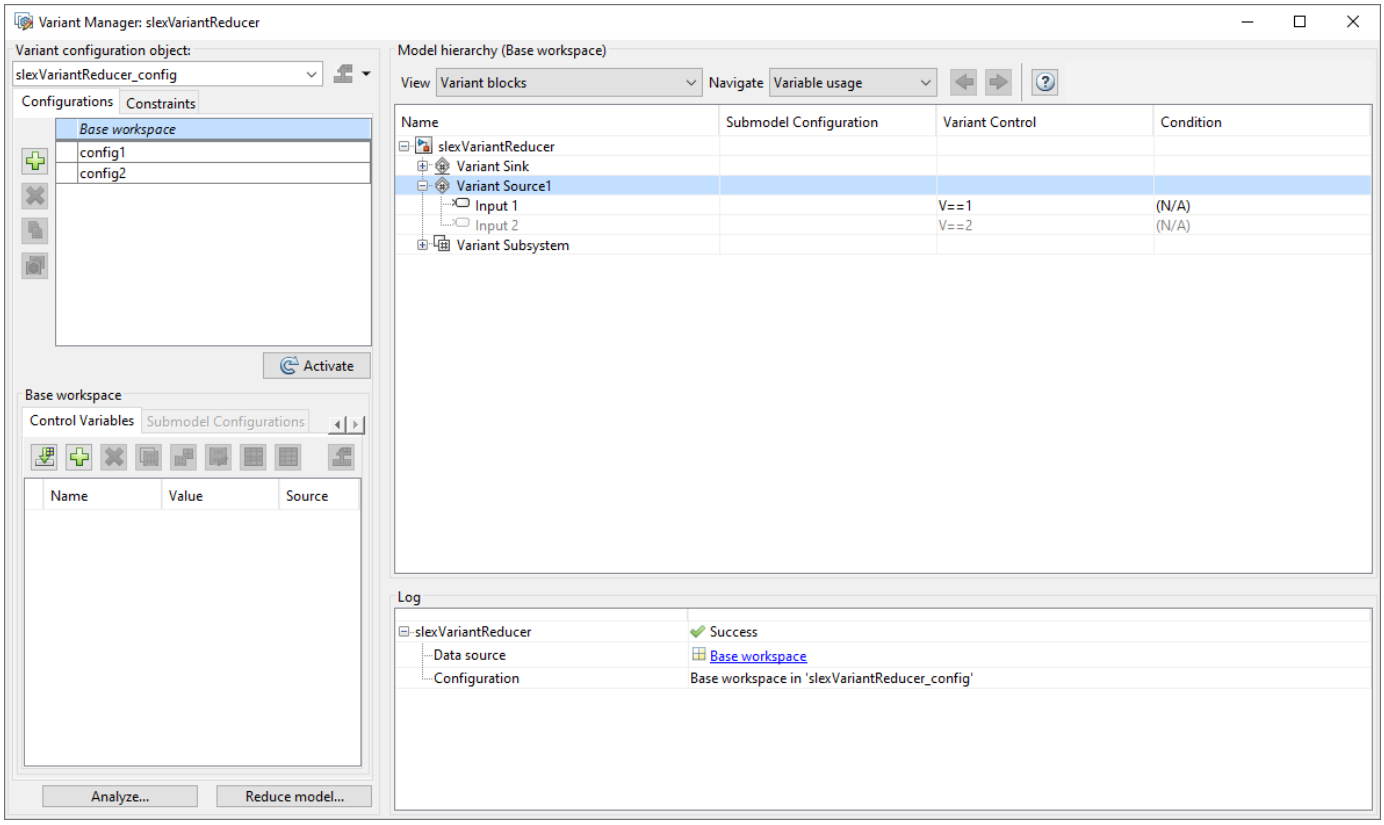
- Variant Source: `V==1` and `V==2`
- Variant Sink: `W==1`, `W==2`, and `W==3`
- Variant Subsystem: `V==1` and `V==2`

Assume that the model has two predefined variant configurations, named `config1` (`V==1 && W==2`) and `config2` (`V==2 && W==2`). These configurations are saved in a variant configuration data object, `varConfig`.

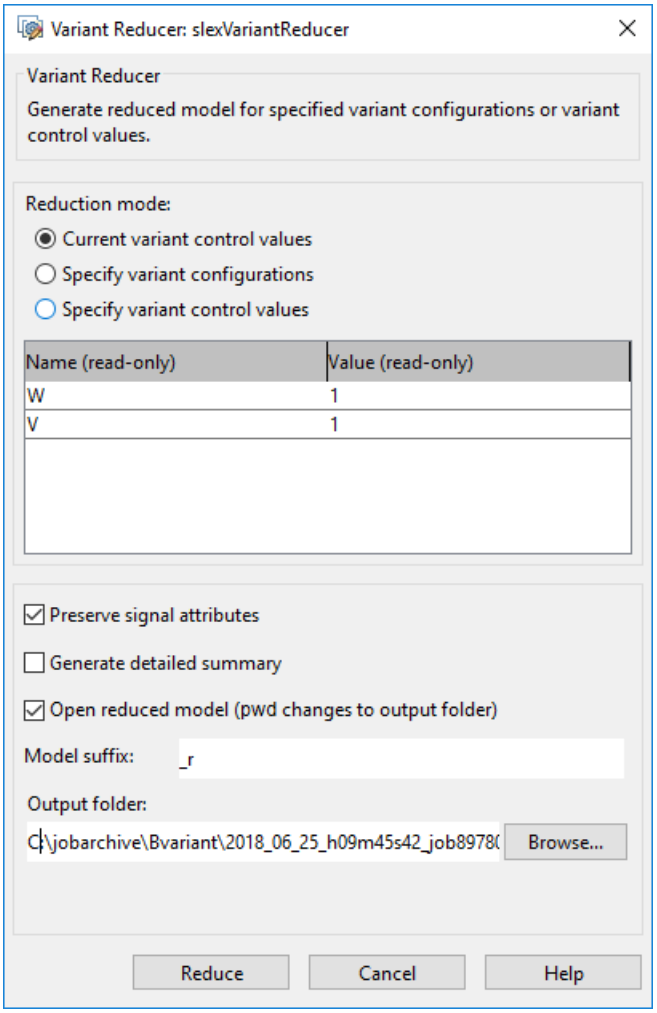


To reduce the model, perform the following steps:

1. Right-click the variant badge, and select **Open in Variant Manager**. The Variant Manager opens displaying the predefined configurations.

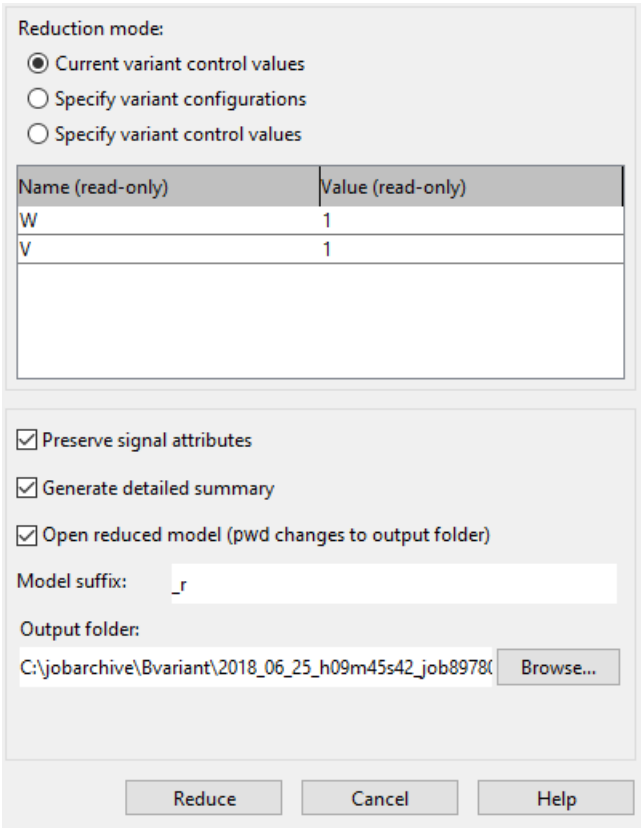


2 Click **Reduce model**. The Variant Reducer dialog box opens.

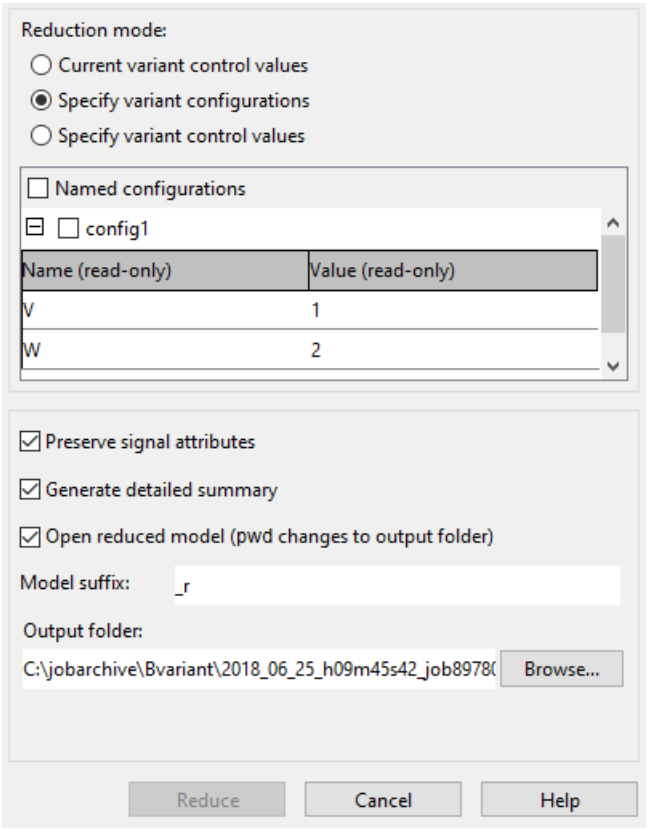


3 In the **Reduction mode** section, select:

- **Current variant control values** : To reduce the model based on its variant control variable values in the global workspace.



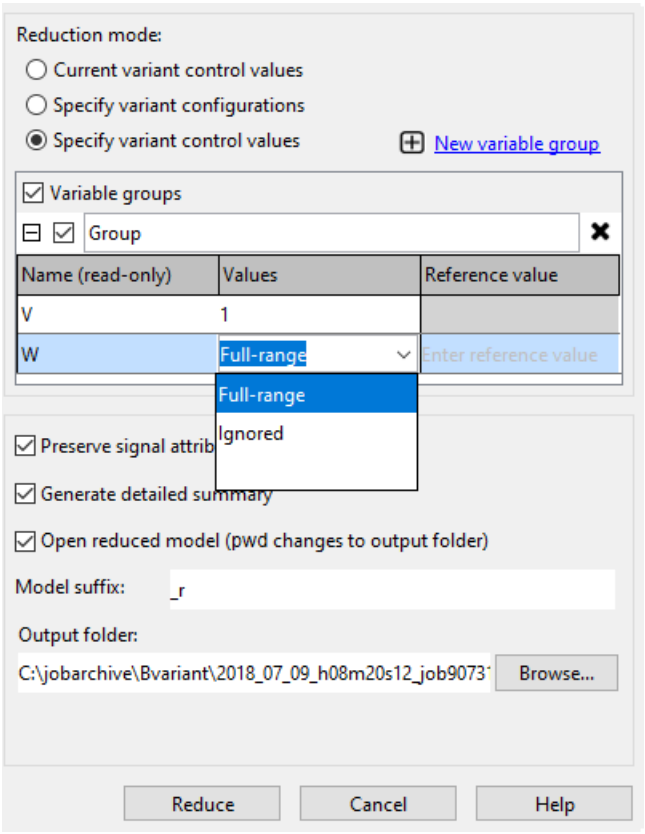
- **Specify variant configurations**: To reduce the model that is associated with a variant configuration data object and configurations to be retained in the reduced model.



Note During reduction, the control variable values from the last selected configuration are stored in the global workspace.

- **Specify variant control values**: To reduce the model based on the variant control variable values. You can create multiple variable groups corresponding to different configurations. Click **New variable group** to set the values for variant control variables. You can either specify a variant control value or select `Full-range` or `Ignored` from the drop-down list. Specifying a variant control value as a vector also allows you to reduce a model for all combinations of that variable. For example, if you specify values `V = 1` and `W = [1,2]`, then the model is reduced for the configurations `{V==1, W==1}` and `{V==1, W==2}`. If you select `Full-range` as a variant control value, **Reference Value** column is activated to enter a reference value required for successful model compilation. The model is reduced for all valid values of the specified variant control variable. If you select `Ignored` as a variant control value, then that variant control variable is not considered while reducing the model.

Note To use a full-range variant control variable, **Variant activation time** in the block parameters dialog of the blocks which uses that Variant control variable must be set to `code compile`.



Note If you invoke variant reduction by specifying variable groups, the reduced model will have the variant configurations corresponding to the variable groups associated with it. This overwrites any existing variant configurations present in the original model.

4 Select **Preserve signal attributes** to preserve the compiled signal attributes between the original and reduced model. When this option is selected, the Variant Reducer tries to preserve the compiled signal attributes between the original and reduced models by adding signal specification blocks at appropriate block ports in the reduced model. Compiled signal attributes include signal data types, signal dimensions, compiled sample times, and so on.

5 Select **Generate detailed summary** to generate the Variant Reducer summary in the output folder. The Variant Reducer summary contains summary of **Variant Reducer Options**, **Original and Reduced Model Differences**, **Dependent Artifacts**, **Callbacks** and **Warnings**.

The detailed summary can be used to get traceability information between the original and reduced model. It also helps in identifying artifacts which cannot be handled automatically and need manual intervention. For example, callback codes that may need to be modified.

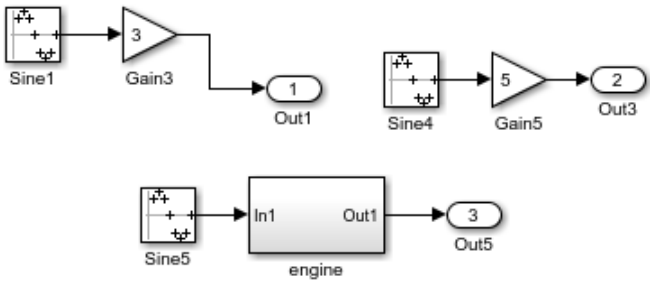
Note To generate detailed summary, you must have **Simulink Report Generator** license.

6 Specify a value as the suffix in the **Model suffix** field. The model suffix value is appended to the reduced models, data dictionaries, and the related artifacts. By default, `_r` is the suffix.

7 Specify the output folder to store the reduced model.

Note Selecting the **Open reduced model** check box changes the current working folder to the output folder.

8 Click **Reduce**. The reduced model for the required configurations are now created. If the model contains resolved library links or referenced models, the corresponding parent is reduced for the specified configuration and is referenced in the model. The reduced model, reduced referenced model, and the reduced library get their names from the corresponding model, referenced model, or the library with `_r` (**Model suffix**) appended to it.



Consider a Variant model that contains a Simulink Function block with Variant condition on the Simulink Function block as `V==1 || V==2 || V==3 || V==4`. If the model is reduced for any or a combination of the available Variant conditions, the Simulink Function block in the reduced model is unconditional. For example, if the model is reduced for Variant condition, `V=1, V=2, and V=3`, the Simulink Function block in the reduced model is unconditional. Whereas, if the model is reduced for Variant condition, `V=1, V=2, and V=5`, the Simulink Function block in the reduced model remains conditional with `V==1 || V==2` as the Variant condition.

Reduce Model Programmatically

To reduce a model programmatically, use the syntax:

```
Simulink.VariantManager.reduceModel(model,<Name>,<Value>)
```

For example,

- To reduce the model based on its variant control variable values in the global workspace.

  ```
  Simulink.VariantManager.reduceModel('sldemo_variant_subsystems')
  ```

- To reduce the model based on its variant control variable values in the global workspace to a specified folder.

  ```
  Simulink.VariantManager.reduceModel('sldemo_variant_subsystems', 'OutputFolder', 'outdir')
  ```

- To reduce the model that is associated with a variant configuration data object and configurations to be retained in the reduced model.

  ```
  Simulink.VariantManager.reduceModel('sldemo_variant_subsystems','NamedConfigurations', {'LinIr
  ```

- To reduce the model by specifying configurations in the form of a structure of variant control variables.

  ```
  Simulink.VariantManager.reduceModel('iv_model', 'VariableConfigurations', {'V',1,'W',[1 2]})
  ```

  Here, two configurations are specified corresponding to `{V=1, W=1}` and `{V=1, W=2}`, respectively.

- To reduce the model by specifying variant control values where 'w' is a full-range variant control value.

```
Simulink.VariantManager.reduceModel('slexVariantReducer','VariableConfigurations',{'V',1},'Ful
```

Here, four configurations are computed corresponding to `{V==1, W==1}`, `{V==1, W==2}`, `{V==1, W==3}` and `{V==1, W==0}` respectively.

For more information on reducing model programmatically, see `reduceModel`.

Considerations and Limitations

- The output folder to store the reduced model must not be under `matlabroot`.
- If a model has dependencies on files that are located under `matlabroot`, these files are not modified or copied to the output folder during model reduction. File dependency can include files from the Simulink libraries, `.m` files, `.mat` files, `.sldd` files.
- If the output folder contains the `variant_reducer.log` file from the previous model reduction, the reducer overwrites all the files available in that output for any subsequent reduction.
- Callback code:
  - Model callbacks, mask initialization code and mask parameter callback codes must be modified manually.
  - InitFcn, MaskEval, PreLoad, PostLoad and any edit-time callback codes from variant inactive components (models, blocks, signals, etc.) are removed. This can cause unexpected behavior in the reduced model.
- Additional blocks are added automatically to the reduced model to ensure consistent simulation semantics. Additional blocks can include Signal Specification blocks for consistent signal attributes (data type, dimensions, complexity) or the Ground and the Terminator blocks for unconnected signals.
- During model reduction, commented blocks present on the active path are retained while the commented blocks present on an inactive path are deleted.
- During model reduction, elements in Stateflow canvas, including variant transitions are not modified.
- Signal attributes (data type, complexity, dimensions, etc.) coming from the inactive elements in Stateflow charts may not be retained in the reduced model.

See Also

Related Examples

- “Create and Validate Variant Configurations” on page 12-69

More About

- “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60
- Variant System Design

Condition Propagation with Variant Subsystem

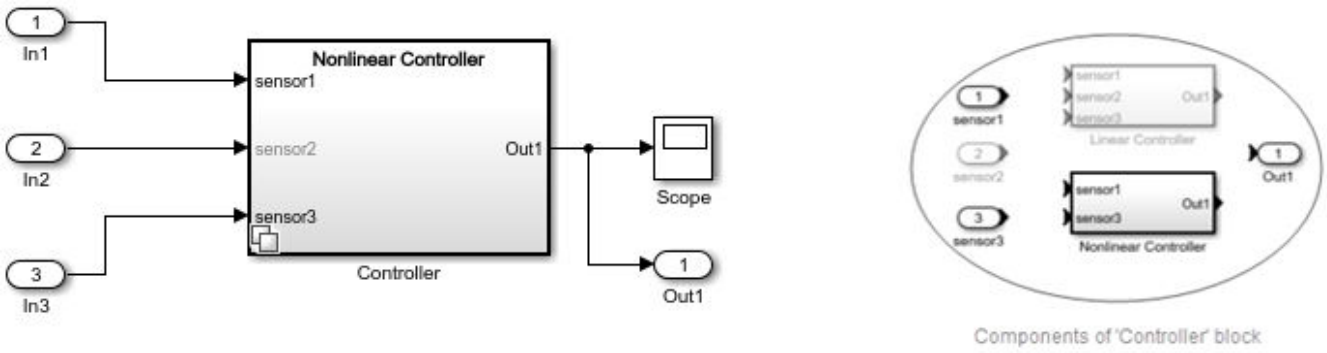
When you specify variant conditions in models containing Variant Subsystem blocks, Simulink propagates these conditions to determine which components of the model are active during simulation. A variant condition can be a condition expression or a variant object.

The variant condition annotations help you visualize the propagated conditions. To view the variant condition annotations, on the **Debug** tab, select **Information Overlays > Variant Legend**.

Note If **Variant Legend** is not available, on the **Debug** tab, select **Information Overlays > Variant Conditions**.

In the legend, the `(default)` keyword is displayed as negated condition.

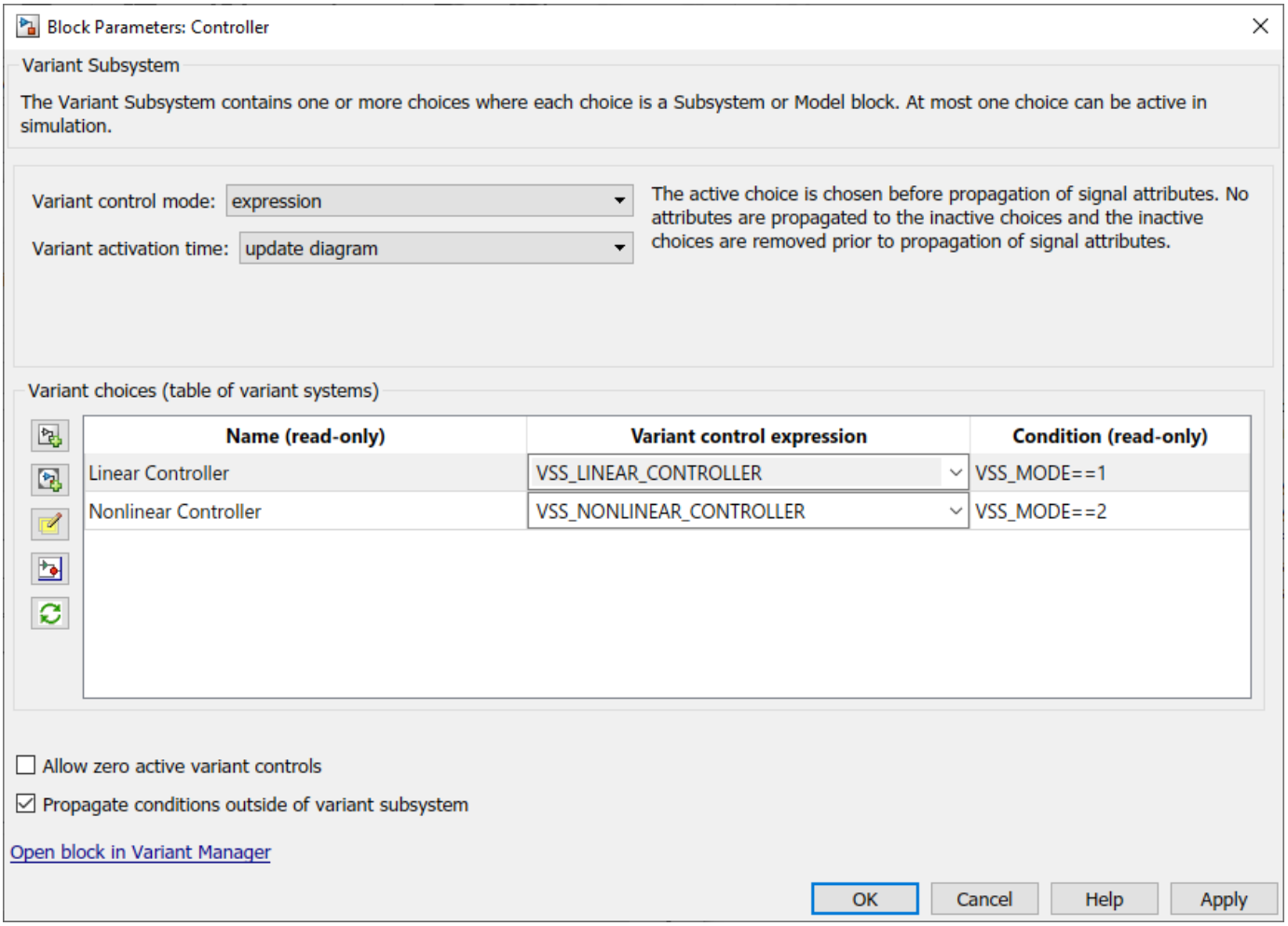
Consider this model containing a Variant Subsystem block with variant choices. A specific variant condition activates each block.



In the Variant Subsystem (`Controller`), `sensor1` and `sensor3` are used both in the Linear Controller and Nonlinear Controllers but `sensor2` is used only in the Linear Controller. Hence, the `sensor2` block is executed only when the Linear Controller choice is active and is not executed for any other choice. To ensure that the components outside of the Variant Subsystem (`Controller`) are aware of the active or inactive state of blocks with the Variant Subsystem block, the block condition must propagate outside of the Variant Subsystem block.

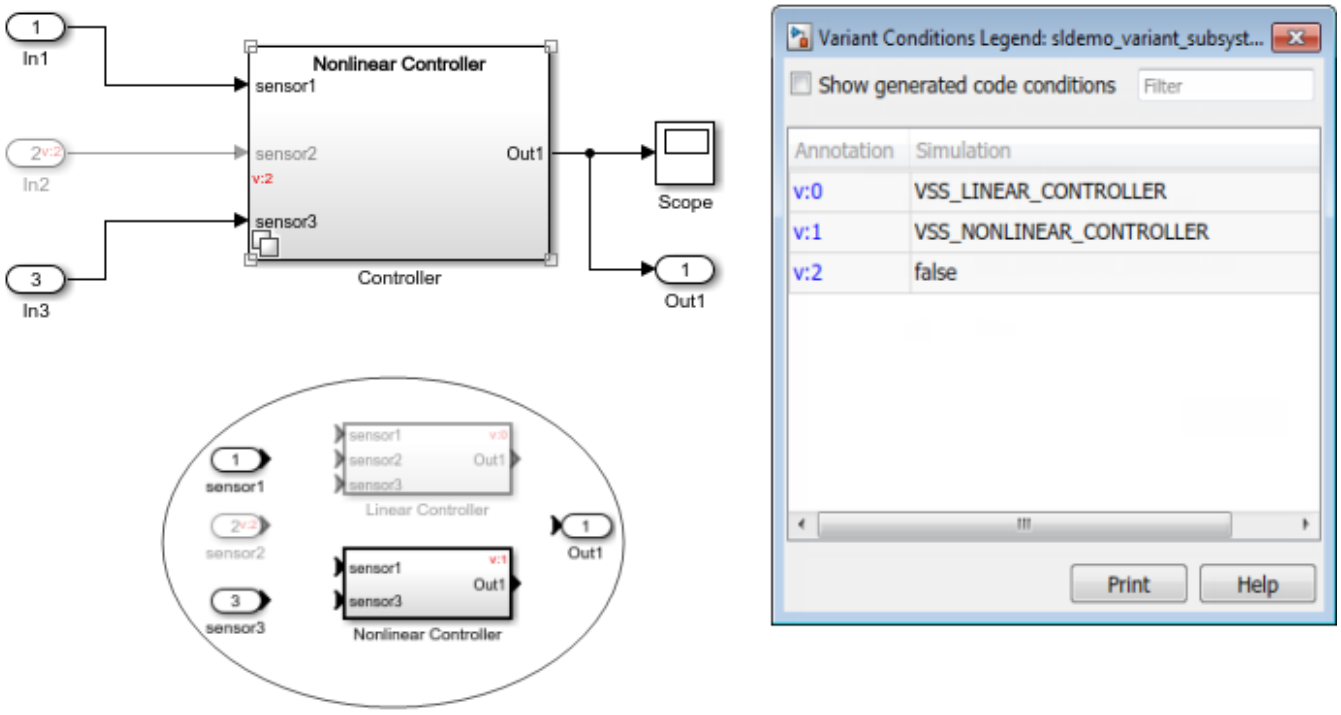
Propagate Conditions Without Generate Preprocessor Conditionals

To propagate conditions outside of Variant Subsystems without generate preprocessor conditionals, select the **Propagate conditions outside of variant subsystem** check box in the **Block Parameter** dialog box of the Variant Subsystem block. By default, **Propagate conditions outside of variant subsystem** is not selected.



When you simulate the model with the active choice as Nonlinear Controller and the **Propagate condition outside of variant subsystem** selected, only the active choice is analyzed. Notice that the **Variant activation time** is set to `update diagram`.

The **Variant Condition Legend** displays the inactive conditions as `false`. Here, `sensor2` is inactive with variant choice as Nonlinear Controller and is marked as `false`. The annotations are displayed on the `sensor2` port and the inactive block that is connected to `sensor2`.



Components of 'Controller' block

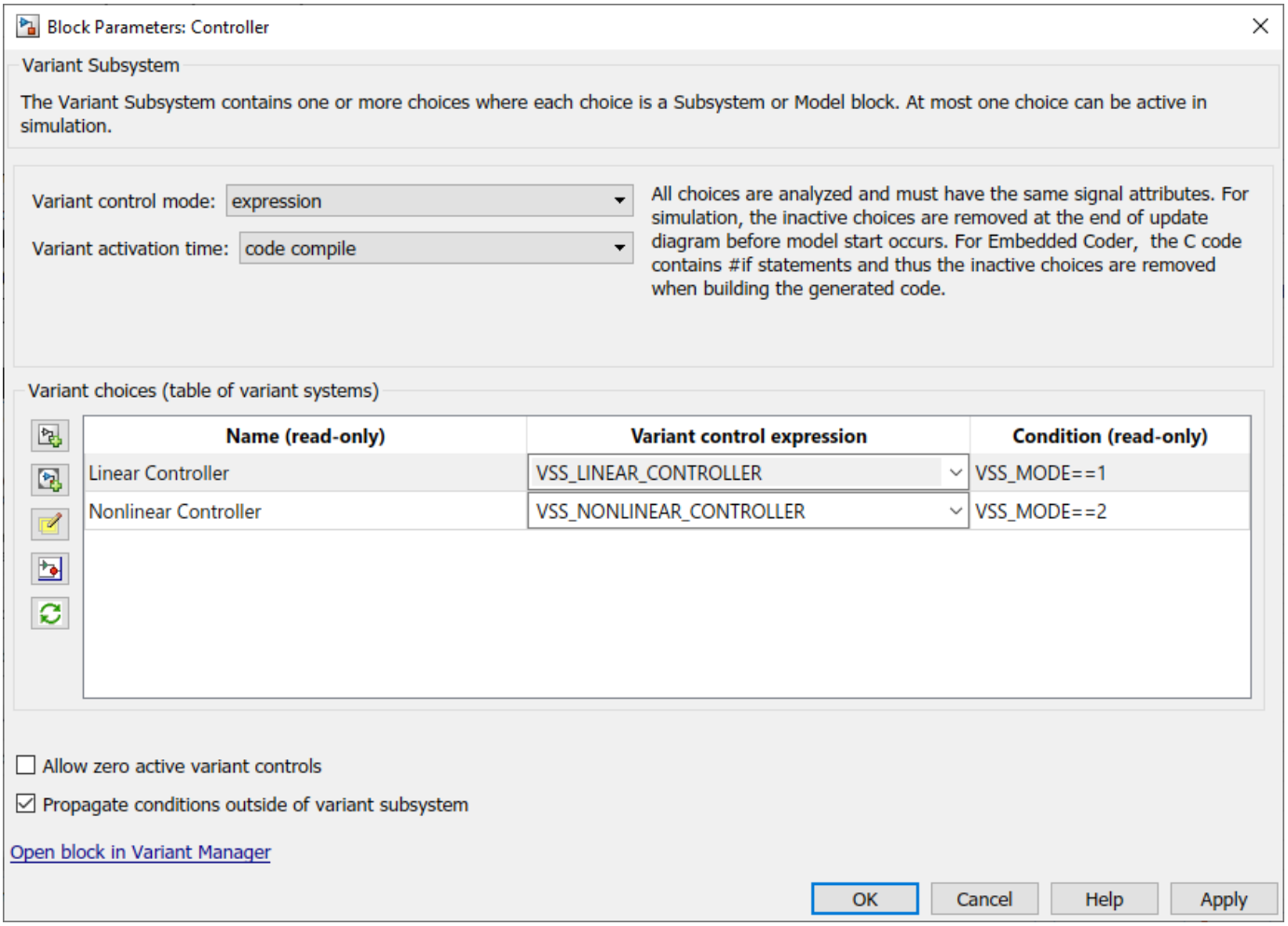
When you generate code for condition propagation without generate preprocessor conditionals, the inactive blocks are ignored. In this example, the input port `In2` is not shown in the generated code.

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1;                          /* '<Root>/In1' */
  real_T In3;                          /* '<Root>/In3' */
} EXTU_variant_subsystem_propagate_T;
```

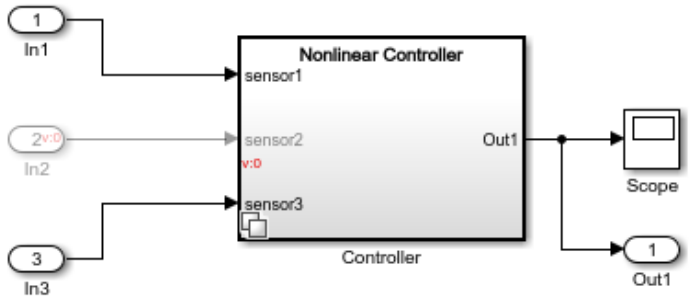
Propagate Conditions with Generate Preprocessor Conditionals

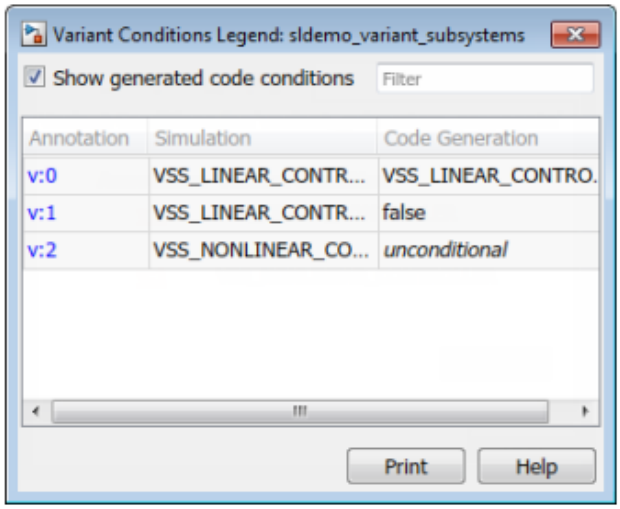
To propagate conditions outside of Variant Subsystem with generate preprocessor conditionals, select the **Propagate conditions outside of variant subsystem** check box and set the **Variant activation time** to `code compile` in the **Block Parameters** dialog box of the Variant Subsystem.

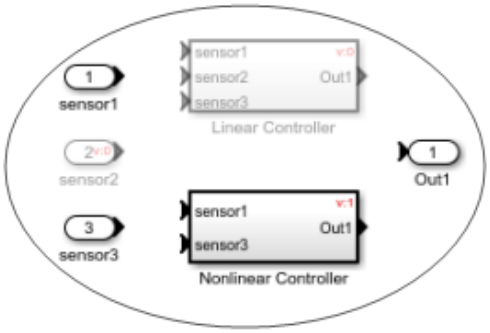
Note **Variant activation time** is available only when you specify **Variant control mode** as `Expression`.



When you simulate the model with active choice as Nonlinear Controller and **Propagate conditions outside of variant subsystem** check box and the **Variant activation time** set to `code compile`, all the variant choices are analyzed. The **Variant Condition Legend** displays the variant conditions associated with the model.







Components of 'Controller' block

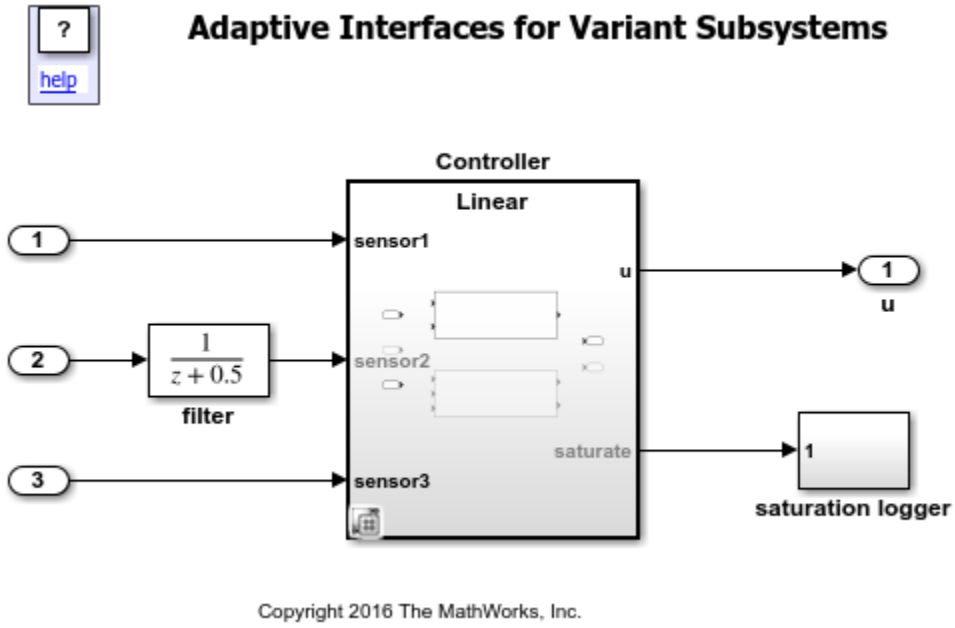
When you generate code for condition propagation with generate preprocessor conditionals, the model is analyzed for all the choices. In this example, the input port `In2` is guarded with necessary conditions.

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1;                          /* '<Root>/In1' */

#if VSS_LINEAR_CONTROLLER

  real_T In2;                          /* '<Root>/In2' */

#define   EXTU_VARIANT_SUBSYSTEM_PROPAGATE_T_VARIANT_EXISTS
#endif                                 /* VSS_LINEAR_CONTROLLER */

  real_T In3;                          /* '<Root>/In3' */
} EXTU_variant_subsystem_propagate_T;
```

Note To propagate variant conditions outside a Variant Subsystem block for zero or one active variant control specified, select **Allow zero active variant controls**.

Adaptive Interface for Variant Subsystems

When you select the **Propagate conditions outside of variant subsystem** check box in the **Block Parameters** dialog box, the Variant Subsystem adapts its interface to the connected blocks. Consider this model.



The Controller block is a Variant Subsystem that provides a `Linear` and a `Nonlinear` choice. The `Linear` choice is active when `V = 1`, and the `Nonlinear` choice is active when `V = 2`. Here, `V` is a variant control variable of the `Simulink.Parameter` type. Select the Controller block and, in Simulink click **Diagram** > **Block Parameters** (Subsystem). Verify that the **Propagate condition outside of variant subsystem** check box is selected.

To change the value of the variant control variable, in the MATLAB command window, type `V.Value = 1` or `V.Value = 2`.

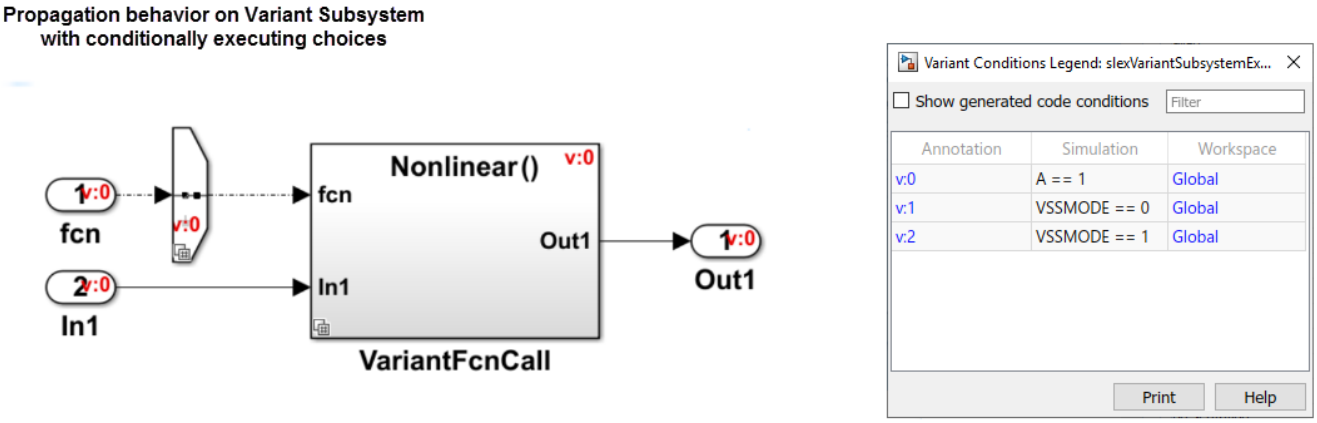
Double-click the Controller block to view its contents. The `Linear` choice is using `sensor1` and `sensor3` inputs of the Controller (Variant Subsystem block). It is not using `sensor2` and, therefore, does not produce a `saturate` output.

When you simulate this model, the Variant Subsystem block adapts its interface such that the condition `V = 2` (`v:1 V=2`) propagates the `In2`, the `filter`, and the `saturation logger` blocks.

Condition Propagation with Conditional Systems

When you propagate a condition to a conditional system, the same condition is set to all ports. For more information, see “Propagate Variant Condition to Conditional Subsystem” on page 12-149.

Consider this model. Here, when the condition is propagated to the Inport block (`fcn`), the same condition propagates to all the Inport and Outport blocks (as shown in the Variant Legend) and makes the Variant Subsystem block conditional.



Known Limitations

- Propagated variant conditions from variant subsystems cannot be set on Simscape or Stateflow blocks.
- C++ code generation is not supported for models that contain propagated conditions outside of a Variant Subsystem block.
- Variant condition propagation is not supported in models with root bus element ports.

Note All elements of a Mux, Demux, or a Vector Concatenate block signal must have the same variant condition.

Propagate Conditions Programmatically

To propagate conditions outside of a Variant Subsystem block programmatically, use one of these syntaxes:

- Propagate conditions without generate preprocessor conditionals:

  ```
  set_param(VariantSubsystemName, 'PropagateVariantConditions','on')
  ```

  For example,

  ```
  set_param('sldemo_variant_subsystems/Controller','PropagateVariantConditions','on')
  ```

- Propagate conditions with generate preprocessor conditionals:

  ```
  set_param(VariantSubsystemName,'PropagateVariantConditions',...
  'on','GeneratePreprocessorConditionals','on')
  ```

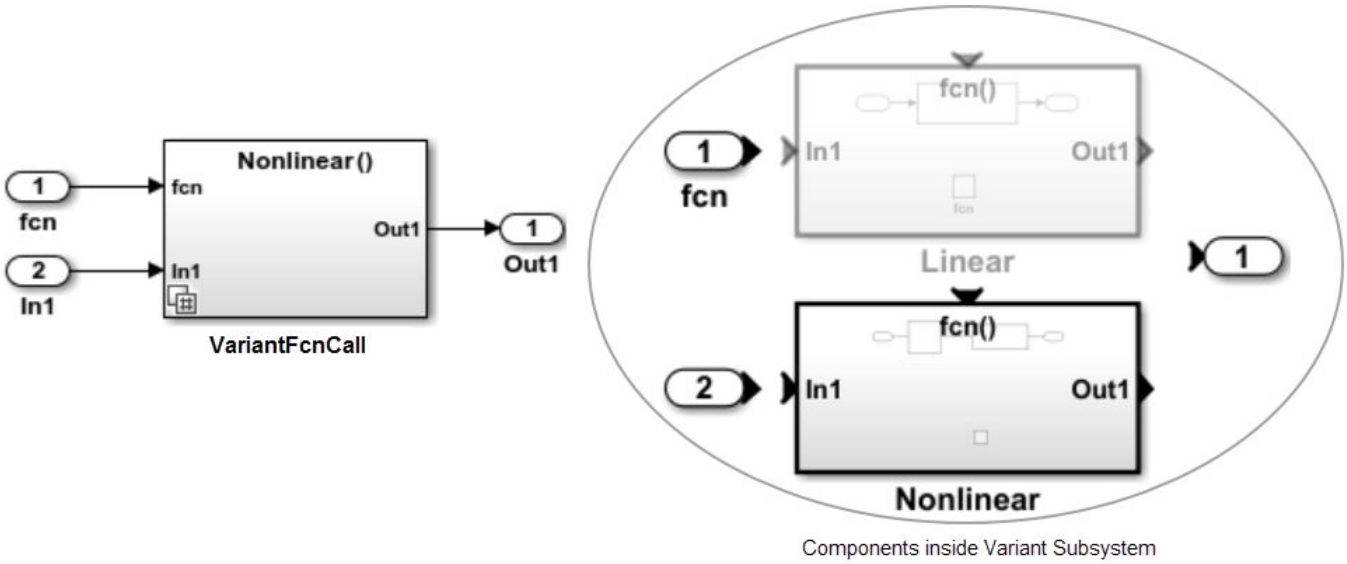
  For example,

  ```
  set_param('sldemo_variant_subsystems/Controller','PropagateVariantConditions','on',...
  'GeneratePreprocessorConditionals','on')
  ```

Code Generation with Conditional Systems

You can generate code for the model. To do so, on the **Apps** tab, click **Embedded Coder**, then on the **C Code** tab, click **Build**. For more information on configuring model to generate code, see “Prepare Variant-Containing Model for Code Generation” on page 12-49.

Consider a variant model containing a Variant Subsystem block for generating code.



In the generated code, the code inside `fcn` definition is guarded by C preprocessor conditionals `#if` and `#endif`.

```
void fcn(void)
{
  /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor_fcn_at_outport_1' */
#if VSSMODE == 0

    rtDWork.Linear.DiscreteFilter = rtU.In1 - 0.5 *
    rtDWork.Linear.DiscreteFilter_states;

  rtDWork.Linear.DiscreteFilter_states = rtDWork.Linear.DiscreteFilter;

#endif                                 /* VSSMODE == 0 */

#if VSSMODE == 1

    rtDWork.Nonlinear.DiscreteFilter = look1_binlxpw(rtU.In1,
    rtCP_LookupTable_bp01Data, rtCP_LookupTable_tableData, 4U) - 0.5 *
    rtDWork.Nonlinear.DiscreteFilter_states;

  rtDWork.Nonlinear.DiscreteFilter_states = rtDWork.Nonlinear.DiscreteFilter;

#endif                                 /* VSSMODE == 1 */

#if VSSMODE == 0

  rtY.Out1 = rtDWork.Linear.DiscreteFilter;
```

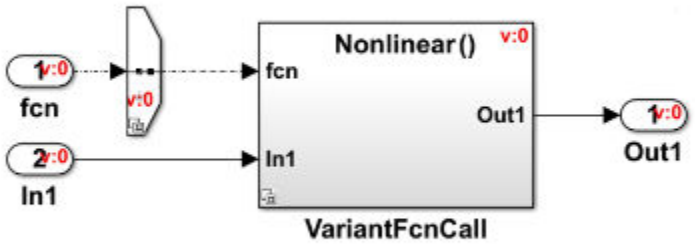
```
#endif                                  /* VSSMODE == 0 */

#if VSSMODE == 1

  rtY.Out1 = rtDWork.Nonlinear.DiscreteFilter;

#endif                                  /* VSSMODE == 1 */

}
```

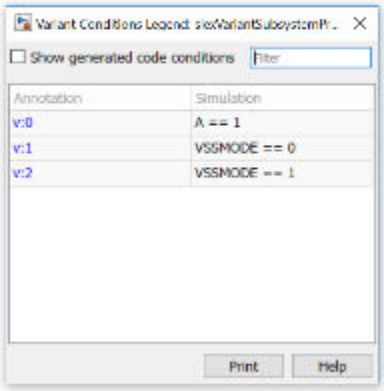
Note

- Configuring model as AUTOSAR component with runnable as Variant Subsystem choices is not supported.

Guarding the Function-Call Definition

To guard the whole definition of Function-Call (`fcn`), use a variant source as shown below.





In the generated code `fcn()` definition is guarded with A==1.

```
#if A == 1
void fcn(void)
{
…..
}
#endif
```

This function can be referred using a code snippet similar to as shown below.

```
…..
#if A==1
fcn()
#endif
……
```

See Also

More About

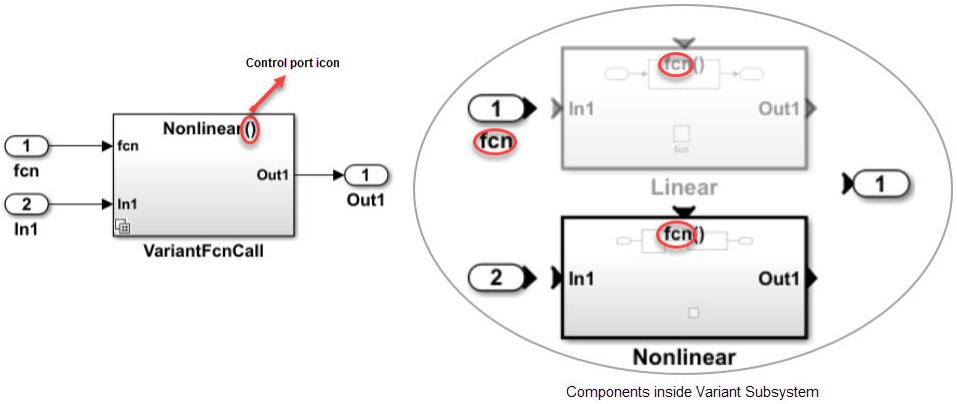
- “Prepare Variant-Containing Model for Code Generation” on page 12-49
- “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60
- “Model AUTOSAR Variants” (AUTOSAR Blockset)
- “Represent Subsystem and Variant Models in Generated Code” (Embedded Coder)
- Masking Variant Model
- Variant System Design

Variant Systems with Conditional Systems

You can use the conditionally executed systems (control ports) such as Enable, Trigger, Reset, and the Function-Call Subsystems within the Variant Subsystem block. When the **Variant activation time** parameter is set to `code compile` in the Block Parameters dialog box, a mix of control ports in a Variant Subsystem block as variant choice is not supported. Also, all control port types must have the same names. For more information on conditionally executed systems, see “Conditionally Executed Subsystems Overview” on page 10-3.

Export-Function Model with Variant Subsystem

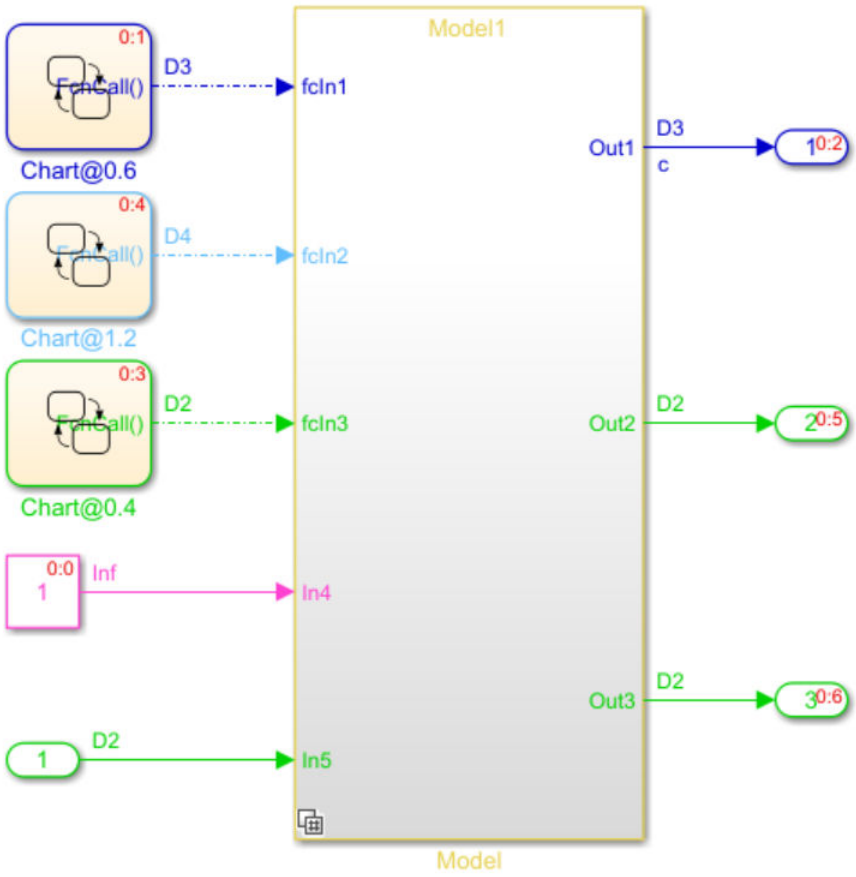
Consider a Variant model containing a Variant Subsystem block. If you use Function-Call system as a variant choice for the `Linear` Subsystem block then the other Subsystem block (`Nonlinear`) within the Variant Subsystem block must also be Function-Call system. Additionally, the control ports in `Linear` and `Nonlinear` blocks and the corresponding inport block must have the same name (`fcn`).



In the generated code, the code inside fcn definition is guarded by C preprocessor conditionals inside export function.

```
fcn() {
#if VSSMODE==0
// code for Linear choice
.......
#endif
#if VSSMODE==1
// code for Nonlinear choice
.......
#endif
```

You can also have a similar modeling pattern with multi-point entry function using Model block. An example is as follows:



In the above example, `fcln1`, `fcln2`, and `fcln3` are routed through the Variant Subsystem using Model blocks as variant choices.

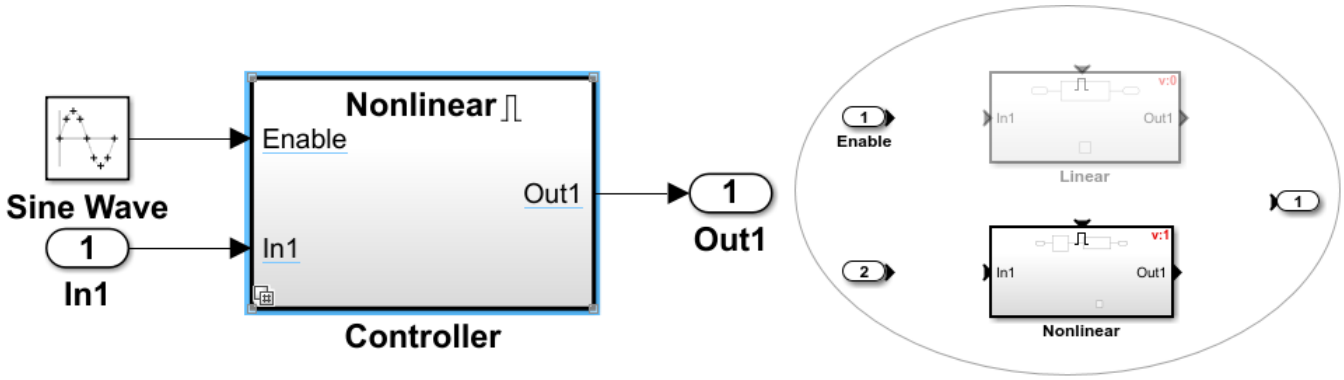
These conditions are not supported when using conditionally executed systems within a Variant Subsystem block as variant choice:

- Action ports as Variant choices
- Variant choices containing Iterator port with generate preprocessor conditionals set to ON
- Models with Initialize, Reset, Terminate, and Simulink functions

> **Note** Initialize and Terminate event ports are always unconditional because they control both the model default and block-specific initialize and terminate events of the referenced model. If you define an Initialize function block in the referenced model, it corresponds to an explicit initialize event.

Variant Subsystem with Enable Subsystem as Choice

Consider a Variant model containing a Variant Subsystem block. If you use Enable Subsystem as a variant choice for the `Linear` Subsystem block then the other Subsystem block (`Nonlinear`) within the Variant Subsystem block must also be Enable Subsystem.



This model simulates and generates code based on `VSSMODE` value.

Snippet of the generated code is as shown below.

```
step() {
#if VSSMODE==0
// code for Linear choice
.......
#elif VSSMODE==1
// code for Nonlinear choice
.......
#endif
```

See Also

More About

- “Variants Example Models” on page 12-121
- “Prepare Variant-Containing Model for Code Generation” on page 12-49
- “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60
- “Propagate Variant Condition to Conditional Subsystem” on page 12-149
- “Variants Example Models” on page 12-121
- Variant System Design

Convert Configurable Subsystem to Variant Subsystem

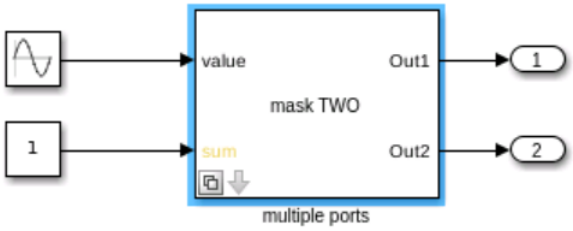
Note Configurable Subsystem will be removed in a future release. The Configurable Subsystem blocks in existing models must be converted to Variant Subsystem blocks.

Variant Subsystems offer more capabilities than Configurable Subsystems, with these advantages:

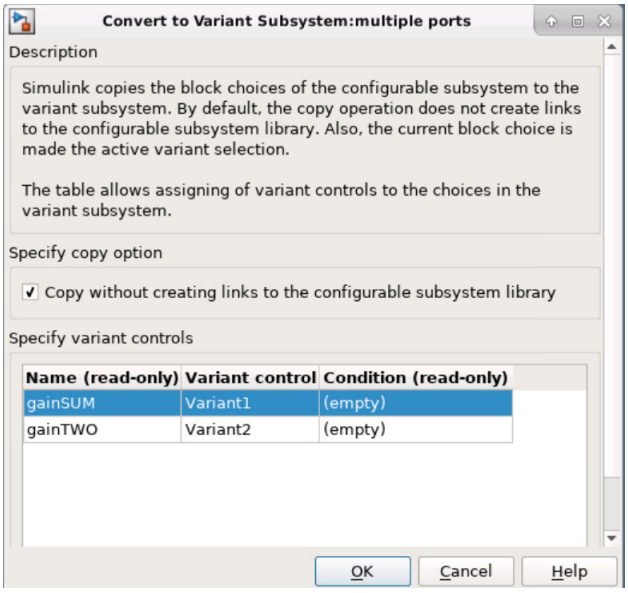
- You can mix Model blocks and Subsystem blocks as variant choices.
- You can specify variants that have different numbers of input and output ports.

Perform these steps to convert Configurable Subsystem block to a Variant Subsystem block:

1. Open a model containing Configurable Subsystem block.



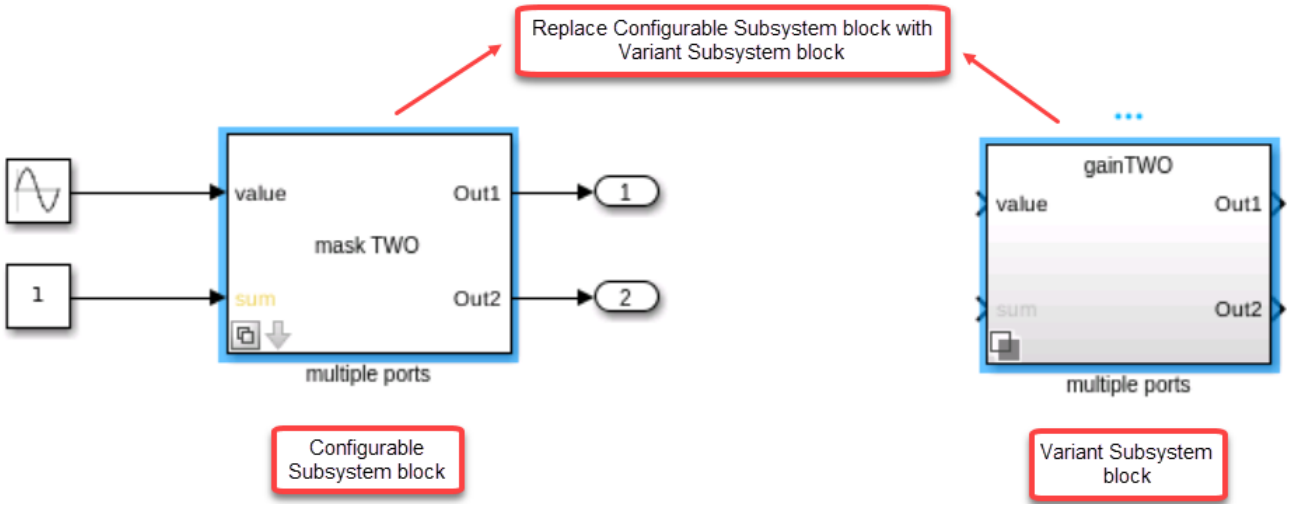
2. Right-click the Configurable Subsystem block and in the context menu, select **Subsystem & Model Reference > Convert to > Variant Subsystem**.



3. By default, the **Copy without creating links to the configurable subsystem library** check box is selected. This creates Variant choices without creating links to library.
4. Click **OK**. The Configurable Subsystem block is converted to Variant Subsystem block and is displayed in a new window.

Note When Configurable Subsystem block is converted to a Variant Subsystem block, the Block choice of Configurable Subsystem block is changed to `LabelModeActiveChoice` in Variant Subsystem block.

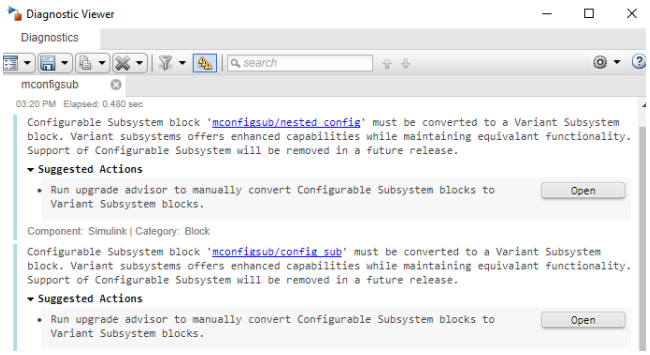
5 Manually replace the Configurable Subsystem block with the converted Variant Subsystem block in the original model.



Behavior of Configurable Subsystem on Loading

When you load a model containing Configurable Subsystem blocks, a warning is displayed which will instruct you to convert the Configurable Subsystem block to a Variant Subsystem block.

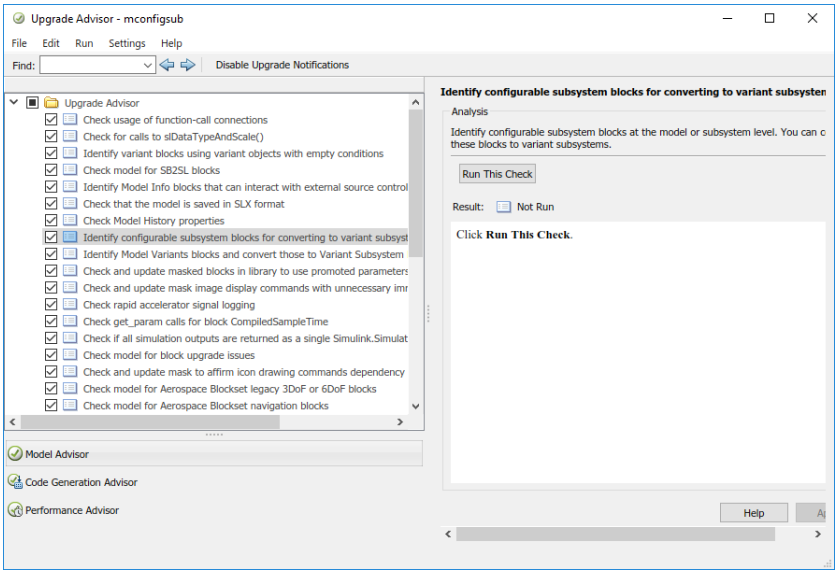
1 Consider a model with Configurable Subsystem block. When you load this model, a warning is displayed in the **Diagnostic Viewer**.



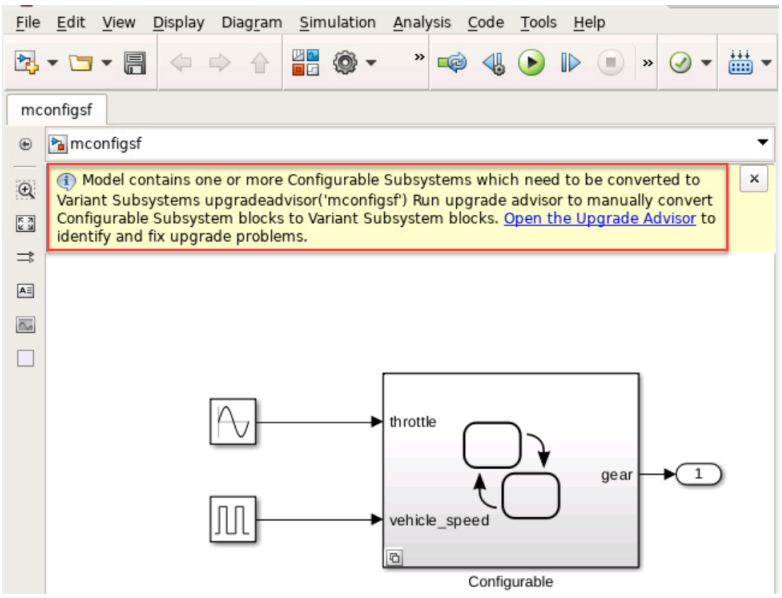
Note When you create a new Configurable Subsystem block, an upgrade advisor alert is displayed to convert the Configurable Subsystem block to Variant Subsystem block.

2 In the **Diagnostic Viewer**, click **Open** in the **Suggested Actions** section.

3 In the Upgrade Advisor, select **Identify configurable subsystem blocks for converting to variant subsystem blocks** and then click **Run This Check**. A list of all the Configurable Subsystem blocks in the model and recommend action to be performed is displayed.



Also, when you simulate the model containing Configurable Subsystem blocks, Upgrade Advisor warning is displayed in the editor. A sample screenshot is as shown below.



Changing Active Variant

When Configurable Subsystem block is converted to a Variant Subsystem block, the Block Choice of Configurable Subsystem block is changed to Label mode active choice in Variant Subsystem block.

To change the active variant, perform one of these steps:

- Right-click the badge on the Variant Subsystem block and select **Block Parameters (Subsystem)**. In the block parameters dialog box, select the active variant from `Label mode active choice` drop-down list.

- Right-click the badge on the Variant Subsystem block and select **Label Mode Active Choice**.

Note When a Configurable Subsystem block with a mask is converted to Variant Subsystem block, the `Label mode active choice` option and all other parameters in block parameters dialog box is disabled. To change the active variant, right-click the badge on the Variant Subsystem block and select **Label Mode Active Choice**.

Convert Configurable Subsystem Blocks to Variant Subsystem Blocks Programmatically

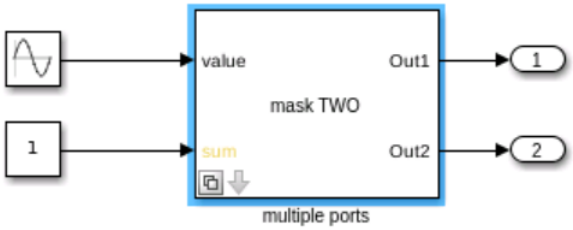
Configurable Subsystem blocks in a model can be converted to Variant Subsystem blocks programmatically using `convertToVariant` method.

When a Configurable Subsystem is converted to a Variant Subsystem block, the Block choice of Configurable Subsystem block is changed to `LabelModeActiveChoice` in the Variant Subsystem block.

If the Block choice of the Configurable Subsystem template block is linked, then the block is copied to the Variant Subsystem graph and the copied blocks will have its links retained. If the Block choice of the Configurable Subsystem template block is not linked, then the block is copied to the Variant Subsystem graph and the block in the Configurable Subsystem library is linked to it.

Perform these steps to convert Configurable Subsystem blocks in a model to Variant Subsystem blocks:

1 Open a model containing Configurable Subsystem block.



2 In the command-line, enter the `find_system` command to find all the Configurable Subsystem blocks in the model:

```
find_system(bdroot, 'Regexp', 'on', 'LookUnderMasks', 'on','FollowLinks', 'on', 'Variants', '
```

The Configurable Subsystem blocks present in the model are listed:

```
{'mconfigsub/config_sub'}
{'mconfigsub/nested config'}
```

3 Find the library template blocks from the list using `get_param` command.

```
get_param('mconfigsub/nested config','TemplateBlock')
ans =
    'mconfiglib/nested config'

get_param('mconfigsub/config_sub','TemplateBlock')
ans =
    'mconfiglib/Subsystem/config_sub'
```

4. Convert the library template blocks to Variant Subsystem blocks using `convertToVariant` method:

   ```
   Simulink.VariantManager.convertToVariant('mconfiglib/nested config')

   Simulink.VariantManager.convertToVariant('mconfiglib/Subsystem/config_sub')
   ```

   For information on using this method, see `convertToVariant`.
5. Save the libraries. You can use `save_system` command to save the libraries.
6. Close and open the model again. The Configurable Subsystem blocks in the model will be converted to Variant Subsystem blocks.

See Also

More About

- Variant Subsystem, Variant Model
- “Variants Example Models” on page 12-121
- Variant System Design

Variant Elements within Buses

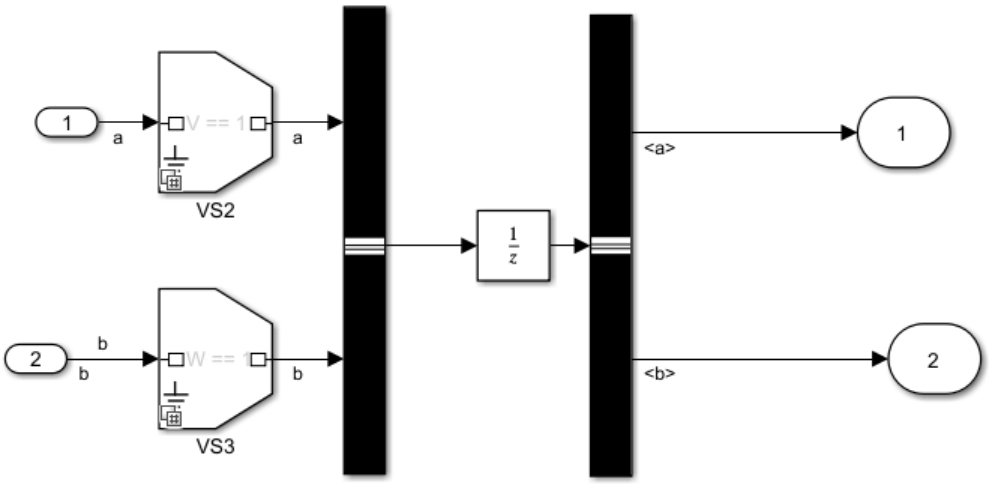
Create Buses with Variant Conditions

This example shows how to create bus signals with elements having different variant conditions. This model has two signals (‘a’, ‘b’) that are merged to create a bus signal. These two signals have different variant conditions, V==1 and W == 1.

The bus selector is fed by two signals having two different variant conditions (V==1 and W==1). When this bus signal is fed into a bus selector and when you select the individual signals from the bus, the variant condition is also selected.

Note Variant bus supports using Composite ports as input and output ports.

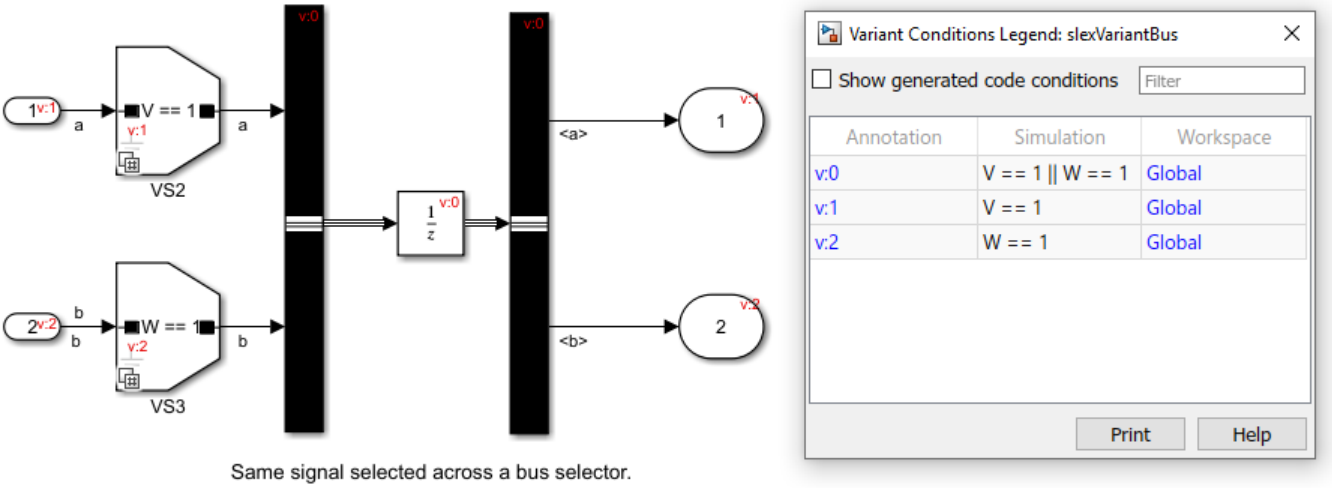
To see the completed model, open the `slexVariantBus` model.



Variant Condition Propagation with Bus

A Variant Source block can accept either virtual or nonvirtual bus inputs.

Consider this model.



Same signal selected across a bus selector.

Bus is created with variant conditions V = 1 and W = 1. When conditions V and W are propagated through the bus and when individual elements are selected from the bus, conditions are also selected.

Code Generation

You can generate code for the model. To do so, on the **Apps** tab of toolstrip, click **Embedded Coder**, then on the **C Code** tab, click **Build**. For more information on configuring model to generate code, see “Prepare Variant-Containing Model for Code Generation” on page 12-49.

When generating code with preprocessor conditionals, the bus types and hierarchies of all bus inputs must be the same.

Virtual and Nonvirtual Bus Behavior

Virtual buses provide the simplest approach for using buses to reduce signal clutter in a block diagram. Nonvirtual buses support modeling components (such as S-functions or MATLAB Function blocks) that require explicitly specified interfaces. There is no change in the propagation behavior of variant conditions when variant conditions are propagated through a virtual or nonvirtual bus.

Code generated for a virtual bus is as shown below.

```
/* Block states (default storage) for system '<Root>' */
typedef struct {

#if V == 1

real_T UnitDelay_1_DSTATE;           /* '<Root>/Unit Delay' */

#define D_WORK_EX_BUS_VAR_COND_VARIANT_EXISTS
#endif                                 /* V == 1 */
#if W == 1

int32_T UnitDelay_2_DSTATE;          /* '<Root>/Unit Delay' */

#define D_WORK_EX_BUS_VAR_COND_VARIANT_EXISTS
#endif                                 /* W == 1 */
```

```
#ifndef D_WORK_EX_BUS_VAR_COND_VARIANT_EXISTS

char _rt_unused;

#endif
} D_Work_ex_bus_var_cond;
```

Code generated for a nonvirtual bus is as shown below.

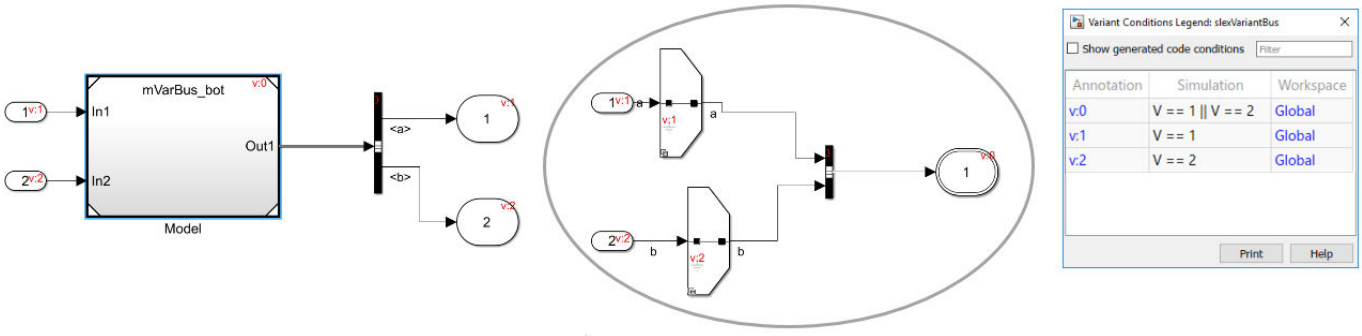
```
/* Block states (default storage) for system '<Root>' */
typedef struct {

#if V == 1 || W == 1

myBus UnitDelay_DSTATE;              /* '<Root>/Unit Delay' */

#define D_WORK_EX_BUS_VAR_COND_VARIANT_EXISTS
#endif                                 /* V == 1 || W == 1 */

#ifndef D_WORK_EX_BUS_VAR_COND_VARIANT_EXISTS

char _rt_unused;

#endif

} D_Work_ex_bus_var_cond;
```

You must have an associated bus object in the bus, which provides properties that Simulink uses to validate the bus signal. For more information on bus objects, see “Specify Bus Properties with Simulink.Bus Objects” on page 76-44. The bus object used in the bus is unconditional and it generates unguarded code as shown below.

```
typedef struct {
real_T a;
int32_T b;
} myBus;
```

Variant Bus with Model Block

Consider this model containing a Model block.



This model has two signals (‘a’, ‘b’) which have different variant conditions, V==1 and V==2.

From the Model block, the bus selector is fed two signals having two different variant conditions (V==1 and V==2). When you select the individual signals from the bus, the variant conditions are also selected.

Known Limitations

- Root bus element ports are not supported with variant elements.
- Bus objects do not support variant elements.
- State logging is not supported for a block (for example, Unit Delay) that takes in a bus in which some elements are removed.

See Also

More About

- “Group Signal Lines into Virtual Buses” on page 76-8
- “Variant Condition Propagation with Variant Sources and Sinks” on page 12-60
- Variant System Design

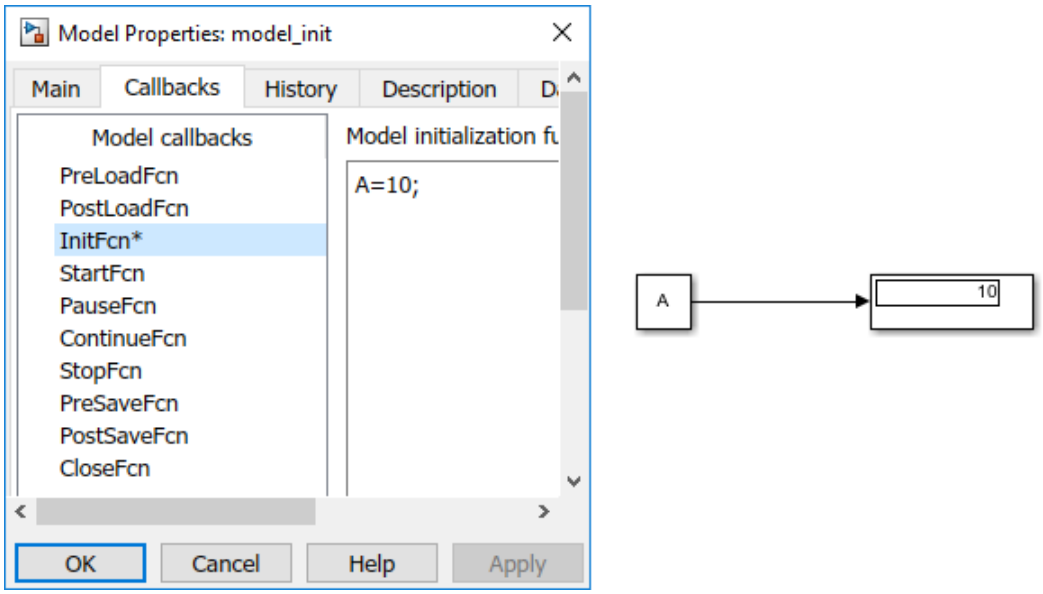
Initialization Function

Initialization function (`InitFcn`) is a type of callback that is executed or evaluated at the beginning of model compilation. You can use `InitFcn` in a model (model `InitFcn`) or a block (block `InitFcn`).

Note Variant controls can be defined only in model `InitFcn` callback.

Model InitFcn

The model `InitFcn` callback is used to initialize parameters and environment settings that are specific to the model used.



Note It is not recommended to use the model `InitFcn` callback to get the simulation status. If you simulate a model in `rtwbuild`, or SIL(software-in-the-loop), or Rapid Accelerator mode, the model `InitFcn` callback status may show as `'stopped'`. The model `InitFcn` callback must be used only to initialize parameters and environment settings that are specific to the model used.

Best practices for using Model InitFcn

- Do use model `InitFcn` to initialize data required for the model. For example, to initialize:
  - Variables used in model parameters
  - License checks for the required software
- Do not use model `InitFcn` to modify models other than self. This also means that the block `InitFcn` of Model block must not modify the parameters (and structure) of the referenced model.
- Do not use model `InitFcn` in the top model to overwrite any variable used in the referenced model. For example, if top and the referenced models use the variable ‘k’, the model `InitFcn` of the top model must not modify ‘k’ of the referenced model. In such modeling patterns, it is recommended that you use different variable names. Alternatively, you can use data dictionary.

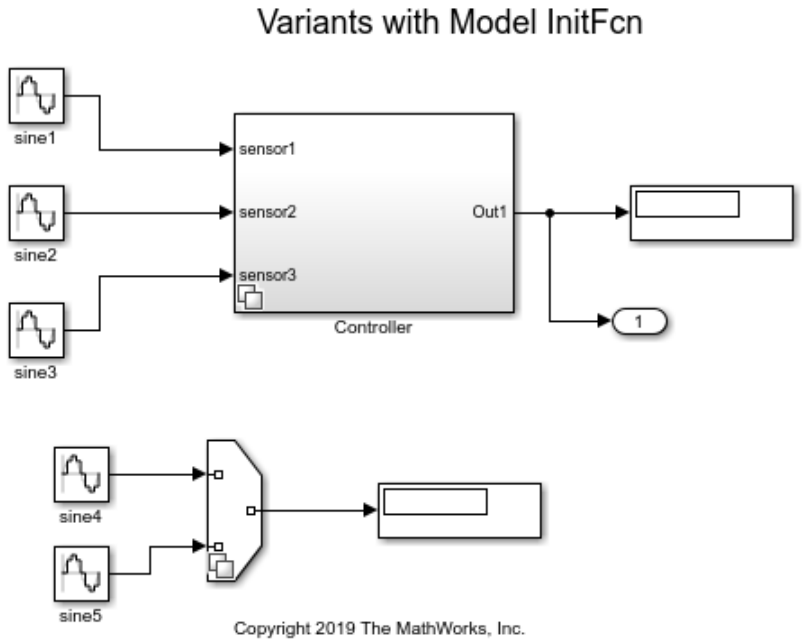
- Do not use simulation commands in model `InitFcn`. For example, using commands like, `set_param(ModelName, 'SimulationCommand', 'start')` or `set_param(ModelName, 'SimulationCommand', 'update')` in the model `InitFcn` are not recommended.

If you use the `InitFcn` callback for the model, edit-time checking for missing variables in block parameters is disabled for the entire model.

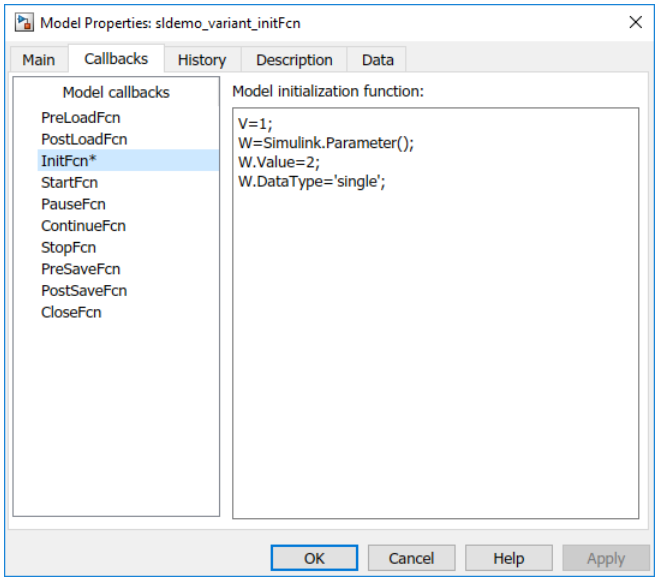
Variants with Model InitFcn

You can use model properties to define the callbacks for a model. For example, `PreLoadFcn`, `PostLoadFcn`, `InitFcn` callbacks. Model properties are used to view the model information, description, history, and callback functions. You can use the Property Inspector to view and edit model version properties, description history, and callback functions. For more information on Model properties, see “Manage Model Properties” on page 4-58.

Consider a model with Variant Subsystem and Variant Source blocks. For example, Variant - InitFcn.



In this model, the model `InitFcn` is used to initialize parameters for the model.

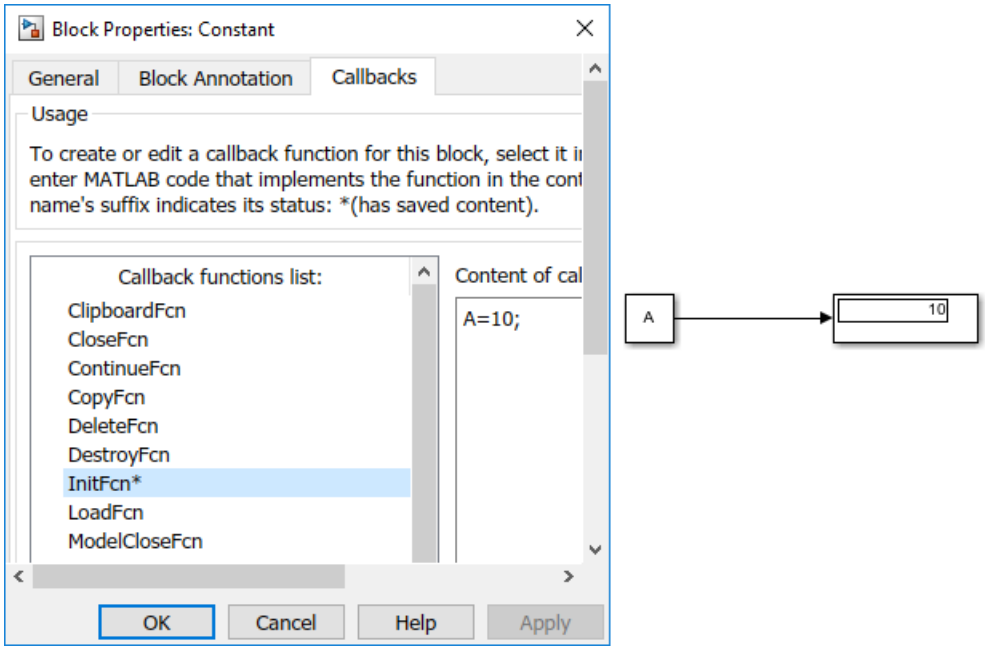


Similarly, these parameters can also be defined in `PreLoadFcn` or `PostLoadFcn`.

However, when the parameters are defined in `InitFcn` callback and if MATLAB workspace is cleared using the `Clear` command, the items in the workspace are cleared and they are re-created when you simulate the model. The items cleared will not be recreated when parameters are defined in `PreLoadFcn` or `PostLoadFcn`.

Block InitFcn

The block `InitFcn` callback is used to initialize block specific parameters and settings.



Best practices for using Block InitFcn

- Do not use block `InitFcn` to modify the parameters (or variables) of blocks other than self.
- Do not use block `InitFcn` on a child block to modify the parameters of the parent subsystem block or other child blocks. However, you can use block `InitFcn` on a parent subsystem block to modify the parameters of the direct child blocks.
- Do not use block `InitFcn` to make structural changes like adding or deleting block (`add_block` or `delete_block`).
- Do not use block `InitFcn` in the Model block to modify the parameters (and structure) of the referenced model.

If you use an `InitFcn` callback for a block, edit-time checking for missing variables in block parameters is disabled for that block.

See Also

Related Examples

- “Model Callbacks” on page 4-45
- “Block Callbacks” on page 4-49

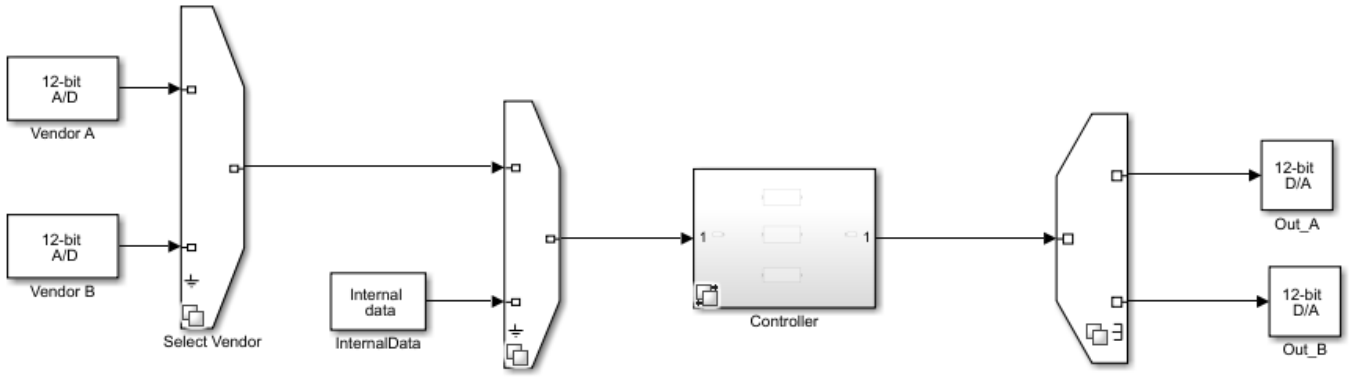
Analyze Variant Configurations in Models Containing Variant Blocks

A variant model can have one or multiple variant configurations associated with it. Using the Variant Manager, you can create variant configurations. Each variant configuration contains a set of variant control variable values which activates the variant choices in the model. Each variant configuration can be used to produce a specific implementation of the model. The number of variant configurations can be high depending on the number of variant control variables used in the model. Because of the high number of variant configurations, it can be difficult to make sure all the variant choices have been activated at least once, and that the model is covered completely for simulation and code generation. It is also difficult to ensure that the active, implemented model is different between different variant configurations. You can use the Variant Configuration Analysis tool in the Variant Manager to compare different variant configurations for a model. Additionally, you can get information on the dependent models and libraries used for a particular variant configuration.

The Variant Configuration Analysis tool also helps you to determine which blocks are used in different variant configurations. You can identify which blocks are unused and which are inside variant region and are always used. The unused blocks are highlighted in red, which represents untested and uncovered parts of the model. The heatmap view helps you determine the similarities and differences in the active, implemented model between different variant configurations. The analysis results can be used to refine the variant configurations and to update the model to provide full simulation coverage across all variant configurations.

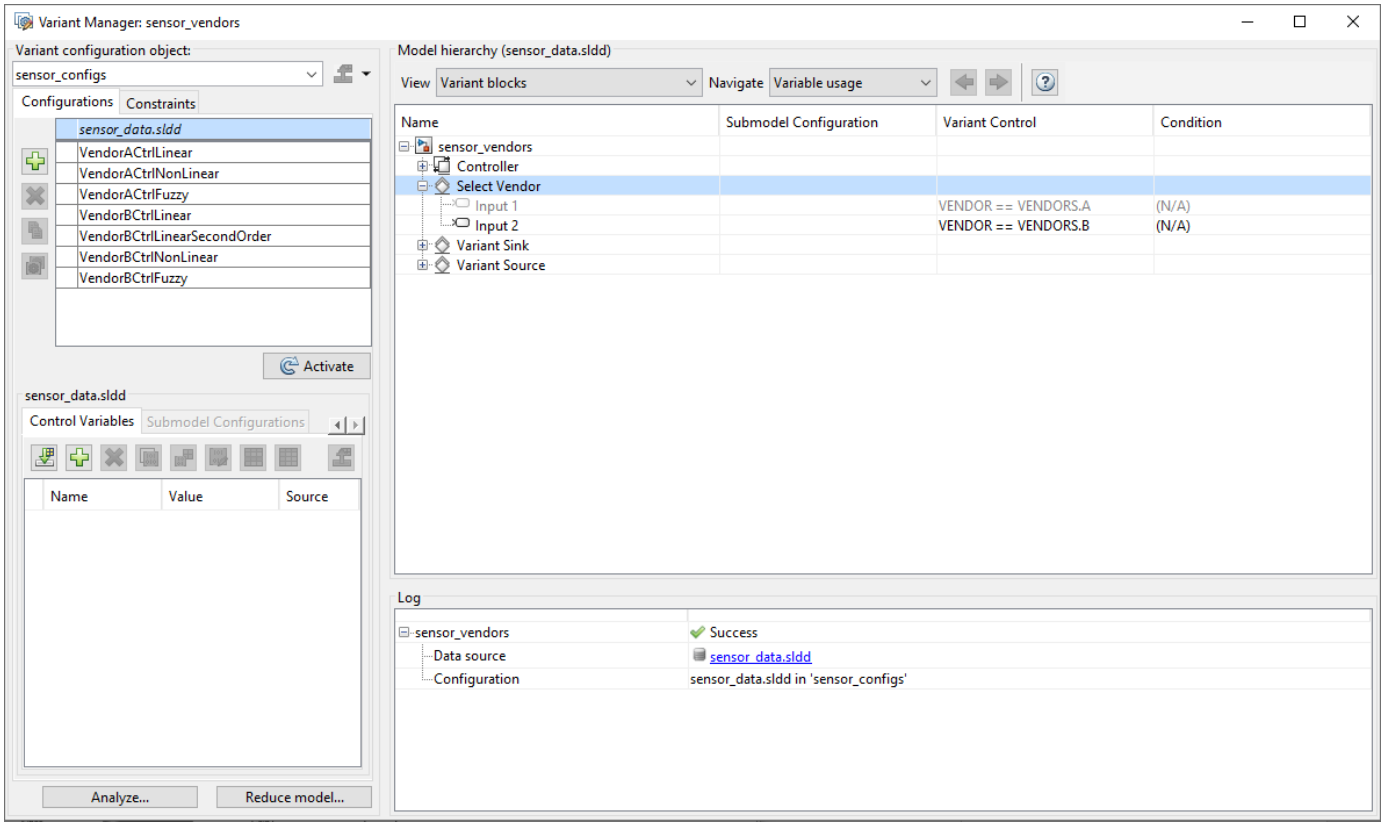
Analyze a Model with Variant Configurations

1. Open a model containing variant blocks. For example, Sensor Vendors.

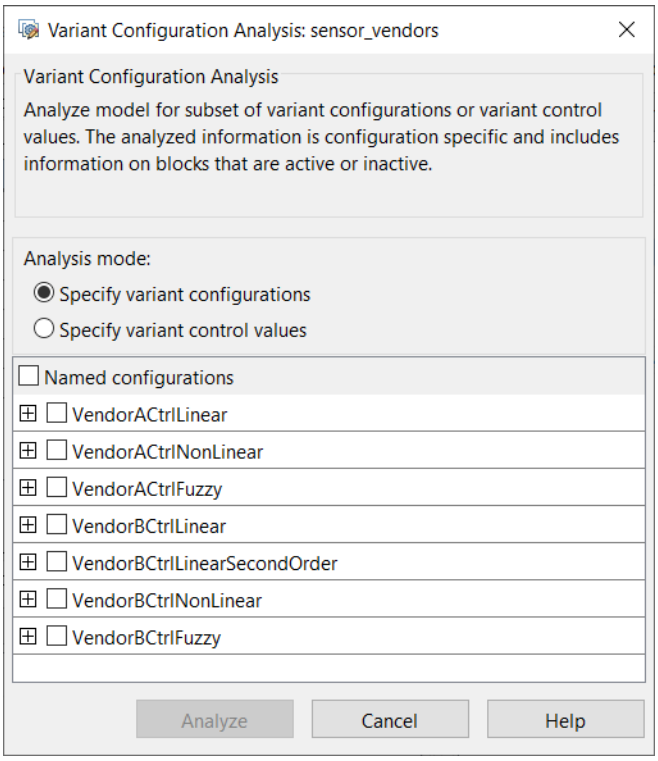


   This model contains different vendor sources for the sensors and different controller implementations based on sensor input. The choice of vendor sensors is modeled with Variant Source and Variant Sink blocks. The different controller choices are modeled using variant subsystems.

2. Right-click the variant badge and select **Open in Variant Manager**. The Variant Manager opens and displays the predefined configurations. Alternatively, select a variant block and then in the **Variant** tab of the toolstrip select **Variant Manager**.

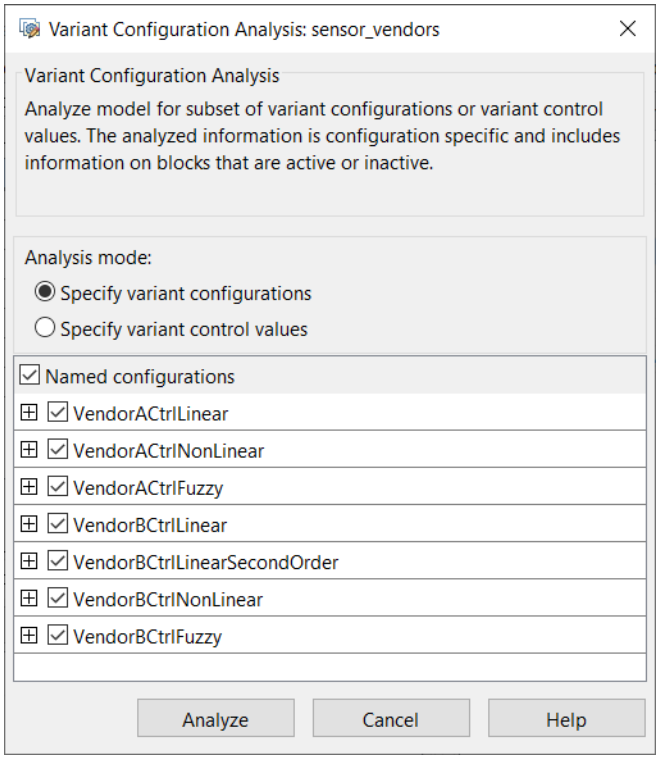


3 Click **Analyze**. The Variant Configuration Analysis dialog box opens.

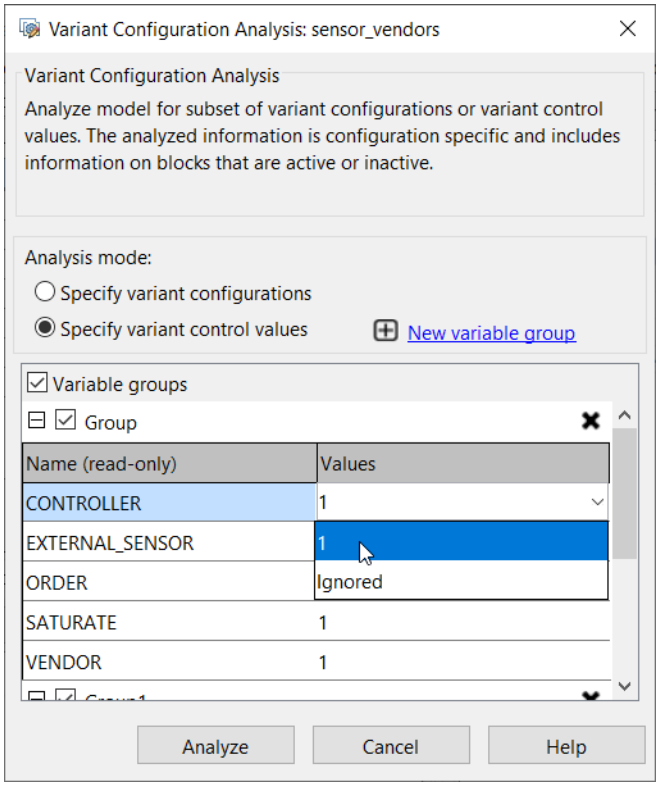


4 In the **Analysis mode**, select the required option.

- **Specify variant configurations**: Select the required Variant Configurations or select **Named configurations** to select all configurations.

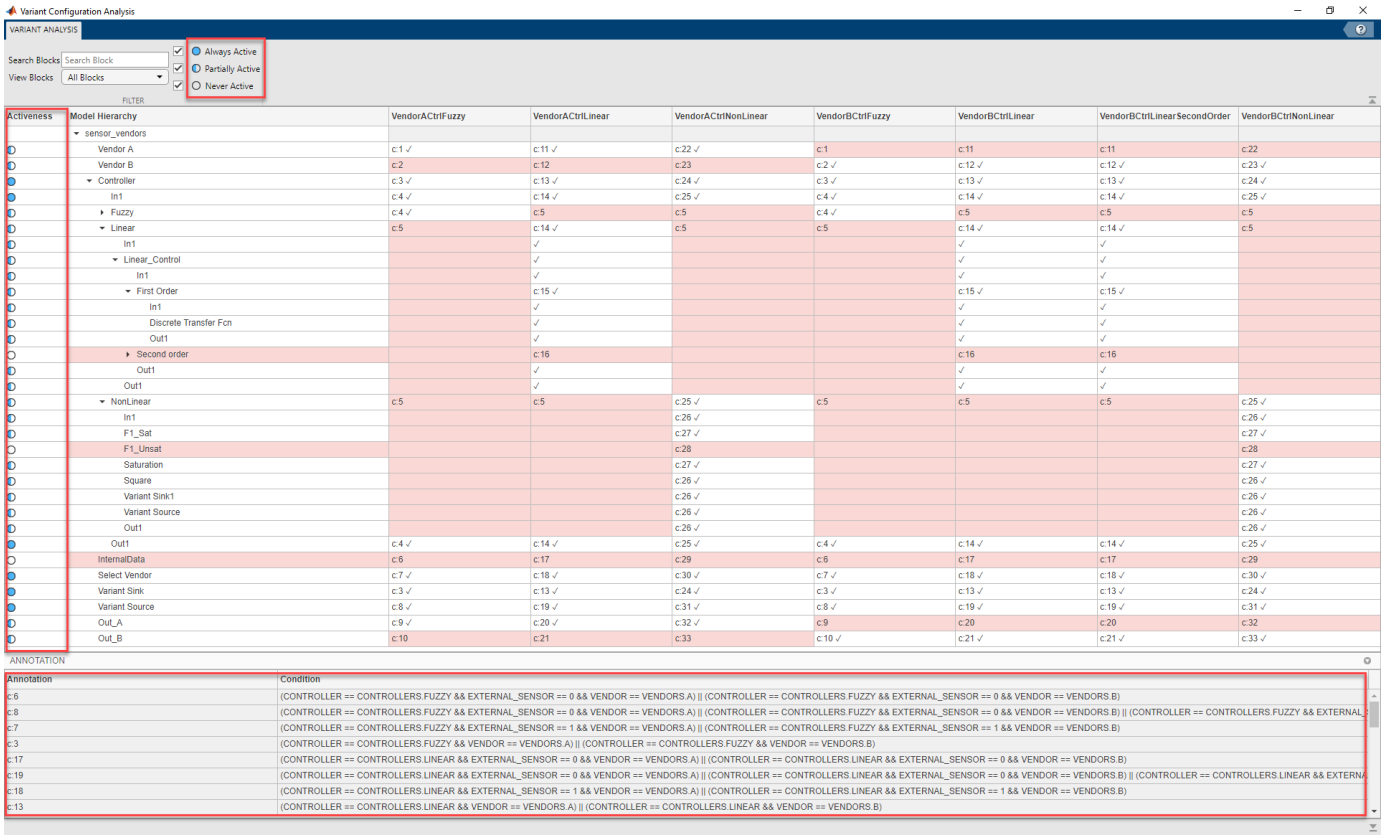


- **Specify variant control values**: If you select this option, you can create multiple variable groups that correspond to different configurations. To create a new variable group, click **New variable group** and set the values for variant control variables. You can either specify a variant control value or select `Ignored` from the drop-down list. If you select `Ignored`, then that variant control variable is not considered while analyzing the model.



5 Click **Analyze**. The report for analyzed variant configurations opens.

Note In this example, option **Specify variant configurations** is selected in **Analysis mode**.



The report displays the tree-table view of the model hierarchy and all of the analyzed variant configurations. Each row in the table corresponds to a block in the model and each column represents a variant configuration. A check mark indicates that the corresponding block is active in the corresponding variant configuration. Entries highlighted in red indicate that the block is inactive for that specific variant configuration. For example, in the above image, the **Vendor A** block is active in variant configurations **VendorACtrlLinear**, **VendorACtrlFuzzy** and **VendorACtrlNonLinear** and is inactive in variant configurations **VendorBCtrlFuzzy**, **VendorBCtrlLinear**, **VendorBCtrlSecondOrder**, and **VendorBCtrlNonLinear**.

The annotations in the report (“c:#”) corresponds to a variant condition. The variant conditions are dependent on which variant configuration is active. The mapping between the annotation and the condition is shown in the image. The variant conditions help you in understanding why a block is active for given variant configuration (the variant condition is evaluated based on the values of the control variable defined in the configuration).

The controls on the **Variant Configuration Analysis** window allow you to perform the following actions:

- Search blocks.
- Filter the results to selectively display blocks.
- Selectively display analyzed blocks.

View Blocks

Selectively display blocks in the **Variant Analysis** window by:

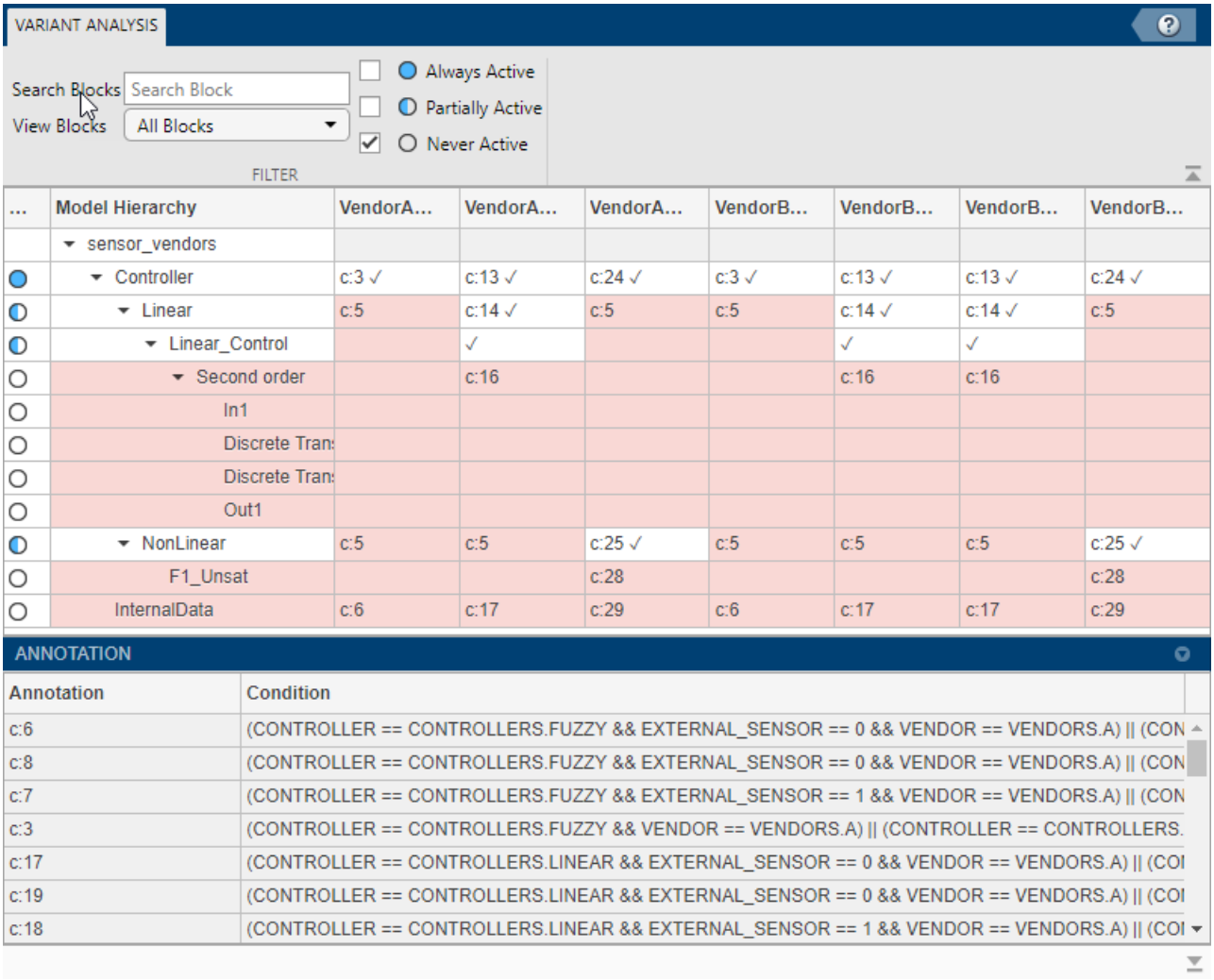
- Selecting **Variant** to display only variant blocks in the model.
- Selecting **Hierarchical** to display all hierarchical blocks (for example, Subsystem or Model blocks) in the model. This view allows you to explore subsystems or model references one-by-one instead of looking through the complete model.
- Selecting **All blocks** to display all blocks in the model.

Block Activeness

Selectively display blocks by their activeness in the **Variant Configuration Analysis** window by:

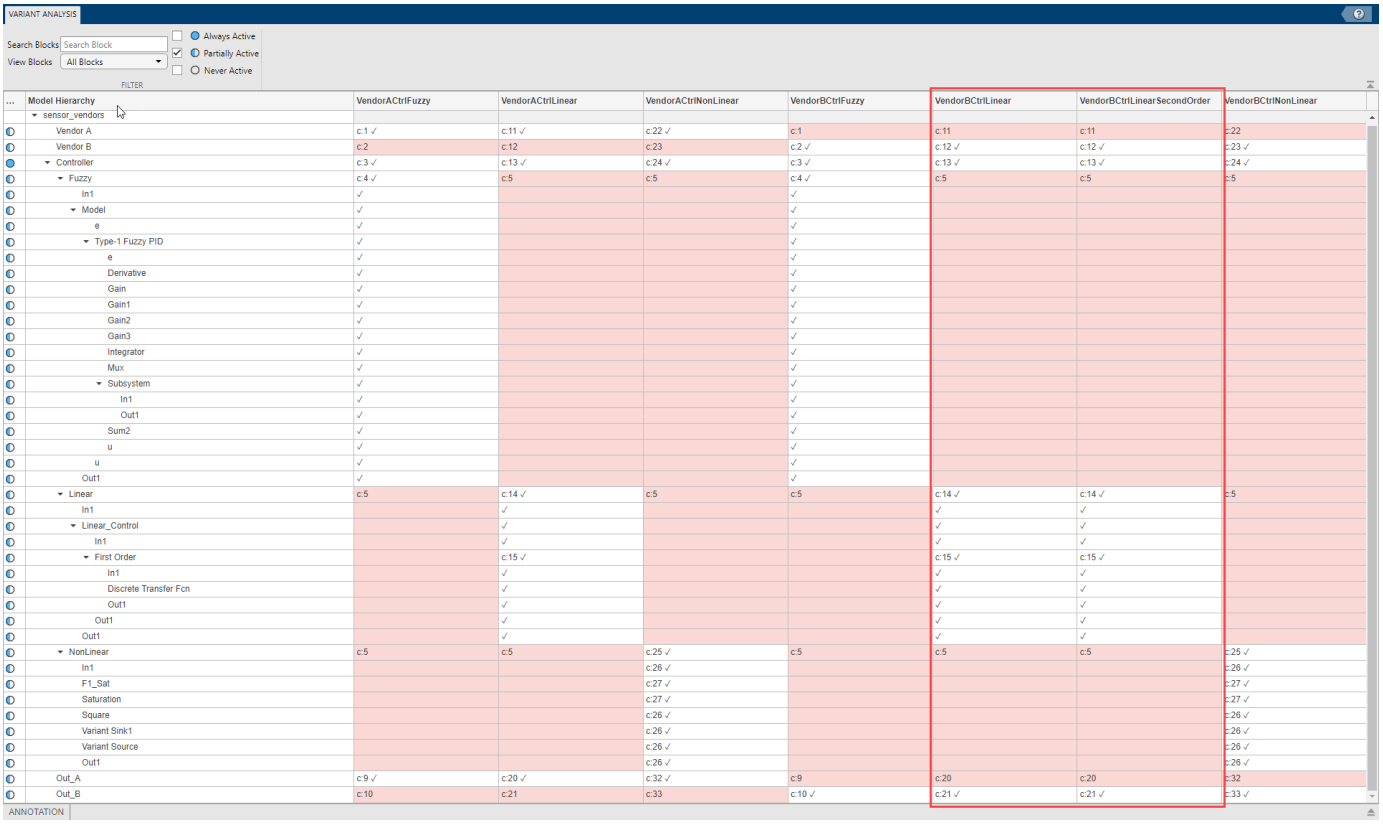
- Selecting **Always Active** to display blocks that are always active in the model.
- Selecting **Partially Active** to display blocks that are active in some configurations and inactive in others among the selected configurations.
- Selecting **Never Active** to display blocks that are never active in the model. These blocks are additionally highlighted in red to indicate that they are unused parts of the model and require fixing.

For example, the image below shows a model with several unused blocks. **Never Active** option is selected to view the unused blocks.



The blocks in the **Second order** controller inside the **Linear_Control** and **F1_Unsat** filter are unused. To make these unused blocks part of the active model in at least one of the variant configurations, modify the model or update the variant configurations.

The image below shows a model with two identical variant configurations. **Partially Active** option is selected to get this result.

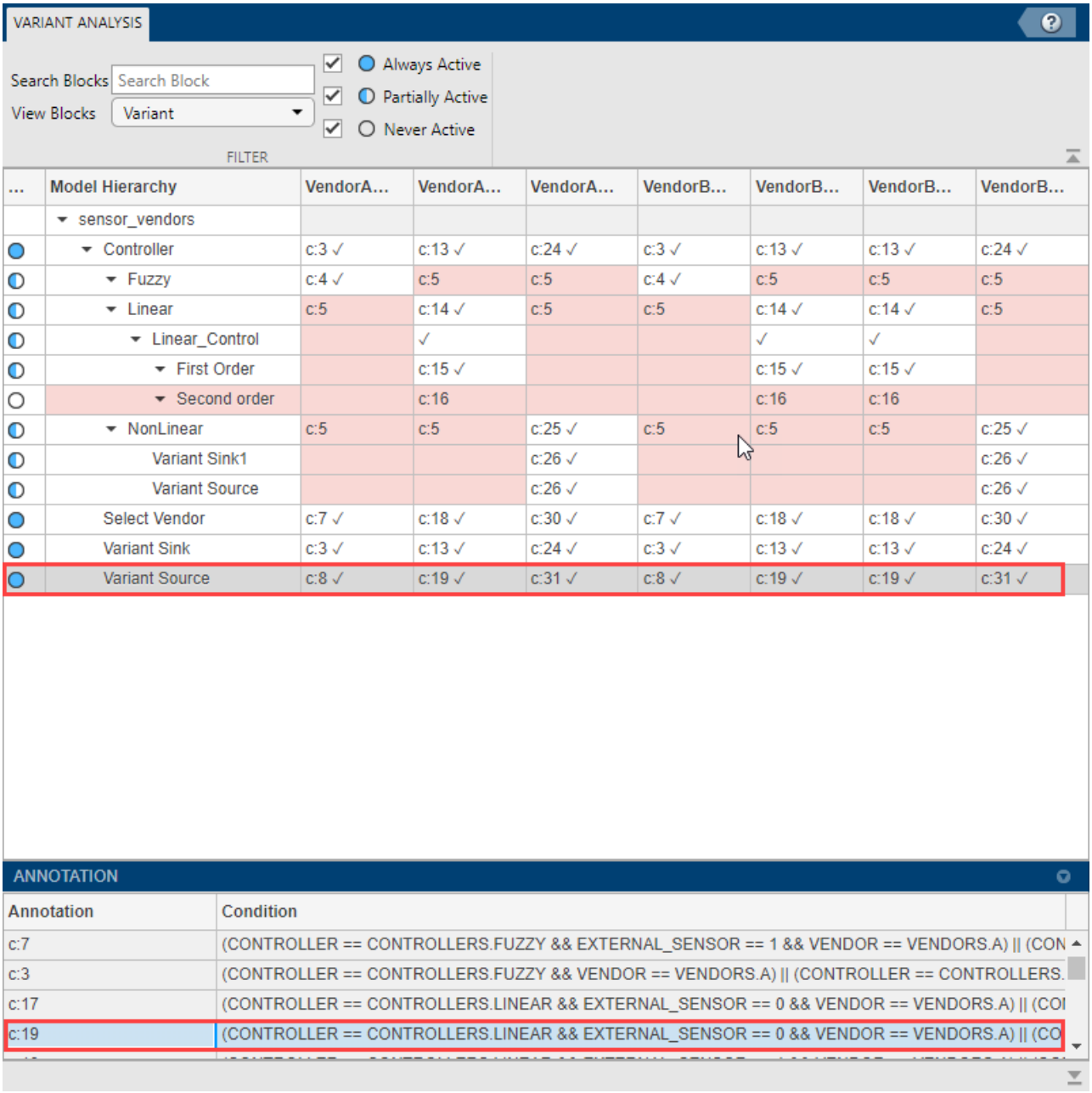


Model Hierarchy	VendorACtrlFuzzy	VendorACtrlLinear	VendorACtrlNonLinear	VendorBCtrlFuzzy	VendorBCtrlLinear	VendorBCtrlLinearSecondOrder	VendorBCtrlNonLinear
sensor_vendors							
Vendor A	c:1 ✓	c:11 ✓	c:22 ✓	c:1	c:11	c:11	c:22
Vendor B	c:2	c:12	c:23	c:2 ✓	c:12 ✓	c:12 ✓	c:23 ✓
Controller	c:3 ✓	c:13 ✓	c:24 ✓	c:3 ✓	c:13 ✓	c:13 ✓	c:24 ✓
Fuzzy	c:4 ✓	c:5	c:5	c:4 ✓	c:5	c:5	c:5
In1	✓			✓			
Model	✓			✓			
e	✓			✓			
Type-1 Fuzzy PID	✓			✓			
e	✓			✓			
Derivative	✓			✓			
Gain	✓			✓			
Gain1	✓			✓			
Gain2	✓			✓			
Gain3	✓			✓			
Integrator	✓			✓			
Mux	✓			✓			
Subsystem	✓			✓			
In1	✓			✓			
Out1	✓			✓			
Sum2	✓			✓			
u	✓			✓			
u	✓			✓			
Out1	✓			✓			
Linear	c:5	c:14 ✓	c:5	c:5	c:14 ✓	c:14 ✓	c:5
In1		✓			✓	✓	
Linear_Control		✓			✓	✓	
In1		✓			✓	✓	
First Order		c:15 ✓			c:15 ✓	c:15 ✓	
In1		✓			✓	✓	
Discrete Transfer Fcn		✓			✓	✓	
Out1		✓			✓	✓	
Out1		✓			✓	✓	
Out1		✓			✓	✓	
NonLinear	c:5	c:5	c:25 ✓	c:5	c:5	c:5	c:25 ✓
In1			c:26 ✓				c:26 ✓
F1_Sat			c:27 ✓				c:27 ✓
Saturation			c:27 ✓				c:27 ✓
Square			c:26 ✓				c:26 ✓
Variant Sink1			c:26 ✓				c:26 ✓
Variant Source			c:26 ✓				c:26 ✓
Out1			c:26 ✓				c:26 ✓
Out_A	c:9 ✓	c:20 ✓	c:32 ✓	c:9	c:20	c:20	c:32
Out_B	c:10	c:21	c:33	c:10 ✓	c:21 ✓	c:21 ✓	c:33 ✓

The variant configurations **VendorBCtrlLinear** and **VendorBCtrlSecondOrder** have no differences between them. This indicates that the resulting active model for both these configurations will be same. To resolve this, update the variant configurations or update the model appropriately.

Viewing Annotation

The annotations in the table correspond to a variant condition. The variant conditions depend on the active variant configuration. The variant conditions help you to understand whether the block is active for the given variant configuration. The variant condition is evaluated based on the values of the control variable defined in the configuration.



Click on the required annotation to view the block that has the selected annotation.

See Also

More About

- “Variant Manager Overview”
- `Simulink.VariantConfigurationAnalysis`
- Variant System Design

Variants Example Models

The Simulink Variants Example models help you to understand and use the variant blocks and features.

Variants Example Models	Goal	Related Topics
Variant Subsystems	Specify an active variant control for a variant subsystem programmatically.	“Variant Subsystems” on page 12-136
Variant Subsystems - Enumerations	Improve readability in the conditions of the `Variant` object.	“Variant Subsystems” on page 12-136
Adaptive Interfaces for Variant Subsystems	Observe how the Variant Subsystem block adapts its interface to the connected blocks.	“Adaptive Interface for Variant Subsystems” on page 12-91
Variant Source and Variant Sink Blocks	Understand variant condition propagation in a model containing Variant Source or Variant Sink blocks.	“Variant Source and Variant Sink Blocks” on page 12-143
Manual Variant Source and Sink Blocks	Understand propagation of the active variant choice in a model containing Manual Variant Source or Manual Variant Sink blocks.	Manual Variant Source Manual Variant Sink
Variant Source and Sink blocks in car wiper motor	Understand how to use Variant Source and Variant Sink blocks in a car windshield skeleton model.	
Variant Management	Create and manage various variant configurations of a model through the Variant Manager.	“Variant Manager Overview”
Variant Reducer	Reduce a variant model based on specified variant configurations through Variant Manager.	“Reduce Models Containing Variant Blocks” on page 12-77
Variant Sensors	Understand how variant conditions propagate in a model containing cascaded Variant Source blocks.	
Masking a Variant Subsystem	Understand how to select a variant choice for a variant subsystem using mask parameters and mask initialization code.	“Approaches to Control Active Variant Choice of a Variant Subsystem” on page 12-124

Variants Example Models	Goal	Related Topics
Variants with Function-Call Subsystem	Understand propagation of variant condition in a model containing Function-Call Subsystem block.	“Propagate Variant Condition to Conditional Subsystem” on page 12-149
Variant Simulink Functions - Inherit Condition	Understand how a Variant Simulink Function can optimally exist based on its Function-callers.	
Variant Simulink Functions - Specified Condition	Understand how to conditionally define the existence of a Simulink Function.	
Variant Condition Propagation to Subsystems	Understand propagation of variant conditions to different types of subsystems.	“Propagating Variant Conditions to Subsystems” on page 12-132
Variant Condition Propagation to Conditionally Executed Subsystems	Understand propagation of variant condition in a model containing a conditional subsystem.	“Propagate Variant Condition to Conditional Subsystem” on page 12-149
Variant Subsystem with Conditionally Executed Systems	Understand the modeling of Variant Subsystem with Enable Subsystem as choice.	“Variant Systems with Conditional Systems” on page 12-96
Export function model with Variant Subsystem	Understand the export function modeling of Variant Subsystem with Function-Call blocks as choice.	“Variant Systems with Conditional Systems” on page 12-96
Variant Conditions and Data Stores	Understand the functioning of local Data Store Memory blocks with the Variant blocks.	
Variant Condition Propagation and Model Blocks	Understand propagation of variant conditions from the output port of the Model block.	“Variant Condition Propagation with Model Block” on page 12-65
Controlling and Stopping Variant Condition Propagation	Control or stop the Variant condition propagation upstream and downstream for a model containing the Subsystem block.	“Control Variant Condition Propagation” on page 12-146
Generate preprocessor conditionals for Variant Subsystems	Generate and understand code for a model containing the Variant Subsystem block.	“Use Variant Subsystem To Generate Code That Uses C Preprocessor Conditionals” (Embedded Coder)

Variants Example Models	Goal	Related Topics
Dimension Variants	Understand how to generate code for a model with dimension variants.	
Model Reference Variants	Understand how to use Model blocks as variants.	“Model Reference Variants”
Model Reference Variants - Enumerations and Reuse	Understand the enumerations and reuse capabilities of a model.	“Model Reference Variants”
Generate preprocessor conditionals for model subsystem	Generate and understand the code for Model blocks within the Variant Subsystem block.	“Use Variant Models to Generate Code That Uses C Preprocessor Conditionals” (Embedded Coder)
Bus - Variant Condition	Understand how to simulate or generate code from bus signals with variant conditions.	“Variant Elements within Buses” on page 12-104
Variant Subsystem - Verification &Validation Workflow	Understand how Verification and Validation activities are done on Variant models.	

See Also

More About

- “Introduction to Variant Controls” on page 12-24
- “Create a Simple Variant Model” on page 12-36
- Variant System Design

Approaches to Control Active Variant Choice of a Variant Subsystem

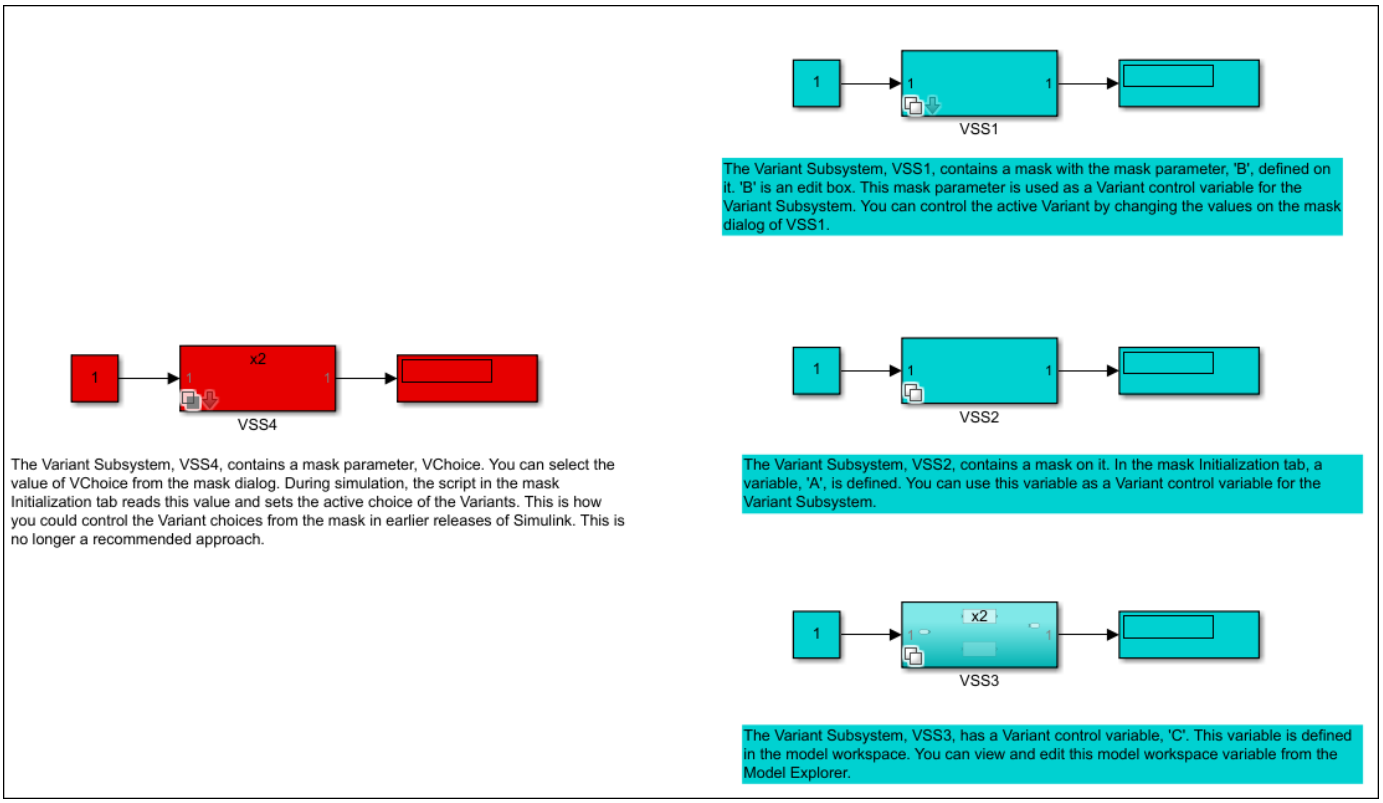
In this section...

This example shows different approaches to control the active choice of a Variant Subsystem from a mask or a model workspace. For more information on Variant Subsystems, see “Variant Subsystems”.

Model

To open the Simulink model, type `slexVariantSubsystemCtrlFromMaskandModelWks` in the MATLAB Command Window.

Three Simulink models are highlighted in green, and one model is highlighted in red. The models in green represent the recommended approaches to control the active choice of Variant Subsystems. These approaches let you limit the scope of the variant control variable, avoid name conflicts, and establish a clear ownership of the variable between Variant blocks. They also allow you to use the same names for variables in different scopes.



Limitations in Recommended Approaches

The recommended approaches:

- Must be implemented only on Variant Subsystem blocks. The Variant Sink and the Variant Source blocks do not support these approaches.
- Work only if the **Variant control mode** parameter is set to `expression` and the **Variant activation time** parameter is set to `update diagram`.
- Do not support using `Simulink.Variant` objects or `Simulink.Parameter` as variant control variables.
- Do not support using model arguments variables as variant control variables.

Approach 1: Use Mask Parameter as a Variant Control Variable

1 Consider the model with Variant Subsystem block VSS1.

The VSS1 subsystem specifies two potential variants, `x2` and `x3`. The control expression for `x2` is `B == 1` and for `x3` is `B == 2`. The variable `B` is a mask parameter. To view the properties of `B`:

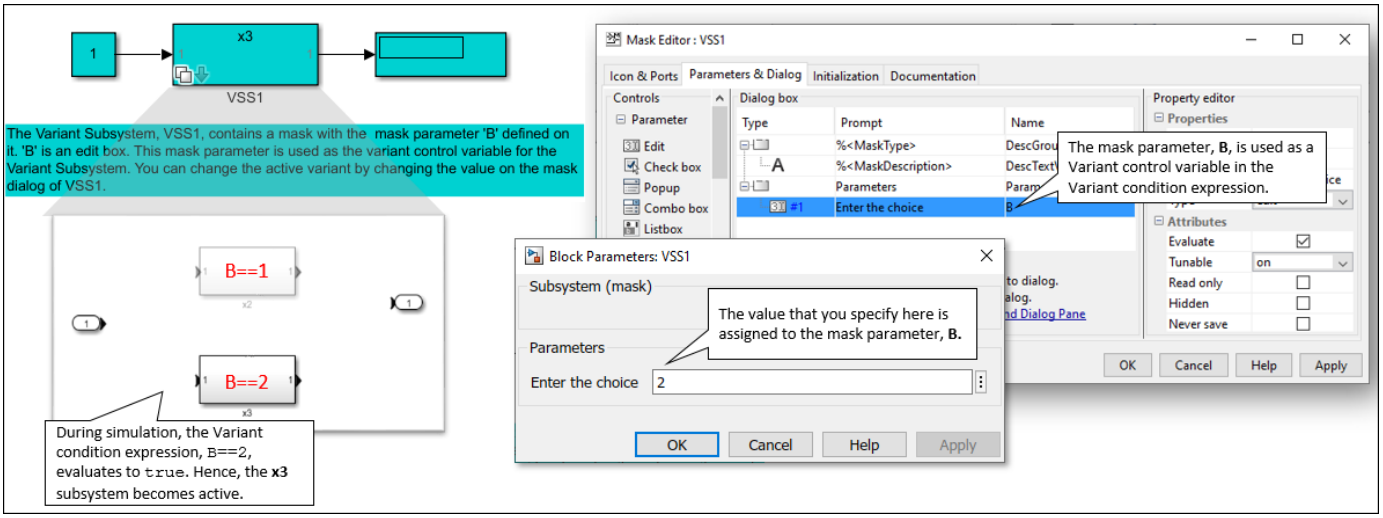
a Right-click the VSS1 subsystem.

b Select **Mask > Edit Mask**. In the **Parameters & Dialog** pane, under **Parameters**, the **Prompt** column specifies the label of the parameter on the mask dialog box, and the **Name** column specifies the name of the mask parameter. In this example, **Prompt** is specified as `Enter the choice`, and **Name** is specified as `B`.

2. To open the mask dialog box, double-click the VSS1 subsystem. During simulation, the value that you specify here is mapped to the underlying variable `B`, which is then used to evaluate the variant condition expressions associated with the block.

   In this example, the default value of **Enter the choice** is 2. When you simulate this model, the variant condition `B == 2` evaluates to `true`. The x2 subsystem becomes inactive, and the x3 subsystem becomes active.

3. To modify the active choice, specify the value as 1 in the mask dialog box, then simulate the model again. During simulation, the value of the `B` is set to 1 which in turn evaluates the Variant condition, `B== 1` to true. The x2 subsystem becomes active, and the x3 subsystem becomes inactive.



Approach 2: Use Mask Initialization Variable as a Variant Control Variable

1. Consider the model with Variant Subsystem block VSS2.

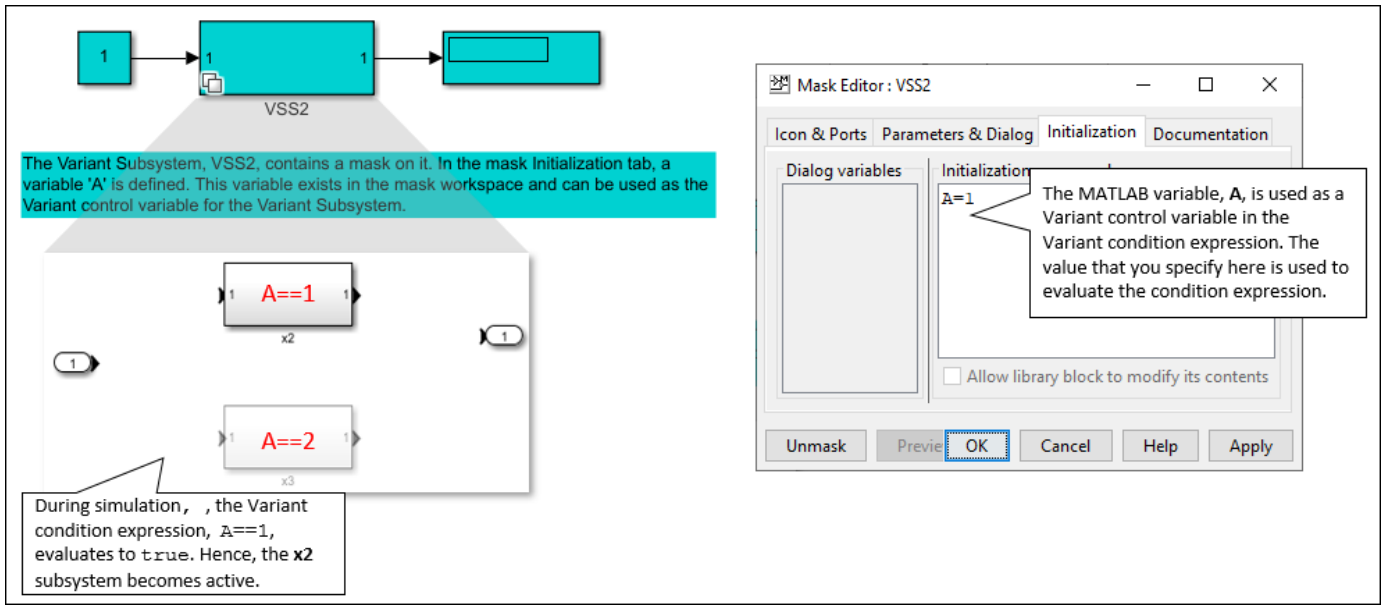
   In the VSS2 subsystem, the control expression for `x2` is `A == 1` and for `x3` is `A == 2`. The variable `A` used in the control expression is a regular MATLAB variable that is defined in the **Initialization** tab of the mask workspace. To view the properties of `A`:

   a. Right-click the VSS2 subsystem.

   b. Select **Mask > Edit Mask**. In the **Initialization** tab, under **Initialization commands**, the value of `A` is set to `1`.

   During simulation, this value is used to evaluate the variant condition expressions associated with the block. When you simulate this model, the variant condition `A == 1` evaluates to `true`. The x2 subsystem becomes active, and the x3 subsystem becomes inactive.

2. To modify the active choice, specify the value of `A` as 2 in the **Initialization** tab, then simulate the model again. During simulation, `A == 2` evaluates to `true`. The x2 subsystem becomes active, and the x3 subsystem becomes inactive.



Approach 3: Use Model Workspace Variable as a Variant Control Variable

1. Consider the model with Variant Subsystem block VSS3.

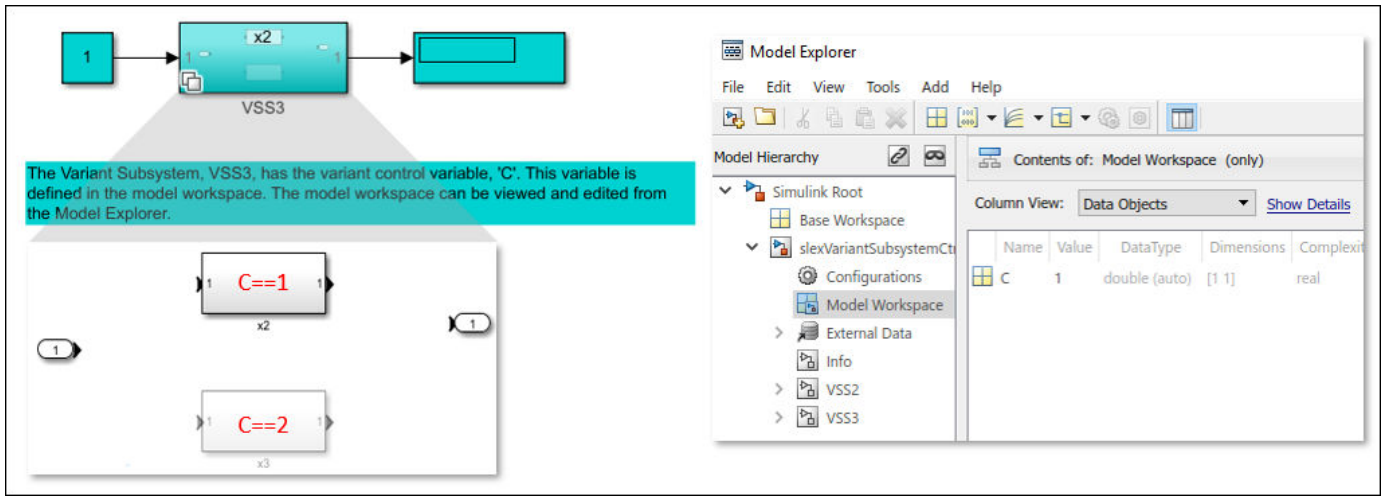
   In the VSS3 subsystem, the control expression for x2 is `C == 1` and for x3 is `C == 2`. The variable `C` used in the condition expression is a regular MATLAB variable that is defined in the model workspace. To view the properties of `C`:

   a. On the **Modeling** tab, click **Model Explorer**.

   b. In the **Model Hierarchy** pane, click **Model Workspace**. The value of `C` is set to `1`.

   During simulation, this value is used to evaluate the variant condition expressions associated with the block. When you simulate this model, the variant condition `C == 1` evaluates to `true`. The x2 subsystem becomes active, and the x3 subsystem becomes inactive.

2. To modify the active choice, specify the value of `C` as `2`, then simulate the model again. During simulation, the Variant condition `C == 2` evaluates to `true`. The x2 subsystem becomes active, and the x3 subsystem becomes inactive.



Approach 4: Use Mask Initialization Script to Control Active Variant Choices

This approach is not recommended for controlling the active variant choice of Variant Subsystems. However, if the **Variant control mode** of the subsystem is set to `label` mode, you can follow this approach. For more information, see “Mask a Variant Subsystem” on page 39-82.

See Also

Related Examples

- “Control Active Choice of Locked Custom Library Variant Subsystem Using Mask Parameter” on page 12-129

Control Active Choice of Locked Custom Library Variant Subsystem Using Mask Parameter

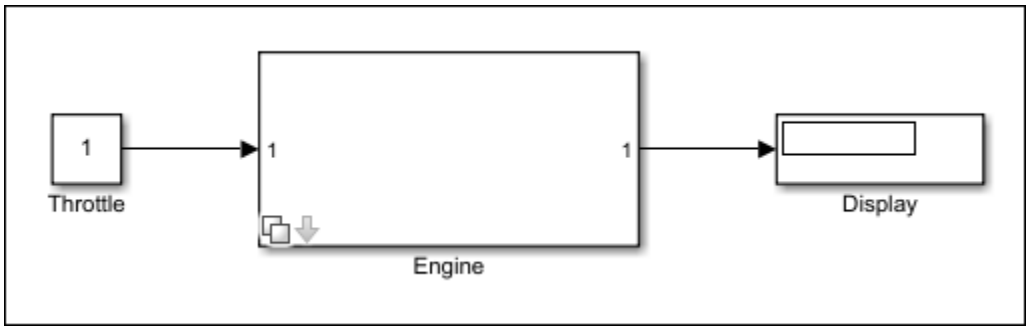
In this section...

This example shows how to control the active choice of a Variant Subsystem that belongs to a locked custom library by using a mask parameter as Variant control variable. Mask parameters limit the scope of the variable, which allows you to use the same names for control variables in different scopes. This example includes promoted parameters and control expressions with enumeration classes to simplify the interface of the mask dialog and control expressions with enumeration classes to improve the readability of the variant condition expressions. For more information on Variant Subsystems, see “Variant Subsystems”.

Model

To open the Simulink model, type `slexVariantSubsystemUsingMaskAndEnums` in the MATLAB Command Window.

Consider the Engine subsystem block in the locked custom library, slexVarEngineLibrary.

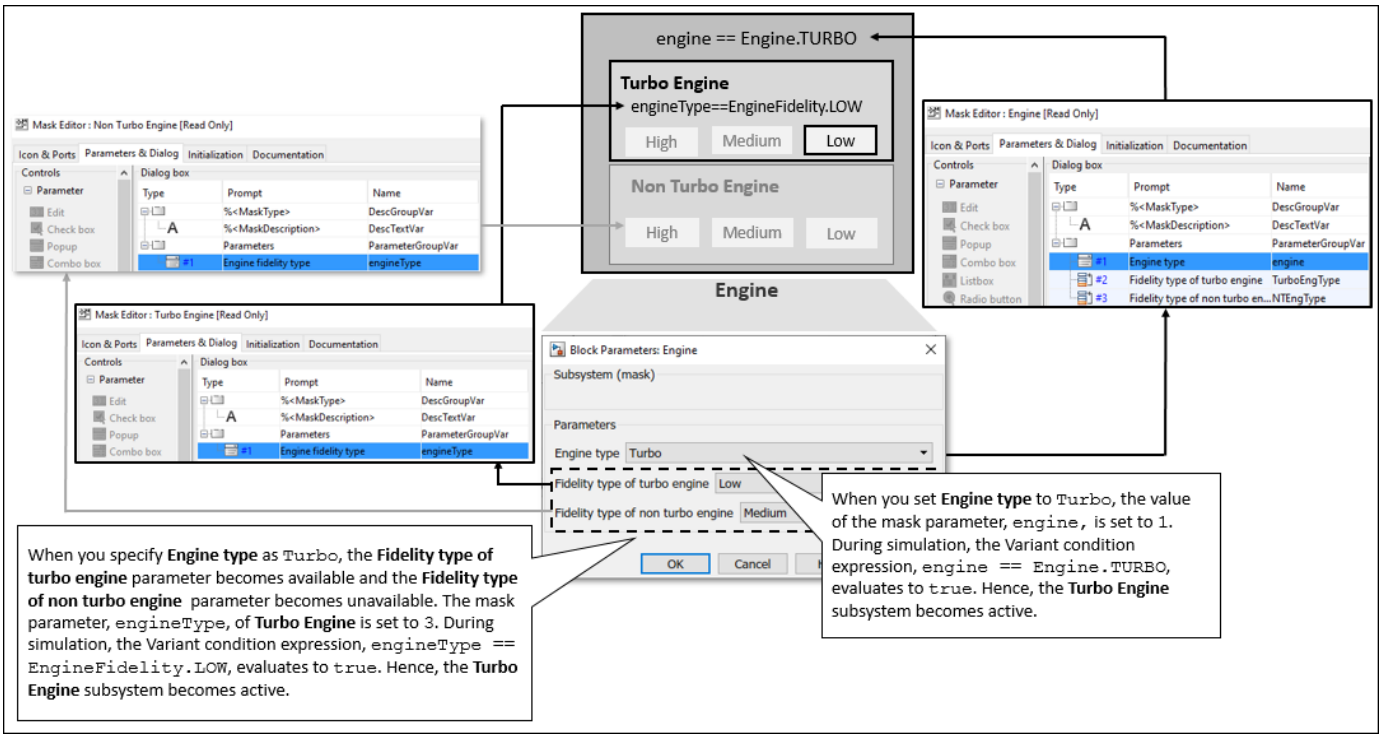


The mask dialog box of the subsystem contains these parameters:

- **Engine type**: When you select a value for this parameter, Simulink assigns the index of that value to the mask parameter `engine`. During simulation, the value of `engine` is used to evaluate the variant condition expressions to activate or deactivate the underlying Turbo Engine and Non Turbo Engine subsystems.
- **Fidelity type for turbo engine**: This parameter becomes available only if the **Engine type** parameter is set to `Turbo`. This parameter is promoted on to the mask dialog of the Engine subsystem from the underlying layer of the Turbo Engine subsystem. When you select a value for this parameter, Simulink assigns the index of that value to the mask parameter `enginetype` of the Turbo Engine subsystem. During simulation, the value of `enginetype` is used to evaluate the variant condition expression to activate or deactivate the underlying High, Medium, and Low subsystems.
- **Fidelity type for non turbo engine**: This parameter becomes available only if the **Engine type** parameter is set to `Non Turbo`. This parameter is promoted on to the mask dialog of the Engine subsystem from the underlying layer of the Non Turbo Engine subsystem. When you select a value for this parameter, Simulink assigns the index of that value to the mask parameter `enginetype` of

the Non Turbo Engine subsystem. During simulation, the value of `enginetype` is used to evaluate the variant condition expression to activate or deactivate the underlying High, Medium, and Low subsystems.

The scope of `enginetype` in the Turbo Engine subsystem is different from the scope of `enginetype` in the Non Turbo Engine subsystem. `enginetype` of Turbo Engine is visible only to the underlying layers of the Turbo Engine subsystem. Similarly, `enginetype` of Non Turbo Engine is visible only to the underlying layers of the Non Turbo Engine subsystem. Limiting the scope by using mask parameters as Variant control variables allows you to use the same name for variables with holding different values in the Turbo Engine and the Non Turbo Engine subsystems.



Switch Between Active Choices

1. To simulate the model, on the **Simulation** tab, click Run. On the mask dialog of the Engine subsystem, the **Engine type** parameter is set to `Non Turbo`, and the **Fidelity type for non turbo engine** is set to `Medium`. As these parameters are mapped to the index of the mask parameters `engine` and `engineType`, the value of `engine` is set to `2`, and the value of `engineType` is set to `1`. Here, `2` specifies the index of the `Non Turbo` option, and `1` specifies the index of the `High` option.

   During simulation, the condition expressions `engine == Engine.NONTURBO` and `engineType == EngineFidelity.MEDIUM` evaluate to `true`. Here, `Engine` and `EngineFidelity` are integer-based enumeration classes defined in `Engine.m` and `EngineFidelity.m` files.

   The NonTurbo Engine subsystem becomes active and the Turbo Engine subsystem becomes inactive.

2. To modify the active choice, select `Turbo` in the mask dialog box, then simulate the model again. During simulation, the value of `engine` is set to `1`, which evaluates the variant condition `engine`

`== Engine.TURBO` to `true`. The Turbo Engine subsystem becomes active, and the Non Turbo Engine subsystem becomes inactive.

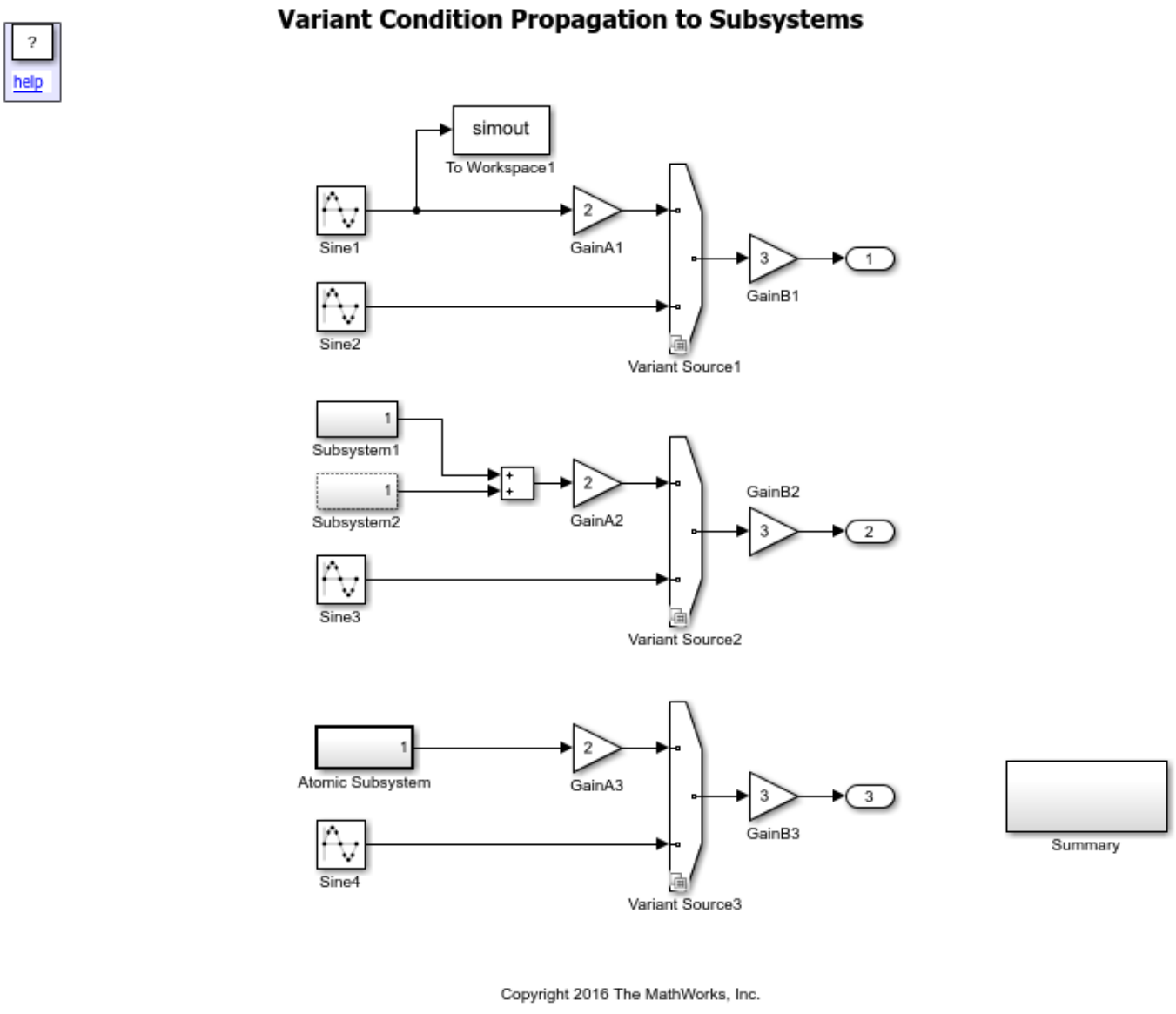
See Also

Related Examples

- “Approaches to Control Active Variant Choice of a Variant Subsystem” on page 12-124

Propagating Variant Conditions to Subsystems

A Subsystem can be virtual or atomic. Simulink propagates variant conditions differently to such Subsystems. This example shows the propagation of variant conditions from Inline variants to Subsystem blocks. Consider a model as shown:



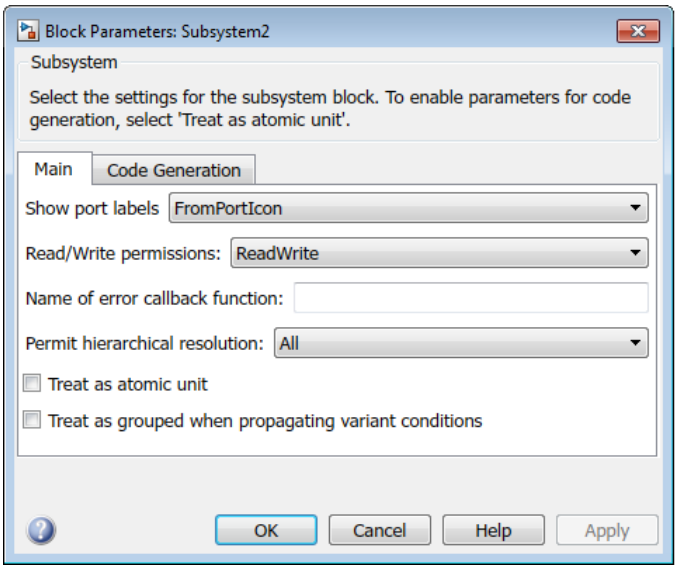
Click **Simulation** > **Run** to simulate this model and see the variant conditions being propagated from the Variant Source blocks to the blocks connected to it.

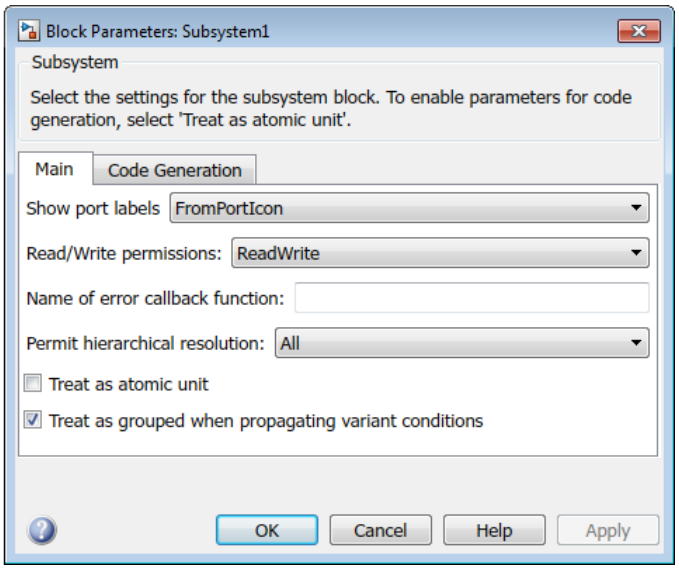
The variant condition annotation helps you visualize the propagated conditions. To be able to view the variant condition annotation, click **Display** > **Blocks** > **Variant Condition Legend**.

The model contains three Variant Source blocks: `Variant Source1` , `Variant Source2` , and `Variant Source3` , respectively.

`Variant Source1` contains conditions `V = 1` and `V = 2` at inport. The variant condition `V = 1` propagates to `GainA1` while `V = 2` propagates to `Sine2` . The `Sine1` block does not get any propagated variant conditions because it is connected to a block, which is always consumed irrespective of the variant condition. If the `To Workspace block1` did not exist or was commented-out before simulating the model, variant condition `V = 1` propagates to `Sine1` .

`Variant Source2` is connected to virtual subsystems `Subsystem1` and `Subsystem2` that have identical contents, a `Sine Wave` block connected to a `To Workspace` and an `Output` blocks. `Subsystem1` is a grouped virtual subsystem (**Treat as grouped when propagating variant conditions** is selected) while Subsystem2 (**Treat as grouped when propagating variant conditions** is clear) is an ungrouped virtual subsystem.





A Subsystem block becomes a grouped virtual subsystem when you select the **Treat as grouped when propagating variant conditions** checkbox in the block parameters dialog box. When the **Treat as grouped when propagating variant conditions** checkbox is clear, the Subsystem is an ungrouped virtual subsystem.

A grouped subsystem represents a system of equation and hence the propagated conditions also apply to the blocks within this system. A grouped subsystem has a continuous boundary line. An ungrouped subsystem does not represent a system of equation and the blocks within it have ungrouped semantics. An ungrouped subsystem has a dotted boundary line and the conditions are propagated into the subsystem.

The variant condition `V = 1` propagates to `Subsystem1` and further to the blocks within it as `Subsystem1` is a grouped virtual subsystem (represents a system of equation). Since Subsystem1 is a system, the condition also applies the blocks within the system.

`Subsystem2` that is an ungrouped virtual subsystem (does not represent a system of equation) also receives `V = 1` as the propagated condition, and the propagated variant condition `V = 1` propagates into `Subsystem 2` as if the subsystem were expanded. The dotted lines on the Subsystem 2 icon indicates that it is flattened during Simulink compilation and hence you can see variant condition for those blocks inside it.

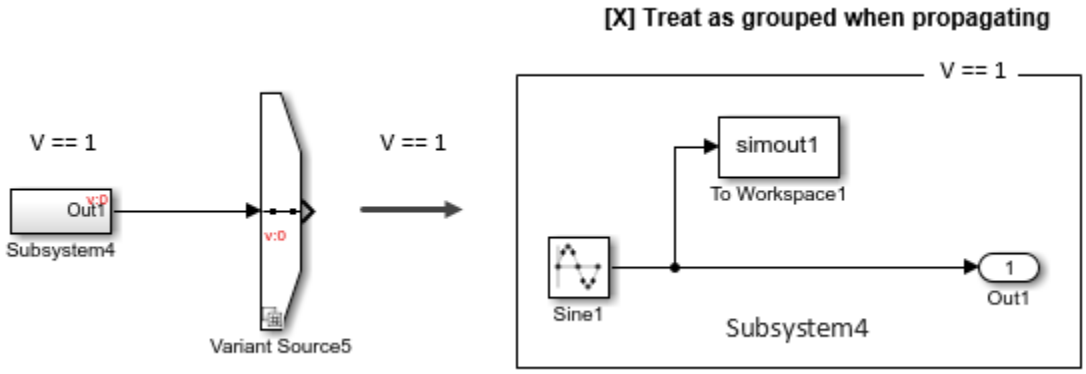
`Variant Source3` is connected to a nonvirtual (atomic) subsystem with `V = 1` as the propagated variant condition. A nonvirtual (atomic) subsystem always represents a system of equations. An

atomic subsystem has a continuous solid boundary line. The variant condition does not propagate inside the nonvirtual subsystem. Instead, it stays on the boundary. However, all blocks inside that subsystem get same variant condition as the Subsystem. The nonvirtual subsystem behaves as an entity.

Virtual subsystems by default works like a grouped collection of blocks where all the blocks contained inside the subsystem have the same variant condition. This is true when the **Treat as grouped when propagating variant conditions** parameter is selected. Virtual subsystems behave like an ungrouped collection of blocks when the **Treat as grouped when propagating variant conditions** parameter is cleared. In this example, the behavior of the Subsystem2 (Ungrouped) may not be as expected as the block never becomes conditional. Nonvirtual subsystems always behave like an entity and the contents only execute when the condition assigned to the subsystem is satisfied. Nonvirtual subsystems, Model blocks, and grouped virtual subsystems behave the same.

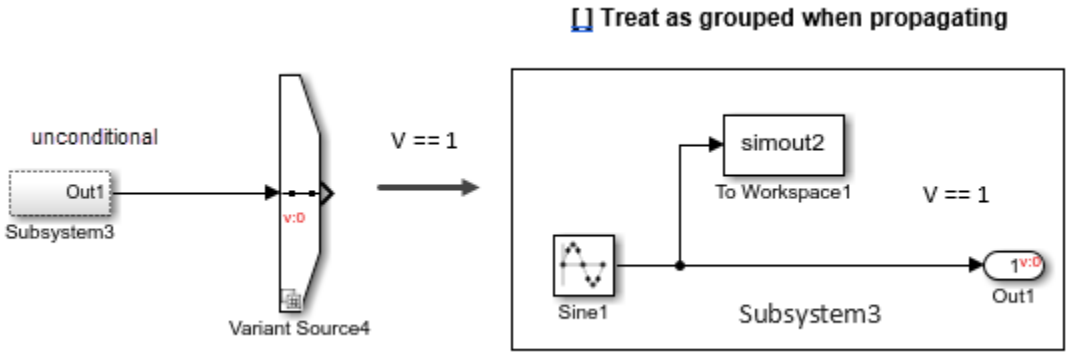
Virtual Subsystems with Treat as Grouped when Propagating Variant Conditions

Treat as grouped when propagating variant conditions option is On: Consider an example with Variant Source block: `Variant Source5` and a virtual subsystem `Subsystem4`. The `Variant Source5` has a condition `V = 1`. `Subsystem4` has a Sine Wave connected to a `To Workspace1` and `Outport` blocks.



Subsystem4 gets condition `V = 1` as `V = 1` is propagated from `Variant Source5`. The blocks inside `Subsystem4` indirectly inherits the condition from `Subsystem4`. `Subsystem4` is conditional as its inports/outports are conditional. To make the `Subsystem4` as unconditional, add a new outport or clear the **Treat as grouped when propagating variant conditions** option.

Treat as grouped when propagating variant conditions option is Off: Consider an example with a Variant Source block: `Variant Source4` and a virtual subsystem `Subsystem3`. The `Variant Source4` has condition `V = 1`. `Subsystem3` has a Sine Wave connected to a `To Workspace` and `Outport` blocks.



`Subsystem3` gets the condition `V = 1` as `V = 1` propagates from `Variant Source4`. However due to unconditional block `To Workspace1`, the propagation stops and the condition `V = 1` is set only to the `Out1` port. Now the `Subsystem3` is also unconditional due to presence of unconditional blocks within.

More About

- “Condition Propagation with Variant Subsystem” on page 12-86

Variant Subsystems

This model illustrates Simulink® variant subsystems. Variant subsystems let you provide multiple implementations for a subsystem where only one implementation is active during simulation. You can programmatically swap out the active implementation and replace it with one of the other implementations without modifying the model.

Overview of Variant Subsystems

A Variant Subsystem block contains two or more child subsystems where one child is active during model execution. The active child subsystem is referred to as the *active variant*. You can programmatically switch the active variant of the Variant Subsystem block by changing values of variables in the global workspace, or by manually overriding variant selection using the Variant Subsystem block dialog. The *active variant* is programmatically wired to the Inport and Outport blocks of the Variant Subsystem by Simulink during model compilation.

To programmatically control variant selection, a `Simulink.Variant` object is associated with each child subsystem in the Variant Subsystem block dialog. `Simulink.Variant` objects are created in the MATLAB® global workspace. These objects have a property named `Condition`, which is an expression, that evaluates to a boolean value and is used to determine the active variant child subsystem.

Note: You can specify variant controls in the MATLAB® global workspace, model workspace, mask workspace, or a data dictionary.

For example, defining `VSS_LINEAR_CONTROLLER=Simulink.Variant('VSS_MODE==1');`

in the global workspace creates a `Simulink.Variant` object where the constructor argument `('VSS_MODE==1')` defines when the variant is active. Using the Variant Subsystem dialog, you then associate `VSS_LINEAR_CONTROLLER` with one of the child subsystems within the Variant Subsystem. Defining

```
VSS_MODE=1
```

in the global workspace, activates the `VSS_LINEAR_CONTROLLER` variant. The condition argument can be a *simple expression* consisting of scalar variables, enumerations, equality, inequality, &&, , and ~. Parenthesis () can be used for precedence grouping.

Using Variant Subsystems

The model in this example uses the following variant objects and variant control variable, which are defined in the MATLAB global workspace:

```
VSS_LINEAR_CONTROLLER=Simulink.Variant('VSS_MODE==1');

VSS_NONLINEAR_CONTROLLER=Simulink.Variant('VSS_MODE==2');

VSS_MODE=2;
```

Opening the example model `sldemo_variant_subsystems` runs the **PreLoadFcn** defined in `File -> ModelProperties -> Callbacks`. This populates the global workspace with the variables for the Variant Subsystem block named Controller:

```
open_system('sldemo_variant_subsystems')
```

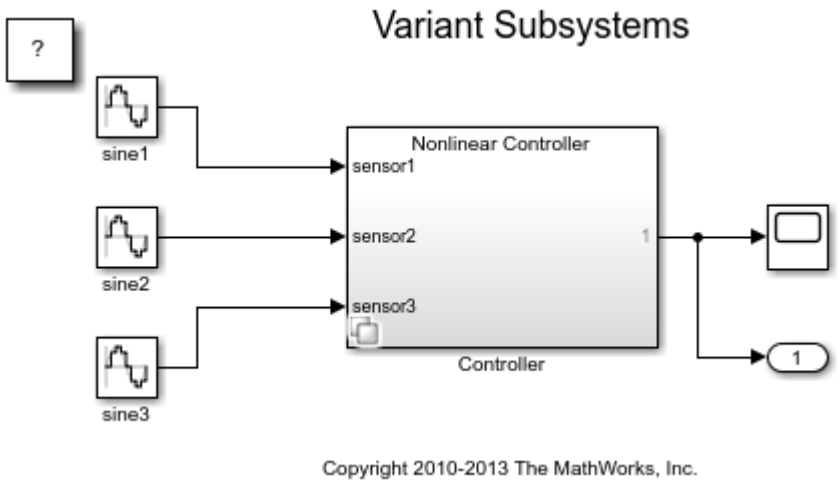


Figure 1: The example model, `sldemo_variant_subsystems`

To specify the `Simulink.Variant` objects association for the Controller subsystem, right-click on the Controller subsystem and select `Subsystem Parameters`, which will open the Controller subsystem block dialog.

The Controller subsystem block dialog specifies two potential variants. The two variants are in turn associated with the two `Simulink.Variant` objects `VSS_LINEAR_CONTROLLER` and `VSS_NONLINEAR_CONTROLLER`, which exist in the global workspace. These objects have a property named **Condition**, an expression that evaluates to a boolean and that determines which variant is active. The condition is also shown in the Variant Subsystem block dialog. In this example, the Condition properties of `VSS_LINEAR_CONTROLLER` and `VSS_NONLINEAR_CONTROLLER` are `VSS_MODE == 1` and `VSS_MODE == 2`, respectively. The variable `VSS_MODE` resides in the global workspace, and can be a standard MATLAB variable or a `Simulink.Parameter`.

If there is no associated variant object or a '%' (comment) character prefixes the variant object in the Variant Subsystem parameters dialog box, then the child subsystem is considered commented out and is not used during model execution.

```
open_system('sldemo_variant_subsystems/Controller');
```

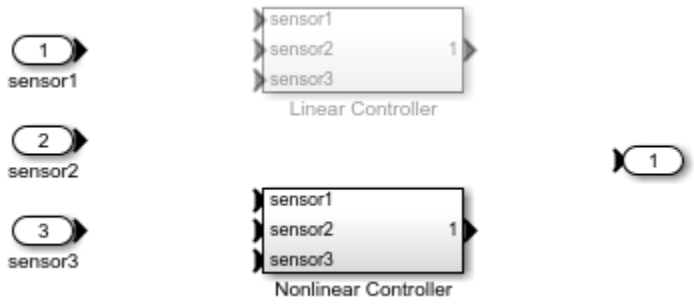


Figure 2: Contents of the Controller subsystem block

Within a Variant Subsystem block, you can place Inport, Outport, and Subsystem blocks. In this example, the `Linear Controller` Subsystem block is associated with the variant object,

VSS_LINEAR_CONTROLLER, and the Nonlinear Controller Subsystem block is associated with the variant object, VSS_NONLINEAR_CONTROLLER.

Signal connections are not allowed in the Variant Subsystem. Simulink programmatically wires up the Inport and Outport blocks to the active variant when simulating the model.

Switching Active Variants

To simulate using the Linear Controller variant, define:

```
VSS_MODE=1
```

in the global workspace and then simulate the model.

```
open_system('sldemo_variant_subsystems/Controller','parameter');
close_system('sldemo_variant_subsystems/Controller')
VSS_MODE=1; %#ok (used by sldemo_variant_subsystems)
sim sldemo_variant_subsystems;
```

Figure 3: Simulation using the Linear Controller variant

To simulate using the Nonlinear Controller, define

```
VSS_MODE=2
```

in the global workspace and then simulate the model.

```
VSS_MODE=2; % (used by sldemo_variant_subsystems)
sim sldemo_variant_subsystems;
```

Figure 4: Simulation using the Nonlinear Controller variant

Enumerations and Reuse

The sldemo_variant_subsystems_enum model illustrates the following Simulink.Variant capabilities:

1. **Enumerations:** MATLAB enumeration classes can be used to improve readability in the conditions of the variant object.

2. **Reuse:** Simulink.Variant objects can be reused in different Variant Subsystem blocks.

This example uses following variables which are defined in the MATLAB global workspace:

```
VSSE_LINEAR_CONTROLLER=Simulink.Variant( ...
'VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.LINEAR')

VSSE_NONLINEAR_CONTROLLER=Simulink.Variant( ...
'VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.NONLINEAR')

VSSE_MODE=sldemo_vss_CONTROLLER_TYPE.LINEAR

VSSE_PROTOTYPE=Simulink.Variant( ...
'VSSE_MODE_BUILD==sldemo_vss_BUILD_TYPE.PROTOTYPE')

VSSE_PRODUCTION=Simulink.Variant( ...
'VSSE_MODE_BUILD==sldemo_vss_BUILD_TYPE.PRODUCTION')
```

`VSSE_MODE_BUILD=sldemo_vss_BUILD_TYPE.PRODUCTION`

In these `Simulink.Variant` objects, we use the enumeration classes, sldemo_vss_BUILD_TYPE.m, and sldemo_vss_CONTROLLER_TYPE.m to define the `Simulink.Variant` **Condition** parameters which improves readability.

The three filter Variant Subsystems blocks, Filter1, Filter2, and Filter3 all use the `VSSE_PROTOTYPE` and `VSSE_PRODUCTION` `Simulink.Variant` objects.

Note: The name of the enumeration class must be unique among data type names and global workspace variable names, and is case-sensitive.

Opening the example model `sldemo_variant_subsystems_enum` runs the **PreLoadFcn** defined in `File -> ModelProperties -> Callbacks`. This populates the global workspace with variables for the Variant Subsystem blocks:

```
open_system('sldemo_variant_subsystems_enum')
```

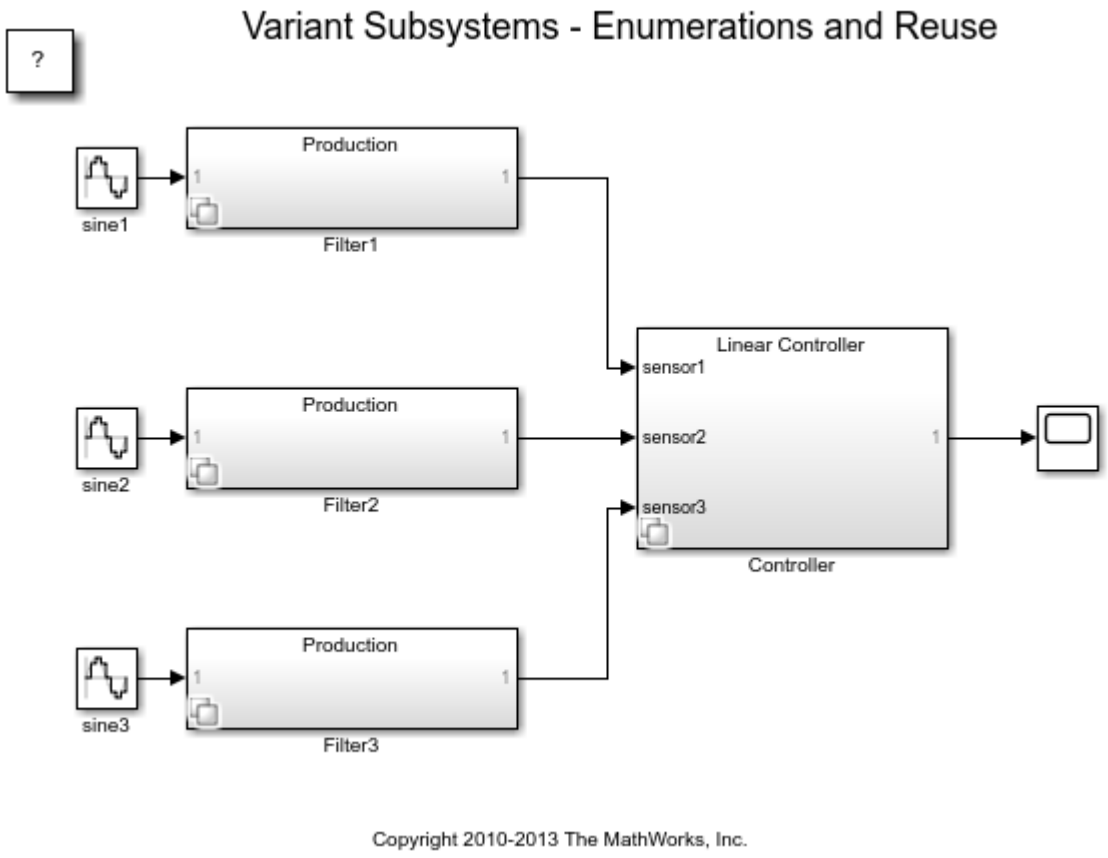


Figure 5: The example model, `sldemo_variant_subsystems_enum`

Code Generation Using Enumerated types as Variant Control Variables

You can use enumerated types to give meaningful names to integers used as variant control values.

Consider the model `rtwdemo_preprocessor_subsys`.

In the MATLAB Editor, define the classes that map enumerated values to meaningful names.

```
% ControllerModes.m
classdef ControllerModes < Simulink.IntEnumType
    enumeration
        Linear(1)
        NonLinear(2)
    end
end
```

If the enumerated types are not defined correctly, an error will be displayed. Here are a couple of scenarios which results in error.

Invalid definition1: In this case, the `Simulink.IntEnumType` is not defined.

```
classdef VModesU
    enumeration
        Linear (1)
        NonLinear  (2)
    end
end
```

Invalid definition2: In this case, the variables are not initialized.

```
classdef VModesU < Simulink.IntEnumType
    enumeration
        Linear
        NonLinear
    end
end
```

Enter the variant control expression as shown in the next example:

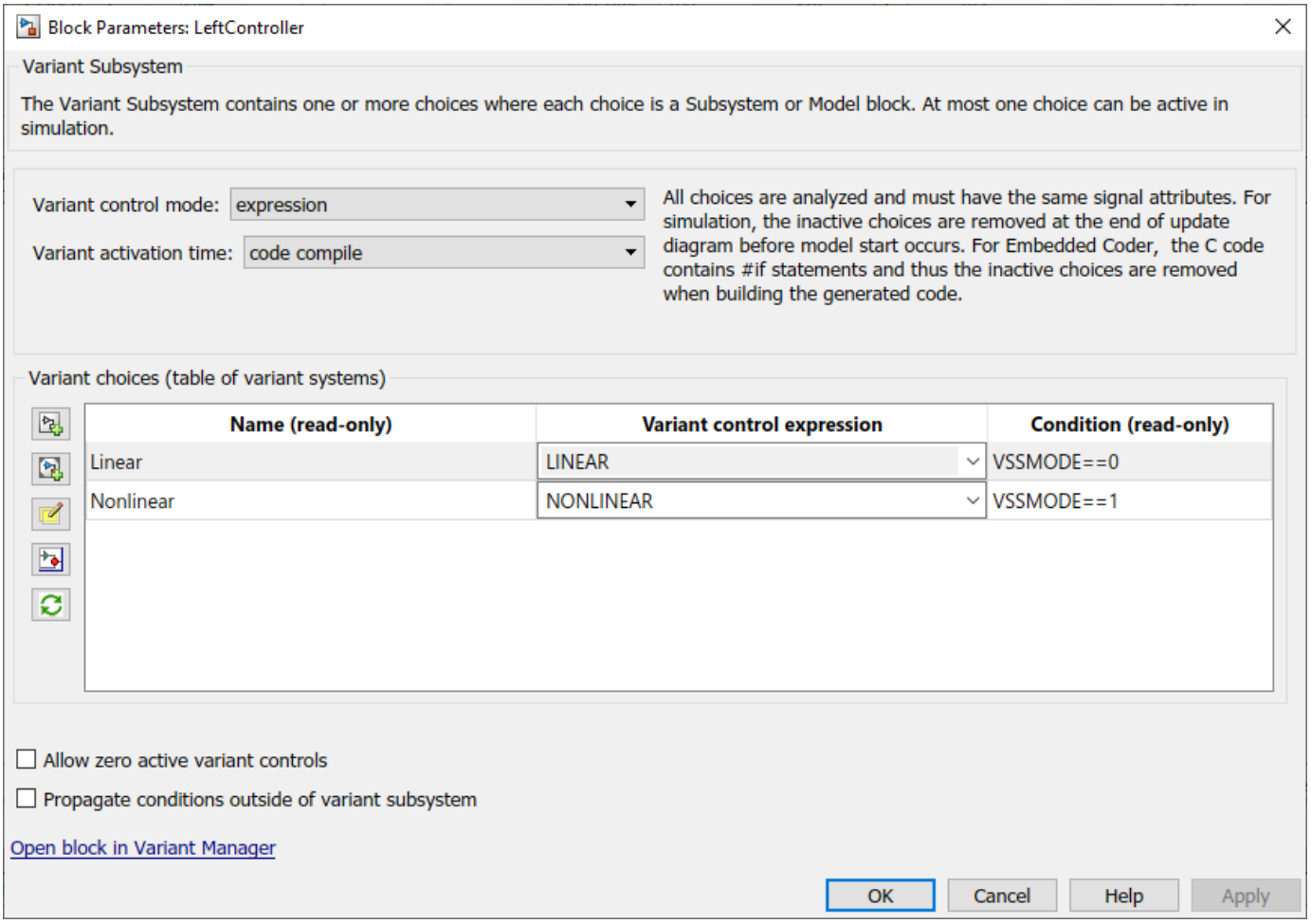


Figure 6: Block Parameters

Define the value of V in the global workspace. For example, V=2;. The value can be a normal MATLAB variable or a `Simulink.Parameter` object. However, the value cannot be an enumerated type.

Now generate code with **Variant activation time** set to `code compile`. Sample code is as shown below.

```
/* Simulink enumerals used in Code Variant condition expressions */
#ifndef _RTW_ENUMS_ControllerModes_
#define _RTW_ENUMS_ControllerModes_
#define ControllerModes_Linear         1
#define ControllerModes_NonLinear      2
#endif
```

Figure 7: Generated Code

For information on using `Simulink.Variant` or `Simulink.Parameter` object or MATLAB variables as a variant control variable, see the **Approaches for Specifying Variant Controls** section in “Introduction to Variant Controls” on page 12-24.

See the Embedded Coder documentation for more information on code generation for variant subsystems.

More About

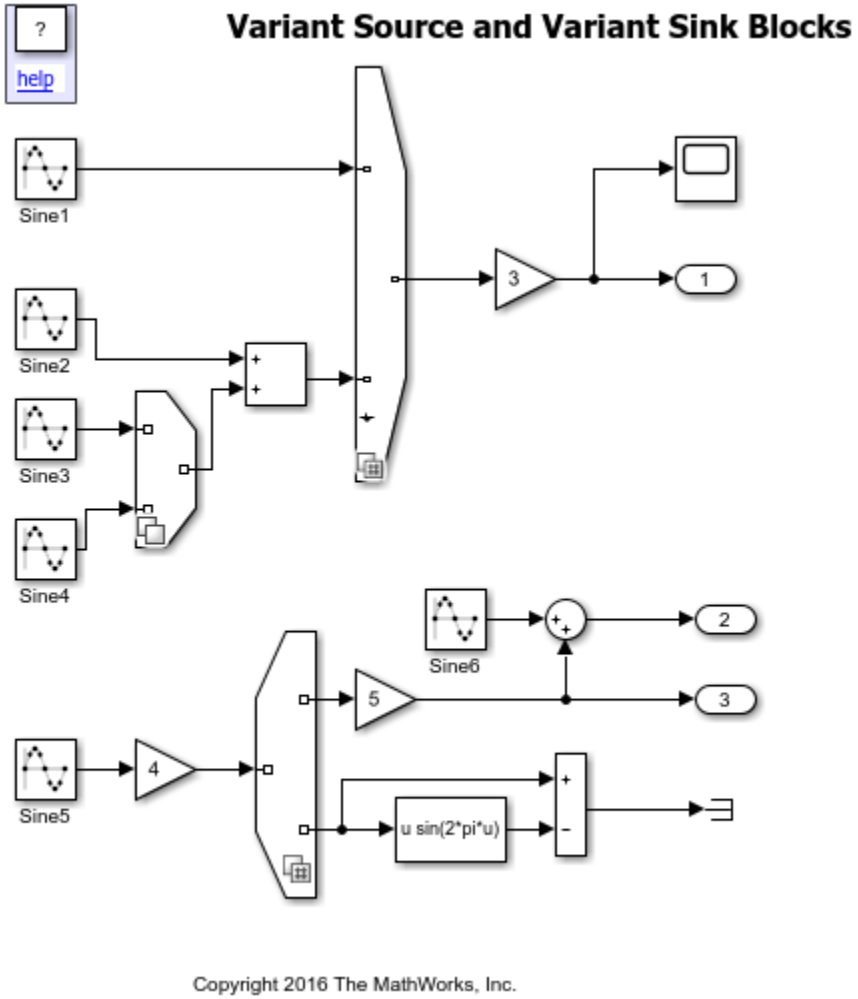
Variant System Design

Variant Source and Variant Sink Blocks

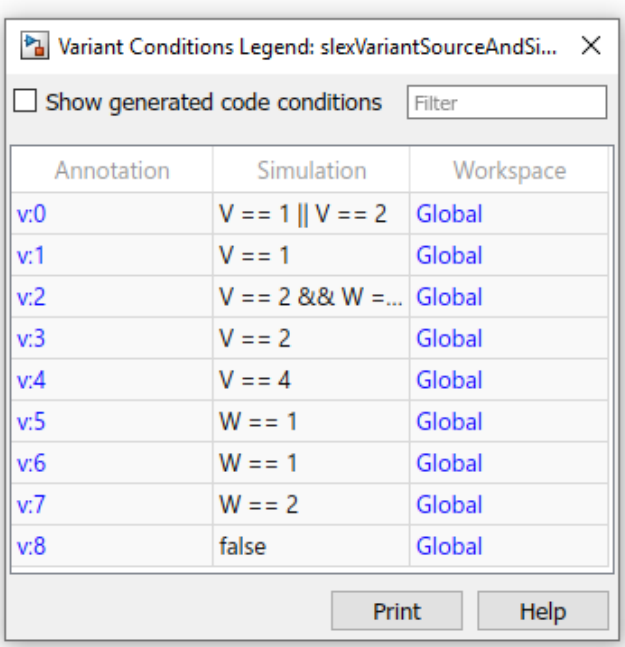
Define variant choice regions in the Variant Source and Sink blocks based on the block connectivity. The variant choice regions are computed by Simulink when you update diagram (**Simulation > Prepare > Update Model**).

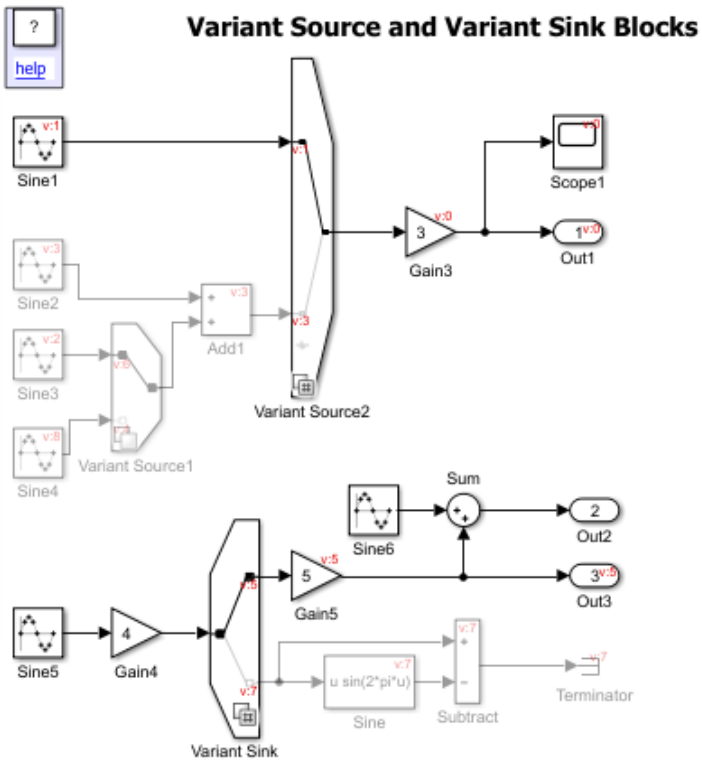
The process of computing the variant choice regions is called `variant condition propagation`. The Variant Source block provides variation on the source of a signal, and the Variant Sink blocks provides variation on the destination (sink) of a signal.

Consider a model containing two Variant Source blocks (`Variant Source1` , `Variant Source2`) and a Sink block (`Variant Sink`).



The variant conditions at the inports and outports of Variant Source and Sink blocks, respectively, determine the activation and deactivation of the blocks connected to them. To view the annotations and the variant conditions, in the **Debug** tab of the toolstrip, click **Information Overlays > Variant Conditions**.





Let us analyze the variant conditions and the block activation state.

- In `Variant Source1`, when `W==1`, the `Sine3` block is active, and when `V==4`, the `Sine4` block is active.
- In `Variant Source2`, when `V==1`, the Sine1 block is active, and when `V==2`, the `Add1` block is active.
- At `Add1` block the condition propagation continues making `Variant Source1` block to be active only when the `V==2`. This further propagates to `Sine3` block and `Sine4` block, making the `Sine3` block active at `V==2 && W==1` and the `Sine4` block active at `V==2 && W==2`, respectively.
- The `Gain3` block is active when either `V==1` or `V==2`, and hence the condition `V==2 || V==1`. The variant condition is further propagated to `Scope1` and `Out1`.
- The blocks connected to the outport of `Variant Sink` are active when `W==1` (`Gain5`), or `W==2` (`Sine`, `Subtract`, `Terminator`).
- The `Sum` block illustrates two key concepts in variant condition propagation: Signals are only variant if explicitly marked or when all paths can be proven to be variant. To make the `Sine6`, `Sum`, `Out2` variant, place a Single-Input Single-Output Variant Source before Out2 (or after the `Sine6`). Reading an inactive signal is equivalent to reading ground. When `W ~= 1`, then the bottom input to the `Sum` block is inactive and `Out2 = Sine6 + ground`.

If you set the **Variant activation time** parameter to `code compile` for the Variant Source and Variant Sink block, the generated code contains the code for the active and the inactive (`#if COND`). If this parameter is not selected, then code is generated only for the active choices.

If you select the **Allow zero active variant controls** parameter for the Variant Source and Variant Sink block, you can simulate the variant model without an active variant. In such cases, Simulink

disables the blocks connected to the input and output stream of Variant Source and Variant Sink. These disabled blocks are ignored from update diagram or simulation.

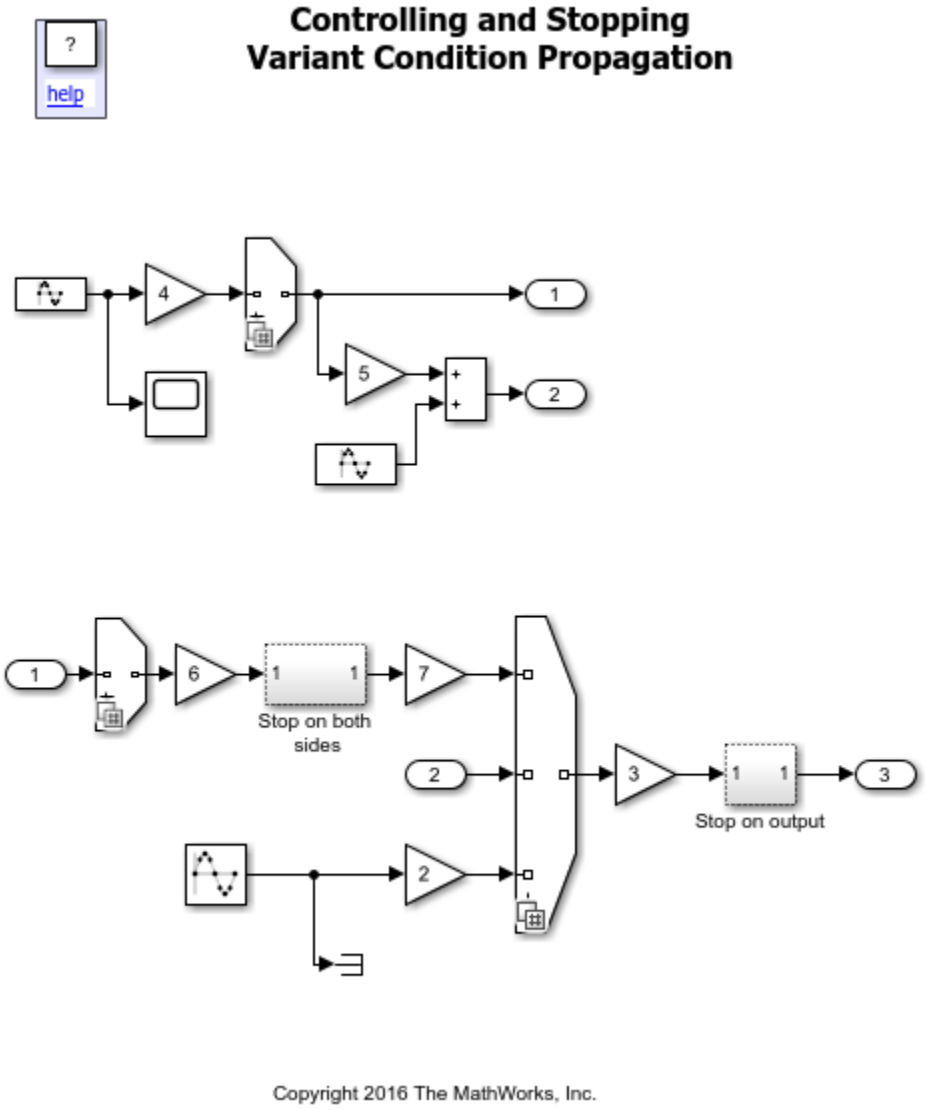
More About

- “Define and Configure Variant Sources and Sinks” on page 12-55
- Variant System Design

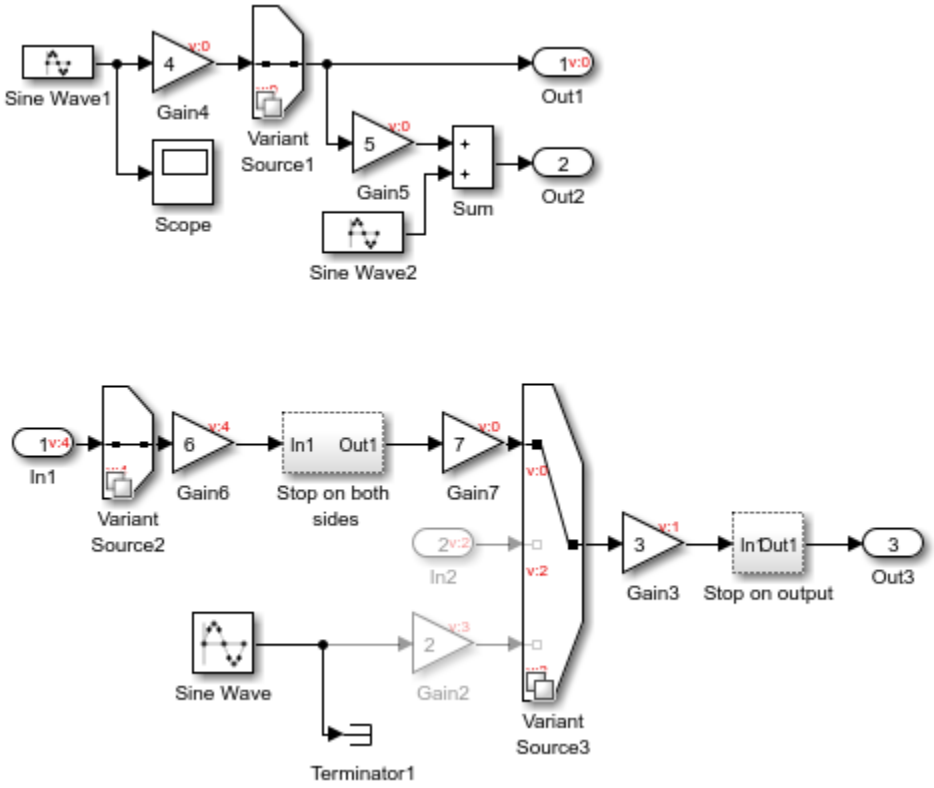
Control Variant Condition Propagation

During variant condition propagation, Simulink automatically assigns conditions to blocks. You can control how the variant condition propagates upstream and downstream in a model.

Consider this model.



In Simulink, click **Simulation > Run** to view the variant condition propagation to blocks.



The `Variant Source1` block has the `A==1` condition, which propagates backward and forward to the blocks connected to `Variant Source1` block. The variant condition propagates to `Gain4` block but does not propagate to the `Sine Wave1` block.

The `Scope` block is unconditional and receives its inputs from the `Sine Wave1` block. Therefore, the `Sine Wave1` block is unconditional. If you remove the `Scope` block, the variant condition propagates to the `Sine Wave1` block.

If you replace the `Scope` block with any other block (including the `Terminator` block), the `Sine Wave1` block remains unconditional.

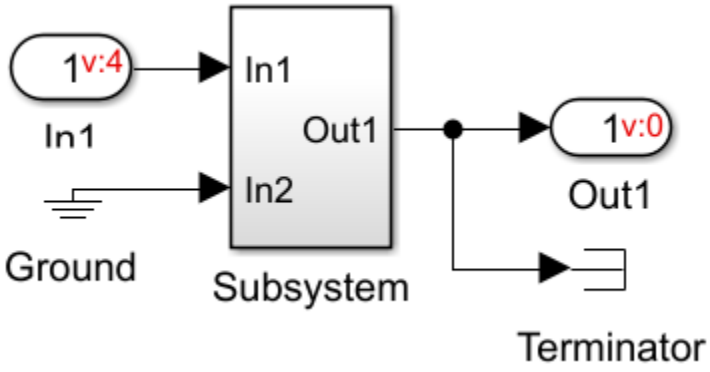
A block is unconditional if at least one of its inputs is unconditional. The input side of the `Sum` block is connected to `Gain5` (conditional) block and to the `Sine Wave2` (unconditional) block. Therefore, the `Sum` block is unconditional.

You can use these concepts to create a Subsystem block that controls the propagation of variant conditions to both sides or to one side.

Stop Propagation of Variant Condition Upstream and Downstream

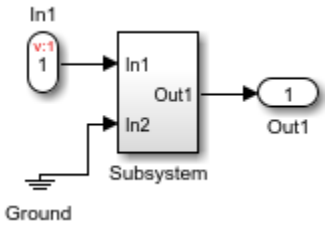
Consider the section of the model that is connected to the `Variant Source2` and `Variant Source3` blocks. When you simulate the model, the Variant condition from the `Variant Source2` block and the `Variant Source3` blocks propagates upstream and downstream.

The `Stop on both sides` block between `Gain6` and the `Gain7` block prevents the Variant condition from propagating upstream or downstream. Double-click the `Stop on both sides` block to view its components.



The `Stop on both sides` block uses a `Terminator` to stop the variant condition propagation on upstream of the `Subsystem` block. To stop the condition propagation on the downstream side of the `Subsystem` block, one of the inports is connected to `Ground` (unconditional). Therefore, this arrangement stops the variant condition propagation upstream and downstream. Similarly, you can selectively stop the condition propagation of variant condition upstream or downstream for a model. For example, if you remove the `Terminator` block, variant condition propagates upstream but is stopped downstream.

Stop Propagation of Variant Condition Downstream



Here, one input port of the Subsystem block is unconditional making the Subsystem block unconditional at input side and thus stopping the propagation of variant condition downstream.

More About

- “Define and Configure Variant Sources and Sinks” on page 12-55
- Variant System Design

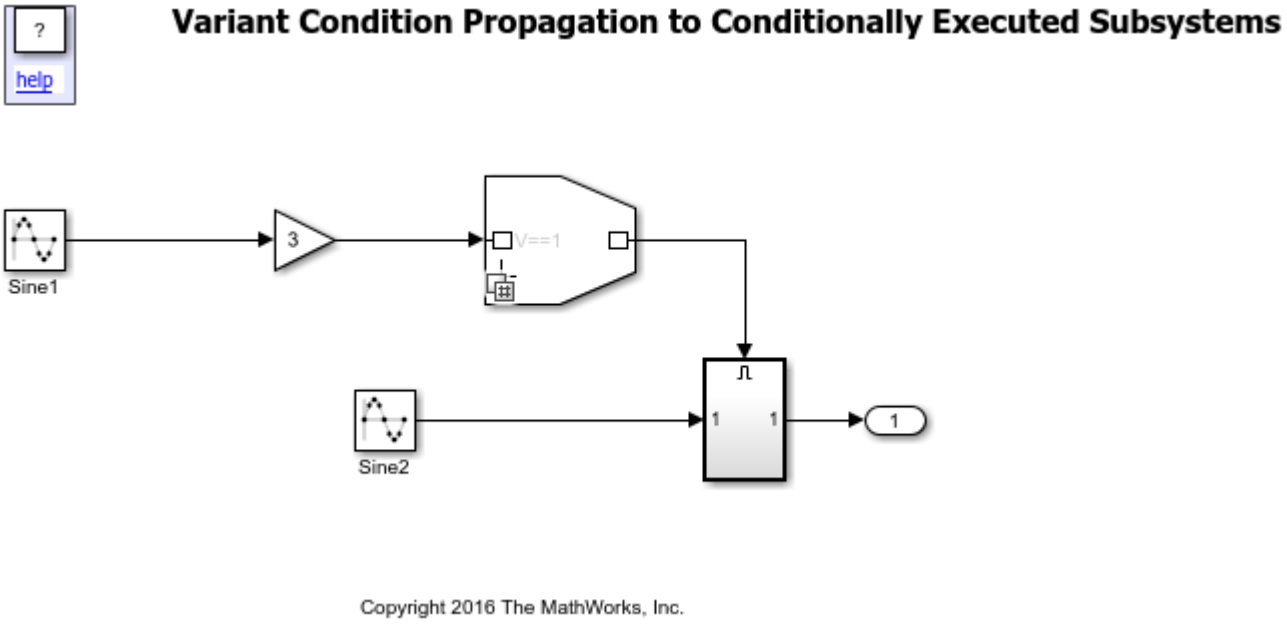
Propagate Variant Condition to Conditional Subsystem

A conditional subsystem (also known as a conditionally executed subsystem) is a type of subsystem where you can control the execution using an external signal.

Enabled, Triggered, and Function-Call Subsystems are examples of conditional subsystems. The signal that controls a conditional subsystem is called the *control signal* and the port from which the signal enters the block is called the *control port*. For more information on conditional subsystems, see “Conditionally Executed Subsystems Overview” on page 10-3.

You can use a variant block to control the execution of a conditional subsystem blocks.

Consider this model.



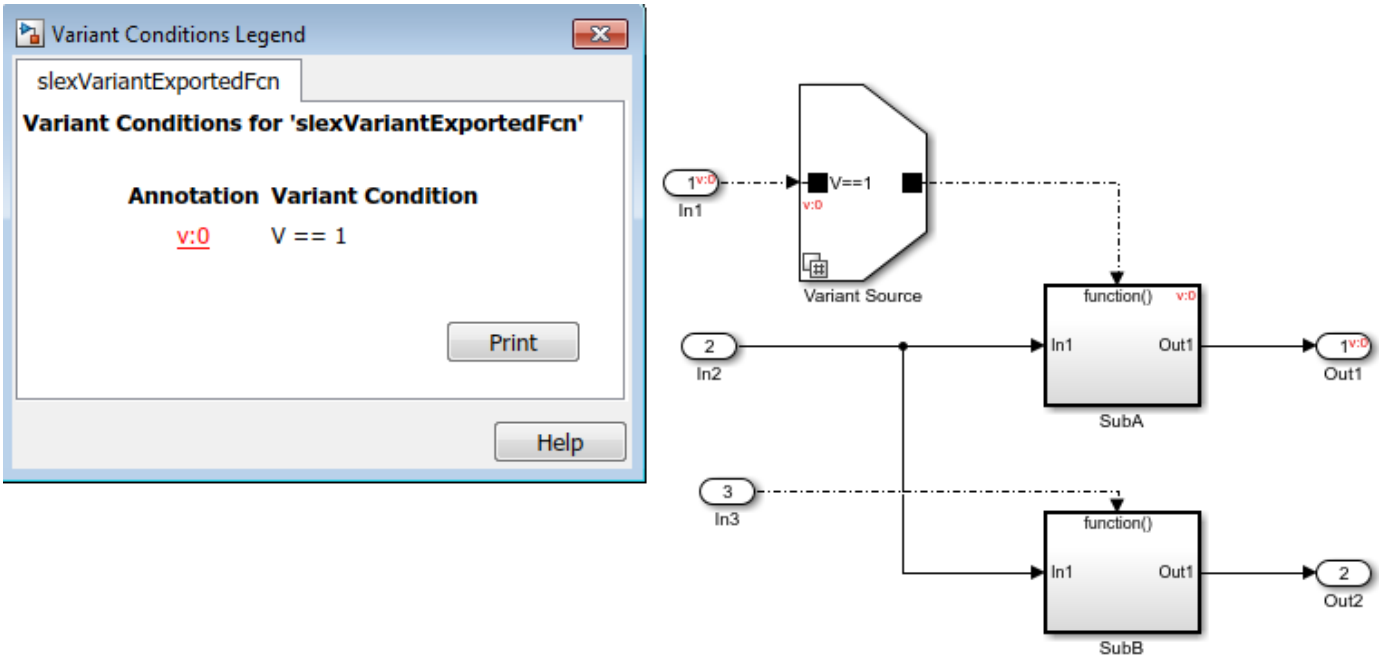
`Variant Source1` is a single-input/single-output Variant Source block with variant condition as `V==1`. When you simulate this model, the variant condition from the `Variant Source1` block propagates to the control port of the `Subsystem` block and then to the blocks connected to its inports and outports.

For example, when `V=1`, `Variant Source1` is active and the Variant condition propagates to the control port of the `Subsystem` block. Therefore, the `Subsystem` block is also active and the variant condition propagates to the blocks connected to the input and output ports of the `Subsystem` block.

Propagate Variant Condition to Function-Call Subsystem

A `Function-Call Subsystem` block is a subsystem that another block can invoke directly during simulation. The `Function-Call Subsystem` block is analogous to a function in procedural programming language. For more information, see “Using Function-Call Subsystems” on page 10-34.

You can use a single-input/single-output variant block to make the `Function-Call Subsystem` block conditional.



The `Variant Source` block has condition `V==1`, where `V` is a `Simulink.Parameter`.

When you simulate this model, the variant condition from the Variant Source block propagates to the control port of the `SubA` subsystem block and further propagates to the blocks connected to its inports and outports.

For example, when `V=1`, the `SubA` block is active and the variant condition propagates backward and forward to the blocks connected to the input (`In1`) and output (`Out1`) ports.

When `V~=1` (for example, `V=0`), `SubA` becomes inactive, making `Out1` to be inactive. `In2` remains active as it is connected to `SubB`, which is active.

If `In2` is not connected to `SubB`, `In2` becomes inactive when `V~=1`.

Note: If the `Function-Call Subsystem` is placed inside a virtual grouped subsystem, the variant condition triggering the `Function-Call Subsystem` must match the corresponding condition on the input of the higher level subsystem block.

More About

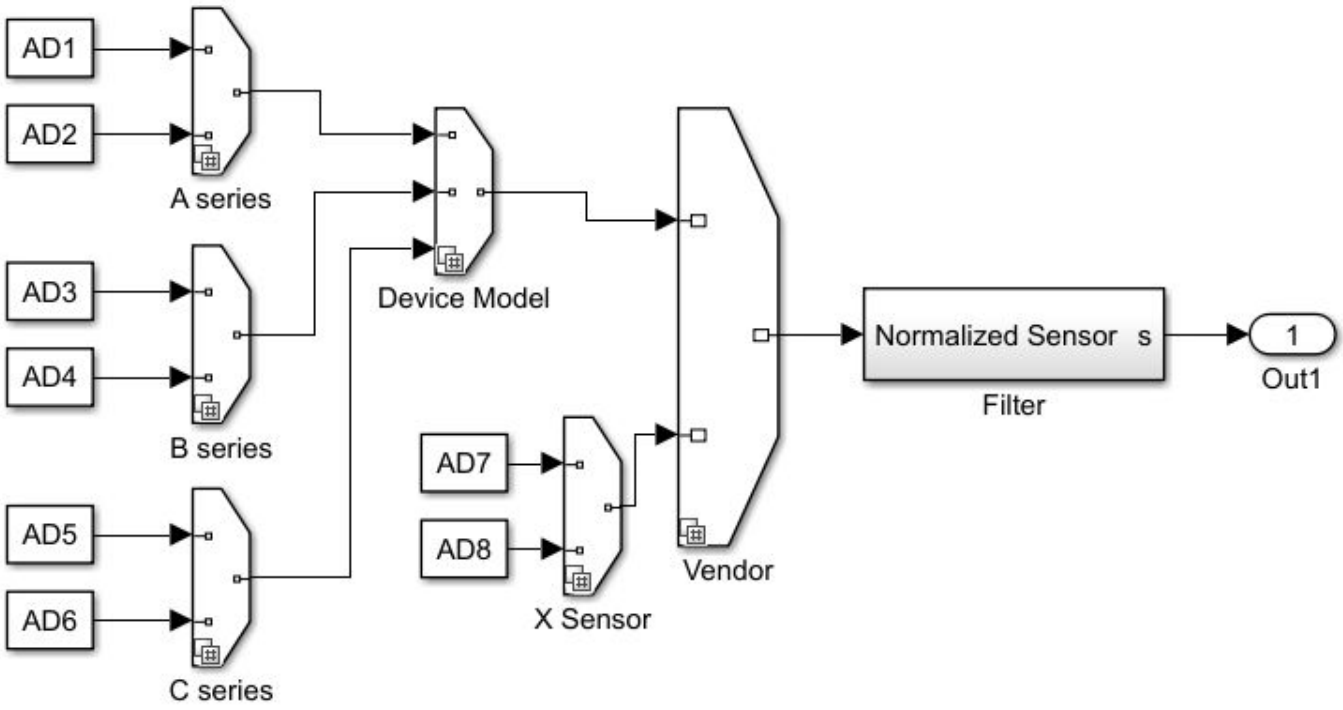
- “Define and Configure Variant Sources and Sinks” on page 12-55
- Variant System Design

Hierarchical Nesting of Variant Sources and Variant Sinks

This example shows how to use Variant Source blocks to provide Variant selection on sensors.

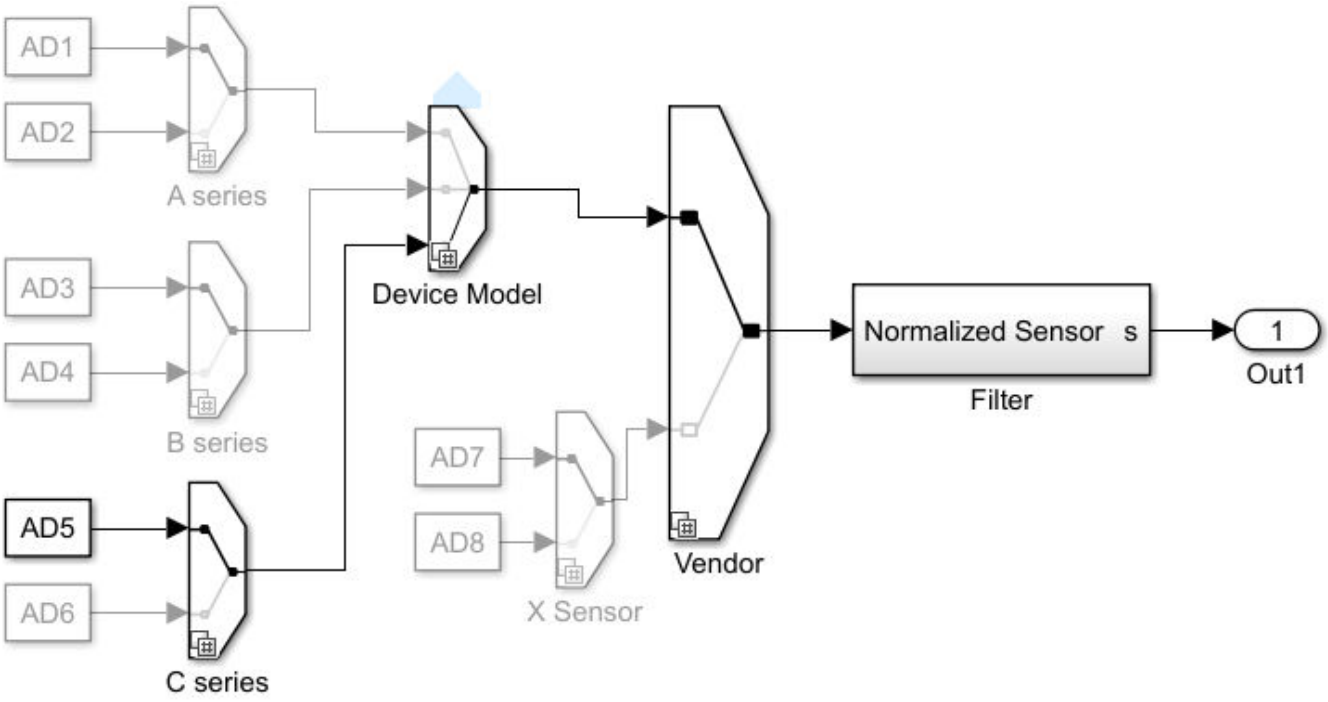
This model illustrates how you can use multiple Variant Source blocks to provide variant selection on sensors. In this model, multiple Variant Source blocks are used to create hierarchical nesting of variant choices. Choices are first grouped by series: A Series, B Series, and C Series. A combination of one or more series is provided as input for a device model. The resulting device model is provided as input to the vendor by including or excluding a sensor selection. In this model, a constant block is masked as a place holder for analog-to-digital converter (A/D) blocks. This model shows variation of sensor inputs. Alternatively, you may use Variant Sink blocks to create variation of actuator outputs. The variant control variables that parameterize the Variant Source blocks are defined in the `PostLoadFcn` callback.

Consider this model with Variant Source blocks.



In this model, the Variant Source block "Vendor" gets inputs from Variant Source blocks "X Sensor" and "Device Model" blocks. The "X Sensor" block gets inputs from constants AD7 and AD8. The "Device Model" block gets inputs from Variant Source blocks "A Series", "B Series", and "C Series". The Variant Source blocks "A Series", "B Series", and "C Series" get inputs from Constant blocks.

Now simulate the model.



When you simulate the model, the constant block AD5 is active. The Variant Source block "Vendor" selects between two vendors, VENDOR==1 or VENDOR==2 and in the base workspace, VENDOR is a Simulink.Parameter with VENDOR.Value=1. The Variant Source block "Device Model" selects between DEVICE_MODEL==1, DEVICE_MODEL==2, or DEVICE_MODEL==3 and in the base workspace, DEVICE_MODEL.Value=3. The Variant Source block "C Series" selects between C_SERIES==1 and C_SERIES==2 and in the base workspace, C_SERIES.Value=2.

Code Generation

You can use Simulink Coder to generate code from a model containing Variant Subsystem blocks. By default, the generated code contains only the active variant. Alternatively, you can generate code for all variants guarded by C preprocessor conditionals (#if, #elif, #endif) when using the Embedded Coder.

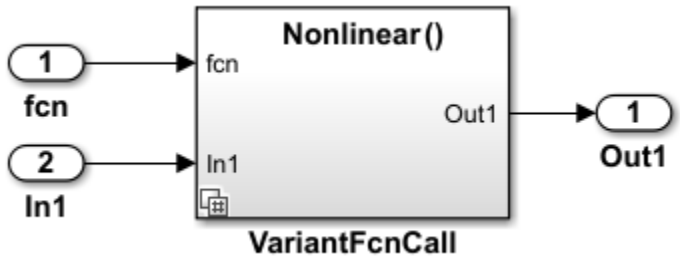
More about

Export-Function model with Variant Subsystem

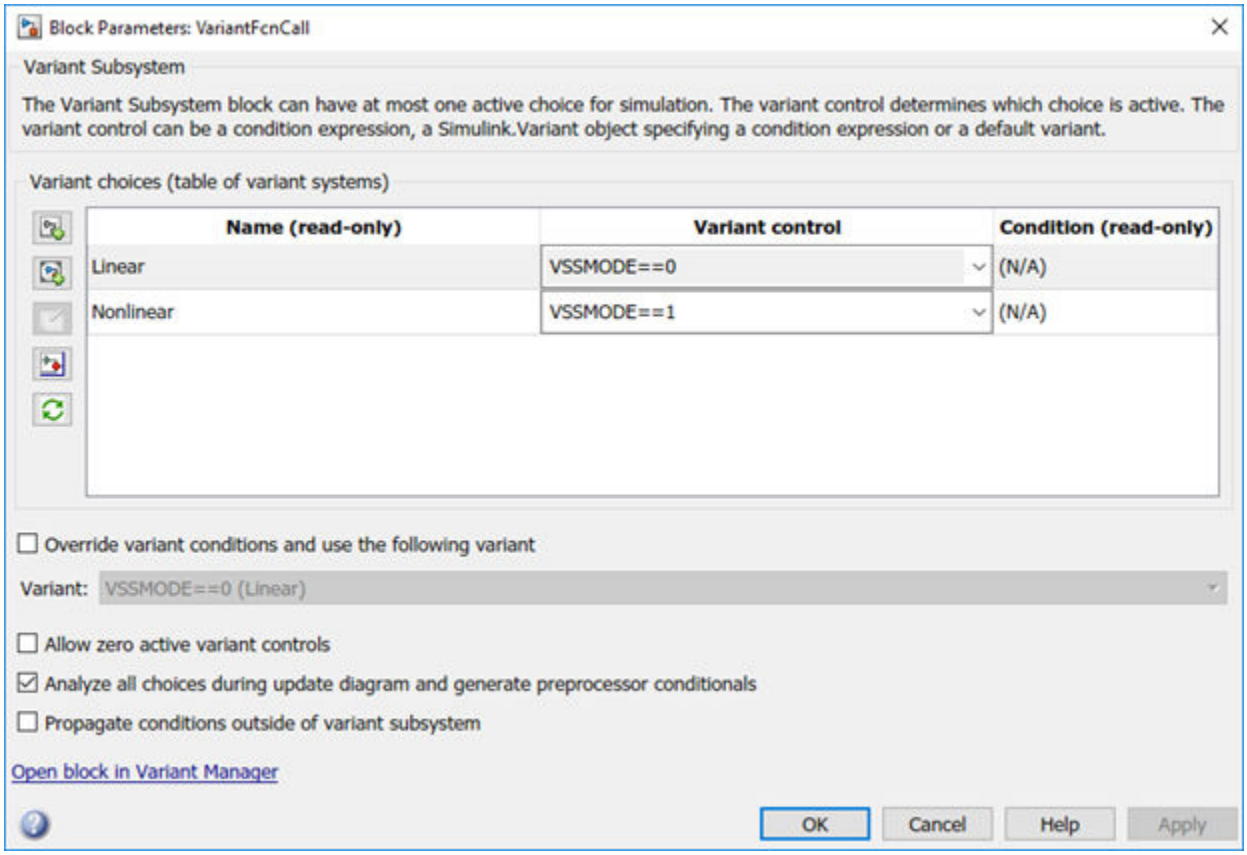
This example shows how to use Variant Subsystem model with Function-Call blocks as choice.

Consider this model containing a Variant Subsystem block.

Note: If you use Function-Call system as a variant choice for the Linear Subsystem block, then the other Subsystem block (Nonlinear) within the Variant Subsystem must also be Function-Call system. Additionally, the control ports in Linear and Nonlinear blocks and the corresponding inport block must have the same name (fcn).



The Variant Subsystem "VariantFcnCall" gets the inputs from two inport blocks. The "VariantFcnCall" block has two variant choices, Linear and Nonlinear governed by conditions VSSMODE==0 and VSSMODE==1. Root inport(fcn) is a function-call inport connecting variant subsystem making this model as export function model.



You can simulate the model slexVariantSubsystemEnableChoice and verify the behavior of the counters by examining the results.

Code Generation

You can use the Simulink Coder to generate code from a model containing Variant Subsystem blocks. By default the generated code contains only the active variant. Alternatively, you can generate code for all variants guarded by C preprocessor conditionals (#if, #elif, #endif) when using the Embedded Coder. In the generated code, the code inside fcn definition is guarded by C preprocessor conditionals inside export function.

```
% fcn() {
% #if VSSMODE==0
% // code for Linear choice
% .......
% #endif
% #if VSSMODE==1
% // code for Nonlinear choice
% .......
% #endif
```

More about

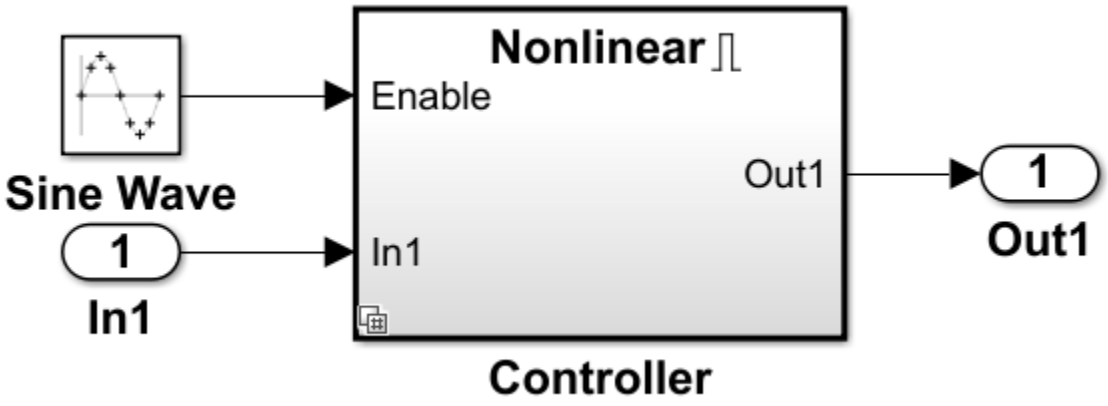
- “Variant Systems with Conditional Systems” on page 12-96

Variant Subsystem with Enable Subsystem as Choice

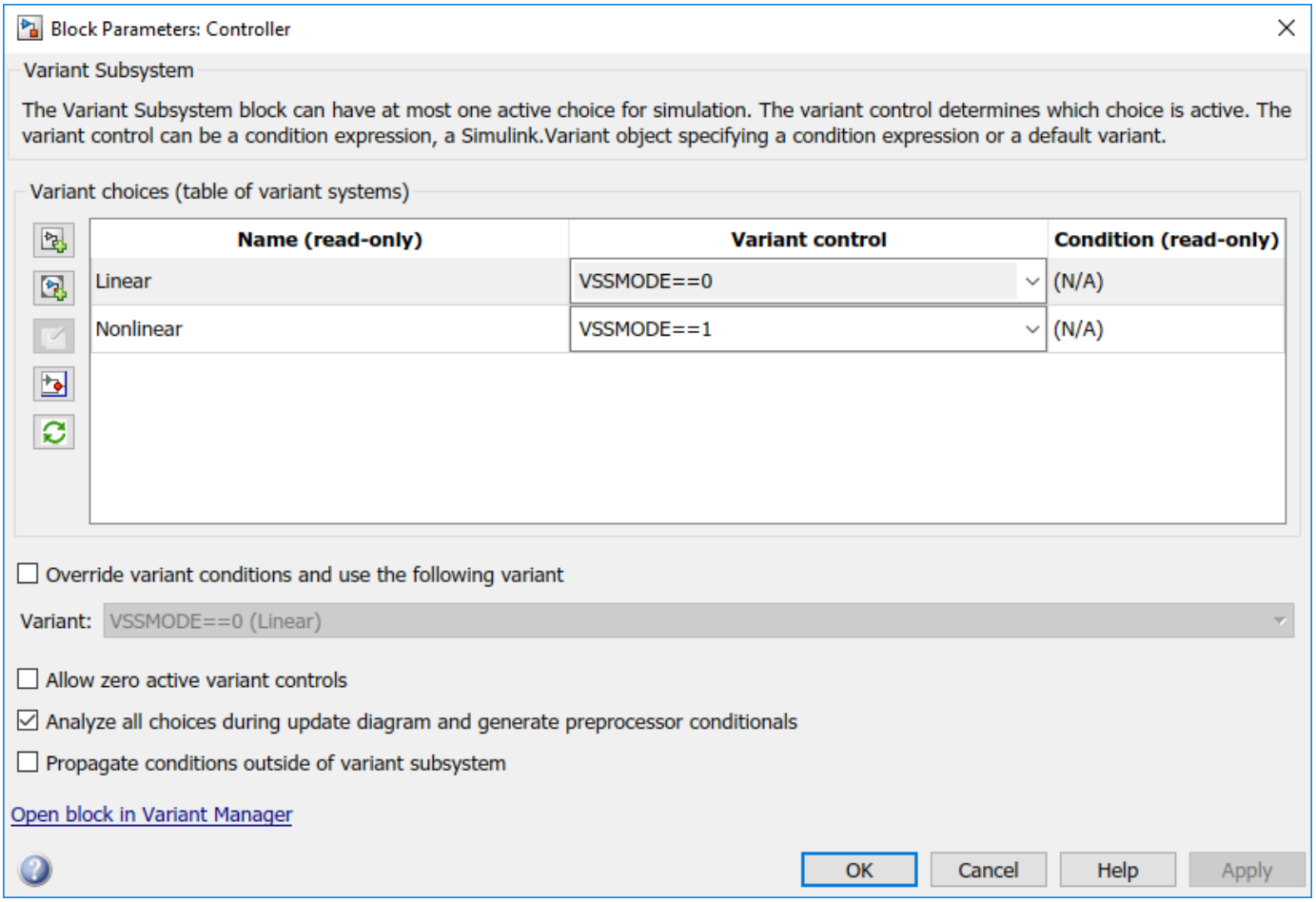
This example shows how to use Variant Subsystem model with Enable Subsystem as choice.

Consider this model with a Variant Subsystem block.

Note: If you use Enable Subsystem as a variant choice for the Linear Subsystem block then the other Subsystem block (Nonlinear) within the Variant Subsystem must also be Enable Subsystem.



The Variant Subsystem block "Controller" gets the input from Sine Wave and Inport blocks. The "Controller" block has two Variant choices, Linear and Nonlinear. The Variant choices are governed by conditions VSSMODE==0 and VSSMODE==1. This model can simulate and generate code based on VSSMODE value.



You can simulate the model slexVariantSubsystemEnableChoice and verify the behavior of the counters by examining the results.

Code Generation

You can use the Simulink Coder to generate code from a model containing Variant Subsystem blocks. By default the generated code contains only the active variant. Alternatively, you can generate code for all variants guarded by C preprocessor conditionals (#if, #elif, #endif) when using the Embedded Coder. In the generated code, the code inside fcn definition is guarded by C preprocessor conditionals inside export function.

```
%step() {
% #if VSSMODE==0
% // code for Linear choice
% .......
% #elif VSSMODE==1
% // code for Nonlinear choice
% .......
% #endif
```

More about

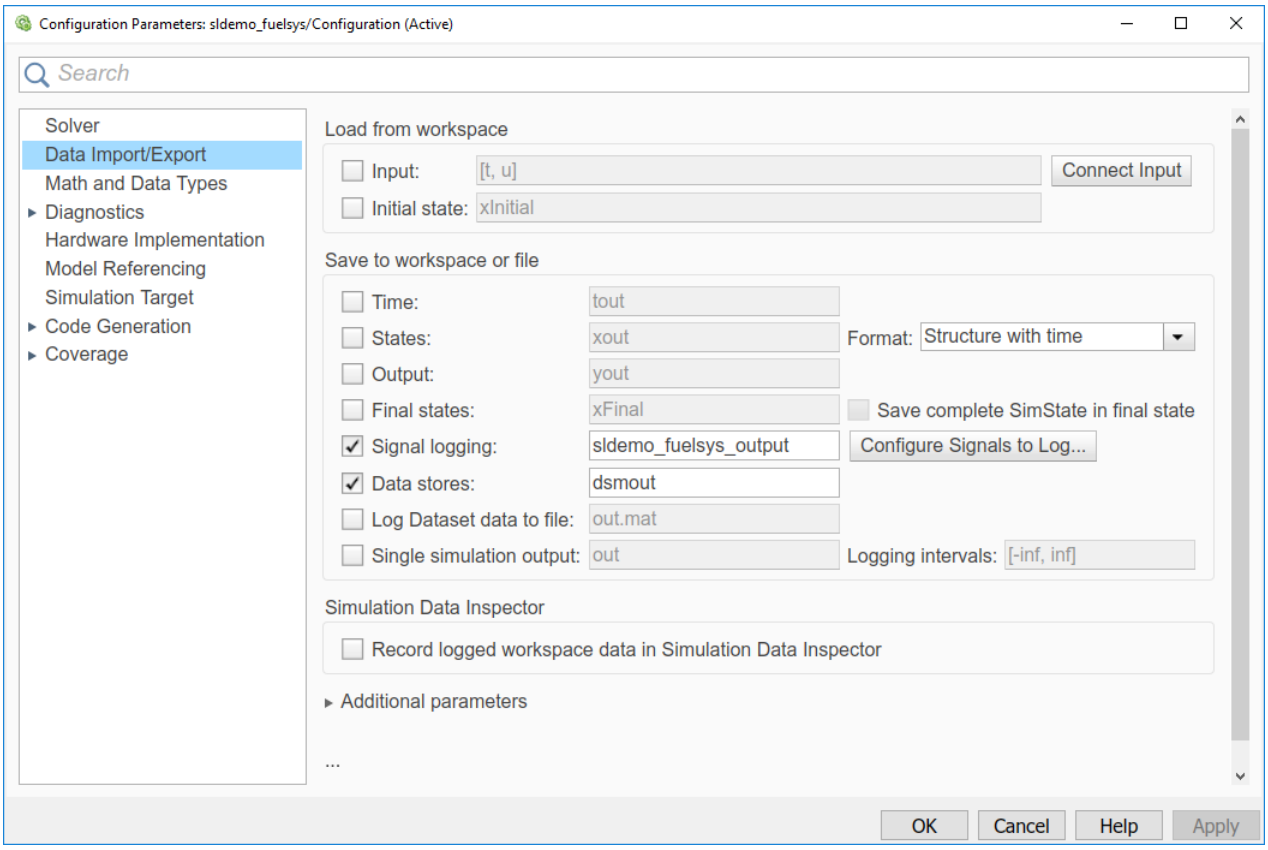
- “Variant Systems with Conditional Systems” on page 12-96

Managing Model Configurations

Set Model Configuration Parameters for a Model

Model configuration parameters determine how your model runs by specifying the type of solver used, import and export settings, and other settings that control model behavior. Every model has a *configuration set* that contains the model configuration parameters and their specified values. When you create a new model, it contains the default configuration set, called `Configuration`, that specifies the default values for the model configuration parameters.

To view and set the configuration parameters for your model, open the Configuration Parameters dialog box. In the Simulink Editor, on the **Modeling** tab, click **Model Settings**  .



Right-click a parameter name and select **What's This?** to see:

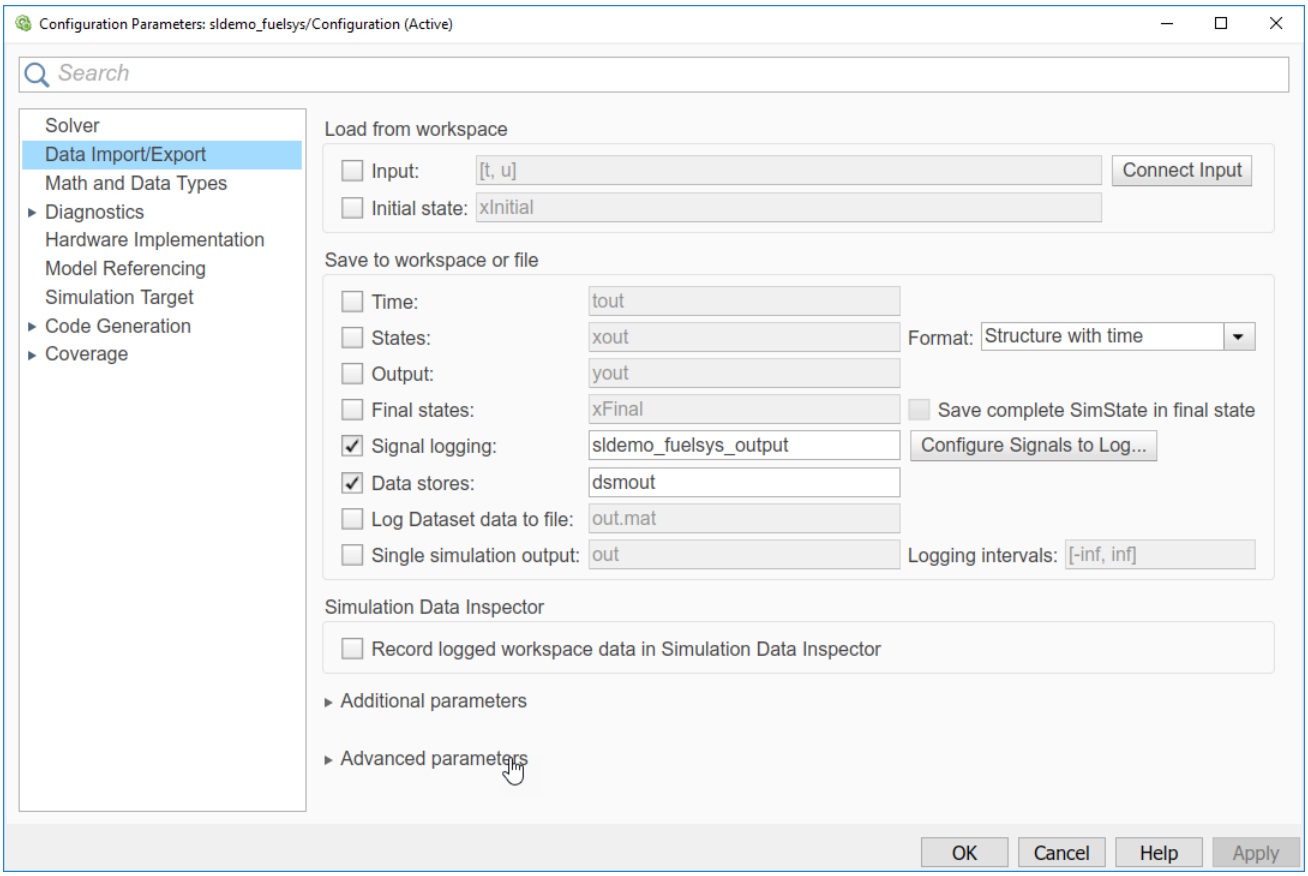
- A short parameter description.
- The parameter name that you can use in scripts.
- Parameter dependencies.

From the **What's This?** dialog box, click **Show more information** for the complete parameter documentation.

To find a parameter by using its name, command-line name, value, or description, use the **Search** box at the top of the dialog box. The search tool supports regular expressions. Type `.` in the search box to see a list of all parameters.

Configuration Panes

The configuration set is organized in panes that contain parameters related to that category. To display the parameters for a specific category, click the associated pane in the tree on the left side of the dialog box. You can access the advanced parameters for each pane by mousing over the ellipsis toward the bottom of the dialog box and clicking **Advanced parameters**.



For more information about the parameters on each of the panes, see:

- “Solver Pane”
- “Model Configuration Parameters: Data Import/Export”
- “Math and Data Types Pane”
- “Model Configuration Parameters: Diagnostics”
- “Hardware Implementation Pane”
- “Model Configuration Parameters: Model Referencing”
- “Model Configuration Parameters: Simulation Target”

Some MathWorks products that work with Simulink define additional parameters. For example, parameters related to Simulink Coder are located on the “Model Configuration Parameters: Code Generation” (Simulink Coder) pane. If such a product is installed on your system, the configuration set also contains the associated configuration parameters and panes.

See Also

More About

- “Manage Configuration Sets for a Model” on page 13-5
- “Automate Model Configuration by Using a Script” on page 13-22

Manage Configuration Sets for a Model

In this section...

A model configuration set is a named collection of values for the parameters of a model.

You can associate multiple sets of parameter values with your model. The configuration sets associated with a model can specify different values for any or all configuration parameters. The model uses the parameter values of the *active* configuration. You can quickly change the active configuration to any of the configuration sets that are attached to the model.

Use multiple configuration sets in a model when you want to:

- Compare the difference in model execution after changing the values of several parameters.
- Use different configurations for your model when you use the model in different contexts.

For this example, you set up the model `sldemo_fuelsys_dd` to have two configuration sets that specify different solvers. You then copy one of the configurations to the model `vdp` and compare it with the default configuration set of `vdp`.

Create a Configuration Set in a Model

The model `sldemo_fuelsys_dd` contains one configuration set, which uses a variable-step solver. Add another configuration to use a fixed-step solver.

1. Open the model. At the command line, type `sldemo_fuelsys_dd`.
2. Open the Model Explorer. On the **Modeling** tab, click **Design** > **Model Explorer**.
3. In the **Model Hierarchy** pane, expand the model node and select the model name `sldemo_fuelsys_dd`.
4. You can create a new configuration set in any of these ways:
   - From the **Add** menu, select **Configuration**.
   - On the toolbar, click the **Add Configuration** button  .
   - Select the **Configurations** node below the model node. In the **Contents** pane, right-click an existing configuration set and copy and paste the configuration set.
5. Select the **Configurations** node below the model node. The new configuration set, `Configuration1`, appears in the **Contents** pane. The default configuration, `Configuration`, is still the active configuration for the model.
6. On the **Contents** pane, double-click the name `Configuration1` and rename the configuration to `FixedStepConfig`. You specify the fixed-step solver in the following section.

7 Rename the configuration `Configuration` to `VariableStepConfig`.

Change Configuration Parameter Values in a Configuration Set

To change the parameter values of a configuration set, open the Configuration Parameters dialog box for that configuration. You can open and change any configuration set, whether or not it is active.

For this example, change the configuration `FixedStepConfig` to specify a fixed-step solver instead of the default variable-step solver.

1 Open the Model Explorer.

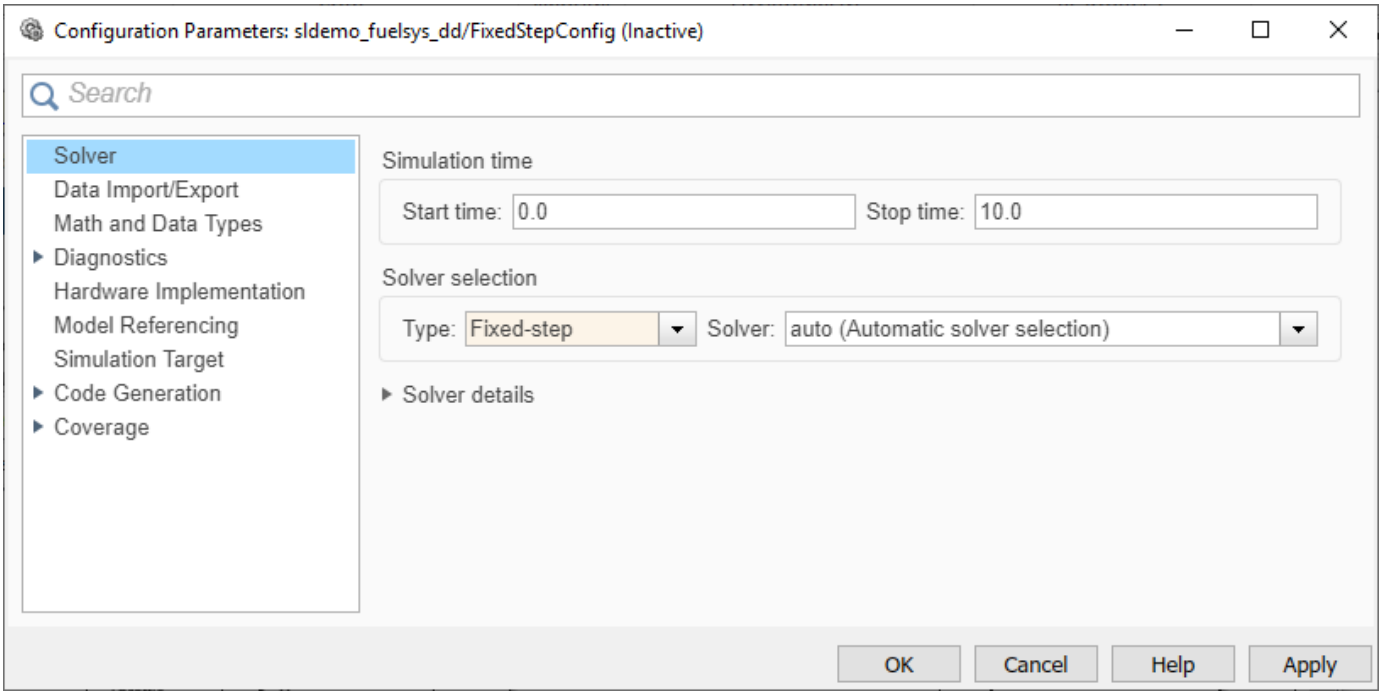
2 Expand the model node and select the **Configurations** node below it.

3 In the **Contents** pane, right-click the configuration set `FixedStepConfig` and click **Open**.

The configuration set opens in the Configuration Parameters dialog box.

Note Every configuration set has its own Configuration Parameters dialog box. As you change the state of a configuration set, the title bar of the dialog box changes to reflect the state.

4 On the **Solver** pane, set the **Type** parameter to `Fixed-step`. Click **OK**.



The model now contains two configurations, `VariableStepConfig` and `FixedStepConfig`, which use different solver types. You can compare how the solver settings affect simulation by changing the active configuration and simulating the model.

Activate a Configuration Set

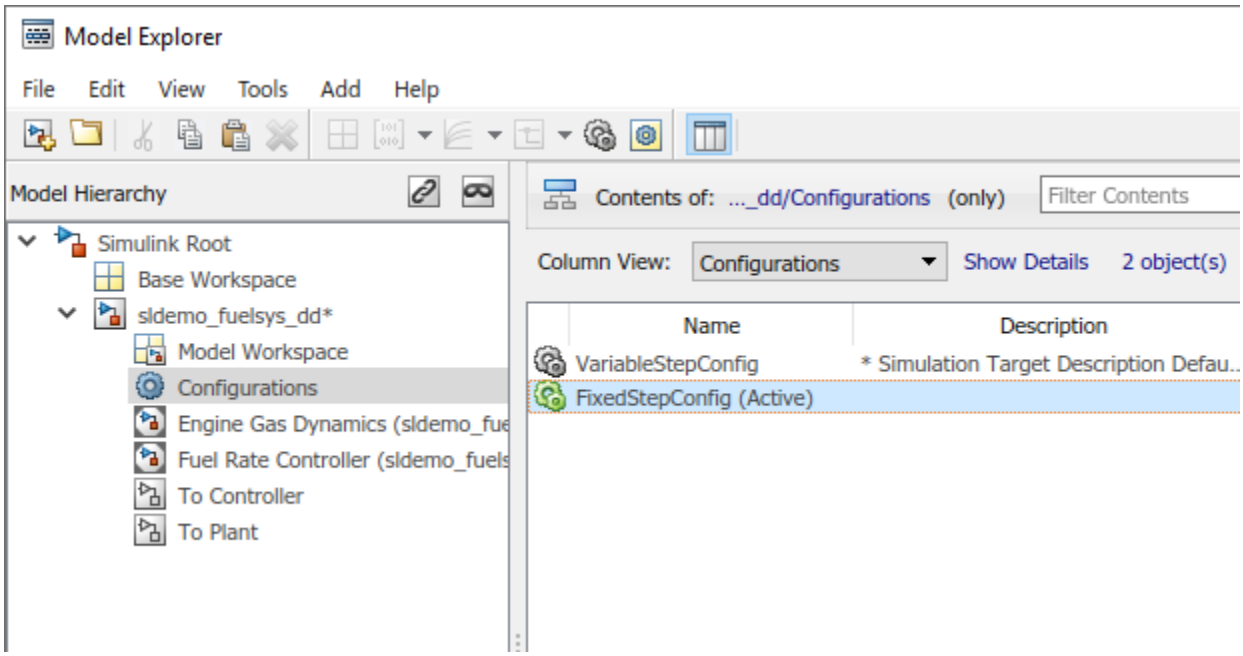
Only one configuration set associated with a model is active at any given time. The active set determines the current values of the model parameters. You can change parameter values in the

active or inactive set at any time (except when executing the model). In this way, you can quickly reconfigure a model for different purposes, such as testing and production.

To activate the fixed-step configuration that you created in the previous section:

1. Open the Model Explorer.
2. Expand the model node and select the **Configurations** node below it.
3. In the **Contents** pane, right-click the configuration set `FixedStepConfig` and click **Activate**.

   The active configuration displays (Active) to the right of the configuration name.



Copy, Delete, and Move a Configuration Set

You can use the Model Explorer **Edit** or context menus to delete, copy, and move configuration sets among models displayed in the **Model Hierarchy** pane.

For this example, copy your configuration `FixedStepConfig` to the model `vdp`.

1. Open the model `vdp` and open the Model Explorer.
2. In the **Model Hierarchy** pane, expand the node of the model `sldemo_fuelsys_dd` and select the Configurations node below it.
3. In the **Contents** pane, right-click `FixedStepConfig` and click **Copy** in the context menu.
4. In the **Model Hierarchy** pane, right-click the model node `vdp` and click **Paste**.
5. Activate the configuration `FixedStepConfig` for the model `vdp`.

To copy the configuration set using drag-and-drop, hold down the right mouse button and drag the configuration set to the Configurations node of the model in which you want to create the copy.

To move a configuration set from one model to another using drag-and-drop, hold the left mouse button down and drag the configuration set to the Configurations node of the destination model.

Note You cannot move or delete an active configuration set from a model.

Save a Configuration Set

When you store a configuration set in a model, the configuration set is saved when you save the model. Alternatively, you can store the configuration set outside of the model as a freestanding configuration set to share the configuration with other models. You can save the configuration set in a Simulink data dictionary, or export the configuration set to a MAT-file or to a script. If you store a freestanding configuration set in the base workspace, to save it, you must export it to a MAT-file or script.

Simulink Data Dictionary

To save your configuration set outside of the model, store the configuration in a Simulink data dictionary. You can share and archive the configuration by using the data dictionary. To use the configuration in a model, use a configuration reference. For more information, see “Share a Configuration with Multiple Models” on page 13-10.

Exported File

You can also save the settings of a configuration set as a `Simulink.ConfigSet` object in a MAT-file or as a MATLAB function or script. Export the configuration set to a MATLAB function or script when you want to compare the settings in different configuration sets. However, when you want to preserve a freestanding configuration set, exporting the configuration to a file is not recommended because you must manually load the configuration set when you want to use it. Save the configuration set in a Simulink data dictionary instead.

For this example, use the Model Explorer to save the configuration set `FixedStepConfig`, which you copied to the model `vdp`.

1. Open the Model Explorer.
2. In the **Model Hierarchy** pane, expand the model node `vdp` and select the Configurations node below it.
3. In the **Contents** pane, right-click the configuration `FixedStepConfig` and select **Export**.
4. In the Export Configuration to File dialog box, specify the name of the file and the file type. For this example, save the configuration as `FixedStepConfig.m`.

   If you specify a `.m` extension, the file contains a MATLAB function that creates a configuration set object. If you specify a `.mat` extension, the file contains a configuration set object.

   Note

   - Do not specify the name of the file to be the same as a model name. If the file and model have the same name, the software cannot determine which file contains the configuration set object when loading the file.
   - To use the configuration set in a later release, specify a `.mat` extension. If you specify a `.m` extension, in rare cases, parameter values might change due to changing dependencies.

5. Click **Save**. The Simulink software saves the configuration set.

Load a Saved Configuration Set

To load the configuration set that you saved from the model `vdp`:

1. Open the model `vdp`.
2. Open the Model Explorer.
3. In the **Model Hierarchy** pane, right-click the model and select **Configuration** > **Import**.
4. In the Import Configuration From File dialog box, select the M file that contains the function to create the configuration set object, or the MAT-file that contains the configuration set object. For this example, select `FixedStepConfig.m`.
5. Click **Open**. The Simulink software loads the configuration set.

Note

- If you load a configuration set object that contains an invalid custom target, the software sets the "System target file" (Simulink Coder) parameter to `ert.tlc`.
- If you load a configuration set that contains a component that is not available on your system, the parameters in the missing component are reset to their default values.

Compare Configuration Sets

When you save two configuration sets as M files or as MAT-files, you can visually compare them by using the `visdiff` function. This function opens the Comparison Tool and presents the differences between the two files. For more information about the Comparison Tool, see "Compare Simulink Models" on page 21-6.

For this example, compare the default configuration `Configuration` to the configuration `FixedStepConfig`, which you copied to the model `vdp`.

1. Save the configuration `FixedStepConfig` to the file `FixedStepConfig.m`, as shown in "Save a Configuration Set" on page 13-8.
2. Save the second configuration, `Configuration`, to the file `DefaultConfig.m` by following the same procedure.
3. Compare the files.

```
visdiff('FixedStepConfig.m','DefaultConfig.m');
```

See Also

More About

- "Set Model Configuration Parameters for a Model" on page 13-2

Share a Configuration with Multiple Models

In this section...

To share a configuration set with multiple models, store it as a *freestanding configuration set* in a Simulink data dictionary or in the base workspace. By default, a configuration set resides within a single model so that only that model can use it. A freestanding configuration set is a `Simulink.ConfigSet` object that you store outside of your models so that multiple models can use it.

To use a freestanding configuration set in a model, create a *configuration reference* in the model that points to the freestanding configuration set. You can then activate the configuration reference in the same way as a standard configuration set. Multiple models can reference the same freestanding configuration set.

Use configuration references when you want to:

- Use the same configuration parameters in multiple models. When you change parameter values in the freestanding configuration, the changes apply to each model that references the configuration.

  To share a configuration set across a model hierarchy, you can propagate the reference from the top model to its referenced models. For more information, see “Share a Configuration Across Referenced Models” on page 13-18.
- Change configuration parameters for any number of models without changing the model files. When you store a configuration set in a Simulink data dictionary, changing parameter values in the configuration changes the data dictionary file. Models that are linked to the data dictionary and reference the configuration set use the new values, but their model files are not changed.
- Quickly replace the configuration sets of any number of models without changing the model files. When you store a configuration set in a Simulink data dictionary, you can point to that configuration from a reference that is also stored in the data dictionary. Your models can then reference the data dictionary's configuration reference. When you change the data dictionary's reference to point to a different configuration set, the models use the new configuration.

  When a configuration reference references a configuration in the base workspace, it points to a variable that represents the `Simulink.ConfigSet` object. Assigning a different configuration set to the variable assigns that configuration set to each model that references the variable.

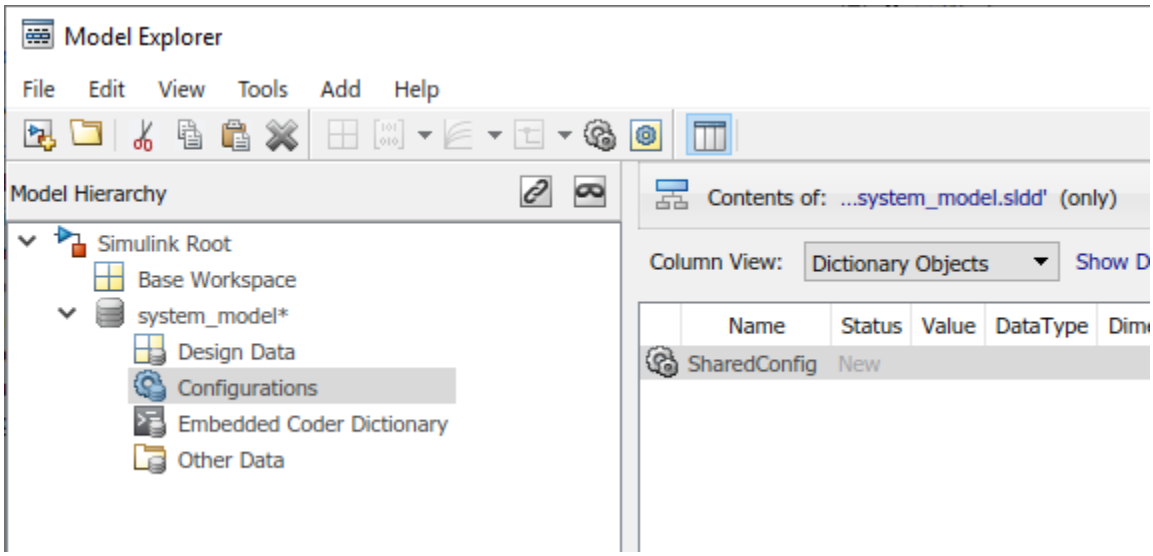
For this example, you store a configuration set in a Simulink data dictionary and reference the configuration set from models that are linked to the data dictionary. To create and link a Simulink

data dictionary, see “Migrate Models to Use Simulink Data Dictionary” on page 74-6. To share a configuration set that is already saved in a model, convert the configuration to a referenced freestanding configuration, as shown in “Share a Configuration Across Referenced Models” on page 13-18.

Create a Configuration Set in the Data Dictionary

Before you reference a freestanding configuration set from your models, create the configuration in a Simulink data dictionary and link it to your models. For this example, use the Simulink project `sldemo_slproject_airframe`. The project contains the Simulink data dictionary `system_model` and multiple models that are linked to the data dictionary.

1. Open the project. At the command line, type `sldemo_slproject_airframe`.
2. In the project folder, in the folder `data`, double-click the Simulink data dictionary `system_model.sldd`. The data dictionary opens in the Model Explorer.
3. In the **Model Hierarchy** pane, expand the data dictionary node for `system_model`. Right-click the node and click **Show Empty Sections**.
4. Select the Configurations node and click the **Add Configuration** button  . The configuration set object appears in the **Contents** pane, with the default name, `Configuration`.
5. Name the new configuration `SharedConfig`.

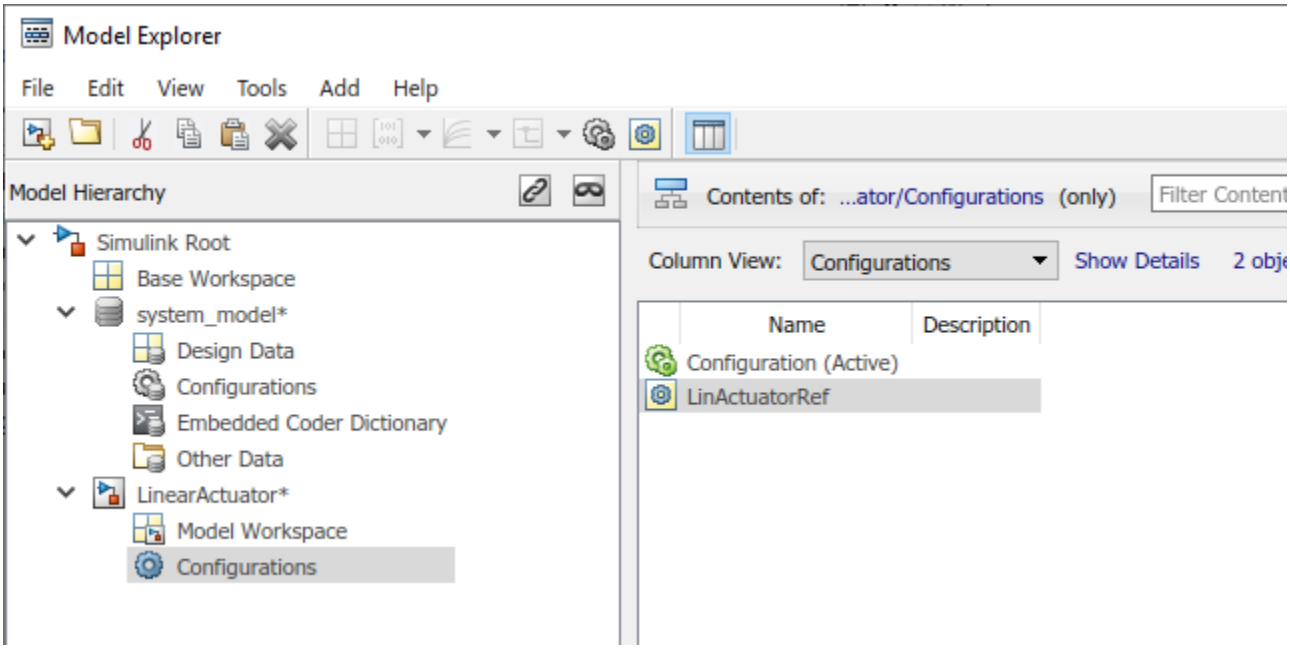


Models that have access to the data dictionary `system_model.sldd` can reference the new configuration.

Create and Attach a Configuration Reference

To use your freestanding configuration set in a model, attach a configuration reference that points to the configuration set. For this example, use the model `LinearActuator`. The model is linked to the Simulink data dictionary `system_model.sldd`, which contains your new configuration `SharedConfig`.

1. Open the model. At the command line, type `LinearActuator`.
2. In the Model Explorer, in the **Model Hierarchy** pane, select the model node.
3. Select **Add** > **Configuration Reference** or click the **Add Configuration Reference** button  .
4. Under the model node, click **Configurations**. A new configuration reference named `Reference` is listed.
5. Name the configuration reference `LinActuatorRef`.



The new configuration reference is attached to the model, but it does not point to a freestanding configuration yet and it is not active. To complete the setup, resolve and activate the configuration reference.

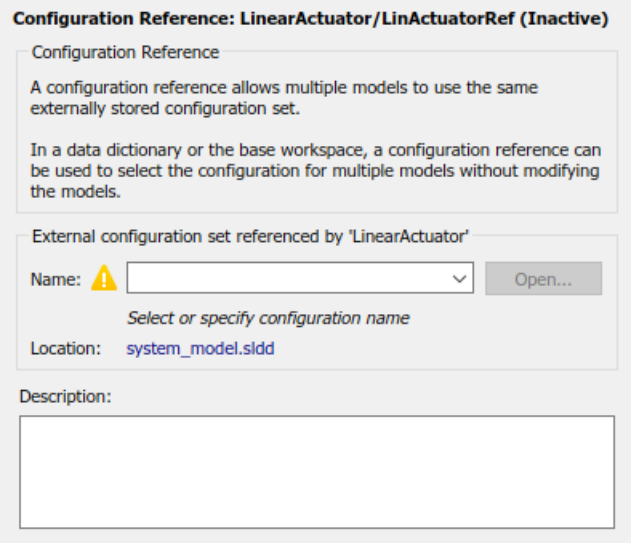
Resolve a Configuration Reference

An unresolved configuration reference is a configuration reference that is not pointing to a valid configuration set object. When you create a configuration reference by using the preceding steps, the reference is unresolved.

To resolve the configuration reference that you created:

1. In the **Model Hierarchy** pane, under the model node for the `LinearActuator` model, select the Configurations node. In the **Contents** pane, select the unresolved configuration reference, `LinActuatorRef`.

   The right pane shows that the configuration reference is unresolved.



2 Use the **Name** drop-down menu to select `SharedConfig`, which you created in the data dictionary.

Tip You can specify the name of a configuration reference instead of a configuration set. However, nesting a configuration reference beyond this depth results in an error.

3 Click **Apply**. The warning icon disappears and the reference points to your freestanding configuration set.

If your configuration reference is already resolved, you can follow these steps to change which configuration set it references.

Activate a Configuration Reference

After you create the configuration reference and attach it to the model, activate the reference to use the referenced configuration in the model.

1 In the **Model Hierarchy** pane, under the model node for the `LinearActuator` model, select the Configurations node. In the **Contents** pane, select the configuration reference `LinActuatorRef`.

2 Right-click the configuration reference `LinActuatorRef` and select **Activate**.

When the configuration reference is active, the Model Explorer shows the name of the reference with the suffix `(Active)`. The freestanding configuration set now provides the configuration parameters for the model.

Create a Configuration Reference in Another Model

For this example, you will update the configuration set and see how it affects its associated models. Repeat the process above to associate `SharedConfig` with a second model:

1 Open the model `NonLinearActuator`. This model is also linked to the data dictionary that contains the freestanding configuration set.

2. In Model Explorer, add a configuration reference to the model `NonLinearActuator`.
3. Name the configuration reference `NonLinActuatorRef`.
4. Point the reference to the freestanding configuration set `SharedConfig`.
5. Activate the configuration reference.

Both models now contain a configuration reference that points to the same configuration set object in the Simulink data dictionary.

Change Parameter Values in a Referenced Configuration Set

You can edit a freestanding configuration set by opening it from the Configuration Reference dialog box of a reference that points to the configuration set. Changing the freestanding configuration set affects the configuration references that refer to it, except for parameters that are overridden in those references. To edit the configuration set that you reference from the models:

1. Open one of the models that references the configuration set. For this example, open the model `LinearActuator`.
2. To open the Configuration Reference dialog box, on the **Modeling** tab, click the **Model Settings** button  . The Configuration Reference dialog box displays a read-only view of the referenced configuration `SharedConfig`. In the dialog box you can browse, search, and get context-sensitive help for the parameters in the same way you do in the Configuration Parameters dialog box.
3. At the top of the Configuration Reference dialog box, click the Model Configuration Parameters icon  . The Configuration Parameters dialog box opens. You can now change and apply parameter values as you would for any configuration set.

   Note Some options in the configuration set cannot be used in a freestanding configuration because they perform actions on one specific model. For example, the **Data Import/Export > Connect Input** button is not supported in freestanding configuration sets because it opens the Root Inport Mapper for the model that uses the configuration.
4. On the **Solver** pane, set the **Type** parameter to `Fixed-step`. Click **Apply**, then **OK**.
5. Your applied changes appear in the Configuration Reference dialog box. The models that reference the freestanding configuration `SharedConfig` use the new solver type.

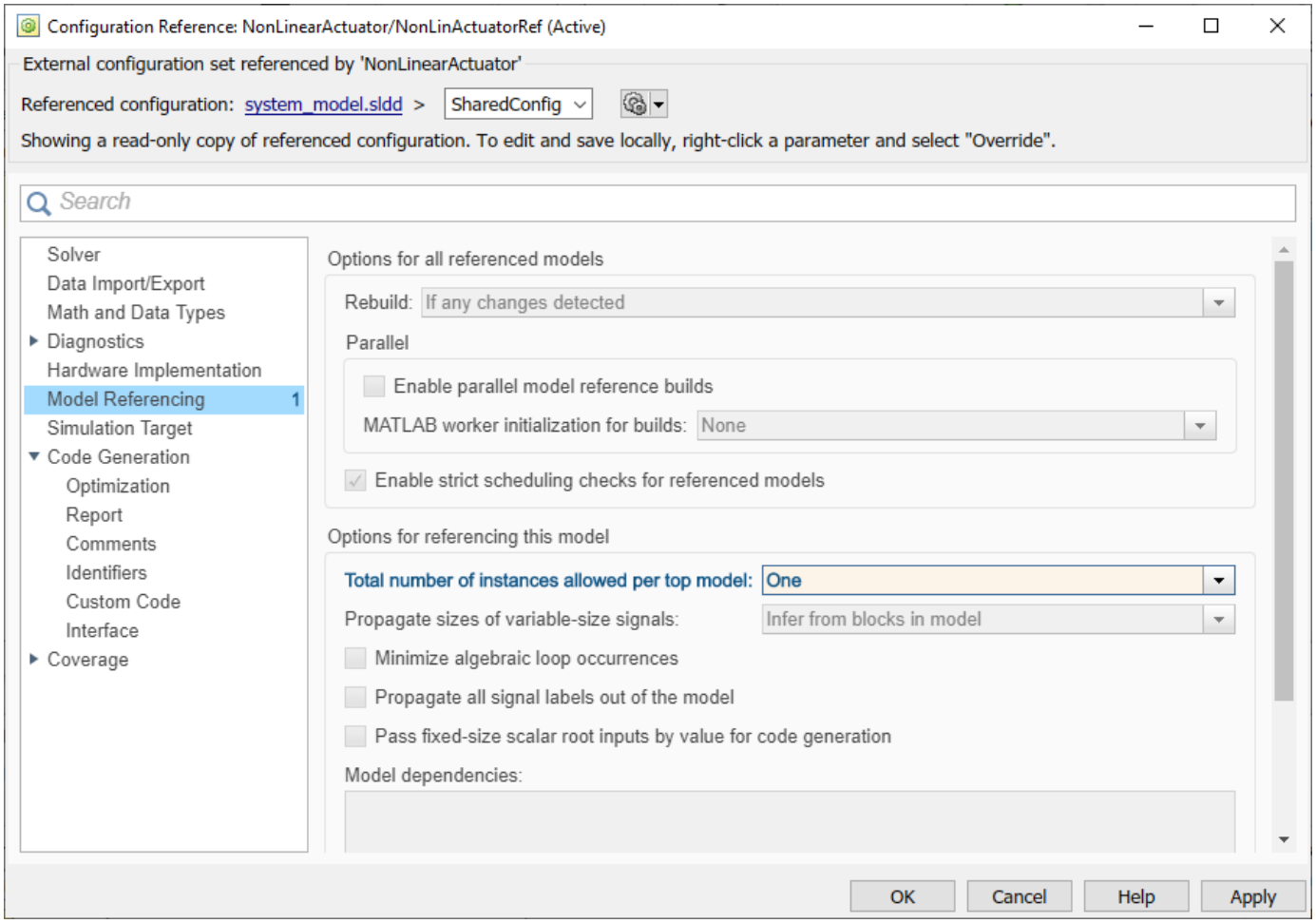
Change Parameter Value in a Configuration Reference

You can override individual parameter values for models that reference freestanding configuration sets without changing the freestanding configuration. For an overridden parameter, the reference uses the value you assign locally instead of the value in the referenced configuration set. For example, suppose that `LinearActuator` and `NonLinearActuator` are both in the same model hierarchy. You want `LinearActuator` to be referenced as many times as it needs to be, but want Simulink to return an error if `NonLinearActuator` is referenced more than one time. In this example, you can override the **Model Referencing > Total number of instances allowed per top model** parameter for only the `NonLinearActuator` model.

For this example, override the parameter in the configuration reference for the model `NonLinearActuator`.

1. Open the model `NonLinearActuator`.
2. To open the Configuration Reference dialog box, on the **Modeling** tab, click the **Model Settings** button  . The active configuration reference displays a read-only view of the referenced configuration.
3. Right-click the parameter you want to change and select **Override** from the context menu. You can now change the parameter value as you would for any configuration set. For this example, override the parameter **Model Referencing > Total number of instances allowed per top model** and set the value to `One`.

   In the left pane, each pane displays the number of overridden parameters it contains.



4. Click **Apply** to keep the changes or **Cancel** to restore the parameter to the referenced value.

   Changes you make to a parameter value apply only to the configuration reference in which you override and edit the parameter. They do not affect other references to the referenced configuration set. For this example, the model `NonLinearActuator` allows a top model to reference it once, while the model `LinearActuator` allows a top model to reference it multiple times.

To restore an overridden parameter to its value from the referenced configuration set, right-click the overridden parameter and select **Restore** from the context menu. The overridden parameter resets to the value in the referenced configuration and becomes read-only again.

Save a Referenced Configuration Set

If you store your freestanding configuration set in a Simulink data dictionary, you can save changes to the configuration by saving the data dictionary.

If your model references a configuration set that you store in the base workspace, before you exit MATLAB, you need to save the referenced configuration set to a MAT-file or MATLAB script.

1. In the Model Explorer, in the Model Hierarchy, select **Base Workspace**.
2. In the **Contents** pane, right-click the name of the referenced configuration set object.
3. From the context menu, select **Export Selected**.
4. Specify the filename for saving the configuration set as either a MAT-file or a MATLAB script.

Tip When you reopen the model you must load the saved configuration set, otherwise the configuration reference is unresolved. To set up your model to automatically load the configuration set object, see “Callbacks for Customized Model Behavior” on page 4-44.

Load a Saved Referenced Configuration Set

If your configuration reference uses a configuration set that you exported to a MAT-file or MATLAB script, you need to load the referenced configuration set from the file to the base workspace.

1. In the Model Explorer, in the Model Hierarchy, right-click **Base Workspace**.
2. From the context menu, select **Import**.
3. Specify the filename for the saved configuration set and select OK. The configuration set object appears in the base workspace.

Configuration Reference Limitations

- A configuration reference can point to another configuration reference, but you cannot nest a configuration reference beyond the second reference.
- If you activate a configuration reference when using a custom target, the `ActivateCallback` function does not trigger to notify the corresponding freestanding configuration set. Likewise, if a freestanding configuration set switches from one target to another, the `ActivateCallback` function does not trigger to notify the new target. This behavior occurs even if an active configuration reference points to that target. For more information about `ActivateCallback` functions, see “rtwgensettings Structure” (Simulink Coder).
- Not all parameters in a reference can be overridden, for example, parameters that must be consistent in a model reference hierarchy cannot be overridden.

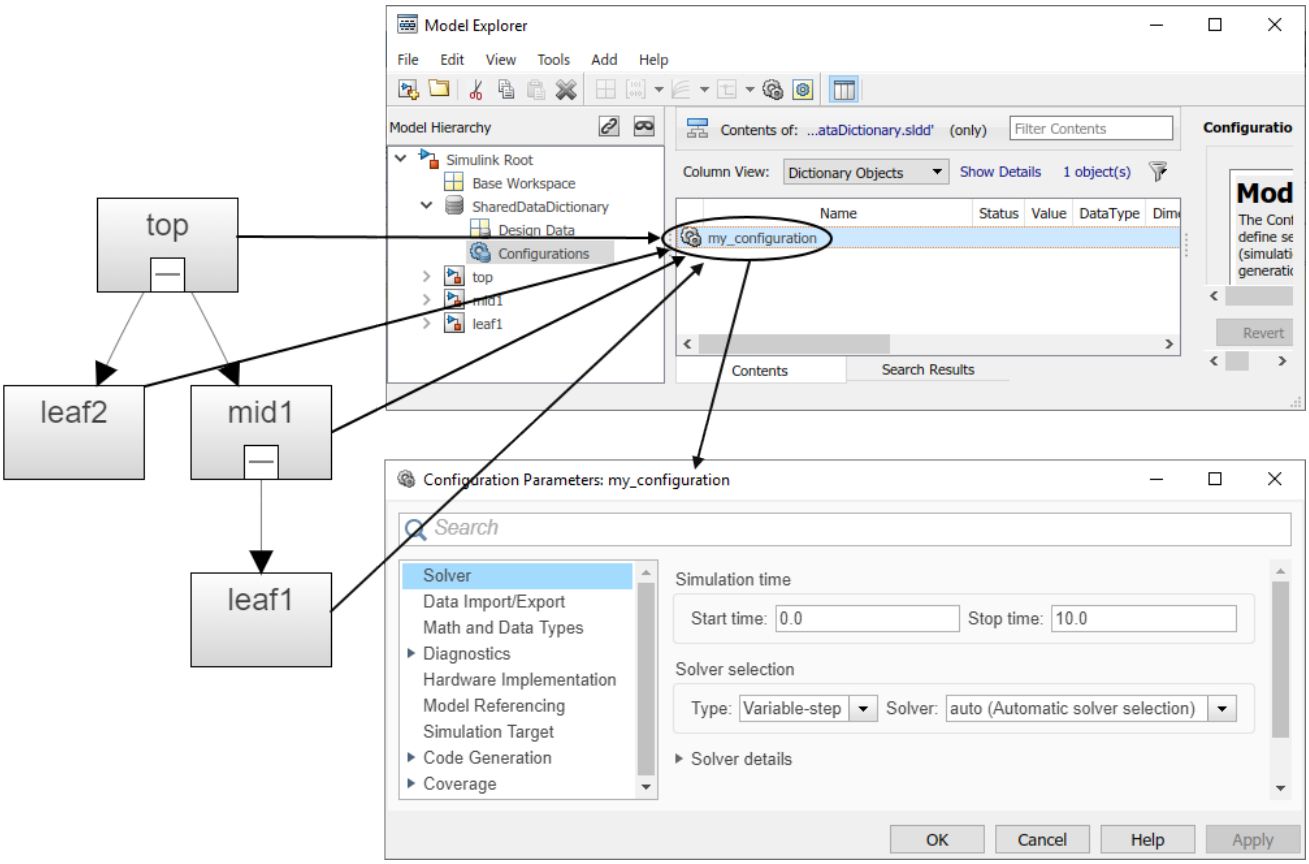
See Also

Related Examples

- “Share a Configuration Across Referenced Models” on page 13-18
- “Create a Template from a Model” on page 4-2

Share a Configuration Across Referenced Models

This example shows how to share the same configuration set for the top model and referenced models in a model reference hierarchy. You can use a configuration reference in each of the models to reference the same configuration set object in a Simulink data dictionary that the models are linked to.



In the diagram, each model shown in the Dependency Analyzer specifies a configuration reference as its active configuration set. Each reference points to the freestanding configuration set, `my_configuration`. Therefore, the parameter values in `my_configuration` apply to all four models. Any parameter change in `my_configuration` applies to all four models. For more information about configuration references, see “Share a Configuration with Multiple Models” on page 13-10.

Link Models to Simulink Data Dictionary

Create a Simulink data dictionary to store the configuration set. When you link the models in the hierarchy to the data dictionary, they can reference the configuration set.

1. Open the `sldemo_mdlref_depgraph` model. At the command line, type `sldemo_mdlref_depgraph`. Verify that your current folder is a writable folder.
2. On the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
3. In the Model Properties dialog box, click **New**. Name the new Simulink data dictionary `SharedDataDictionary` and click **Save**.

4 Click **OK**. Because this model includes referenced models, a pop-up window asks you if `SharedDataDictionary.sldd` should be used for all referenced models that do not already use a dictionary. Click **Change all models**. The current model and all of its referenced models are linked to the new data dictionary. When the data dictionary is linked, click **OK**.

The models are now linked to the Simulink data dictionary, `SharedDataDictionary.sldd`. When you store a configuration set in this data dictionary, the models that are linked to it can reference the configuration set.

Convert Configuration Set to Configuration Reference

In the top model, you must convert the active configuration set to a configuration reference:

1 Open the Model Explorer.

2 In the **Model Hierarchy** pane, expand the top model, `sldemo_mdlref_depgraph`. In the list, select the **Configurations** node, and right-click `Configuration (Active)` in the **Contents** pane. In the context menu, select **Convert to Configuration Reference**.

3 In the **Name** field, use the default name, `Configuration`. This configuration set object is stored in the data dictionary `SharedDataDictionary.sldd`.

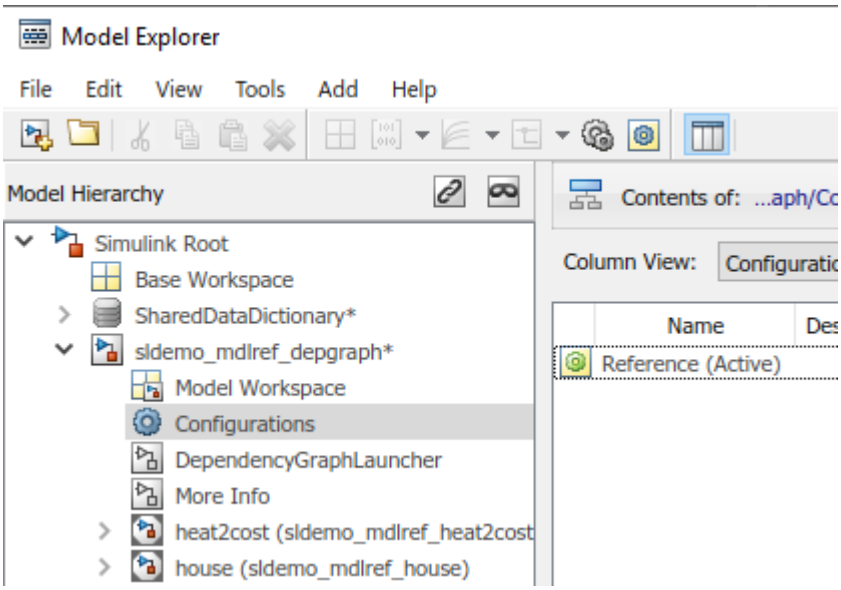
4 Click **OK**.

The original configuration set is now stored as a configuration set object, `Configuration`, in the Simulink data dictionary. The active configuration for the top model is now a configuration reference that points to the configuration set object in the data dictionary.

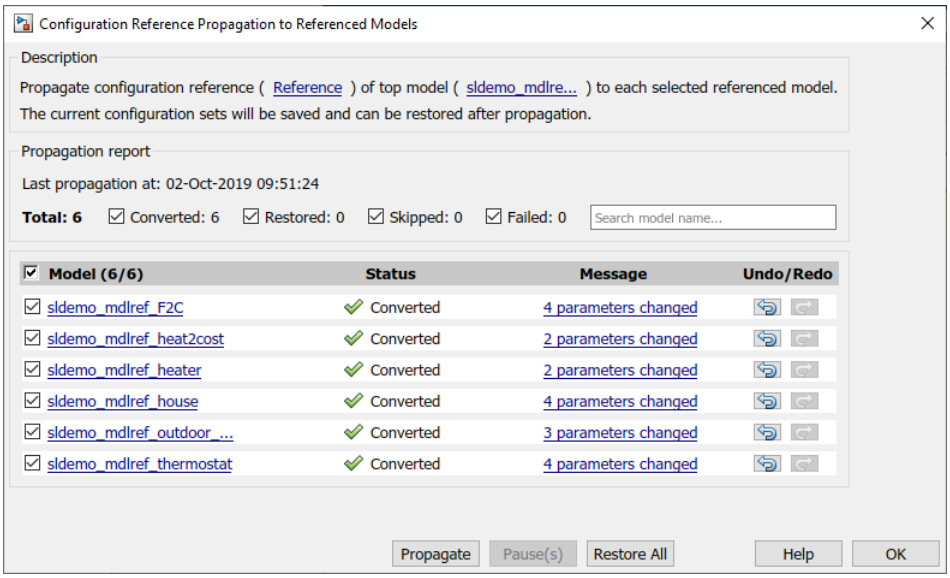
Propagate a Configuration Reference

Now that the top model contains an active configuration reference, you can propagate this configuration reference to all of the child models. Propagation creates a copy of the top model configuration reference in each referenced model and makes it the active configuration. The configuration references point to the configuration set object in the data dictionary.

1 In the Model Explorer, in the **Model Hierarchy** pane, expand the `sldemo_mdlref_depgraph` node and select the **Configurations** node.



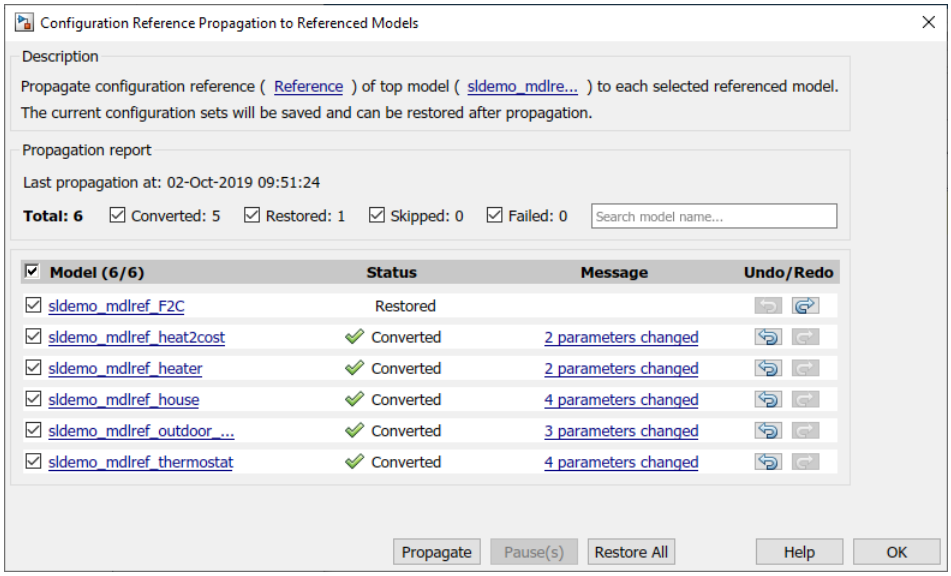
2. In the **Contents** pane, right-click the active configuration reference, **Reference (Active)**. In the context menu, select **Propagate to Referenced Models**.
3. In the Configuration Reference Propagation dialog box, select the check box for each referenced model. In this example, they are already selected.
4. The propagation mechanism saves the original configuration parameters for each referenced model so that you can undo the propagation. Click **Propagate**.
5. In the Propagation Confirmation dialog box, click **OK**.
6. In the Configuration Reference Propagation dialog box, the Propagation Report is updated and the **Status** for each referenced model is marked as `Converted`.



Now, each model in the hierarchy references the freestanding configuration `Configuration`. If you want one model to use a different value for a parameter, you can override individual parameters within the reference that the model uses. For more information, see “Change Parameter Value in a Configuration Reference” on page 13-14.

Undo a Configuration Reference Propagation

After propagating a configuration reference from a top model to the referenced models, you can undo the propagation for all referenced models by clicking **Restore All**. If you want to undo the propagation for individual referenced models, in the **Undo/Redo** column, click the **Undo** button. The Propagation Report is updated and the **Status** for the referenced model is set to `Restored`.



See Also

Related Examples

- “Share a Configuration with Multiple Models” on page 13-10

Automate Model Configuration by Using a Script

If you want to use the same configuration setup for many models, you can write a script to programmatically configure each model in the same way. You can use the script to archive and compare the configuration settings that your models use.

This example shows three different ways to programmatically set up your model's configuration:

- Edit the model's existing active configuration set
- Create and edit a new configuration set in the model
- Create a configuration reference that points to a freestanding configuration set

For this example, use the model `sldemo_mdlref_depgraph`.

```
model = 'sldemo_mdlref_depgraph';
open_system(model)
```

Edit the Active Configuration Set

To manipulate a configuration set that is associated with a model, use the `ConfigSet` object that represents the configuration set. For this example, use the configuration that is active for the model.

```
activeConfigObj = getActiveConfigSet(model);
get_param(activeConfigObj,'Name')
```

```
ans =
'Configuration'
```

The active configuration for the model is `Configuration`. To see the current values of parameters in the configuration, use the `get_param` function and the `ConfigSet` object.

```
get_param(activeConfigObj,'StopTime')
```

```
ans =
'2*24*60*60'
```

Rename the configuration set to `UpdatedConfig`.

```
set_param(activeConfigObj,'Name','UpdatedConfig');
```

For this example, set a stop time of `200` and change the solver type to a variable-step solver.

```
set_param(activeConfigObj,'StopTime','200');
set_param(activeConfigObj,'SolverType','Variable-step');
```

Create and Activate a Configuration Set

When you want to change the model's configuration and preserve the original parameter values of its active configuration, create and activate a new configuration set in the model. To create another configuration set, copy an existing configuration set and attach the copy to the model. To avoid naming conflicts when you attach the copy, either rename the copy before attaching it or set `allowRename`, the optional third argument of `attachConfigSet`, to `true`.

For this example, copy the active configuration set. Rename the copy to `ConfigCopy` and attach it to the model.

```
newConfigObj = copy(activeConfigObj);
set_param(newConfigObj,'Name','ConfigCopy');
attachConfigSet(model, newConfigObj);
```

When you attach a configuration set to a model, it is inactive. You can manipulate inactive configurations in the same way that you manipulate the active configuration set. To use the new configuration, activate it for the model.

```
set_param(newConfigObj,'SolverType','Fixed-step');

setActiveConfigSet(model,'ConfigCopy');
activeConfigSet = getActiveConfigSet(model);
get_param(activeConfigObj,'Name')
```

```
ans =
'UpdatedConfig'
```

Now, `ConfigCopy` is the active configuration set.

Set Up a Configuration Reference

If you want to store the configuration set outside of your model, create a script that sets up a configuration reference in the model. The reference is stored in the model and it points to a freestanding configuration set, which is stored in either a Simulink data dictionary or in the base workspace. Use a freestanding configuration set and configuration references to share one configuration with multiple models. You can also use a freestanding configuration set when you want to edit the configuration without changing the model file.

For this example, configure the model `vdp` to use a configuration reference. First, create a freestanding configuration set in the base workspace by copying the model's active configuration set. The freestanding configuration is a `ConfigSet` object represented by the variable `freeConfigSet`. You can skip this step if you want to reference an existing freestanding configuration set.

```
model = 'vdp';
open_system(model)

freeConfigSet = copy(getActiveConfigSet(model));
```

Create a configuration reference. To point the reference to your freestanding configuration, set the `SourceName` property to `freeConfigSet`, the variable that represents your configuration. The new reference is a `ConfigSetRef` object represented by the variable `configRef`. Name the reference `vdpConfigRef`.

```
configRef = Simulink.ConfigSetRef;
set_param(configRef,'SourceName','freeConfigSet')
set_param(configRef,'Name','VdpConfigRef')
```

Attach the configuration reference to the model `vdp` by using the `ConfigSetRef` object. You can attach the reference to only one model. To use the configuration reference in the model, activate it.

```
attachConfigSet('vdp',configRef);
setActiveConfigSet('vdp','VdpConfigRef');
```

Now, when you change the configuration set that the object `freeConfigSet` represents, the changes apply to the model.

You can obtain parameter values in a configuration reference by using `get_param`. However, you cannot change parameter values directly in the configuration reference. To change the values, you

must use the `ConfigSet` object that represents the referenced freestanding configuration set. Get the freestanding configuration set from a configuration reference by using the `getRefConfigSet` method.

```
referencedConfigObj = getRefConfigSet(configRef);
```

Now, `referencedConfigObj` represents the same freestanding configuration set that your models reference. `freeConfigSet` represents that configuration set as well. Use the configuration set object to change parameter values in the referenced configuration set. These changes apply to each model that references the configuration.

```
set_param(referencedConfigObj,'SignalLogging','off');
set_param(referencedConfigObj,'StartTime','10');
```

See Also

Related Examples

- “Manage Configuration Sets for a Model” on page 13-5
- “Create a Template from a Model” on page 4-2

Configuration Object Functions

To interact with a configuration, programmatically, use a `Simulink.ConfigSet` object or a `Simulink.ConfigSetRef` object. You can use the following functions to get information about the configuration. For more information about using configurations, see “Manage Configuration Sets for a Model” on page 13-5.

Function	Purpose
`getFullName`	Return the full path name of a configuration set or configuration reference as a character vector.
`getModel`	Return the model that owns the configuration set or configuration reference as a handle to the model.
`getRefConfigSet`	Return the configuration set that a configuration reference points to.
`isActive`	Determine if the configuration set or configuration reference is the active configuration of the model, returned as a boolean value.
`isValidParam`	Determine if a specified parameter is a valid parameter of a configuration set.
`refresh`	Update 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.
`saveAs`	Save a configuration set to a MATLAB file.

Configuring Models for Targets with Multicore Processors

Concepts in Multicore Programming

In this section...

Basics of Multicore Programming

Multicore programming helps you create concurrent systems for deployment on multicore processor and multiprocessor systems. A *multicore processor system* is a single processor with multiple execution cores in one chip. By contrast, a *multiprocessor system* has multiple processors on the motherboard or chip. A multiprocessor system might include a Field-Programmable Gate Array (FPGA). An FPGA is an integrated circuit containing an array of programmable logic blocks and a hierarchy of reconfigurable interconnects. A *processing node* processes input data to produce outputs. It can be a processor in a multicore or multiprocessor system, or an FPGA.

The multicore programming approach can help when:

- You want to take advantage of multicore and FPGA processing to increase the performance of an embedded system.
- You want to achieve scalability so your deployed system can take advantage of increasing numbers of cores and FPGA processing power over time.

Concurrent systems that you create using multicore programming have multiple tasks executing in parallel. This is known as *concurrent execution*. When a processor executes multiple parallel tasks, it is known as *multitasking*. A CPU has firmware called a scheduler, which handles the tasks that execute in parallel. The CPU implements tasks using operating system threads. Your tasks can execute independently but have some data transfer between them, such as data transfer between a data acquisition module and controller for the system. Data transfer between tasks means that there is a *data dependency*.

Multicore programming is commonly used in signal processing and plant-control systems. In signal processing, you can have a concurrent system that processes multiple frames in parallel. In plant-control systems, the controller and the plant can execute as two separate tasks. Using multicore programming helps to split your system into multiple parallel tasks, which run simultaneously, speeding up the overall execution time.

To model a concurrently executing system, see “Partitioning Guidelines” on page 14-31.

Types of Parallelism

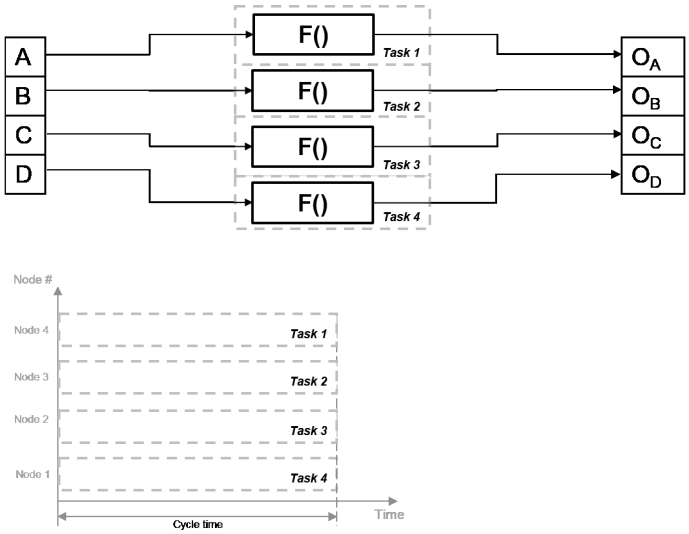
The concept of multicore programming is to have multiple system tasks executing in parallel. Types of parallelism include:

- Data parallelism
- Task parallelism
- Pipelining

Data Parallelism

Data parallelism involves processing multiple pieces of data independently in parallel. The processor performs the same operation on each piece of data. You achieve parallelism by feeding the data in parallel.

The figure shows the timing diagram for this parallelism. The input is divided into four chunks, A, B, C, and D. The same operation `F()` is applied to each of these pieces and the output is O_A, O_B, O_C, and O_D respectively. All four tasks are identical, and they run in parallel.



The time taken per processor cycle, known as cycle time, is `t = tF`.

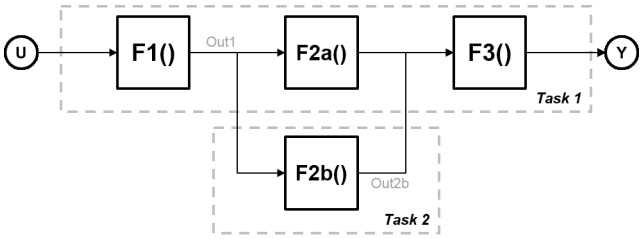
The total processing time is also `tF`, since all four tasks run simultaneously. In the absence of parallelism, all four pieces of data are processed by one processing node. The cycle time is `tF` for each task but the total processing time is `4*tF`, since the pieces are processed in succession.

You can use data parallelism in scenarios where it is possible to process each piece of input data independently. For example, a web database with independent data sets for processing or processing frames of a video independently are good candidates for data parallelism.

Task Parallelism

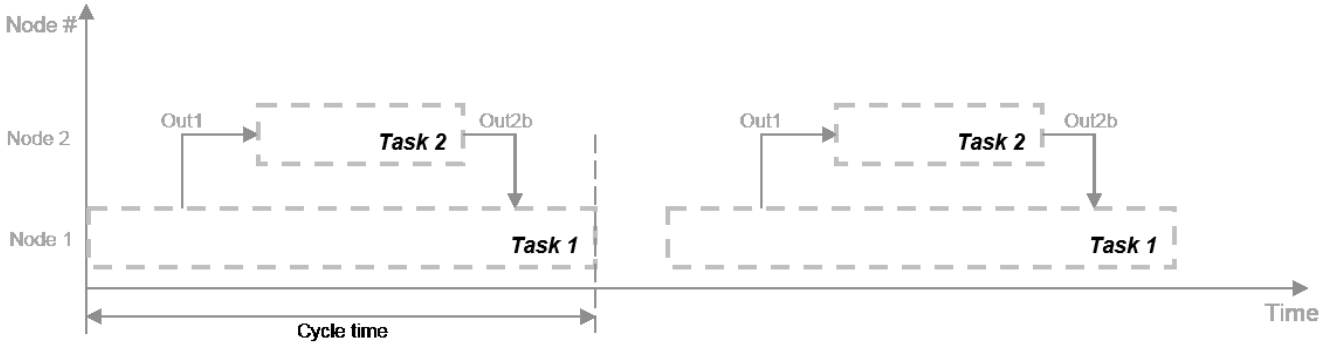
In contrast to data parallelism, task parallelism doesn’t split up the input data. Instead, it achieves parallelism by splitting up an application into multiple tasks. Task parallelism involves distributing tasks within an application across multiple processing nodes. Some tasks can have data dependency on others, so all tasks do not run at exactly the same time.

Consider a system that involves four functions. Functions F2a() and F2b() are in parallel, that is, they can run simultaneously. In task parallelism, you can divide your computation into two tasks. Function F2b() runs on a separate processing node after it gets data Out1 from Task 1, and it outputs back to F3() in Task 1.



The figure shows the timing diagram for this parallelism. Task 2 does not run until it gets data Out1 from Task 1. Hence, these tasks do not run completely in parallel. The time taken per processor cycle, known as cycle time, is

t = tF1 + max(tF2a, tF2b) + tF3.

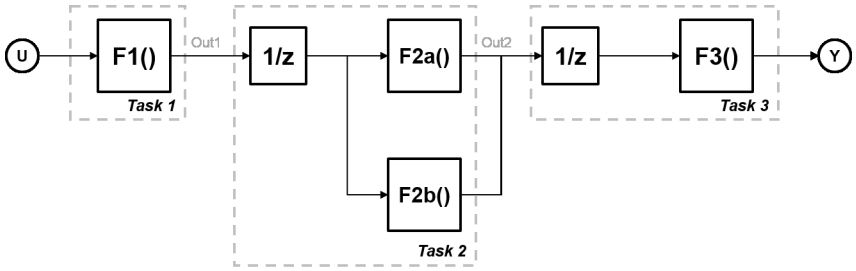


You can use task parallelism in scenarios such as a factory where the plant and controller run in parallel.

Model Pipeline Execution (Pipelining)

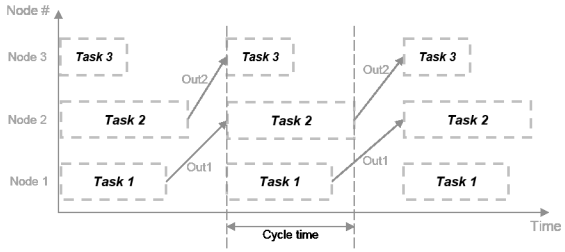
Use model pipeline execution, or pipelining, to work around the problem of task parallelism where threads do not run completely in parallel. This approach involves modifying your system model to introduce delays between tasks where there is a data dependency.

In this figure, the system is divided into three tasks to run on three different processing nodes, with delays introduced between functions. At each time step, each task takes in the value from the previous time step by way of the delay.



Each task can start processing at the same time, as this timing diagram shows. These tasks are truly parallel and they are no longer serially dependent on each other in one processor cycle. The cycle time does not have any additions but is the maximum processing time of all the tasks.

t = max(Task1, Task2, Task3) = max(tF1, tF2a, tF2b, tF3).



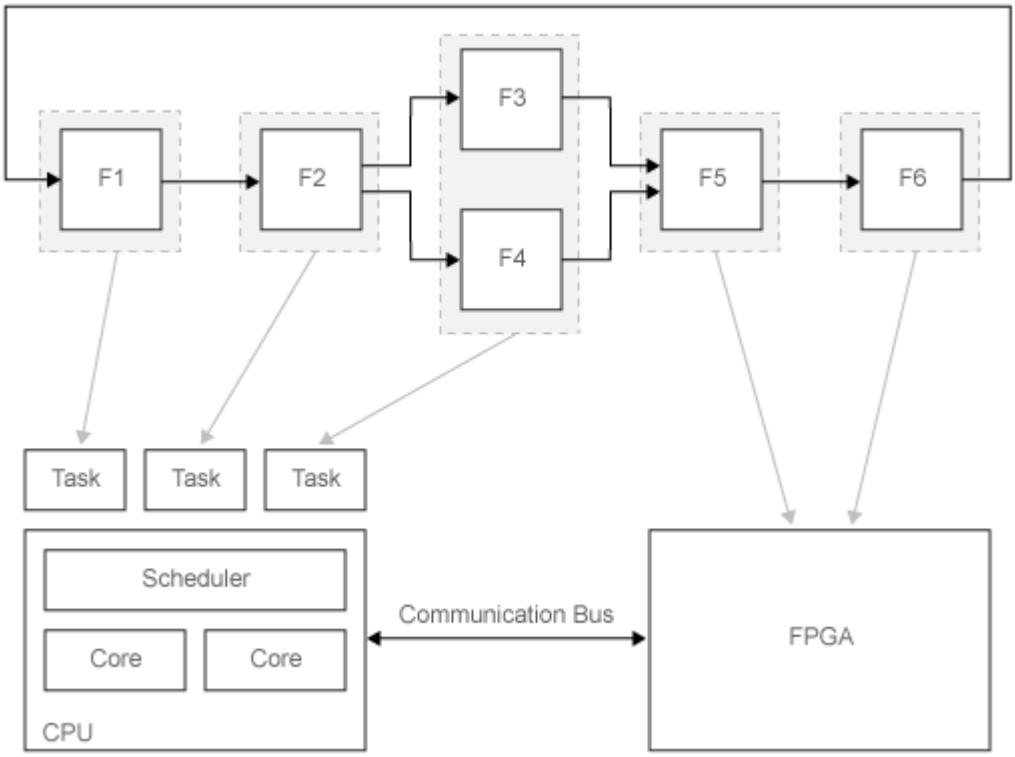
You can use pipelining wherever you can introduce delays artificially in your concurrently executing system. The resulting overhead due to this introduction must not exceed the time saved by pipelining.

System Partitioning for Parallelism

Partitioning methods help you to designate areas of your system for concurrent execution. Partitioning allows you to create tasks independently of the specifics of the target system on which the application is deployed.

Consider this system. F1–F6 are functions of the system that can be executed independently. An arrow between two functions indicates a data dependency. For example, the execution of F5 has a data dependency on F3.

Execution of these functions is assigned to the different processor nodes in the target system. The gray arrows indicate assignment of the functions to be deployed on the CPU or the FPGA. The CPU scheduler determines when individual tasks run. The CPU and FPGA communicate via a common communication bus.



The figure shows one possible configuration for partitioning. In general, you test different configurations and iteratively improve until you get the optimal distribution of tasks for your application.

Challenges in Multicore Programming

Manually coding your application onto a multicore processor or an FPGA poses challenges beyond the problems caused by manual coding. In concurrent execution, you must track:

- Scheduling of the tasks that execute on the embedded processing system multicore processor
- Data transfers to and from the different processing nodes

Simulink manages the implementation of tasks and data transfer between tasks. It also generates the code that is deployed for the application. For more information, see “Multicore Programming with Simulink” on page 14-8.

In addition to these challenges, there are challenges when you want to deploy your application to different architectures and when you want to improve the performance of the deployed application.

Portability: Deployment to Different Architectures

The hardware configuration that runs the deployed application is known as the architecture. It can contain multicore processors, multiprocessor systems, FPGAs, or a combination of these. Deployment of the same application to different architectures can require effort due to:

- Different number and types of processor nodes on the architecture
- Communication and data transfer standards for the architecture
- Standards for certain events, synchronization, and data protection in each architecture

To deploy the application manually, you must reassign tasks to different processing nodes for each architecture. You might also need to reimplement your application if each architecture uses different standards.

Simulink helps overcome these problems by offering portability across architectures. For more information, see “How Simulink Helps You to Overcome Challenges in Multicore Programming” on page 14-9.

Deployment Efficiency

You can improve the performance of your deployed application by balancing the load of the different processing nodes in the multicore processing environment. You must iterate and improve upon your distribution of tasks during partitioning, as mentioned in “System Partitioning for Parallelism” on page 14-5. This process involves moving tasks between different processing nodes and testing the resulting performance. Since it is an iterative process, it takes time to find the most efficient distribution.

Simulink helps you to overcome these problems using profiling. For more information, see “How Simulink Helps You to Overcome Challenges in Multicore Programming” on page 14-9.

Cyclic Data Dependency

Some tasks of a system depend on the output of other tasks. The data dependency between tasks determines their processing order. Two or more partitions containing data dependencies in a cycle creates a data dependency loop, also known as an *algebraic loop*.

Simulink identifies loops in your system before deployment. For more information, see “How Simulink Helps You to Overcome Challenges in Multicore Programming” on page 14-9.

See Also

Related Examples

- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Task Parallelism in Simulink” on page 14-14
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Multicore Programming with Simulink” on page 14-8

Multicore Programming with Simulink

In this section...

Using the process of partitioning, mapping, and profiling in Simulink, you can address common challenges of designing systems for concurrent execution.

Partitioning enables you to designate regions of your model as tasks, independent of the details of the embedded multicore processing hardware. This independence enables you to arrange the content and hierarchy of your model to best suit the needs of your application.

In a partitioned system, mapping enables you to assign partitions to processing elements in your embedded processing system. Use the Simulink mapping tool to represent and manage the details of executing threads, HDL code on FPGAs, and the work that these threads or FPGAs perform. While creating your model, you do not need to track the partitions or data transfer between them because the tool does this work. Also, you can reuse your model across multiple architectures.

Profiling simulates deployment of your application under typical computational loads. It enables you to determine the partitioning and mapping for your model that gives the best performance, before you deploy to your hardware.

Basic Workflow

To deploy your model to the target.

1. Set up your model for concurrent execution.

   For more information about configuring your model for concurrent execution, see “Configure Your Model for Concurrent Execution” on page 14-20. With these settings, Simulink partitions your model based on the sample time of blocks at the root level, with each sample time in your model corresponding to a partition, and all blocks of a single rate or sample time belonging to the same partition.

   If you want to specify how to partition your model, use explicit partitioning. With explicit partitioning, you must specify a target architecture, and then explicitly partition your model. For more information, see “Specify a Target Architecture” on page 14-21, and “Partition Your Model Using Explicit Partitioning” on page 14-26.
2. Generate code and deploy it to your target. You can choose to deploy onto multiple targets.

   - To build and deploy on a desktop target, see “Build on Desktop” on page 14-37.
   - To deploy onto embedded targets using Embedded Coder, see “Deployment” (Embedded Coder).
   - To build and deploy on a real-time target using Simulink Real-Time™, see “Standalone Operation” (Simulink Real-Time).
   - To deploy onto FPGAs using HDL Coder, see “Deployment” (HDL Coder).

   Note Deployment onto FPGAs is supported only for explicitly partitioned models.

3 Optimize your design. This step is optional, and includes iterating over the design of your model and mapping to get the best performance, based on your metrics. One way to evaluate your model is to profile it and get execution times.

Product	Information
Desktop target	“Profile and Evaluate Explicitly Partitioned Models on a Desktop” on page 14-38
Simulink Real-Time	“Execution Profiling for Real-Time Applications” (Simulink Real-Time)
Embedded Coder	“Code Execution Profiling” (Embedded Coder)
HDL Coder	“Speed and Area Optimization” (HDL Coder)

How Simulink Helps You to Overcome Challenges in Multicore Programming

Manually programming your application for concurrent execution poses challenges beyond the typical challenges with manual coding. With Simulink, you can overcome the challenges of portability across multiple architectures, efficiency of deployment for an architecture, and cyclic data dependencies between application components. For more information on these challenges, see “Challenges in Multicore Programming” on page 14-6.

Portability

Simulink enables you to determine the content and hierarchical needs of the modeled system without considering the target system. While creating model content, you do not need to keep track of the number of cores in your target system. Instead, you select the partitioning methods that enable you to create model content. Simulink generates code for the architecture you specify.

You can select an architecture from the available supported architectures or add a custom architecture. When you change your architecture, Simulink generates only the code that needs to change for the second architecture. The new architecture reuses blocks and functions. For more information, see “Supported Targets For Multicore Programming” on page 14-44 and “Specify a Target Architecture” on page 14-21.

Deployment Efficiency

To improve the performance of the deployed application, Simulink allows you to simulate it under typical computational loads and try multiple configurations of partitioning and mapping the application. Simulink compares the performance of each of these configurations to provide the optimal configuration for deployment. This is known as profiling. Profiling helps you to determine the optimum partition configuration before you deploy your system to the desired hardware.

You can create a mapping for your application in which Simulink maps the application components across different processing nodes. You can also manually assign components to processing nodes. For any mapping, you can see the data dependencies between components and remap accordingly. You can also introduce and remove data dependencies between different components.

Cyclic Data Dependency

Some tasks of a system depend on the output of other tasks. The data dependency between tasks determines their processing order. Two or more partitions containing data dependencies in a cycle

creates a data dependency loop, also known as an *algebraic loop*. Simulink does not allow algebraic loops to occur across potentially parallel partitions because of the high cost of solving the loop using parallel algorithms.

In some cases, the algebraic loop is artificial. For example, you can have an artificial algebraic loop because of Model-block-based partitioning. An algebraic loop involving Model blocks is artificial if removing the use of Model partitioning eliminates the loop. You can minimize the occurrence of artificial loops. In the Configuration Parameter dialog boxes for the models involved in the algebraic loop, select **Model Referencing > Minimize algebraic loop occurrences**.

Additionally, if the model is configured for the Generic Real-Time target (`grt.tlc`) or the Embedded Real-Time target (`ert.tlc`) in the Configuration Parameters dialog box, clear the **Single output/ update function** check box.

If the algebraic loop is a true algebraic condition, you must either contain all the blocks in the loop in one Model partition, or eliminate the loop by introducing a delay element in the loop.

The following examples show how to implement different types of parallelism in Simulink. These examples contain models that are partitioned and mapped to a simple architecture with one CPU and one FPGA.

See Also

Related Examples

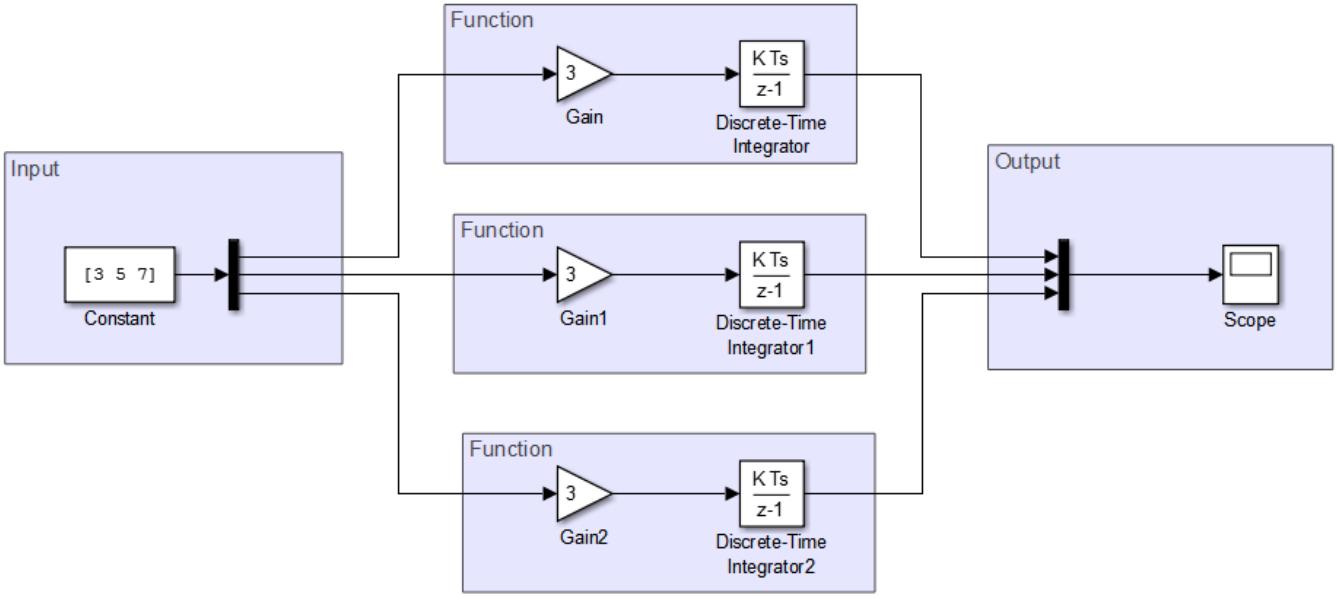
- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Task Parallelism in Simulink” on page 14-14
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Concepts in Multicore Programming” on page 14-2
- “Supported Targets For Multicore Programming” on page 14-44
- “Limitations with Multicore Programming in Simulink” on page 14-46

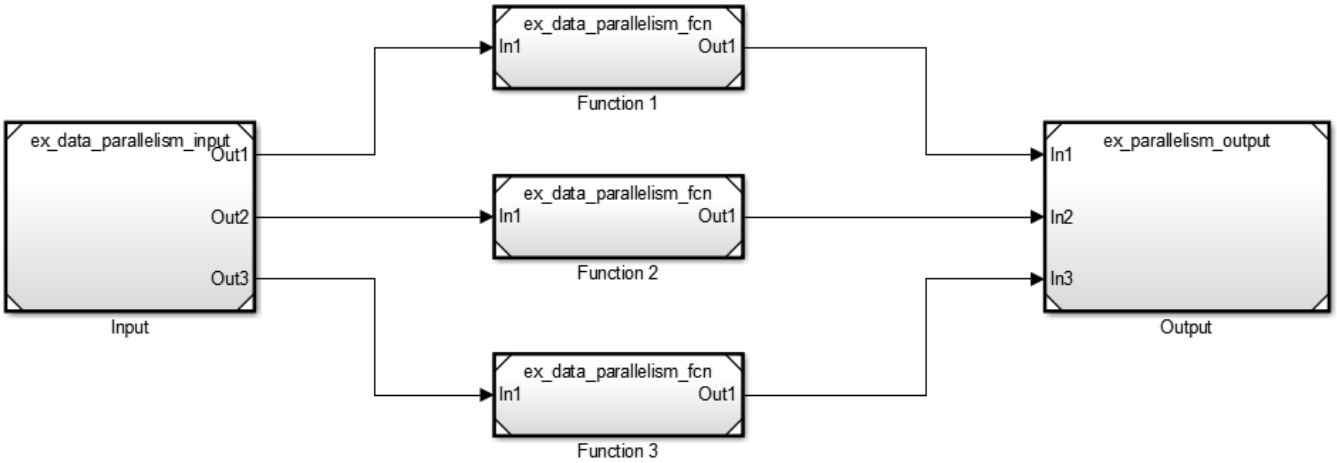
Implement Data Parallelism in Simulink

This example shows how to implement data parallelism for a system in a Simulink model. The model consists of an input, a functional component that applies to each input, and a concatenated output. For more information on data parallelism, see “Types of Parallelism” on page 14-2.



Set up this model for concurrent execution. To see the completed model, open ex_data_parallelism_top.

1. Convert areas in this model to referenced models. Use the same referenced model to replace each of the functional components that process the input. The figure shows a sample configuration.

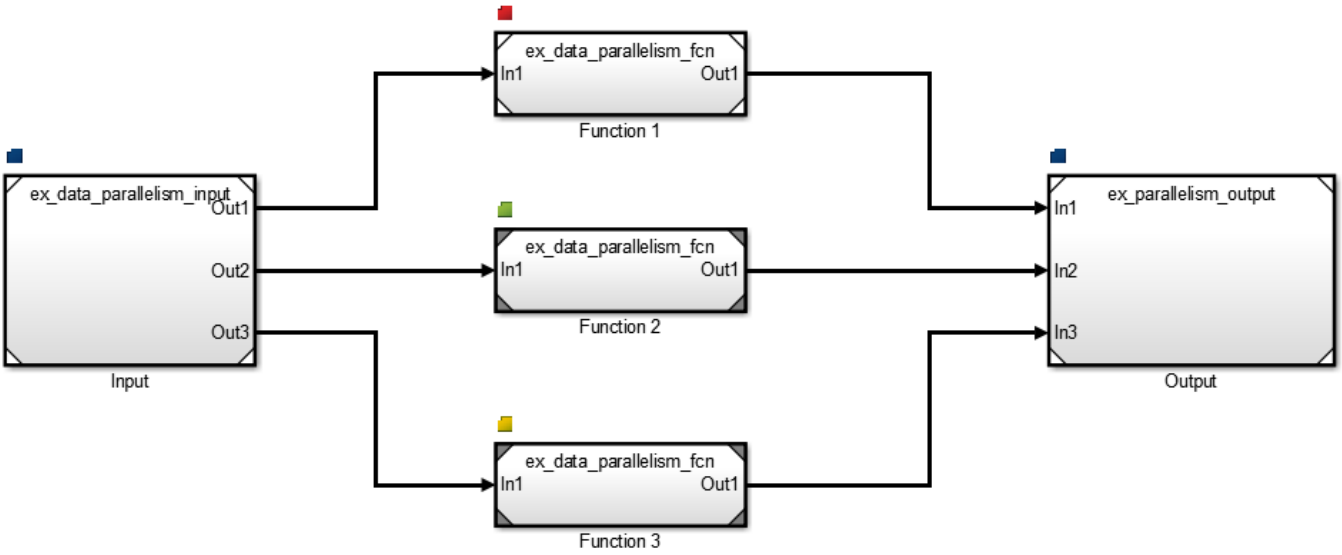


2. Open the model configuration parameters for the top level model. Clear the **MAT-file logging** check box.

3. On the **Solver** pane, set **Type** to `Fixed-step` and click **Apply**. Also ensure that the **Periodic sample time constraint** is set to `Unconstrained`. Under **Additional options**, select **Allow tasks to execute concurrently on target** and click **Configure Tasks**.
4. In the Concurrent Execution dialog box, in the right pane, select the **Enable explicit model partitioning for concurrent behavior** check box. With explicit partitioning, you can partition your model manually.
5. In the selection pane, select **CPU**. Click **Add task**  four times to add four new tasks.
6. In the selection pane, select **Tasks and Mapping**. On the **Map block to tasks** pane:
   - Under **Block: Input**, click `select task` and select `Periodic: Task`.
   - Under **Block: Function 1**, select `Periodic: Task1`.
   - Under **Block: Function 2**, select `Periodic: Task2`.
   - Under **Block: Function 3**, select `Periodic: Task3`.
   - Under **Block: Output**, select `Periodic: Task`.

   This maps your partitions to the tasks you created. The Input and Output model blocks are on one task. Each functional component is assigned a separate task.
7. In the selection pane, select **Data transfer**. In the **Data Transfer Options** pane, set the parameter **Periodic signals** to `Ensure deterministic transfer (minimum delay)`. Click **Apply** and close the Concurrent Execution dialog box.
8. Apply these configuration parameters to all referenced models. For more information, see “Share a Configuration with Multiple Models” on page 13-10.

Update your model to see the tasks mapped to individual model blocks.



See Also

Related Examples

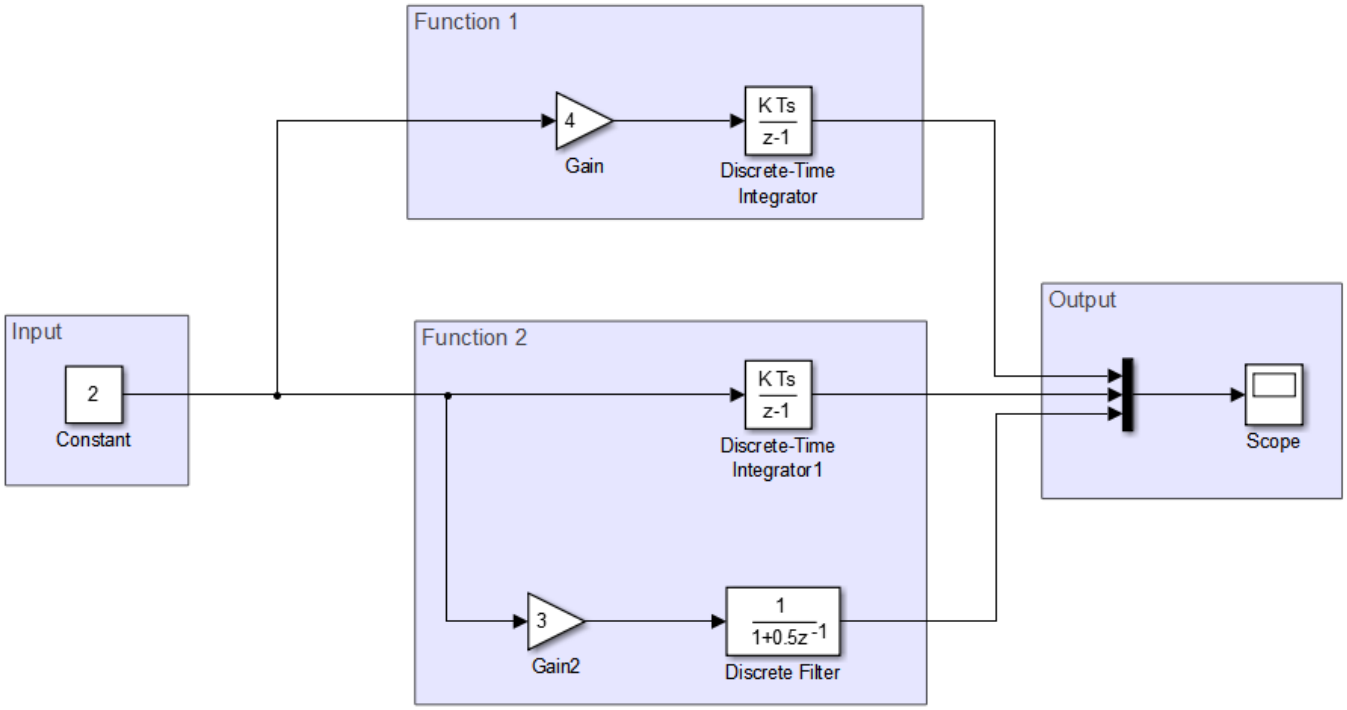
- “Implement Task Parallelism in Simulink” on page 14-14
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44
- “Limitations with Multicore Programming in Simulink” on page 14-46

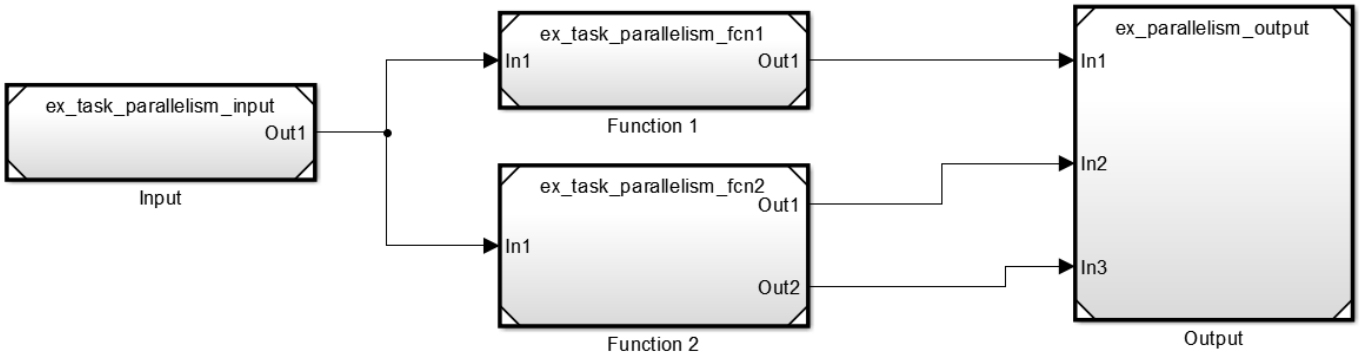
Implement Task Parallelism in Simulink

This example shows how to implement task parallelism for a system in a Simulink model. The model consists of an input, functional components applied to the same input, and a concatenated output. For more information on Task Parallelism, see “Types of Parallelism” on page 14-2.



Set up the model for concurrent execution. To see the completed model, open ex_task_parallelism_top.

1. Convert areas in this model to referenced models. Use the same referenced model to replace each of the functional components that process the input. The figure shows a sample configuration.

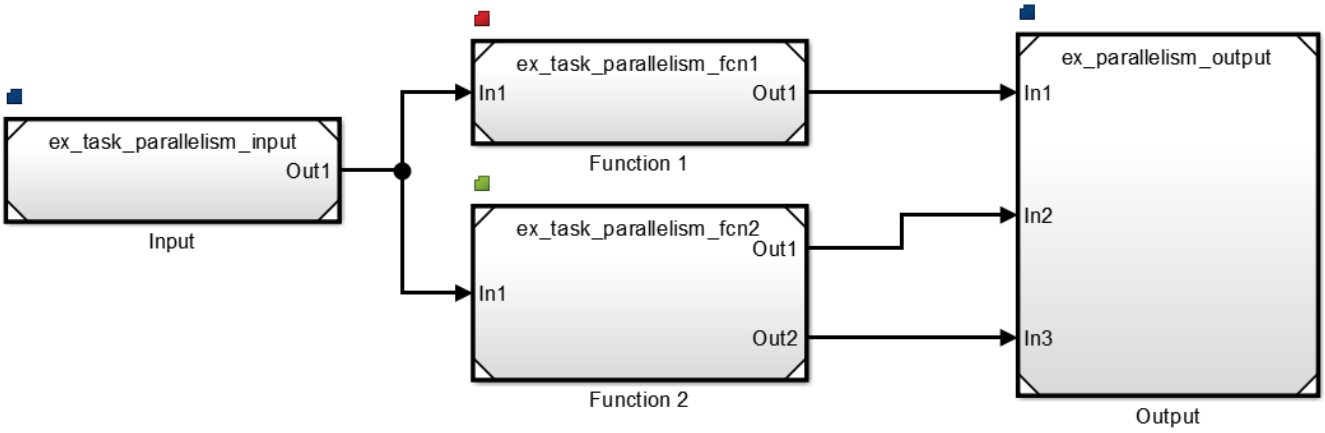


2. Open the model configuration parameters for the top level model. Clear the **MAT-file logging** check box.

3. On the **Solver** pane, set **Type** to `Fixed-step` and click **Apply**. Also ensure that the **Periodic sample time constraint** is set to `Unconstrained`. Under **Additional options**, select **Allow tasks to execute concurrently on target** and click **Configure Tasks**.
4. In the Concurrent Execution dialog box, in the right pane, select the **Enable explicit model partitioning for concurrent behavior** check box. With explicit partitioning, you can partition your model manually.
5. In the selection pane, select **CPU**. Click **Add task**  three times to add new tasks.
6. In the selection pane, select **Tasks and Mapping**. To map partitions to the tasks you created, on the **Map block to tasks** pane:
   - Under **Block: Input**, click `select task` and select `Periodic: Task`.
   - Under **Block: Function 1**, select `Periodic: Task1`.
   - Under **Block: Function 2**, select `Periodic: Task2`.
   - Under **Block: Output**, select `Periodic: Task`.

   The Input and Output model blocks are on one task. Each functional component is assigned a separate task.
7. In the selection pane, select **Data transfer**. In the **Data Transfer Options** pane, set the parameter **Periodic signals** to `Ensure deterministic transfer (minimum delay)`. Click **Apply** and close the Concurrent Execution dialog box.
8. Apply these configuration parameters to all referenced models. For more information, see “Share a Configuration with Multiple Models” on page 13-10.

Update your model to see the tasks mapped to individual model blocks.



See Also

Related Examples

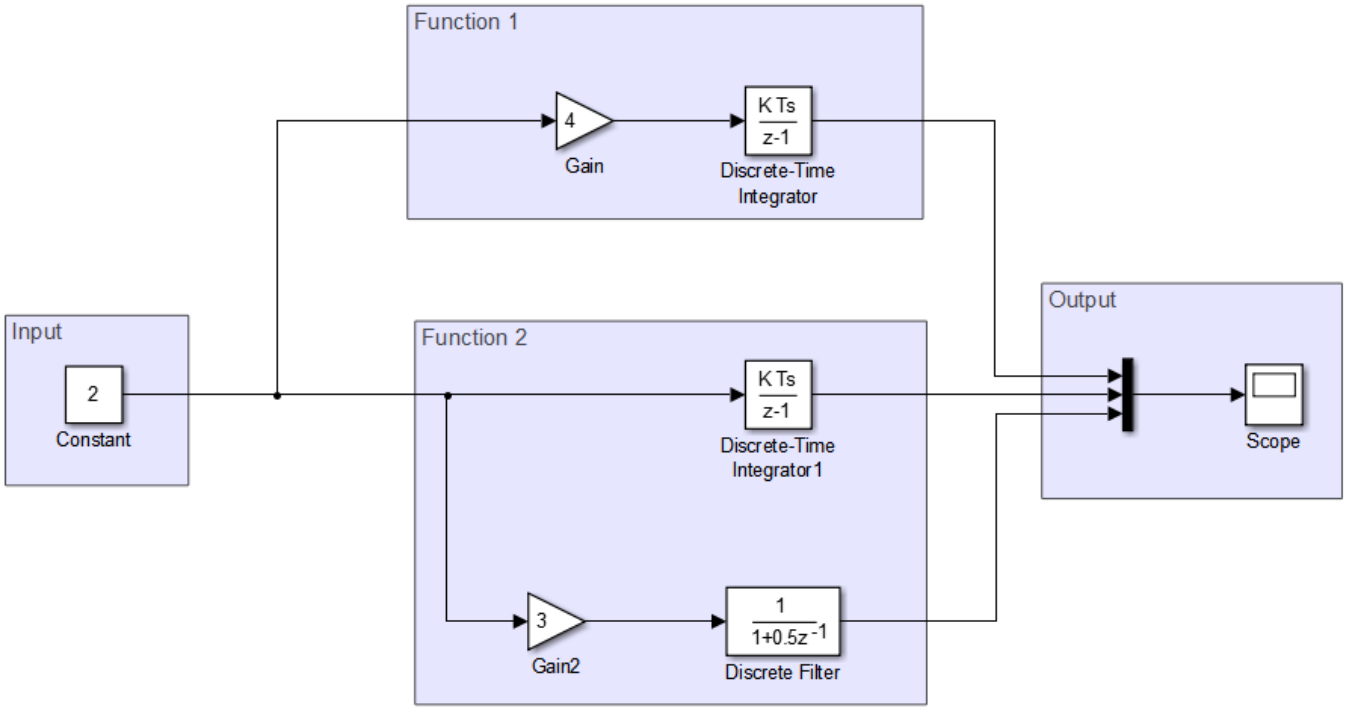
- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44

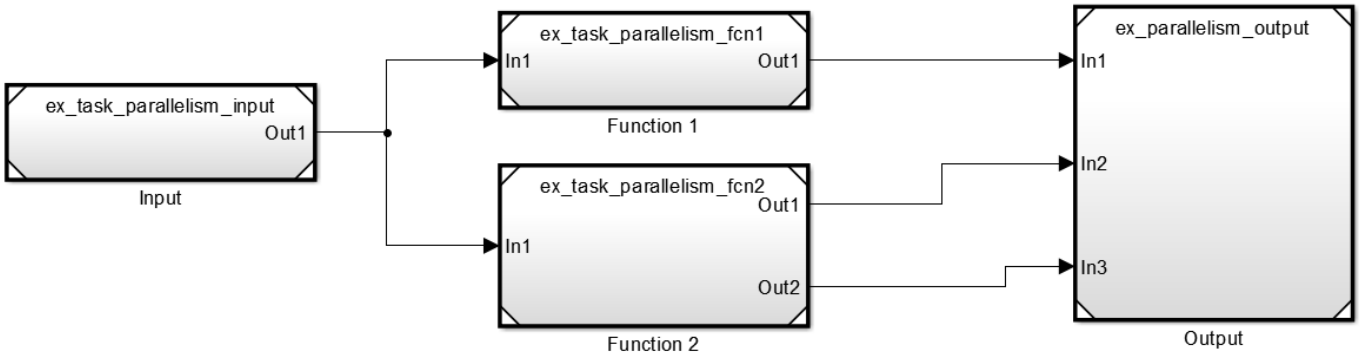
Implement Pipelining in Simulink

This example shows how to implement pipelining for a system in a Simulink model. The model consists of an input, functional components applied to the same input, and a concatenated output. For more information on pipelining, see “Types of Parallelism” on page 14-2.



Setup this model for concurrent execution. To see the completed model, open ex_pipelining_top.

1 Convert areas in this model to referenced models. Use the same referenced model to replace each of the functional components that process the input. The figure shows a sample configuration.

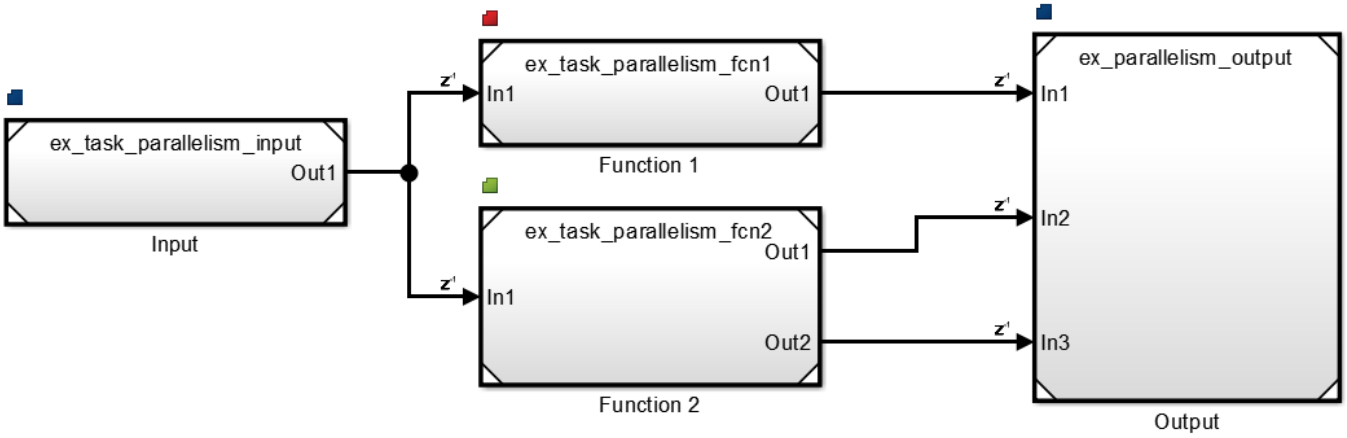


2 Open the model configuration parameters for the top level model. Clear the **MAT-file logging** check box.

3. On the **Solver** pane, set **Type** to `Fixed-step` and click **Apply**. Also ensure that the **Periodic sample time constraint** is set to `Unconstrained`. Under **Additional options**, select **Allow tasks to execute concurrently on target** and click **Configure Tasks**.
4. In the Concurrent Execution dialog box, in the right pane, select the **Enable explicit model partitioning for concurrent behavior** check box. With explicit partitioning, you can partition your model manually.
5. In the selection pane, select **CPU**. Click **Add task**  three times to add three new tasks.
6. In the selection pane, select **Tasks and Mapping**. On the **Map block to tasks** pane:
   - Under **Block: Input**, click `select task` and select `Periodic: Task`.
   - Under **Block: Function 1**, select `Periodic: Task1`.
   - Under **Block: Function 2**, select `Periodic: Task2`.
   - Under **Block: Output**, select `Periodic: Task`.

   This maps your partitions to the tasks you created. The Input and Output model blocks are on one task. Each functional component is assigned a separate task.
7. Close the Concurrent Execution dialog box.
8. Apply these configuration parameters to all referenced models. For more information, see “Share a Configuration with Multiple Models” on page 13-10.

Update your model to see the tasks mapped to individual model blocks.



Note Notice that delays are introduced between different tasks, indicated by the z-1 badge. Introducing these delays may cause different model outputs in Simulink. Ensure that your model has an expected output on simulating the parallelized model.

See Also

Related Examples

- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Task Parallelism in Simulink” on page 14-14

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44

Configure Your Model for Concurrent Execution

Follow these steps to configure your Simulink model to take advantage of concurrent execution.

1. Open your model.
2. On the **Modeling** tab, click **Model Settings**.
3. Select **Solver**, then in the **Solver selection** section , choose `Fixed-step` for the **Type** and `auto (Automatic solver selection)` for the **Solver**.
4. Under **Solver details**, select **Allow tasks to execute concurrently on target**. Selecting this check box is optional for models referenced in the model hierarchy. When you select this option for a referenced model, Simulink allows each rate in the referenced model to execute as an independent concurrent task on the target processor.
5. Select **Code Generation > Interface > Advanced parameters**, clear the **MAT-file logging** check box.

Once you have a model that executes concurrently on your computer, you can further configure your model in the following ways:

- “Specify a Target Architecture” on page 14-21
- “Partition Your Model Using Explicit Partitioning” on page 14-26.
- “Configure Data Transfer Settings Between Concurrent Tasks” on page 14-33

See Also

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Implicit and Explicit Partitioning of Models” on page 14-31
- “Specify a Target Architecture” on page 14-21
- “Partition Your Model Using Explicit Partitioning” on page 14-26
- “Configure Data Transfer Settings Between Concurrent Tasks” on page 14-33

Specify a Target Architecture

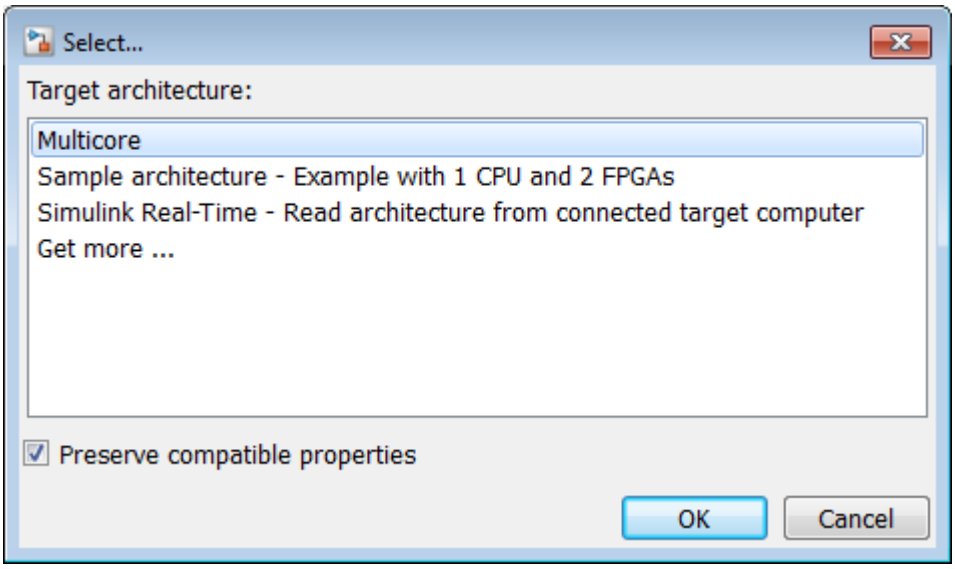
In this section...

For models configured for concurrent execution, you can choose the architecture to which you want to deploy your model. Choose from a set of predefined architectures in Simulink, or you can create an interface for a custom architecture. After selecting your architecture, you can use explicit partitioning to specify which tasks run on it. For more information, see “Partition Your Model Using Explicit Partitioning” on page 14-26.

Choose from Predefined Architectures

You can choose from the predefined architectures available in Simulink, or you can download support packages for different available architectures.

1 In the Concurrent Execution dialog box, in the **Concurrent Execution** pane, click **Select**. The concurrent execution target architecture selector appears.



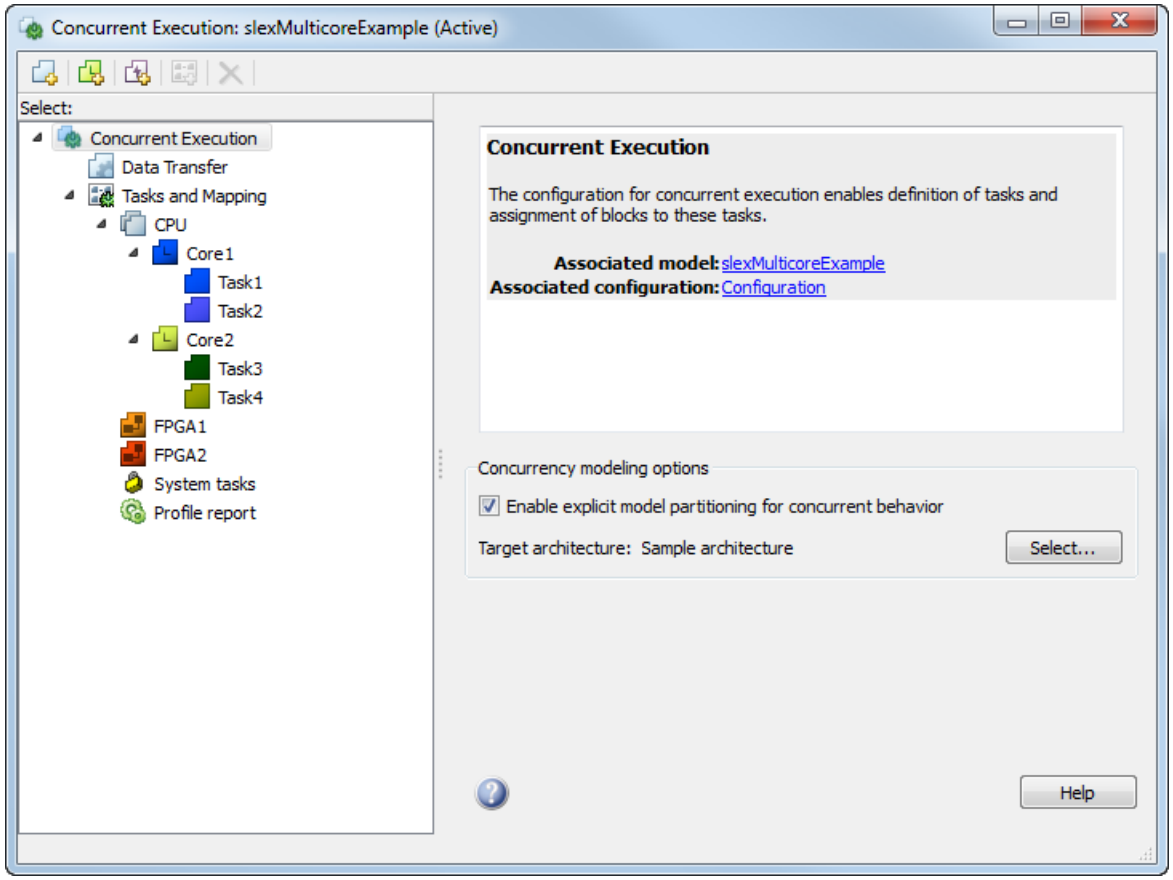
2 Select your target.

Property	Description
`Multicore`	Single CPU with multiple cores.
`Sample Architecture`	Single CPU with multiple cores and two FPGAs.
`Simulink Real-Time`	Simulink Real-Time target.
`Get more ...`	Click **OK** to start the Support Package Installer. From the list, select the target and follow the instructions.

3 In the Target architecture window, clear the **Preserve compatible properties** check box to reset existing target property settings to their default. Alternatively, select the **Preserve compatible properties** check box to preserve existing target property settings.

4 Click **OK**.

Simulink adds the corresponding software and hardware nodes to the configuration tree hierarchy. For example, the following illustrates one software node and two hardware nodes added to the configuration tree when you select `Sample architecture` as the target architecture.



Define a Custom Architecture File

A custom architecture file is an XML file that allows you to define custom target properties for tasks and triggers. For example, you may want to define custom properties to represent threading APIs. Threading APIs are necessary to take advantage of concurrency on your target processor.

The following is an example custom architecture file:

```
<architecture brief="Multicore with custom threading API"
              format="1.1"  revision="1.1"
              uuid="MulticoreCustomAPI"  name="MulticoreCustomAPI">
<configurationSet>
     <parameter name="SystemTargetFile" value="ert.tlc"/>
     <parameter name="SystemTargetFile" value="grt.tlc"/>
</configurationSet>
<node name="MulticoreProcessor" type="SoftwareNode" uuid="MulticoreProcessor"/>
```

```
<template name="CustomTask" type="Task" uuid="CustomTask">
     <property name="affinity" prompt="Affinity:" value="1" evaluate="true"/>
     <property name="schedulingPolicy" prompt="Scheduling policy:" value="Rate-monotonic">
           <allowedValue>Rate-monotonic</allowedValue>
           <allowedValue>Round-robin</allowedValue>
        </property>
   </template>
</architecture>
```

An architecture file must contain:

- The architecture element that defines basic information used by Simulink to identify the architecture.
- A `configurationSet` element that lists the system target files for which this architecture is valid.
- One node element that Simulink uses to identify the multicore processing element.

  Note The architecture must contain exactly one node element that identifies a multicore processing element. You cannot create multiple nodes identifying multiple processing elements or an architecture with no multicore processing element.

- One or more template elements that list custom properties for tasks and triggers.
  - The type attribute can be `Task`, `PeriodicTrigger`, or `AperiodicTrigger`.
  - Each property is editable and has the default value specified in the `value` attribute.
  - Each property can be a text box, check box, or combo box. A check box is one where you can set the `value` attribute to `on` or `off`. A combo box is one where you can optionally list `allowedValue` elements as part of the property.
  - Each text box property can also optionally define an `evaluate` attribute. This lets you place MATLAB variable names as the value of the property.

  Assuming that you have saved the custom target architecture file in `C:\custom_arch.xml`, register this file with Simulink using the `Simulink.architecture.register` function.

For example:

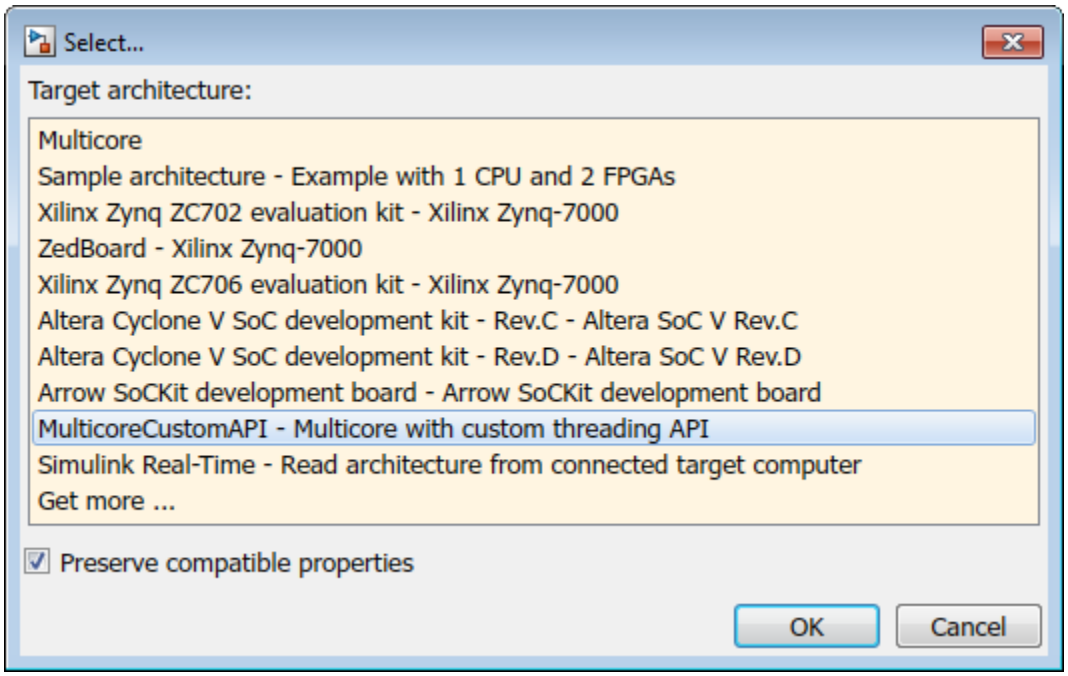
1. Save the contents of the listed XML file in `custom_arch.xml`.
2. In the MATLAB Command Window, type:

   ```
   Simulink.architecture.register('custom_arch.xml')
   ```

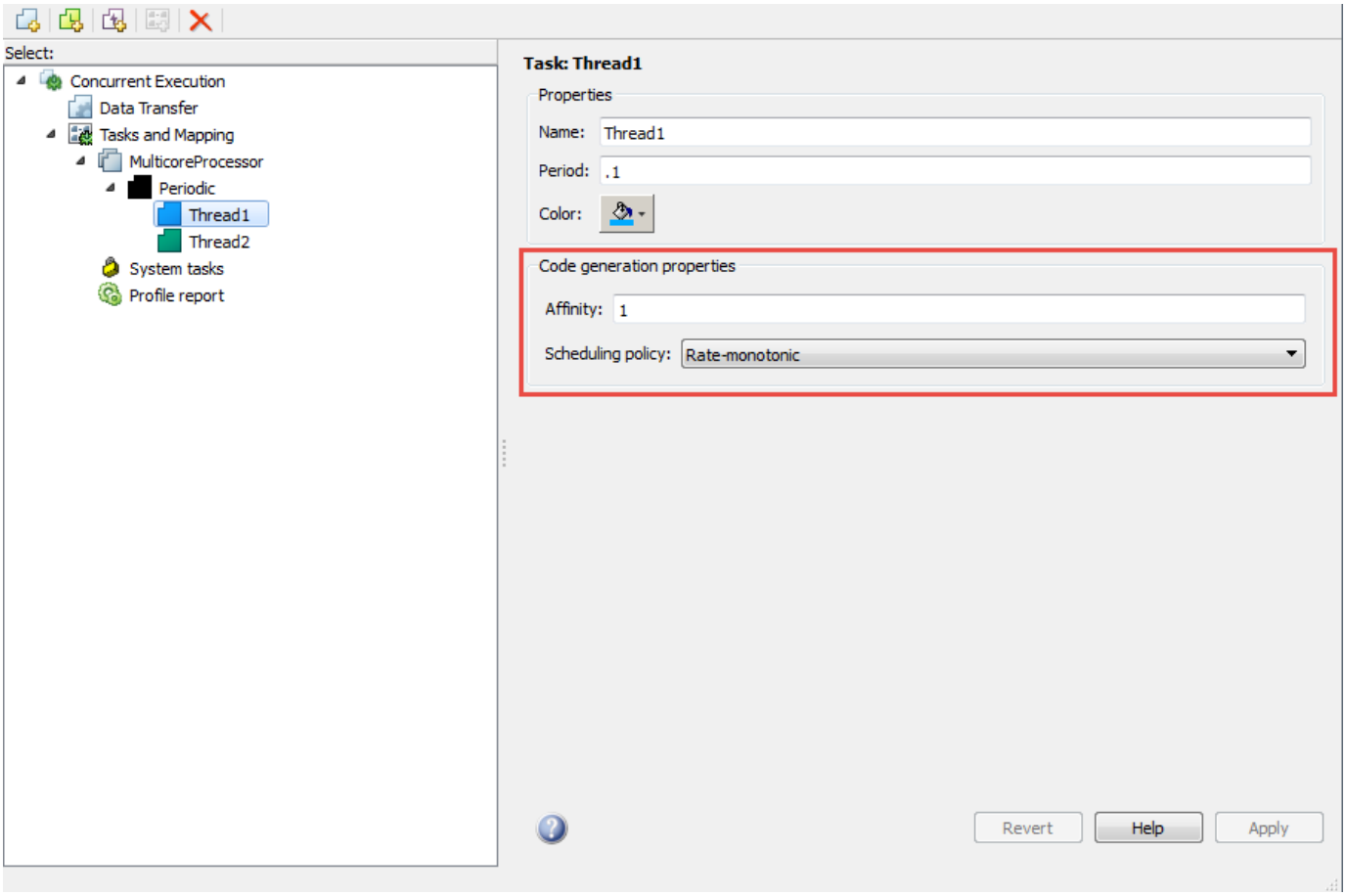
3. In the MATLAB Command Window, type:

   ```
   slexMulticoreSolverExample
   ```

4. In the Simulink editor, open the **Configuration Parameters** > **Solver** pane and click **Configure Tasks**. The Concurrent Execution dialog box displays.
5. In the **Concurrent Execution** pane, click **Select...** under **Target Architecture**. The Target architecture window displays.
6. Select `MulticoreCustomAPI` and click **OK**.



Your Concurrent Execution dialog box updates to contain Code Generation properties for the tasks as shown. These are the properties defined in the XML file.



See Also

More About

- “Configure Your Model for Concurrent Execution” on page 14-20
- “Partition Your Model Using Explicit Partitioning” on page 14-26
- “Implicit and Explicit Partitioning of Models” on page 14-31

Partition Your Model Using Explicit Partitioning

When you have a model that is configured for concurrent execution, you can add tasks, create partitions, and map individual tasks to partitions using explicit partitioning. This enables you to execute different parts of your model to different parts of your architecture. For more information, see “Implicit and Explicit Partitioning of Models” on page 14-31.

Prerequisites for Explicit Partitioning

To use explicit partitioning, you must meet the following prerequisites:

1. Set up your model for concurrent execution. For more information, see “Configure Your Model for Concurrent Execution” on page 14-20.
2. Convert all blocks at the root level of your model into one of the following types of blocks.

   - Models that are referenced using Model blocks
   - Subsystem blocks
   - MATLAB System blocks
   - MATLAB Function blocks
   - Stateflow charts

   For more information, see “Implicit and Explicit Partitioning of Models” on page 14-31.

   Note When using referenced models, replicate the model configuration parameters of the top model to the referenced models. Consider using a single configuration reference to use for all of your referenced models. For more information, see “Model Configuration Sets”.
3. Select the target architecture on which to deploy your model. For more information, see “Specify a Target Architecture” on page 14-21.

Add Periodic Triggers and Tasks

Add periodic tasks for components in your model that you want to execute periodically. To add aperiodic tasks whose execution is trigger based, see “Add Aperiodic Triggers and Tasks” on page 14-27.

If you want to explore the effects of increasing the concurrency on your model execution, you can create additional periodic tasks in your model.

1. In the Concurrent Execution dialog box, right-click the **Periodic** node and select **Add task**.

   A task node appears in the Configuration Execution hierarchy.
2. Select the task node and enter a name and period for the task, then click **Apply**.

   The task node is renamed to the name you enter.
3. Optionally, specify a color for the task. The color illustrates the block-to-task mapping. If you do not assign a color, Simulink chooses a default color. If you enable sample time colors for your model, the software honors the setting.
4. Click **Apply** as necessary.

To create more periodic triggers, click the **Add periodic trigger** symbol. You can also create multiple periodic triggers with their own trigger sources.

Note Periodic triggers let you represent multiple periodic interrupt sources such as multiple timers. The periodicity of the trigger is either the base rate of the tasks that the trigger schedules, or the period of the trigger. Data transfers between triggers can only be `Ensure Data Integrity Only` types. With blocks mapped to periodic triggers, you can only generate code for `ert.tlc` and `grt.tlc` system target files.

To delete tasks and triggers, right-click them in the pane and select **Delete**.

When the periodic tasks and trigger configurations are complete, configure the aperiodic (interrupt) tasks as necessary. If you do not need aperiodic tasks, continue to “Map Blocks to Tasks, Triggers, and Nodes” on page 14-28.

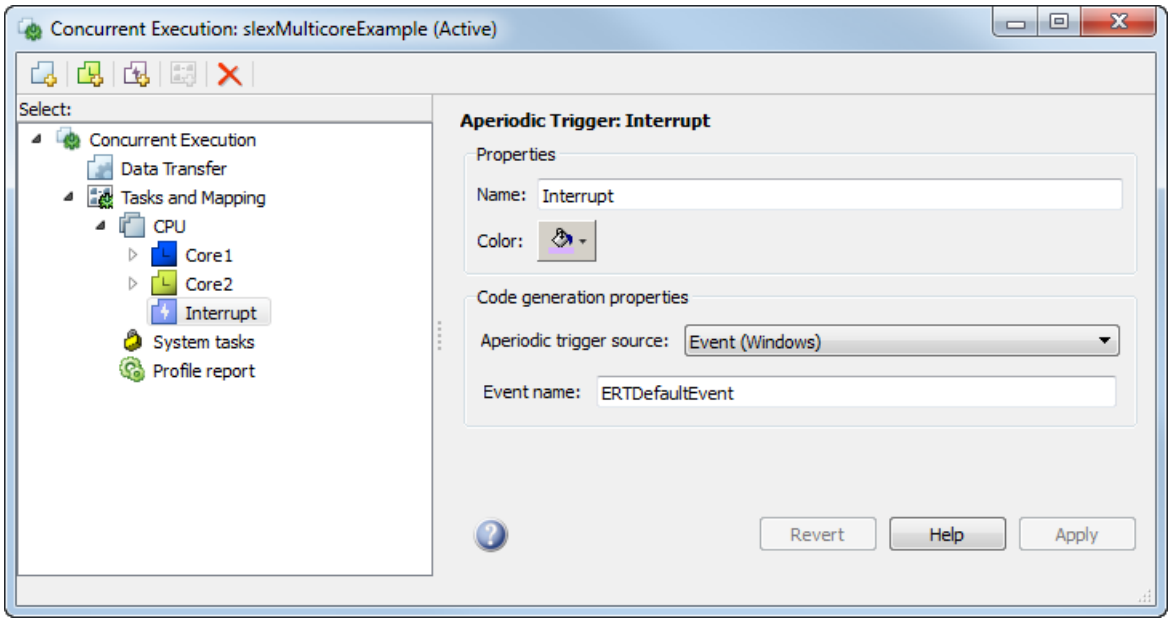
Add Aperiodic Triggers and Tasks

Add aperiodic tasks for components in your model whose execution is interrupt-based. To add periodic tasks whose execution is periodic, see “Add Periodic Triggers and Tasks” on page 14-26.

1. To create an aperiodic trigger, in the Concurrent Execution dialog box, right-click the **Concurrent Execution** node and click the **Add aperiodic trigger** symbol.

   A node named **Interrupt***N* appears in the configuration tree hierarchy, where *N* is an integer.
2. Select **Interrupt**.

   This node represents an aperiodic trigger for your system.
3. Specify the name of the trigger and configure the aperiodic trigger source. Depending on your deployment target, choose either `Posix Signal (Linux/VxWorks 6.x)` or `Event (Windows)`. For POSIX® signals, specify the signal number to use for delivering the aperiodic event. For Windows events, specify the name of the event.



4 Click **Apply**.

The software services aperiodic triggers as soon as possible. If you want to process the trigger response using a task:

1 Right-click the **Interrupt** node and select **Add task**.

A new task node appears under the **Interrupt** node.

2 Specify the name of the new task node.

3 Optionally, specify a color for the task. The color illustrates the block-to-task mapping. If you do not assign a color, Simulink chooses a default color.

4 Click **Apply**.

To delete tasks and triggers, right-click them in the pane and select **Delete**.

Once you have created your tasks and triggers, map your execution components to these tasks. For more information, see “Map Blocks to Tasks, Triggers, and Nodes” on page 14-28.

Map Blocks to Tasks, Triggers, and Nodes

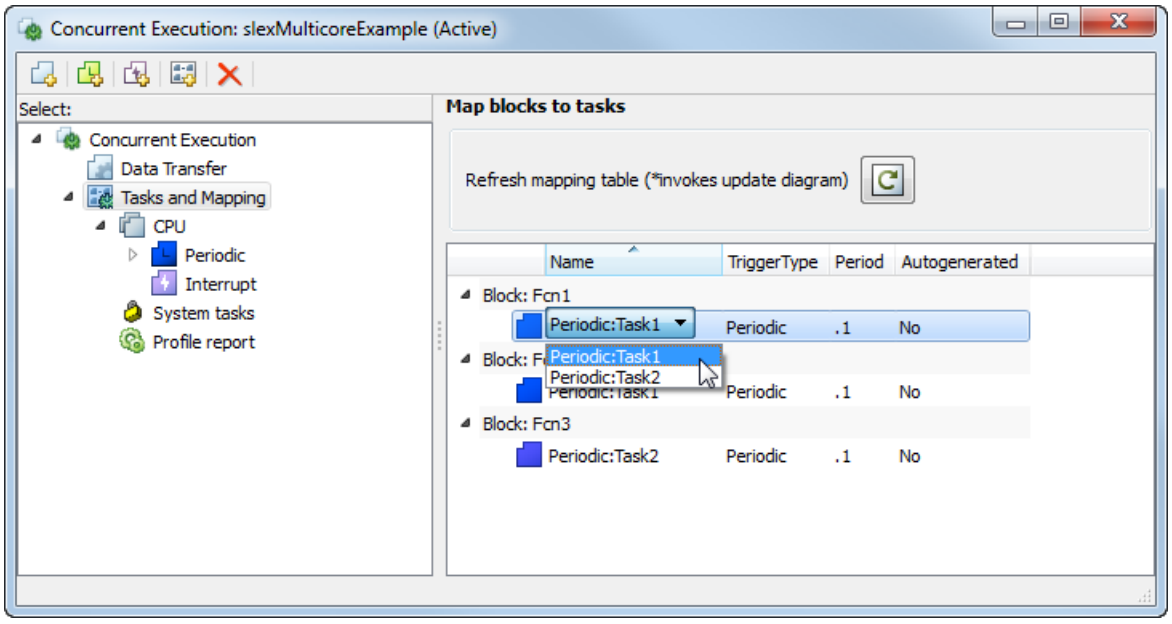
After you create the tasks and triggers, you can explicitly assign partitions to these execution elements.

1 In the Concurrent Execution dialog box, click the **Tasks and Mapping** node.

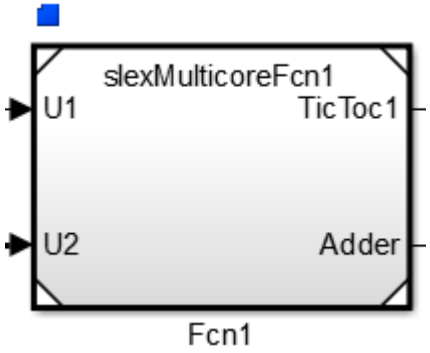
The **Tasks and Mapping** pane appears. If you add a Model block to your model, the new block appears in the table with a **select task** entry under it.

2 If you want to add a task to a block, in the **Name** column, right-click a task under the block and select **Add new entry**.

3 To assign a task for the entry, click the box in the **Name** column and select an entry from the list. For example:



The block-to-task mapping symbol appears on the top-left corner of the Model block. For example:



If you assign a Model block to multiple tasks, multiple task symbols are displayed in the top-left corner.

To display the Concurrent Execution dialog box from the block, click the block-to-task mapping symbol.

4 Click **Apply**.

Note

- System tasks allow you to perform mapping incrementally. This means that if there is only one periodic trigger, Simulink assigns any Model blocks, subsystem blocks, or MATLAB System blocks that you have not explicitly mapped to a task, trigger, or hardware node to a task created by the system. Simulink creates at most one system task for each rate in the model. If there are multiple periodic triggers created, explicitly map the Model block partitions, subsystems, or MATLAB System blocks to a task, trigger, or hardware node.
- Map Model block partitions that contain continuous blocks to the same periodic trigger.
- You can map only Model blocks to hardware nodes. Also, if you map the Model block to a hardware node, and the Model block contains multiple periodic sample times, clear the **Allow tasks to execute concurrently on target** check box in the **Solver** pane of the Configuration Parameters dialog box.

When the mapping is complete, simulate the model again.

See Also

Related Examples

- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Task Parallelism in Simulink” on page 14-14
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Implicit and Explicit Partitioning of Models” on page 14-31

Implicit and Explicit Partitioning of Models

When implementing multicore programming for your application in Simulink, there are two ways to partition your model for running on individual processing nodes. If you are new to multicore programming in Simulink, use the default (implicit partitioning) for your first iteration of implementing multicore programming.

The automated way of creating tasks and mapping them to your processing nodes is called implicit partitioning. Simulink partitions your model based on the sample time of blocks at the root level. Each sample time in your model corresponds to a partition, and all blocks of a single rate or sample time belong to the same partition. Simulink maps these partitions to tasks that run on your processor. Implicit partitioning assumes your architecture to be a single multicore CPU. The CPU task scheduler handles all the partitioned tasks.

If you want to specify how to partition your model, use explicit partitioning. In explicit partitioning, you create partitions in the root-level model by using referenced models, MATLAB system blocks, MATLAB Function blocks, Stateflow charts, and Simulink subsystems. For example, if your model has data acquisition and a controller, partition your model by putting these components in two referenced models at the model root-level. Each sample time of the blocks in the model corresponds to a partition. You can add tasks to run on processing nodes in the Concurrent Execution dialog box and assign your partitions to these tasks. If some partitions are left unassigned, Simulink automatically assigns them to tasks.

In explicit partitioning, you can specify your own architecture. The default architecture is a multicore CPU, the same as the assumed architecture in implicit partitioning. Explicit partitioning has more restrictions on your root-level model than implicit partitioning. For more information, see “Limitations with Multicore Programming in Simulink” on page 14-46.

Partitioning Guidelines

There are multiple ways to partition your model for concurrent execution in Simulink. Rate-based and model-based approaches give you primarily graphical means to represent concurrency for systems that are represented using Simulink and Stateflow blocks. You can partition MATLAB code using the MATLAB System block and the MATLAB Function block. You can also partition models of physical systems using multisolver methods.

Each method has additional considerations to help you decide which to use.

Goal	Valid Partitioning Methods	Considerations
Increase the performance of a simulation on the host computer.	No partitioning method	Simulink tries to optimize the host computer performance regardless of the modeling method you use. For more information on the ways that Simulink helps you to improve performance, see “Optimize Performance”.

Goal	Valid Partitioning Methods	Considerations
Increase the performance of a plant simulation in a multicore HIL (hardware-in-the-loop) system.	You can use any of the partitioning methods and their combinations.	The processing characteristics of the HIL system and the embedded processing system can vary greatly. Consider partitioning your system into more units of work than there are number of processing elements in the HIL or embedded system. This convention allows flexibility in the mapping process.
Create a valid model of a multirate concurrent system to take advantage of a multicore processing system.	You can use any of the partitioning methods and their combinations.	Partitioning can introduce signal delays to represent the data transfer requirements for concurrent execution. For more information, see “Configure Data Transfer Settings Between Concurrent Tasks” on page 14-33.
Create a valid model of a heterogeneous system to take advantage of multicore and FPGA processing.	• Multicore processing: Use any of the partitioning methods. • FPGA processing: Partition your model using Model blocks.	Consider partitioning for FPGA processing where your computations have bottlenecks that could be reduced using fine-grained hardware parallelism.

See Also

Related Examples

- “Implement Data Parallelism in Simulink” on page 14-11
- “Implement Task Parallelism in Simulink” on page 14-14
- “Implement Pipelining in Simulink” on page 14-17

More About

- “Concepts in Multicore Programming” on page 14-2
- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44

Configure Data Transfer Settings Between Concurrent Tasks

Data dependencies arise when a signal originates from one block in one partition and is connected to a block in another partition. To create opportunities for parallelism, Simulink provides multiple options for handling data transfers between concurrently executing partitions. These options help you trade off computational latency for numerical signal delays.

Use the **Data Transfer Options** pane to define communications between tasks. Set the values for the `Use global setting` option of the Data Transfer tab in the Signal Properties dialog box. The table provides the model-level options that you can apply to each signal that requires data transfer in the system.

Data Transfer Options

Goal	Data Transfer Type	Simulation	Deployment
• Create opportunity for parallelism. • Reduce signal latency.	Ensure data integrity only.	Data transfer is simulated using a one-step delay.	Simulink Coder ensures data integrity during data transfer. Simulink generates code to operate with maximum responsiveness and data integrity. However, the implementation is interruptible, which can lead to loss of data during data transfer. Use a deterministic execution schedule to achieve determinism in the deployment environment.
• Create opportunity for parallelism. • Produce numeric results that are repeatable with each run of the generated code.	Ensure deterministic transfer (maximum delay).	Data transfer is simulated using a one-step delay, which can have impact on the numeric results. To compensate, you might need to specify an initial condition for these delay elements.	Simulink Coder ensures data integrity during data transfer. In addition, Simulink Coder ensures that data transfer is identical with simulation.
• Enforce data dependency. • Produce numeric results that are repeatable with each run of the generated code.	Ensure deterministic transfer (minimum delay).	Data transfer occurs within the same step.	

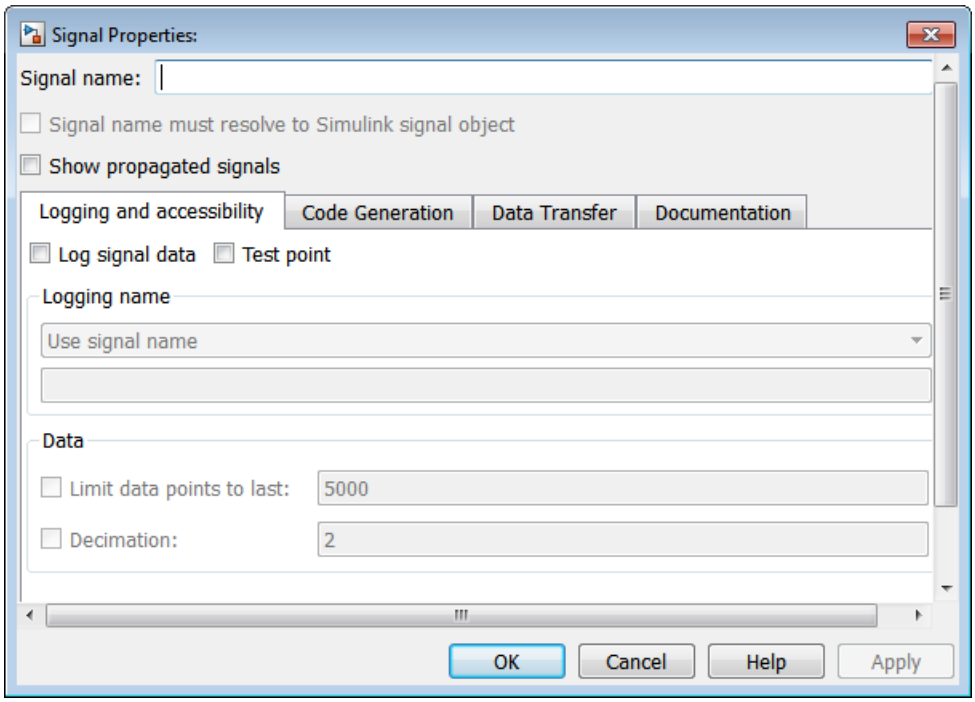
You can also override these options for each signal from the **Data Transfer** pane of the Signal Properties dialog box. For more information, see “Data Transfer Options for Concurrent Execution”.

For example, consider a control application in which a controller that reads sensory data at time T must produce the control signals to the actuator at time $T+\Delta$.

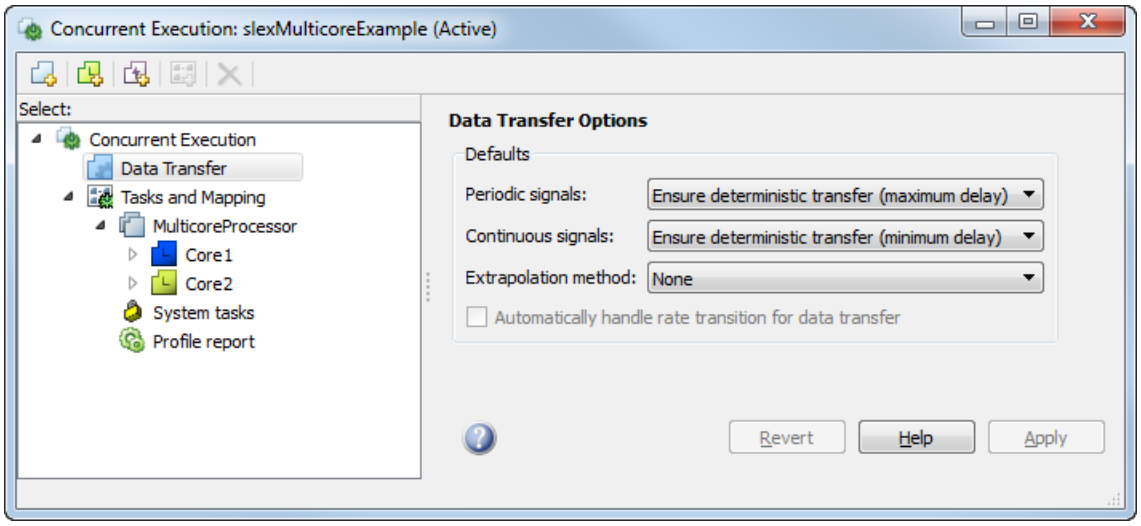
- If the sequential algorithm meets the timing deadlines, consider using option 3.
- If the embedded system provides deterministic scheduling, consider using option 2.
- Otherwise, use option 1 to create opportunities for parallelism by introducing signal delays.

For continuous signals, Simulink uses extrapolation methods to compensate for numerical errors that were introduced due to delays and discontinuities in data transfer.

To avoid numerical errors in signals configured for `Ensure Data Integrity Only` and `Ensure deterministic transfer (maximum delay)` data transfers, you may need to provide an initial condition. You can specify this initial condition in the **Data Transfer** tab of the Signal Properties dialog box. To access this dialog box, right-click the signal line and select **Properties** from the context menu. A dialog box like the following is displayed.



1. From the Data Transfer Options table, determine how you want your tasks to communicate.
2. In the Concurrent Execution dialog box, select Data Transfer defaults and apply the settings from step 1.



3 Apply your changes.

See Also

More About

- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44
- “Implicit and Explicit Partitioning of Models” on page 14-31

Optimize and Deploy on a Multicore Target

In this section...

This topic shows how to use a model that is configured for concurrent execution using explicit partitioning and deploy it onto a target. To set up your model for concurrent execution, see “Configure Your Model for Concurrent Execution” on page 14-20. To specify the target architecture, see “Specify a Target Architecture” on page 14-21. To use explicit partitioning in a model that is set up for concurrent execution, see “Partition Your Model Using Explicit Partitioning” on page 14-26.

Generate Code

To generate code for a model that is configured for concurrent execution, on the **Apps** tab of the Simulink editor, select **Simulink Coder**. On the **C Code** tab, select **Build**. The resulting code includes:

- C code for parts of the model that are mapped to tasks and triggers in the Concurrent Execution dialog box. C code generation requires a Simulink Coder license. For more information, see “Code Generation” (Simulink Coder) and “Code Generation from Simulink Models” (Embedded Coder).
- HDL code for parts of the model that are mapped to hardware nodes in the Concurrent Execution dialog box. HDL code generation requires an HDL Coder license. For more information, see “HDL Code Generation from Simulink” (HDL Coder).
- Code to handle data transfer between the concurrent tasks and triggers and to interface with the hardware and software components.

The generated C code contains one function for each task or trigger defined in the system. The task and trigger determines the name of the function:

```
void <TriggerName>_TaskName(void);
```

The content for each such function consists of target-independent C code, except for:

- Code corresponding to blocks that implement target-specific functionality
- Customizations, including those derived from custom storage classes (see “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder)) or “Code Replacement Libraries” (Simulink Coder)
- Code that is generated to handle how data is transferred between tasks. In particular, Simulink Coder uses target-specific implementations of mutual exclusion primitives and data synchronization semaphores to implement the data transfer as described in the following table of pseudocode.

Data Transfer	Initialization	Reader	Writer

Data Integrity Only	`BufferIndex = 0;`
	`Initialize Buffer[1] with IC`
		`Begin mutual exclusion`
		`  Tmp = 1 - BufferIndex;`
		`End mutual exclusiton`
		`  Read Buffer[ Tmp ];`
			`Write Buffer[ BufferIndex ];`
			`Begin mutual exclusion`
			`BufferIndex = 1 - BufferIndex;`
			`End mutual exclusion`

Data Transfer	Initialization	Reader	Writer

Ensure Determinism (Maximum Delay)	`WriterIndex = 0;`
	`ReaderIndex = 1;`
	`Initialize Buffer[1] with IC`
		`Read Buffer[ ReaderIndex ];`
		`ReaderIndex = 1 - ReaderIndex;`
			`Write Buffer[ WriterIndex ]`
			`WriterIndex = 1 - WriterIndex;`

Ensure Determinism (Minimum Delay)	N/A	`Wait dataReady;`
		`Read data;`
		`Post readDone;`
			`Wait readDone;`
			`Write data;`
			`Post dataReady;`

Data Integrity Only C-HDL interface	The Simulink Coder and HDL Coder products both take advantage of target-specific communication implementations and devices to handle the data transfer between hardware and software components.		

The generated HDL code contains one HDL project for each hardware node.

Build on Desktop

Simulink Coder and Embedded Coder targets provide an example target to generate code for Windows, Linux and Mac OS operating systems. It is known as the native threads example, which is used to deploy your model to a desktop target. The desktop may not be your final target, but can help to profile and optimize your model before you deploy it on another target.

If you have specified an Embedded Coder target, make the following changes in the Configuration Parameters dialog box.

1. Select the **Code Generation** > **Templates** > **Generate an example main program** check box.
2. From the **Code Generation** > **Templates** > **Target Operating System** list, select `NativeThreadsExample`.
3. Click **OK** to save your changes and close the Configuration Parameters dialog box.
4. Apply these settings to all referenced models in your model.

Once you have set up your model, press **Ctrl-B** to build and deploy it to your desktop. The native threads example illustrates how Simulink Coder and Embedded Coder use target-specific threading APIs and data management primitives, as shown in “Threading APIs Used by Native Threads Example” on page 14-37. The data transfer between concurrently executing tasks behaves as described in Data Transfer Options. The coder products use the APIs on supported targets for this behavior, as described in “Data Protection and Synchronization APIs Used by Native Threads Example” on page 14-38.

Threading APIs Used by Native Threads Example

Aspect of Concurrent Execution	Linux Implementation	Windows Implementation	Mac OS Implementation
Periodic triggering event	POSIX timer	Windows timer	Not applicable
Aperiodic triggering event	POSIX real-time signal	Windows event	POSIX non-real-time signal

Aspect of Concurrent Execution	Linux Implementation	Windows Implementation	Mac OS Implementation
Aperiodic trigger	For blocks mapped to an aperiodic task: thread waiting for a signal For blocks mapped to an aperiodic trigger: signal action	Thread waiting for an event	For blocks mapped to an aperiodic task: thread waiting for a signal For blocks mapped to an aperiodic trigger: signal action
Threads	POSIX	Windows	POSIX
Threads priority	Assigned based on sample time: fastest task has highest priority	Priority class inherited from the parent process. Assigned based on sample time: fastest task has highest priority for the first three fastest tasks. The rest of the tasks share the lowest priority.	Assigned based on sample time: fastest task has highest priority
Example of overrun detection	Yes	Yes	No

Data Protection and Synchronization APIs Used by Native Threads Example

API	Linux Implementation	Windows Implementation	Mac OS Implementation
Data protection API	• `pthread_mutex_init` • `pthread_mutex_-destroy` • `pthread_mutex_lock` • `pthread_mutex_unlock`	• `CreateMutex` • `CloseHandle` • `WaitForSingleObject` • `ReleaseMutex`	• `pthread_mutex_init` • `pthread_mutex_-destroy` • `pthread_mutex_lock` • `pthread_mutex_unlock`
Synchronization API	• `sem_init` • `sem_destroy` • `sem_wait` • `sem_post`	• `CreateSemaphore` • `CloseHandle` • `WaitForSingleObject` • `ReleaseSemaphore`	• `sem_open` • `sem_unlink` • `sem_wait` • `sem_post`

Profile and Evaluate Explicitly Partitioned Models on a Desktop

Profile the execution of your code on the multicore target using the **Profile Report** pane of the Concurrent Execution dialog box. You can profile using Simulink Coder (GRT) and Embedded Coder (ERT) targets. Profiling helps you identify the areas in your model that are execution bottlenecks. You can analyze the execution time of each task and find the task that takes most of the execution time. For example, you can compare the average execution times of the tasks. If a task is computation intensive, or does not satisfy real-time requirements and overruns, you can break it into tasks that are less computation intensive and that can run concurrently.

When you generate a profile report, the software:

1. Builds the model.
2. Generates code for the model.
3. Adds tooling to the generated code to collect data.
4. Executes the generated code on the target and collects data.
5. Collates the data, generates an HTML file (*`model_name`*`_ProfileReport.html`) in the current folder, and displays that HTML file in the **Profile Report** pane of the Concurrent Execution dialog box.

Note If an HTML profile report exists for the model, the **Profile Report** pane displays that file.

To generate a new profile report, click  .

Section	Description
Summary	Summarizes model execution statistics, such as total execution time and profile report creation time. It also lists the total number of cores on the host machine.
Task Execution Time	Displays the execution time, in microseconds, for each task in a pie chart color coded by task. Visible for Windows, Linux, and Mac OS platforms.
Task Affinitization to Processor Cores	Platform-dependent. For each time step and task, Simulink displays the processor core number the task started executing on at that time step, color coded by processor. If there is no task scheduled for a particular time step, `NR` is displayed. Visible for Windows and Linux platforms.

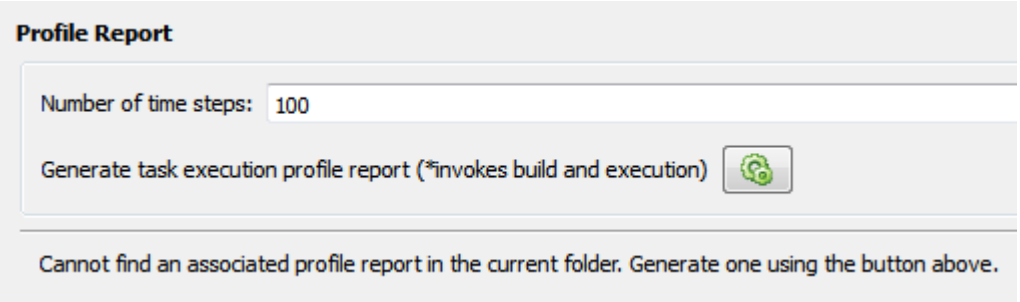
After you analyze the profile report, consider changing the mapping of Model blocks to efficiently use the concurrency available on your multicore system (see “Map Blocks to Tasks, Triggers, and Nodes” on page 14-28).

Generate Profile Report

This topic assumes a previously configured model ready to be profiled for concurrent execution. For more information, see “Configure Your Model for Concurrent Execution” on page 14-20.

1. In the Concurrent Execution dialog box, click the **Profile report** node.

   The profile tool looks for a file named *`model_name`*`_ProfileReport.html`. If such a file does not exist for the current model, the **Profile Report** pane displays the following.



Note If an HTML profile report exists for the model, the **Profile Report** pane displays that file.

To generate a new profile report, click  .

2 Enter the number of time steps for which you want the profiler to collect data for the model execution.

3 Click the **Generate task execution profile report** button.

This action builds the model, generates code, adds data collection tooling to the code, and executes it on the target, which also generates an HTML profile report. This process can take several minutes. When the process is complete, the contents of the profile report appear in the **Profile Report** pane. For example:

Task Execution Profiling Report for slexMulticoreExample

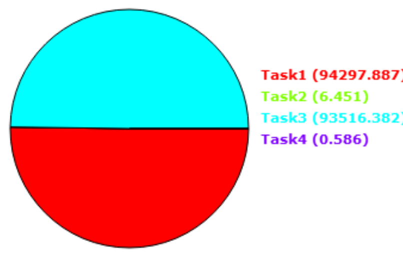
1. Summary

Total simulation time [seconds]	9.9
Profile data created	Thu Nov 12 16:47:52 2015
Model name	slexMulticoreExample
Number of processor cores on the host machine	12

2. Task Execution Time

Average Turnaround Time [microseconds]

Maximum Turnaround Time [microseconds]



3. Task Affinitization to Processor Cores

Clock Tick / Task	0										10										20				
Task1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Task2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Task3	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1
Task4	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1

Clock Tick / Task						30										40									
Task1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Task2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Task3	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR	1	NR
Task4	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR	NR	NR	1	NR

Legend

NR - Not scheduled to run

The profiling report shows the summary, execution time for each task, and the mapping of each task to processor cores. We see that tasks 1 and 2 run on core 0, where tasks 3 and 4 run on core 1. The **Task Execution Time** section of the report indicates that task 1 and task 3 take the most amount of time to run. Note that the period of task 3 is twice that of tasks 1 and 2, and the period of task 4 is twice that of task 3.

4 Analyze the profile report. Create and modify your model or task mapping if needed, and regenerate the profile report.

Generate Profile Report at Command Line

Alternatively, you can generate a profile report for a model configured for concurrent execution at the command line. Use the `Simulink.architecture.profile` function.

For example, to create a profile report for the model `slexMulticoreSolverExample`:

```
Simulink.architecture.profile('slexMulticoreSolverExample');
```

To create a profile report with a specific number of samples (100) for the model `slexMulticoreSolverExample`:

```
Simulink.architecture.profile('slexMulticoreSolverExample',120);
```

The function creates a profile report named `slexMulticoreSolverExample_ProfileReport.html` in your current folder.

Customize the Generated C Code

The generated code is suitable for many different applications and development environments. To meet your needs, you can customize the generated C code as described in “Target Development” (Embedded Coder). In addition to those customization capabilities, for multicore and heterogeneous targets you can further customize the generated code as follows:

- You can register your preferred implementation of mutual exclusion and data synchronization primitives using the code replacement library.
- You can define a custom target architecture file that allows you to specify target specific properties for tasks and triggers in the Concurrent Execution dialog box. For more information, see “Define a Custom Architecture File” on page 14-22.

See Also

Related Examples

- “Assigning Tasks to Cores for Multicore Programming”

More About

- “Multicore Programming with Simulink” on page 14-8

Programmatic Interface for Concurrent Execution

Use these functions to configure models for concurrent execution.

To	Use
Create or convert configuration for concurrent execution.	`Simulink.architecture.config`
Add triggers to the software node or add tasks to triggers.	`Simulink.architecture.add`
Delete triggers or tasks.	`Simulink.architecture.delete`
Find objects with specified parameter values.	`Simulink.architecture.find_system`
Get configuration parameters for target architecture.	`Simulink.architecture.get_param`
Import and select architecture.	`Simulink.architecture.importAndSelect`
Generate profile report for model configured for concurrent execution.	`Simulink.architecture.profile`
Add custom target architecture.	`Simulink.architecture.register`
Set properties for a concurrent execution object (such as task, trigger, or hardware node).	`Simulink.architecture.set_param`
Configure concurrent execution data transfers.	`Simulink.GlobalDataTransfer`

Map Blocks to Tasks

To map blocks to tasks, use the `set_param` function.

Map a block to one task:

```
set_param(block,'TargetArchitectureMapping',[bdroot 'CPU/PeriodicTrigger1/Task1']);
```

Map a block to multiple tasks:

```
set_param(block,'TargetArchitectureMapping',...
{[bdroot 'CPU/PeriodicTrigger1/Task1'];...
[bdroot 'CPU/PeriodicTrigger1/Task2']});
```

Get the current mapping of a block:

```
get_param(block,'TargetArchitectureMapping');
```

See Also

More About

- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44

Supported Targets For Multicore Programming

Supported Multicore Targets

You can build and download models that are implicitly or explicitly partitioned for the following multicore targets using system target files:

- Linux, Windows, and Mac OS using `ert.tlc` and `grt.tlc`.
- Simulink Real-Time using `slrealtime.tlc`.

Note

- To build and download your model, you must have Simulink Coder software installed.
- To build and download your model to a Simulink Real-Time system, you must have Simulink Real-Time software installed. You must also have a multicore target system supported by the Simulink Real-Time product.
- Deploying to an embedded processor that runs Linux and VxWorks® operating systems requires the Embedded Coder product.

Supported Heterogeneous Targets

In addition to multicore targets, Simulink also supports building and downloading partitions of an explicitly partitioned model to heterogeneous targets that contain a multicore target and one or more field-programmable gate arrays (FPGAs).

Select the heterogeneous architecture using the Target architecture option in the Concurrent Execution dialog box **Concurrent Execution** pane:

Item	Description
`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 containing FPGA boards.
`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.
`Altera Cyclone V SoC development kit Rev. C`	Altera® Cyclone® SoC Rev. C development kit target.
`Altera Cyclone V SoC development kit Rev. D`	Altera Cyclone SoC Rev. D development kit target.
`Arrow SoCKit development board`	Arrow® SoCKit development board target.

Note Building HDL code and downloading it to FPGAs requires the HDL Coder product. You can generate HDL code if:

- You have an HDL Coder license
- You are building on Windows or Linux operating systems

You cannot generate HDL code on Macintosh systems.

See Also

More About

- “Multicore Programming with Simulink” on page 14-8

Limitations with Multicore Programming in Simulink

The following limitations apply when partitioning a model for concurrent execution.

- Configure the model to use the fixed-step solver.
- Do not use the following modes of simulation for models in the concurrent execution environment:
  - External mode for desktop targets
  - Logging to MAT-files (**MAT-file logging** check box selected). However, you can use the To Workspace and To File blocks.
  - If you are simulating your model using Rapid Accelerator mode, the top-level model cannot contain a root level Inport block that outputs function calls.
  - In the Configuration Parameters dialog box, set the **Diagnostics > Sample Time > Multitask conditionally executed subsystem** and **Diagnostics > Data Validity > Multitask data store** parameters to `error`.
- If you want to use explicit partitioning, at the root level of your model, the model must consist entirely of
  - Models that are referenced using Model blocks
  - Subsystem blocks
  - MATLAB System blocks
  - MATLAB Function blocks
  - Stateflow charts
  - Rate Transition blocks
  - Virtual connectivity blocks

    The following are valid virtual connectivity blocks:

    - Goto and From blocks
    - Ground and Terminator blocks
    - Inport and Outport blocks
    - Virtual subsystem blocks that contain permitted blocks

See Also

More About

- “Multicore Programming with Simulink” on page 14-8
- “Supported Targets For Multicore Programming” on page 14-44

Modeling Best Practices

General Considerations when Building Simulink Models

In this section...

Avoiding Invalid Loops

You can connect the output of a block directly or indirectly (i.e., via other blocks) to its input, thereby, creating a loop. Loops can be very useful. For example, you can use loops to solve differential equations diagrammatically (see “Model a Continuous System” on page 15-6) or model feedback control systems. However, it is also possible to create loops that cannot be simulated. Common types of invalid loops include:

- Loops that create invalid function-call connections or an attempt to modify the input/output arguments of a function call (see “Using Function-Call Subsystems” on page 10-34 for a description of function-call subsystems)
- Self-triggering subsystems and loops containing non-latched triggered subsystems (see “Using Triggered Subsystems” on page 10-17 in the Using Simulink documentation for a description of triggered subsystems and Inport in the Simulink reference documentation for a description of latched input)
- Loops containing action subsystems

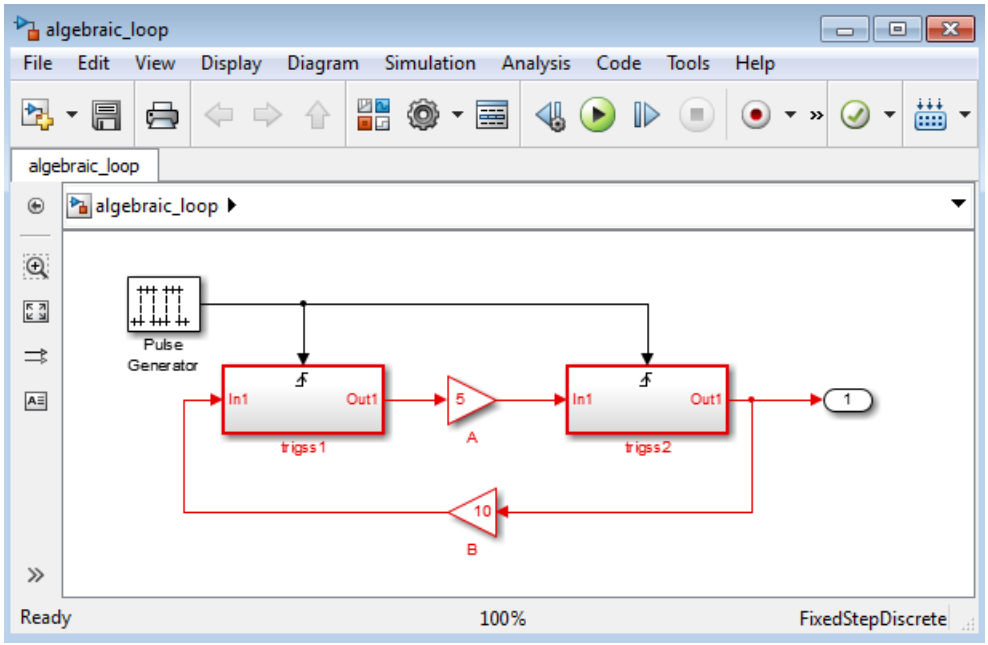
The Subsystem Examples block library in the Ports & Subsystems library contains models that illustrates examples of valid and invalid loops involving triggered and function-call subsystems. Examples of invalid loops include the following models:

- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1` (`sl_subsys_trigerr1`)
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2` (`sl_subsys_trigerr2`)
- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call systems/sl_subsys_fcncallerr3` (`sl_subsys_fcncallerr3`)

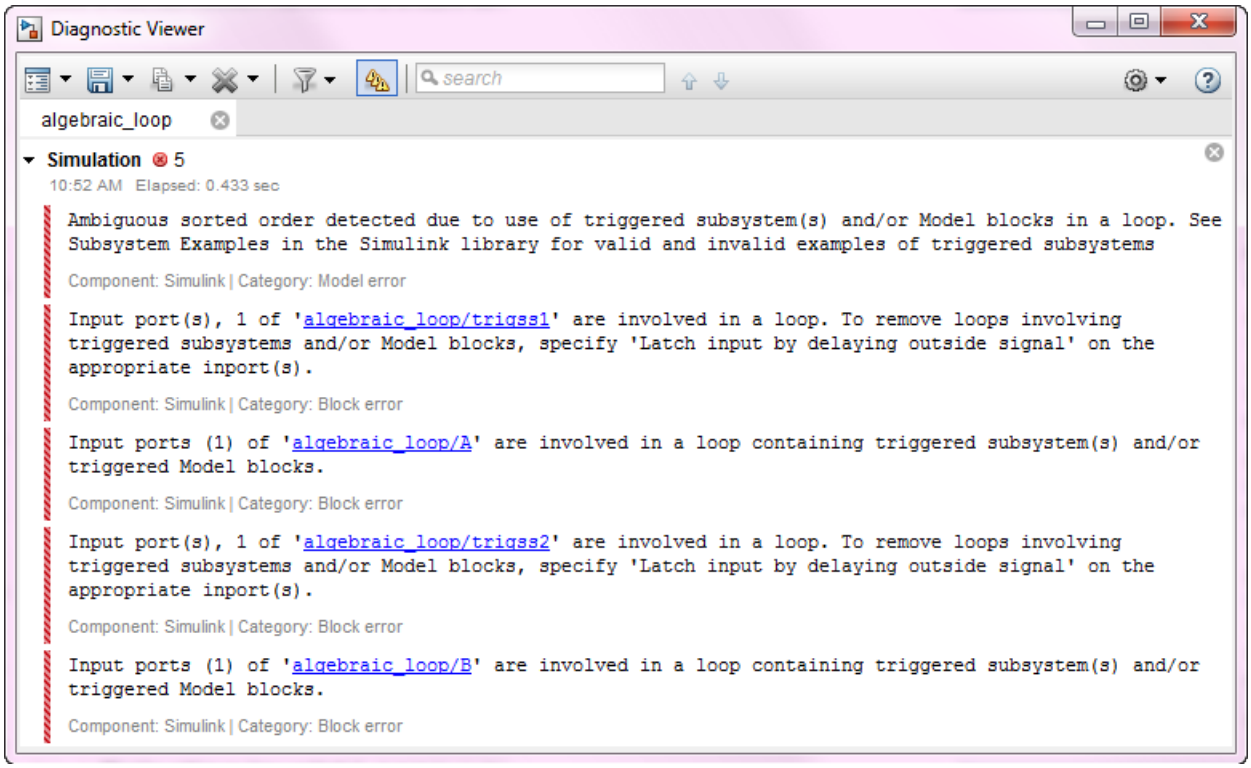
You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update Model** from the **Modeling** tab of the toolstrip. If the model contains invalid loops, the invalid loops are highlighted. This is illustrated in the following model ,



and displays an error message in the Diagnostic Viewer.



Shadowed Files

If there are two Model files with the same name (e.g. `mylibrary.slx`) on the MATLAB path, the one higher on the path is loaded, and the one lower on the path is said to be "shadowed".

Tip To help avoid problems with shadowed files, turn on the Simulink preference **Do not load models that are shadowed on the MATLAB path**. See “Do not load models that are shadowed on the MATLAB path”.

The rules Simulink software uses to find Model files are similar to those used by MATLAB software. See “What Is the MATLAB Search Path?”. However, there is an important difference between how Simulink block diagrams and MATLAB functions are handled: a loaded block diagram takes precedence over any unloaded ones, regardless of its position on the MATLAB path. This is done for performance reasons, as part of the Simulink software's incremental loading methodology.

The precedence of a loaded block diagram over any others can have important implications, particularly since a block diagram can be loaded without the corresponding Simulink window being visible.

Making Sure the Correct Block Diagram Is Loaded

When using libraries and referenced models, you can load a block diagram without showing its window. If the MATLAB path or the current MATLAB folder changes while block diagrams are in memory, these block diagrams can interfere with the use of other files of the same name.

For example, open a model with a library called `mylib`, change to another folder, and then open another model with a library also called `mylib`. When you run the first model, it uses the library associated with the second model.

This can lead to problems including:

- Simulation errors
- "Unresolved Link" icons on blocks that are library links
- Wrong results

Detecting and Fixing Problems

To help avoid problems with shadowed files, you can turn on the Simulink preference **Do not load models that are shadowed on the MATLAB path**. See “Do not load models that are shadowed on the MATLAB path”.

When updating a block diagram, the Simulink software checks the position of its file on the MATLAB path and will issue a warning if it detects that another file of the same name exists and is higher on the MATLAB path. The warning reads:

```
The file containing block diagram 'mylibrary' is shadowed
by a file of the same name higher on the MATLAB path.
```

This may indicate that the wrong file called `mylibrary.slx` is being used. To see which file called `mylibrary.slx` is loaded into memory, enter:

```
 which mylibrary

C:\work\Model1\mylibrary.slx
```

To see all the files called `mylibrary` which are on the MATLAB path, including MATLAB scripts, enter:

```
which -all mylibrary
```

```
C:\work\Model1\mylibrary.slx
C:\work\Model2\mylibrary.slx  % Shadowed
```

To close the block diagram called `mylibrary` and let the Simulink software load the file which is highest on the MATLAB path, enter:

```
close_system('mylibrary')
```

Model Building Tips

Here are some model-building hints you might find useful:

- Memory issues

  In general, more memory will increase performance.

- Using hierarchy

  More complex models often benefit from adding the hierarchy of subsystems to the model. Grouping blocks simplifies the top level of the model and can make it easier to read and understand the model. For more information, see “Create Subsystems” on page 4-15. The Model Browser provides useful information about complex models (see **Simulink Editor**).

- Cleaning up models

  Well organized and documented models are easier to read and understand. Signal labels and model annotations can help describe what is happening in a model. For more information, see “Signal Names and Labels” on page 75-3 and “Describe Models Using Notes and Annotations” on page 4-3.

- Modeling strategies

  If several of your models use the same blocks, you can save these blocks for easy reuse. For example, you can save a collection of blocks in a custom library. Then, when you build new models, you can copy these blocks from the library.

  Generally, when building a model, design it first on paper, then build it using the computer. Then, when you start putting the blocks together into a model, add the blocks to the model window before adding the lines that connect them. This way, you can reduce how often you need to open block libraries.

See Also

Related Examples

- “Model a Continuous System” on page 15-6
- “Best-Form Mathematical Models” on page 15-9
- “Model a Simple Equation” on page 15-12
- “Model Differential Algebraic Equations” on page 15-14

More About

- “Component-Based Modeling Guidelines” on page 22-2

Model a Continuous System

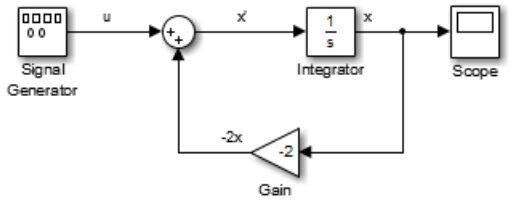
To model the differential equation

$x' = -2x(t)+u(t)$,

where $u(t)$ is a square wave with an amplitude of 1 and a frequency of 1 rad/sec, use an integrator block and a gain block. The Integrator block integrates its input x' to produce x. Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator block and select the Square Wave form but change the default units to radians/sec. Again, view the output using a Scope block. Gather the blocks and define the gain.

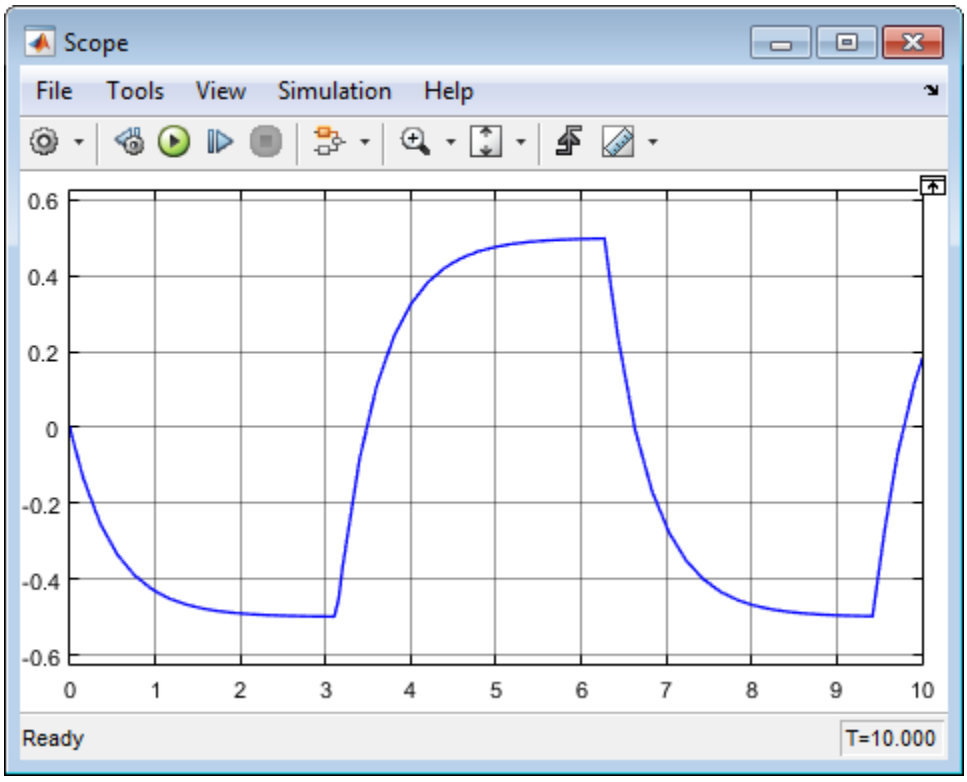
In this model, to reverse the direction of the Gain block, select the block, then on the **Format** tab, click Flip left-right  . To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see “Branch a Connection” on page 1-16.

Now you can connect all the blocks.



An important concept in this model is the loop that includes the Sum block, the Integrator block, and the Gain block. In this equation, x is the output of the Integrator block. It is also the input to the blocks that compute x', on which it is based. This relationship is implemented using a loop.

The Scope displays x at each time step. For a simulation lasting 10 seconds, the output looks like this:



The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts u as input and outputs x. So, the block implements x/u. If you substitute sx for x' in the above equation, you get

$sx = -2x + u$.

Solving for x gives

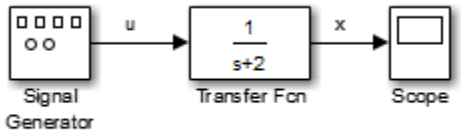
$x = u/(s + 2)$

or,

$x/u = 1/(s + 2)$.

The Transfer Fcn block uses parameters to specify the numerator and denominator coefficients. In this case, the numerator is `1` and the denominator is `s+2`. Specify both terms as vectors of coefficients of successively decreasing powers of `s`.

In this case the numerator is `[1]` (or just `1`) and the denominator is `[1 2]`.



The results of this simulation are identical to those of the previous model.

See Also

More About

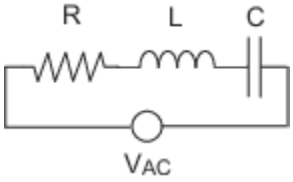
- “General Considerations when Building Simulink Models” on page 15-2
- “Component-Based Modeling Guidelines” on page 22-2

Best-Form Mathematical Models

In this section...

Series RLC Example

You can often formulate the mathematical system you are modeling in several ways. Choosing the best-form mathematical model allows the simulation to execute faster and more accurately. For example, consider a simple series RLC circuit.



According to Kirchoff's voltage law, the voltage drop across this circuit is equal to the sum of the voltage drop across each element of the circuit.

$$V_{AC} = V_R + V_L + V_C$$

Using Ohm's law to solve for the voltage across each element of the circuit, the equation for this circuit can be written as

$$V_{AC} = Ri + L\frac{di}{dt} + \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

You can model this system in Simulink by solving for either the resistor voltage or inductor voltage. Which you choose to solve for affects the structure of the model and its performance.

Solving Series RLC Using Resistor Voltage

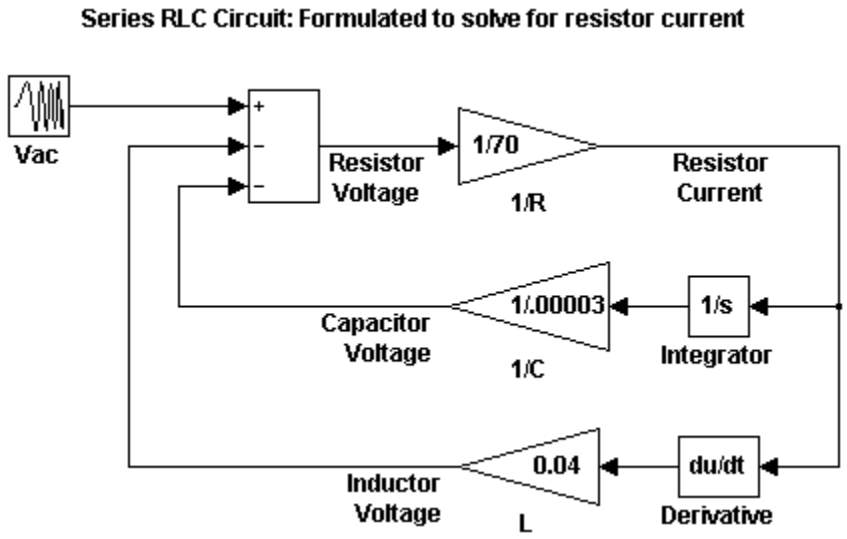
Solving the RLC circuit for the resistor voltage yields

$$V_R = Ri$$

$$Ri = V_{AC} - L\frac{di}{dt} - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

Circuit Model

The following diagram shows this equation modeled in Simulink where R is `70`, C is `0.00003`, and L is `0.04`. The resistor voltage is the sum of the voltage source, the capacitor voltage, and the inductor voltage. You need the current in the circuit to calculate the capacitor and inductor voltages. To calculate the current, multiply the resistor voltage by a gain of *1/R*. Calculate the capacitor voltage by integrating the current and multiplying by a gain of *1/C*. Calculate the inductor voltage by taking the derivative of the current and multiplying by a gain of *L*.



This formulation contains a Derivative block associated with the inductor. Whenever possible, you should avoid mathematical formulations that require Derivative blocks as they introduce discontinuities into your system. Numerical integration is used to solve the model dynamics though time. These integration solvers take small steps through time to satisfy an accuracy constraint on the solution. If the discontinuity introduced by the Derivative block is too large, it is not possible for the solver to step across it.

In addition, in this model the Derivative, Sum, and two Gain blocks create an algebraic loop. Algebraic loops slow down the model's execution and can produce less accurate simulation results. See “Algebraic Loop Concepts” on page 3-27 for more information.

Solving Series RLC Using Inductor Voltage

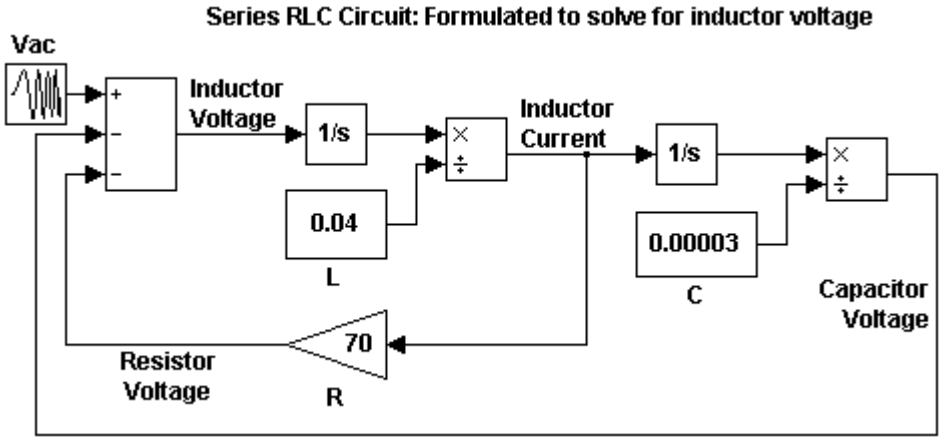
To avoid using a Derivative block, formulate the equation to solve for the inductor voltage.

$$V_L = L\frac{di}{dt}$$

$$L\frac{di}{dt} = V_{AC} - Ri - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

Circuit Model

The following diagram shows this equation modeled in Simulink. The inductor voltage is the sum of the voltage source, the resistor voltage, and the capacitor voltage. You need the current in the circuit to calculate the resistor and capacitor voltages. To calculate the current, integrate the inductor voltage and divide by L. Calculate the capacitor voltage by integrating the current and dividing by C. Calculate the resistor voltage by multiplying the current by a gain of R.



This model contains only integrator blocks and no algebraic loops. As a result, the model simulates faster and more accurately.

See Also

Related Examples

- “Model a Simple Equation” on page 15-12
- “Model Differential Algebraic Equations” on page 15-14

More About

- “General Considerations when Building Simulink Models” on page 15-2
- “Component-Based Modeling Guidelines” on page 22-2

Model a Simple Equation

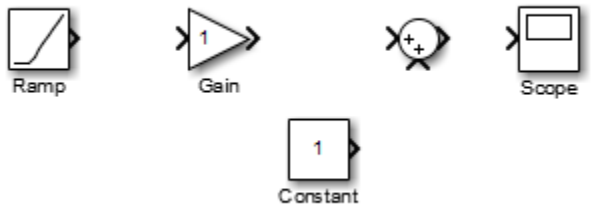
To model the equation that converts Celsius temperature to Fahrenheit

$T_F = 9/5(T_C) + 32$

First, consider the blocks needed to build the model:

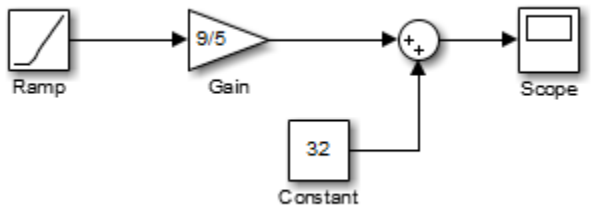
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library
- A Gain block to multiply the input signal by 9/5, from the Math Operations library
- A Sum block to add the two quantities, also from the Math Operations library
- A Scope block to display the output, from the Sinks library

Next, gather the blocks into your model window.



Assign parameter values to the Gain and Constant blocks by opening (double-clicking) each block and entering the appropriate value. Then, click the **OK** button to apply the value and close the dialog box.

Now, connect the blocks.



The Ramp block inputs Celsius temperature. Open that block and change the **Initial output** parameter to 0. The Gain block multiplies that temperature by the constant 9/5. The Sum block adds the value 32 to the result and outputs the Fahrenheit temperature.

Open the Scope block to view the output. Now, choose **Run** from the **Simulation** menu to run the simulation. The simulation runs for 10 seconds.

See Also

Related Examples

- “Best-Form Mathematical Models” on page 15-9
- “Model Differential Algebraic Equations” on page 15-14

More About

- “General Considerations when Building Simulink Models” on page 15-2
- “Component-Based Modeling Guidelines” on page 22-2

Model Differential Algebraic Equations

In this section...

Overview of Robertson Reaction Example

Robertson [1] on page 15-21 created a system of autocatalytic chemical reactions to test and compare numerical solvers for stiff systems. The reactions, rate constants (k), and reaction rates (V) for the system are given as follows:

$$A \xrightarrow{k_1} B \qquad k_1 = 0.04 \qquad V_1 = k_1[A]$$

$$B + B \xrightarrow{k_2} C + B \quad k_2 = 3 \cdot 10^7 \quad V_2 = k_2[B][B]$$

$$B + C \xrightarrow{k_3} A + C \quad k_3 = 1 \cdot 10^4 \quad V_3 = k_3[B][C]$$

Because there are large differences between the reaction rates, the numerical solvers see the differential equations as stiff. For stiff differential equations, some numerical solvers cannot converge on a solution unless the step size is extremely small. If the step size is extremely small, the simulation time can be unacceptably long. In this case, you need to use a numerical solver designed to solve stiff equations.

Simulink Model from ODE Equations

A system of ordinary differential equations (ODE) has the following characteristics:

- All of the equations are ordinary differential equations.
- Each equation is the derivative of a dependent variable with respect to one independent variable, usually time.
- The number of equations is equal to the number of dependent variables in the system.

Using the reaction rates, you can create a set of differential equations describing the rate of change for each chemical species. Since there are three species, there are three differential equations in the mathematical model.

$$A' = -0.04A + 1 \cdot 10^4 BC$$

$$B' = 0.04A - 1 \cdot 10^4 BC - 3 \cdot 10^7 B^2$$

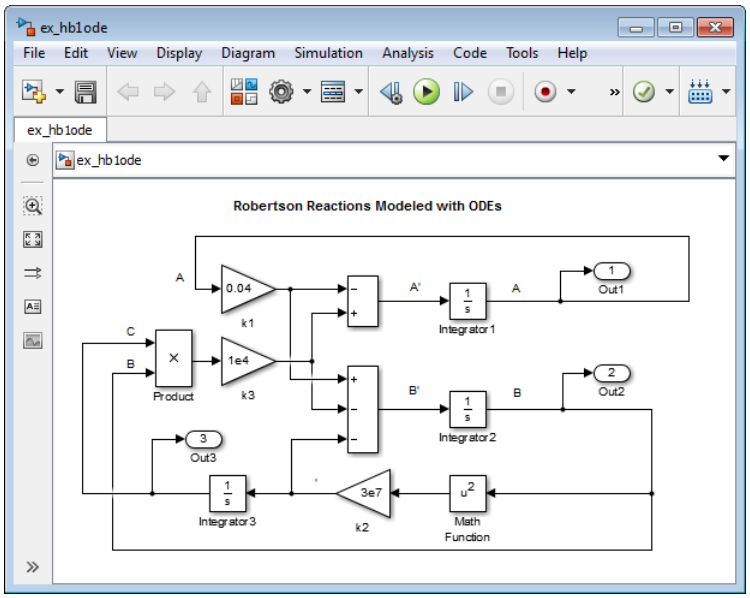
$$C' = 3 \cdot 10^7 B^2$$

Initial conditions: $A = 1$, $B = 0$, and $C = 0$.

Build the Model

Create a model, or open the model `ex_hblode`.

1. Add three Integrator blocks to your model. Label the inputs `A'`, `B'`, and `C'`, and the outputs `A`, `B`, and `C` respectively.
2. Add Sum, Product, and Gain blocks to solve each differential variable. For example, to model the signal `C'`,
   a. Add a Math Function block and connect the input to signal `B`. Set the **Function** parameter to `square`.
   b. Connect the output from the Math Function block to a Gain block. Set the **Gain** parameter to `3e7`.
   c. Continue to add the remaining differential equation terms to your model.
3. Model the initial condition of `A` by setting the **Initial condition** parameter for the `A` Integrator block to `1`.
4. Add Out blocks to save the signals `A`, `B`, and `C` to the MATLAB variable `yout`.



Simulate the Model

Create a script that uses the `sim` command to simulate your model. This script saves the simulation results in the MATLAB variable `yout`. Since the simulation has a long time interval and `B` initially changes very fast, plotting values on a logarithmic scale helps to visually compare the results. Also, since the value of `B` is small relative to the values of `A` and `C`, multiply `B` by $1 \cdot 10^4$ before plotting the values.

1. Enter the following statements in a MATLAB script. If you built your own model, replace `ex_hblode` with the name of your model.

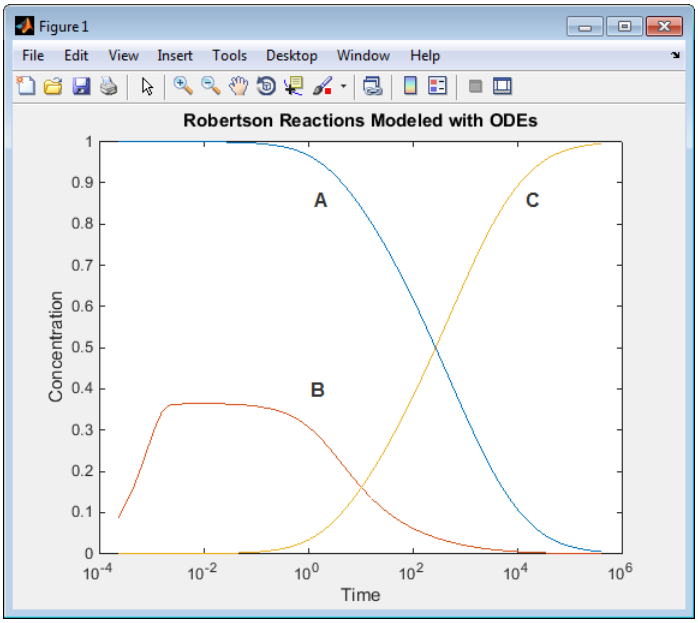
```
sim('ex_hb1ode')
yout(:,2) = 1e4*yout(:,2);
figure;
semilogx(tout,yout);
xlabel('Time');
ylabel('Concentration');
title('Robertson Reactions Modeled with ODEs')
```

2 From the Simulink Editor, on the **Modeling** tab, click **Model Settings**.

— In the Solver pane, set the **Stop time** to `4e5` and the **Solver** to `ode15s (stiff/NDF)`.

— In the Data Import pane, select the **Time** and **Output** check boxes.

3 Run the script. Observe that all of `A` is converted to `C`.



Simulink Model from DAE Equations

A system of differential algebraic equations (DAE) has the following characteristics:

- It contains both ordinary differential equations and algebraic equations. Algebraic equations do not have any derivatives.
- Only some of the equations are differential equations defining the derivatives of some of the dependent variables. The other dependent variables are defined with algebraic equations.
- The number of equations is equal to the number of dependent variables in the system.

Some systems contain constraints due to conservation laws, such as conservation of mass and energy. If you set the initial concentrations to$A = 1$, $B = 0$, and $C = 0$, the total concentration of the three species is always equal to 1 since $A + B + C = 1$. You can replace the differential equation for $C'$with the following algebraic equation to create a set of differential algebraic equations (DAEs).

$$C = 1 - A - B$$

The differential variables A and B uniquely determine the algebraic variable C.

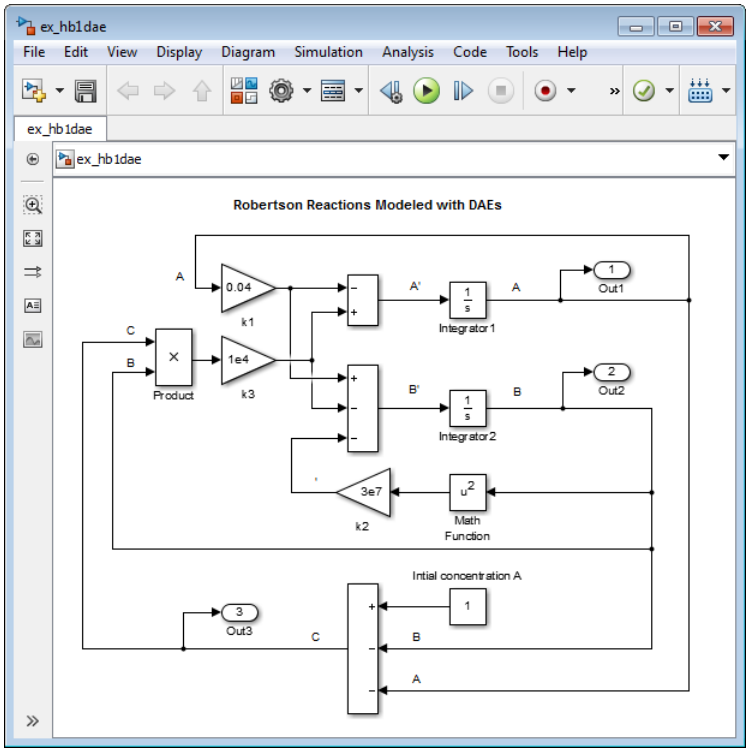
$$A' = -0.04A + 1 \cdot 10^4 BC$$
$$B' = 0.04A - 1 \cdot 10^4 BC - 3 \cdot 10^7 B^2$$
$$C = 1 - A - B$$

Initial conditions: $A = 1$and $B = 0$.

Build the Model

Make these changes to your model or to the model `ex_hb1ode`, or open the model `ex_hb1dae`.

1. Delete the Integrator block for calculating `C`.
2. Add a Sum block and set the **List of signs** parameter to `+- -`.
3. Connect the signals `A` and `B` to the minus inputs of the Sum block.
4. Model the initial concentration of `A` with a Constant block connected to the plus input of the Sum block. Set the **Constant value** parameter to `1`.
5. Connect the output of the Sum block to the branched line connected to the Product and Out blocks.



Simulate the Model

Create a script that uses the `sim` command to simulate your model.

1. Enter the following statements in a MATLAB script. If you built your own model, replace `ex_hbldae` with the name of your model.

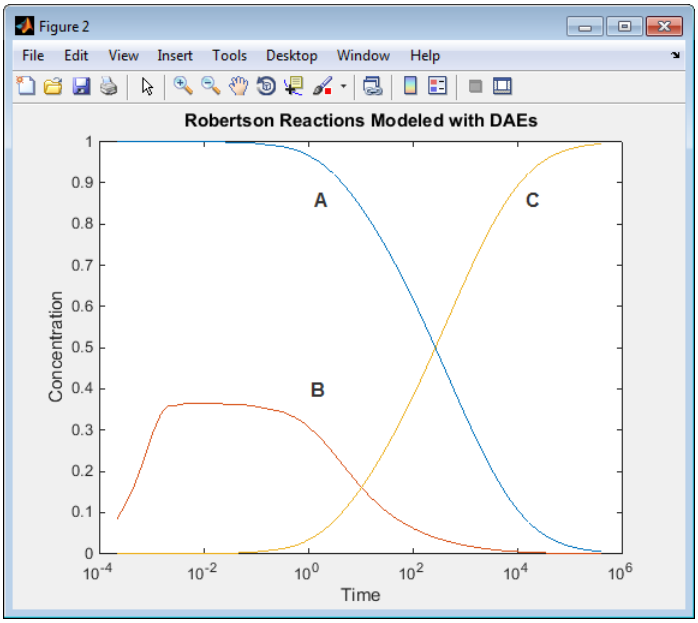
```
sim('ex_hb1dae')
yout(:,2) = 1e4*yout(:,2);
figure;
semilogx(tout,yout);
xlabel('Time');
ylabel('Concentration');
title('Robertson Reactions Modeled with DAEs')
```

2. From the Simulink Editor, on the **Modeling** tab, click **Model Settings**.

   — In the Solver pane, set the **Stop time** to `4e5` and the **Solver** to `ode15s (stiff/NDF)`.

— In the Data Import pane, select the **Time** and **Output** check boxes.

3 Run the script. The simulation results when you use an algebraic equation are the same as for the model simulation using only differential equations.



Simulink Model from DAE Equations Using Algebraic Constraint Block

Some systems contain constraints due to conservation laws, such as conservation of mass and energy. If you set the initial concentrations to$A = 1$, $B = 0$, and $C = 0$, the total concentration of the three species is always equal to 1 since $A + B + C = 1$.

You can replace the differential equation for $C'$with an algebraic equation modeled using an Algebraic Constraint block and a Sum block. The Algebraic Constraint block constrains its input signal F(z) to zero and outputs an algebraic state z. In other words, the block output is a value needed to produce a zero at the input. Use the following algebraic equation for input to the block.

$$0 = A + B + C - 1$$

The differential variables A and B uniquely determine the algebraic variable C.

$$A' = -0.04A + 1 \cdot 10^4 BC$$
$$B' = 0.04A - 1 \cdot 10^4 BC - 3 \cdot 10^7 B^2$$
$$C = 1 - A - B$$

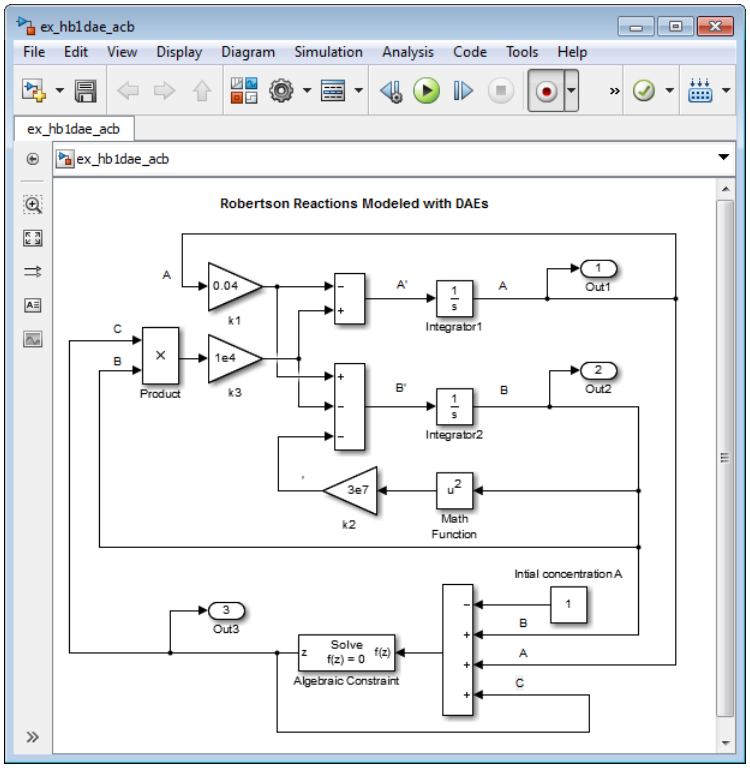
Initial conditions: $A = 1$, $B = 0$, and $C = 1 \cdot 10^{-3}$.

Build the Model

Make these changes to your model or to the model `ex_hb1ode`, or open the model `ex_hb1dae_acb`.

1 Delete the Integrator block for calculating `C`.

2. Add an Algebraic Constraint block. Set the **Initial guess** parameter to `1e-3`.
3. Add a Sum block. Set the **List of signs** parameter to `-+++`.
4. Connect the signals `A` and `B` to plus inputs of the Sum block.
5. Model the initial concentration of `A` with a Constant block connected to the minus input of the Sum block. Set the **Constant value** parameter to `1`.
6. Connect the output of the Algebraic Constraint block to the branched line connected to the Product and Out block inputs.
7. Create a branch line from the output of the Algebraic Constraint block to the final plus input of the Sum block.



Simulate the Model

Create a script that uses the `sim` command to simulate your model.

1. Enter the following statements in a MATLAB script. If you built your own model, replace `ex_hbl_acb` with the name of your model.

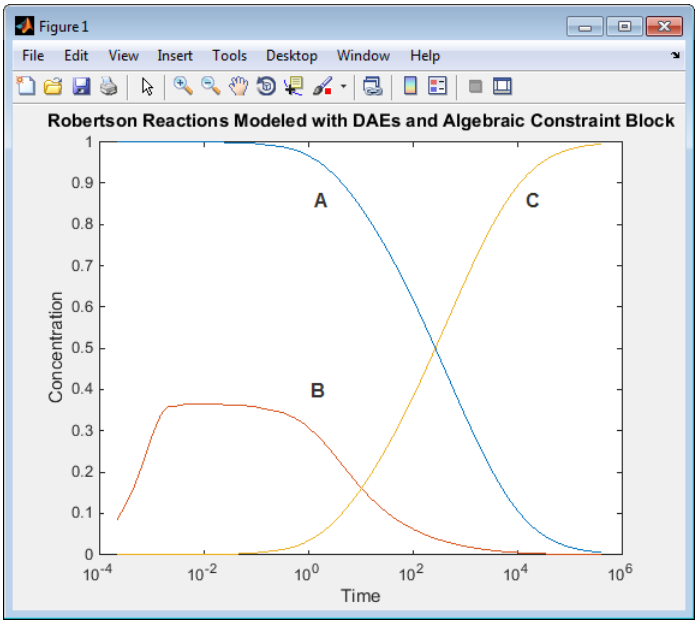
```
sim('ex_hb1dae_acb')
yout(:,2) = 1e4*yout(:,2);
figure;
semilogx(tout,yout);
xlabel('Time');
ylabel('Concentration');
title('Robertson Reactions Modeled with DAEs and Algebraic Constraint Block')
```

2. From the Simulink Editor, on the **Modeling** tab, click **Model Settings**.

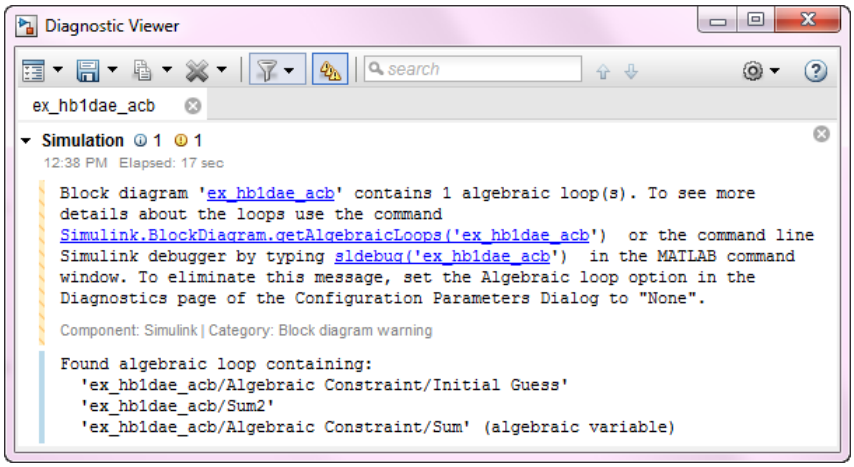
   — In the Solver pane, set the **Stop time** to `4e5` and the **Solver** to `ode15s (stiff/NDF)`.

— In the Data Import pane, select the **Time** and **Output** check boxes.

3 Run the script. The simulation results when you use an Algebraic Constraint block are the same as for the model simulation using only differential equations.



Using an Algebraic Constraint block creates an algebraic loop in a model, If you set the **Algebraic Loop** parameter to `warning` (on the **Modeling** tab, click **Model Settings**, then select **Diagnostics**), the following message displays in the Diagnostic Viewer during simulation.



For this model, the algebraic loop solver was able to find a solution for the simulation, but algebraic loops don’t always have a solution, and they are not supported for code generation. For more information about algebraic loops in Simulink models and how to remove them, see “Algebraic Loop Concepts” on page 3-27.

References

[1] Robertson, H. H. “The solution of a set of reaction rate equations.” *Numerical Analysis: An Introduction* (J. Walsh ed.). London, England:Academic Press, 1966, pp. 178–182.

See Also

Related Examples

- “Best-Form Mathematical Models” on page 15-9
- “Model Differential Algebraic Equations” on page 15-14

More About

- “General Considerations when Building Simulink Models” on page 15-2
- “Compare Capabilities of Model Components” on page 22-8

Basic Modeling Workflow

In large, well-defined projects, you define file management and model architecture upfront. In other words, your predefined requirements drive components and interfaces.

To understand how to create such projects, you must know how to create scalable models that represent system equations. This example shows how, as model size and complexity increases, file storage and model architecture change to satisfy evolving project requirements.

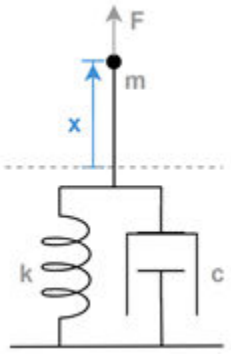
1. “Model a System Algorithm” on page 15-23
2. “Create Model Components” on page 15-25
3. “Manage Signal Lines” on page 15-28
4. “Manage Model Data” on page 15-33
5. “Reuse Model Components from Files” on page 15-35
6. “Create Interchangeable Variations of Model Components” on page 15-38
7. “Set Up a File Management System” on page 15-40

You can use the files provided at any of these steps as starting points for your design. The final example helps you create a project that contains the model components and supporting files developed in this example set.

Model a System Algorithm

When incomplete system requirements and a developing system design prevent you from defining file management and model architecture upfront, you can still model fundamental system algorithms. By organizing the model into inputs, outputs, and systems, you create a general framework for model components as the model grows.

To show the first stage of a modeling workflow that begins with limited information, this example uses a simple mechanical system composed of a mass, spring, and damper.



This second-order differential equation characterizes the system:

$$m\ddot{x} + c\dot{x} + kx = F$$

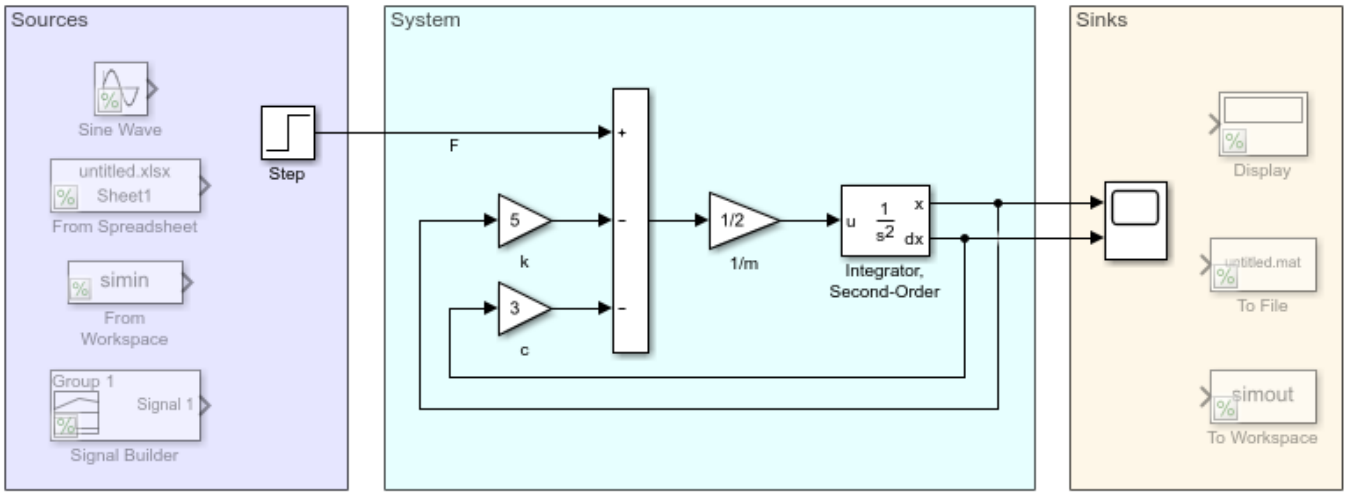
where

- m is mass
- c is the damping coefficient
- k is the spring constant
- $\ddot{x}$ is acceleration
- $\dot{x}$ is velocity
- x is displacement
- F is force

Solving for $\ddot{x}$ provides a form of this equation that maps more clearly to a Simulink® model.

$$\ddot{x} = \frac{1}{m}(F - c\dot{x} - kx)$$

In model `ex_modeling_simple_system`, a Sum block computes the forces applied to the mass, the Gain block labeled `1/m` computes the acceleration of the mass, and the Second-Order Integrator block solves for the velocity and position of the mass.



These blocks, which represent the system, are grouped in an area. The two other areas contain system inputs and outputs. By organizing the model upfront, you create a general framework for model components as the model grows.

Since this example shows a model in the preliminary stages of development, the actual input force is unknown and can be represented by a variety of standard source blocks. Model `ex_modeling_simple_system` uses a Step block connected as input to the system. Some alternative source blocks are shown, but commented out. For example, you can use the From Spreadsheet block to load empirical data if it were available.

Similarly, a variety of sink blocks can accept the output displacement. To check whether simulation results meet expectations, model `ex_modeling_simple_system` uses a Scope block to visualize the signals.

See Also

More About

- “Format a Model” on page 36-7
- “Build and Edit a Model Interactively” on page 1-8
- “Sources”
- “Sinks”

Create Model Components

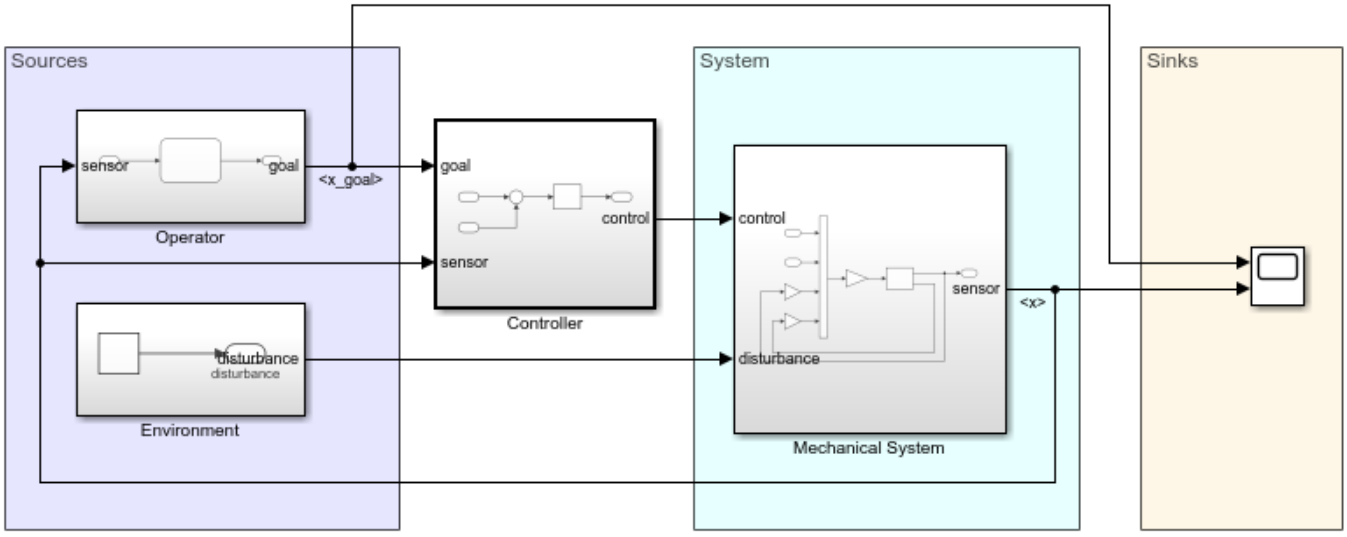
As you gather requirements for a system, you identify model components. You can identify where component interfaces exist even with incomplete specifications.

To define model components without affecting simulation results and specifying an interface that may change, you can create subsystems and visually organize the model.

Some components, such as digital controllers, should execute as a single unit within the model. For these standalone components with known boundaries, you can use an atomic subsystem. Defining atomic components upfront prevents costly refactoring when you want to generate standalone code.

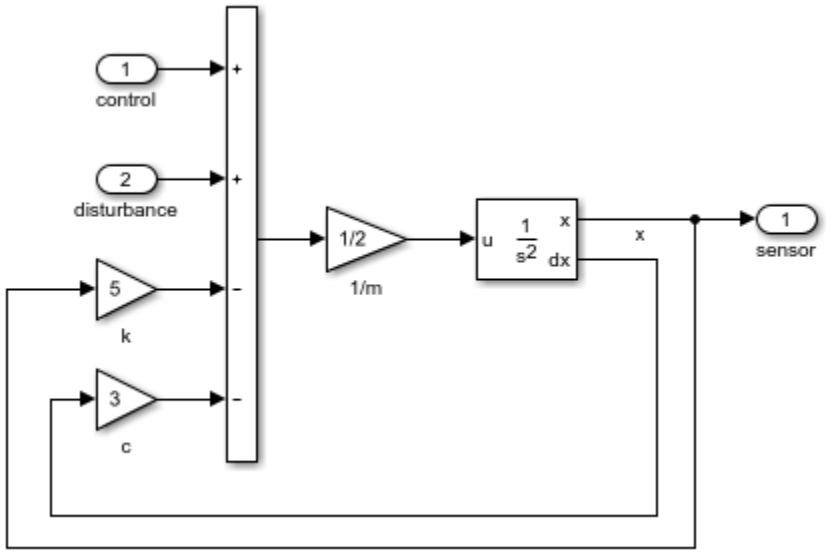
Model `ex_modeling_components` contains four common model components.

- Mechanical System — A mass separated from a surface by a spring and damper
- Controller — Algorithm that controls the motion of the mechanical system
- Operator — Logic that defines the commands sent to the controller
- Environment — External disturbances that affect the mechanical system



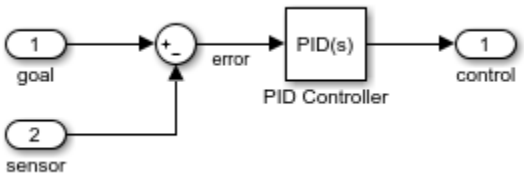
The thicker line weight on the Controller block indicates that the controller is an atomic subsystem.

The ports on each of the Subsystem blocks correspond to input and output blocks within the subsystem. The block label shows the name of the corresponding port. For example, the Inport block labeled `disturbance` corresponds with the `disturbance` port of the Mechanical System block.

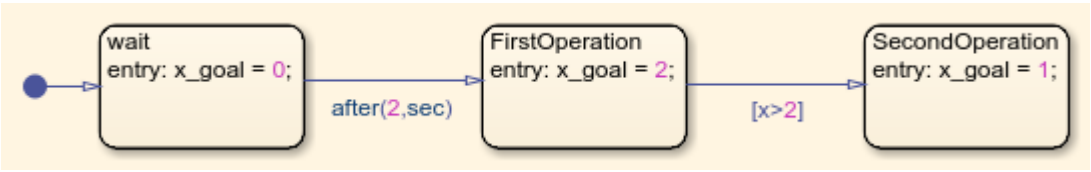


The four model components interact to determine the position of the mass.

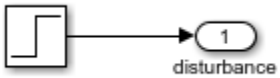
The controller computes the force required to move the mechanical system to its goal position.



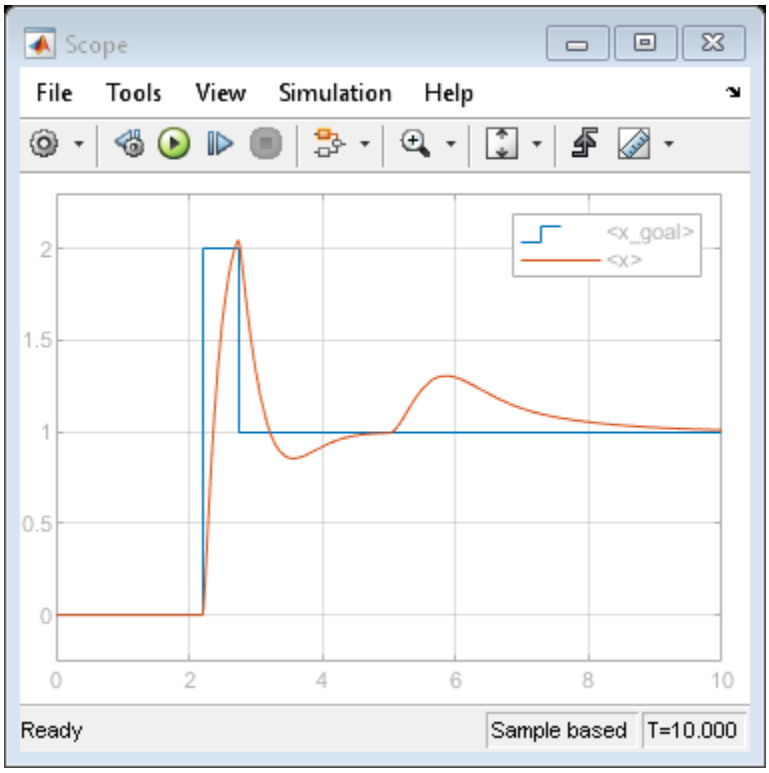
The operator determines the goal position of the mass and implements the related procedural logic with a Stateflow® chart.



The environment generates a disturbance force that affects the mechanical system.



During simulation, the operator tells the controller to wait 2 seconds, then move the mass up 2 meters. When the mass overshoots the goal position, the operator tells the controller to position the mass 1 meter above its original position. After 5 seconds, an environmental disturbance applies a steady force to the physical system and the controller reacts to stabilize the mass at the goal position.



See Also

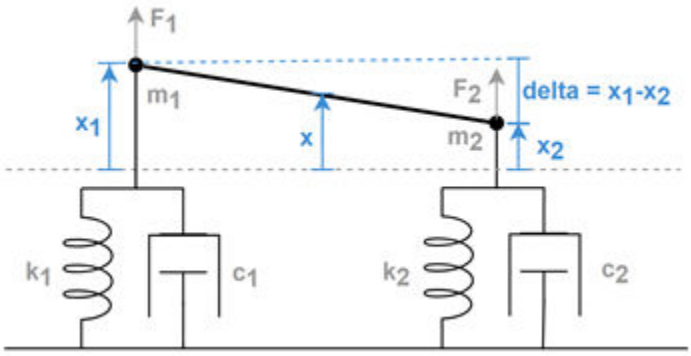
More About

- “Create Subsystems” on page 4-15
- “Expand Subsystem Contents” on page 4-33
- “Share Data with Simulink and the MATLAB Workspace” (Stateflow)

Manage Signal Lines

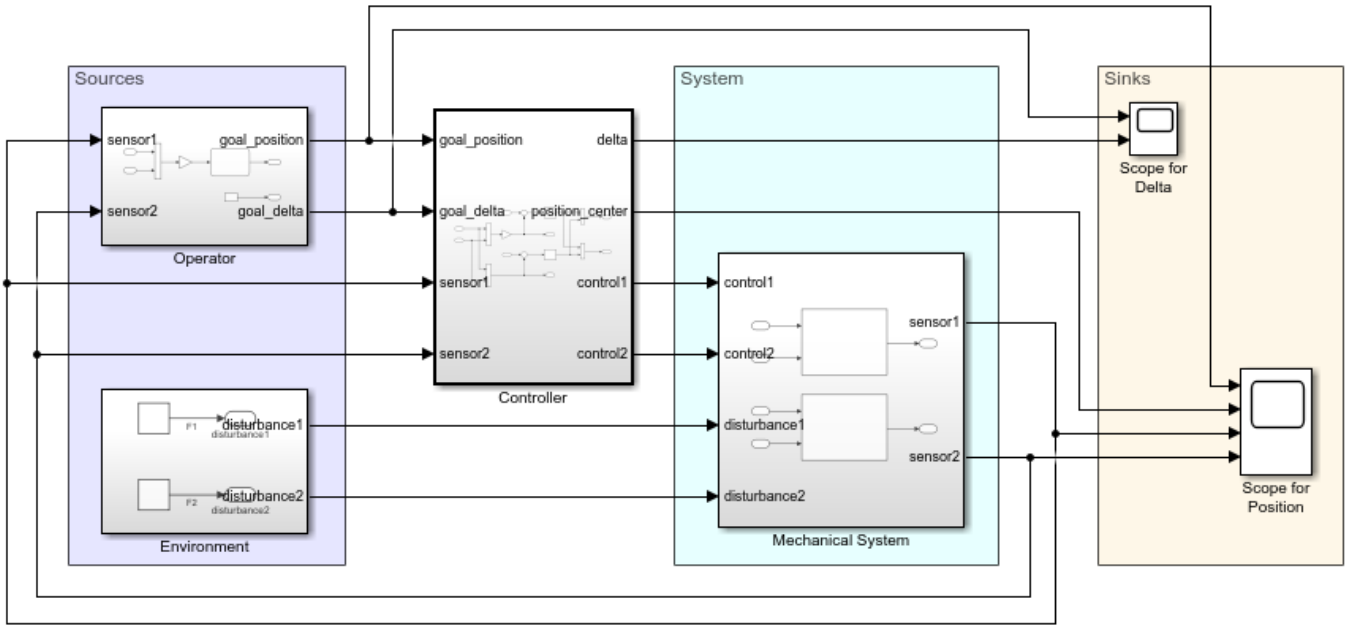
As a model grows, model components help functionally and visually organize blocks in the model. To similarly organize signal lines in the model, you can apply a variety of strategies, such as grouping signals into buses.

To demonstrate how to reduce signal line clutter, this example implements a model with multiple sensors and actuators. The system has two springs, dampers, and masses. A beam connects the two masses, as shown in this image.

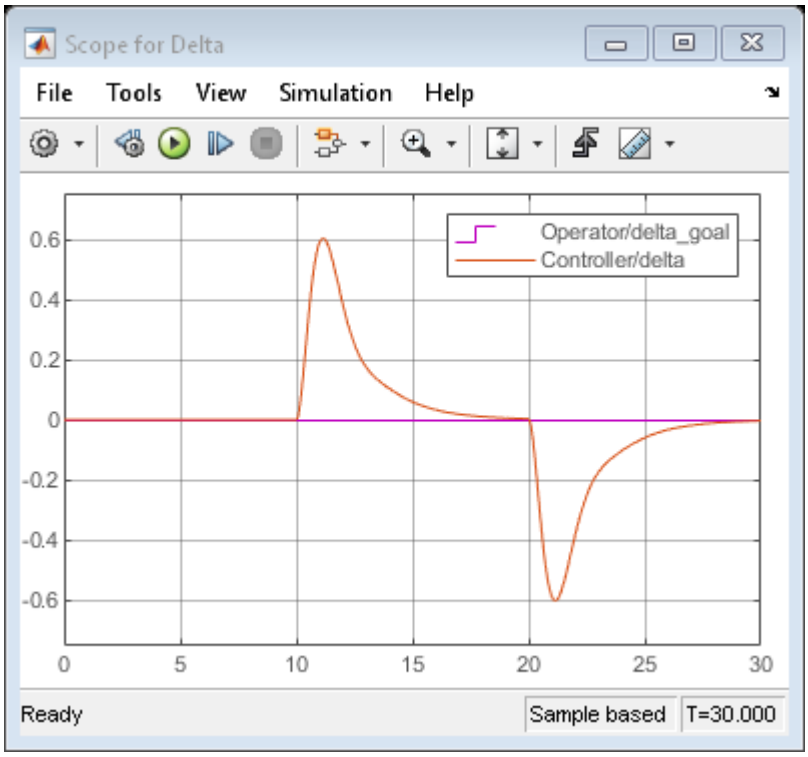


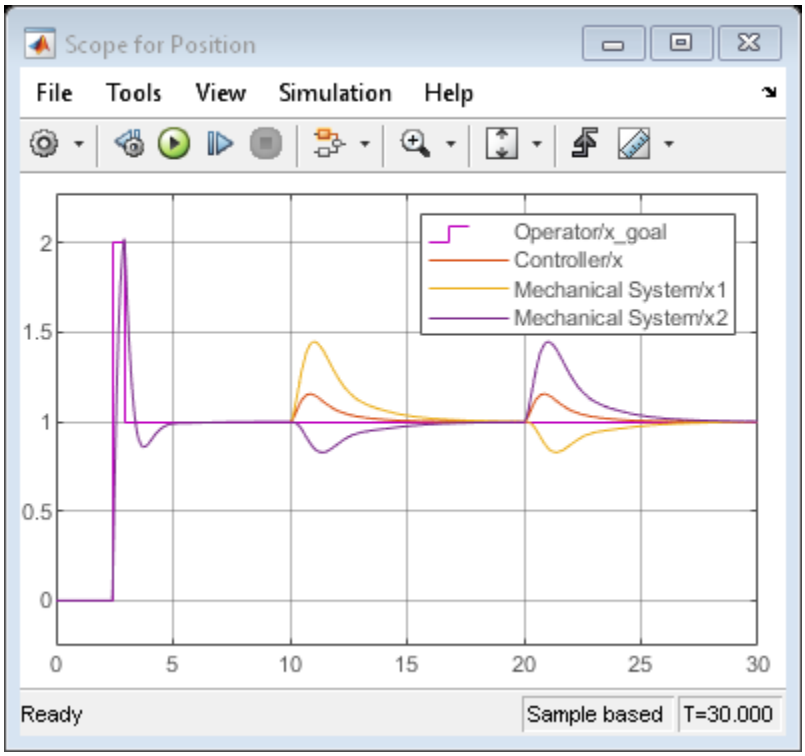
Sensors read the displacement of the masses. The controller regulates the height at the center of the beam x and levels the beam by computing the force the actuators must apply to the masses. The controller uses the height difference between the beam ends, $delta$, to level the beam.

Model `ex_modeling_signals` represents the system.



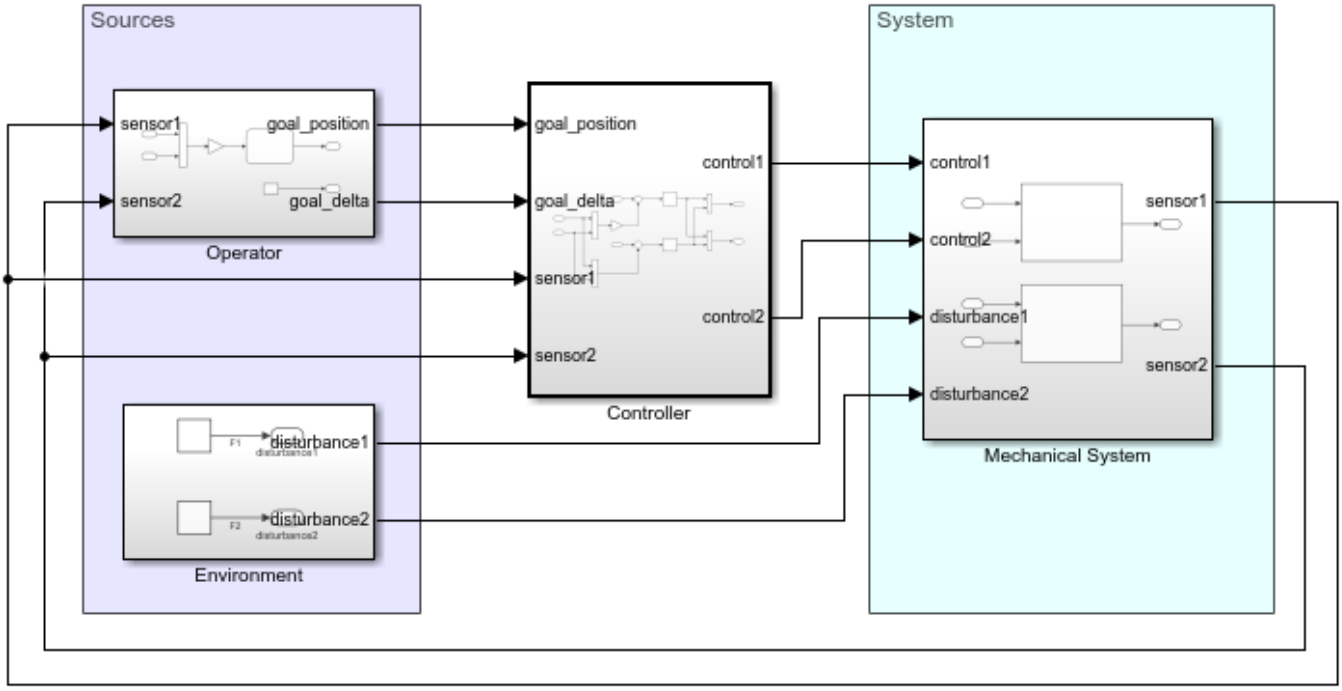
To display signals after simulation, the model uses two Scope blocks. One Scope block shows the goal and actual beam levelness. The other Scope block shows the goal and actual position of the beam at its center, along with the actual beam position at both ends.



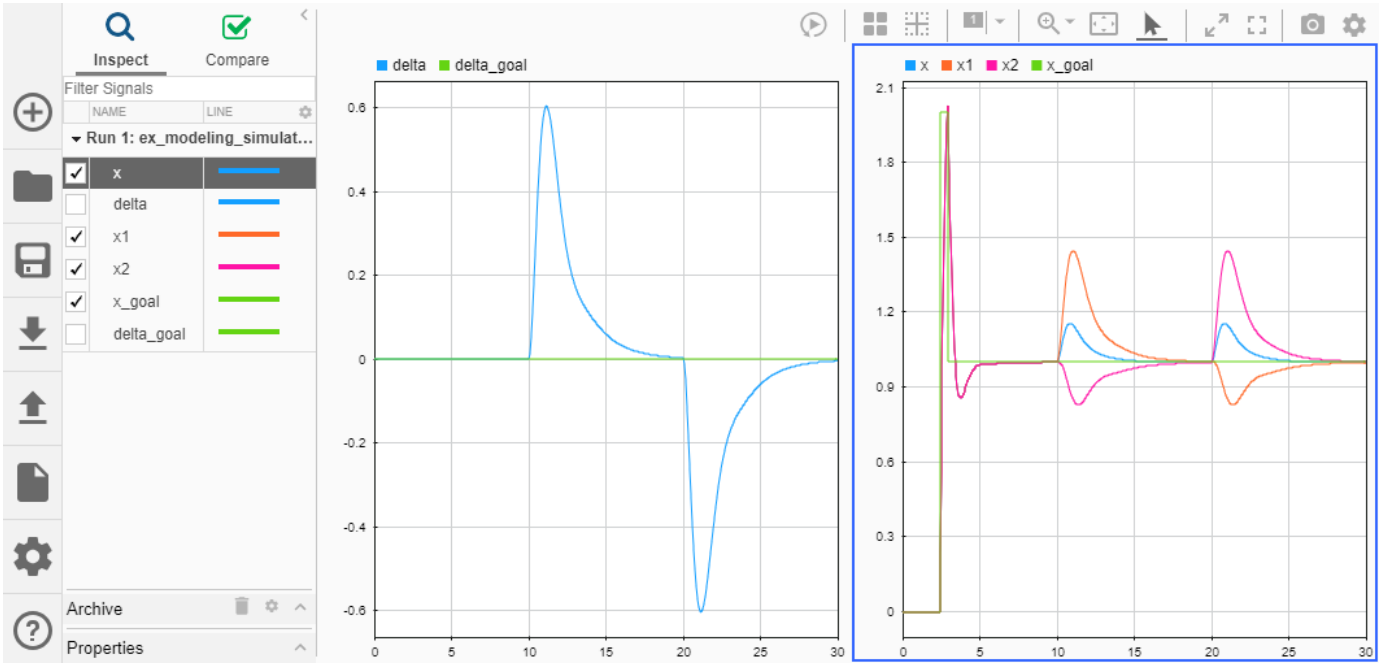


To reduce the number of signal lines, you can connect a viewer directly to signal lines or enable data logging for signal lines. By choosing a way to visualize simulation data without using a sink block, you can avoid extra signal lines.

Model `ex_modeling_simulation_data_inspector` removes the Scope blocks and related signal lines, then enables data logging for those signals.

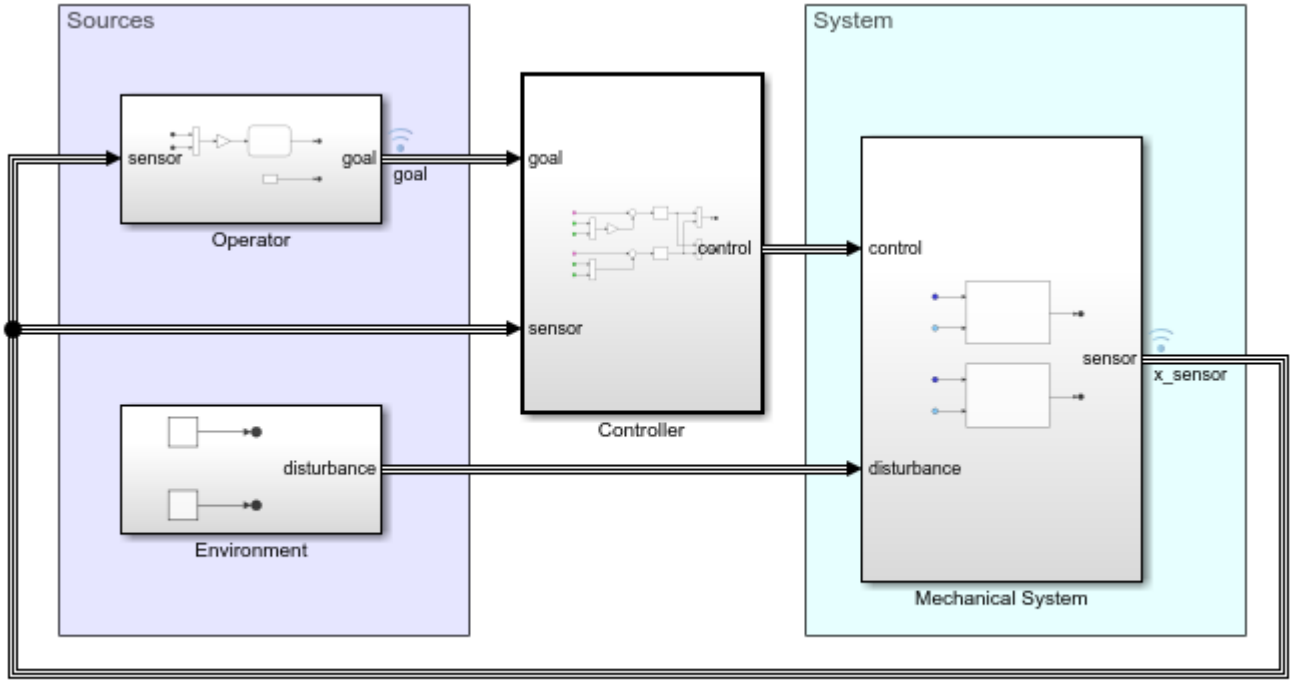


To view the logged signals after simulation, open the Simulation Data Inspector by clicking the highlighted **Simulation Data Inspector** button.



To further reduce the number of signal lines, you can group signal lines into a bus by using a Bus Creator or Out Bus Element block. All signal lines retain their identities when grouped in a bus and can be separated downstream from the bus.

By creating buses, model `ex_modeling_composite_signals` provides an even more readable system representation.



The simulation results remain the same after signals are grouped in buses. This example enables data logging for signal lines associated with buses `x_sensor` and `goal` instead of logging data individually for each of the signals in these buses.

See Also

More About

- “Decide How to Visualize Simulation Data” on page 30-2
- “Scope Blocks and Scope Viewer Overview” on page 28-6
- “Inspect Simulation Data” on page 29-107
- “Types of Composite Signals” on page 76-2
- “Simulink Bus Signals”

Manage Model Data

To define a parameter value that multiple blocks or models use, you can use variables. Separately updating numerical parameter values at each instance of the parameter can be inefficient and error prone. You can update the value of a variable in the workspace or source file that defines it. Having a single source for this information facilitates scalability and reusability of model components.

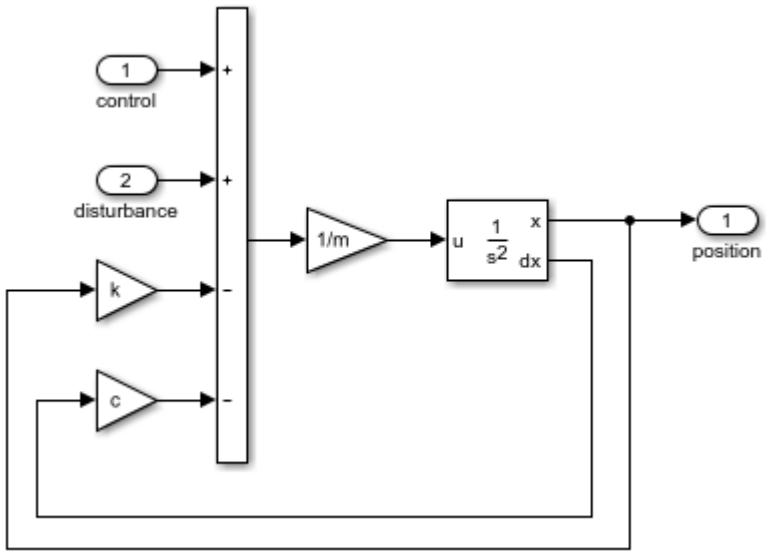
To specify value ranges, data types, tunability, and other characteristics of signals, states, and block parameters, you can use the `Simulink.Parameter` and `Simulink.Signal` objects. While you can use variables or objects to specify parameter values, this example uses variables for simplicity.

You can define variables by using these supporting file types:

- MAT-file
- MATLAB file
- Data dictionary

To load data for small models, you can use model callbacks. For large model hierarchies, different loading methods are more efficient.

In model `ex_modeling_data`, a `PreLoadFcn` model callback evaluates MATLAB file `ex_modeling_data_variables.m`, which defines variables k, c, and m in the base workspace. Gain blocks in the mechanical system and PID Controller blocks in the controller use these variables.



You can interactively change variable values at the MATLAB® command prompt. You can also use tools like the Model Data Editor to edit values.

See Also

More About

- “Create, Edit, and Manage Workspace Variables” on page 67-106

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Data Objects” on page 67-58

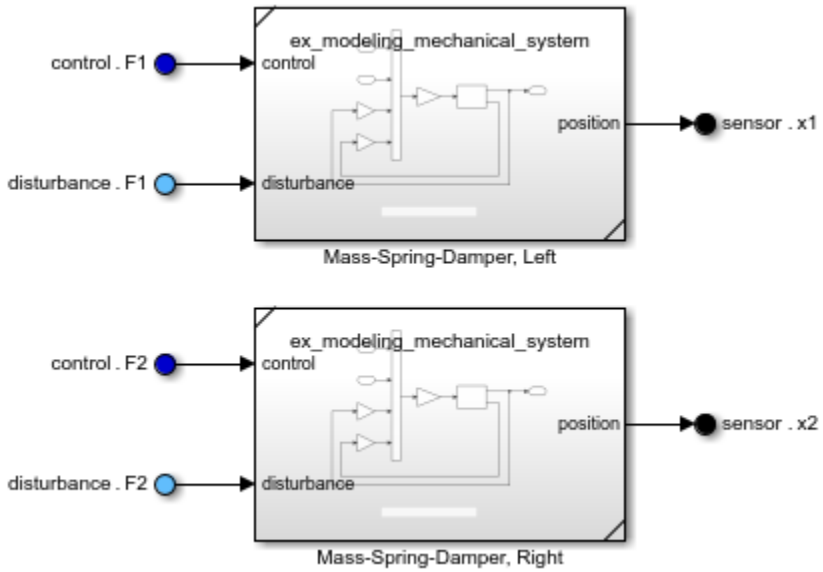
Reuse Model Components from Files

When working on a large model, you can separate it into multiple files so that team members can develop different model components at the same time. You can reuse these components multiple times in a model and in other models.

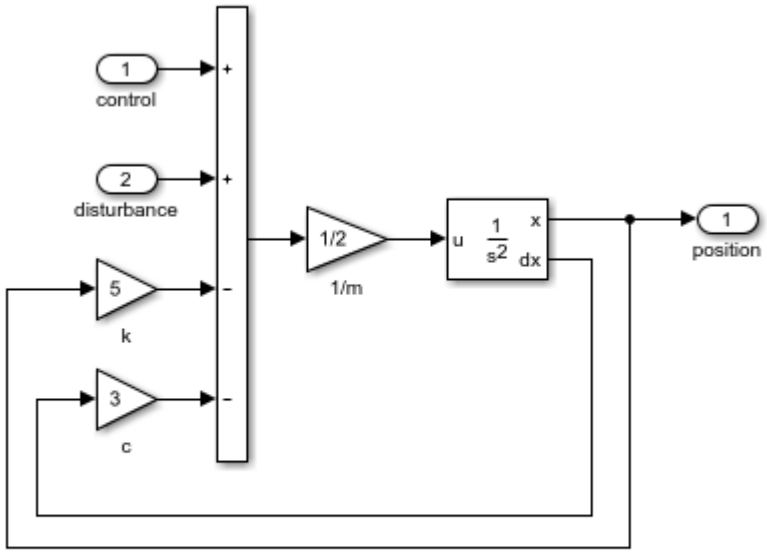
Model, library, and subsystem files provide a single source for multiple instances of the same model component. To learn when you should use each of these componentization methods, see “Component-Based Modeling Guidelines” on page 22-2.

Subsystems

Model `ex_modeling_component_reuse` references the contents of subsystem file `ex_modeling_mechanical_system.slx` twice to represent identical mechanical subsystems.

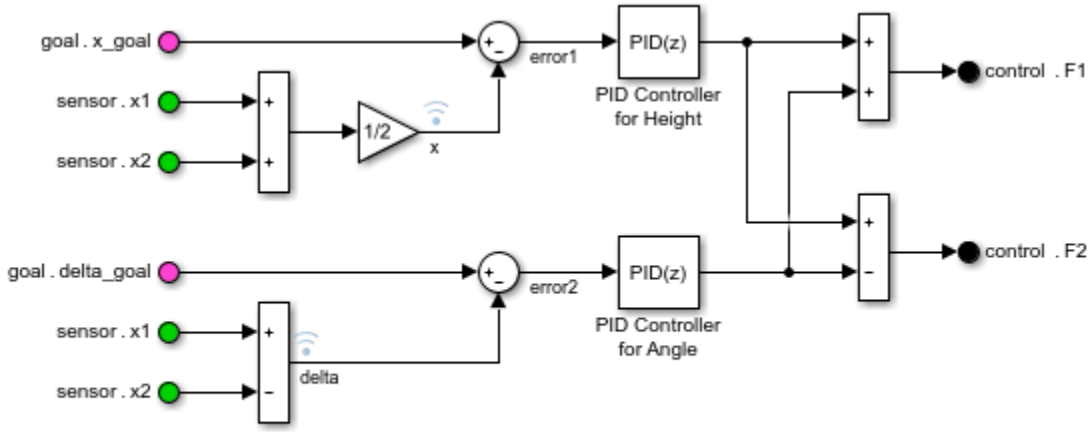


While you can define a subsystem for reuse in either a library or subsystem file, subsystem files allow for easier editing. When you edit a referenced subsystem, the changes apply to the subsystem file and all instances of the referenced subsystem.



Models

Model `ex_modeling_component_reuse` references the contents of model file `ex_modeling_controller.slx`. Controller code is often deployed on embedded systems, so having a standalone controller model is useful.



An embedded processor might not support default properties for the controller. Since a controller model might be used to generate code for an embedded processor, these constraints apply to the referenced controller model and the interface with its parent model:

- Fixed Signal Attributes — To require that buses at model interfaces share the same signal attributes, bus objects specify signal attributes at the three sets of input and output ports.

- Discrete Sample Time — To specify a discrete sample time, model `ex_modeling_controller` specifies a discrete execution domain and script `ex_modeling_data_controller.m` specifies discrete PID controller values.
- Fixed Data Type — To apply the single-precision data type required by the embedded processor, Data Type Conversion blocks convert the bus element data types before they reach the model interface.

See Also

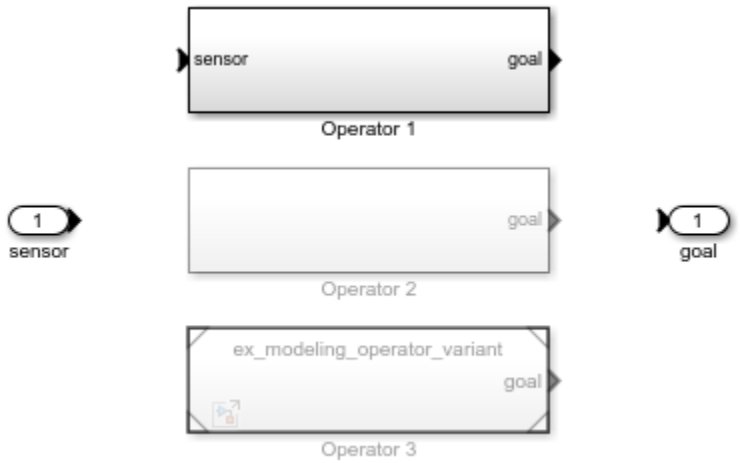
More About

- “Component-Based Modeling Guidelines” on page 22-2
- “Create a Custom Library” on page 41-2
- “Model Reference Basics” on page 8-2

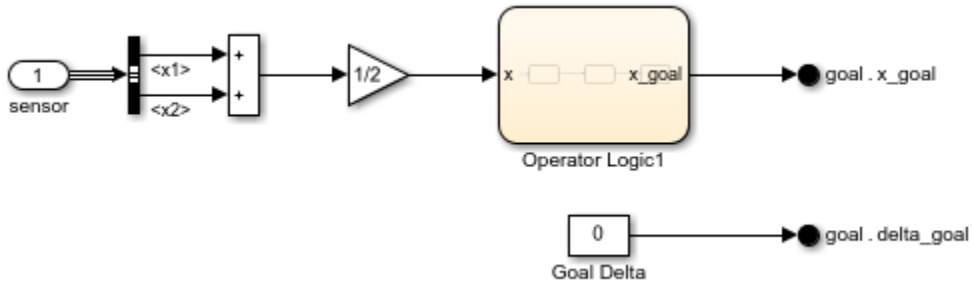
Create Interchangeable Variations of Model Components

To add flexibility to a model so that it can cater to different requirements, you can use variant subsystems and models. Variants allow you to choose among multiple variations of a component within a single model. You can change the active variant without modifying the model by changing the values of variant control variables at the MATLAB® command prompt.

Model `ex_modeling_variants` includes three variant choices for the operator subsystem.



Operator 1 is the active variant, which is defined by script `ex_modeling_variant_choice.m`. To determine the goal position for the mechanical system, this operator implements procedural logic with a Stateflow® chart.



Inactive variants and their contents are grayed out. To define the goal position for the mechanical system position, inactive variant subsystem Operator 2 uses a Waveform Generator block.

Variant choice Operator 3 is an inactive variant model. Variant Subsystem blocks allow both subsystems and models as variant choices.

See Also

More About

- “What Are Variants and When to Use Them” on page 12-2
- “Working with Variant Choices” on page 12-21

Set Up a File Management System

As a model grows, managing referenced files and dependencies becomes more complicated. To reduce the complexity of managing large models, you can use projects. Projects in Simulink help to organize large model hierarchies by finding required files, managing and sharing files and settings, and interacting with source control.

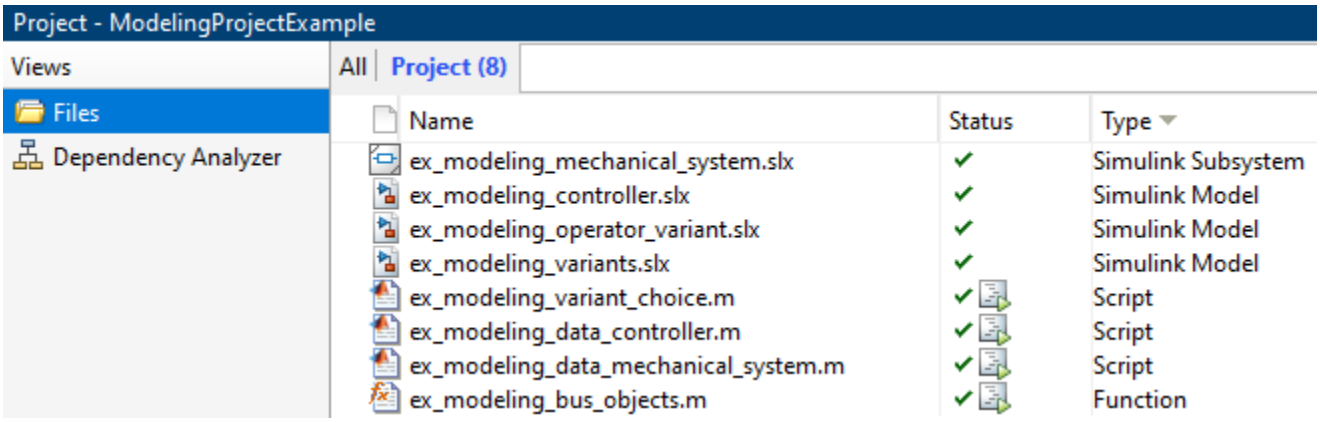
MATLAB® script `ex_modeling_project_setup.m` creates a project that contains these files:

- Subsystem file `ex_modeling_mechanical_system.slx`
- Model file `ex_modeling_variants.slx`
- Model file `ex_modeling_controller.slx`
- Model file `ex_modeling_operator_variant.slx`
- MATLAB script `ex_modeling_data_mechanical_system.m`
- MATLAB script `ex_modeling_data_controller.m`
- MATLAB script `ex_modeling_variant_choice.m`
- MATLAB function `ex_modeling_bus_objects.m`

To open this script, enter this command in the MATLAB command prompt:

```
openExample('simulink/SetupAFileManagementSystemExample')
```

Run the script to create the project.



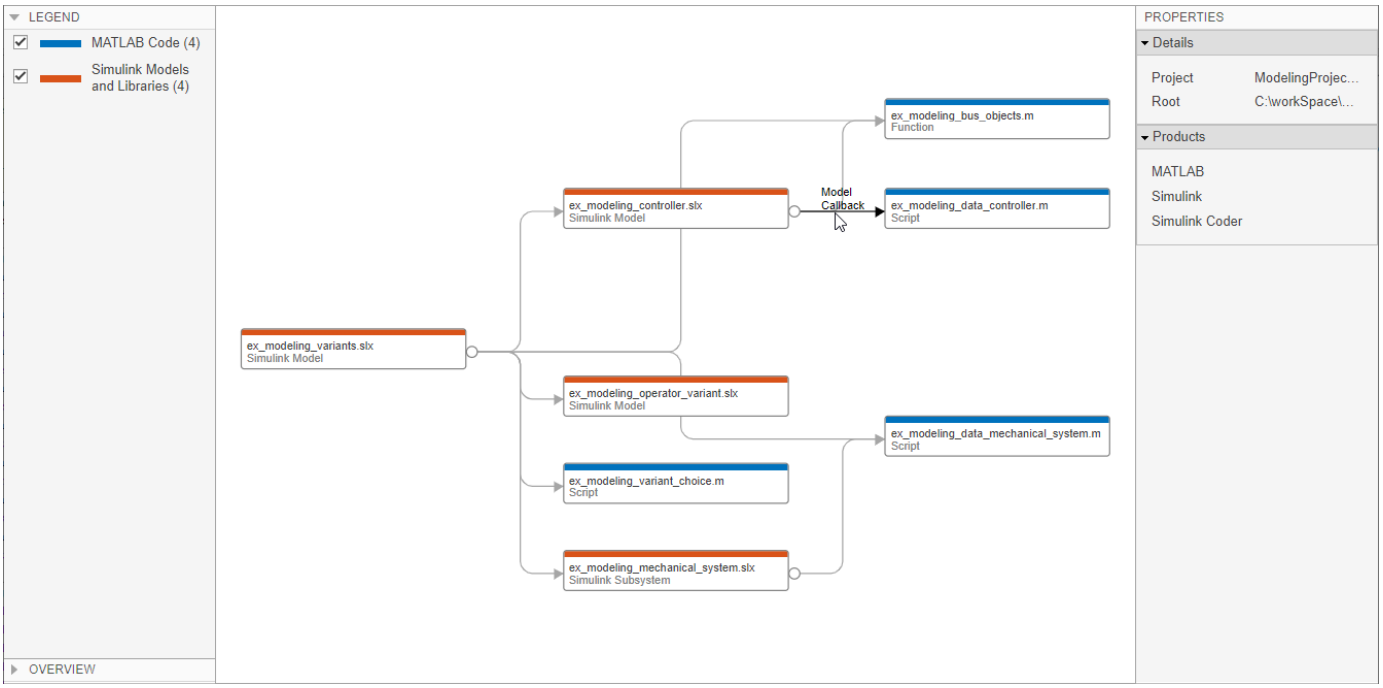
The MATLAB scripts and function are configured to **Run at Startup**.

Using this project, you can explore project capabilities, such as these capabilities:

- Automate tasks
- Create shortcuts for common actions
- Analyze file dependencies
- Analyze the impact of changing files
- Compare model files

In the Project **Views**, select **Dependency Analyzer** and click **Analyze** to run a dependency analysis on all the files in your project. In the dependency graph, hover over the dependency arrows to find

the dependency type. It shows that the MATLAB scripts and functions are being run by model callbacks.



Since these files now run at startup, the model callbacks are redundant and can be removed.

The dependency graph also shows the two-level model hierarchy, in which the top model depends on a library and referenced model.

See Also

More About

- “What Are Projects?” on page 16-3
- “Automate Startup Tasks” on page 16-26
- “What Is Dependency Analysis?” on page 18-2
- “About Source Control with Projects” on page 19-2
- “About Simulink Model Comparison” on page 21-2
- “Perform an Impact Analysis” on page 18-17

Project Setup

Organize Large Modeling Projects

You can use projects to help you organize your work. To get started with managing your files in a project:

1. Try an example project to see how the tools can help you organize your work. See “Explore Project Tools with the Airframe Project” on page 16-5.
2. Create a new project. See “Create a New Project From a Folder” on page 16-14.
3. Use the Dependency Analyzer to analyze your project and check required files. See “Run a Dependency Analysis” on page 18-7.
4. Explore views of your files. See “Work with Project Files” on page 17-7.
5. Create shortcuts to save and run frequent tasks. See “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31.
6. Run custom task operations on batches of files. See “Run a Project Custom Task and Publish Report” on page 17-28.
7. If you use a source control integration, you can use the **Modified** files view to review changes, compare revisions, and commit modified files. If you want to use source control with your project, see “About Source Control with Projects” on page 19-2.

For guidelines on structuring projects, see “Component-Based Modeling Guidelines” on page 22-2.

See Also

More About

- “Choose Among Types of Model Components” on page 22-4
- “Compare Capabilities of Model Components” on page 22-8
- “Define Interfaces of Model Components” on page 22-17
- “Configuration Management” on page 22-21

What Are Projects?

You can use projects to help you organize your work. Find all your required files; manage and share files, settings, and user-defined tasks; and interact with source control.

If your work involves any of the following:

- More than one model file
- More than one model developer
- More than one model version

— then a project can help you organize your work. You can manage all the files you need in one place — all MATLAB and Simulink files, and any other file types you need such as data, requirements, reports, spreadsheets, tests, or generated files.

Projects can promote more efficient team work and individual productivity by helping you:

- Find all the files that belong with your project.
- Create standard ways to initialize and shut down a project.
- Create, store, and easily access common operations.
- View and label modified files for peer review workflows.
- Share projects using built-in integration with Subversion® (SVN) or Git, external source control tools.

Starting in R2019a, you can use projects in MATLAB, with or without Simulink. You can share projects with users who do not have Simulink.

For information on basic project workflows in MATLAB, see “Projects”.

Projects provide additional tools to help with Simulink workflows. For example:

- Opening models and running customizations on startup
- Checking for shadowed model files
- Dependency analysis of models, subsystems, libraries and library blocks, data files, requirements, and generated code
- Automatic refactoring help for models, libraries, library links, model references, model callbacks, S-functions, buses and bus elements
- Comparing and merging differences in models.

For help on project workflows in Simulink, see “Project Management”.

See the Web page `https://www.mathworks.com/products/simulink/projects.html` for the latest information, downloads, and videos.

To get started with managing your files in a project:

1. Try an example project to see how the tools can help you organize your work. See “Explore Project Tools with the Airframe Project” on page 16-5.
2. Create a new project. See “Create a New Project From a Folder” on page 16-14.
3. Analyze your project and check required files by using the Dependency Analyzer. See “Run a Dependency Analysis” on page 18-7.

4. Explore views of your files. See “Work with Project Files” on page 17-7.
5. Create shortcuts to save and run frequent tasks. See “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31.
6. Run custom task operations on batches of files. See “Run a Project Custom Task and Publish Report” on page 17-28.
7. If you use a source control integration, you can use the **Modified** files view to review changes, compare revisions, and commit modified files. If you want to use source control with your project, see “About Source Control with Projects” on page 19-2.

For guidelines on structuring projects, see “Large-Scale Modeling”.

See Also

More About

- “Component-Based Modeling Guidelines” on page 22-2
- “Define Interfaces of Model Components” on page 22-17
- “Configuration Management” on page 22-21

Explore Project Tools with the Airframe Project

In this section...

Explore the Airframe Project

Try an example project to see how the tools can help you organize your work. Projects can help you manage:

- Your design (model and library files, `.m`, `.mat`, and other files, source code for S-functions, and data)
- A set of actions to use with your project (run setup code, open models, simulate, build, and run shutdown code)
- Working with files under source control (check out, compare revisions, tag or label, and check in)

The Airframe example shows how to:

1. Set up and browse some example project files under source control.
2. Examine project shortcuts to access frequently used files and tasks.
3. Analyze dependencies in the example project and locate required files that are not yet in the project.
4. Modify some project files, find and review modified files, compare to an ancestor version, and commit modified files to source control.
5. Explore views of project files only, modified files, and all files under the project root folder.

Set Up Project Files and Open the Project

Run this command to create a working copy of the project files and open the project:

```
sldemo_slproject_airframe
```

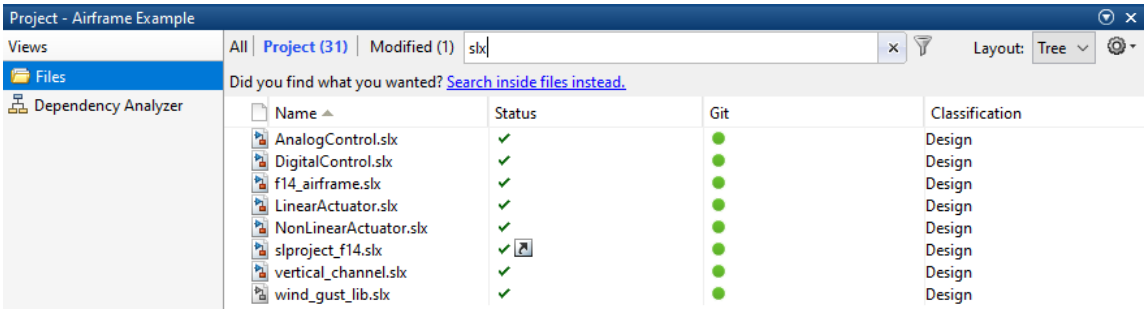
The project example copies files to your temporary folder so that you can edit them and put them under Git source control.

The Project window opens and loads the project. The project is configured to run some startup tasks, including changing the current working folder to the project root folder.

Note Alternatively, you can try this example project using SVN source control, by specifying `sldemo_slproject_airframe_svn`. The following example shows the options when using Git.

View, Search, and Sort Project Files

1. In a Project, examine the **Files** view to manage the files within your project. When the **Project (*number of files*)** view is selected, only the files in your project are shown.
2. To see all the files in your sandbox, click **All**. This view shows all the files that are under the project root, not just the files that are in the project. This view is useful for adding files to the project from your sandbox.
3. To find particular files or file types, in any file view, type in the search box or click the Filter button. You can also search inside files.



   Click the **x** to clear the search.
4. To view files as a list instead of a tree, use the Layout control.
5. To sort files and to customize the columns, click the Organize view button at the far right of the search box.
6. You can dock and undock the Project into the MATLAB Desktop. If you want to maximize space for viewing your project files, undock the Project. Drag the title bar to undock it.

Open and Run Frequently Used Files

You can use shortcuts to make scripts easier to find in a large project. View and run shortcuts on the Project Shortcuts toolstrip. You can organize the shortcuts into groups.

In this example, the script that regenerates S-functions is set as a shortcut so that a new user of the project can easily find it. You can also make the top-level model, or models, within a project easier to find. In this example, the top-level model, `slproject_f14.mdl`, is a shortcut.

Regenerate the S-functions.

1. On the Project Shortcuts tab in the toolstrip, click the shortcut **Rebuild Project's S-functions**.

   The shortcut file builds a MEX-file. If you do not have a compiler set up, follow the instructions to choose a compiler.
2. Open the `rebuild_s_functions.m` file to explore how it works.

Open the top model.

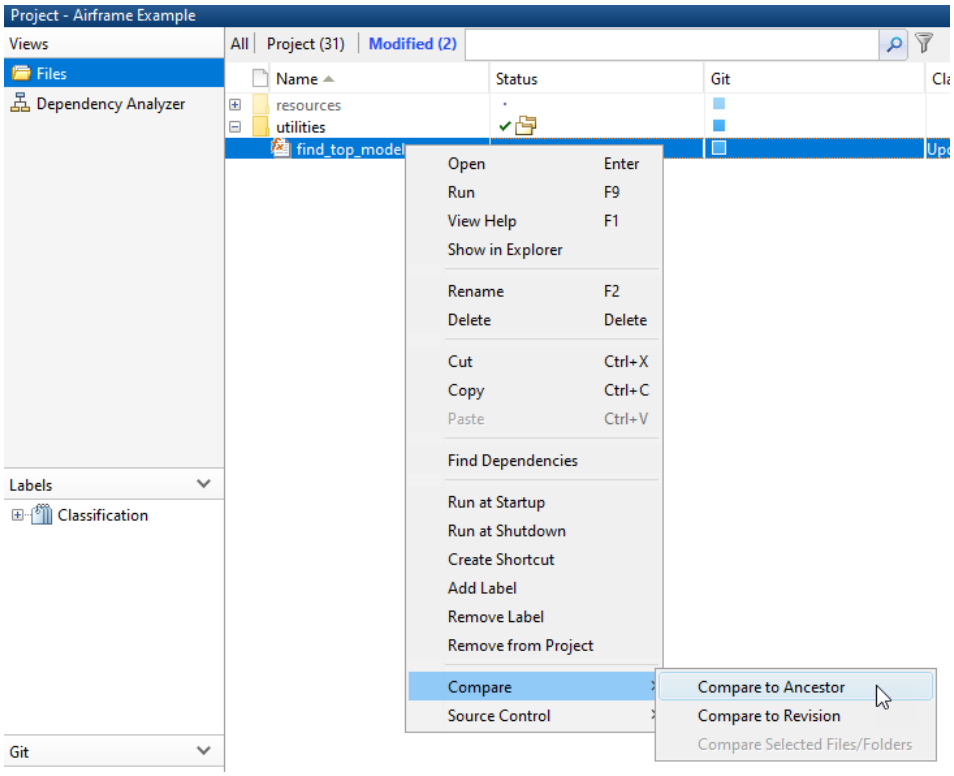
- On the Project Shortcuts tab, click the shortcut **F14 Model** to open the root model for this project.
- To create shortcuts to access frequently used files, select the **Files** view, right-click a file, and select **Create Shortcut**.

You can also specify files to run at startup and shutdown. See “Automate Startup Tasks” on page 16-26.

Review Changes in Modified Files

Open and make changes to files and review changes.

1. Select the Files view. View folders using the tree layout, and then expand the `utilities` folder.
2. Either double-click to open the `find_top_models` file for editing from the project, or right-click and select **Open**.
3. Make a change in the Editor, such as adding a comment, and save the file.
4. In the project Files view, select the tab **Modified (*number of files*)**. After editing the file, you see **Modified (2)**. The files you changed appear in the list. You edited a file in the utilities folder. Observe that the Modified files list also includes a `resources` folder. The files stored in the `resources` folder are internal project definition files generated by your changes. The project definition files allow you to add metadata to files, for example, by creating shortcuts, adding labels, and adding a project description. Project definition files also define the files that are added to your project. You can review changes in revisions of project definition files like any other project files. See “Project Definition Files” on page 19-36.
5. To review changes, right-click the `find_top_models` file in the **Modified** files view and select **Compare > Compare to Ancestor**.



The MATLAB Comparison Tool opens a report comparing the modified version of the file in your sandbox against its ancestor stored in the version control tool. The comparison report type can differ depending on the file you select. If you select a Simulink model to **Compare > Compare to Ancestor**, this command runs a Simulink model comparison.

To compare models, try the following example.

1. In the Files view, select the **Project (*number of files*)** tab, and expand the `models` folder.
2. Either double-click to open the `AnalogControl` file for editing from the project, or right-click and select **Open**.
3. Make a change in the model, such as opening a block and changing some parameters, and then save the model.
4. To review changes, select the **Modified (*number of files*)** tab. Right-click the modified model file and select **Compare > Compare to Ancestor**.

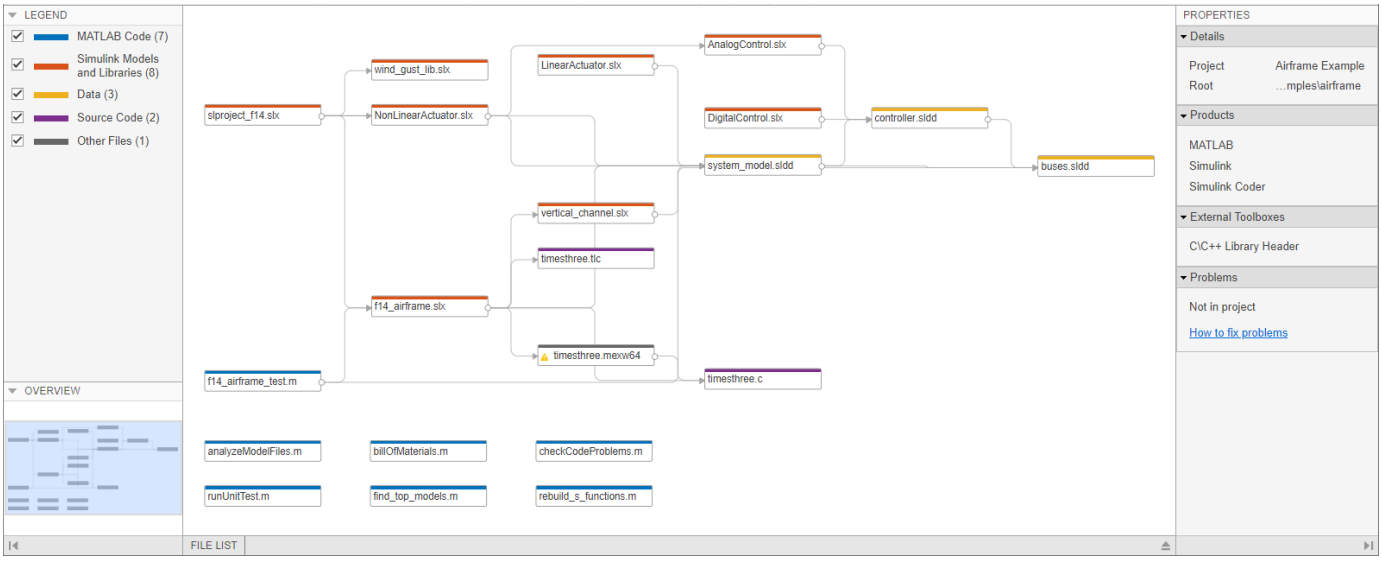
   The Comparison Tool opens a report.

Run Dependency Analysis

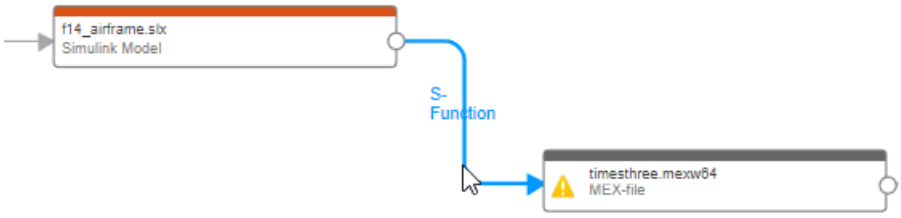
To check that all required files are in the project, run a file dependency analysis on the modified files in your project.

1. In the **Project** tab, in the **Tools** section, click **Dependency Analyzer**.

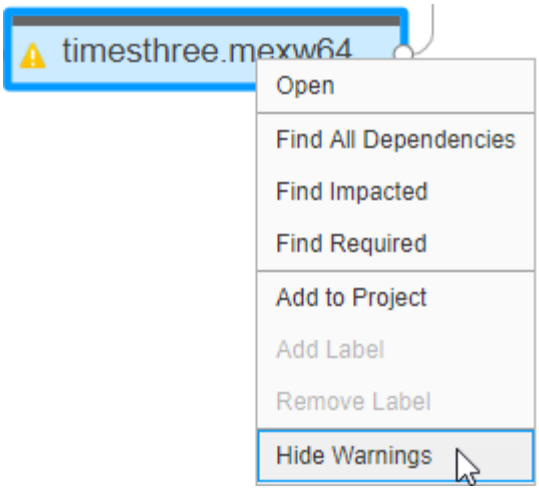
   The dependency graph displays the structure of all analyzed dependencies in the project. The **Properties** pane lists required toolboxes and any problem files.



2. To highlight the problem files, in the **Properties** pane, in the **Problems** section, point to the message **Not in Project** and click the magnifying glass icon  .
3. Select the dependency arrow to examine the dependency type. `timesthree.mexw64` is an S-function binary file required by `f14_airframe.slx`. You can add binary files to your project or, as in this project, provide a utility script that regenerates them from the source code that is part of the project.



4 To remove the file from the problem files list, right-click the file and select **Hide Warnings**. The next time you run a dependency analysis, the file does not appear as a problem file.



In this example, you do not want to add the binary file to the project, but instead use the script to regenerate the binary file from the source code in the project. Use **Hide Warnings** to stop such files being marked as problems.

5 View dependencies of the modified files.

- **a** In the Dependency Analyzer toolstrip, in the **Views** section, click **Source Control**. The color of each file in the graph now represents its source control status.
- **b** In the dependency graph, select the modified files. To select multiple files, press **Shift** and click the files.
- **c** In the Dependency Analyzer toolstrip, in the **Impact Analysis** section, click **All Dependencies**.

Run Project Integrity Checks

To make sure that your changes are ready to commit, check your project. On the Project tab in the toolstrip, click **Run Checks** to run the project integrity checks. The checks look for missing files, files to add to source control or retrieve from source control, and other issues. The checks dialog box can offer automatic fixes to problems found. When you click a **Details** button in the Checks dialog box, you can view recommended actions and decide whether to make the changes.

For an example using the project checks to fix issues, see “Convert from MDL to SLX in a Project and Preserve Revision History”.

Commit Modified Files

After you modify files and you are satisfied with the results of the checks, you can commit your changes to the source control repository.

1. In the Files view, select the **Modified (*number of files*)** tab. The files you changed appear in the list.
2. To commit your changes to source control, on the Project tab, in the Source Control section, click **Commit**.
3. Enter a comment for your submission, and click **Submit**.

   Watch the messages in the status bar as the source control commits your changes. Git commits to your local repository. To commit to the remote repository, use **Push** in the Source Control section. See “Pull, Push, and Fetch Files with Git” on page 19-46

View Project and Source Control Information

- To view and edit project details, on the **Project** tab, in the **Environment** section, click **Details**. View and edit details such as the name, description, project root, startup folder, and generated files folders such as the `slprj` folder.
- To view details about the source control integration and repository location, on the Project tab, in the Source Control section, click **Git Details**. This Airframe example project uses Git source control.

Alternatively, use the project API to get the current project:

```
project = currentProject;
```

You can use the project API to get all the project details and manipulate the project at the command line. See `currentProject`.

For next steps, see “Project Management”.

See Also

Related Examples

- “Create a New Project From a Folder” on page 16-14
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Automate Startup Tasks” on page 16-26
- “Perform an Impact Analysis” on page 18-17
- “Add a Project to Source Control” on page 19-5
- “View Modified Files” on page 19-36

More About

- “What Are Projects?” on page 16-3
- “What Can You Do With Project Shortcuts?” on page 16-25
- “What Is Dependency Analysis?” on page 18-2

- “Sharing Projects” on page 17-30
- “About Source Control with Projects” on page 19-2

Create a Project from a Model

Create a project to organize your model and any dependent files. Use **Create Project from Model** to run a dependency analysis on your top model to identify required files.

Tip For a simpler option that automates more steps for you, see instead “Create a New Project From a Folder” on page 16-14.

Projects can help you organize your work and collaborate in teams. The project can help you to:

- Find all your required files
- Manage and share files, settings, and user-defined tasks
- Interact with source control.

1. In a Simulink model, on the **Simulation** tab, select **New > Project > New Project from this Model**.

   Simulink runs dependency analysis on your model to identify required files and a project root location that contains all dependencies.
2. In the New Project dialog box, edit any settings you want to change:
   - **Project name** — By default, the name of the suggested project root folder. Edit if desired.
   - **Project folder** — A folder that dependency analysis identified to contain all dependencies. If you want, click to select a different folder in the file system hierarchy between the file system root and the model folder.
   - **Files to include** — Files to include in the project. Files with selected check boxes are identified by dependency analysis. Select check boxes to specify all the files you want to include.

     Any external dependencies are listed. If required files are outside your project root, then you cannot add these files to your project. An external dependency might not indicate a problem if the file is on your path and is a utility or other resource that is not part of your project.
   - If you do not want to make a shortcut to the top-level file, or add all the folders to the project path, under **More Options**, clear the check boxes. Alternatively, you can edit these project settings later.
3. Click **Create** to create the project containing your model and any other specified files.

For an example showing what you can do with projects, see “Explore Project Tools with the Airframe Project” on page 16-5.

See Also

Related Examples

- “Create a New Project From a Folder” on page 16-14
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Automate Startup Tasks” on page 16-26

- “Open Recent Projects” on page 16-22
- “Explore Project Tools with the Airframe Project” on page 16-5

More About

- “What Are Projects?” on page 16-3
- “What Can You Do With Project Shortcuts?” on page 16-25
- “Sharing Projects” on page 17-30
- “About Source Control with Projects” on page 19-2

Create a New Project From a Folder

If you have many files that you want to organize into a project, with or without source control, use the following steps to create a new project.

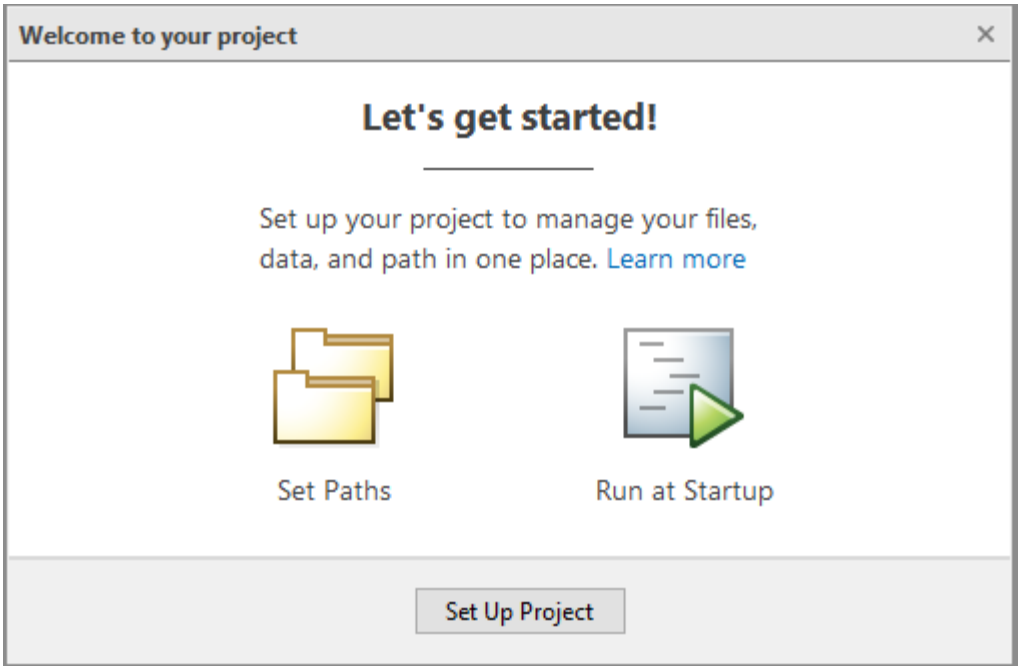
Note If you want to retrieve a project from a source control repository, see instead “Clone Git Repository” on page 19-25 or “Check Out SVN Repository” on page 19-27.

Easily convert a folder of files into a project by using the **Folder to Project** template in the Simulink start page. The template automatically adds your files to the project and prompts you to set up the path and startup files. This simple, quick process sets up your project to manage your files and introduces you to project features. When you open the project, it automatically puts the folders you need on the path, and runs any setup operations to configure your environment. Startup files automatically run (.m and .p files), load (.mat files), and open (Simulink models) when you open the project.

To create a new project to manage your files:

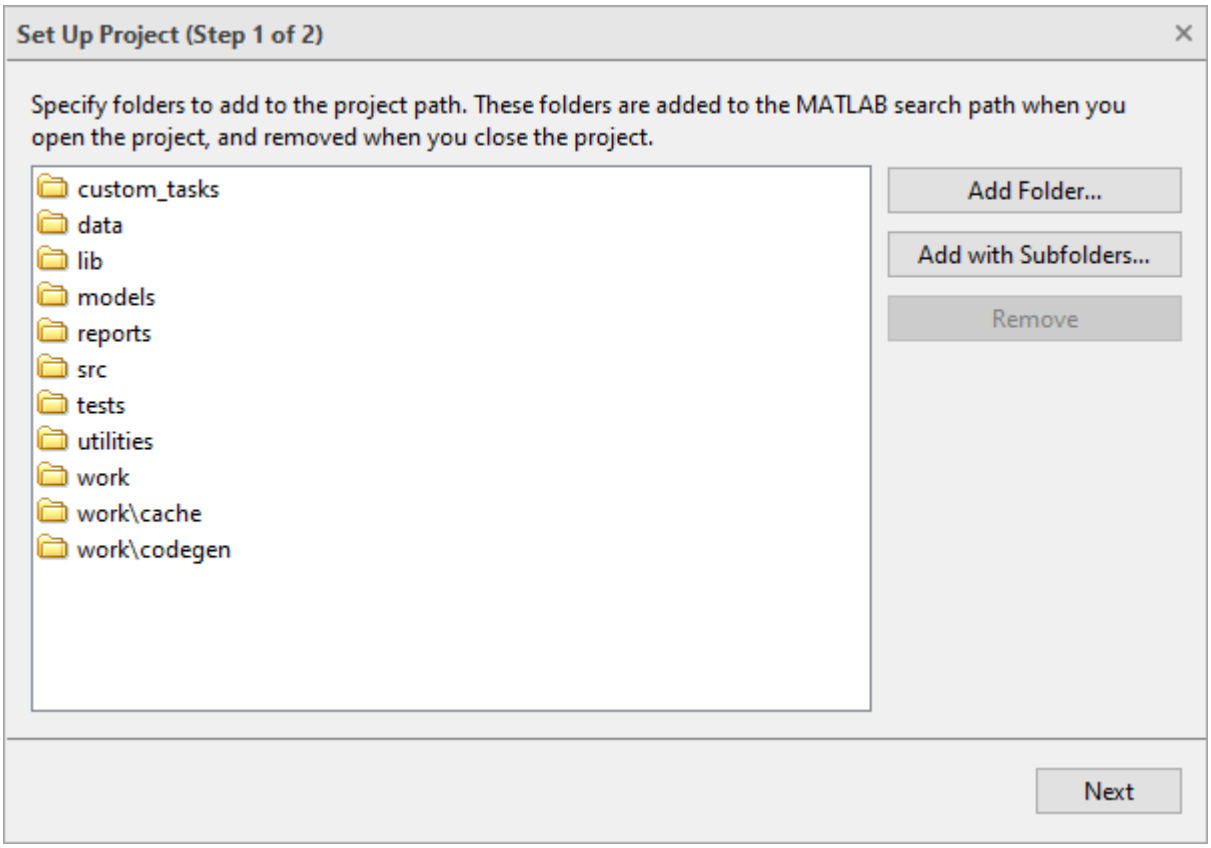
- On the MATLAB **Home** tab, select **New > Project > From Folder**.
- Alternatively, in the Simulink start page, click the **Folder to Project** template. You can also, on the **Simulation** tab, select **New > Project > New Project** from the Model Editor.

1. In the New Project dialog box, enter a project name, browse to select the folder containing your files, and click **OK**.
2. In the Welcome to your project dialog box, click **Set Up Project** to continue.



3. In the Set Up Project (Step 1 of 2) dialog box, optionally choose folders to add to the project path. When you open the project, it adds these folders to your MATLAB search path, and removes them when you close the project. Add project folders to ensure dependency analysis detects

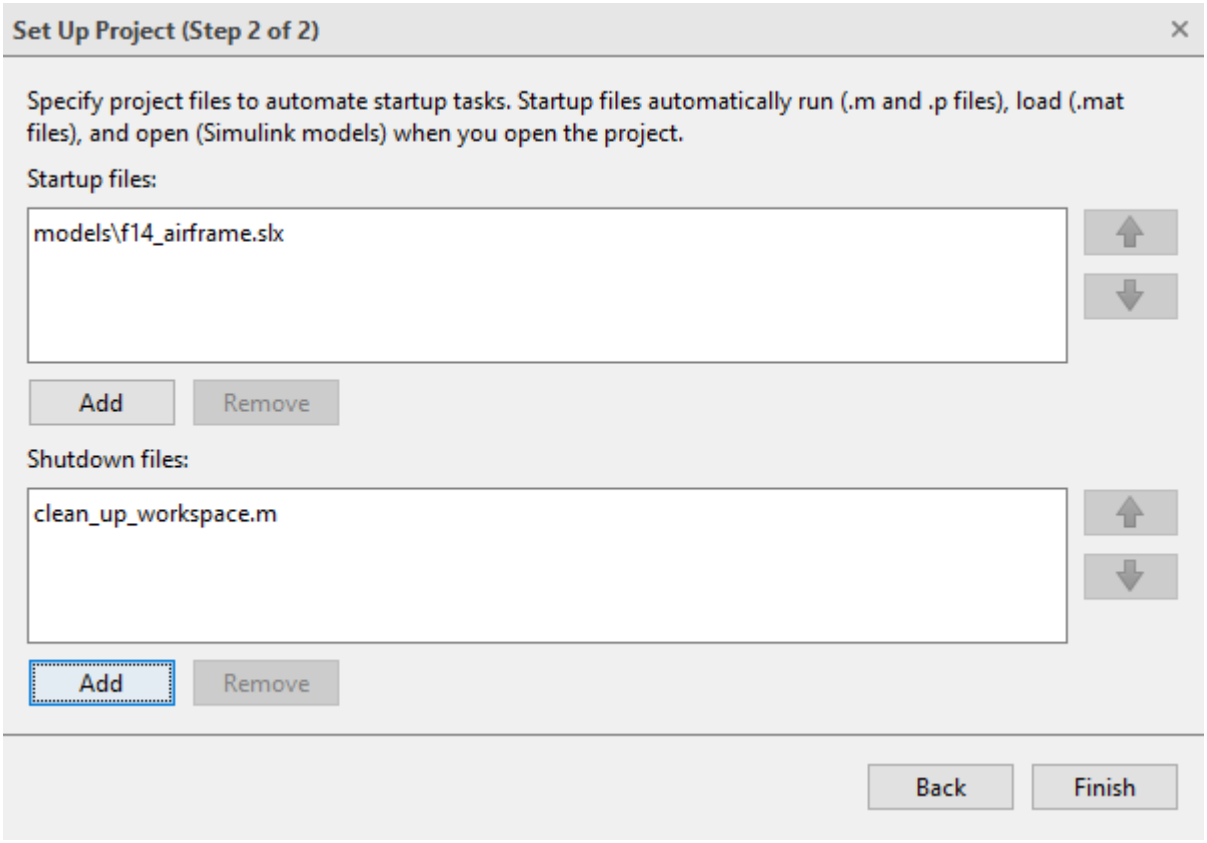
project files. To add all project folders, click **Add with Subfolders** and then select the project folder containing all your subfolders. Alternatively, you can set up the project path later. Click **Next**.



4 In the Set Up Project (Step 2 of 2) dialog box, optionally specify startup and shutdown files.

- Use startup files to configure settings when you open the project. Startup files automatically run (`.m` and `.p` files), load (.mat files), or open (Simulink models) when you open the project.
- Use shutdown files to specify MATLAB code to run as the project shuts down. You do not need to use shutdown files to close models when you close a project, because it automatically closes any project models that are open, unless they are dirty. The project prompts you to save or discard changes.

Click **Add** to specify startup or shutdown files. Alternatively, you can setup these files later.



5 Click **Finish** and your new project opens. The **Folder to Project** template automatically adds all your files to the project. The template does not add derived files to your project.

For next steps, try dependency analysis to visualize the structure of your project, or consider adding source control to help you manage versions. For details, see “Run a Dependency Analysis” on page 18-7 or “Add a Project to Source Control” on page 19-5.

As alternatives to the **Folder to Project** template, you can:

- Create a project from a model by analyzing it for dependent files that you want to put in a project. See “Create a Project from a Model” on page 16-12
- Create projects manually, but then you need to add files to the project and configure the path, startup, and shutdown files. To examine the alternative templates, or to use your own templates:

  1 In the Simulink start page, click templates in the list to read the descriptions. If you selected a new project option to open the start page, the list shows only project templates, or you can filter the list for Project Templates using the list next to the Search box.

  - Select the `Blank Project` template if you are creating a project in a folder with existing files and want to set up the project manually. The `Blank Project` template creates a project in your selected folder and leaves any other files untouched. You must manually set up the project, for example by adding files to the project, configuring startup files, configuring the project path, etc.
  - Try the `Simple Project` template if you are creating a project in a new folder and want a blank model. The `Simple Project` template creates a simple project containing a blank model and some utilities. The model is a shortcut so you can open it from the toolstrip. The

project manages your path and the location of the temporary files (slprj folder) when you open and close the project. You can modify any of these files, folders, and settings later.

- You can create your own templates. See “Using Templates to Create Standard Project Settings” on page 16-32.

2. In the start page, select a template and click **Create Project**.
3. In the Create Project dialog box, specify your project folder and edit the project name, and click **Create Project**. You can control the default folder for new projects using the project preferences.

   The Project displays the project files view for the specified project root. You need to add files to the project. See “Add Files to the Project” on page 16-18.

For next steps using your new project, try dependency analysis to visualize the structure of your files.

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Create a Project from a Model” on page 16-12
- “Add Files to the Project” on page 16-18
- “Work with Project Files” on page 17-7
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Add a Project to Source Control” on page 19-5

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Add Files to the Project

If you create a project from a folder, from a model, or with a Git or SVN template from the start page, then the project setup helps you add initial files to the project. If you create a new blank project, then the project files view is empty and you need to add files to the project.

To display all files in your project folder (or `projectroot`), in the **Files** view, click **All**. You might not want to include all files in your project. For example, you might want to exclude some files under `projectroot` from your project, such as SVN or CVS source control folders.

To add existing files to your project:

1. On the **Project** tab, in the **Tools** section, select **Run Checks** > **Add Files**.
2. Select from the list of unmanaged files in the project folder.

   Use **Add Files** any time you want to check for new files not yet added to the project.

Alternatively, you can use these methods to add files:

- In the **All** files view, select files or folders, right-click, and select **Add to Project** or **Add to Project (including child files)**.
- To add files to your project, cut and paste or drag and drop files from a file browser or the Current Folder browser onto the **Project** files view. If you drag a file from outside the project root, this moves the file and adds it to your project. You can drag files within project root to move them.
- Add and remove project files at the command line using `addFile`.

To create new files or folders in the project, right-click a white space in the **Files** view and select **New Folder** or **New File**. The new files are added to the project.

To learn how to set up your project with all required files, see “Run a Dependency Analysis” on page 18-7.

To add or remove project folders from the MATLAB search path, see “Specify Project Path” on page 16-24.

To configure your project to automatically run startup and shutdown tasks, see “Automate Startup Tasks” on page 16-26.

You can access your recent projects direct from MATLAB. See “Open Recent Projects” on page 16-22.

If you want to add source control, see “Add a Project to Source Control” on page 19-5.

See Also

Related Examples

- “Work with Project Files” on page 17-7
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Add a Project to Source Control” on page 19-5

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Create a New Project from an Archived Project

To create a new project from an archived project:

1. Locate and double-click the `mlproj` file in the Current Folder, Windows Explorer, or Apple Finder.

   If you do not have a product that the project requires, you see a warning with a link to the Add-On Explorer to get missing products.

2. In the Extract Project to dialog box, specify the location of the new project and click **Select Folder**.

   The new project opens. The current working folder is the location of the new project, which is a new subfolder of the extraction folder.

If you have a project archived in a zip file, double click or right click to extract the project. The current working folder, for example, `C:\myNewProject`, contains the imported project folders. If the zip project contains referenced projects, the project imports files into two subfolders, `main` and `refs`. The current working folder, for example, `C:\myNewProject\main` contains the project folders and `C:\myNewProject\refs` contains the referenced project folders. To open the project, navigate into `main` to locate and double-click the `prj` file.

See Also

Related Examples

- “Archive Projects” on page 17-34

More About

- “Sharing Projects” on page 17-30

Create a New Project Using Templates

In a project, you can use templates to create and reuse a standard project structure.

1. To browse for templates, click **Simulink** on the MATLAB Home tab, or on the Project tab, click **New**.
2. In the Simulink start page, click a template in the list to read the description. For example, click `Simple Project`.
3. The start page shows all project templates (`*.sltx`) on the MATLAB path. If your templates do not appear, locate them by clicking **Open**. In the Open dialog box, make `*.sltx` files visible by changing the file type list `Model Files` to `All MATLAB files`, and browse to your template.
4. In the start page, select a template and click **Create Project**.

   Templates created in R2017b or later warn you if required products are missing. Click the links to open Add-On Explorer and install required products.
5. In the Create Project dialog box, specify your project folder and edit the project name, and click **Create Project**.

Use Project Templates from R2014a or Before

To use project templates created in R2014a or earlier (`.zip` files), upgrade them to `.sltx` files using `Simulink.exportToTemplate`.

After you upgrade the templates to `.sltx` and put them on the MATLAB path, you can use the templates from the start page.

See Also

Related Examples

- “Create a New Project From a Folder” on page 16-14
- “Create Templates for Standard Project Settings” on page 16-32
- “Create a Template from a Project Under Version Control” on page 16-33
- “Edit a Template” on page 16-33
- “Create a Template from a Model” on page 4-2
- “Clone Git Repository” on page 19-25

More About

- “Using Templates to Create Standard Project Settings” on page 16-32

Open Recent Projects

You can use any of these methods to open recent projects:

- On the MATLAB **Home** tab, click **Simulink**, and select your project in the recent list.

  If you select a recent model that is part of a project, you can choose to also open the project.
- On the MATLAB **Home** tab, click the **Open** arrow and select your project under the **Recent Projects** list.
- From the Current Folder browser, double-click the `.prj` file.
- In the Simulink Editor, if an open model, library, or chart belongs to a project, you can, on the **Simulation** tab, select **Project > View Project**.

If you already have an open project, to load another project, click on the **Project** tab and open a recent project with these methods:

- Click the **Open** arrow and select your project under the **Recent** list.
- Select **Open > Open**. Browse and select your project `.prj` file.

Note You can have one project open at a time, to avoid conflicts. If you open another project, any currently open project closes.

When you open a project, you are prompted if loaded files shadow your project model files. To avoid working on the wrong files, close the shadowing files. See “Manage Shadowed and Dirty Models and Other Project Files” on page 17-8.

See Also

Related Examples

- “Work with Project Files” on page 17-7

Specify Project Details, Startup Folder, and Derived Files Folders

On the Project tab, in the Environment section, click **Details**. Use the Project Details dialog box for the following tasks:

- Edit the project name or add a description.
- View the **Project root** folder. You can change your project root by moving your entire project on your file system, and reopening your project in its new location. All project file paths are stored as relative paths. To change the current working folder to your project root, click **Set as Current Folder**.
- View or edit the **Start Up** folder. By default, this is set to the project root. When you open the project, the current working folder changes to the project root folder. You can specify a different startup folder or click **Clear**.

  You can also configure startup scripts that set the current folder and perform other setup tasks. If you configure startup files to set the current folder, your startup setting takes precedence over the startup folder at the Project Details dialog box. To set up startup files, see “Automate Startup Tasks” on page 16-26.
- View or edit the **Generated Files** folders. You can set the **Simulation cache folder** and **Code generation folder**. For details, see “Manage Build Process Folders” (Simulink Coder).
- If you edit any project details, then click **OK** to save your changes.

If you are looking for source control information for your project, see instead the details button for your source control in the Source Control section of the Project tab, e.g., **SVN Details**. See “Add a Project to Source Control” on page 19-5.

See Also

Related Examples

- “Automate Startup Tasks” on page 16-26
- “Add a Project to Source Control” on page 19-5

Specify Project Path

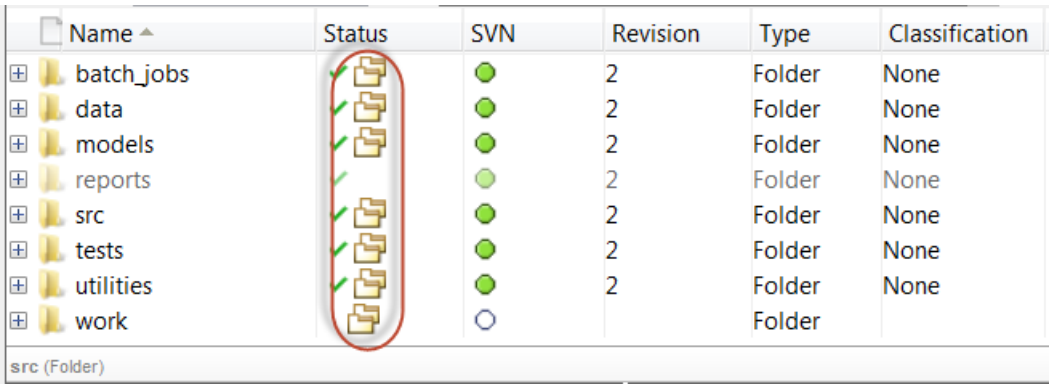
- When you open your project, it adds the project path to the MATLAB search path before applying startup shortcuts.
- When you close your project, it removes the project path from the MATLAB search path after applying shutdown shortcuts.

You can add or remove folders from the project path. Add project folders to ensure dependency analysis detects project files. On the **Project** tab, in the **Environment** section, click **Project Path**:

- To add a folder (without subfolders) to the project path, click **Add Folder**. If you want to add a folder and its subfolders, click **Add with Subfolders** instead. Then use the Open dialog box to add the new folder.
- To remove a folder from the project path, from the display list, select the folder. Then click **Remove**.

In the **Project > Files** view, you can use the context menu to add or remove folders from the project path. Right-click a folder and select **Project Path > Add to Project Path**, or **Add to the Project Path (Including Subfolders)**, or one of the options to remove the folder from the path.

Folders on the project path display the project path icon in the **Status** column.



See Also

Related Examples

- “Specify Project Details, Startup Folder, and Derived Files Folders” on page 16-23

More About

- “What Is the MATLAB Search Path?”
- “What Can You Do With Project Shortcuts?” on page 16-25

What Can You Do With Project Shortcuts?

In a project, use shortcuts to make it easy for any project user to find and access important files and operations. You can use shortcuts to make top models or scripts easier to find in a large project. You can group shortcuts to organize them by type and annotate them to use meaningful names instead of cryptic file names.

Using the Project Shortcuts tab in the toolstrip, you can execute, group, or annotate shortcuts. Run shortcuts by clicking them in the Project Shortcuts tab or execute them manually from the context menu.

To automate tasks, use startup and shutdown files instead. You can use startup files to help you set up the environment for your project, and shutdown shortcuts to help you clean up the environment for the current project when you close it.

See Also

Related Examples

- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31
- “Automate Startup Tasks” on page 16-26
- “Automate Shutdown Tasks” on page 16-28
- “Specify Project Path” on page 16-24

Automate Startup Tasks

In a project, startup files help you set up the environment for your project.

Startup files are automatically run (`.m` and `.p` files), loaded (`.mat` files), and opened (Simulink models) when you open the project.

Note Startup scripts can have any name. You do not need to use `startup.m`

You can use a file named `startup.m` on the MATLAB path which runs when you start MATLAB. If your `startup.m` file calls the project with `currentProject`, an error appears because no project is loaded yet. To avoid the error, rename `startup.m` and use it as a project startup file instead.

Configure an existing file to run when you open your project.

- Right-click the file and select **Run at Startup**.
- Alternatively, on the Project tab, click **Startup Shutdown**. In the Manage Project Startup and Shutdown dialog box, you can add and remove startup and shutdown files. If execution order is important, change the order using the arrow buttons.

In the files view, the **Status** column displays an icon and tooltip indicating the file will run at startup.

Note Startup file settings are included when you commit modified files to source control. Any startup tasks you create run for all other project users.

To stop a file running at startup, change back by right-clicking it and selecting **Remove from Startup**.

On the Manage Project Startup and Shutdown dialog box, use the check boxes to specify environment options:

- **Start Simulink before this project starts**–This option starts Simulink when you open the project.
- **Refresh Simulink customizations**– This option runs `sl_customization` files on project startup and shutdown.

When you open the project, the startup files run. Also, the current working folder changes to the project startup folder. If you want to set the startup folder, on the Project tab, click **Details** and edit the **Start Up** folder. See “Specify Project Details, Startup Folder, and Derived Files Folders” on page 16-23.

You can create new startup and shutdown files interactively in the project or at the command line. For details, see `addStartupFile`.

See Also

Related Examples

- “Specify Project Path” on page 16-24

- “Automate Shutdown Tasks” on page 16-28
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Automate Shutdown Tasks

In a project, shutdown files help you clean up the environment for the current project when you close it. Shutdown files should undo the settings applied in startup files.

Configure an existing file to run when you close your project.

1 Right-click the file and select **Run at Shutdown**.

The **Status** column displays an icon and tooltip indicating the file will run at shutdown.

Note Shutdown files are included when you commit modified files to source control. Any shutdown files you set run for all other project users.

To stop a file running at shutdown, change it back by right-clicking it and selecting **Remove from Shutdown**.

See Also

Related Examples

- “Automate Startup Tasks” on page 16-26
- “Specify Project Path” on page 16-24
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Create Shortcuts to Frequent Tasks

In this section...

Create Shortcuts

In a project, create shortcuts for common project tasks and to make it easy to find and access important files and operations. For example, find and open top models, run code (for example, to load data), and simulate models. To run startup or shutdown code, see instead “Automate Startup Tasks” on page 16-26.

To configure an existing project file as a shortcut, use any of the following methods:

- In the **Files** view, right-click the project file and select **Create Shortcut**. In the Create New Shortcut dialog box, you can edit the shortcut name, choose an icon, and select a group if you have created a shortcut group you want to use. You can change shortcut group later. Click **OK**.
- Click **New Shortcut** on the Project Shortcuts tab on the toolstrip and browse to select a file.

The shortcut appears on the Project Shortcuts tab on the toolstrip.

In the files view, the **Status** column displays an icon and tooltip indicating the file is a shortcut.

Note Shortcuts are included when you commit your modified files to source control, so you can share shortcuts with other project users.

Group Shortcuts

You can group shortcuts to organize them by type. For example, you can group shortcuts for loading data, opening models, generating code, and running tests.

You can select a shortcut group when creating shortcuts, or manage groups later on the Project Shortcuts toolstrip tab.

Create new shortcut groups to organize your shortcuts:

- On the Project Shortcuts tab, click **Organize Groups**.
- Click **Create**, enter a name for the group and click **OK**.

  The new shortcut group appears on the Project Shortcuts tab.

To organize your shortcuts by group, either:

- Select a group when creating a shortcut.
- In the **Project** files view, right-click a file and select **Edit Shortcut**.

- On the Project Shortcuts tab, right-click a file and select **Edit Shortcut**.

The shortcuts are organized by group in the Project Shortcuts toolstrip tab.

Annotate Shortcuts to Use Meaningful Names

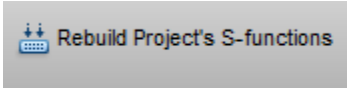
Annotating shortcuts makes their purpose visible, without changing the file name or location of the script or model the shortcut points to. For example, you can change a cryptic file name to a descriptive name for the shortcut. To put shortcuts in a workflow order on the toolstrip, prefix the shortcut names with numbers.

When creating a shortcut, edit the **Name**. The shortcut name does not affect the file name or location.

Your specified shortcut name appears on the Project Shortcuts tab, to make it easier to find your shortcuts.

Customize Shortcut Icons

You can specify an icon to use for your shortcut buttons on the Project Shortcuts tab. Icons such as "build" can aid other project users to recognize frequent tasks.



1. When creating a shortcut, choose an icon.
2. Select an image file. Images must be exactly 16 pixels square, and a `png` or a `gif` file.

See Also

Related Examples

- “Use Shortcuts to Find and Run Frequent Tasks” on page 16-31
- “Automate Startup Tasks” on page 16-26

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Use Shortcuts to Find and Run Frequent Tasks

In a project, use shortcuts to make it easy for any project user to find and access important files and operations. You can use shortcuts to make top models or scripts easier to find in a large project. Shortcuts are available from any file view via the toolstrip.

If your project does not yet contain any shortcuts, see “Create Shortcuts to Frequent Tasks” on page 16-29.

To use shortcuts:

- In the Project Shortcuts toolstrip tab, click the shortcut. Clicking a shortcut in the toolstrip performs the default action for the file type, for example, run `.m` files, load `.mat` files, and open models. Hover over a shortcut to view the full path.

Choose which behavior you want when running shortcuts:

- If the script is not on the path, and you want to switch to the parent folder and run the script without being prompted, then click the shortcut in the Project Shortcuts toolstrip tab. If you use this option, the result of `pwd` in the script is the parent folder of the script.
- If you select **Run** in the **Files** view context menu, and the script is not on the path, then MATLAB asks if you want to change folder or add the folder to the path. This is the same behavior as running from the Current Folder browser. If you use this option, the result of `pwd` in the script is the current folder when you run the script.

See Also

Related Examples

- “Create Shortcuts” on page 16-29
- “Group Shortcuts” on page 16-29
- “Annotate Shortcuts to Use Meaningful Names” on page 16-30
- “Automate Startup Tasks” on page 16-26

More About

- “What Can You Do With Project Shortcuts?” on page 16-25

Create Templates for Standard Project Settings

In this section...

Using Templates to Create Standard Project Settings

In a project, use templates to create and reuse a standard project structure. Templates help you make consistent projects across teams. You can use templates to create new projects that:

- Use a standard folder structure.
- Set up a company standard environment, for example, with company libraries on the path.
- Have access to tools such as company Model Advisor checks.
- Use company standard startup and shutdown scripts.
- Share labels and categories.

You can use templates to share information and best practices. You or your colleagues can create templates.

Create a template from a project when it is useful to reuse or share with others. You can use the template when creating new projects.

Create a Template from the Current Project

In a project, when you create a template, it contains the structure and all the contents of the current project, enabling you to reuse scripts and other files for your standard project setup. You can choose whether to include the contents of referenced projects in the template.

1. Before creating the template, create a copy of the project, and edit the copy to contain only the files you want to reuse. Use the copy as the basis for the template.

   Note If the project is under version control, see instead “Create a Template from a Project Under Version Control” on page 16-33.

2. On the **Project** tab, in the **File** section, select **Share** > **Simulink Template**.
3. On the Create Project Template dialog box, edit the name and author, select or create a group, and add a description to help template users.
4. If you have referenced projects and want to export the referenced project files, then select the **Include referenced projects** check box.
5. Click **Save As**. Choose a file location and click **Save**

Create a Template from a Project Under Version Control

1. Get a new working copy of the project. See “Clone Git Repository” on page 19-25 or “Check Out SVN Repository” on page 19-27.
2. To avoid accidentally committing changes to your project meant only for the template, stop using source control with this sandbox as you work on the template. In the Source Control view, under **Available Source Control Integrations**, select `No Source Control Integration` and click **Reload**.
3. Remove the files that you do not want in the template. For example, you might want to reuse only the utility functions, startup and shutdown scripts, and labels. In the **Files** view, right-click unwanted files and select **Remove from Project**.
4. On the **Project** tab, in the **File** section, select **Share** > **Simulink Template** and use the dialog box to name and save the file.

To verify that your template behaves as you expect, create a new project that uses your new template. See “Create a New Project Using Templates” on page 16-21.

Edit a Template

1. Create a new project that uses the template you want to modify. See “Create a New Project Using Templates” on page 16-21.
2. Make the changes.
3. On the **Project** tab, in the **File** section, select **Share** > **Simulink Template**.

   Use the dialog box to create a new template or overwrite the existing one.

Remove a Template

To remove a template from the Simulink start page:

1. Browse to the template location specified in the template details.
2. Delete the template or move it to another location.

Explore the Example Templates

You can use the example templates as example structures for a new project.

You can explore the templates using the Simulink start page. To search for templates, use the start page search box. See “Create a New Project Using Templates” on page 16-21 and “Create a New Project From a Folder” on page 16-14.

See Also

Related Examples

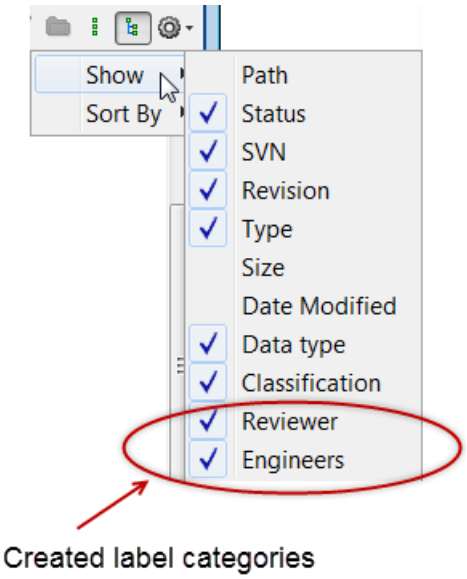
- “Create a New Project Using Templates” on page 16-21
- “Compare Project or Model Templates” on page 21-26
- “Clone Git Repository” on page 19-25

Project File Management

Group and Sort File Views

In a project, to group and sort the views in the Files view:

- Use the `List` or `Tree` options under **Layout** to switch between a flat list of files and a hierarchical tree of files.
- Click the Actions button to specify display columns and sort order. For example, you can display columns for label categories that you created and sort files by label category.



See Also

Related Examples

- “Search Inside Project Files and Filter File Views” on page 17-3

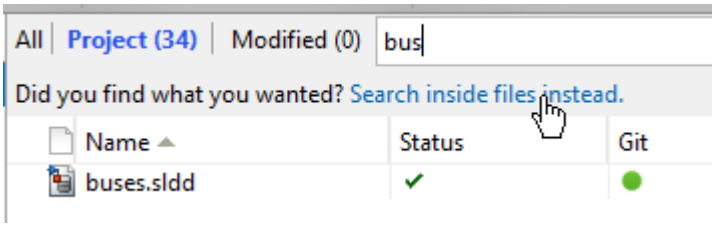
Search Inside Project Files and Filter File Views

In this section...

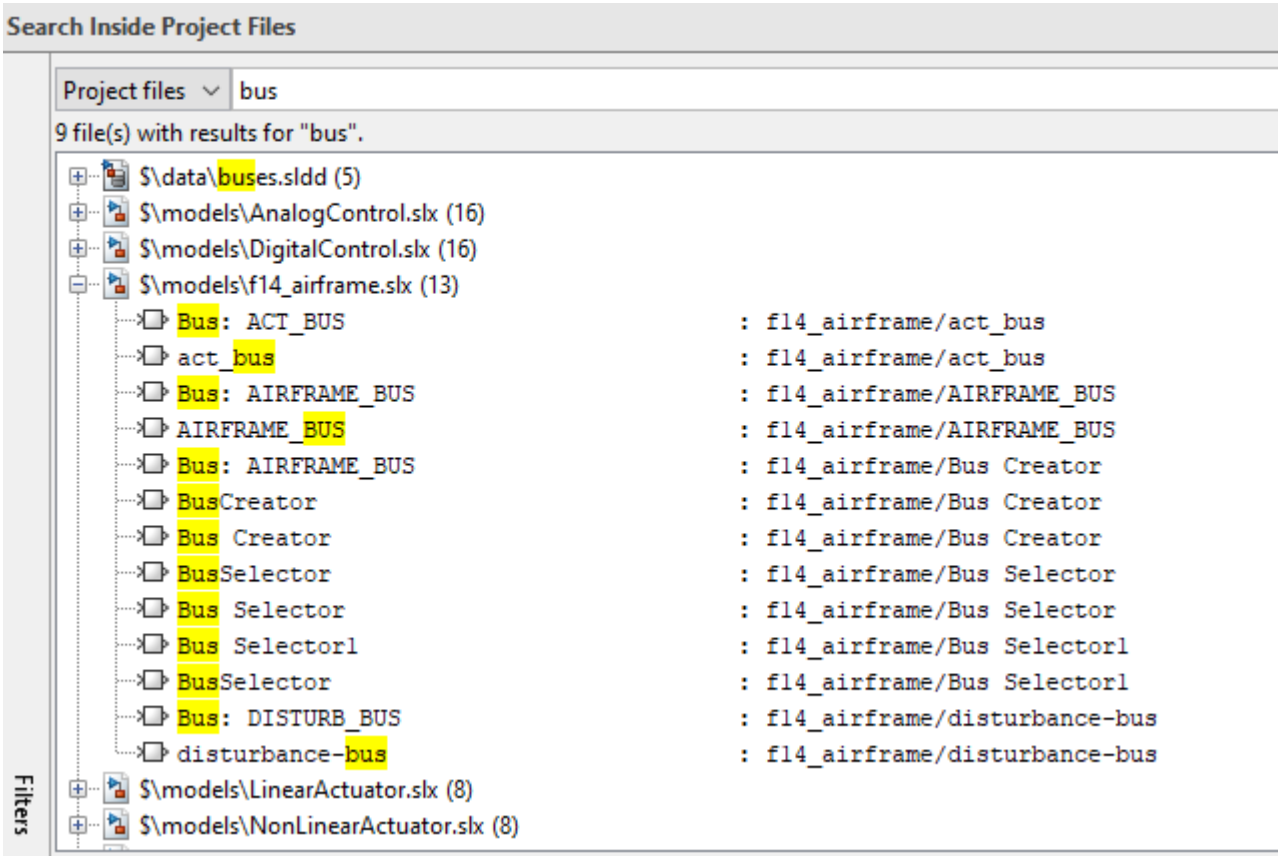
Project-Wide Search

In a project, you can search inside all your models and supporting files. You can find matches inside model files, MATLAB files, data dictionaries, and other project files such as PDF and Microsoft Word files. You search only the current project. If you want to search referenced project files, open the referenced project.

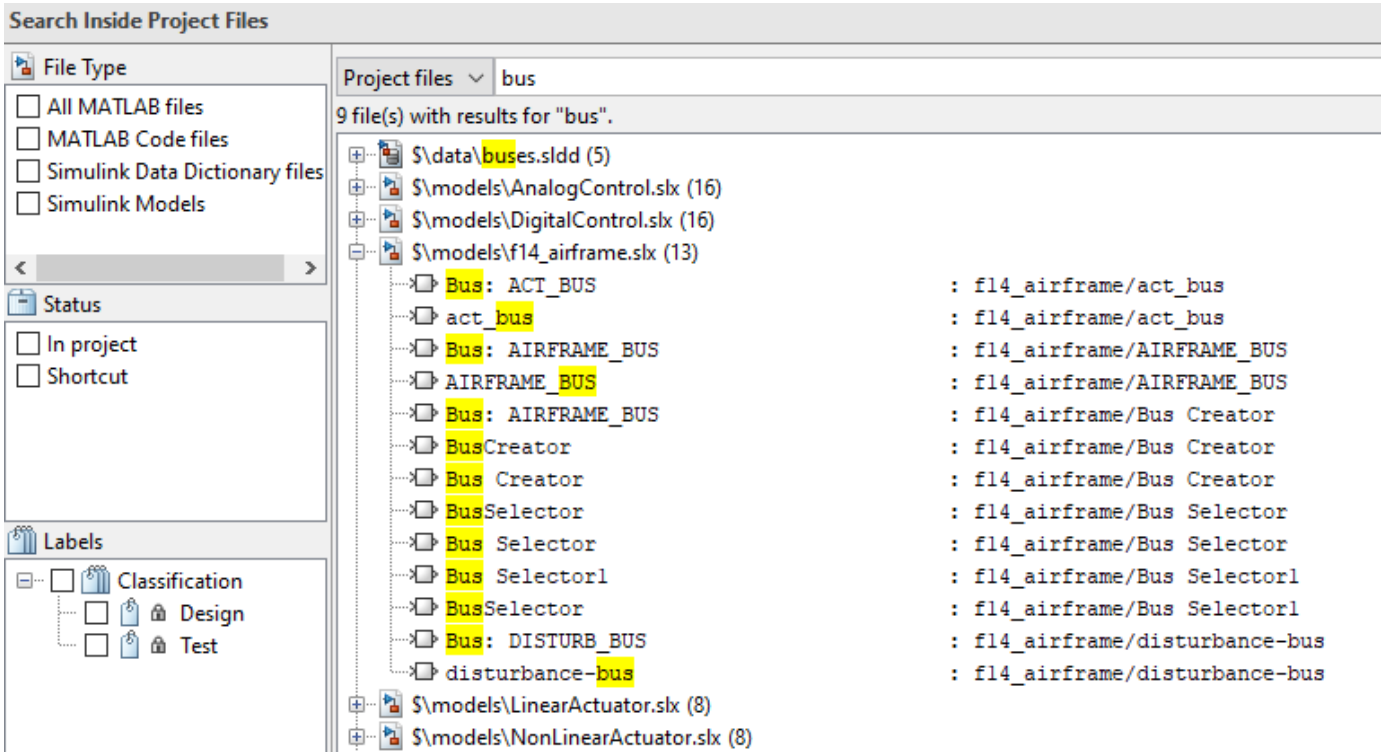
1. On the **Project** tab, click **Search**. Alternatively, type in the file filter box, and the project provides a link to try searching inside files instead.



2. In the Search Inside Project Files dialog box, enter some characters to search for. Do not use quotes around phrases, or hyphens.
3. Expand files in the list to see results inside the file. Double-click results to locate specific items, e.g., highlight blocks in Simulink models, or highlight specific lines in MATLAB files.



4 Click **Filters** to refine results by file type, status, or label.



Filter Project File Views

In a project, in the Files view and in the Custom Task dialog box, you can use the search box and filtering tools to specify file display.

- To view files, select the **Files** node. When the **Project (*number of files*)** view is selected, only the files in your project are shown. To see all the files in your sandbox, click **All**. This view shows all the files that are under the project root, not just the files that are in the project.
- To search, type a search term in the search box, for example, part of a file name or a file extension. You can use wildcards, for example, `*.m`, or `*.m*`.

  Click **X** to clear the search.

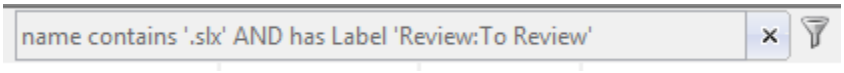
- To build a filter for the current view, click the filter button  .

  In the Filter Builder dialog box you can select multiple filter criteria to apply using names, file types, project status, and labels.

  The dialog box reports the resulting filter at the bottom, for example:

  ```
  Filter = file type is 'Model files (*.slx, *.mdl)' AND project status
   is 'In project' AND has label 'Engine Type:Diesel'
  ```

  When you click **Apply**, the search box shows the filter that you are applying.



More Ways to Search

You can also search:

- Model contents without loading the models into memory. On the MATLAB **Home** tab, in the **File** section, click **Find Files**. You can search a folder or the entire path. However, you cannot highlight results in models from the results in the Find Files dialog box the same way you do with project search. See “Advanced Search for Files”.
- A model hierarchy. In the Simulink Editor, on the **Modeling** tab, click **Find**. Select options to look inside masks, links, and references. This search loads the models into memory. See Model Explorer
- For variables, block parameter values, or search a model hierarchy and contents using more options, using the Model Explorer. This search loads the models into memory. Use the Model Explorer to search for variables in workspaces and data dictionaries, and variable usage in a model. See “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110

See Also

Model Explorer

Related Examples

- “Group and Sort File Views” on page 17-2
- “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110

Work with Project Files

In a project, in the **Files** view, use the context menus to perform actions on the files that you are viewing. Right-click a file (or selected multiple files) to perform project options such as:

- Open files.
- Add and remove files from the project.
- Add, change, and remove labels. See “Add Labels to Files” on page 17-16.
- Create entry point shortcuts (for example, code to run at startup or shutdown, open models, simulate, or generate code). See “Create Shortcuts to Frequent Tasks” on page 16-29.
- If a source control interface is enabled, you can also:
  - Refresh source control status.
  - Update from source control.
  - Check for modifications.
  - Revert.
  - Compare against revision (select a version to compare).

  See “About Source Control with Projects” on page 19-2.

In the **Files** view, if you select, for example a model file, the bottom right-hand pane displays file information, a model preview, and file labels.

See Also

Related Examples

- “Open Recent Projects” on page 16-22
- “Add Files to the Project” on page 16-18
- “Move, Rename, Copy, or Delete Project Files” on page 17-10
- “Back Out Changes” on page 17-14
- “Group and Sort File Views” on page 17-2
- “Search Inside Project Files and Filter File Views” on page 17-3
- “Create a Custom Task Function” on page 17-27

More About

- “What Can You Do With Project Shortcuts?” on page 16-25
- “About Source Control with Projects” on page 19-2

Manage Shadowed and Dirty Models and Other Project Files

In this section...

Identify Shadowed Project Files When Opening a Project

If there are two model files with the same name on the MATLAB path, then the one higher on the path is loaded, and the one lower on the path is shadowed. This shadowing applies to all models and libraries (SLX and MDL files).

A loaded model always takes precedence over unloaded ones, regardless of its position on the MATLAB path. Loaded models can interfere when you try to use other files of the same name, especially when models are loaded but not visible. Simulink warns when you try to load a shadowed model, because the other model is already loaded and can cause conflicts. The project checks for shadowed files when you open a project.

1. When you open a project, it warns you if any models of the same names as your project models are already loaded. This check enables you to find and avoid shadowed files before opening any project models.

   The Configuring Project Environment dialog box reports the **Identify shadowed project files** check fails. Click **Details**.
2. In the dialog box, you can choose to show or close individual files, or close all potentially shadowing files, by clicking the hyperlinks. To avoid working on the wrong files, close the loaded models.
3. After deciding whether to show or close the loaded models, click **OK** to return to the Configuring Project Environment dialog box.
4. Inspect the other project loading tasks, then click **Continue** to view the project.

Tip To help avoid problems with shadowed files, turn on the Simulink preference **Do not load models that are shadowed on the MATLAB path**. See “Do not load models that are shadowed on the MATLAB path”.

When you open a project with many referenced projects, identifying shadowed files can be time-consuming. You can turn off this check using the MATLAB project preference **Detect project files shadowed by open models**.

To learn more about shadowed files, see “Shadowed Files” on page 15-3.

Find Models and Other Project Files With Unsaved Changes

You can check your project for models, data dictionaries and MATLAB files with unsaved changes. On the **Project** tab, in the **File** section, click **Unsaved Changes**.

In the Unsaved Changes dialog box, you can see all dirty project models, data dictionaries, and MATLAB files. Project only detects unsaved changes edited in the MATLAB and Simulink editors.

Manually examine changes edited in other tools. If you have referenced projects, files are grouped by project. You can save or discard all detected changes.

Manage Open Models and Data Dictionaries When Closing a Project

When you close a project, it closes any project models or data dictionaries, unless they are dirty.

When you close a project, if there are model files or data dictionaries with unsaved changes, a message prompts you to save or discard changes. You can see all dirty files, grouped by project if you have referenced projects. To avoid losing work, you can save or discard changes by file, by project, or globally.

Control this behavior using the project shutdown preferences.

See Also

Related Examples

- “Do not load models that are shadowed on the MATLAB path”
- “Shadowed Files” on page 15-3

Move, Rename, Copy, or Delete Project Files

In this section...

Move or Add Files

To move or add project files, you can drag them to the project, or use clipboard operations.

- To add files to your project, you can paste files or drag them from your operating system file browser or the MATLAB Current Folder browser onto the **Project** files view in the project. When you drag a file to the **Project** files view, you add the file to the project. For projects under source control, you also add the file to source control.
- To move files within your project, cut and paste or drag files in the project.

See also “Add Files to the Project” on page 16-18.

Automatic Updates When Renaming, Deleting, or Removing Files

When you rename, delete, or remove files or folders in a project, the project checks for impact in other project files. You can find and fix impacts such as changed library links, model references, and model callbacks. You can avoid refactoring pain tracking down other affected files. Automatic renaming helps to prevent errors that result from changing names or paths manually and overlooking or mistyping one or more instances of the name.

For example:

- When renaming a library, the project offers to automatically update all library links to the renamed library.
- When renaming a class, the project offers to automatically update all classes that inherit from it. If you rename a `.m` or `.mlx` file, the project offers to automatically update any files and callbacks that call it.
- When deleting files or removing them from the project, the project prompts you if other files refer to them. You must decide how to fix the affected files manually.
- When renaming a C file, the project prompts you to update the S-function that uses it.
- When renaming buses or bus elements using the Simulink Bus Editor, the project prompts you to update all usages in the project.

To use automatic updates:

1. Rename a model, library, or MATLAB file in a project.

   The project runs a dependency analysis to look for impacts in other files.
2. In the Rename dialog box, you can examine impacted files, choose to rename and update, just rename, or cancel renaming the file.
3. If you choose automatic updates, you can examine the results in updated files.

Automatic Renaming Using the Power Window Project

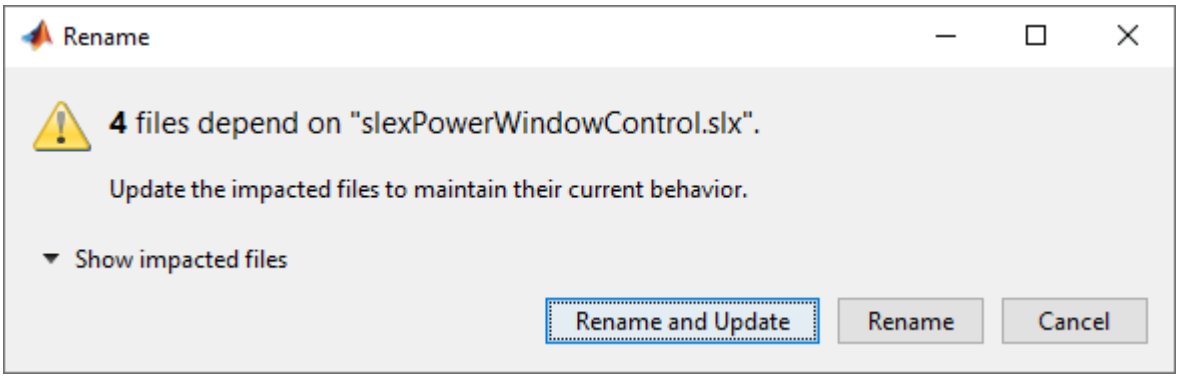
1. Open the power window example project by entering in MATLAB:

   ```
   slexPowerWindowStart
   ```

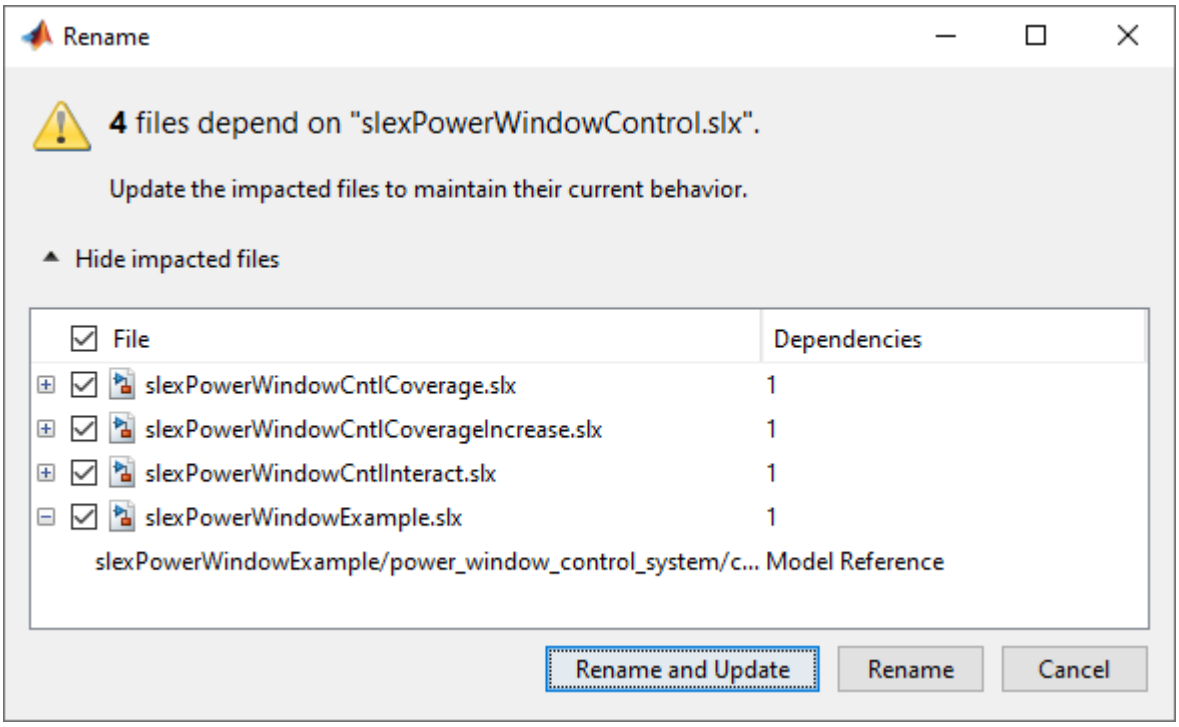
   The project opens the top model, some scopes, and an animation window.

2. In a project, expand the `model` folder, and rename the `slexPowerWindowControl.slx` model to `slexPowerWindowControlSystem.slx`.

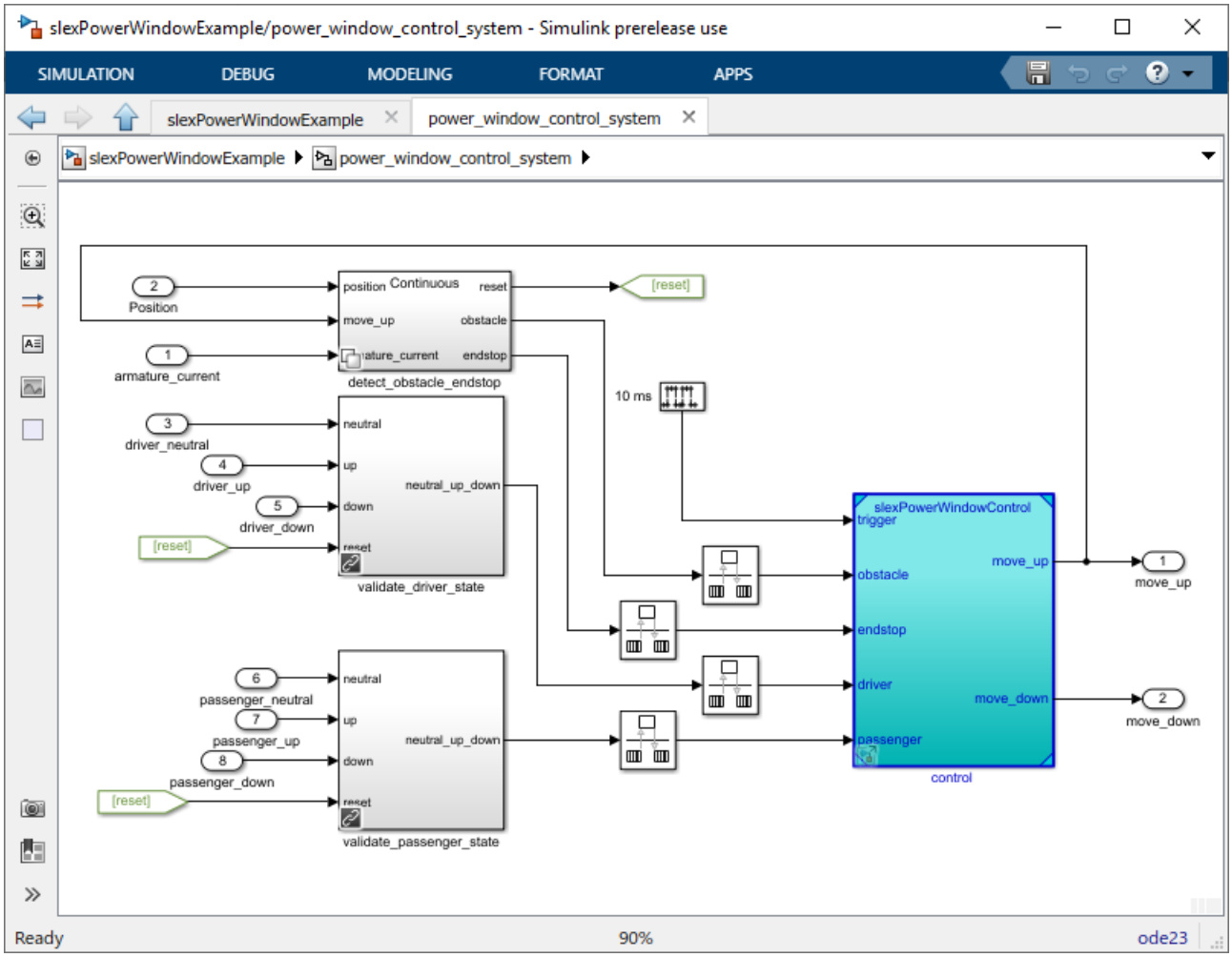
   The project runs a dependency analysis to look for impacts in other files, and then the Rename dialog box reports impacted files.



3. In the Rename dialog box, click **Show impacted files**. Expand the last impacted file to view the dependency, which is a model reference.

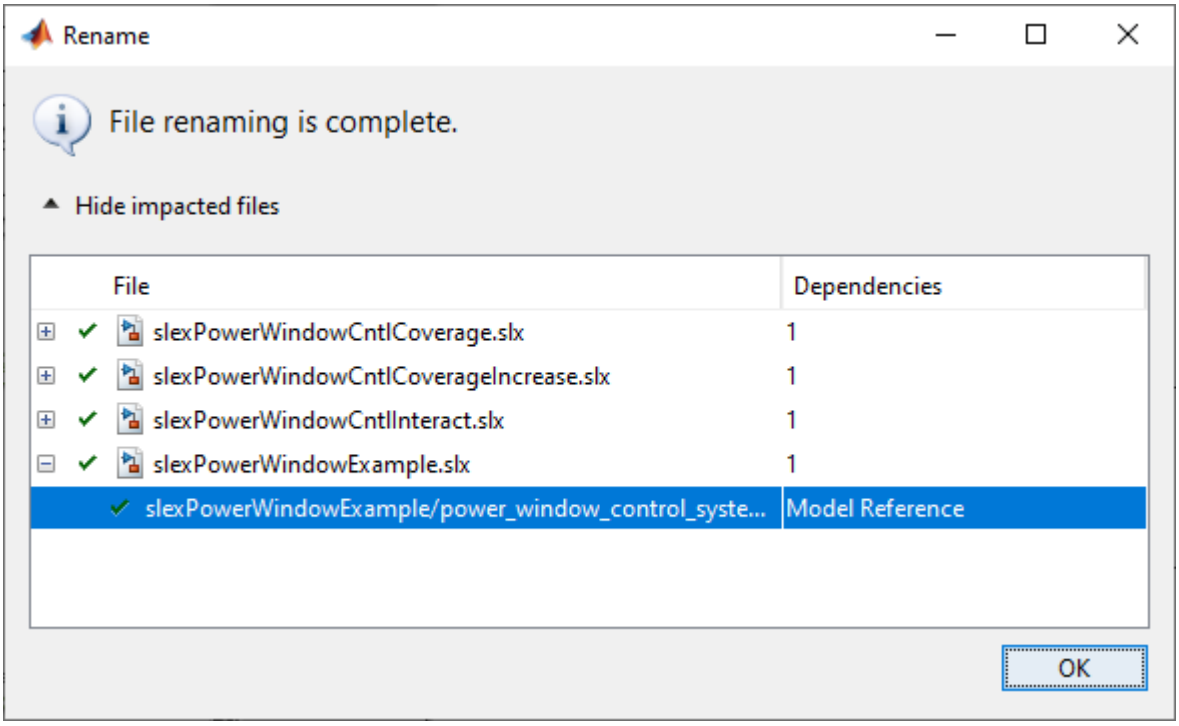


4. To view the dependency highlighted in the model, double-click the last **Model Reference** line in the Rename dialog box. Observe the model name on the highlighted control block, `slexPowerWindowControl`.



5 In the Rename dialog box, click **Rename and Update**.

The project updates the impact files to use the new model name in model references. When the project can automatically rename items, it reports success with a check mark. With some impacts, you must decide how to fix the affected files manually.



6 Examine the results by double-clicking items in the Rename dialog box. Double-click the last **Model Reference** line. Check if the model name on the highlighted control block is updated to `slexPowerWindowControlSystem`.

See Also

Related Examples

- “Work with Project Files” on page 17-7

Back Out Changes

Similar to many applications, the project enables you to Undo and Redo, to back out recent changes.

1. Click the arrow next to the Undo or Redo button.
2. Select the actions you want to undo or redo. You can select multiple actions. Hover over each action to view details in a tooltip.

If you are using source control, you can revert to particular versions of files or projects. See “Revert Changes” on page 19-44.

See Also

Related Examples

- “Revert Changes” on page 19-44

Create Labels

In a project, use labels to organize files and communicate information to project users. You can create these types of label categories:

- Single-valued — You can attach only one label from the category to a file.
- Multi-valued — You can attach multiple labels from the category to a file.

The **Labels** tree has built-in labels in the single-valued **Classification** category:

- You cannot rename or delete **Artifact**, **Convenience**, **Derived**, **Design**, **None**, **Test**, and **Other**.
- You cannot annotate built-in labels.

To create a label category:

1. In a project, right-click the **Labels** pane. Then select **Create New Category**.
2. In the Create Category dialog box, enter a name for the new category.
3. If you require a single-valued label category, select the **Single Valued** check box. The default is multi-valued.
4. If you want to specify a data type for the label other than `String`, from the **Type** list, select `Double`, `Integer`, `Logical`, or `None`.
5. Click **Create**.

To create a label in a category:

1. In the **Labels** pane, right-click the label category and select **Create New Label**.
2. In the Create Label dialog box, enter a name for the new label and click **OK**.

To rename or delete a category or label, right-click it and select **Rename** or **Remove**.

To create new labels at the command line, see “Automate Project Tasks Using Scripts” on page 17-18.

See Also

Related Examples

- “Add Labels to Files” on page 17-16
- “View and Edit Label Data” on page 17-17

Add Labels to Files

In a project, use labels to organize files and communicate information to project users.

To add a label to a file, use one of these methods:

- Drag the label from the **Labels** pane onto the file.
- In the Files view, select the file. Then, drag the label from the **Labels** pane into the label editor.

To add a label to multiple files, in the **Files** view or Dependency Analyzer graph, select the files that require the label. Then right-click and select **Add Label**. Use the dialog box to specify the label.

To add labels programmatically, for example, in custom task functions, see “Automate Project Tasks Using Scripts” on page 17-18.

Note After you add a label to a file, the label persists across file revisions.

After you add labels, you can organize files by label in the Files view and dependency graph. See “Group and Sort File Views” on page 17-2 and “Perform an Impact Analysis” on page 18-17.

If the project is under SVN source control, adding tags to all your project files enables you to mark versions. See “Tag and Retrieve Versions of Project Files” on page 19-29.

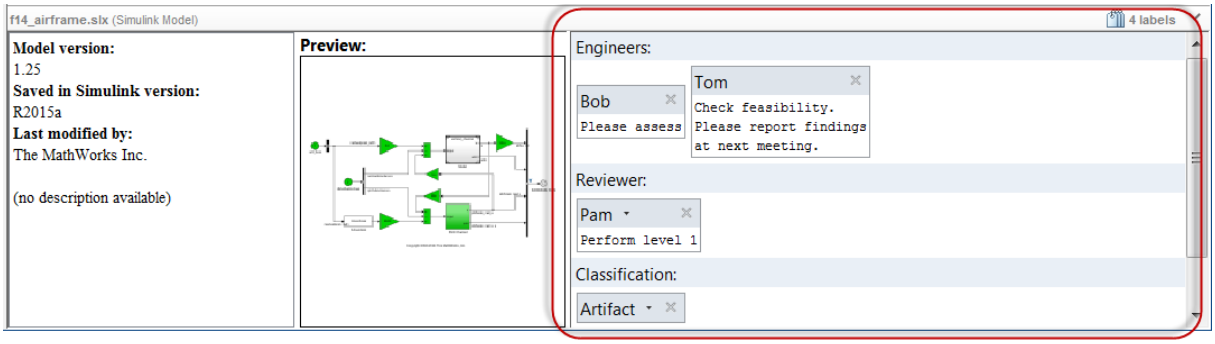
See Also

Related Examples

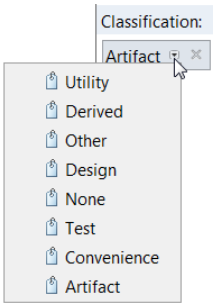
- “Create Labels” on page 17-15
- “View and Edit Label Data” on page 17-17

View and Edit Label Data

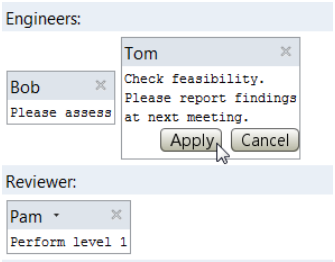
When you select a file in the project Files view, the file labels appear in the label editor view.



To change a label that belongs to a single-valued category, select the new value from the label list.



You can annotate labels from categories that you create. In the label, insert or modify text. Then, click **Apply**.



See Also

Related Examples

- “Create Labels” on page 17-15
- “Add Labels to Files” on page 17-16

Automate Project Tasks Using Scripts

This example shows how to use the project API to automate project tasks manipulating files, including working with modified files, dependencies, shortcuts, and labels.

Get Project at the Command Line

Open the Airframe project and use currentProject to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = currentProject
```

```
Building with 'MinGW64 Compiler (C)'.
MEX completed successfully.

proj =

  Project with properties:

                       Name: "Airframe Example"
    SourceControlIntegration: "Git"
          RepositoryLocation: "C:\workSpace\examples\repositories\airframe1"
       SourceControlMessages: [1×3 string]
                    ReadOnly: 0
                    TopLevel: 1
                Dependencies: [1×1 digraph]
                  Categories: [1×1 matlab.project.Category]
                       Files: [1×31 matlab.project.ProjectFile]
                   Shortcuts: [1×7 matlab.project.Shortcut]
                 ProjectPath: [1×7 matlab.project.PathFolder]
           ProjectReferences: [1×0 matlab.project.ProjectReference]
                StartupFiles: [1×0 string]
               ShutdownFiles: [1×0 string]
                 Description: "This is an example project.↵↵Use the "Project Shortcuts" toolstri
                  RootFolder: "C:\workSpace\examples\airframe1"
       SimulinkCodeGenFolder: "C:\workSpace\examples\airframe1\work\codegen"
        ProjectStartupFolder: "C:\workSpace\examples\airframe1"
         SimulinkCacheFolder: "C:\workSpace\examples\airframe1\work\cache"
```

Find Project Commands

Find out what you can do with your project.

```
methods(proj)
```

```
Methods for class matlab.project.Project:

addFile                       listModifiedFiles
addFolderIncludingChildFiles  listRequiredFiles
addPath                       refreshSourceControl
addReference                  reload
addShortcut                   removeCategory
addShutdownFile               removeFile
addStartupFile                removePath
addprop                       removeReference
```

```
close                          removeShortcut
createCategory                 removeShutdownFile
export                         removeStartupFile
findCategory                   runChecks
findFile                       updateDependencies
isLoaded

Call "methods('handle')" for methods of matlab.project.Project inherited from handle.
```

Examine Project Files

After you get a project object, you can examine project properties such as files.

```
files = proj.Files
```

```
files =

  1×31 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Use indexing to access files in this list. The following command gets file number 10. Each file has properties describing its path and attached labels.

```
proj.Files(10)
```

```
ans =

  ProjectFile with properties:

                  Path: "C:\workSpace\examples\airframe1\data\system_model.sldd"
                Labels: [1×1 matlab.project.Label]
              Revision: "3e2de2b2a43515259c0843aacf330775d8fda493"
    SourceControlStatus: Unmodified
```

Examine the labels of the 10th file.

```
proj.Files(10).Labels
```

```
ans =

  Label with properties:

            File: "C:\workSpace\examples\airframe1\data\system_model.sldd"
        DataType: 'none'
            Data: []
            Name: "Design"
    CategoryName: "Classification"
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.slx')
```

```
myfile =

  ProjectFile with properties:

                 Path: "C:\workSpace\examples\airframe1\models\AnalogControl.slx"
               Labels: [1×1 matlab.project.Label]
             Revision: "3e2de2b2a43515259c0843aacf330775d8fda493"
    SourceControlStatus: Unmodified
```

Find out what you can do with the file.

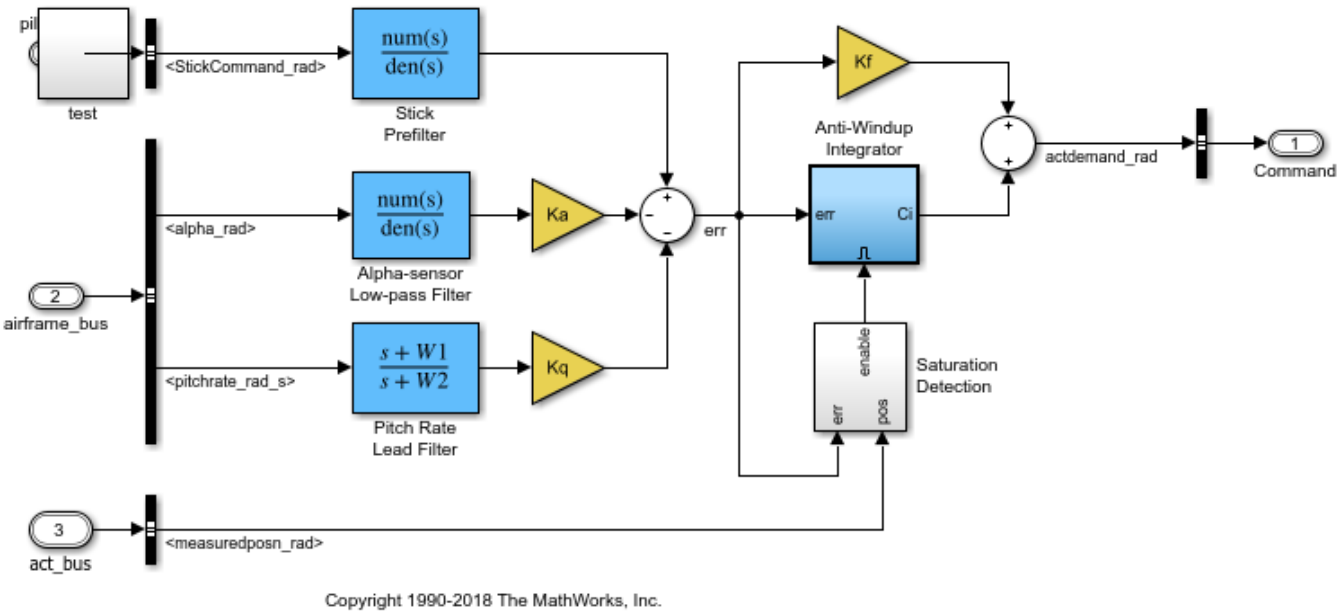
```
methods(myfile)
```

```
Methods for class matlab.project.ProjectFile:

addLabel     findLabel    removeLabel
```

Get Modified Files

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. Observe two modified files. Compare with the Modified Files view in Project, where you can see a modified model file, and the corresponding project definition file.

```
modifiedfiles = listModifiedFiles(proj)
```

```
modifiedfiles =

  1×2 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Get the second modified file. 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 Project.

```
modifiedfiles(2)
```

```
ans =

  ProjectFile with properties:

                Path: "C:\workSpace\examples\airframe1\resources\project\uuid-0988ac9d-5add-4
            Revision: ""
    SourceControlStatus: Added
```

Refresh source control status before querying individual files. You do not need to do before using `listModifiedFiles` .

```
refreshSourceControl(proj)
```

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.Unmodified))
```

```
ans =

  1×22 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Get File Dependencies

Update the file dependencies. The project runs a dependency analysis to update the known dependencies between project files.

```
updateDependencies(proj)
```

Get the list of dependencies in the Airframe project. The Dependencies property contains the graph of dependencies between project files, stored as a MATLAB digraph object.

```
g = proj.Dependencies
```

```
g =

  digraph with properties:

    Edges: [21×1 table]
    Nodes: [21×1 table]
```

Get the files required by a model.

```
requiredFiles = bfsearch(g, which('AnalogControl'))
```

```
requiredFiles =

  3×1 cell array

    {'C:\workSpace\examples\airframe1\models\AnalogControl.slx'}
    {'C:\workSpace\examples\airframe1\data\controller.sldd'    }
    {'C:\workSpace\examples\airframe1\data\buses.sldd'         }
```

Get the top-level files of all types in the graph.

```
g.Nodes.Name(indegree(g)==0);
```

Get the top-level files that have dependencies.

```
g.Nodes.Name(indegree(g)==0 & outdegree(g)>0)
```

```
ans =

  4×1 cell array

    {'C:\workSpace\examples\airframe1\models\DigitalControl.slx'}
    {'C:\workSpace\examples\airframe1\models\LinearActuator.slx'}
    {'C:\workSpace\examples\airframe1\models\slproject_f14.slx' }
    {'C:\workSpace\examples\airframe1\tests\f14_airframe_test.m'}
```

Find impacted (or "upstream") files by creating a transposed graph.

```
transposed   = flipedge(g)
impacted = bfsearch(transposed, which('vertical_channel'))
```

```
transposed =

  digraph with properties:

    Edges: [21×1 table]
    Nodes: [21×1 table]


impacted =
```

```
  4×1 cell array

    {'C:\workSpace\examples\airframe1\models\vertical_channel.slx'}
    {'C:\workSpace\examples\airframe1\models\f14_airframe.slx'    }
    {'C:\workSpace\examples\airframe1\models\slproject_f14.slx'   }
    {'C:\workSpace\examples\airframe1\tests\f14_airframe_test.m'  }
```

Find files impacted by a data dictionary.

```
impacted2 = bfsearch(transposed, which('buses.sldd'))
```

```
impacted2 =

  11×1 cell array

    {'C:\workSpace\examples\airframe1\data\buses.sldd'             }
    {'C:\workSpace\examples\airframe1\data\controller.sldd'        }
    {'C:\workSpace\examples\airframe1\data\system_model.sldd'      }
    {'C:\workSpace\examples\airframe1\tests\f14_airframe_test.m'   }
    {'C:\workSpace\examples\airframe1\models\AnalogControl.slx'    }
    {'C:\workSpace\examples\airframe1\models\DigitalControl.slx'   }
    {'C:\workSpace\examples\airframe1\models\f14_airframe.slx'     }
    {'C:\workSpace\examples\airframe1\models\LinearActuator.slx'   }
    {'C:\workSpace\examples\airframe1\models\NonLinearActuator.slx'}
    {'C:\workSpace\examples\airframe1\models\slproject_f14.slx'    }
    {'C:\workSpace\examples\airframe1\models\vertical_channel.slx' }
```

Get information on your files, such as the number of dependencies and orphans.

```
averageNumDependencies = mean(outdegree(g));
numberOfOrphans = sum(indegree(g)+outdegree(g)==0);
```

If you want to make changes to a model hierarchy, starting from the bottom up, find the topological order.

```
ordered = g.Nodes.Name(flip(toposort(g)));
```

Query Shortcuts

Examine the project's Shortcuts property.

```
shortcuts = proj.Shortcuts
```

```
shortcuts =

  1×7 Shortcut array with properties:

    Name
    Group
    File
```

Examine a shortcut in the array.

```
shortcuts(7)
```

```
ans =

  Shortcut with properties:

     Name: "Rebuild Project's S-functions"
    Group: "Utility"
     File: "C:\workSpace\examples\airframe1\utilities\rebuild_s_functions.m"
```

Get the file path of a shortcut.

```
shortcuts(7).File
```

```
ans =

    "C:\workSpace\examples\airframe1\utilities\rebuild_s_functions.m"
```

Examine all the files in the shortcuts cell array.

```
{shortcuts.File}'
```

```
ans =

  7×1 cell array

    {["C:\workSpace\examples\airframe1\custom_tasks\analyzeModelFiles.m"]}
    {["C:\workSpace\examples\airframe1\custom_tasks\billOfMaterials.m"  ]}
    {["C:\workSpace\examples\airframe1\custom_tasks\checkCodeProblems.m"]}
    {["C:\workSpace\examples\airframe1\custom_tasks\runUnitTest.m"      ]}
    {["C:\workSpace\examples\airframe1\models\slproject_f14.slx"        ]}
    {["C:\workSpace\examples\airframe1\reports\slproject_f14.pdf"       ]}
    {["C:\workSpace\examples\airframe1\utilities\rebuild_s_functions.m" ]}
```

Label files

Create a new category of labels, of type char. In the Project, the new Engineers category appears in the Labels pane.

```
createCategory(proj, 'Engineers', 'char')
```

```
ans =

  Category with properties:

                Name: "Engineers"
        SingleValued: 0
            DataType: "char"
    LabelDefinitions: [1×0 matlab.project.LabelDefinition]
```

Find out what you can do with the new category.

```
category = findCategory(proj, 'Engineers');
methods(category)
```

```
Methods for class matlab.project.Category:

createLabel  findLabel    removeLabel
```

Define a new label in the new category.

```
createLabel(category, 'Bob');
```

Get a label definition.

```
ld = findLabel(category, 'Bob')
```

```
ld =

  LabelDefinition with properties:

            Name: "Bob"
    CategoryName: "Engineers"
```

Attach a label to the retrieved file, myfile. If you select the file in Project, you can see this label in the label editor pane.

```
addLabel(myfile, 'Engineers', 'Bob');
```

Get a particular label and attach data to it, for example, some text.

```
label = findLabel(myfile, 'Engineers', 'Bob');
label.Data = 'Please assess'
```

```
label =

  Label with properties:

            File: "C:\workSpace\examples\airframe1\models\AnalogControl.slx"
        DataType: 'char'
            Data: 'Please assess'
            Name: "Bob"
    CategoryName: "Engineers"
```

You can specify a variable for the label data, for example:

```
mydata = label.Data
```

```
mydata =

    'Please assess'
```

Create a new label category with numeric data type.

```
createCategory(proj, 'Assessors', 'double');
category = findCategory(proj, 'Assessors');
createLabel(category, 'Sam');
```

Attach the new label to a specified file and assign data value 2 to the label.

```
myfile = proj.Files(14);
addLabel(myfile, 'Assessors', 'Sam', 2)
```

```
ans =

  Label with properties:

           File: "C:\workSpace\examples\airframe1\models"
       DataType: 'double'
           Data: 2
           Name: "Sam"
   CategoryName: "Assessors"
```

Close Project

Closing the project at the command line is the same as closing the project using the Project tool. For example, the project runs shutdown scripts ad checks for unsaved models.

```
close(proj)
```

More Information

For more details on using the API, enter: `doc currentProject`.

To automate start and shutdown tasks, see “Automate Startup Tasks” on page 16-26.

See Also

`addLabel` | `createLabel` | `currentProject` | `listModifiedFiles` | `refreshSourceControl`

Related Examples

- “Perform an Impact Analysis” on page 18-17
- “Automate Startup Tasks” on page 16-26
- “View Modified Files” on page 19-36
- “Create Labels” on page 17-15
- “Add Labels to Files” on page 17-16
- “View and Edit Label Data” on page 17-17
- Automate Label Management in a Simulink Project

Create a Custom Task Function

In a project, you can create functions and run them on selected project files.

For example custom task functions, see “Running Custom Tasks with a Project”.

To create a custom task function:

1. In a Project, select **Custom Tasks > Manage Custom Tasks**.
2. In the Manage Custom Tasks dialog box, select **Add > Add Using New Script** or **Add Using Existing Script** .

   Either browse to an existing script, or name and save the new file on your MATLAB path. The file is added to the project.
3. The MATLAB Editor opens the new file containing a simple example custom task function for you to edit. Edit the function to perform the desired action on each file. The instructions guide you to create a custom task with the correct function signature. Save the file.
4. To run your task, in the project, click **Custom Tasks**. In the Custom Task dialog box, select project files to include using the check boxes, select your custom task in the list, and click **Run Task**.
5. After your custom task function runs, view the results in the Custom Task Report.
6. To save the results in a file, click **Publish Report**.

See Also

Related Examples

- “Run a Project Custom Task and Publish Report” on page 17-28

Run a Project Custom Task and Publish Report

1. In a project, click **Custom Tasks**, and then select the check boxes of project files you want to include in the custom task.

   Tip If the function can identify the files to operate on, include all files. For example, the custom task function `saveModelFiles` in the `airframe` project checks that the file is a Simulink model and does nothing if it is not.

   To select multiple files, **Shift** or **Ctrl**+click, and then right-click a file and select **Include** or **Exclude**.

2. Specify the custom task function to run in the **Custom task** box. Enter the name, or click **Browse**, or choose from a list of custom tasks.

   If your project does not yet contain any custom task functions, see “Create a Custom Task Function” on page 17-27.

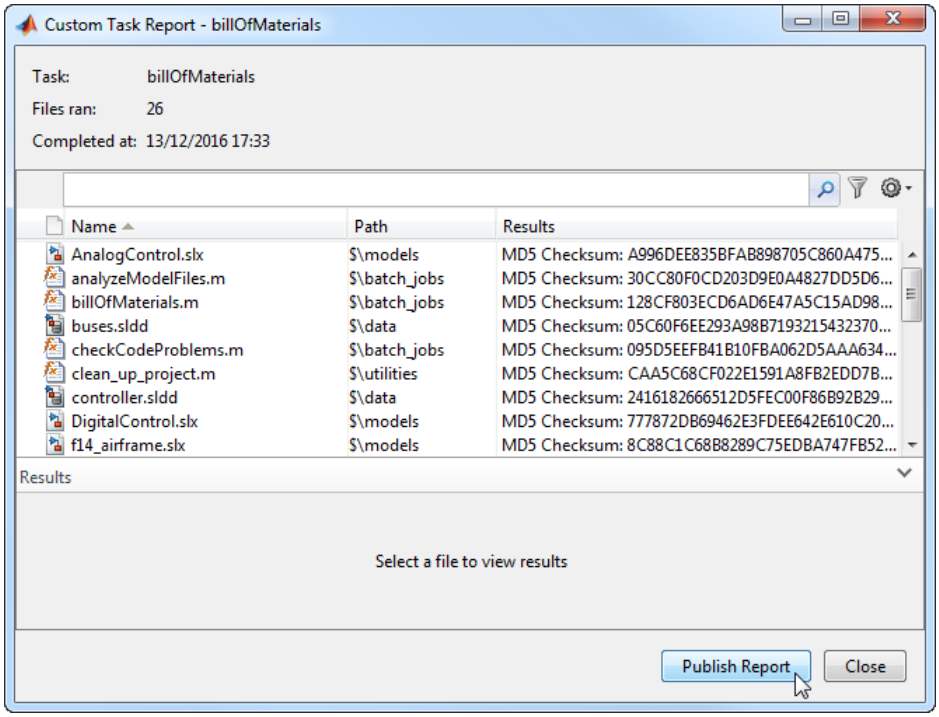
3. Click **Run Task**.

   The project displays the results.

4. To view details of results for the currently selected file, click a file and check the Results pane.

You can publish a report of your custom task results. For example, try this custom task:

1. Open an example project by entering `sldemo_slproject_customtasks`.
2. In a Project, click **Custom Tasks**.
3. In the Custom Task dialog box, click the **Custom task** drop-down arrow to choose from a list of tasks, and select **Generate Bill of Materials Report**.
4. Click **Run Task**. Results appear.



5 Click **Publish Report**.

6 In the file browser, specify a name and location for the report, and choose a file type from HTML or Microsoft Word. If you have MATLAB Report Generator, you can also choose PDF.

7 View the results in the report.

The example custom task function **Generate Bill of Materials Report** creates a list of project files, their source control status, revision numbers, and MD5 checksums. You can view the code for this custom task in the file `billOfMaterials.m`.

8 To see the report file and add it to your project, switch to the **All** files view.

Tip To try example custom tasks in a project, see the example “Running Custom Tasks with a Project”.

See Also

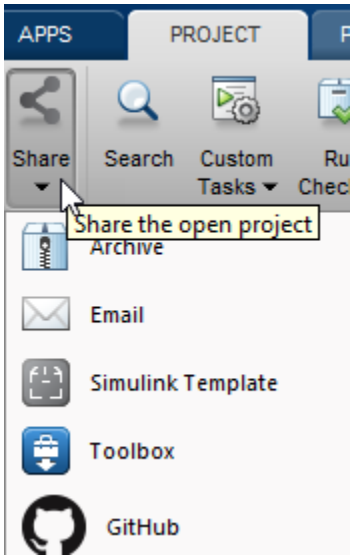
Related Examples

- “Create a Custom Task Function” on page 17-27
- “Running Custom Tasks with a Project”
- “Perform Impact Analysis with a Project”

Sharing Projects

Projects help you collaborate. Use the **Share** menu to share your project in these ways:

- Archive your project in a single file.
- Share your project by email (Windows only).
- Create a template from your project.
- Package your project as a MATLAB toolbox.
- Make your project publicly available on GitHub.



You can also collaborate by using source control within projects.

See Also

Related Examples

- “Archive Projects” on page 17-34
- “Create Templates for Standard Project Settings” on page 16-32
- “Share Project by Email” on page 17-31
- “Share Project as a MATLAB Toolbox” on page 17-32
- “Share Project on GitHub” on page 17-33

More About

- “About Source Control with Projects” on page 19-2

Share Project by Email

To package and share project files on Windows, you can email your project as an archive file attachment. For example, you can share the project with people who do not have access to the connected source control tool.

1. With a project loaded, on the **Project** tab, select **Share** > **Email**.
2. (Optional) To export only the specified files, choose an **Export profile**. To exclude files with particular labels, select **Manage Export Profiles** and create an export profile.
3. If you have referenced projects and want to export the referenced project files, then select the **Include referenced projects** check box.
4. Click **Attach to Email**. The project opens a new email in your default email client with the project attached as an archive `.mlproj` file.
5. Edit and send the email.

See Also

Related Examples

- “Create a New Project from an Archived Project” on page 16-20
- “Archive Projects” on page 17-34

More About

- “Sharing Projects” on page 17-30

Share Project as a MATLAB Toolbox

To package and share project files, you can create a MATLAB toolbox from your project.

1. With a project loaded, on the **Project** tab, select **Share** > **Toolbox**.

   The packager adds all project files to the toolbox and opens the Package a Toolbox dialog box.
2. The **Toolbox Information** fields are populated with the project name, author, and description. Edit the information if needed.
3. If you want to include files not already included in the project files, edit the excluded files and folders.
4. Click **Package**.

See Also

Related Examples

- “Create and Share Toolboxes”

More About

- “Sharing Projects” on page 17-30

Share Project on GitHub

To share your project, you can make your project publicly available on GitHub. First, create a login on GitHub.

You can share any project. Sharing adds Git source control to the open project. If your project is already under source control, sharing replaces the source control configuration with Git, and the project's remote repository is GitHub.

Note If you do not want to change your current source control in the open project, share a copy of the project instead. To create a copy to share, see "Archive Projects" on page 17-34.

1. With a project loaded, on the Project tab, select **Share > Change Share Options**.
2. Add the **GitHub** option to your **Share** menu. In the Manage Sharing dialog box, select **GitHub** and click **Close**.
3. Select **Share > GitHub**.
4. In the Create GitHub Repository dialog box, enter your GitHub user name and personal access token, and edit the name for the new repository. Click **Create**.

   A warning prompts you to confirm that you want to create a public repository and modify the current project's remote repository location. To continue, click **Yes**.
5. The Create GitHub Repository dialog box displays the URL address for your new repository. Click the link to view the new repository on the GitHub website. The repository contains the initial check-in of your project files.
6. The source control in your current project now references the new repository on GitHub as the remote repository. To use the project with the new repository, in the Create GitHub Repository dialog box, click **Reload Project**.

In the project, you can find the URL address for your remote repository on the Project tab, under **Source Control**, using the **Git Details** button.

If you have not already set up Git, you need some additional setup steps before you can merge branches. You can use other Git functionality without any additional installation. See "Set Up Git Source Control" on page 19-16.

See Also

Related Examples

- "Archive Projects" on page 17-34
- "Set Up Git Source Control" on page 19-16

More About

- "Sharing Projects" on page 17-30

Archive Projects

To package and share project files, you can export all project files to an archive file. For example, you can share an archived project with people who do not have access to the connected source control tool.

1. With a project loaded, on the **Project** tab, select **Share** > **Archive**.
2. (Optional) To export only the specified files, choose an **Export profile**.
3. If you have referenced projects and want to export the referenced project files, then select the **Include referenced projects** check box.
4. Click **Save As**.
5. Use the file browser to specify a file path in the **File name** field. By default, the file *myProjectName*`.mlproj` is created in the current working folder. You can choose the file type project archive (`.mlproj`) or zip file.

If you want to exclude files from the archive, create an export profile to exclude files with particular labels.

1. Create labels and add them to the project files you want to exclude. See “Create Labels” on page 17-15.
2. Specify an export profile. On the **Project** tab, select **Share** > **Manage Export Profiles**.
   a. Click **+** and specify a name for the export profile.
   b. In the Files pane, click **+** and select the labels for the files you do not want to export, and click **OK**. You can also choose not to export custom labels.
   c. Click **Apply** and close the Manage Export Profiles dialog box.
3. When you share the project to an archive, in the **Export profile** list, select the name of your export profile to export only the specified files.

Note Export profiles do not apply changes to referenced projects. When you share your project, MATLAB exports the entire referenced projects.

Before sharing projects with other users, it can be useful to examine the required toolboxes for your project. See “Find Required Products and Toolboxes” on page 18-19.

See Also

Related Examples

- “Create a New Project from an Archived Project” on page 16-20
- “Export a Subset of a Project Using an Export Profile”
- “Share Project by Email” on page 17-31

More About

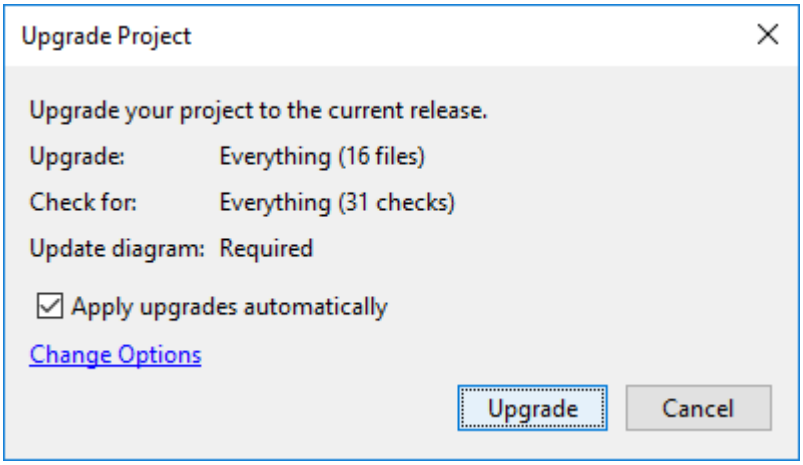
- “Sharing Projects” on page 17-30

Upgrade All Project Models, Libraries, and MATLAB Code Files

Tip Before upgrading, if you put your project under source control, you can easily revert changes later if you want. See “Add a Project to Source Control” on page 19-5.

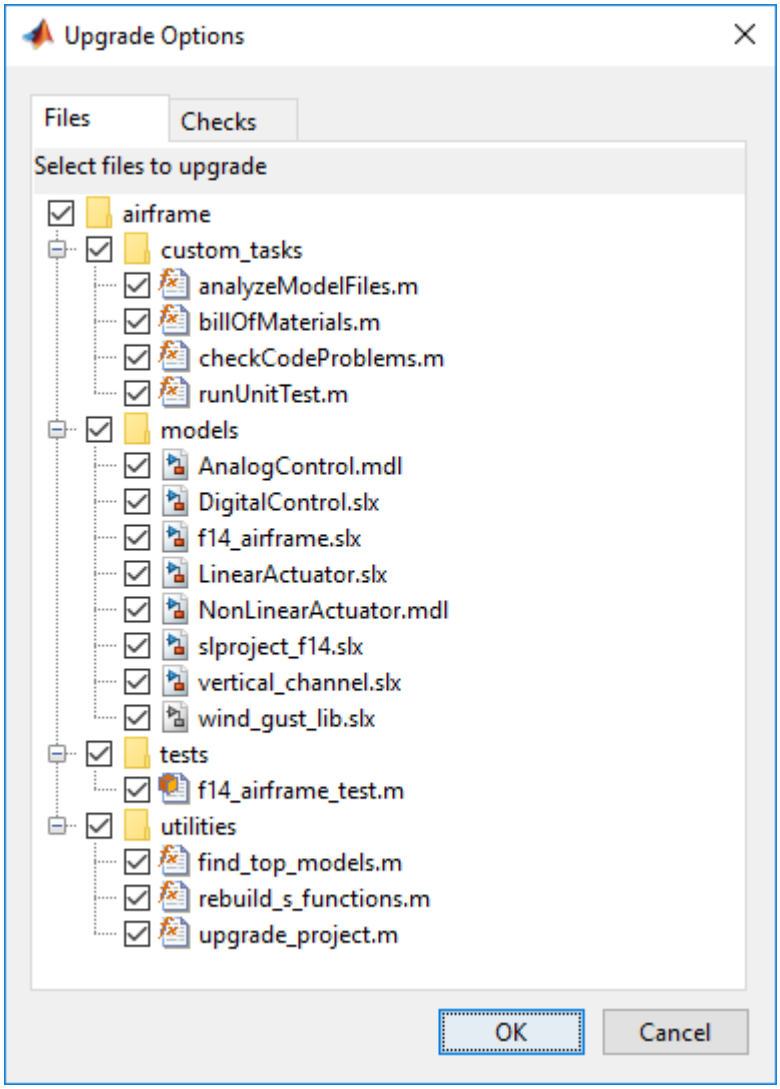
Upgrade all models, libraries, and MATLAB code files in your project to the latest release using a simple workflow. The Upgrade Project tool can apply all fixes automatically when possible, upgrade all model hierarchies in the project at once, and produce a report. You do not need to open the Upgrade Advisor.

1 On the Project tab, select **Run Checks** > **Upgrade**.



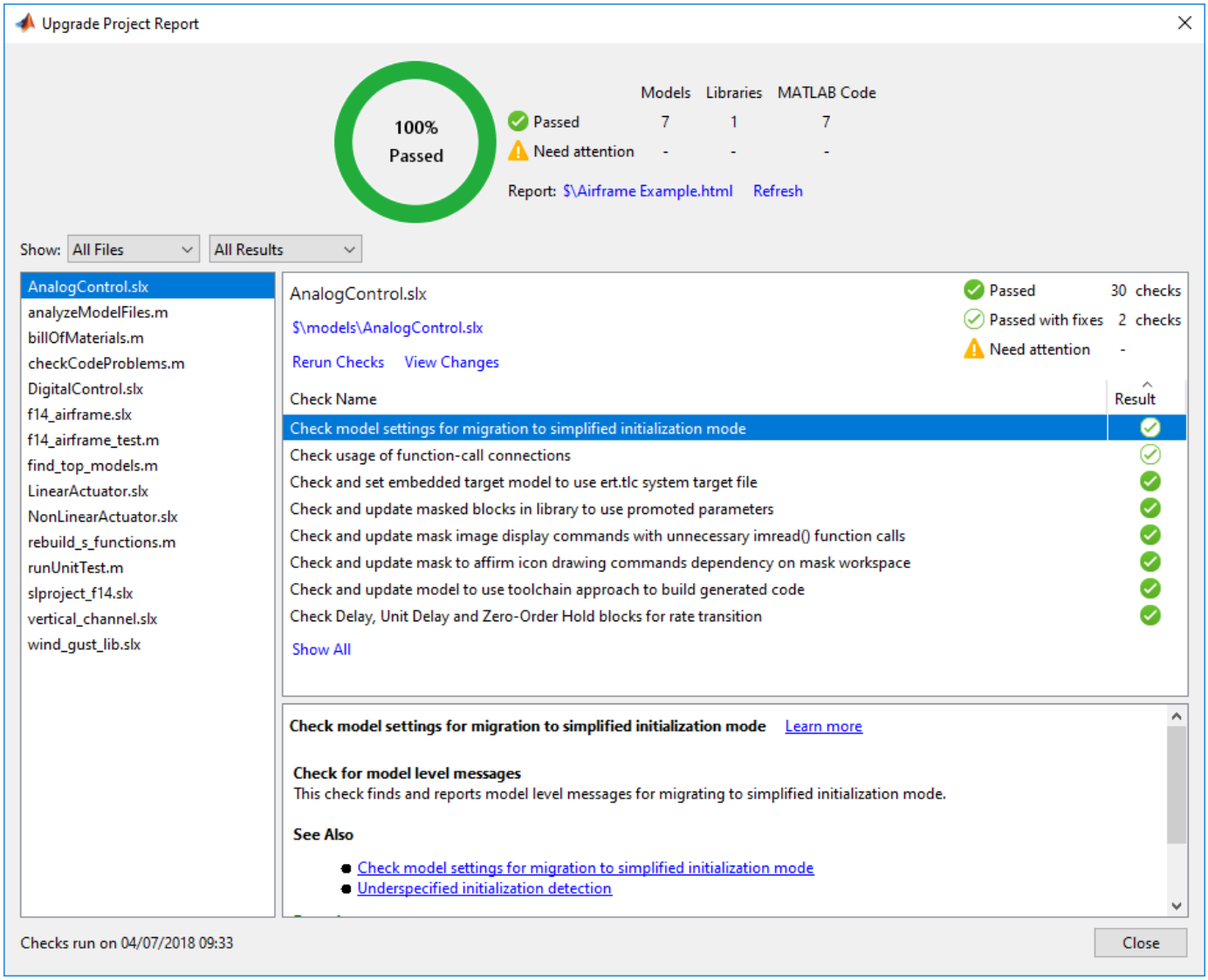
2 In the Upgrade Project dialog box, to upgrade all files, run all checks, and apply fixes automatically where possible, click **Upgrade**. If you want to change the settings, use these options before clicking **Upgrade**:

- If you want to run upgrade checks but not apply fixes automatically where possible, clear the check box **Apply upgrades automatically**.
- If you want to change which files to upgrade and which checks to run, click **Change Options**. In the Upgrade Options dialog box, clear check boxes for models and checks you want to exclude from the upgrade. For example, you might want to exclude checks that require an **Update Diagram**.



When you click **Upgrade**, the tool runs the checks and applies fixes if specified. Upgrading can take several minutes.

3 Examine the Upgrade Project Report. The summary at the top shows how many files passed and how many files require attention.



a Select files in the left list to view check results on the right. By default, the left list shows any files that need attention. Show instead all files, files types, all results, files that passed, or files that passed with fixes, by using the **Show** controls.

b Select checks in the right list to read details of results and any applied fixes in the lower pane. Examine checks marked as needing attention, with an orange circle in the **Result** column. For details of upgrading libraries, see “Upgrade Libraries” on page 17-38.

c If your project is under source control, you can examine the upgrade changes in your files using a comparison report. To see the differences before and after upgrade, in the Upgrade Project Report, click **View Changes**.

4. The project saves an HTML report of the upgrade results in the project root folder. To open the published report, click the **Report** link at the top of the Upgrade Project Report.
5. To close the interactive report, click **Close**.

Upgrade Libraries

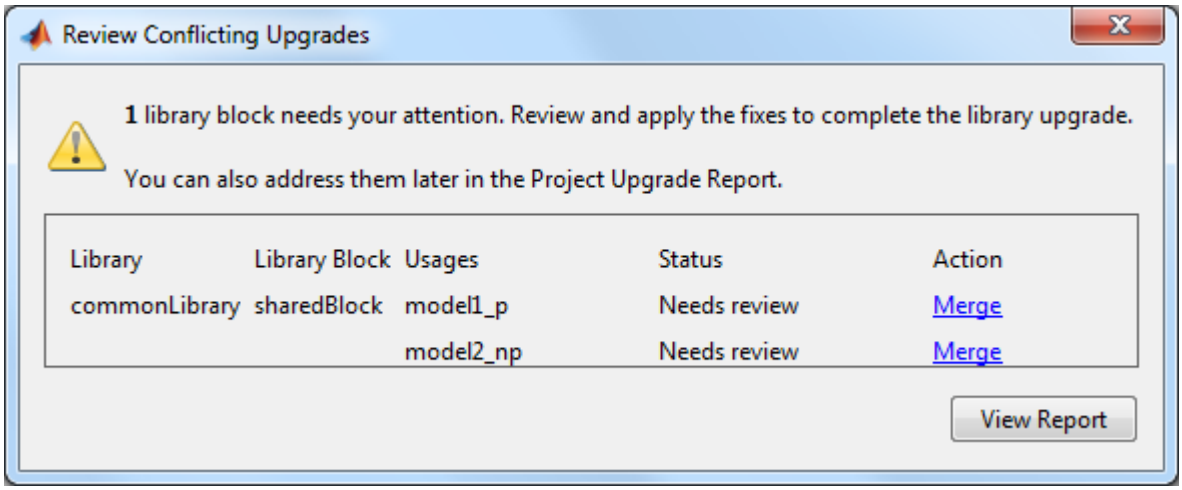
The project automatically runs all upgrade checks on multiple libraries, including any checks that require an Update Diagram.

You cannot run Update Diagram on a library, so project upgrade runs the Update Diagram checks in the models where the library blocks are used. This means that project upgrade can only fully upgrade library blocks that are used in a model. If the library block is used in a model, project upgrade automatically runs all checks, including Update Diagram checks, and then upgrades the block in the library.

If a library block is not used in any project model, then the check **Run checks that require Update Diagram on library blocks** is marked as needing attention, with an orange circle in the **Result** column. Select the check and in the details pane you see the message `Unable to upgrade blocks unused by a model`.

- To upgrade unused library blocks, use the blocks in a model and then upgrade.
- If you want to upgrade library blocks that use forwarding tables, disable the library link and save the model before upgrading, upgrade and then restore the link.

The upgrade of library blocks depends on the model context. The same library block might be used in multiple models. Linked library blocks inherit attributes from the surrounding models, such as data types and sample rate. The blocks' behavior can differ depending on the context where they are used, and this can lead to conflicting upgrades for Update Diagram checks. If models require a different upgrade of the same library block, you are prompted to view and resolve the upgrade conflict.



If you need to review conflicting upgrades, click **Merge**. Alternatively you can review conflicting upgrades later from the report. Review changes in the comparison report and choose which upgrades to save.

See Also

`upgradeadvisor`

More About

- “What Are Projects?” on page 16-3

- “About Source Control with Projects” on page 19-2

Analyze Model Dependencies

In this section...

Examine models, subsystems, and libraries referenced directly or indirectly by the model using the Dependency Analyzer. Use the dependency graph to identify all required files and products. To package, share, or put your design under source control, create a project from your model. For more details, see “Create Project from the Dependency Graph” on page 17-44.

Open and Explore Dependency Graph

1. Open the `sldemo_mdlref_depgraph` model.

```
open_system("sldemo_mdlref_depgraph")
```

2. To open the model dependency graph, on the **Modeling** tab, on the far right of the **Design** section, click the arrow. Under **System Design**, click **Dependency Analyzer**.

The Dependency Analyzer opens the dependency graph using the **Model Hierarchy** view by default. To switch to the model instances view, in the **Views** section, click **Model Instances**. For more details, see “Model Dependency Views” on page 17-41.

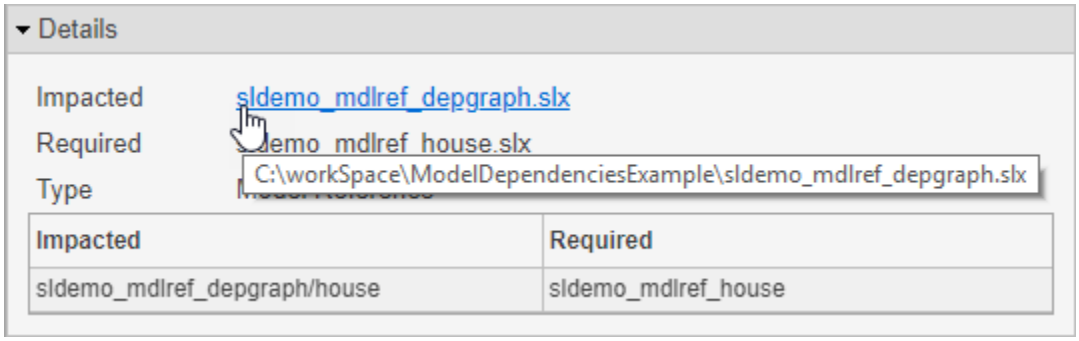
After you run the first dependency analysis, subsequent analyses incrementally update the results. The Dependency Analyzer determines which files changed since the last analysis and updates the dependency data for those files. To perform a complete analysis, in the Dependency Analyzer, select **Analyze > Reanalyze All**.

To analyze the dependencies inside external toolboxes, select **Analyze > External Toolboxes**. For more details about available options, see “Analysis Scope” on page 18-4.

3. To view the dependencies laid out horizontally, in the **Layout** section, click **Horizontal**.

4. In the dependency graph, double-click a box to open the corresponding model in the Simulink® editor.

5. To see more information about how two files are related, select their dependency arrow. In the **Properties** pane, in the **Details** section, you can see the full paths of the files you are examining, the dependency type, and where the dependency is introduced.



To open the file and highlight where the dependency is introduced, in the **Details** section, click the link under **Impacted**.

For example, to open the `sldemo_mdlref_depgraph` model and highlight where the dependency to the `sldemo_mdlref_house` block is introduced, select the dependency arrow between `sldemo_mdlref_depgraph` and `sldemo_mdlref_house`. In the **Properties** pane on the right, under **Impacted**, click `sldemo_mdlref_depgraph/house`.

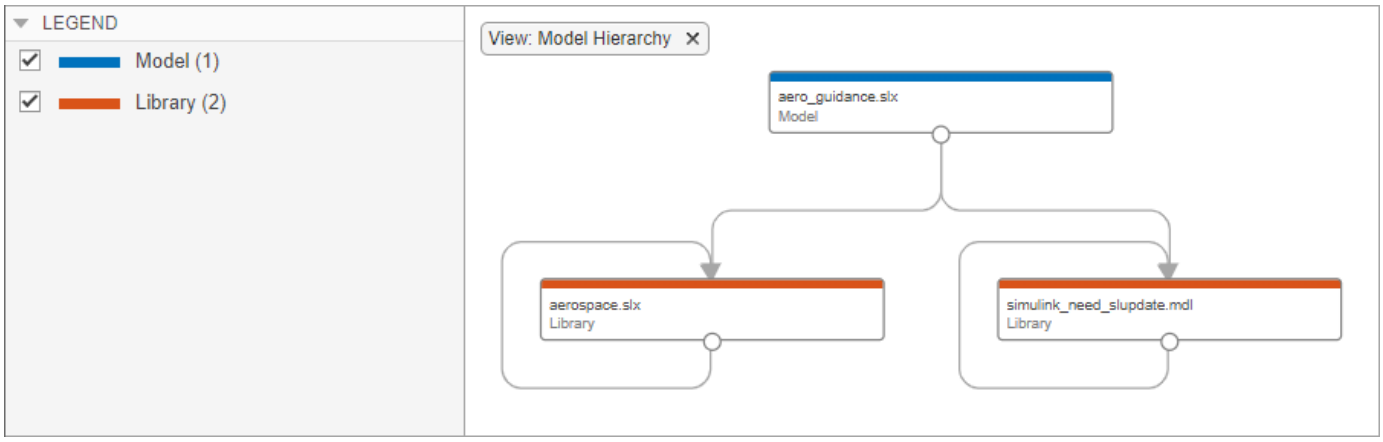
Model Dependency Views

You can explore model dependencies using the model hierarchy or the model instances views.

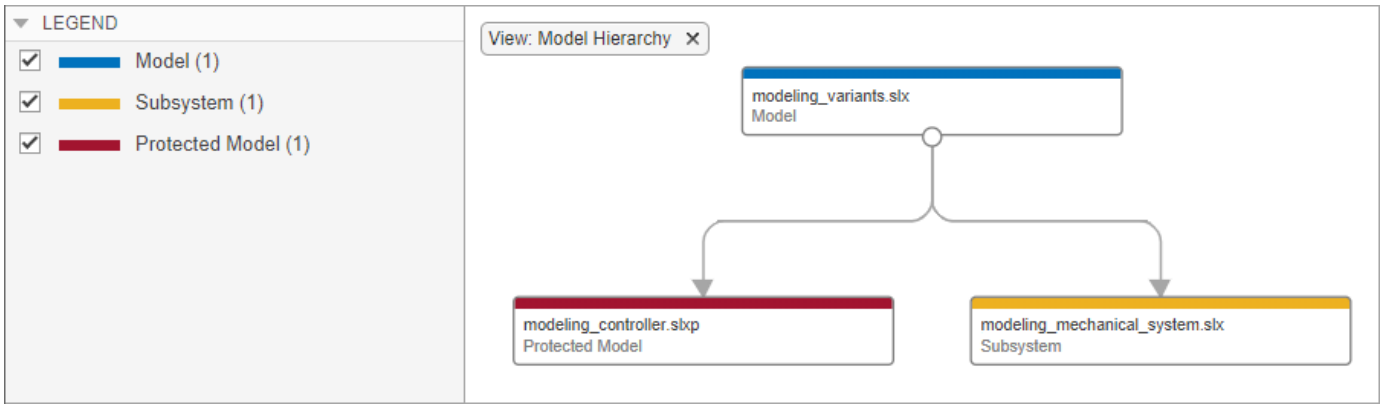
Model Hierarchy View

The **Model Hierarchy** view shows the model, subsystem, library and data dictionary files referenced by a top model.

- A referenced file appears only once in the view even if it is referenced more than once in the model.
- Blue boxes represent model files, red boxes represent libraries, and yellow boxes represent subsystem references. Arrows represent dependencies. For example, the arrows in this example indicate that the `aero_guidance` model references two libraries: `aerospace` and `simulink_need_slupdate`.

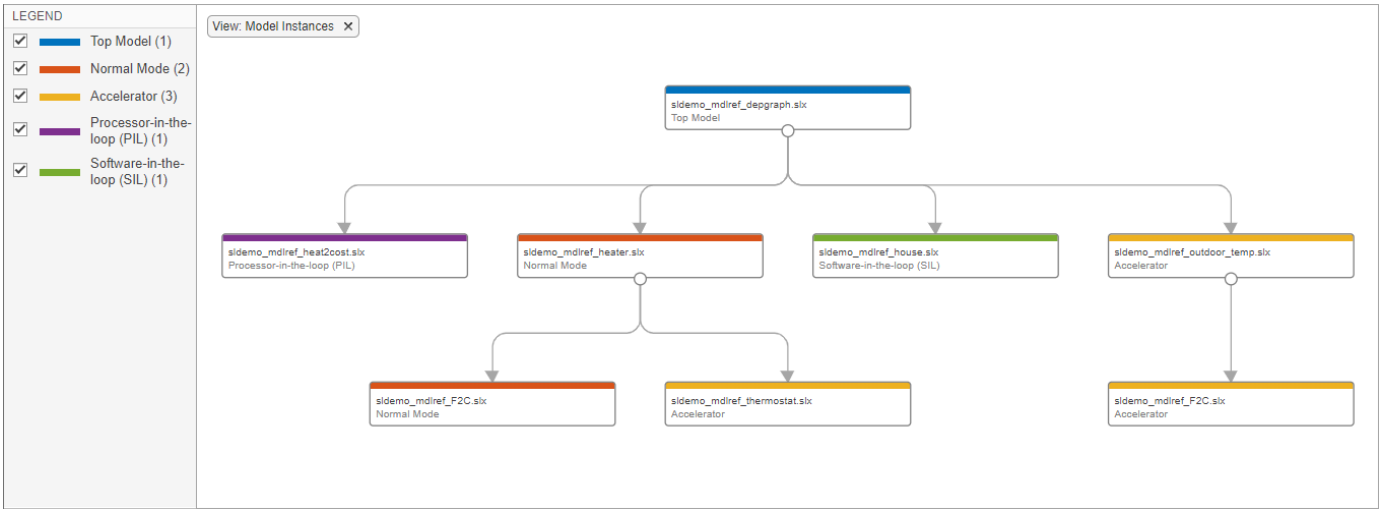


- An arrow from a library that points to itself indicates that the library references itself. Blocks in the library reference other blocks in that same library. The example view shows that the libraries `aerospace` and `simulink_need_slupdate` reference themselves.
- Dark red boxes represent protected models (`.slxp` files). You cannot open or edit protected referenced models. See “Reference Protected Models from Third Parties” on page 8-13.



Model Instances View

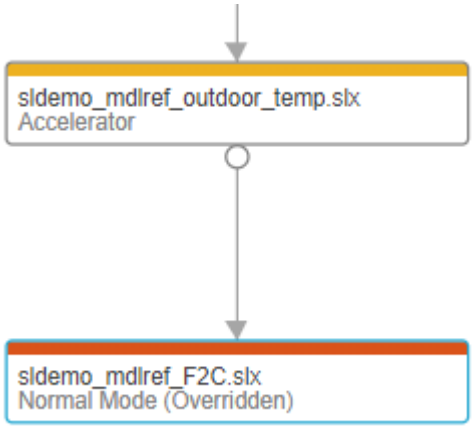
The **Model Instances** view shows every reference to a model in a model reference hierarchy with the top model at the root of the hierarchy. Boxes represent the top model and its references. See “Model References”.



- If a model hierarchy references the same model more than once, the referenced model appears multiple times in the instance view, once for each reference. This example graph shows that the model reference hierarchy for `sldemo_mdlref_depgraph` contains two references to the model `sldemo_mdlref_F2C`.
- Yellow boxes represent accelerated-mode instances, red boxes represent normal-mode instances, purple boxes represent processor-in-the-loop mode instances, and green boxes represent software-in-the-loop mode instances. See “Choose Simulation Modes for Model Hierarchies” on page 8-39.

The previous example graph shows that one of the references to `sldemo_mdlref_F2C` operates in normal mode and the other operates in accelerated mode.

- The Dependency Analyzer detects when a simulation mode is overridden and appends `(Overridden)` to the simulation mode. If a referenced model is configured to run in normal mode and it runs in accelerator mode, its simulation mode is overridden. This occurs when another model that runs in accelerator mode directly or indirectly references it.



Find Required Products

To find required products and toolboxes for a file in your design, select a box in the dependency graph. The Dependency Analyzer shows the list of required products by your selection in the **Products** section in the **Properties** pane.

To find required toolboxes for the whole design, click the graph background to clear all selection. Examine the list of products in the **Products** section in the **Properties** pane.

To highlight files that use a certain product in the graph, for example *`Simulink`*, in the **Products** section, in the **Properties** pane, point to product and click the magnifying glass icon .

To go through these files, use the arrows in the search box (e.g., **Files using** "*`productName`*").

To undo the highlighting, close the search box.

To investigate further, you can list the files that use a product and examine where in these files the dependency is introduced. In the **Products** section, in the **Properties** pane, point to a product and click the search folder icon .

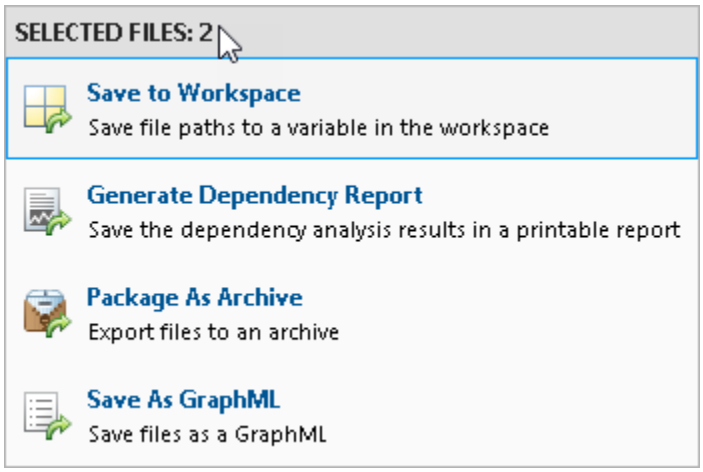
Export Dependency Analysis Results

To export all the files displayed in the dependency graph, click the graph background to clear the selection on all files. In the Dependency Analyzer toolstrip, in the **Export** section, click **Export**. Select from the available options:

- **Export to Workspace** — Save file paths to a variable in the workspace.
- **Generate Dependency Report** — Save dependency analysis results in a printable report (HTML, Word, or PDF).

- **Package As Archive** — Export files in the graph as an archive.
- **Save As GraphML** — Save dependency analysis results as a GraphML file.

You can also export a subset of files in the graph. Select the files, then click **Export**. The menu displays how many files are selected. The Dependency Analyzer exports only the selected files.



Note When you use **Package As Archive**, the Dependency Analyzer includes the selected files and all their dependencies in the archive.

Create Project from the Dependency Graph

To package, share, or put your design under source control, create a project from your model. You can create a project from the model dependency graph.

To create a project from all the files displayed in the dependency graph, click the graph background. This action clears all selected files.

1. In the Dependency Analyzer toolstrip, in the **Export** section, click **Create Project**.
2. In the **New Project** window, click **Create**.

   The Dependency Analyzer creates a project and reloads the graph.

You can also create a project from a subset of files in the graph. Select the files, then click **Create Project**. The Dependency Analyzer includes the selected files and all their dependencies in the project.

See Also

More About

- “Custom Libraries”
- “Model References”

View Linked Requirements in Models and Blocks

In this section...

Requirements Traceability in Simulink

If your Simulink model has links to requirements in external documents, you can review these links. To identify which model objects satisfy certain design requirements, use the following requirements features available in Simulink software:

- Highlighting objects in your model that have links to external requirements
- Viewing information about a requirements link
- Navigating from a model object to its associated requirement
- Filtering requirements highlighting based on specified keywords

Having a Simulink Requirements license enables you to perform the following additional tasks, using the Requirements Management Interface (RMI):

- Adding new requirements
- Changing existing requirements
- Deleting existing requirements
- Applying user tags to requirements
- Creating reports about requirements links in your model
- Checking the validity of the links between the model objects and the requirements documents

Highlight Requirements in a Model

You can highlight a model to identify which objects in the model have links to requirements in external documents. Both the Simulink Editor and the Model Explorer provide this capability.

- “Highlight a Model Using the Simulink Editor” on page 17-46
- “Highlight a Model Using the Model Explorer” on page 17-47

Note If your model contains a Model block whose referenced model contains requirements, those requirements are not highlighted. If you have Simulink Requirements, you can view this information only in requirements reports. To generate requirements information for referenced models and then see highlighted snapshots of those requirements, follow the steps in “Report for Requirements in Model Blocks” (Simulink Requirements).

Highlight a Model Using the Simulink Editor

If you are working in the Simulink Editor and want to see which model objects in the `slvnvdemo_fuelsys_officereq` model have requirements, follow these steps:

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

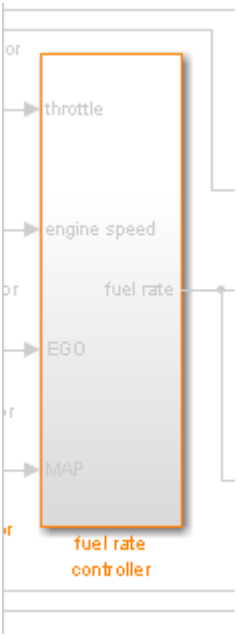
2. Select **Coverage Highlighting** from the **Coverage** app.

   Two types of highlighting indicate model objects with requirements:

   - Yellow highlighting indicates objects that have requirements links for the object itself.

   - Orange outline indicates objects, such as subsystems, whose child objects have requirements links.

   Objects that do not have requirements are colored gray.

3. You remove the highlighting from the model from the **Coverage** app. Alternatively, you can right-click anywhere in the model, and select **Remove Highlighting**.

While a model is highlighted, you can still manage the model and its contents.

Highlight a Model Using the Model Explorer

If you are working in Model Explorer and want to see which model objects have requirements, follow these steps:

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

2. In the **Modeling** tab, click **Model Explorer**.
3. To highlight all model objects with requirements, click the **Highlight items with requirements on model** icon ().

   The Simulink Editor window opens, and all objects in the model with requirements are highlighted.

Note If you are running a 64-bit version of MATLAB, when you navigate to a requirement in a PDF file, the file opens at the beginning of the document, not at the specified location.

View Information About a Requirements Link

Using Simulink, you can view detailed information about a requirements link, such as identifying the location and type of document that contains the requirement.

Note You can modify the requirements information only if you have a Simulink Requirements license.

For example, to view information about the requirements link from the MAP Sensor block in the `slvnvdemo_fuelsys_officereq` example model, follow these steps:

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

2. Right-click the MAP sensor block, and select **Requirements** > **Edit/Add Links**.

   The Requirements dialog box opens and displays the following information about the requirements link:

   - The description of the link (which is the actual text of the requirement).
   - The Microsoft Excel workbook named `slvnvdemo_FuelSys_TestScenarios.xlsx`, which contains the linked requirement.
   - The requirements text, which appears in the named cell `Simulink_requirement_item_2` in the workbook.
   - The user tag `test`, which is associated with this requirement.

Navigate to Requirements from a Model

Navigate from the Model Object

You can navigate directly from a model object to that object's associated requirement. When you take these steps, the external requirements document opens in the application, with the requirements text highlighted.

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

2. Open the fuel rate controller subsystem.
3. To open the linked requirement, right-click the Airflow calculation subsystem and select **Requirements > 1. “Mass airflow estimation”**.

   The Microsoft Word document `slvnvdemo_FuelSys_DesignDescription.docx`, opens with the section **2.1 Mass airflow estimation** selected.

Note If you are running a 64-bit version of MATLAB, when you navigate to a requirement in a PDF file, the file opens at the top of the page, not at the bookmark location.

Navigate from a System Requirements Block

Sometimes you want to see all the requirements links at a given level of the model hierarchy. In such cases, you can insert a System Requirements block to collect all requirements links in a model or subsystem. The System Requirements block lists requirements links for the model or subsystem in which it resides; it does not list requirements links for model objects inside that model or subsystem, because those are at a different level of the model hierarchy.

In the following example, you insert a System Requirements block at the top level of the `slvnvdemo_fuelsys_officereq` model, and navigate to the requirements using the links inside the block.

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

2. Enable **Model Highlighting** in the **Coverage** app.
3. Open the fuel rate controller subsystem.

   The Airflow calculation subsystem has a requirements link.
4. Open the Airflow calculation subsystem.
5. In the Simulink toolstrip, click **Library Browser**.
6. In the **Libraries** tree view, select **Simulink Requirements**.

   This library contains only one block—the System Requirements block.
7. Drag a System Requirements block into the Airflow calculation subsystem.

   The RMI software collects and displays any requirements links for that subsystem in the System Requirements block.
8. In the System Requirements block, double-click **1. “Mass airflow subsystem”**.

The Microsoft Word document, `slvnvdemo_FuelSys_DesignDescription.docx`, opens, with the section **2.1 Mass airflow estimation** selected.

Filter Requirements in a Model

Filtering Requirements Highlighting by User Tag

Some requirements links in your model can have one or more associated user tags. User tags are keywords that you create to categorize a requirement, for example, `design` or `test`.

For example, in the `slvnvdemo_fuelsys_officereq` model, the requirements link from the MAP sensor block has the user tag `test`.

To highlight only all the blocks that have a requirement with the user tag `test`:

1. Open the example model:

   ```
   slvnvdemo_fuelsys_officereq
   ```

2. In the Simulink Editor, on the **Apps** tab, click **Requirements Viewer**. Then on the **Requirements Viewer** tab, click **Link Settings**.

   The Requirements Settings dialog box opens. If you do not have a Simulink Requirements license, the **Filters** tab is the only option available.

   By default, your model has no requirements filtering enabled.

3. Select **Filter links by user tags when highlighting and reporting requirements**.
4. In the **Include links with any of these tags** text box, delete `design`, and enter `test`.
5. Press **Enter**.
6. Highlight the `slvnvdemo_fuelsys_officereq` model for requirements. On the **Requirements Viewer** tab, click **Highlight Links**.

   In the top-level model, only the MAP sensor block and the Test inputs block are highlighted.

7. To disable the filtering by user tag, on the **Requirements Viewer** tab, click **Link Settings**, and then clear **Filter links by user tags when highlighting and reporting requirements**.

   The model highlighting updates immediately.

Filtering Options for Highlighting Requirements

On the **Filters** tab, you select options that designate which objects with requirements are highlighted. The following table describes these settings, which apply to all requirements in your model for the duration of your MATLAB session.

Option	Description
Filter links by user tags when highlighting and reporting requirements	Enables filtering for highlighting and reporting, based on specified user tags.

Option	Description
Include links with any of these tags	Highlights all objects whose requirements match at least one of the specified user tags. The tag names must match exactly. Separate multiple user tags with commas or spaces.
Exclude links with any of these tags	Excludes from the highlighting all objects whose requirements match at least one of the specified user tags. The tag names must match exactly. Separate multiple user tags with commas or spaces.
Apply same filters in context menus	Disables navigation links in context menus for all objects whose requirements do not match at least one of the specified user tags.
Under **Link type filters**, **Disable DOORS surrogate item links in context menus**	Disables links to IBM® Rational® DOORS® surrogate items from the context menus when you right-click a model object. This option does not depend on current user tag filters.

Project Dependency Analysis

What Is Dependency Analysis?

Every design, whether it is a Simulink model or a project, requires a set of files and products to run successfully. Dependencies include data files, model references, linked libraries, MATLAB and C/C++ code, Stateflow charts, and requirements documents.

Dependency Analysis for Projects

You perform a dependency analysis to analyze project structure and discover required files and products. You can use the dependency graph to identify all required files and products, and perform an impact analysis.

On the **Project** tab, in the **Tools** section, click **Dependency Analyzer**. For more details, see “Run a Dependency Analysis” on page 18-7.

With the Dependency Analyzer, you can:

- Find required products and toolboxes. See “Find Required Products and Toolboxes” on page 18-19.
- Check project dependencies and problems before sharing, packaging, or submitting your project to source control. See “Check Dependency Results and Resolve Problems” on page 18-23.
- Perform an impact analysis to find the impact of changing particular files. See “Perform an Impact Analysis” on page 18-17.
- Export the dependency analysis results as workspace variables, or `.graphml` files, or send the files for custom task processing. Exporting the results enables further processing or archiving. See “Export Dependency Analysis Results” on page 18-29.

Tip For an example showing how to perform file-level impact analysis to find and run the tests affected by modified files, see “Perform Impact Analysis with a Project”.

Dependency Analysis for Models

You perform a dependency analysis to examine models, subsystems, and libraries referenced directly or indirectly by the model. You can use the dependency graph to identify all required files and products.

On the **Modeling** tab, on the far right of the **Design** section, click the arrow. Under **System Design**, click **Dependency Analyzer**.

- You can explore model dependencies using the model hierarchy or the model instances views. For more details, see “Analyze Model Dependencies” on page 17-40.
- To package, share, or put your design under source control, create a project from your model. For more details, see “Create a Project from a Model” on page 16-12. Perform a project dependency analysis to explore the dependency graph using source control and project-specific views.

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Check Dependency Results and Resolve Problems” on page 18-23
- “Perform an Impact Analysis” on page 18-17
- “Find Requirements Documents in a Project” on page 18-28

Dependency Analyzer Scope and Limitations

Analysis Scope

The Dependency Analyzer identifies the required files and toolboxes for your project or model. The analysis covers a wide range of dependencies, including model references, subsystem references, linked libraries, MATLAB and C/C++ code, Stateflow charts, data files, S-functions, and requirements documents.

When the Dependency Analyzer encounters MATLAB code, such as in a model or block callback, or in a `.m` file S-function, it attempts to identify the files it references. For more information, see “Analysis Limitations” on page 18-4.

For files under the MATLAB root folder, the Dependency Analyzer only shows required products. It does not analyze dependencies.

The Dependency Analyzer identifies dependencies inside user-defined toolboxes and dependencies that were introduced by code generation or by MATLAB code in model parameters. These options are off by default because they can be time consuming for large designs.

To specify the scope of the analysis, in the Dependency Analyzer toolstrip, click **Analyze** and select one or more of the following options:

Option	Default	Description
C/C++ Code	On	Analyze dependencies introduced by C/C++ code files.
External Toolboxes	Off	Analyze dependencies inside user-defined toolboxes.
Model Parameters	Off	Analyze dependencies introduced by MATLAB code in model block parameters.
Generated Code Traceability	Off	Analyze dependencies introduced by code generated from a model.

Analysis Limitations

- The Dependency Analyzer has limitations specific to MATLAB code analysis:
  - The Dependency Analyzer only identifies function input arguments when they are literal character vectors or strings:

    ```
    load("mydatafile")
    load mydatafile
    ```

    If you define a file name as a variable and pass it to a function, the Dependency Analyzer is unable to identify the dependency. In the following example, since the code is not executed, the Dependency Analyzer does not have the value of `str`. The Dependency Analyzer might report a missing dependency.

    ```
    str = "mydatafile";
    load(str);
    ```

This limitation extends to functions similar to `load`, such as `fopen`, `xlsread`, `importdata`, `dlmread`, `eval`, or `imread`.

- The Dependency Analyzer does not always determine type automatically. Depending on the way you call an object method, the Dependency Analyzer might confuse a method with a function and report a missing dependency.

  In MATLAB, you can call an object method in two different ways. For example, for an object `p`, you can call the method `addFile` using the function notation:

  ```
  p = currentProject;
  addFile(p,"myfile");
  ```

  or by using the dot notation:

  ```
  p = currentProject;
  p.addFile("myfile");
  ```

  If you do not declare the type of `p` explicitly, the Dependency Analyzer might confuse a method call that uses a function notation with a function call. The analyzer reports `addFile` as a missing dependency.

  To work around this limitation, use dot notation to call a method or use arguments to explicitly declare the variable type in your function:

  ```
  function myfunction(p)

     arguments
        p matlab.project.Project
     end

     addFile(p,"myfile");
  end
  ```

- The Dependency Analyzer might not report certain blocksets or toolboxes required by a model.

  The Dependency Analyzer is unable to detect blocksets that do not introduce dependencies on any files, such as Fixed-Point Designer™.

  To include dependencies that the analysis cannot detect, add the file that introduces the dependency to your project. To create a project from your model, see “Create a Project from a Model” on page 16-12.

- The Dependency Analyzer might not report dependencies for dynamic content in masked blocks.

  Based on the parameters of the masked blocks, dynamic masks can modify the masked subsystem and change the block dependencies. If the dynamic mask is in a library, the Dependency Analyzer is unable to detect the dynamic changes.

- The Dependency Analyzer does not support Simulink functions called from MATLAB function blocks.
- The Dependency Analyzer does not support Stateflow charts that use MATLAB as the action language.
- Some MathWorks products and toolboxes share code and Simulink libraries. The Dependency Analyzer might report dependencies on all of them.

  To investigate where shared code is used, in the **Properties** panel, in the **Products** section, point to a product under **Shared Functionalities Among:** and click the search folder icon  .

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Check Dependency Results and Resolve Problems” on page 18-23
- “Perform an Impact Analysis” on page 18-17
- “Find Requirements Documents in a Project” on page 18-28

Run a Dependency Analysis

Note You can analyze only files that are in your project. If your project is new, add files to the project before running a dependency analysis. See “Add Files to the Project” on page 16-18.

To investigate dependencies, run a dependency analysis on your project. On the **Project** tab, in the **Tools** section, click **Dependency Analyzer**. Alternatively, in the project **Views** pane, select **Dependency Analyzer** and click **Analyze**.

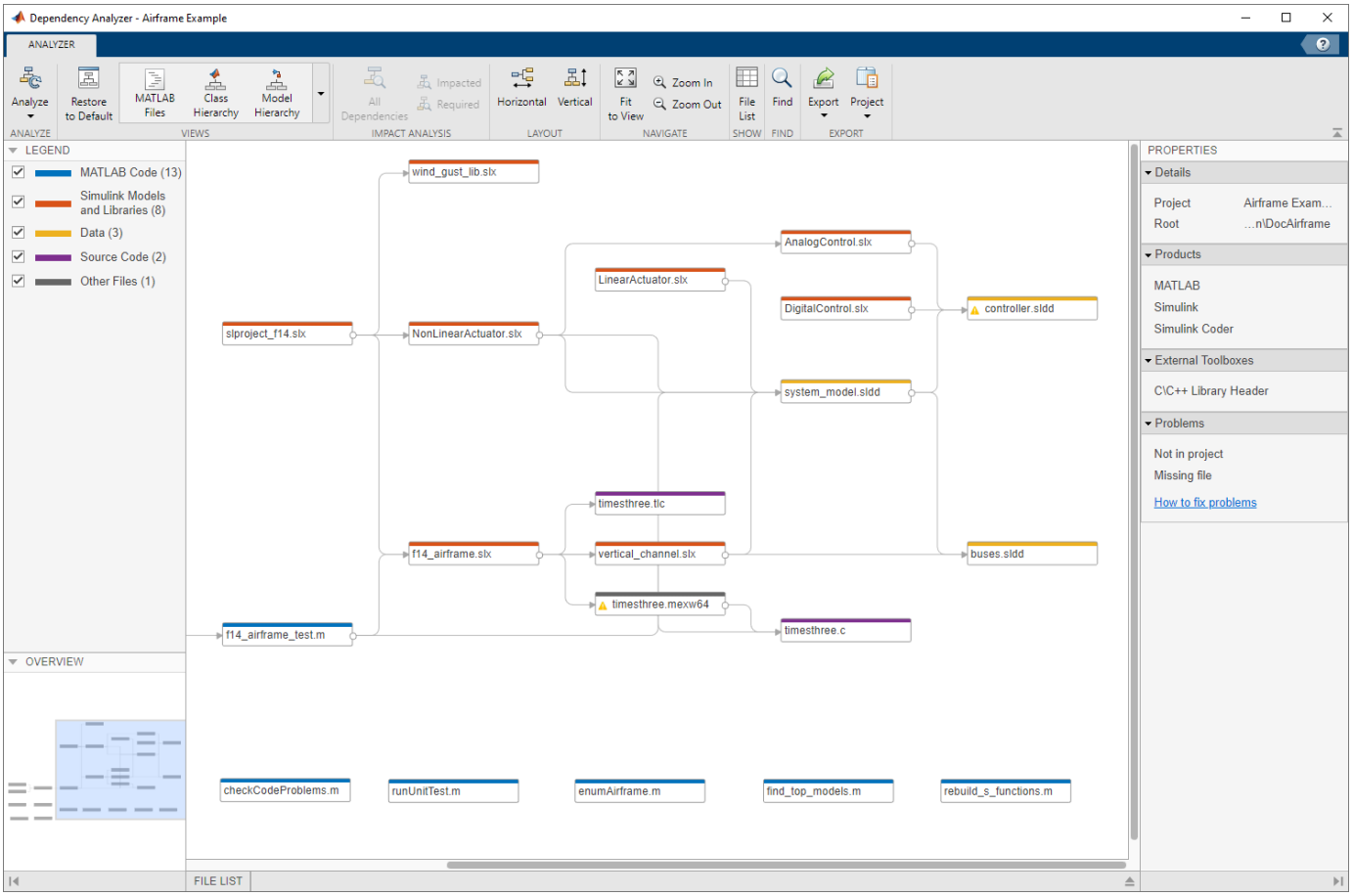
To analyze the dependencies of specific files, in the dependency graph, select the files. In the **Impact Analysis** section, click **All Dependencies** or use the context menu and select **Find All Dependencies**.

To analyze the dependencies inside external toolboxes, select **Analyze > External Toolboxes**. For more details about available options, see “Analysis Scope” on page 18-4.

You can also check dependencies directly in Project. In the Project **Files** view, right-click the project files you want to analyze and select **Find Dependencies**.

After you run the first dependency analysis of your project, subsequent analyses incrementally update the results. The Dependency Analyzer determines which files changed since the last analysis and updates the dependency data for those files. However, if you update external toolboxes or installed products and want to discover dependency changes in them, you must perform a complete analysis. To perform a complete analysis, in the Dependency Analyzer, click **Analyze > Reanalyze All**.

Note In the Simulink Editor, if an open model, library, or chart belongs to a project, you can find file dependencies. On the **Simulation** tab, select **Project > Run Dependency Analysis**. Simulink analyzes the whole project and shows all dependencies for the file.



For next steps, see:

- “Find Required Products and Toolboxes” on page 18-19
- “Find Dependencies of Selected Files” on page 18-20
- “Check Dependency Results and Resolve Problems” on page 18-23

Tip To try a dependency analysis on example files, see “Perform Impact Analysis with a Project”.

See Also

Related Examples

- “Check Dependency Results and Resolve Problems” on page 18-23
- “Perform an Impact Analysis” on page 18-17
- “Export Dependency Analysis Results” on page 18-29
- “Find Requirements Documents in a Project” on page 18-28

Explore the Dependency Graph, Views, and Filters

If you have not yet run an analysis, on the **Project** tab, in the **Tools** section, click **Dependency Analyzer**.

The dependency graph displays your project structure, dependencies, and how files relate to each other. Each item in the graph represents a file and each arrow represents a dependency. For more details, see “Investigate Dependency Between Two Files” on page 18-9.

By default, the dependency graph shows all files required by your project. To help you investigate dependencies or a specific problem, you can simplify the graph using one of the following filters:

- Use the **Views** to color the files in the graph by type, class, source control status, and label. See “Color Files by Type, Status, or Label” on page 18-10.
- Use the check boxes in the **Legend** pane to filter out a group of files.
- Use the **Impact Analysis** tools to simplify the graph. See “Find Dependencies of Selected Files” on page 18-20.

Select, Pan, and Zoom

- To select an item in the graph, click it.

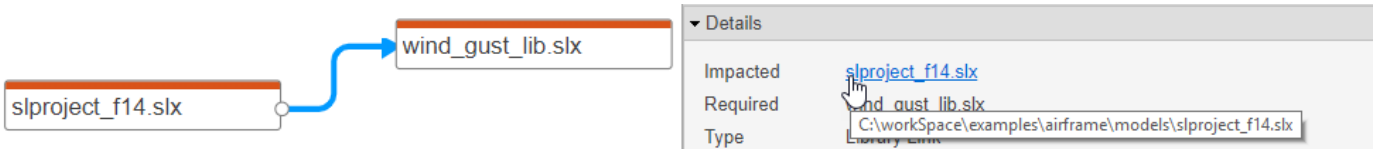
  To select multiple files, press **Shift** and click the files.

  To clear all selection, click the graph background.
- To open a file, double-click it.
- To pan the dependency graph, hold the **Space** key, click and drag the mouse. Alternatively, press and hold the mouse wheel and drag.

  For large graphs, navigate using the **Overview** pane.
- To zoom in and out, in the **Navigate** section, click **Zoom In** and **Zoom Out**. Alternatively, use the mouse wheel.
- To center and fit the dependency graph to view, in the **Navigate** section, click **Fit to View**. Alternatively, press the **Space** bar.

Investigate Dependency Between Two Files

To see more information about how two files are related, select their dependency arrow. In the **Properties** pane, in the **Details** section, you can see the full paths of the files you are examining, the dependency type (such as function call, inheritance, S-function, data dictionary, model reference, and library link), and where the dependency is introduced.

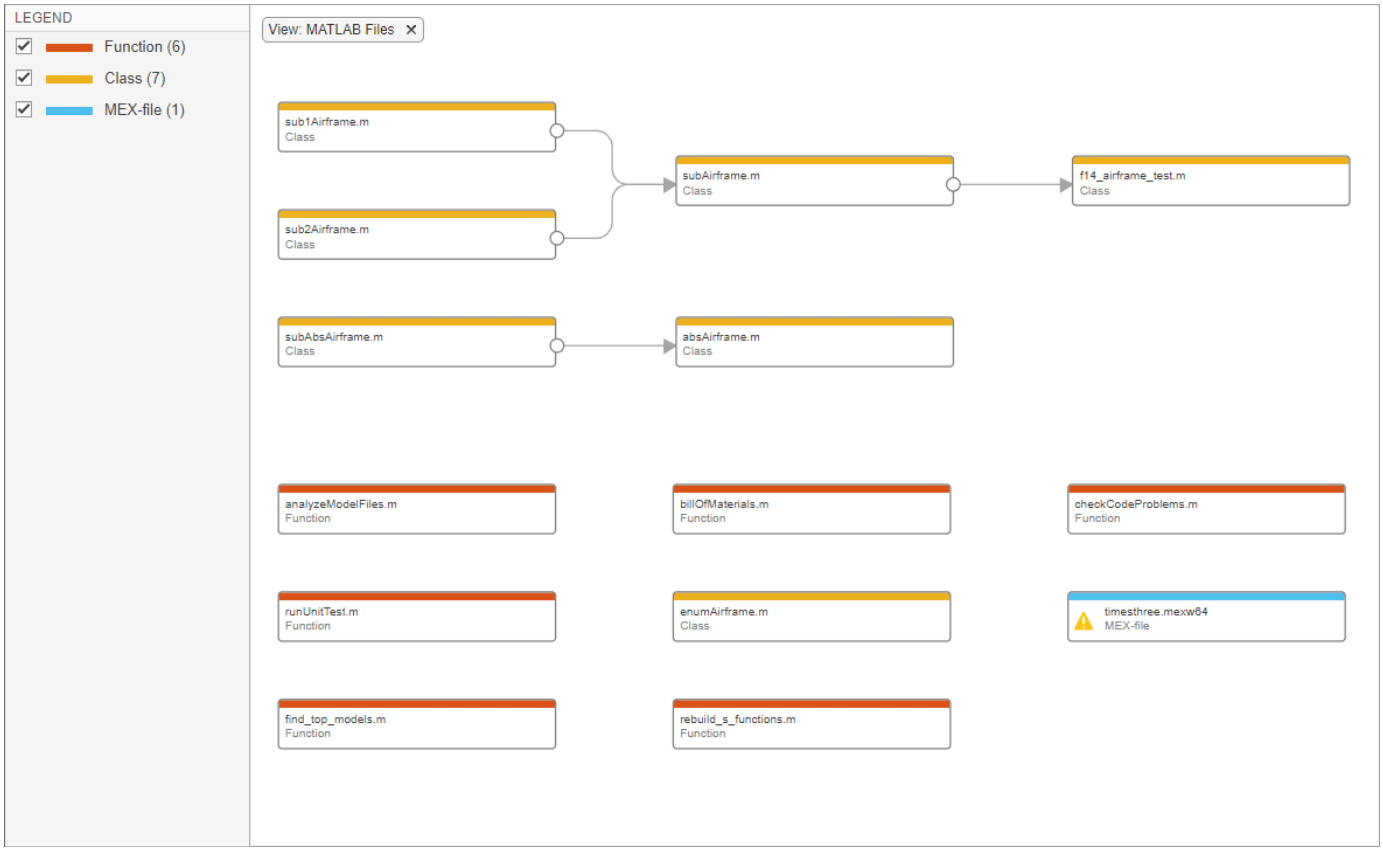


To open the file and highlight where the dependency is introduced, in the **Details** section, click the link under **Impacted**.

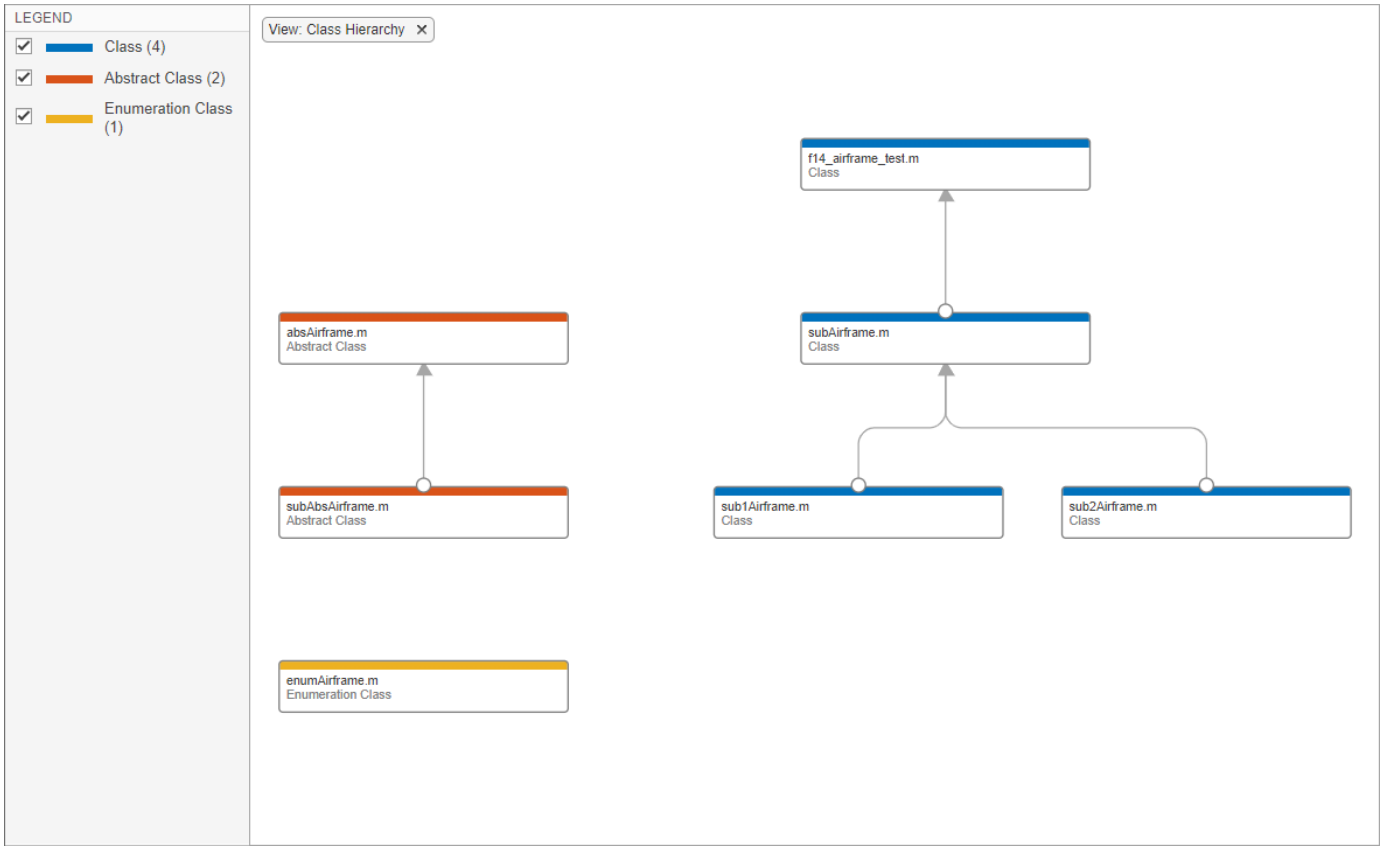
Color Files by Type, Status, or Label

Explore the different views in the **Views** section of the Dependency Analyzer toolstrip to explore your project files dependencies.

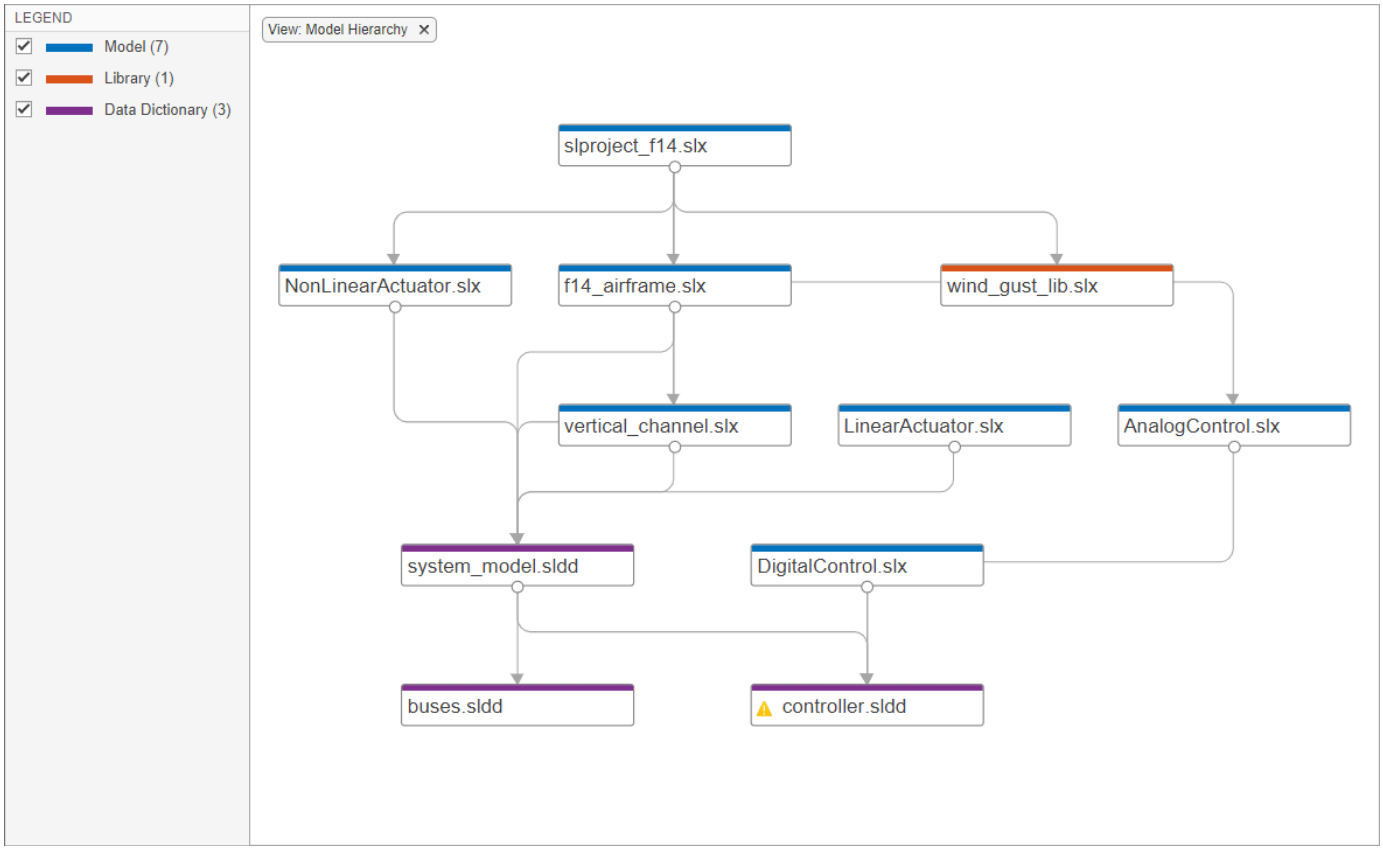
- The **MATLAB Files** view shows only MATLAB files (such as `.m`, `.mlx`, `.p`, `.mlapp`, `.fig`, `.mat`, and `.mex`) in the view and colors them by type.



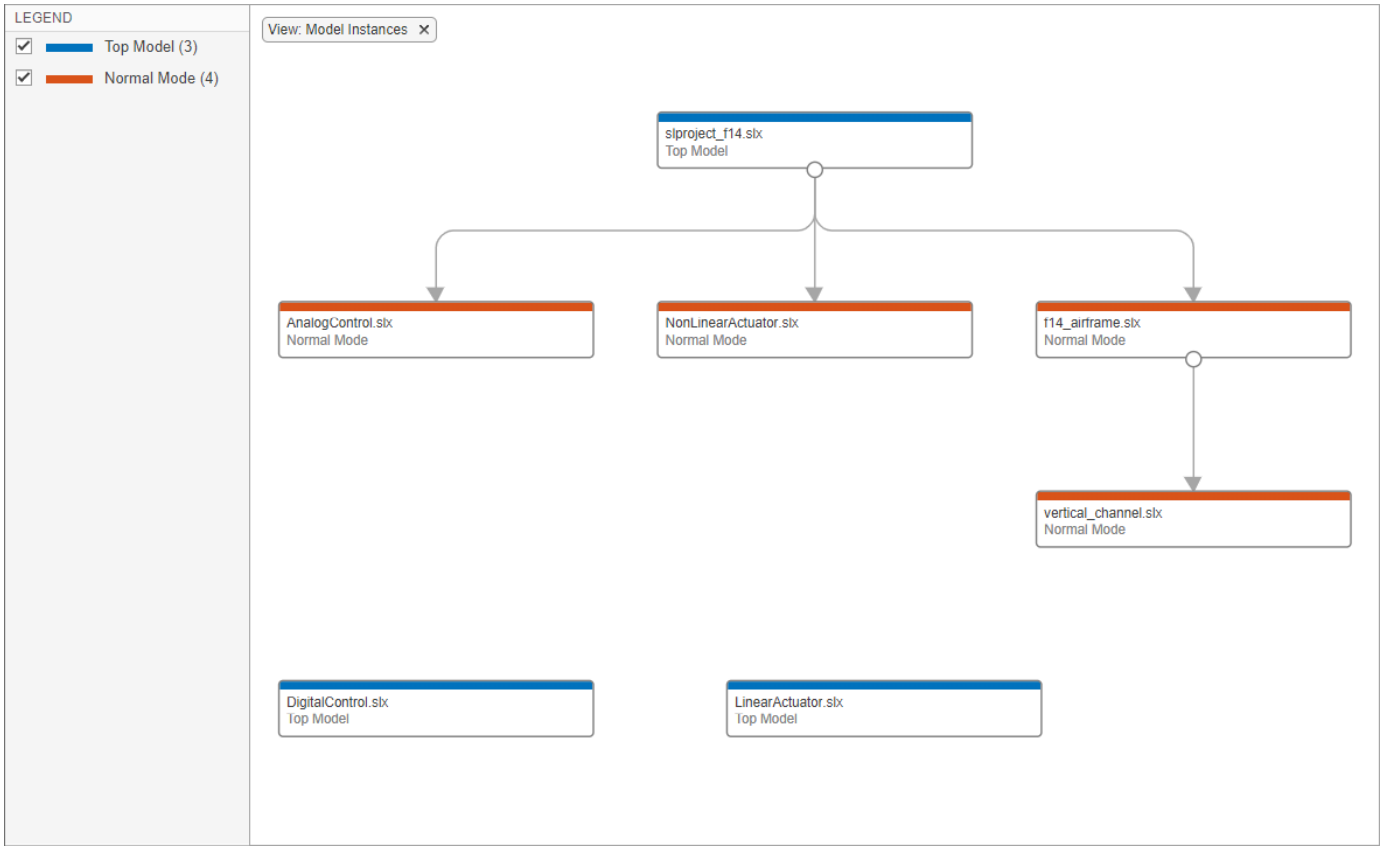
- The **Class Hierarchy** view shows the class inheritance graph and colors the files by type (class, enumeration class, or abstract class). If the class is not on the search path, the Dependency Analyzer cannot determine the class type.



- The **Model Hierarchy** view shows the model, subsystem, library and data dictionary files referenced by a top model. A referenced file appears only once in the view even if it is referenced more than once in the model. For more details, see “Model Hierarchy View” on page 17-41.

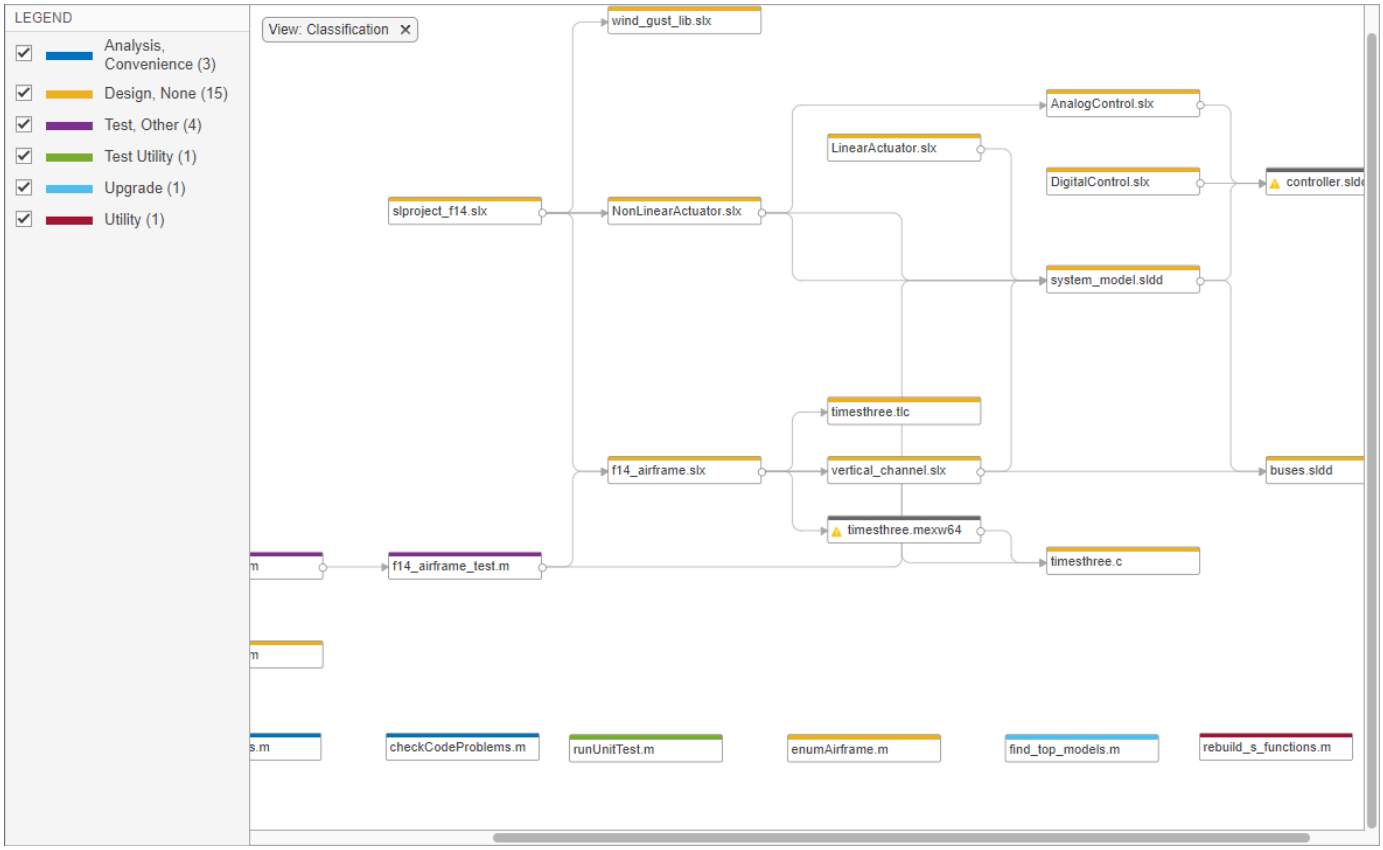


- The **Model Instances** view shows every instance to a model in a model reference hierarchy. A top model is at the root of this hierarchy. If a model hierarchy references the same model more than once, the referenced model appears multiple times in the instance view. For more details, see “Model Instances View” on page 17-42.



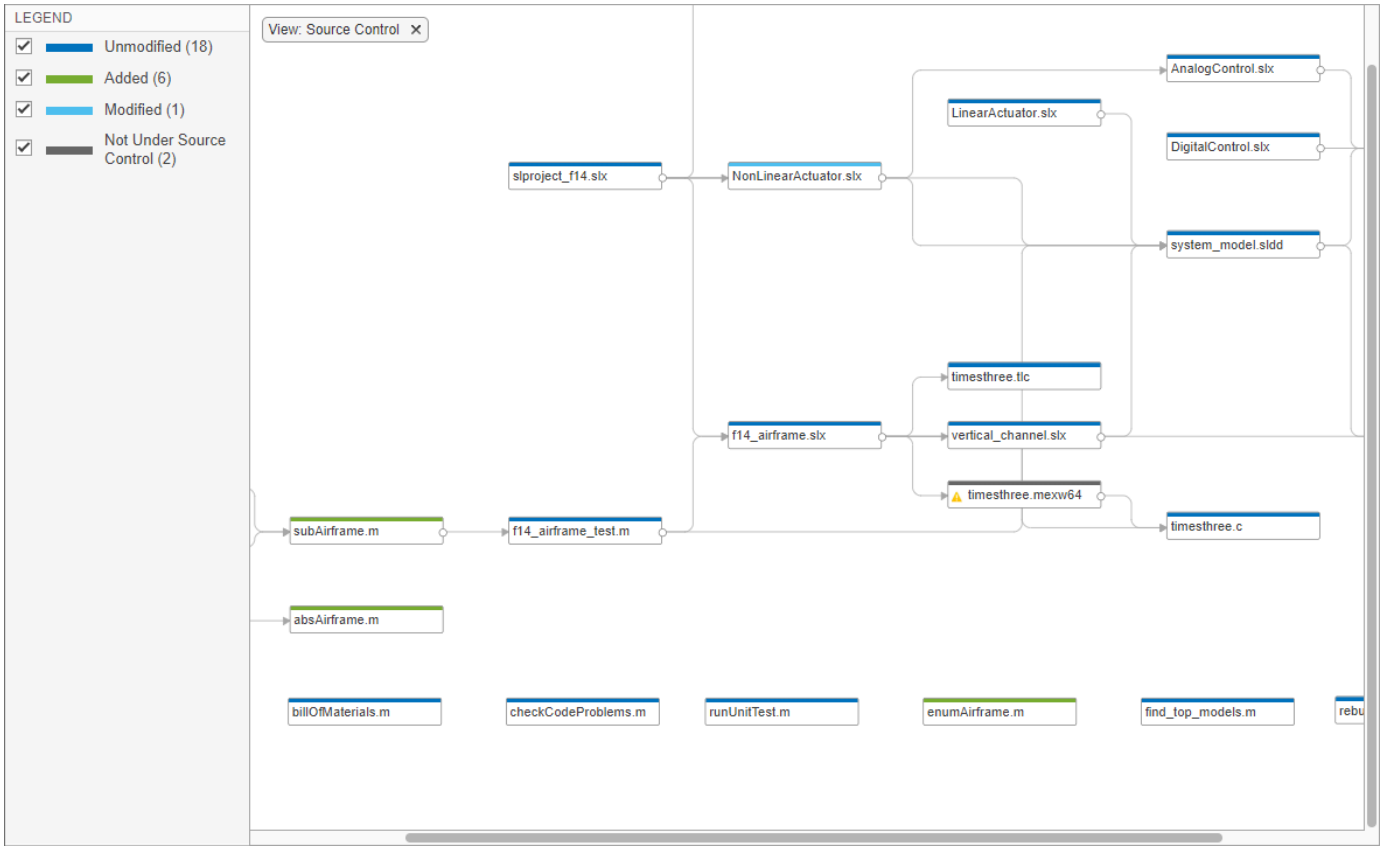
- The **Classification** view shows all files in the graph and colors them by file label (such as test, design, and artifact).

  Use the classification view to identify which tests you need to run to validate the changes in your design. For more information, see “Identify Tests to Run” on page 18-21.



- The **Source Control** view shows all files in the graph and colors them by source control status. This view is only enabled if your project is under source control.

  Use the source control view to find modified files in your project and to examine the impact of these changes on the rest of the project files. For more information, see “Investigate Impact of Modified Files” on page 18-21.



- **Restore to Default** clears all filters.

  This is equivalent to manually removing all of the filters. Filters appear at the top of the graph. For example, if you have the **Source Control** view selected, you can remove it by clicking View: Source Control ✕.

Apply and Clear Filters

In large projects, when investigating problems or dependencies, use the different filters to show only the files you want to investigate:

- To filter out a subgroup of files from the graph, such as files labeled *`test`* or modified files, use the check boxes in the **Legend** pane. To remove the legend filter, click the **Legend Filter** Legend Filter ✕.
- To color the files in the graph by type, class, label, or source control status, use the filtered **Views**. To remove the view filter, click **View: *`viewName`*** at the top of the graph. For example, if you have the **Source Control** view selected, you can remove it by clicking View: Source Control ✕.
- To show only the dependencies of a specific file, select the file and, in the **Impact Analysis** section, click **All Dependencies**. The graph shows the selected file and all its dependencies. To reset the graph to show all project dependencies, remove the filter at the top of the graph. For example, if you filtered by all dependencies of `NonLinearActuator.slx`, to remove the filter click All dependencies: NonLinearActuator.slx ✕.

- To clear all filters and restore the graph to show all analyzed dependencies in the project, click **Restore to Default**. Alternatively, manually remove all filters at the top of the graph.

See Also

Related Examples

- “Check Dependency Results and Resolve Problems” on page 18-23
- “Perform an Impact Analysis” on page 18-17

Perform an Impact Analysis

In this section...

About Impact Analysis

In a project, you can use impact analysis to find out the impact of changing particular files. Investigate dependencies visually and explore the structure of your project. Analyze selected or modified files to find their required files and the files they affect. Impact analysis can show you how a change affects other files before you make the change. For example, you can:

- Investigate the potential impact of a change in requirements by finding the design files linked to the requirements document.
- Investigate change set impact by finding upstream and downstream dependencies of modified files before committing the changes. Finding these dependencies can help you identify design and test files that need modification, and help you find the tests you need to run.

After performing dependency analysis, you can open or label the files, export the results as workspace variables, or reloadable files, or send files for custom task processing. Exporting the results enables further processing or archiving of impact analysis results.

Tip For an example showing how to perform file-level impact analysis to find and run the tests affected by modified files, see “Perform Impact Analysis with a Project”.

Run a Dependency Analysis

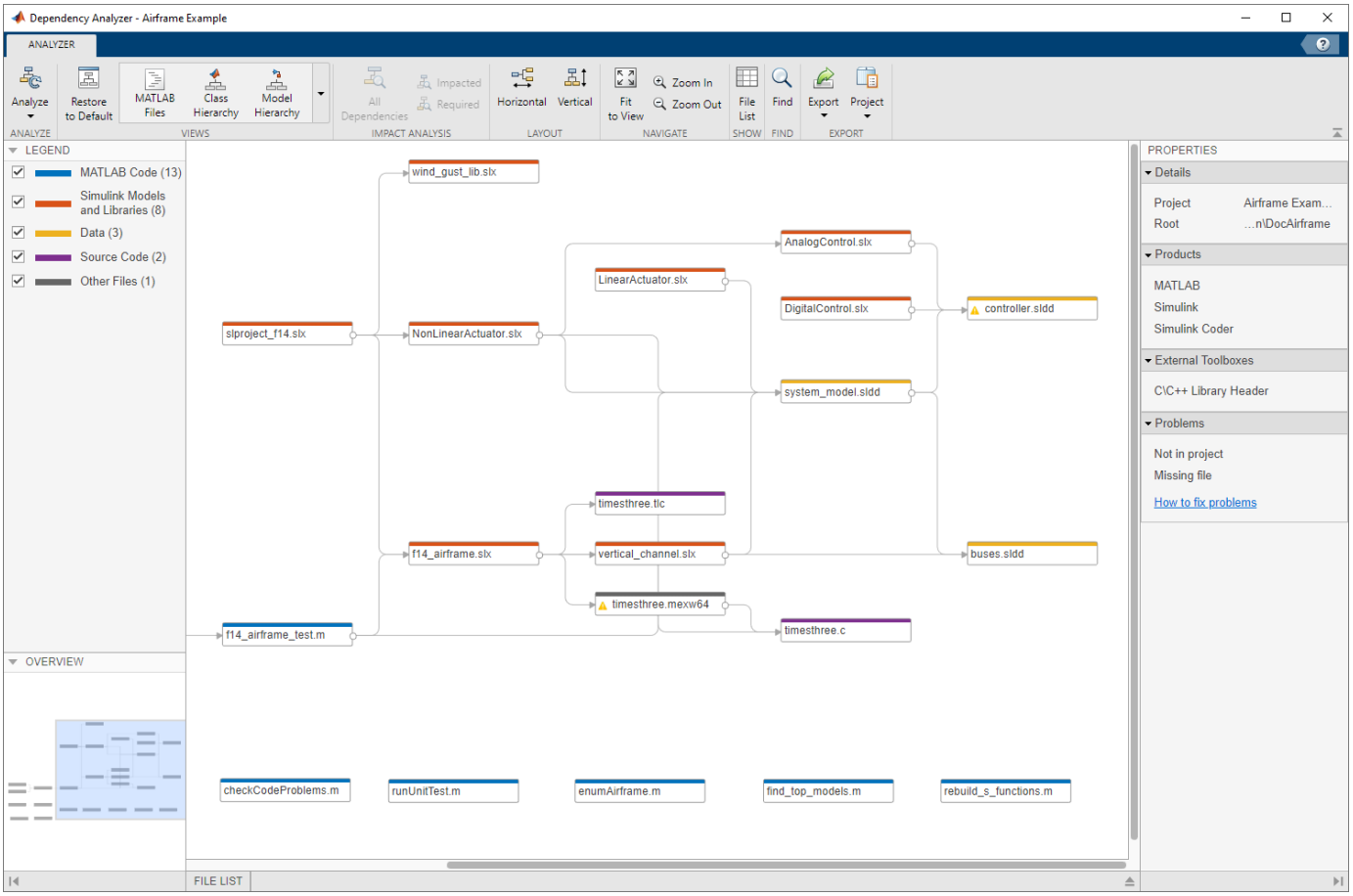
Before running a dependency analysis on a project, make sure that you have added all your files to the project. For more information, see “Add Files to the Project” on page 16-18.

To start analyzing your project, on the **Project** tab, in the **Tools** section, click **Dependency Analyzer**. Alternatively, in the project **Views** pane, select **Dependency Analyzer** and click **Analyze**.

To analyze the dependencies of specific files, in the dependency graph, select the files. In the **Impact Analysis** section, click **All Dependencies** or use the context menu and select **Find All Dependencies**.

To analyze the dependencies inside external toolboxes, select **Analyze > External Toolboxes**. For more details about available options, see “Analysis Scope” on page 18-4.

You can also check dependencies directly in Project. In the Project **Files** view, right-click the project files you want to analyze and select **Find Dependencies**.



The dependency graph shows:

- Your project structure and its file dependencies, including how files such as models, libraries, functions, data files, source files, and derived files relate to each other.
- Required products and toolboxes.
- Relationships between source and derived files (such as `.m` and `.p` files, `.slx` and `.slxp`, `.ssc` and `.sscp`, or `.c` and `.mex` files), and between C/C++ source and header files. You can see what code is generated by each model, and find what code needs to be regenerated if you modify a model.
- Warnings about problem files, such as missing files, files not in the project, files with unsaved changes, and out-of-date derived files.

You can examine project dependencies and problem files using the **File List**. In the toolstrip, click **File List**.

After you run the first dependency analysis of your project, subsequent analyses incrementally update the results. The Dependency Analyzer determines which files changed since the last analysis and updates the dependency data for those files. However, if you update external toolboxes or installed products and want to discover dependency changes in them, you must perform a complete analysis. To perform a complete analysis, in the Dependency Analyzer, click **Analyze** > **Reanalyze All**.

For next steps:

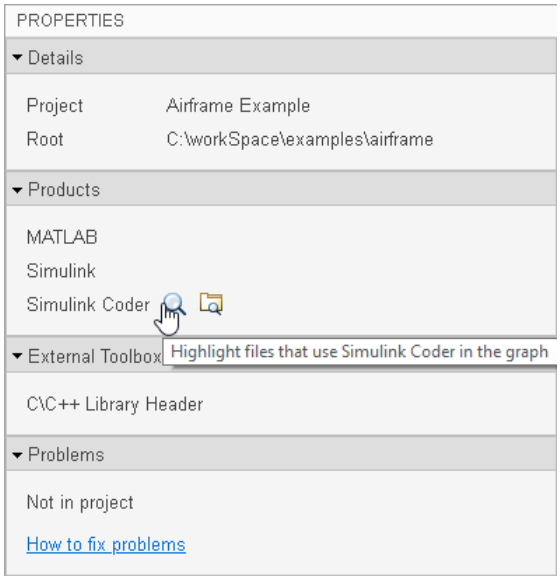
Tip To try a dependency analysis on example files, see “Perform Impact Analysis with a Project”.

Find Required Products and Toolboxes

After running a dependency analysis on a project, the graph shows the required toolboxes for the whole project or for selected files. You can see which products are required to use the project or find which file is introducing a product dependency.

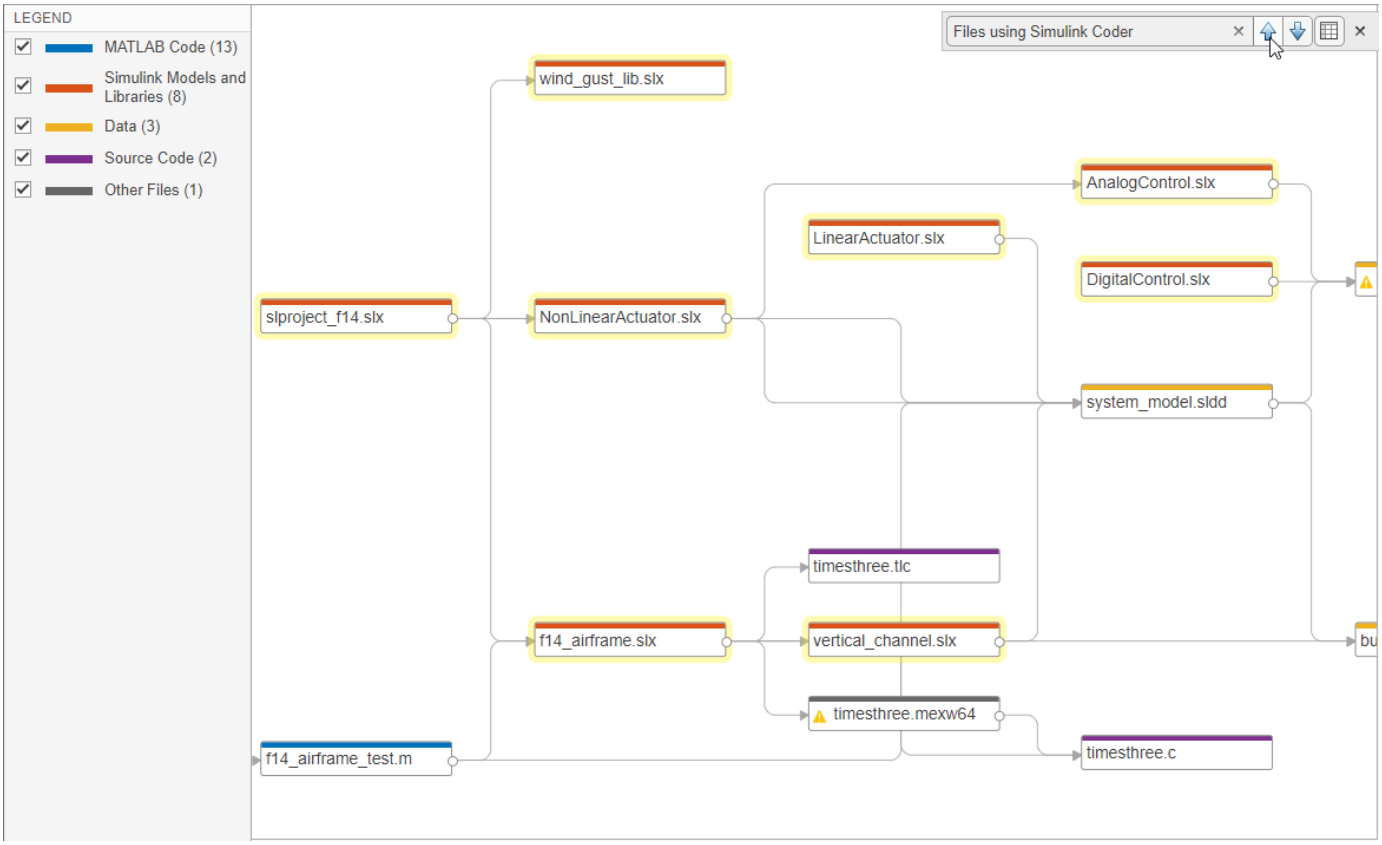
In the Dependency Analyzer, in the **Properties** pane, the **Product** section displays the required products for the whole project. To view products required by a specific file, select a file by clicking the graph.

To find which file is introducing a product dependency, point to the product name and click the magnifying glass icon . The graph highlights the files that use the selected product.



To go through these files, use the arrows in the search box (e.g., **Files using** "*productName*").

To undo the highlighting, close the search box.



To investigate further, you can list the files that use a product and examine where the dependency is introduced. In the **Products** section, in the **Properties** pane, point to a product and click the search folder icon  .

If a required product is missing, the products list labels it as missing. The product is also listed in the **Problems** section as ***`productName`* not installed**. To resolve a missing product, install the product and rerun the dependency analysis.

Find Dependencies of Selected Files

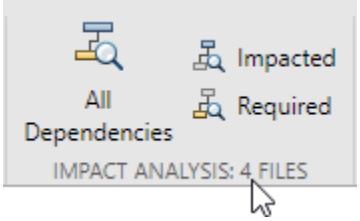
After a dependency analysis, to find out the impact of particular files, select files in the dependency graph use the context menu, or use controls in the legend, **View** and **Impact Analysis** sections of the Dependency Analyzer. You can simplify the graph by investigating dependencies of particular files.

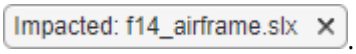
To investigate the dependencies of a file after running a dependency analysis, in the dependency graph, select a file.

- In the **Impact Analysis** section, click **All Dependencies**. The graph shows the selected file and all its dependencies.
- To show only files needed by the selected file to run properly, click **Required**.
- To show only files impacted by a potential change to the selected file, click **Impacted**.

Finding these dependencies can help you identify the impact of a change and identify the tests you need to run to validate your design before committing the changes.

To investigate the dependencies of multiple files, click files while holding the **Shift** key. The **Impact Analysis** section displays how many files are selected.



To reset the graph, click the filter at the top of the graph. For example, if you had filtered by files impacted by `f14_airframe.slx`, click 

.

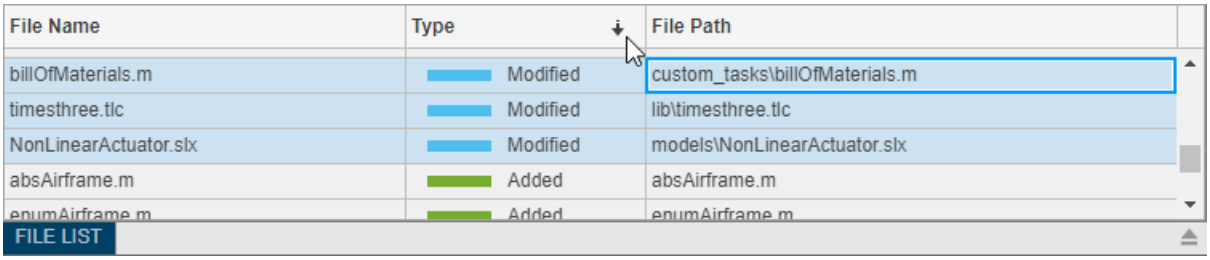
Investigate Impact of Modified Files

To examine the impact of the changes you made on the rest of the project files, perform an impact analysis on the modified files in your project.

1. In the **Views** section, select the **Source Control** view. The graph colors the files by their source control status. The modified files are in light blue.
2. Select all the modified files in the graph.

   Tip If you changed a large number of files, use the file list to select all files instead.

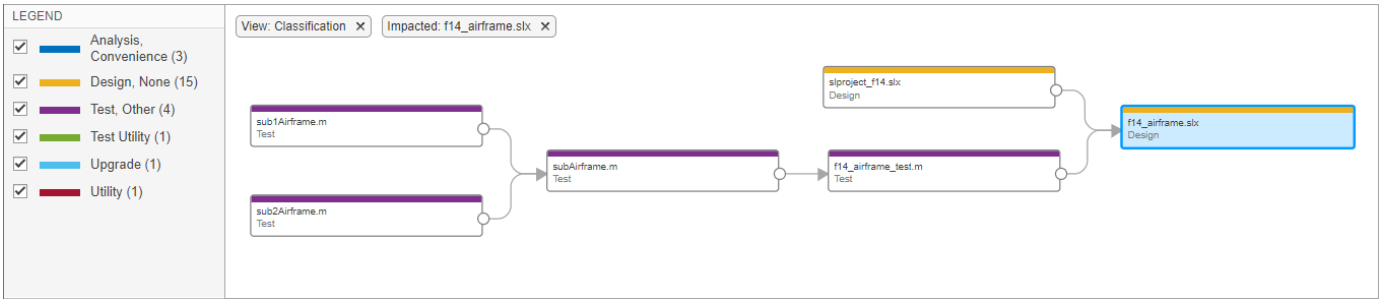
   In the Dependency Analyzer toolstrip, click **File List**. Point to **Type** and click the arrow to sort the list by the source control status. Select all the modified files.

3. In the **Impact Analysis** section, click **Impacted**. Alternatively, use the context menu and select **Find Impacted**.

Identify Tests to Run

To identify the tests you need to run to validate your design before committing the changes, use the **Classification** view when you perform an impact analysis on the file you changed.



1. In the **Views** section, select the **Classification** view. The graph colors the files by their project label.
2. Select the file you changed, for example `f14_airframe.slx`.
3. In the **Impact Analysis** section, click **Impacted**. Alternatively, use the context menu and select **Find Impacted**.

   The example graph shows four tests you need to run to qualify the change made to `f14_airframe.slx`.

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Check Dependency Results and Resolve Problems” on page 18-23
- “Export Dependency Analysis Results” on page 18-29
- “Find Requirements Documents in a Project” on page 18-28
- “Automate Project Tasks Using Scripts” on page 17-18
- “Perform Impact Analysis with a Project”

Check Dependency Results and Resolve Problems

If you have not yet run an analysis, on the **Project** tab, in the **Tools** section, click **Dependency Analyzer**.

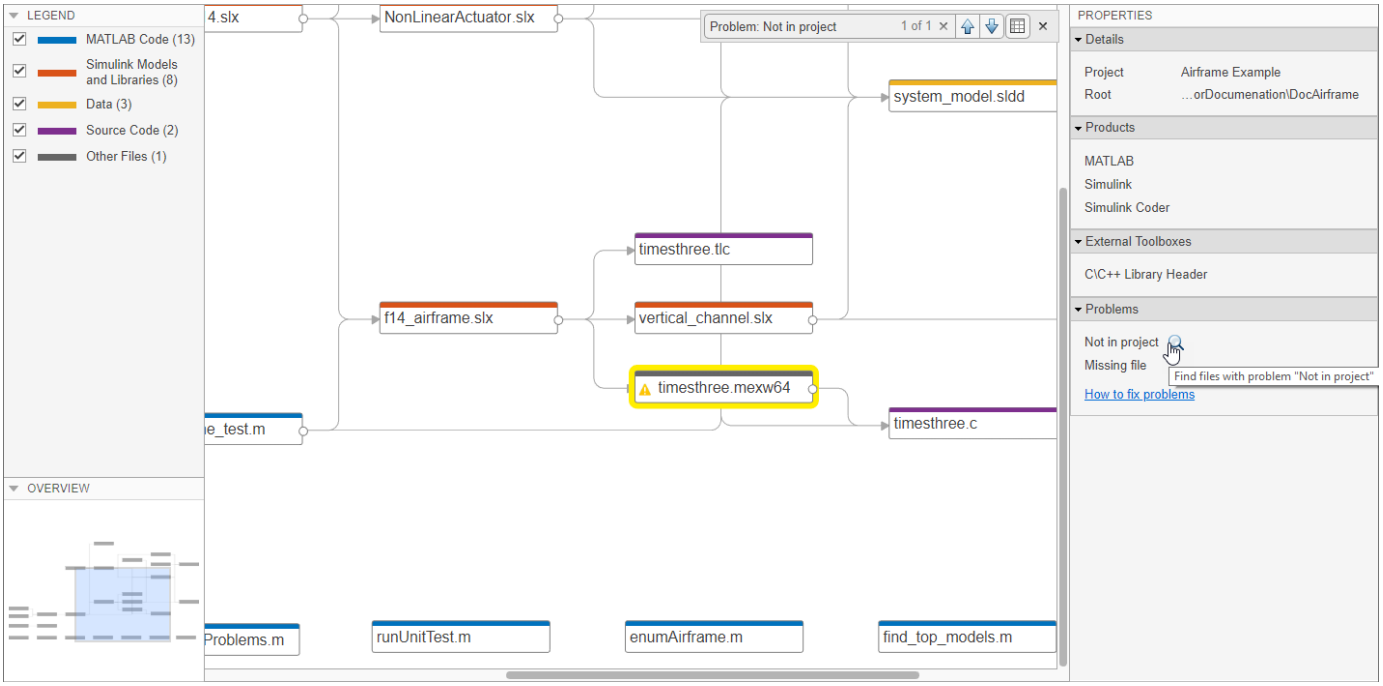
When you run a dependency analysis, the Dependency Analyzer identifies problems, such as missing files, files not in the project, unsaved changes, and out-of-date derived files. You can examine problem files using the dependency graph or the file list. When no file is selected, the **Properties** pane on the right shows the toolbox dependencies and a list of problems for the entire project.

Problem Message	Description	Fix
Not in project	The file is not in the project.	Right-click the problem file in the graph and select **Add to Project**. To remove a file from the problem list without adding it to the project, right-click the file and select **Hide Warnings**.
Missing file	The file is in the project but does not exist on disk.	Create the file or recover it using source control.
	The file or variable cannot be found.	If this status is acceptable, right-click the file and select **Hide Warnings**. Depending on the way you call an object method, the Dependency Analyzer might confuse a method with a function and report a missing dependency. See “Analysis Limitations” on page 18-4.
Outside project root	The file is outside the project root folder.	If this status is acceptable, right-click the file and select **Hide Warnings**. Otherwise, move it under the project root. If required files are outside your project root, you cannot add these files to your project. This dependency might not indicate a problem if the file is on your path and is a utility or resource that is not part of your project. Use dependency analysis to ensure that you understand the design dependencies.
Unsaved changes	The file has unsaved changes in the Simulink editor.	Save the file.

Problem Message	Description	Fix
Derived file out of date	The derived file is older than the source file it was derived from.	Regenerate the derived file. If it is a `.p` file, you can regenerate it automatically by running the project checks. In MATLAB, on the **Project** tab, select **Run Checks > Check Project** and follow the prompts to rebuild the files. If you rename a source file, the project detects the impact to the derived file and prompts you to update it.
Product not installed	The project has a dependency on a missing product.	Fix models by installing missing products. If you open a model that contains built-in blocks or library links from missing products, you see labels and links to help you fix the problem. • Blocks are labeled with missing products (for example, **SimEvents not installed**) • Tooltips include the name of the missing product • Messages provide links to open Add-On Explorer and install the missing products To find a link to open Add-On Explorer and install the product: • For built-in blocks, open the Diagnostic Viewer, and click the link in the warning. • For unresolved library links, double-click the block to view details and click the link. Product dependencies can occur in many other ways, for example in callbacks, so in this case you cannot easily see where the missing product is referenced. Fix models by installing missing products.

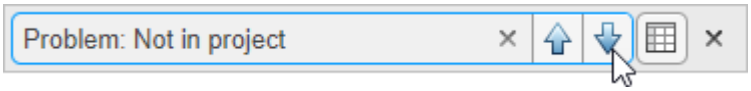
Investigate Problem Files in Dependency Graph

Use the graph to investigate problem files graphically.

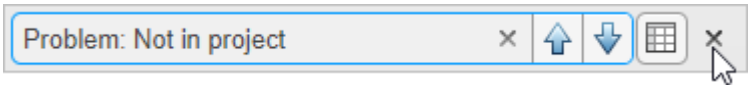


1 In the **Properties** pane, in the **Problems** section, point to a problem, such as *`Not in project`*, and click the magnifying glass icon . The graph highlights the files with this specific problem.

To go through these files, use the arrows in the search box (e.g., **Problem: *`Not in project`***).

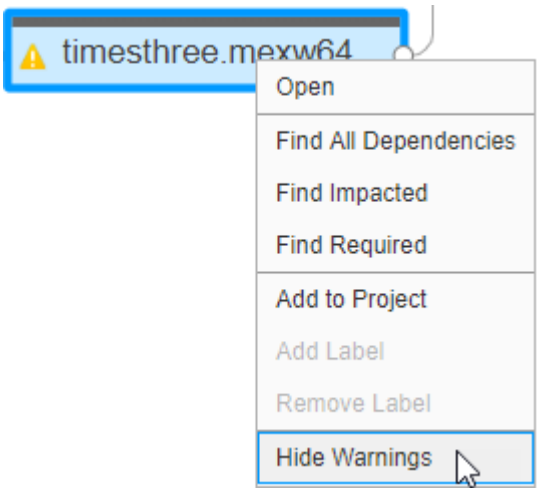


To undo the highlighting, close the search box.



2 To see more information about a specific problem file, select the file in the graph. In the **Properties** pane, in the **Problems** section, you can see details including the path, type, and the problems for this file.

Take actions to resolve the problem file. For example, if a file is *`Not in project`*, right-click the problem file in the graph and select **Add to Project**. To remove the file from the problem list without adding it to the project, right-click the file and select **Hide Warnings**.



3 Investigate the next problem listed in the **Problems** section. Repeat the steps until you resolve all problems.

   To update the graph and the **Problems** list, click **Analyze**.

Tip For large projects, viewing the results in a list can make navigation easier.

For more ways to work with the dependency graph, see “Perform an Impact Analysis” on page 18-17.

Investigate Problem Files in File List

For large projects, use the **File List** to investigate your project problem files.

1 In the Dependency Analyzer toolstrip, click **File List**.

2 In the **Properties** pane, in the **Problems** section, point to a problem, such as *`Not in project`*, and click the magnifying glass icon .

   The **File List** shows only files with the specific problem.

   To fix the *`Not in project`* problem, select all the files in the list. Use the context menu and select **Add to Project**. To remove a file from the problem list without adding it to the project, right-click the file and select **Hide Warnings**.

3 Investigate the next problem listed in the **Problems** section, for example *`Missing file`*. Repeat the steps until you resolve all problems.

   To update the graph and the **Problems** list, click **Analyze**.

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Perform an Impact Analysis” on page 18-17

- “Export Dependency Analysis Results” on page 18-29

Find Requirements Documents in a Project

In a project, a dependency analysis finds requirements documents linked using the Requirements Management Interface.

- You can view and navigate to and from the linked requirements documents.
- You can create or edit Requirements Management links only if you have Simulink Requirements.

1. On the **Project** tab, click **Dependency Analyzer**.

   Alternatively, in the project **Views** pane, select **Dependency Analyzer** and click the **Analyze** button.
2. The dependency graph displays the structure of all analyzed dependencies in the project. Project files that are not detectable dependencies of the analyzed files are not visible in the graph.
3. Use the dependency graph legend to locate the requirements documents in the graph. Arrows connect requirements documents to the files with the requirement links.
4. To find the specific block containing a requirement link, select the arrow connecting requirements documents to the files. In the **Properties** pane, in the **Impacted** column of the table, click the file to open it and highlight the block containing the dependency.
5. To open a requirements document, double-click the document in the graph.

See Also

Related Examples

- “Run a Dependency Analysis” on page 18-7
- “Check Dependency Results and Resolve Problems” on page 18-23
- “Perform an Impact Analysis” on page 18-17
- “Export Dependency Analysis Results” on page 18-29
- “View Linked Requirements in Models and Blocks” on page 17-45

Export Dependency Analysis Results

You can export dependency analysis results for your project in several formats. If you have not yet run an analysis, on the **Project** tab, in the **Tools** section, click **Dependency Analyzer**.

To export all the files displayed in the dependency graph, click the graph background to clear the selection on all files. In the Dependency Analyzer toolstrip, in the **Export** section, click **Export**. Select from the available options:

- **Export to Workspace** — Save file paths to a variable in the workspace.
- **Generate Dependency Report** — Save dependency analysis results in a printable report (HTML, Word, or PDF).
- **Package As Archive** — Export files in the graph as an archive.
- **Save As GraphML** — Save dependency analysis results as a GraphML file.

Tip You can open and compare different analysis results without having to repeat the analysis.

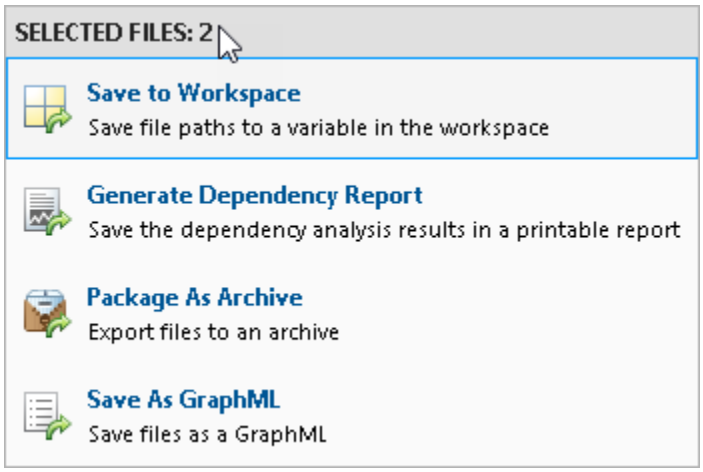
- To compare previously saved graphs, in MATLAB, in the **Current Folder**, right-click two GraphML files and select **Compare Selected Files/Folders**.
- To open saved dependency analysis results, use the `depview` function, then restore the view to default.

  1. In the Command Window, type: `depview("myDepResults.graphml");`
  2. In the Dependency Analyzer toolstrip, in the **Views** section, click **Restore to Default**.

To export a subset of files in the graph, select the files, then click **Export**.

- Use the **Legend** check boxes, the filtered **Views**, or the **Impact Analysis** tools to simplify the graph.
- To select multiple files, press **Shift** and select the files.
- To select all files in the filtered graph, press **Ctrl+A**.

The menu displays how many files are selected. The Dependency Analyzer exports only the selected files.



Note When you use **Package As Archive**, the Dependency Analyzer includes the selected files and all their dependencies in the archive.

Alternatively, you can work with the graph information programmatically. See “Automate Project Tasks Using Scripts” on page 17-18.

Send Files to Project Tools

You can send files to other Project tools using the **Project** menu. The Dependency Analyzer exports only the selected files in the current filtered view.

Select the desired files. In the Dependency Analyzer toolstrip, in the **Export** section, click **Project**. Select from the available options:

- **Show in Project** — Switch to the project **Files** view with the files selected.
- **Send to Custom Task** — Run a project custom task on the selected files.

See Also

Related Examples

- “What Is Dependency Analysis?” on page 18-2
- “Perform an Impact Analysis” on page 18-17

Project Source Control

About Source Control with Projects

You can use a project to work with source control. You can perform operations such as update, commit, merge changes, and view revision history directly from the project environment.

In MATLAB, projects have interfaces to:

- Git— See “Set Up Git Source Control” on page 19-16.
- Subversion (SVN) — See “Set Up SVN Source Control” on page 19-9.
- Software Development Kit (SDK) — You can use the SDK to integrate projects with third-party source control tools. See “Write a Source Control Integration with the SDK” on page 19-24.

Tip You can check for updated source control integration downloads on the projects Web page: `https://www.mathworks.com/discovery/simulink-projects.html`

To use source control in your project, use any of the following workflows:

- Add source control to a project. See “Add a Project to Source Control” on page 19-5.
- Retrieve files from an existing repository and create a new project. See “Clone Git Repository” on page 19-25 or “Check Out SVN Repository” on page 19-27.
- Create a new project in a folder already under source control and click **Detect**. See “Create a New Project From a Folder” on page 16-14.
- Make your project publicly available on GitHub. See “Share Project on GitHub” on page 17-33.

When your project is under source control, you can:

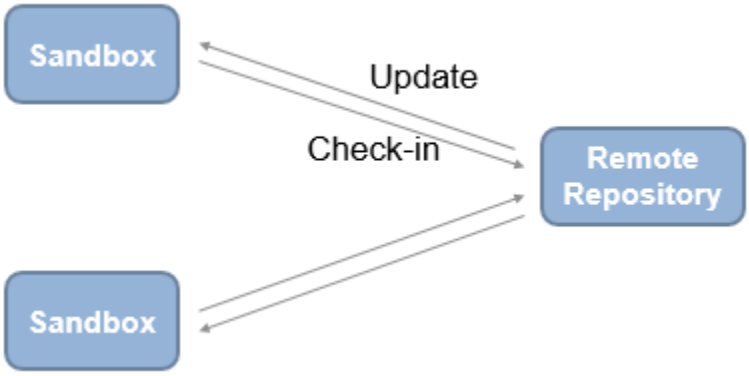
- “Clone Git Repository” on page 19-25 or “Check Out SVN Repository” on page 19-27
- “Compare Revisions” on page 19-39
- “Commit Modified Files to Source Control” on page 19-42

Caution Before using source control, you must register model files with your source control tools to avoid corrupting models. See “Register Model Files with Source Control Tools” on page 19-8.

To view an example project under source control, see “Explore Project Tools with the Airframe Project” on page 16-5.

Classic and Distributed Source Control

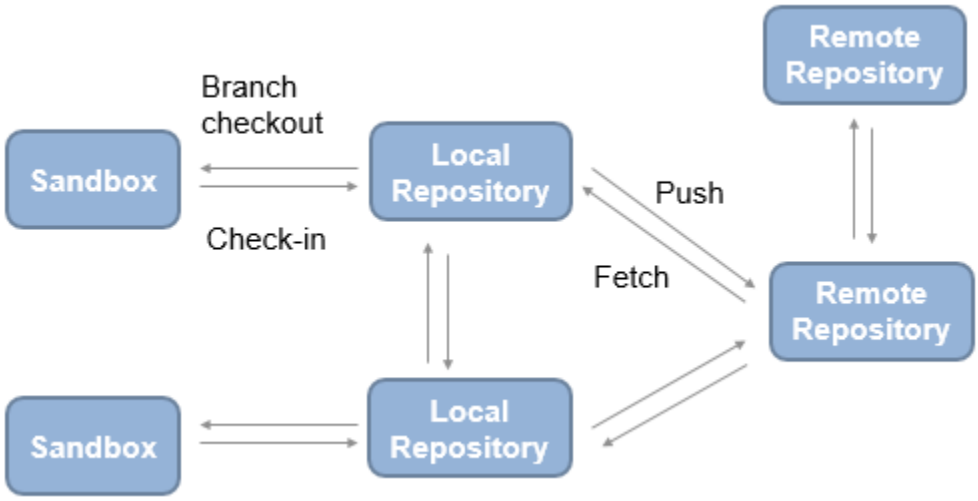
This diagram represents the classic source control workflow (for example, using SVN).



Benefits of classic source control:

- Locking and user permissions on a per-file basis (e.g., you can enforce locking of model files)
- Central server, reducing local storage needs
- Simple and easy to learn

This diagram represents the distributed source control workflow (for example, using Git).



Benefits of distributed source control:

- Offline working
- Local repository, which provides full history
- Branching
- Multiple remote repositories, enabling large-scale hierarchical access control

To choose classic or distributed source control, consider these tips.

Classic source control can be helpful if:

- You need file locks.

- You are new to source control.

Distributed source control can be helpful if:

- You need to work offline, commit regularly, and need access to the full repository history.
- You need to branch locally.

See Also

Related Examples

- “Set Up Git Source Control” on page 19-16
- “Set Up SVN Source Control” on page 19-9
- “Add a Project to Source Control” on page 19-5
- “Clone Git Repository” on page 19-25
- “Check Out SVN Repository” on page 19-27
- “Register Model Files with Source Control Tools” on page 19-8

Add a Project to Source Control

In this section...

Add a Project to Git Source Control

If you want to add version control to your project files without sharing with another user, it is quickest to create a local Git repository in your sandbox.

1. On the Project tab, in the Source Control section, click **Use Source Control**.
2. In the Source control Information dialog box, click **Add Project to Source Control**.
3. In the Add to Source Control dialog box, in the **Source control tool** list, select `Git` to use the Git source control tool provided by the project.
4. Click **Convert** to finish adding the project to source control.

   Git creates a local repository in your sandbox project root folder. The project runs integrity checks.
5. Click **Open Project** to return to your project.

   The Project node displays the source control name `Git` and the repository location `Local Repository: yoursandboxpath`.
6. Select the **Modified** files view and click **Commit** to commit the first version of your files to the new repository.

   In the dialog box, enter a comment if you want, and click **Submit**.

Tip If you want to use Git and share with other users:

- To clone an existing remote Git repository, see “Clone Git Repository” on page 19-25.
- To connect an existing project to a remote repository, on the Project tab, in the Source Control section, click **Remote** and specify a single remote repository for the origin branch.
- To make your project publicly available on GitHub, see “Share Project on GitHub” on page 17-33.

Add a Project to SVN Source Control

Caution Before you start, check that your sandbox folder is on a local hard disc. Using a network folder with SVN is slow and unreliable.

This procedure adds a project to the built-in SVN integration that comes with the project. If you want to use a different version of SVN, see “Set Up SVN Source Control” on page 19-9.

1. On the Project tab, in the Source Control section, click **Use Source Control**.

2. In the Source control Information dialog box, click **Add Project to Source Control**.
3. In the Add to Source Control dialog box, select `SVN` as the **Source control tool**.
4. Next to **Repository path**, click **Change**.
5. In the Specify SVN Repository URL dialog box, select an existing repository or create a new one.

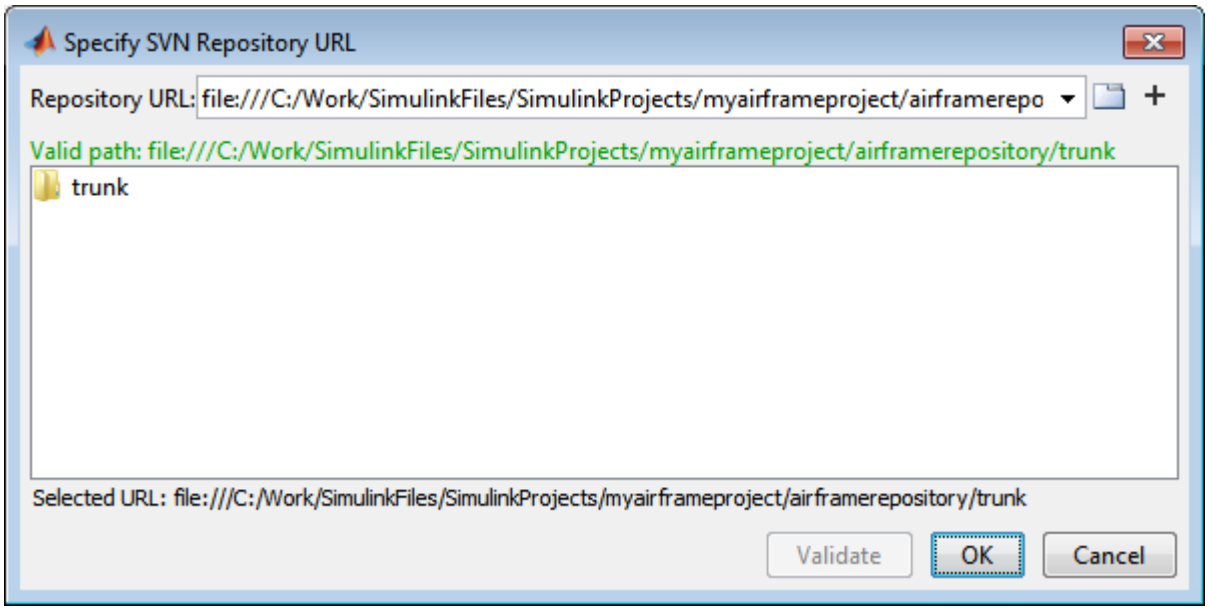
   - To specify an existing repository, click the button to browse for your repository, paste a URL into the box, or use the list to select a recent repository.
   - To create a new repository, click Create an SVN repository in a folder +. Using the file browser, create a folder where you want to create the new repository and click **Select Folder**. Do not place the new repository inside the existing project folder.

     The project creates a repository in your folder, and you return to the Specify SVN Repository URL dialog box. The URL of the new repository is in the **Repository** box, and the project automatically selects the `trunk` folder.

   Caution Specify `file://` URLs and create new repositories for single users only. For multiple users, see “Share a Subversion Repository” on page 19-14.

6. Click **Validate** to check the path to the selected repository.

   When the path is valid, you can browse the repository folders. For example, select the `trunk` folder, and verify the selected URL at the bottom of the dialog box, as shown.



7. Click **OK** to return to the Add to Source Control dialog box.

   If your repository has a file URL, a warning appears that file URLs are for single users. Click **OK** to continue.

8. Click **Convert** to finish adding the project to source control.

   The project runs integrity checks.

9 After the integrity checks run, click **Open Project** to return to your project.

The Project node displays details of the current source control tool and the repository location.

10 If you created a new repository, select the **Modified** files view and click **Commit** to commit the first version of your files to the new repository. In the dialog box, enter a comment if you want, and click **Submit**.

Caution Before using source control, you must register model files with your source control tools to avoid corrupting models. See “Register Model Files with Subversion” on page 19-11.

See Also

Related Examples

- “Set Up Git Source Control” on page 19-16
- “Set Up SVN Source Control” on page 19-9
- “Register Model Files with Source Control Tools” on page 19-8
- “Clone Git Repository” on page 19-25
- “Check Out SVN Repository” on page 19-27
- “Get SVN File Locks” on page 19-34
- “Work with Project Files” on page 17-7
- “View Modified Files” on page 19-36
- “Commit Modified Files to Source Control” on page 19-42

More About

- “About Source Control with Projects” on page 19-2

Register Model Files with Source Control Tools

If you use third-party source control tools, you must register your model file extensions (`.mdl` and `.slx`) as binary formats. If you do not, these third-party tools can corrupt your model files when you submit them, by changing end-of-line characters, expanding tokens, substituting keywords, or attempting to automerge. Corruption can occur whether you use the source control tools outside of Simulink or if you try submitting files from a project without first registering your file formats.

Also check that other file extensions are registered as binary to avoid corruption at check-in for files such as `.mat`, `.mlx`, `.mlapp`, `.mdlp`, `.slxp`, `.sldd`, `.p`, MEX-files, `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc.

For instructions with SVN, see “Register Model Files with Subversion” on page 19-11. You must register model files if you use SVN, including the `SVN` integration provided by the project.

For instructions with Git, see “Register Model Files with Git” on page 19-18.

Tip You can reduce your Git repository size by saving Simulink models without compression. Turning off compression results in larger SLX files on disk but reduces repository size.

To use this setting with new SLX files, create your models using a model template with SLX Compression set to none. For existing SLX files, set compression and then save the model. For more information, see “Set SLX Compression Level” on page 4-59.

See Also

Related Examples

- “Register Model Files with Subversion” on page 19-11
- “Register Model Files with Git” on page 19-18

Set Up SVN Source Control

In this section...

Set Up SVN Provided with Projects

Projects provide `SVN` for use with Subversion (SVN) sandboxes and repositories at version 1.9. You do not need to install SVN to use this integration because it includes an implementation of SVN.

Note This integration ignores any existing SVN installation.

The project `SVN` supports secure logins.

To use the version of SVN provided with the project, do one of the following:

- On the MATLAB Home tab, select **New > Project > From SVN**.
- Alternatively, in the start page, select **Project from SVN** to retrieve from source control, or when you add a project to source control, select `SVN` in the **Source control tool** list

. For instructions, see

- “Add a Project to Source Control” on page 19-5, or
- “Check Out SVN Repository” on page 19-27.

Caution Place your project sandbox folder on a local hard disc. Using a network folder with SVN is slow and unreliable. If you use a Windows network drive, SVN move operations can result in incorrect "not existing" status for files visible in file browsers.

When you create a new sandbox using the project `SVN`, the new sandbox uses the latest version of SVN provided by the project.

When your project is under source control, you can use these project features:

- “Check Out SVN Repository” on page 19-27
- “Compare Revisions” on page 19-39
- “Commit Modified Files to Source Control” on page 19-42

You can check out from a branch, but the project SVN does not support branch merging. Use an external tool such as TortoiseSVN to perform branch merging. You can use the project tools for

comparing and merging by configuring TortoiseSVN to generate a comparison report when you perform a diff on model files. See “Merge Simulink Models from the Comparison Report” on page 21-16.

Set Up Project SVN for SVN Version Already Installed

If you want to use projects with an earlier SVN version you already have installed, create a new project in a folder already under SVN source control. The project detects SVN.

For example:

1. Create the sandbox using TortoiseSVN from Windows Explorer.
2. In MATLAB, create a new project in that folder. The project detects the existing source control. If the sandbox is version 1.6, for example, it remains a version 1.6 sandbox.

Note Before using source control, you must register model files with the tools. See “Register Model Files with Subversion” on page 19-11.

Set Up Project SVN for SVN Version Not Yet Provided with Projects

If you need to use a later version of SVN than 1.9, you can use `Command-Line SVN Integration (compatibility mode)`, but you must also install a command-line SVN client.

Note Select `Command-Line SVN Integration (compatibility mode)` only if you need to use a later version of SVN than 1.9. Otherwise, use `SVN` instead, for more features, improved performance, and no need to install an additional command-line SVN client.

Command-line SVN integration communicates with any Subversion (SVN) client that supports the command-line interface.

1. Install an SVN client that supports the command-line interface.

   Note TortoiseSVN does not support the command-line interface unless you choose the option to install command-line tools. Alternatively, you can continue to use TortoiseSVN from Windows Explorer after installing another SVN client that supports the command-line interface. Ensure that the major version numbers match, for example, both clients are SVN 1.7.

   You can find Subversion clients on this Web page:

   `https://subversion.apache.org/packages.html`
2. In a project, select `Command-Line SVN Integration (compatibility mode)`.

With `Command-Line SVN Integration (compatibility mode)`, if you try to rename a file in a project and the folder name contains an @ character, an error appears because command-line SVN treats all characters after the @ symbol as a peg revision value.

Tip You can check for updated source control integration downloads on the projects Web page: `https://www.mathworks.com/discovery/simulink-projects.html`

Register Model Files with Subversion

You must register model files if you use SVN, including the `SVN` integration provided by projects.

If you do not register your model file extension as binary, SVN might add annotations to conflicted Simulink files and attempt automerge. This corrupts model files so you cannot load the models in Simulink.

To avoid this problem when using SVN, register file extensions.

1. Locate your SVN `config` file. Look for the file in these locations:
   - `C:\Users\`*`myusername`*`\AppData\Roaming\Subversion\config` or `C:\Documents and Settings\`*`myusername`*`\Application Data\Subversion\config` on Windows
   - In `~/.subversion` on Linux or Mac OS X
2. If you do not find a `config` file, create a new one. See “Create SVN Config File” on page 19-11.
3. If you find an existing `config` file, you have previously installed SVN. Edit the `config` file. See “Update Existing SVN Config File” on page 19-12.

Create SVN Config File

1. If you do not find an SVN `config` file, create a text file containing these lines:

```
[miscellany]
enable-auto-props = yes
[auto-props]
*.mlx = svn:mime-type=application/octet-stream
*.mat = svn:mime-type=application/octet-stream
*.fig = svn:mime-type=application/octet-stream
*.mdl = svn:mime-type=application/octet-stream
*.slx = svn:mime-type= application/octet-stream
*.mlapp = svn:mime-type= application/octet-stream
*.p = svn:mime-type=application/octet-stream
*.mdlp = svn:mime-type=application/octet-stream
*.slxp = svn:mime-type=application/octet-stream
*.sldd = svn:mime-type=application/octet-stream
*.slxc = svn:mime-type=application/octet-stream
*.mlproj = svn:mime-type=application/octet-stream
*.mldatx = svn:mime-type=application/octet-stream
*.slreqx = svn:mime-type=application/octet-stream
*.sfx = svn:mime-type=application/octet-stream
*.sltx = svn:mime-type=application/octet-stream
```

2. Check for other file types you use in your projects that you also need to register as binary to avoid corruption at check-in. Check for files such as MEX-files (`.mexa64`, `.mexmaci64`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the attributes file for each file type you need. Examples:

```
*.mexa64 = svn:mime-type=application/octet-stream
*.mexw64 = svn:mime-type=application/octet-stream
*.mexmaci64 = svn:mime-type=application/octet-stream
*.xlsx = svn:mime-type=application/octet-stream
*.docx = svn:mime-type=application/octet-stream
*.pdf = svn:mime-type=application/octet-stream
*.jpg = svn:mime-type=application/octet-stream
*.png = svn:mime-type=application/octet-stream
```

3. Name the file `config` and save it in the appropriate location:

   - `C:\Users\`*`myusername`*`\AppData\Roaming\Subversion\config` or `C:\Documents and Settings\`*`myusername`*`\Application Data\Subversion\config` on Windows
   - `~/.subversion` on Linux or Mac OS X

After you create the SVN config file, SVN treats new model files as binary.

If you already have models in repositories, see “Register Models Already in Repositories” on page 19-13.

Update Existing SVN Config File

If you find an existing `config` file, you have previously installed SVN. Edit the `config` file to register files as binary.

1. Edit the `config` file in a text editor.
2. Locate the `[miscellany]` section, and verify the following line enables `auto-props` with `yes`:

   ```
   enable-auto-props = yes
   ```

   Ensure that this line is not commented (that is, that it does not start with a `#`). Config files can contain example lines that are commented out. If there is a `#` character at the beginning of the line, delete it.
3. Locate the `[auto-props]` section. Ensure that `[auto-props]` is not commented. If there is a `#` character at the beginning, delete it.
4. Add the following lines at the end of the `[auto-props]` section:

   ```
   *.mlx = svn:mime-type=application/octet-stream
   *.mat = svn:mime-type=application/octet-stream
   *.fig = svn:mime-type=application/octet-stream
   *.mdl = svn:mime-type=application/octet-stream
   *.slx = svn:mime-type= application/octet-stream
   *.mlapp = svn:mime-type= application/octet-stream
   *.p = svn:mime-type=application/octet-stream
   *.mdlp = svn:mime-type=application/octet-stream
   *.slxp = svn:mime-type=application/octet-stream
   *.sldd = svn:mime-type=application/octet-stream
   *.slxc = svn:mime-type=application/octet-stream
   *.mlproj = svn:mime-type=application/octet-stream
   *.mldatx = svn:mime-type=application/octet-stream
   *.slreqx = svn:mime-type=application/octet-stream
   *.sfx = svn:mime-type=application/octet-stream
   *.sltx = svn:mime-type=application/octet-stream
   ```

   These lines prevent SVN from adding annotations to MATLAB and Simulink files on conflict and from automerging.
5. Check for other file types you use in your projects that you also need to register as binary to avoid corruption at check-in. Check for files such as MEX-files (`.mexa64`, `.mexmaci64`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the `config` file for each file type you need.

   Examples:

   ```
   *.mexa64 = svn:mime-type=application/octet-stream
   *.mexw64 = svn:mime-type=application/octet-stream
   ```

```
*.mexmaci64 = svn:mime-type=application/octet-stream
*.xlsx = svn:mime-type=application/octet-stream
*.docx = svn:mime-type=application/octet-stream
*.pdf = svn:mime-type=application/octet-stream
*.jpg = svn:mime-type=application/octet-stream
*.png = svn:mime-type=application/octet-stream
```

6 Save the `config` file.

After you create or update the SVN config file, SVN treats new model files as binary.

If you already have models in repositories, register them as described next.

Register Models Already in Repositories

Caution Changing your SVN config file does not affect model files already committed to an SVN repository. If a model is not registered as binary, use `svn propset` to manually register models as binary.

To manually register a file in a repository as binary, use the following command with command-line SVN:

```
svn propset svn:mime-type application/octet-stream modelfilename
```

If you need to install a command-line SVN client, see “Set Up Project SVN for SVN Version Not Yet Provided with Projects” on page 19-10.

Enforce SVN Locking Model Files Before Editing

To ensure users remember to get a lock on model files before editing, you can configure SVN to make specified file extensions read only. To locate your SVN config file, see “Register Model Files with Subversion” on page 19-11.

After this setup, SVN sets model files to read only when you open the project, so you need to select **Source Control > Get File Lock** before you can edit them. Doing so helps prevent editing of models without getting the file lock. When the file has a lock, other users know the file is being edited, and you can avoid merge issues.

1 To make SLX files read only, add a property to your SVN config file. Find this line in the `[auto-props]` section that registers SLX files as binary:

```
*.slx = svn:mime-type= application/octet-stream
```

2 Add the `needs-lock` property to the end of the existing `slx` line, separated by a semicolon, so the line looks like this:

```
*.slx = svn:mime-type=application/octet-stream;svn:needs-lock=yes
```

You can combine properties in any order, but multiple entries (e.g., for `slx`) must be on a single line separated by semicolons.

3 Recreate the sandbox for the config to take effect.

4 You need to select **Get File Lock** before you can edit model files. See “Get SVN File Locks” on page 19-34.

If you need to resolve merge issues, see “Resolve Conflicts” on page 19-54.

Share a Subversion Repository

You can specify a repository location using the `file://` protocol. However, Subversion documentation strongly recommends only single users access a repository directly via `file://` URLs. See the Web page:

`http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.choosing.recommendations`

Caution Do not allow multiple users to access a repository directly via `file://` URLs or you risk corrupting the repository. Use `file://` URLs only for single-user repositories.

Be aware of this caution with these workflows:

- If you specify a repository with a `file://` URL, or
- If you use a project to create a repository, this uses the `file://` protocol. Creating new repositories is provided for local single-user access only, for testing and debugging.

Also, accessing a repository via `file://` URLs is slower than using a server.

When you want to share a repository, you need to set up a server. You can use `svnserve` or the Apache SVN module. See the Web page references:

`http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.svnserve`
`http://svnbook.red-bean.com/en/1.7/svn-book.html#svn.serverconfig.httpd`

Standard Repository Structure

Create your repository with the standard `tags`, `trunk`, and `branches` folders, and check out files from `trunk`. The Subversion project recommends this structure. See the Web page:

`https://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html`

If you use a project to create an SVN repository, it creates the standard repository structure. To enable tagging, the repository must have `trunk/` and `tags/` folders.

After you create a repository with this structure, to add tags to all your project files, on the Project tab, in the Source Control section, click **Tag**. See “Tag and Retrieve Versions of Project Files” on page 19-29.

Manage SVN Externals

To get files into your project from another repository or from a different part of the same repository, use SVN externals.

1. In a project, right-click a project folder and select **Source Control > Manage Externals**.
2. In the Manage Externals dialog box, click **Add entry**. You can browse to and validate a repository location, specify the relative reference format, specify the subfolder, choose the revision, e.g., the `HEAD` node, etc.
3. After specifying the externals, click **OK**. The project displays the externals definition in the Manage Externals dialog box.

   Alternatively, enter or paste an `svn:external` definition in the Manage Externals dialog box. The project applies an SVN version 1.6 compliant externals definition.

4. Click **Set** to validate and apply your changes.
5. To retrieve the external files, click **Update** to update the sandbox.

If two users modify the `svn:external` for a folder, you can get a conflict. To resolve the conflict, in the All Files View, locate the `.prej` file and examine the conflict details. Open the Manage Externals dialog box and specify the desired `svn:external`, mark the folder conflict resolved, and then commit the changes.

See Also

Related Examples

- “Check Out SVN Repository” on page 19-27
- “Get SVN File Locks” on page 19-34

More About

- “About Source Control with Projects” on page 19-2

Set Up Git Source Control

In this section...

To use the version of Git provided with projects, when you add a project to source control or retrieve from source control, select `Git` in the **Source control tool** list.

- If you add an existing project to Git source control, you create a local Git repository in that sandbox. You can specify a remote repository later. See “Add a Project to Source Control” on page 19-5.
- If you want to clone a remote Git repository to create a project, on the MATLAB Home tab, select **New > Project > From Git**. After you specify a remote repository to clone, a local repository is created. You can also pull, fetch, and push changes to and from the remote repository. See “Clone Git Repository” on page 19-25.

  To use a Git server for your remote repository, you can use a Git server hosting solution or set up your own Apache Git server. If you cannot set up a server and must use a remote repository via the file system using the `file:///` protocol, make sure it is a bare repository with no working copy checked out.
- To make your project publicly available on GitHub, see “Share Project on GitHub” on page 17-33. Sharing adds Git source control to the open project and the project’s remote repository is GitHub.

Configure MATLAB on Windows

Note Starting in R2020b, you do not need to install command-line Git to fully use Git with MATLAB. You can now merge branches using the built-in Git integration.

To setup Git for releases before R2020b, see https://www.mathworks.com/help/releases/R2020a/simulink/ug/set-up-git-source-control.html.

Several operations, such as committing, merging, and receiving pushed commits, use Git Hooks. To use Git Hooks on Windows with MATLAB, install Cygwin and add it to the MATLAB library path:

1. Download the installer from https://www.cygwin.com/. Run the installer.
2. In the MATLAB Command Window, type `edit(fullfile(matlabroot,"toolbox","local","librarypath.txt"))`.

   Add the Cygwin bin folder location to the end of `librarypath.txt`, for example, `C:\cygwin64\bin`.

   If you do not have permission to edit the `librarypath.txt` file, create a copy and save it to your MATLAB start folder.
3. Restart MATLAB for the changes to take effect.

You can clone a remote repository like GitHub and GitLab using HTTPS or SSH. To prevent frequent login prompts when you interact with your remote repository using HTTPS, add a new public key and

clone the repository using SSH instead. To avoid problems connecting using SSH, set the `HOME` environment variable and use it to store your SSH keys. For more information, see “Use SSH Authentication with MATLAB” on page 19-17.

For new projects under Git source control, MATLAB automatically registers your binary files to prevent corruption when merging. For existing projects, register the binary files before using Git to merge branches. For more information, see “Register Model Files with Git” on page 19-18.

If you are working with long path files, run this command in MATLAB:

```
!git config --global core.longpaths true
```

Use SSH Authentication with MATLAB

To prevent frequent login prompts when you interact with your remote repository using HTTPS, add a new public key and clone the repository using SSH instead.

MATLAB Git integration uses the user `HOME` environment variable to locate the `.ssh` folder containing SSH keys. If the `HOME` environment variable is not set or the SSH keys are not stored properly, you will encounter problems using SSH to connect to remote repositories like GitHub and GitLab.

To use SSH authentication inside MATLAB:

1. Use `ssh-keygen` to generate valid SSH keys. In the Command Prompt, enter:

   ```
   ssh-keygen
   ```

   ```
   Generating public/private rsa key pair.
   Enter file in which to save the key (C:\Users\username/.ssh/id_rsa):
   Created directory 'C:\Users\username/.ssh'.
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in C:\Users\username/.ssh/id_rsa.
   Your public key has been saved in C:\Users\username/.ssh/id_rsa.pub.
   The key fingerprint is:
   SHA256:/Nc9/tnZ7Dmh77+iJMxmPVrlPqaFd6J1j1YRXEk3Tgs company\username@us-username
   ```

   `ssh-keygen` confirms where to save the key (for example, `.ssh/id_rsa`) and asks for a passphrase. If you do not want to type a password when you use the key, leave the passphrase empty. If you already have keys in the specified folder, `ssh-keygen` asks if you want to override them.

   Note It is not possible to generate SSH keys directly in MATLAB. Generate SSH keys using the `ssh-keygen` provided with a command-line Git install.

2. Place your keys in the `HOME/.ssh` folder. To verify which `HOME` directory the MATLAB Git integration is working with, in the MATLAB Command Window, enter:

   ```
   getenv('HOME')
   ```

3. If `getenv('HOME')` returns nothing, you need to set your `HOME` environment variable.

   To set the `HOME` environment variable in Windows:

   - In the Start Search box, search for and select "advanced system settings".

- On the **Advanced** tab, click **Environment Variables**.
- In the User Variables section, click **New**. Create the `HOME` environment variable and specify its value.

The `HOME` environment variable is always set on Linux and Mac.

4 Configure your GitHub or GitLab account to use the SSH keys:

- Copy the contents of `.pub` file in the `.ssh` folder.
- Paste the contents in the Add SSH key field in the SSH keys section of your account settings.

Register Model Files with Git

You can prevent Git from corrupting your Simulink models by registering binary files in your `.gitattributes` file.

- For new projects and projects that switched from another source control system, MATLAB automatically creates a `.gitattributes` file and populates it with a list of binary files to register. This specifies that Git should not make automatic line feed, diff, and merge attempts for registered files.
- For existing projects, create a `.gitattributes` file and populate it with the list of binary files to register.

  1 In the Command Window, type:

  ```
  edit .gitattributes
  ```

  2 Add a line to the attributes file for each file type you need. For example, `*.mlapp binary`.

  Tip You can copy a `.gitattributes` file that contains the list of common binary files to register.

  ```
  copyfile(fullfile(matlabroot,'toolbox','shared','cmlink','git','auxiliary_files','mwgitatt
  ```

  3 Restart MATLAB so you can start using the Git client.

Tip You can reduce your Git repository size by saving Simulink models without compression. Turning off compression results in larger SLX files on disk but reduces repository size.

To use this setting with new SLX files, create your models using a model template with SLX Compression set to none. For existing SLX files, set compression and then save the model. For more information, see “Set SLX Compression Level” on page 4-59.

See Also

Related Examples

- “Clone Git Repository” on page 19-25
- “Branch and Merge Files with Git” on page 19-50

Add Git Submodules

To reuse code from another repository, you can specify Git submodules to include in your project.

To clone an external Git repository as a submodule:

1. On the Project tab, in the Source Control section, click **Submodules**.
2. In the Submodules dialog box, click the **+** button.
3. In the Add Submodule dialog box, in the **Remote** box, specify a repository location. Optionally, click **Validate**.
4. In the **Path** box, specify a location for the submodule in your project and click **OK**. The Submodules dialog box displays the status and details of the submodule.
5. Check the status message, and click **Close** to return to your project.

Update Submodules

After using **Pull** on the top-level project, check submodules are up to date by clicking Submodules and then click **Update**. If any submodule definition have changed, then the update ensures that the submodule folder contains the correct files. Update applies to all child submodules in the submodule hierarchy.

Use Fetch and Merge with Submodules

When you want to manage a submodule, open the Submodules dialog box.

1. To get the latest version of a submodule, in the Submodules dialog box, click **Fetch**.
2. After fetching, you must merge. Check the **Status** message in the Submodules dialog box for information about your current branch relative to the remote tracking branch in the repository. When you see the message `Behind`, you need to merge in changes from the repository to your local branch.
3. Click **Branches** and merge in the origin changes to your local branch using the Branches dialog box. See “Pull, Fetch, and Merge” on page 19-47.

Use Push to Send Changes to the Submodule Repository

If you make changes in your submodule and want to send changes back to the repository:

1. Perform a local commit in the parent project.
2. Open the Submodules dialog box and click **Push**.

If you want other project users to obtain your changes in the submodule when they clone the parent project, make sure the index and head match.

1. In the Submodules dialog box, check the index and head values. The index points to the head commit at the time you first cloned the submodule, or when you last committed the parent project repository. If the index and head do not match, you must update the index.
2. To update the index, commit your changes in the parent project, and then click **Push** in the Submodules dialog box. This action makes the index and head the same.

See Also

Related Examples

- “Branch and Merge Files with Git” on page 19-50

Create New GitHub Repository

Creating a GitHub repository adds Git source control to your new or existing project. The GitHub repository you create becomes the project remote repository. To create a GitHub repository, you must have a GitHub account.

To create a blank project and a GitHub remote repository:

1. On the **Home** tab, click **New > Project > From Git**.
2. Select **New > GitHub Repository**. In the GitHub dialog box, enter your **User name** and **Password**. Fill the **Repository name** and **Description** fields and click **Create**.

   MATLAB creates a new public GitHub repository and populates the **Repository path** field with information in the https://github.com/myusername/mynewrepository format.
3. In the **Sandbox** field, specify the location for your sandbox. The selected folder must be empty. Click **Retrieve** to create the sandbox.

   To confirm the project name and creation, click **OK**.

After creating the GitHub repository and sandbox, add your files to the sandbox. Commit the first version of your files to your local repository, then push all modifications to your remote GitHub repository.

Tip If you want to create a remote GitHub repository for an existing project, share your project to GitHub instead.

With your project loaded, on the **Project** tab, select **Share > GitHub**. For detailed instructions, see “Share Project on GitHub” on page 17-33.

Disable Source Control

Disabling source control is useful when you are preparing a project to create a template from it, and you want to avoid accidentally committing unwanted changes.

1. On the Project tab, in the Source Control section, click the **Details** button for your source control. For example, **SVN Details** or **Git Details**.
2. Change the selection from the current source control to `No source control integration`.
3. Click **Reload**.

Note Source control tools create files in the project folders (for example, SVN creates an `.svn` folder), so you can put the project back under the same source control only by selecting your previous source control from the list.

See Also

Related Examples

- “Change Source Control” on page 19-23
- “Create a Template from a Project Under Version Control” on page 16-33
- “Add a Project to Source Control” on page 19-5

Change Source Control

Changing source control is useful when you want to create a new local repository for testing and debugging.

1. Prepare your project by checking for any updates from the existing source control tool repository and committing any local changes.
2. Save your project as an archive without any source control information.

   On the **Project** tab, click **Share** > **Archive**. Click **Save As** and specify a file path and name in the **File name** field.

   By default, the archive is a `.mlproj` file. You can choose to archive your project as a `.zip` file. Click **Save** to create the project archive.
3. On the **Project** tab, click **New**, and then in the start page, click **Archive** to create a new project from the archived project.
4. On the Project tab, in the Source Control section, click **Use Source Control**.
5. Click **Add Project to Source Control** and then select a new source control. For details, see “Add a Project to Source Control” on page 19-5.

Tip To avoid accidentally committing changes to the previous source control, delete the original sandbox.

See Also

Related Examples

- “Disable Source Control” on page 19-22
- “Add a Project to Source Control” on page 19-5

Write a Source Control Integration with the SDK

Tip You can check for updated source control integration downloads on the projects Web page: `https://www.mathworks.com/discovery/simulink-projects.html`

The file exchange provides a Software Development Kit (SDK) that you can use to integrate projects with third-party source control tools. See https://www.mathworks.com/matlabcentral/fileexchange/61483-source-control-integration-software-development-kit.

The SDK provides instructions for writing an integration to a source control tool that has a published API you can call from Java®.

You must create a `.jar` file that implements a collection of Java interfaces and a Java Manifest file, that defines a set of required properties.

The SDK provides example source code, Javadoc, and files for validating, building, and testing your source control integration. Build and test your own interfaces using the example as a guide. Then you can use your source control integration with projects. Download the SDK and follow the instructions.

After you write a source control integration, see “Add a Project to Source Control” on page 19-5.

See Also

More About

- “About Source Control with Projects” on page 19-2

Clone Git Repository

Create a new local copy of a project by retrieving files from Git source control.

1. On the **Home** tab, click **New > Project > From Git**. The New Project From Source Control dialog box opens.

   Alternatively, on the Simulink start page, click the **Project from Git** template.
2. Enter your HTTPS repository path into the **Repository path** field.
3. In the **Sandbox** field, select the working folder where you want to put the retrieved files for your new project.
4. Click **Retrieve**.

   If an authentication dialog box for your repository appears, enter the login information for your Git repository account -- for instance, your GitHub user name and password.

   If your repository already contains a project, then the project is ready when the tool finishes retrieving files to your selected sandbox folder.

   If your sandbox does not yet contain a project, then a dialog box asks whether you want to create a project in the folder. To create a project, specify a project name and click **OK**. The Welcome screen appears to help you set up your new project. For more information about setting up a project, see “Create a New Project From a Folder” on page 16-14.

You can now add, delete, and modify your project files. For details on how to commit and push the modified project files, see “Commit Modified Files to Source Control” on page 19-42.

Tip Alternatively, to prevent frequent login prompts when you interact with your remote repository, you can clone a remote repository using SSH instead of HTTPS. To avoid problems connecting using SSH, set the `HOME` environment variable and use it to store your SSH keys. For more information, see “Use SSH Authentication with MATLAB” on page 19-17.

Troubleshooting

If you encounter errors like `OutOfMemoryError: Java heap space` when cloning big Git repositories, then edit your MATLAB preferences to increase the heap size.

1. On the **Home** tab, in the **Environment** section, click **Preferences**.
2. Select **MATLAB > General > Java Heap Memory**.
3. Move the slider to increase the heap size, and then click **OK**.
4. Restart MATLAB.

See Also

Related Examples

- “Set Up Git Source Control” on page 19-16
- “Work with Project Files” on page 17-7

- “Refresh Status of Project Files” on page 19-30
- “Check for Modifications” on page 19-31
- “Update Revisions of Project Files” on page 19-32
- “View Modified Files” on page 19-36
- “Commit Modified Files to Source Control” on page 19-42

More About

- “About Source Control with Projects” on page 19-2

Check Out SVN Repository

Create a new local copy of a project by retrieving files from SVN source control.

1. On the MATLAB Home tab, select **New > Project > From SVN**.

   Alternatively, on the Simulink start page, click the **Project from SVN** template.

2. If you know your repository location, enter it into the **Repository path** field and proceed to step 3.

   Otherwise, to browse for and validate the repository path from which to retrieve files, click **Change**.

   a. In the Specify SVN Repository URL dialog box, enter a URL using the list of recent repositories or the **Repository** button  .

   Caution Use `file://` URLs only for single-user repositories. For more information, see “Share a Subversion Repository” on page 19-14.

   b. Click **Validate** to check the repository path.

   If an authentication dialog box for your repository appears, enter the login information to continue.

   c. If the path is invalid, check the URL against your source control repository browser.

   If necessary, select a deeper folder in the repository tree. If your repository contains tagged versions of files, then you might want to check out from `trunk` or from a branch folder under `tags`. For more information, see “Tag and Retrieve Versions of Project Files” on page 19-29. You can check out from a branch, but the built-in SVN integration does not support branch merging. Use an external tool such as TortoiseSVN to perform branch merging.

   d. When you have finished specifying the URL path you want to retrieve, click **OK**.

3. In the New Project From Source Control dialog box, in the **Sandbox** field, select the working folder where you want to put the retrieved files for your new project.

4. Click **Retrieve**.

   If an authentication dialog box for your repository appears, enter the login information to continue.

   Caution Use local sandbox folders. Using a network folder with SVN slows source control operations.

   If your repository already contains a project, then the project is ready when the tool finishes retrieving files to your selected sandbox folder.

   If your sandbox does not yet contain a project, then a dialog box asks whether you want to create a project in the folder. To create a project, specify a project name and click **OK**. The Welcome screen appears to help you set up your new project. For more information about setting up a project, see “Create a New Project From a Folder” on page 16-14.

Note To update an existing project sandbox from source control, see “Update Revisions of Project Files” on page 19-32.

See Also

Related Examples

- “Set Up SVN Source Control” on page 19-9
- “Get SVN File Locks” on page 19-34
- “Work with Project Files” on page 17-7
- “Tag and Retrieve Versions of Project Files” on page 19-29
- “Refresh Status of Project Files” on page 19-30
- “Check for Modifications” on page 19-31
- “Update Revisions of Project Files” on page 19-32
- “View Modified Files” on page 19-36
- “Commit Modified Files to Source Control” on page 19-42

More About

- “About Source Control with Projects” on page 19-2

Tag and Retrieve Versions of Project Files

With SVN, you can use tags to identify specific revisions of all project files. Not every source control has the concept of tags. To use tags with SVN, you need the standard folder structure in your repository and you need to check out your files from `trunk`. See “Standard Repository Structure” on page 19-14.

1. On the Project tab, in the Source Control section, click **Tag**.
2. Specify the tag text and click **OK**. The tag is added to every project file.

   Errors appear if you do not have a `tags` folder in your repository.

Note You can retrieve a tagged version of your project files from source control, but you cannot tag them again with a new tag. You must check out from `trunk` to create new tags.

To retrieve the tagged version of your project files from source control:

1. On the **Project** tab, click **New**, and then in the start page, click **Source Control**.
2. Click **Change** to select the Repository path that you want to retrieve files from.

   The Specify SVN Repository URL dialog box opens.

   a. Select a recent repository from the **Repository** list, or click the **Repository** button  to browse for the repository location.
   b. Click **Validate** to show the repository browser.
   c. Expand the `tags` folder in the repository tree, and select the tag version you want.
   d. Click **OK** to continue and return to the new project dialog box.
3. Select the local folder to receive the tagged files. You must use an empty project folder. (If you try to retrieve tagged files into an existing project folder, an error appears.)
4. Click **Retrieve**.

Alternatively, you can use labels to apply any metadata to files and manage configurations. You can group and sort by labels, and create batch jobs to export files by label. See “Add Labels to Files” on page 17-16.

With Git, you can switch branches. See “Branch and Merge Files with Git” on page 19-50.

See Also

Related Examples

- “Standard Repository Structure” on page 19-14
- “Add Labels to Files” on page 17-16
- “Branch and Merge Files with Git” on page 19-50

Refresh Status of Project Files

To check for locally modified project files, on the Project tab, in the Source Control section, click **Refresh**.

Refresh queries the local sandbox state and checks for changes made with another tool outside of MATLAB.

Note For SVN, **Refresh** does not contact the repository. To check the repository for later revisions, use **Check for Modifications** instead. To get the latest revisions, use **Update** instead. See “Check for Modifications” on page 19-31 and “Update Revisions of Project Files” on page 19-32.

The buttons in the Source Control section of the Project tab apply to the whole project.

Refresh refreshes the view of the source control status for all files under `projectroot`. Clicking **Refresh** updates the information shown in the **Revision** column and the source control status column (for example, **SVN**, or **Git** columns). Hover over the icon to see the tooltip showing the source control status of a file, e.g., **Modified**.

See Also

Related Examples

- “Check for Modifications” on page 19-31
- “Update Revisions of Project Files” on page 19-32
- “Revert Changes” on page 19-44

Check for Modifications

To check the status of individual files for modifications, right-click files in the project and select **Source Control > Check for Modifications**. Use this to find out if the repository version has moved ahead.

With SVN, this option contacts the repository to check for external modifications. The project compares the revision numbers of the local file and the repository version. If the revision number in the repository is larger than that in the local sandbox folder, then the project displays `(not latest)` next to the revision number of the local file.

If your local file is not the latest version, get the latest revisions from the repository by clicking **Update**. See “Update Revisions of Project Files” on page 19-32. You might need to resolve conflicts after updating. See “Resolve Conflicts” on page 19-54 and “Compare Revisions” on page 19-39.

To check for locally modified files, use **Refresh** instead. See “Refresh Status of Project Files” on page 19-30.

See Also

Related Examples

- “Refresh Status of Project Files” on page 19-30
- “Update Revisions of Project Files” on page 19-32
- “Compare Revisions” on page 19-39
- “Revert Changes” on page 19-44

Update Revisions of Project Files

In this section...

Update Revisions with SVN

In a project, to get the latest revisions of all project files from the source control repository, click **Update** in the source control section of the project tab.

Use **Update** to get other people’s changes from the repository and find out about any conflicts. If you want to back out local changes, use **Revert Project** instead. See “Discard Local Changes” on page 19-44.

After you update, the project displays a dialog box listing all the files that have changed on disk. You can control this behavior using the project preference **Show changes on source control update**.

When your project uses SVN source control, **Update** calls `svn update` to bring changes from the repository into your working copy. If there are other people’s changes in your modified files, SVN adds conflict markers to the file. SVN preserves your modifications.

Caution Ensure you have registered SLX files as binary with SVN before using **Update**. If you do not, SVN conflict markers can corrupt your SLX file. The project warns you about this when you first click **Update** to ensure you protect your model files. See “Register Model Files with Subversion” on page 19-11.

You must resolve any conflicts before you can commit. See “Resolve Conflicts” on page 19-54.

Update Revisions with Git

If you are using Git source control, click **Pull** in the source control pane.

Caution Ensure you have registered SLX files as binary with Git before using **Pull**. If you do not, conflict markers can corrupt your SLX file. See “Set Up Git Source Control” on page 19-16.

Pull fetches the latest changes and merges them into your current branch. If you are not sure what is going to come in from the repository, use fetch to examine the changes first and then merge the changes manually.

Pull might fail if you have conflicts. With a complicated change you might want to create a branch from the origin, make some compatibility changes, then merge that branch into the main tracking branch. For next steps, see “Pull, Push, and Fetch Files with Git” on page 19-46.

Update Selected Files

To update selected files, right-click and select the **Update** command for the source control system you are using. For example, if you are using SVN, select **Source Control > Update from SVN** to get fresh local copies of the selected files from the repository.

See Also

Related Examples

- “Register Model Files with Source Control Tools” on page 19-8
- “Resolve Conflicts” on page 19-54
- “Discard Local Changes” on page 19-44

Get SVN File Locks

To ensure users remember to get a lock on model files before editing, you can configure SVN to make model files read only. Follow the steps in “Enforce SVN Locking Model Files Before Editing” on page 19-13. After you configure SVN to make files with certain extensions read only, then users must get a lock on these read-only files before editing.

1. In a project, in any Files view, select the files you want to check out.
2. Right-click the selected files and select **Source Control > Get File Lock**.

   Get File Lock is for SVN. This option does not modify the file in your local sandbox. Git does not have locks.

   A lock symbol appears in the SVN source control column. Other users cannot see the lock symbol in their sandboxes, but they cannot get a file lock or check in a change when you have the lock. To view or break locks, click **Locks** on the Project tab.

Note To get a fresh local copy of the file from the repository, select **Update from SVN**.

In the Simulink Editor, if an open model belongs to a project under SVN, you can get a lock by selecting **File > Project > Get File Lock**.

If you see an SVN message reporting a `working copy locked` error, remove stale locks by clicking **SVN Cleanup** in the Source Control section on the Project tab. SVN uses working copy locks internally and they are not the file locks you control using **Get File Lock**.

Note Starting in R2020a Update 5, SVN cleanup only removes stale locks and unfinished transactions. It does not remove unversioned or ignored files.

You can manually remove unversioned and ignored files.

1. In the **Files** view, in the **All** tab, click the **SVN** header to sort files by their SVN status.
2. Select the **Not Under Source Control** files.
3. Right-click and select **Delete**.

Manage SVN Repository Locks

To manage global SVN locks for a repository, on the Project tab, in the Source Control section, click **Locks**.

In the SVN Repository Locks dialog box, you can:

- View which users have locks on files.
- Right-click to break locks.
- Group locks by user or file.

See Also

Related Examples

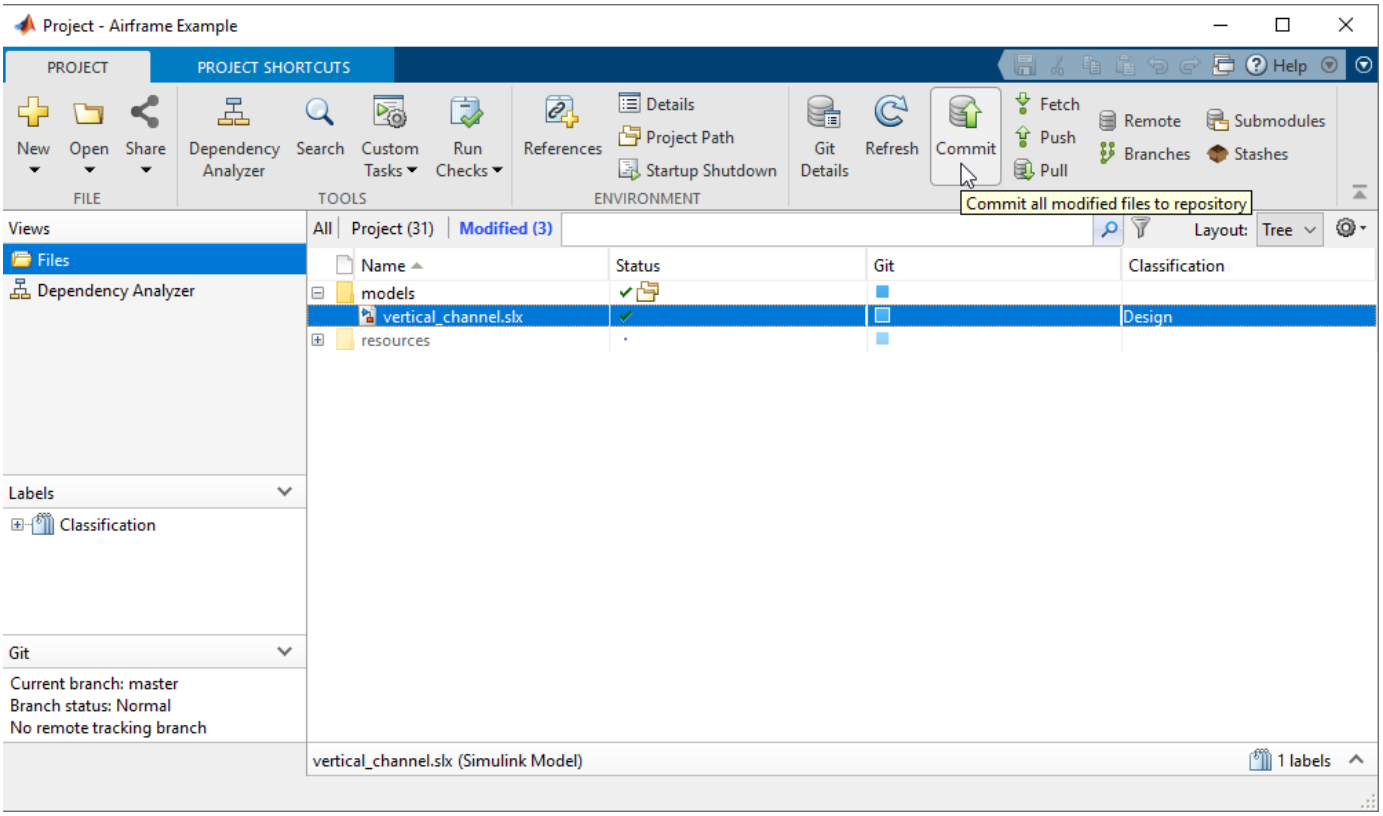
- “Work with Project Files” on page 17-7
- “Enforce SVN Locking Model Files Before Editing” on page 19-13
- “View Modified Files” on page 19-36
- “Commit Modified Files to Source Control” on page 19-42

More About

- “About Source Control with Projects” on page 19-2

View Modified Files

To review, analyze, label, and commit modified project files, use the **Modified (*number of files*)** view. The modified files view is visible only if you are using source control with your project.



If you need to update the modified files list, click **Refresh** in the source control section.

Lists of modified files are sometimes called changesets. You can perform the same operations in the Modified files view as you can in other file views. To view changes, see “Compare Revisions” on page 19-39.

Tip In the **Modified** files view, it can be useful to switch to the `List` layout.

You can identify modified or conflicted folder contents using the source control summary status. In the Files views, folders display rolled-up source control status to help you locate changes in files, particularly conflicted files. Pause on the source control status (for example, the **SVN** or **Git** column) for a folder to see how many files inside are modified, conflicted, added, or deleted.

Project Definition Files

The files in the `resources/project` folder are project definition files generated when you first create or make changes to your project. The project definition files enable you to add project metadata to files without checking them out. Some examples of metadata you can change this way

are shortcuts, labels, and project descriptions. Project definition files also specify the files added to your project. These files are not part of the project.

Any changes you make to your project generate changes in the `resources/project` folder. These files store the definition of your project in XML files whose format is subject to change.

You do not need to view project definition files directly, except when the source control tool requires a merge. The files are shown so that you know about all the files being committed to the source control system. See “Resolve Conflicts” on page 19-54.

Starting in R2020b, the default project definition file type is **Use multiple project files (fixed-path length)**. To change the project definition file management from the type selected when the project was created, use `matlab.project.convertDefinitionFiles`. `matlab.project.convertDefinitionFiles` preserves the source control history of your project.

Warning To avoid merge issues, do not convert the definition file type more than once for a project.

For releases before R2020b, if you want to change the project definition file management from the type selected when the project was created:

1. On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB > Project** and in the **New Projects** section, select one of the options under **Project definition files:**
   - **Use multiple project files** - Helps to avoid merging issues on shared projects
   - **Use multiple project files (fixed-path length)** - Is better if you need to work with long paths
   - **Use a single project file (not recommended for source control)** - Is faster but is likely to cause merge issues when two users submit changes in the same project to a source control tool
2. Create a project archive file (`.mlproj`). For more information, see “Archive Projects” on page 17-34 or `export`.
3. Create a new project from the archived project. For more information, see “Create a New Project from an Archived Project” on page 16-20.

You can use the project preferences to change the project definition folder for new projects. Instead of `resources/project`, you can choose the `.SimulinkProject` or `_SimulinkProject` folder names adopted for releases before R2019a.

To stop managing your folder with a project and delete the `resources/project` folder, see `matlab.project.deleteProject`.

See Also

Related Examples

- “Compare Revisions” on page 19-39
- “Run Project Checks” on page 19-41
- “Refresh Status of Project Files” on page 19-30
- “Check for Modifications” on page 19-31

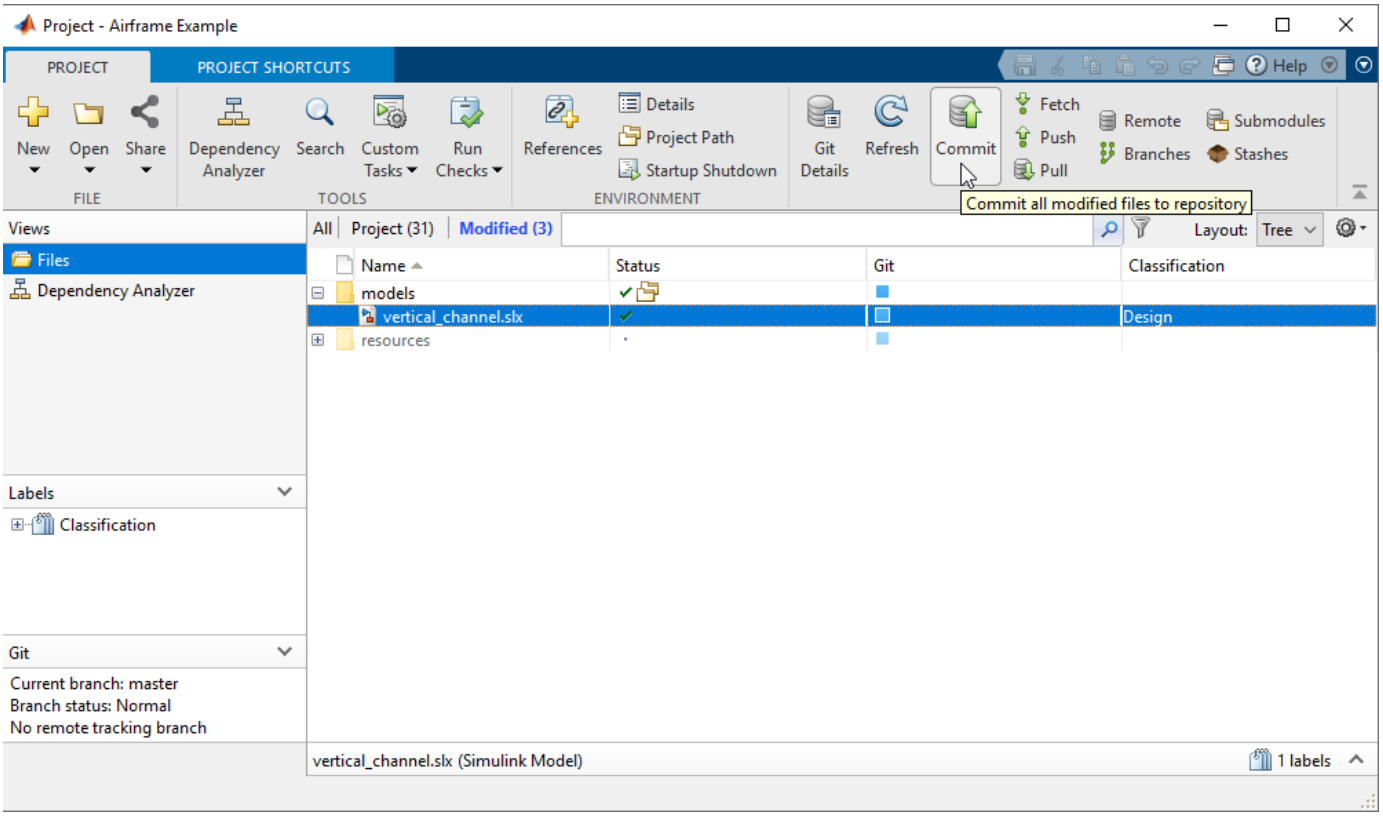
- “Resolve Conflicts” on page 19-54
- “Discard Local Changes” on page 19-44
- “Commit Modified Files to Source Control” on page 19-42

More About

- “About Source Control with Projects” on page 19-2

Compare Revisions

To review changes in modified files in a project, select the **Modified (*number of files*)** tab.



If you need to update the modified files list, click **Refresh** in the source control section of the Project tab.

To review changes in modified files, right-click selected files in any view in a project and:

- Select **Compare > Compare to Ancestor** to run a comparison with the last checked-out version in the sandbox (SVN) or against the local repository (Git). The Comparison Tool displays a report.
- To compare other revisions of a file, select **Compare > Compare to Revision**. In the Compare to Revisions dialog box, you can view information about who previously committed the file, when they committed it, and the log messages. To view a comparison report, select the revisions you want to compare. You can:
  - Select a revision and click **Compare to Local**.
  - Select two revisions and click **Compare Selected**.
  - With SVN, select a revision and you can browse the lower list of files in the change set. Right-click a file in the list to view changes or save revisions..
- To browse the revision history of a file, select **Source Control > Show Revisions**. In the File Revisions dialog box, view information about who previously committed the file, when they committed it, and the log messages. With SVN, select a revision and you can browse the lower list of files in the change set. Right-click a file in the list to view changes or save revisions.

- To browse and compare files within committed SVN change sets, on the Project tab, in the Source Control section, select **Show Log**. In the File Revisions dialog box, select a revision to view a list of modified files. Right-click files in the lower list to view changes or save revisions.

Note In the Simulink Editor, if an open model, library, or chart belongs to a project under source control, you can view changes. To do so, on the **Simulation** tab, select **Project > Compare to Ancestor** or **Compare to Revision**.

When you compare to a revision or ancestor, the MATLAB Comparison Tool opens a report comparing the modified version of the file in your sandbox with the selected revision or against its ancestor stored in the version control tool.

Comparison type depends on the file you select. If you select a Simulink model, this command runs a Simulink model comparison.

When reviewing changes, you can merge Simulink models from the Comparison Tool report. See “Merge Text Files” on page 19-55 and “Merge Models” on page 19-56.

To examine the dependencies of modified files, see “Perform an Impact Analysis” on page 18-17.

See Also

Related Examples

- “Resolve Conflicts” on page 19-54
- “Run Project Checks” on page 19-41
- “Perform an Impact Analysis” on page 18-17
- “Commit Modified Files to Source Control” on page 19-42
- “Revert Changes” on page 19-44
- “Customize External Source Control to Use MATLAB for Diff and Merge” on page 19-59

More About

- “About Source Control with Projects” on page 19-2

Run Project Checks

In a project, you can run checks from any project view by clicking **Run Checks > Check Project** on the **Project** tab. The project checks can find problems with project integrity such as missing files, unsaved files, or files not under source control.

For details on problems the checks can fix, see “Work with Derived Files in Projects” on page 19-58, “Convert from MDL to SLX in a Project and Preserve Revision History”, and “Check Dependency Results and Resolve Problems” on page 18-23.

- Click **Check Project** to check the integrity of the project. For example, is everything under source control in the project? Are all project files under source control? A dialog box reports results. You can click for details and follow prompts to fix problems.

  For an example showing how the checks can help you, see “Convert from MDL to SLX in a Project and Preserve Revision History”.
- If you want to check for required files, click **Dependency Analysis** to analyze the dependencies of the modified files.

  Use the dependency tools to analyze the structure of your project. See “Perform an Impact Analysis” on page 18-17.

Note The files in `resources/project` are project definition files generated by your changes. See “Project Definition Files” on page 19-36.

See Also

Related Examples

- “Find Models and Other Project Files With Unsaved Changes” on page 17-8
- “Commit Modified Files to Source Control” on page 19-42
- “Work with Derived Files in Projects” on page 19-58
- “Convert from MDL to SLX in a Project and Preserve Revision History”
- “Check Dependency Results and Resolve Problems” on page 18-23

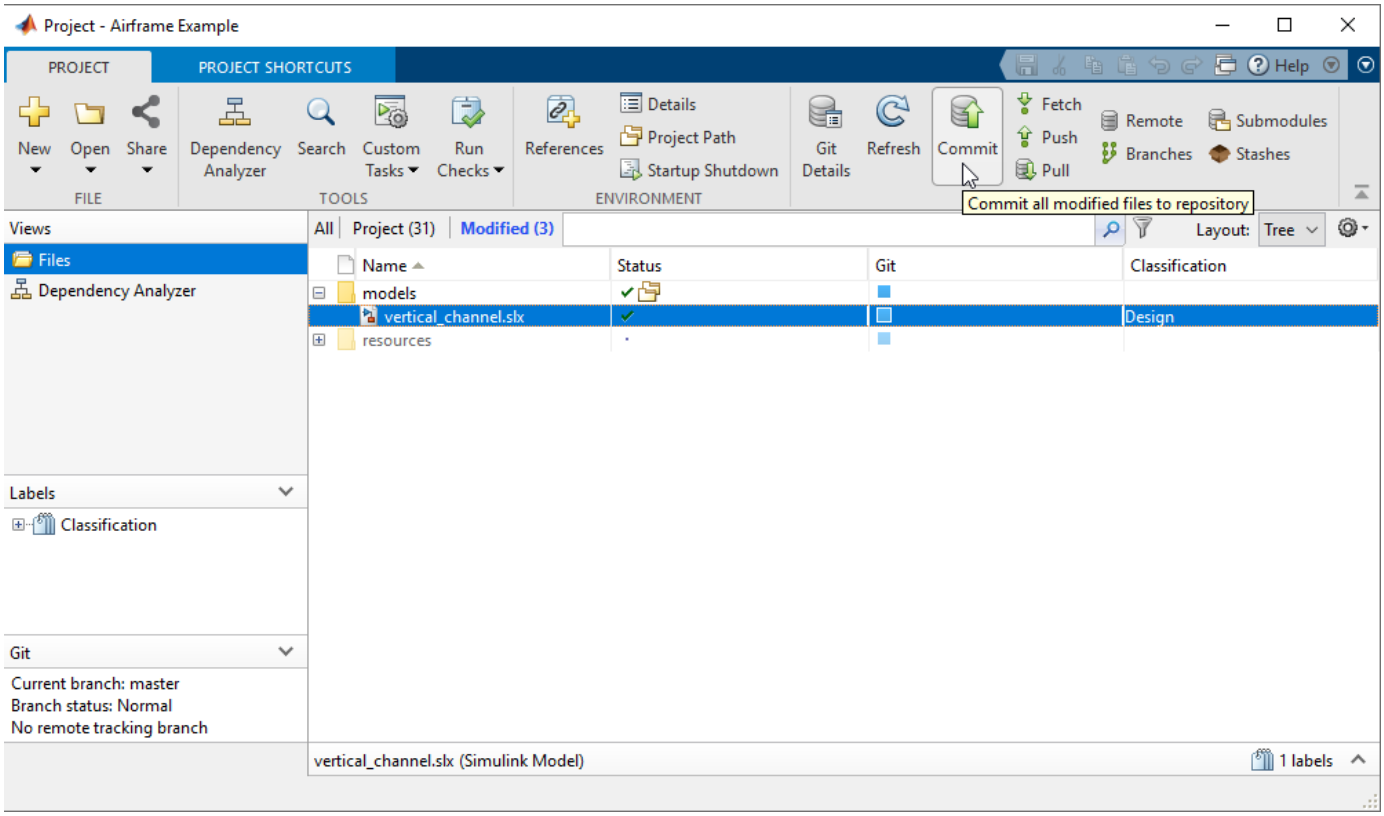
More About

- “About Source Control with Projects” on page 19-2
- “What Is Dependency Analysis?” on page 18-2
- “Project Definition Files” on page 19-36

Commit Modified Files to Source Control

Before you commit modified files, review changes and consider precommit actions. See “Compare Revisions” on page 19-39 and “Run Project Checks” on page 19-41.

1 In a project, select the **Modified (*number of files*)** view.



If you need to update the modified files list, click **Refresh** in the source control section of the Project tab.

2 To check in all files in the modified files list, on the Project tab, in the Source Control section, click **Commit**.

If you are using SVN source control, this commits changes to your repository.

If you are using Git source control, this commits to your local repository. To commit to the remote repository, see “Pull and Push” on page 19-46.

3 Enter comments in the dialog box if you want, and click **Submit**.

4 A message appears if you cannot commit because the repository has moved ahead. Before you can commit the file, you must update its revision up to the current HEAD revision. If you are using SVN source control, click **Update**. If you are using Git source control, click **Pull**. Resolve any conflicts before you commit.

Note You can commit individual files using the context menu, by selecting **Source Control > Commit**. However if you commit individual files, you risk not committing the related project

definition files that keep track of your files. Instead, use the **Modified** files view to see all changes, and on the Project tab, click **Commit** to commit all related changes.

See Also

Related Examples

- “Refresh Status of Project Files” on page 19-30
- “View Modified Files” on page 19-36
- “Run Project Checks” on page 19-41
- “Update Revisions of Project Files” on page 19-32
- “Pull, Push, and Fetch Files with Git” on page 19-46
- “Resolve Conflicts” on page 19-54
- “Revert Changes” on page 19-44

More About

- “About Source Control with Projects” on page 19-2

Revert Changes

In this section...

Discard Local Changes

With SVN, if you want to roll back local changes in a particular file, in a project, right-click the file and select **Source Control > Discard Local Changes and Release Locks** to release locks and revert to the version in the last sandbox update (that is, the last version you synchronized or retrieved from the repository).

In the Simulink Editor, if an open model belongs to a project under source control, you can revert changes. To do so, on the **Simulation** tab, select **Project > Discard Local Changes and Release Locks**.

To abandon all local changes, in a project select all the files in the **Modified** files list, then right-click and select **Discard Local Changes and Release Locks**.

With Git, right-click a file and select **Source Control > Revert Local Changes**. Git does not have locks. To remove all local changes, click **Branches** in the **Git** pane and click **Revert to Head**.

Revert a File to a Specified Revision

1. Right-click a file and select **Source Control > Revert using SVN** or **Source Control > Revert using Git**.
2. In the Revert Files dialog box, choose a revision to revert to. Select a revision to view information about the change such as the author, date, log message.

   With SVN, select a revision and you can browse the lower list of files in the change set. Right-click a file in the list to view changes or save revisions.
3. Click **Revert**.

   The project reverts the selected file.
4. If you revert a file to an earlier revision and then make changes, you cannot commit the file until you resolve the conflict with the repository history.

   With SVN, if you try to commit the file, you see a message that it is out of date. Before you can commit the file, you must update its revision up to the current HEAD revision. click **Update** in the source control section on the Project tab.

   The project marks the file as conflicted because you have made changes to an earlier version of the file than the version in the repository.
5. With either SVN or Git, to examine conflicts, right-click and select **View Conflicts**.

   Decide how to resolve the conflict or to keep your changes to the reverted file. See “Resolve Conflicts” on page 19-54.

6 After you have resolved the conflict, mark the conflict resolved, either by using the merge tool or manually by right-clicking the file and selecting **Source Control > Mark Conflict Resolved**.

7 Select the **Modified (*number of files*)** view to check changes, and on the **Project** tab, click **Commit**.

Revert the Project to a Specified Revision

With SVN, inspect the project revision information by clicking **Show Log** in the in Source Control section on the Project tab. In the Log dialog box, each revision in the list is a change set of modified files. Select a revision to view information about the change such as the author, date, log message and the list of modified files.

To revert the project:

1 On the Project tab, in the Source Control section, click **Revert Project**.

2 In the Revert Files dialog box, choose a revision to revert to.

   Each revision in the list is a change set of modified files. Select a revision to view information about the change such as the author, date, and the log message.

   With SVN, select a revision and you can browse the lower list of files in the change set. Right-click a file in the list to view changes or save revisions.

3 Click **Revert**.

   The project displays progress messages in the SVN pane as it restores the project to the state it was in when the selected revision was committed. Depending on the change set you selected, all files do not necessarily have a particular revision number or matching revision numbers. For example, if you revert a project to revision 20, all files will show their revision numbers when revision 20 was committed (20 or lower).

With Git, you can switch branches. See “Branch and Merge Files with Git” on page 19-50.

See Also

Related Examples

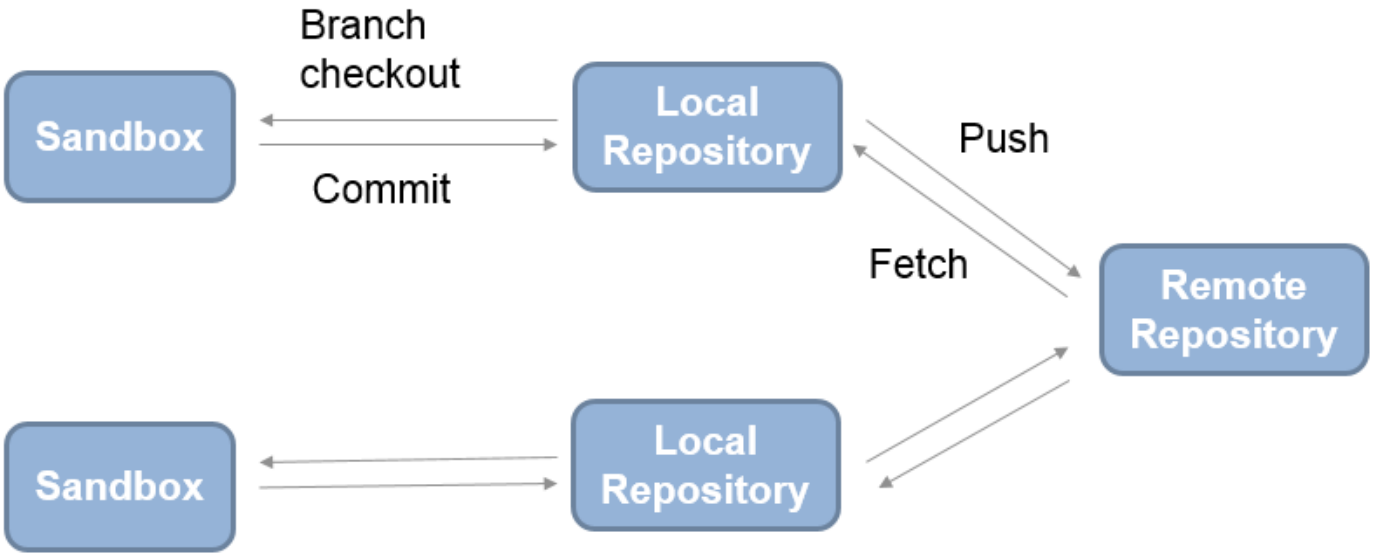
- “Resolve Conflicts” on page 19-54

Pull, Push, and Fetch Files with Git

In this section...

Pull and Push

Use this workflow to work with a Git project connected to a remote repository. With Git, there is a two-step workflow: commit local changes, and then push to the remote repository. In a project, the only access to the remote repository is through the **Pull**, **Push**, and **Fetch** buttons. All other actions use the local repository (such as **Check for Modifications**, **Compare to Ancestor**, and **Commit**). This diagram represents the Git workflow.



1. To get the latest changes, on the Project tab, in the Source Control section, click **Pull**. Pull fetches the latest changes and merges them into your current branch.

   Note Before you can merge, you must register model files as binary to prevent Git from inserting conflict markers. See “Register Model Files with Source Control Tools” on page 19-8.

2. To create branches to work on, on the Project tab, in the Source Control section, click **Branches**. Create branches in the Branches dialog box, as described in “Branch and Merge Files with Git” on page 19-50.

3. When you want to commit changes, select the Modified files view to view files, and on the **Project** tab, click **Commit**. The changes are committed to your current branch in your local repository. Check the **Git** pane for information about the current branch. You see the message `Ahead` when you commit local changes that have moved ahead of the remote tracking branch.

Current branch: master
Branch status: SAFE
Ahead of /origin/master

4 To send your local commits to the remote repository, on the Project tab, in the Source Control section, click **Push**.

5 A message appears if you cannot push your changes directly because the repository has moved on. Click **Fetch** to fetch changes from the remote repository. Merge branches and resolve conflicts, and then you can push your changes. See “Pull, Fetch, and Merge” on page 19-47.

Pull, Fetch, and Merge

Use **Fetch** to get changes and merge manually. Use **Pull** instead to fetch the latest changes and merge them into your current branch.

Note Before you can merge branches, you must register model files as binary to prevent Git from inserting conflict markers. See “Register Model Files with Source Control Tools” on page 19-8.

Pull fetches the latest changes and merges them into your current branch. If you are not sure what is going to come in from the repository, use fetch instead to examine the changes, and then merge the changes manually.

Pull might fail if you have conflicts. With a complicated change you might want to create a branch from the origin, make some compatibility changes, then merge that branch into the main tracking branch.

To fetch changes from the remote repository, click **Fetch** on the Project tab.

Fetch updates all of the origin branches in the local repository.

Note When you click **Fetch**, your sandbox files are not changed. To see others’ changes, you need to merge in the origin changes to your local branches.

Check the Git pane for information about your current branch relative to the remote tracking branch in the repository. When you see the message `Behind`, you need to merge in changes from the repository to your local branch.

For example, if you are on the master branch and want to get changes from the master branch in the remote repository:

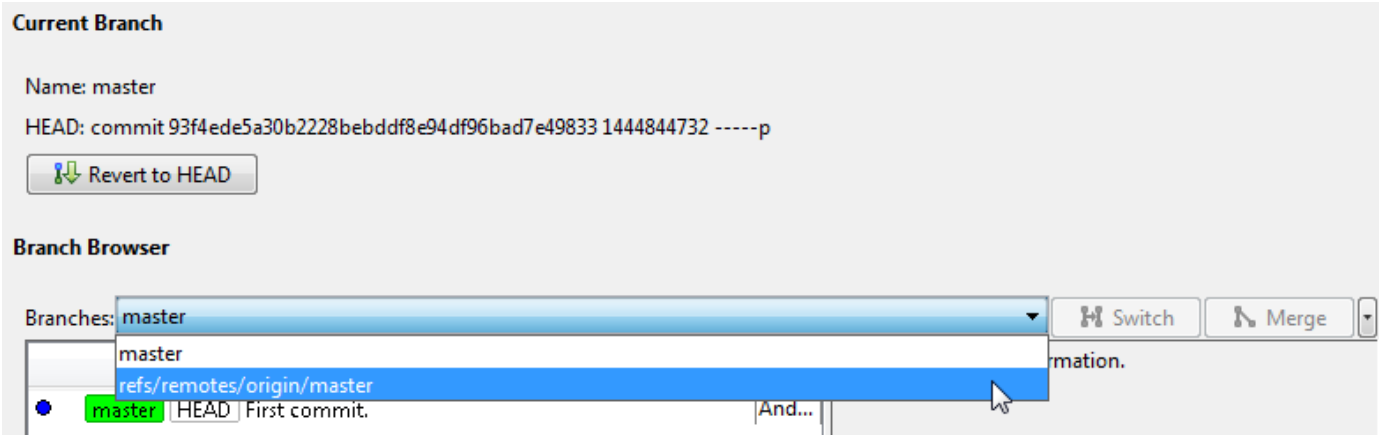
1 Click **Fetch**.

Observe the message in the Git pane, `Behind /origin/master`. You need to merge in the changes from the repository to your local branch, using **Branches**.

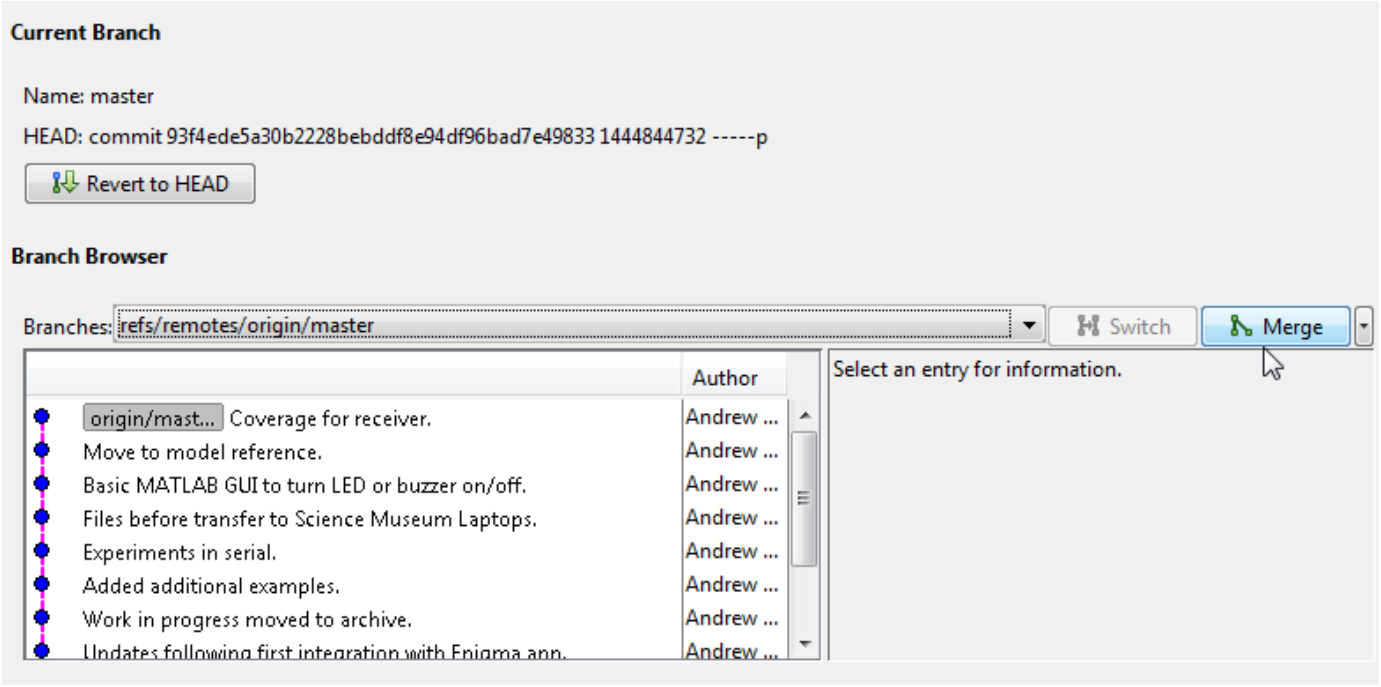
Current branch: master
Branch status: SAFE
Behind /origin/master

2 Click **Branches**.

3 In the Branches dialog box, in the **Branches** list, select **origin/master**.



4 Click **Merge**. This merges the origin branch changes into the master branch in your sandbox.



5 Close the Branches dialog box. Observe the message in the Git pane now says `Coincident with /origin/master`. You can now view the changes fetched and merged from the remote repository in your local sandbox files.

When you fetch and merge, you might need to resolve conflicting changes. If the branch merge causes a conflict that Git cannot resolve automatically, an error dialog box reports that automatic merge failed. Resolve the conflicts before proceeding. See “Resolve Conflicts” on page 19-54.

Push Empty Folders

Using Git, you cannot add empty folders to source control, so you cannot select **Push** and then clone an empty folder. You can create an empty folder in a project, but if you push changes and then sync a new sandbox, then the empty folder does not appear in the new sandbox. You can instead run **Check Project** which creates the empty folder for you.

Alternatively, to push empty folders to the repository for other users to sync, create a `gitignore` file in the folder and then push your changes.

Use Git Stashes

Store uncommitted changes for later use by creating a Git stash. Use stashes to:

- Store modified files without committing them.
- Move changes easily to a new branch.
- Browse and examine the changes within a stash.

To create and manage stashes, on the **Project** tab, in the **Source Control** section, click **Stashes**. In the Stashes dialog box:

- To create a stash containing your currently modified files, click **New Stash**.
- To view modified files in a stash, select the stash under **Available Stashes**. Right-click modified files to view changes or save a copy.
- To apply the stash to your current branch and then delete the stash, click **Pop**.
- To apply the stash and keep it, click **Apply**.
- To delete the stash, click **Drop**.

See Also

Related Examples

- “Set Up Git Source Control” on page 19-16
- “Branch and Merge Files with Git” on page 19-50
- “Resolve Conflicts” on page 19-54

More About

- “About Source Control with Projects” on page 19-2

Branch and Merge Files with Git

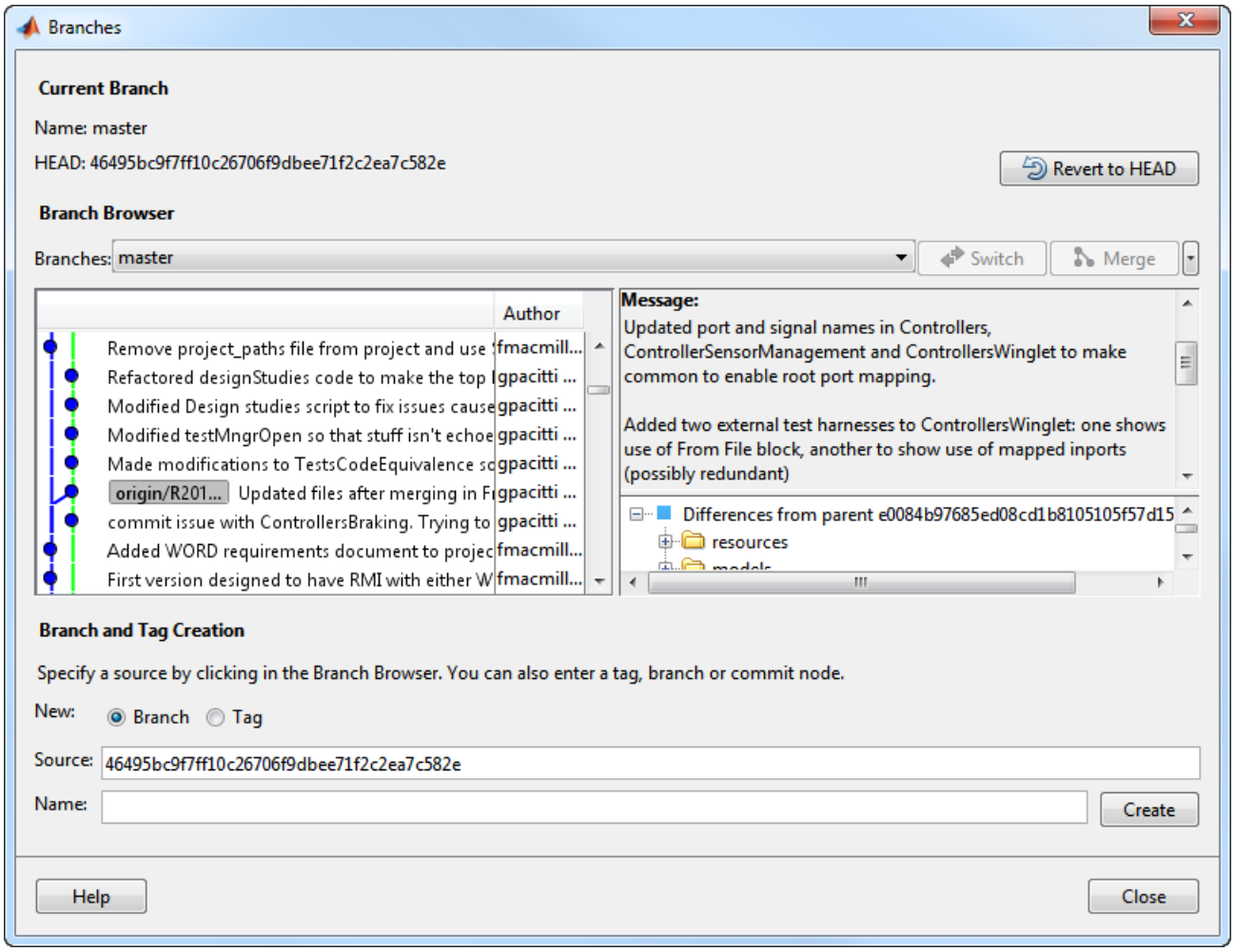
In this section...

Create a Branch

1 In a project using Git source control, click **Branches** on the Project tab. The Branches dialog box appears, where you can view, switch, create, and merge branches.

Tip You can inspect information about each commit node. Select a node in the Branch Browser diagram to view the author, date, commit message, and changed files.

The **Branches** pane in this figure shows an example branch history.



2. Select a source for the new branch. Click a node in the Branch Browser diagram, or enter a unique identifier in the Source text box. You can enter a tag, branch name, or a unique prefix of the SHA1 hash (for example, `73c637` to identify a specific commit). Leave the default to create a branch from the head of the current branch.
3. Enter a name in the **Branch name** text box and click **Create**.
4. To work on the files on your new branch, switch your project to the branch.

   In the **Branches** drop-down list, select the branch you want to switch to and click **Switch**.
5. Close the Branches dialog box to return to the project and work on the files on your branch.

For next steps, see “Pull, Push, and Fetch Files with Git” on page 19-46.

Switch Branch

1. In a project, click **Branches**.
2. In the Branches dialog box, select the branch you want to switch to in the **Branches** list and click **Switch**.
3. Close the Branches dialog box to return to the project and work on the files on the selected branch.

Compare Branches and Save Copies

From within your Git repository folder, right-click the white space of the Current Folder browser and select **Source Control** > **Branches**.

- To examine differences in a file between the current revision and its parent, right-click a file in the tree under `Differences from parent` and select **Show Difference**.
- To examine differences in a file between any two revisions including revisions on two different development branches, hold the **Ctrl** key and select the two different revisions. Right-click a file in the tree under `Differences from selection` and select **Show Difference**.

MATLAB opens a comparison report. You can save a copy of the selected file on either revision. Right-click a file and select **Save As** to save a copy of the file on the selected revision. Select **Save Original As** to save a copy of the file on the prior revision. This is useful if you want to test how the code ran in previous revisions or on other branches.

Merge Branches

Before you can merge branches, you must register model files as binary to prevent Git from inserting conflict markers. See “Register Model Files with Source Control Tools” on page 19-8.

Tip After you use **Fetch**, you must merge. See “Pull, Fetch, and Merge” on page 19-47.

To merge any branches:

1. In a project, click **Branches**.
2. In the Branches dialog box, from the **Branches** drop-down list, select a branch you want to merge into the current branch, and click **Merge**.
3. Close the Branches dialog box to return to the project and work on the files on the current branch.

If the branch merge causes a conflict that Git cannot resolve automatically, an error dialog box reports that automatic merge failed. The Branch status in the **Git** pane displays `MERGING`. Resolve the conflicts before proceeding.

Caution Do not move or delete files outside of MATLAB because this can cause errors on merge.

Keep Your Version

1. To keep your version of the file, right-click the file and select **Mark Conflict Resolved**. The Branch status in **Git** pane displays `MERGE_RESOLVED`. The Modified Files list is empty, because you have not changed any file contents. The local repository index version and your branch version are identical.
2. Click **Commit** to commit your change that marks the conflict resolved.

View Conflicts in Branch Versions

If you merge a branch and there is a conflict in a model file, Git marks the file as conflicted and does not modify the contents. Right-click the file and select **View Conflicts**. The project opens a

comparison report showing the differences between the file on your branch and the branch you want to merge into. Decide how to resolve the conflict. See “Resolve Conflicts” on page 19-54.

Revert to Head

To remove all local changes, in the Branches dialog box, click **Revert to Head**.

Delete Branches

1. In the Branches dialog box, from the **Branches** drop-down list, select a branch you want to delete. You cannot delete the current branch.
2. On the far right, click the down arrow and select **Delete Branch**.

Caution You cannot undo deleting a branch.

See Also

Related Examples

- “Set Up Git Source Control” on page 19-16
- “Pull, Push, and Fetch Files with Git” on page 19-46
- “Resolve Conflicts” on page 19-54

More About

- “About Source Control with Projects” on page 19-2

Resolve Conflicts

In this section...

Resolve Conflicts

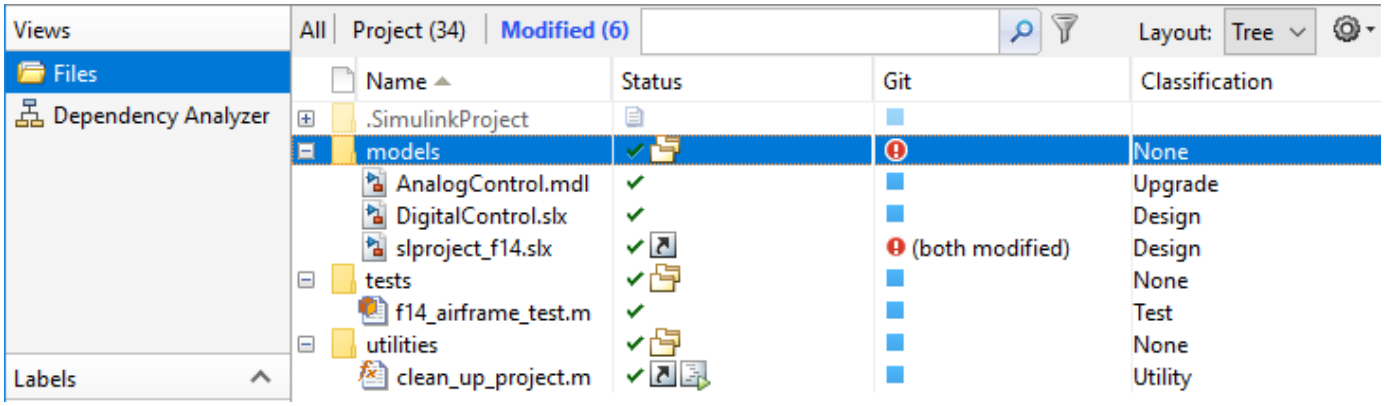
If you and another user change the same file in different sandboxes or on different branches, a conflict message appears when you try to commit your modified files. Extract conflict markers if necessary, compare the differences causing the conflict, and resolve the conflict.

1 Look for conflicted files in the **Modified (*number of files*)** tab.

Identify conflicted folder contents using source control summary status. Folders display rolled-up source control status. This makes it easier to locate changes in files, particularly conflicted files. You can hover over the source control status for a folder to view a tooltip displaying how many files inside are modified, conflicted, added or deleted.

Tip Use the `List` layout to view files without needing to expand folders.

2 Check the source control status column (**Git** or **SVN**) for files with a red warning symbol, which indicates a conflict.



3 Right-click the conflicted file and select **View Conflicts** to compare versions.

4 Examine the conflict. The project opens a comparison report showing the differences between the conflicted files.

- For SVN, the comparison shows the differences between the file and the version of the file in conflict.
- For Git, the comparison shows the differences between the file on your branch and the branch you want to merge into.
- For model files, see “Merge Simulink Models from the Comparison Report” on page 21-16.

5 Use the comparison report to determine how to resolve the conflict.

To resolve conflicts you can:

- Use the report to merge changes between revisions.
- Decide to overwrite one set of changes with the other.
- Make changes manually from the project by editing files, changing labels, or editing the project description.

For details on using the Comparison Tool to merge changes between revisions, see “Merge Text Files” on page 19-55 and “Merge Models” on page 19-56.

6 When you have resolved the changes and want to commit the version in your sandbox, in a project, right-click the file and select **Source Control > Mark Conflict Resolved**. You can use the merge tool to mark the conflict resolved, or you can choose to manually mark the conflict resolved in the project.

For Git, the Branch status in the **Git** pane changes from `MERGING` to `SAFE`.

7 Select the **Modified (*number of files*)** tab to check changes. On the **Project** tab, click **Commit**.

Merge Text Files

When comparing text files, you can merge changes from one file to the other. Merging changes is useful when resolving conflicts between different versions of files.

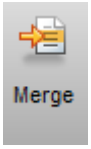
Conflict markers appear in a text comparison report like this:

```
<<<<<<< .mine
```

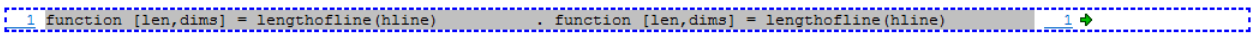
If your comparison report contains conflict markers, extract them before merging, as described in “Extract Conflict Markers” on page 19-56.

Tip You can merge only from left to right. When comparing to another version in source control, the right file is the version in your sandbox. The left file is either a temporary copy of the previous version or another version causing a conflict (e.g., *`filename`*`_theirs`). Observe the file paths of the left and right file at the top of the comparison report. Merge differences from the left (temporary copy) file to the right file to resolve conflicts.

1 In the Comparison Tool report, select a difference in the report and click **Merge**. The selected difference is copied from the left file to the right file.



Merged differences display gray row highlighting and a green merge arrow.



The merged file name at the top of the report displays the dirty flag (*filename*.m*) to show you that the file contains unsaved changes.

2. Click **Save Merged File** to save the file on the right. Check the file path of the right file in the comparison report. (To save to a different file, select **Save Merged File > Save Merged File As**). To resolve conflicts, save the merged file over the conflicted file.
3. If you want to inspect the files in the editor, click the line number links in the report.

Note If you make any further changes in the editor, the comparison report does not update to reflect changes and report links can become incorrect.

4. After merging to resolve conflicts, mark the conflict resolved and commit the changes, as described in “Resolve Conflicts” on page 19-54.

Merge Models

In the Comparison Tool report, you can merge changes between revisions. For details, see “Merge Simulink Models from the Comparison Report” on page 21-16.

After merging to resolve conflicts, the merge tool can mark the conflict resolved for you, or you can choose to manually mark the conflict resolved. Then commit the changes, as described in “Resolve Conflicts” on page 19-54.

Extract Conflict Markers

What Are Conflict Markers?

Source control tools can insert conflict markers in files that you have not registered as binary (e.g., text files). You can use project tools to extract the conflict markers and compare the files causing the conflict. This process helps you to decide how to resolve the conflict.

Caution Register model files with source control tools to prevent them from inserting conflict markers and corrupting models. See “Register Model Files with Source Control Tools” on page 19-8. If your model already contains conflict markers, the project tools can help you to resolve the conflict, but only if you open the model from the project. Opening a model that contains conflict markers from the Current Folder or from a file explorer can fail because Simulink does not recognize conflict markers.

Conflict markers have the following form:

```
<<<<<<<["mine" file descriptor]
["mine" file content]
=======
["theirs" file content]
>>>>>>>["theirs" file descriptor]
```

If you try to open a file marked conflicted that contains conflict markers, the Conflict Markers Found dialog box opens. Follow the prompts to fix the file by extracting the conflict markers. After you extract the conflict markers, resolve the conflicts as described in “Resolve Conflicts” on page 19-54.

To view the conflict markers, in the Conflict Markers Found dialog box, click **Load File**. Do not try to load model files, because Simulink does not recognize conflict markers. Instead, click **Fix File** to extract the conflict markers.

By default, the project checks only conflicted files for conflict markers. You can change this preference to check all files or no files. Click **Preferences** in the Project tab to change the setting.

Extract Conflict Markers

When you open a conflicted file or select **View Conflicts**, the project checks files for conflict markers and offers to extract the conflict markers. The project checks only conflicted files for conflict markers unless you change your preferences setting.

However, some files that are not marked conflicted can still contain conflict markers. This can happen if you or another user marked a conflict resolved without removing the conflict markers and then committed the file. If you see conflict markers in a file that is not marked conflicted, you can remove the conflict markers.

1. In a project, right-click the file and select **Source Control > Extract Conflict Markers to File**.
2. Leave the default option to copy the “mine” revision over the conflicted file. Leave the **Compare** check box selected. Click **Extract**.
3. Use the Comparison Tool report as usual to continue to resolve the conflict.

See Also

Related Examples

- “Register Model Files with Source Control Tools” on page 19-8
- “Merge Simulink Models from the Comparison Report” on page 21-16

Work with Derived Files in Projects

Best practice is to omit derived and temporary files from your project or exclude them from source control. On the **Project** tab, select **Run Checks** > **Check Project** to check the integrity of the project. If you add the `slprj` folder to a project, the project checks advise you to remove this from the project and offer to make the fix.

Best practice is to exclude derived files, such as `.mex*`, the contents of the `slprj` folder, `sccprj` folder, or other code generation folders from source control, because they can cause problems. For example:

- With a source control that can do file locking, you can encounter conflicts. If `slprj` is under source control and you generate code, most of the files under `slprj` change and become locked. Other users cannot generate code because of file permission errors. The `slprj` folder is also used for simulation via code generation (for example, with model reference or Stateflow), so locking these files can have an impact on a team. The same problems arise with binaries, such as `.mex*`.
- Deleting `slprj` is often required. However, deleting `slprj` causes problems such as “not a working copy” errors if the folder is under some source control tools (for example, SVN).
- If you want to check in the generated code as an artifact of the process, it is common to copy some of the files out of the `slprj` cache folder and into a separate location that is part of the project. That way, you can delete the temporary cache folder when you need to. See `packNGo` to discover the list of generated code files, and use the project API to add to the project with appropriate metadata.
- The `slprj` folder can contain many small files. This can affect performance with some source control tools when each of those files is checked to see if it is up-to-date.

See Also

`currentProject` | `packNGo`

Related Examples

- “Add Files to the Project” on page 16-18
- “Run Project Checks” on page 19-41

Customize External Source Control to Use MATLAB for Diff and Merge

In this section...

You can customize external source control tools to use the MATLAB Comparison Tool for diff and merge. If you want to compare MATLAB files such as live scripts, MAT, SLX, or MDL files from your source control tool, then you can configure your source control tool to open the MATLAB Comparison Tool. The MATLAB Comparison Tool provides tools for merging MathWorks files and is compatible with popular software configuration management and version control systems. You can use the automerge tool with Git to automatically merge branches that contain changes in different subsystems in the same SLX file.

To set up your source control tool to use MATLAB as the application for diff and merge, you must first determine the full paths of the `mlDiff`, `mlMerge`, and `mlAutoMerge` executable files, and then follow the recommended steps for the source control tool you are using.

Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge

To get the required file paths and enable external source control tools to reuse open MATLAB sessions, run this command in MATLAB:

```
comparisons.ExternalSCMLink.setup()
```

This command sets the MATLAB preference, under **Comparison**, called **Allow external source control tools to use open MATLAB sessions for diffs and merges**.

This command also displays the file paths to copy and paste into your source control tool setup:

- On Windows:

  `Diff: `*`matlabroot`*`\bin\win64\mlDiff.exe`

  `Merge: `*`matlabroot`*`\bin\win64\mlMerge.exe`

  `AutoMerge: `*`matlabroot`*`\bin\win64\mlAutoMerge.exe`
- On Linux:

  `Diff: `*`matlabroot`*`/bin/glnxa64/mlDiff`

  `Merge: `*`matlabroot`*`/bin/glnxa64/mlMerge`

  `AutoMerge: `*`matlabroot`*`/bin/glnxa64/mlAutoMerge`
- On Mac:

  `Diff: `*`matlabroot`*`/bin/maci64/mlDiff`

  `Merge: `*`matlabroot`*`/bin/maci64/mlMerge`

```
AutoMerge: matlabroot/bin/maci64/mlAutoMerge
```

where `matlabroot` is replaced with the full path to your installation, for example, `C:\Program Files\MATLAB\R2020b`.

Note Your diff and merge operations use open MATLAB sessions when available, and only open MATLAB when necessary. The operations only use the specified MATLAB installation.

Integration with Git

Command Line

To configure MATLAB diff and merge tools with command-line Git:

1. Run this command in MATLAB.

   ```
   comparisons.ExternalSCMLink.setupGitConfig()
   ```

   This command displays the full paths of the `mlDiff`, `mlMerge`, and `mlAutoMerge` executable files. It also automatically populates the global `.gitconfig` file. For example:

   ```
   [difftool "mlDiff"]
      cmd = \"C:/Program Files/MATLAB/R2020b/bin/win64/mlDiff.exe\" $LOCAL $PWD/$REMOTE
   [mergetool "mlMerge"]
      cmd = \"C:/Program Files/MATLAB/R2020b/bin/win64/mlMerge.exe\" $PWD/$BASE $PWD/$LOCAL $PWD/$REMOTE $PWD/$MERGED
   [merge "mlAutoMerge"]
      driver = \"C:/Program Files/MATLAB/R2020b/bin/win64/mlAutoMerge.exe\" %O %A %B %A
   ```

   Note You need to do step 1 only once for your Git setup.

2. Configure your repository to use the `mlAutoMerge` executable file. Open the `.gitattributes` file in your repository and add:

   ```
   *.slx binary merge=mlAutoMerge
   ```

   Now, when you merge branches that contain changes in different subsystems in the same SLX file, MATLAB handles the merge automatically.

To run the MATLAB diff and merge tools from command-line Git, use `git difftool` and `git mergetool`:

- To compare two revisions of a model using the MATLAB diff tool, type:

  ```
  git difftool -t mlDiff <revisonID1> <revisionID2> myModel.slx
  ```

  If you do not provide revision IDs, `git difftool` compares the working copy to the repository copy.

  If you do not specify which model you want to compare, command-line Git will go through all modified files and ask you if you want to compare them one by one.

- To resolve a merge conflict in a model using the MATLAB merge tool, type:

  ```
  git mergetool -t mlMerge myModel.slx
  ```

  If you do not specify which model you want to merge, command-line Git will go through all files and ask you if you want to merge them one by one.

SourceTree

SourceTree is an interactive GUI tool that visualizes and manages Git repositories for Windows and Mac.

1. Configure the MATLAB diff and merge tools as SourceTree external tools:
   a. With SourceTree open, click **Tools > Options**.
   b. On the **Diff** tab, under **External Diff / Merge**, fill the fields with the following information:

```
External Diff tool: Custom
Diff Command: C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe
Arguments: $LOCAL $PWD/$REMOTE
Merge tool: Custom
Merge Command: C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe
Arguments: $PWD/$BASE $PWD/$LOCAL $PWD/$REMOTE $PWD/$MERGED
```

2. Configure your repository to automerge changes in different subsystems in the same SLX file using the `mlAutoMerge` executable file:
   a. Open the global `.gitconfig` file and add:

```
[merge "mlAutoMerge"]
   driver = \"C:/Program Files/MATLAB/R2020b/bin/win64/mlAutoMerge.exe\" %O %A %B %A
```

   b. Open the `.gitattributes` file in your repository and add:

```
*.slx binary merge=mlAutoMerge
```

Tip Customize the full path of the `mlDiff`, `mlMerge`, and `mlAutoMerge` executables to match both the MATLAB installation and the operating system you are using. For more information, see “Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge” on page 19-59.

To use the MATLAB diff tool from within SourceTree, right-click a modified file under **Unstaged files** and select **External Diff**.

To use the MATLAB merge tool when SourceTree detects a merge conflict, select the **Uncommitted changes** branch, right-click a modified file, and select **Resolve Conflicts > Launch External Merge Tool**.

Integration with SVN

TortoiseSVN

With TortoiseSVN, you can customize your diff and merge tools based on the file extension. For example, to use MATLAB diff and merge tools for SLX files:

1. Right-click in any file explorer window and select **TortoiseSVN > Settings** to open TortoiseSVN settings.
2. In the **Settings** sidebar, select **Diff Viewer**. Click **Advanced** to specify the diff application based on file extensions.
3. Click **Add** and fill the fields with the extension and the `mlDiff` executable path:

```
Filename, extension or mime-type: .slx
External Program: "C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe" %base %mine
```

4. Click **OK** and repeat the same steps to add another file extension.

5. In the **Settings** sidebar, select **Diff ViewerMerge Tool**. Click **Advanced** to specify the merge application based on file extensions.
6. Click **Add** and fill the fields with the extension and `mlMerge` executable path:

   ```
   Filename, extension or mime-type: .slx
   External Program: "C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe" %base %mine %theirs %merged
   ```

7. Click **OK** and repeat the same steps to add another file extension.

You can now use the MATLAB tools for diff and merge the same way you would use the TortoiseSVN default diff and merge applications.

Note Automerging binary files with SVN , such as SLX files, is not supported.

Integration with Other Source Control Tools

Perforce P4V

With Perforce® P4V, you can customize your diff and merge tools based on the file extension. To use MATLAB diff and merge tools for SLX files, for example:

1. In Perforce, click **Edit** > **Preferences**.
2. In the **Preferences** sidebar, select **Diff**. Under **Specify diff application by extension (overrides default)**, click **Add**.
3. In the **Add File Type** dialog box, enter the following information:

   ```
   Extension: .slx
   Application: C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe
   Arguments: %1 %2
   ```

4. Click **Save**.
5. In the **Preferences** sidebar, select **Merge**. Under **Specify merge application by extension (overrides default)**, click **Add**.
6. In the **Add File Type** dialog box, enter the following information:

   ```
   Extension: .slx
   Application: C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe
   Arguments: %b %2 %1 %r
   ```

7. Click **Save** and repeat the steps for other file extensions.

Tip Customize the full path of the `mlDiff` and `mlMerge` executables to match both the MATLAB installation and the operating system you are using. For more information, see “Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge” on page 19-59.

You can now use the MATLAB tools for diff and merge the same way you would use the Perforce default diff and merge applications.

See Also

Related Examples

- “Compare Files and Folders and Merge Files”

- “Compare and Merge MAT-Files”
- “Merge Simulink Models from the Comparison Report” on page 21-16

Project Reference

Componentization Using Referenced Projects

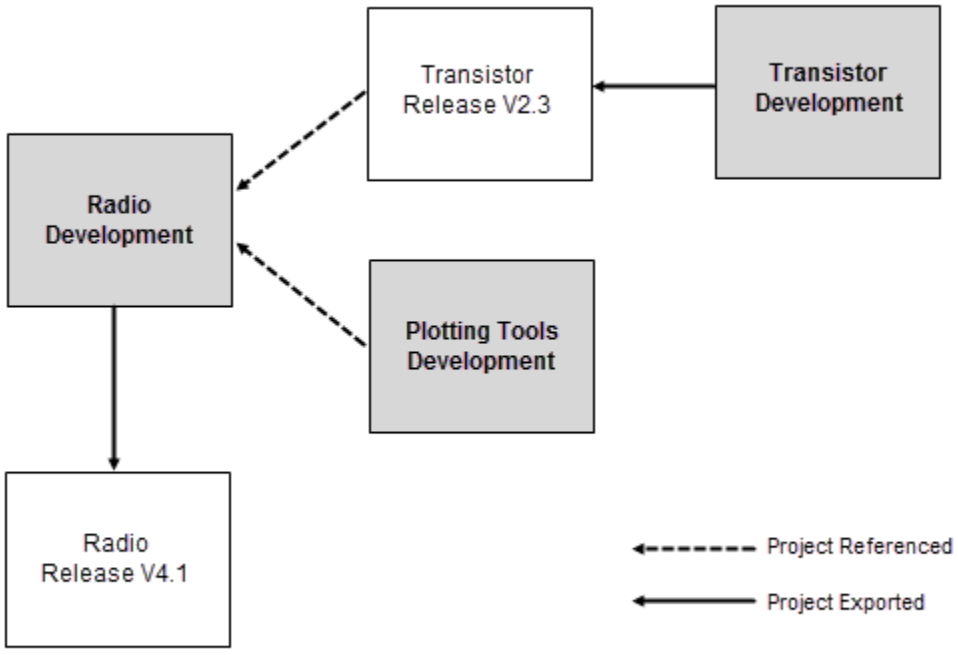
For a large modeling project, organizing the project into components facilitates:

- Component reuse
- Modular, team-based development
- Unit testing
- Independent release of components

Projects supports large-scale project componentization by allowing you to reference other projects from a parent project. A collection of parent and referenced projects constitutes a project reference hierarchy. Project referencing provides these benefits:

- A parent project has access to a referenced project's project paths, entry-point shortcuts, and source control information. For example, from a parent project, you can display the hierarchy of referenced projects. You can select a referenced project and then view, edit, and run files that belong to the referenced project.
- Through a referenced project, your team can develop a component independent of other components.
- In a referenced project, you can test the component separately.
- In a parent project, you can set a checkpoint and then compare the referenced project against the checkpoint to detect any changes.

This project hierarchy illustrates the use of parent and referenced projects as components of a large project.



Through the Transistor Development project, a team independently creates and tests a library of blocks. The team makes the library available to other developers by exporting release versions, for example, version 2.3.

Through the Radio Development project, another team develops and tests the Radio system. This team requires:

- Version 2.3 of the Transistor component. The team sets up the Radio Development project to reference the Transistor Release V2.3 project.
- Tools to plot signals, for example, MATLAB files that are not distributed to customers. The team sets up the Radio Development project to reference the Plotting Tools Development project.

When the Radio system is ready for customers, the team exports a release version, for example, version 4.1.

See Also

Related Examples

- “Organize Large Modeling Projects” on page 16-2
- “Component-Based Modeling Guidelines” on page 22-2
- “Add or Remove a Reference to Another Project” on page 20-4
- “View, Edit, or Run Referenced Project Files” on page 20-5
- “Extract a Folder to Create a Referenced Project” on page 20-6
- “Manage Referenced Project Changes Using Checkpoints” on page 20-8
- “Referencing Projects from Another Project”

Add or Remove a Reference to Another Project

Add new components to your project by referencing other projects. The addition of referenced projects creates a project hierarchy. When the Project loads a referenced project in a project hierarchy, it:

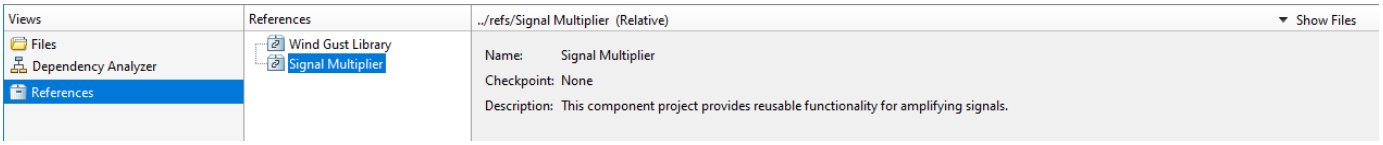
- Adds project paths from the referenced project to the MATLAB search path.
- Runs startup shortcuts from the referenced project.

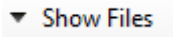
To reference a project:

1. On the Project tab, in the **Environment** section, click  .
2. In the Add Reference dialog box, specify settings:
   - **Referenced project location** –– Click **Browse** and navigate to the project folder. Then, in the project folder, select the required project (`.prj`) file.
   - **Reference type** –– If your project hierarchy has a well-defined root relative to your project root, for example, a folder under source control, click **Relative**. If the project you want to reference is in a location accessible to your computers, for example, a network drive, click **Absolute**.
   - **Set a checkpoint to detect future changes** –– To create a checkpoint, select the check box. To detect changes, you can compare the referenced project against this checkpoint.
3. Click **Add**. The Project creates a References view that displays the referenced project.

   You can reference multiple projects in a hierarchical manner. In the References view, the Project displays the project reference hierarchy as a tree.

To view summary information about a referenced project, in the References tree, select the project.



To view files that belong to the referenced project, click  .

To remove a referenced project from your project hierarchy, in the References tree, right-click the referenced project and select `Remove Reference`.

See Also

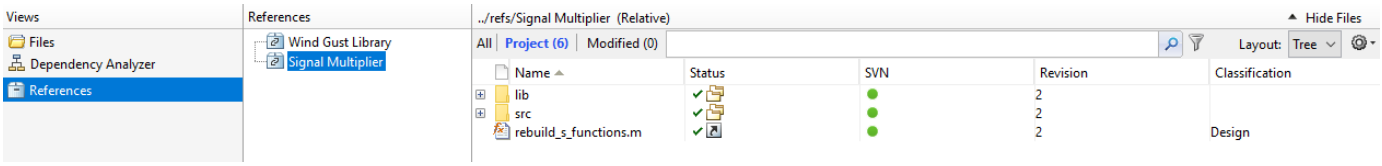
Related Examples

- “Componentization Using Referenced Projects” on page 20-2
- “View, Edit, or Run Referenced Project Files” on page 20-5
- “Extract a Folder to Create a Referenced Project” on page 20-6
- “Referencing Projects from Another Project”

View, Edit, or Run Referenced Project Files

In a project that references other projects, use the References view to see, modify, or run files that belong to the referenced projects.

1. In a Project, select the References view.
2. In the References tree, select the required project reference.
3. Click Show Files. The Project displays files and folders from the referenced project.



4. Right-click a file, and then, from the context menu, select the required action.

See Also

Related Examples

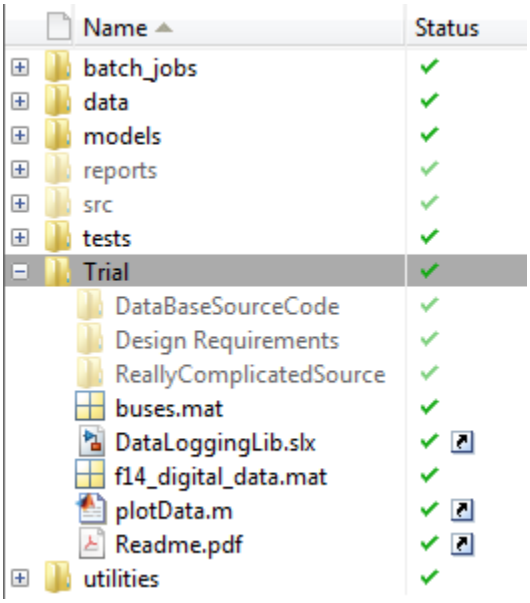
- “Create Shortcuts to Frequent Tasks” on page 16-29
- “Componentization Using Referenced Projects” on page 20-2
- “Add or Remove a Reference to Another Project” on page 20-4
- “Extract a Folder to Create a Referenced Project” on page 20-6
- “Referencing Projects from Another Project”

Extract a Folder to Create a Referenced Project

In a Project, you can partition a large project into components through the use of project references.

Consider the Airframe example project. Suppose you create a folder `Trial` and carry out development work within the folder. You produce:

- Shortcuts to a Simulink library, a MATLAB file, and a `Readme` document
- Design and source code folders
- Data files



For easier management, you want to convert the `Trial` folder into a separate component. In addition, you want access to the folder contents, for example, shortcuts to key files. To fulfill these requirements, extract the folder from the project and convert the folder into a referenced project.

1. In the Files view, right-click the `Trial` folder and select **Extract to Referenced Project**.
2. In the Extract Folder to New Project dialog box, specify these options:
   - **New Project Name** — For example, `DataLogging`.
   - **New Project Location** – For example, `C:\Work\DataLogging`.
   - **Reference type** – The default is `Relative reference`. Use the default if you specify the new project location with reference to the current project root. If you specify the full path for the new location, which is, for example, on a network drive, select `Absolute reference`.
3. Click **More Options**. If you want to disable any of the default content migration actions, clear the corresponding check box.
4. Click **Extract**.
5. In the two Warning dialog boxes that open, click **OK**.

The folder `Trial` and its contents are removed from the project. On the **Project Shortcuts** tab, the **Referenced Projects** section displays a new `DataLogging` button.

See Also

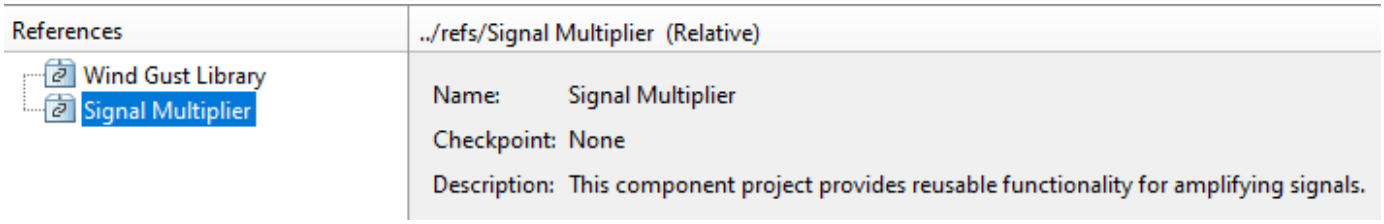
Related Examples

- “Componentization Using Referenced Projects” on page 20-2
- “Add or Remove a Reference to Another Project” on page 20-4
- “View, Edit, or Run Referenced Project Files” on page 20-5
- “Referencing Projects from Another Project”

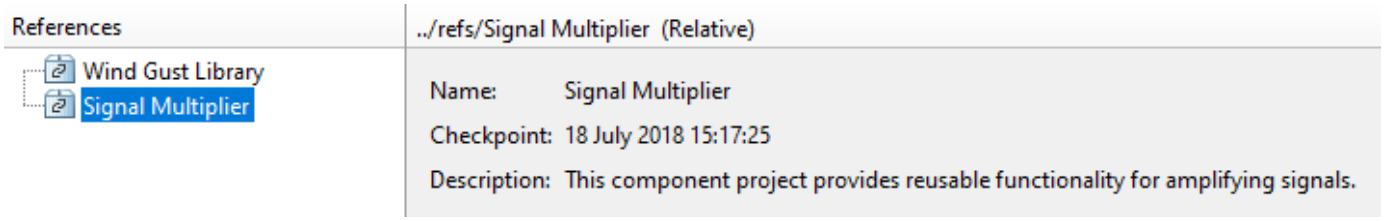
Manage Referenced Project Changes Using Checkpoints

In a Project, you can create a checkpoint for a referenced project. You can then compare the referenced project against the checkpoint to detect changes.

1. In the project that contains the referenced project, select the References view.
2. In the References tree, select the referenced project. If a checkpoint does not exist for the project, in the Details view, the **Checkpoint** field displays `None`.



3. To create a checkpoint, in the Checkpoint section, click Update. In the Details view, the Checkpoint field displays the timestamp of the check point.



In future, to detect changes in the referenced project, in the Checkpoint section, click Checkpoint Report. The Difference to Checkpoint dialog box shows files that have changed on disk since you created the checkpoint.

To remove the checkpoint, in the Checkpoint section, click Clear.

See Also

Related Examples

- “Componentization Using Referenced Projects” on page 20-2
- “Referencing Projects from Another Project”

Compare Simulink Models

About Simulink Model Comparison

In this section...

Creating Model Comparison Reports

In Simulink, you can compare Simulink models. Review and merge differences using three-way model merge or two-way model merge.

You can use models from any version of Simulink. Use the comparison report to explore the differences, view the changes highlighted in the original models, and merge differences.

For details, see “Compare Simulink Models” on page 21-6 and “Merge Simulink Models from the Comparison Report” on page 21-16.

You can access the comparison tool from:

- The MATLAB Current Folder browser context menu
- The MATLAB Comparison Tool
- The MATLAB command line
- The Simulink Editor **Compare** menu
- The Project view

You can use the comparison tool with both model file formats, SLX and MDL. If the selected files are `.mdl` files, or SLX files saved in a previous version, then the comparison tool first exports the `.mdl` files to SLX files in a temporary folder, and produces a comparison report based on the SLX files.

For more information on creating reports, see “Select Simulink Models to Compare” on page 21-3.

Examples of Model Comparison

For examples with instructions, see:

- “Compare and Merge Simulink Models”
- “Compare and Merge Simulink Models Containing Stateflow”
- Resolve Conflicts with Simulink Three-Way Merge

For more information on using and understanding the comparison reports, see “Compare Simulink Models” on page 21-6.

Using Model Comparison Reports

You can display comparison reports in the Comparison Tool. In the interactive report, you can click items in the report to display the corresponding items highlighted in the original models.

The comparison report shows a hierarchical view of the portions of the two files that differ. The report does not show sections of the files that are identical.

If the files are identical you see a message reporting there are no differences.

If files have not been saved, you see an error message informing you that you must save modified or newly created models before running a comparison.

Note It might not be possible for the analysis to detect matches between previously corresponding sections of files that have diverged too much.

Change detection is based on a scoring algorithm. Items match if their score is above a threshold. The tool's algorithm uses a comparison pattern that defines the thresholds assigned to particular node types (e.g., block).

For more information on using the report, see “Compare Simulink Models” on page 21-6.

To control highlighting, see “Display Differences in Original Models” on page 21-14.

To merge differences, see “Merge Simulink Models from the Comparison Report” on page 21-16.

For more information about the Comparison Tool, see “Compare Files and Folders and Merge Files”.

Select Simulink Models to Compare

To learn what you can do with comparison reports, see “About Simulink Model Comparison” on page 21-2.

Select Files from the Simulink Editor

To compare files using the Simulink Editor:

1. On the **Modeling** tab, in the **Evaluate & Manage** section, select **Compare > Compare Models**.

   The Select Files or Folders for Comparison dialog box opens.
2. If the Editor currently displays a model, the current model name and path appear automatically selected in the **First file or folder** edit box. Use the browse buttons to locate and select files for the first and second model files.
3. When you click **Compare**, the comparison tool performs the analysis, and displays the resulting report in the Comparison Tool.

Select Files from the Current Folder Browser

To compare two files from the Current Folder browser:

- For two files in the same view, select two files, right-click and select **Compare Selected Files/ Folders**.
- Alternatively, you can browse to select the second file to compare:

  1. Select a file, right-click and select **Compare Against**
  2. Select the second file to compare in the Select Files or Folders for Comparison dialog box.
  3. For models, leave the default **Comparison type**, `Simulink Model Comparison`.
  4. Click **Compare**.

For more information about comparisons of other file types (e.g., text, MAT, or binary) with the Comparison Tool, see “Compare Files and Folders and Merge Files”.

Select Files from a Project

If you have a project using source control, you can create a model comparison report from the Modified Files view of the project. For details, see “Project Management”.

Select Files from the Comparison Tool

To compare files using the Comparison Tool, from the MATLAB Toolstrip, in the **File** section, select the **Compare** button. In the dialog box select files to compare.

Select Files from the Command Line

To compare XML files from the command line, enter

```
visdiff(filename1, filename2)
```

where `filename1` and `filename2` are XML files or Simulink models.

`visdiff` produces a report in the Comparison Tool.

To create an `xmlcomp.Edits` object at the command line without opening the Comparison Tool, enter:

```
Edits = slxmlcomp.compare(modelname_A,modelname_B)
```

See “Export Results to the Workspace” on page 21-22 for information about the `xmlcomp.Edits` object.

Choose a Comparison Type

To change comparison type, either create a new comparison from the Comparison Tool, or use the **Compare Against** option from the Current Folder browser. You can change comparison type in the Select Files or Folders for Comparison dialog box. For example, if you want the MATLAB text differences report for XML or model files, change the comparison type to `Text comparison` in the dialog before clicking **Compare**. Alternatively, see the `visdiff` function.

See Also

Related Examples

- “Compare Simulink Models” on page 21-6
- “Display Differences in Original Models” on page 21-14
- “Merge Simulink Models from the Comparison Report” on page 21-16
- “Compare Project or Model Templates” on page 21-26

Compare Simulink Models

In this section...

Navigate the Simulink Model Comparison Report

You can compare models from any version of Simulink. The comparison tool produces a comparison report based on the SLX files, resaved in the current version if necessary. Use the report to explore the differences, view the changes highlighted in the original models, and merge differences.

The Comparison report shows changes only, not the entire file contents. The report shows a hierarchical view of the portions of the files that differ, and does not show sections of the files that are identical. To learn about the report, see “About Simulink Model Comparison” on page 21-2.

To *step through differences*, on the **Comparison** tab, in the **Navigate** section, click **Next** or **Previous**. See “Step Through Changes” on page 21-7.

You can also click to select items in the hierarchical trees and observe the following display features:

- Selected items appear highlighted in a box.
- If the selected item is part of a matched pair it is highlighted in a box in both left and right trees.
- When you select an item, the original model displays and the corresponding item is highlighted. See “Explore Changes in the Original Models” on page 21-8.

Report item highlighting indicates the nature of each difference as follows:

Type of report item	Highlighting	Notes
Modified	Purple	Modified items are matched pairs that differ between the two files. When you select a modified item it is highlighted in a box in both trees. Changed parameters for the selected pair are displayed underneath.
Inserted	Blue	When you select an unmatched item it is highlighted in a box in one tree only.
Deleted	Yellow	

Type of report item	Highlighting	Notes
Container	None	Rows with no highlighting indicate a container item that contains other modified or unmatched items.

Icons indicate the category of item, for example: model, subsystem, Stateflow machine or chart, block, line, parameter, etc.

To expand or filter the tree view, use the toolstrip for the following functions:

- **Filter** — Use filters to show only the changes you are interested in. By default the report hides all nonfunctional changes, such as repositioning of items. Turn off filters to explore *all* differences including nonfunctional changes. See “Filter Comparison Reports” on page 21-9.
- **Find** — Opens the Find dialog box where you can search for items.
- If you want to swap the files, on the **Comparison** tab, select **Swap**. The report swaps the sides and reruns the comparison. **Refresh** also runs the analysis again.

To create a new report, see “Select Simulink Models to Compare” on page 21-3.

For examples with instructions, see also “Examples of Model Comparison” on page 21-2.

Step Through Changes

On the **Comparison** tab, in the **Navigate** section, when you click the **Next** arrow button (or press the Down key when the report has focus), you step through groups of changes in the report, in the following order:

1. The first time you click **Next**, it selects the first changed (purple) or inserted (blue) node.
2. Step through the differences with the **Next** button.
   - When selected items have a match in the right tree then they are also highlighted.
   - Next skips white nodes with no color background, if they have no parameter changes underneath. White nodes are parts of the hierarchy that contain no differences.
   - If there is an insertion or deletion with child nodes, **Next** skips the child nodes if they are all also insertions or deletions. For example, if you insert a subsystem, **Next** selects the top subsystem node, then skips all the nodes inside the subsystem (if they are all also insertions) and selects the next difference.
   - **Next** minimizes context switching when highlighting in models. When you click **Next**, the report steps through all differences at the same level of the model, subsystem, or chart, in both left and right trees in the report, before moving to the next level of the report. For example, you step through all differences in a subsystem in the left and right trees, before moving to another subsystem.
3. When you have stepped through all changes, **Next** stops at the end.

If you click an item in the report, the **Next/Previous** controls will step through changes from the point you selected.

Explore Changes in the Original Models

When you compare Simulink models, you can choose to display the corresponding items in the original models when you select report items. You can use this highlighting function to explore the changes in the original models. When you select an item, the report highlights the corresponding item in the model.

Control the display by using the **Highlight Now** button and the **Always Highlight** check box.

For details, see “Display Differences in Original Models” on page 21-14.

Merge Differences

To merge, on the **Comparison** tab, click **Merge Mode**. The Target pane appears at the bottom of the report. Use the buttons to select differences to keep in the target. For more information, see “Merge Simulink Models from the Comparison Report” on page 21-16.

Open Child Comparison Reports for Selected Nodes

If additional comparisons are available for particular parameters, you see a **Compare** button to open a report for that pair of nodes. For example, if there are differences in the Model Workspace, you can click **Compare** to open a new report to explore differences in variables.

- You can open a new comparison for parameters when the report cannot display all the details, e.g., long strings or a script.
- If the original models contain MATLAB Function block components, and if differences are found, click the **Compare** button at the end of the MATLAB Function block report items to open new comparisons in the Comparison Tool, showing the text difference reports for the MATLAB Function block components. You can merge differences in MATLAB Function block code from the text comparison report. See “Merge Simulink Models from the Comparison Report” on page 21-16.
- If the original models contain truth tables, and if differences are found:
  - Click the **Compare** button at the end of the MATLAB Function node to see a summary of all changes.
  - Click the `truthtable` node to reverse annotate and display both truth table editors.
  - Click the **Compare** button on the parameter to open a new text comparison showing only Condition table differences.
  - Similarly click the **Compare** button for `Action Table` to view only Action changes.

Understand the Report Hierarchy and Matching

Note It might not be possible for the analysis to detect matches between previously corresponding sections of files that have diverged too much.

If you cannot see changes you expected to see in the report, turn off filters and see *all* identified changes. See “Filter Comparison Reports” on page 21-9.

Filter Comparison Reports

You can define custom filters to simplify reports and focus on specified elements. You can import and export filters for sharing. Use built-in filters to control display of categories of changes. Turn off filtering to view all identified changes.

To see the available filters, and whether or not they apply to the current report, on the **Comparison** tab, in the **Filter** section, click the down arrow to expand the filter gallery. Click filter names to toggle whether they are applied. In the **Filter** section, click **Show** to include the selected filters changes in the report or **Hide** to exclude the selected filters changes from the report.

Use the filters to include only the changes you are interested in. By default the report hides all nonfunctional changes. These changes have no impact on the design behavior, such as repositioning of items. Turn off filters to explore all differences including nonfunctional changes. Try this if you cannot see changes you expected to see in the report.

Built-in filters include:

- **Lines**. Select all changes to signal lines, including functional changes.
- **Nonfunctional Changes**. The report identifies certain items in the model file as nonfunctional, for example, items representing parameters such as block, system, chart, or label positions; font and color settings for blocks and lines; and system print and display settings.
- **Block Defaults**. Block defaults rarely change and cause longer reports when there are added or deleted blocks. Often the report is simpler when you hide block defaults.

To show all changes, use either of these methods:

- Hide nothing – Click **Hide** and disable all filters in the gallery.
- Show everything – Create a custom filter. In the New Filter dialog box, delete the contents of the first column so it shows `Any`, then remove the rest of the row under **Parameter Name**. Click **Show** and enable the "everything" filter in the gallery.

To define a new custom filter:

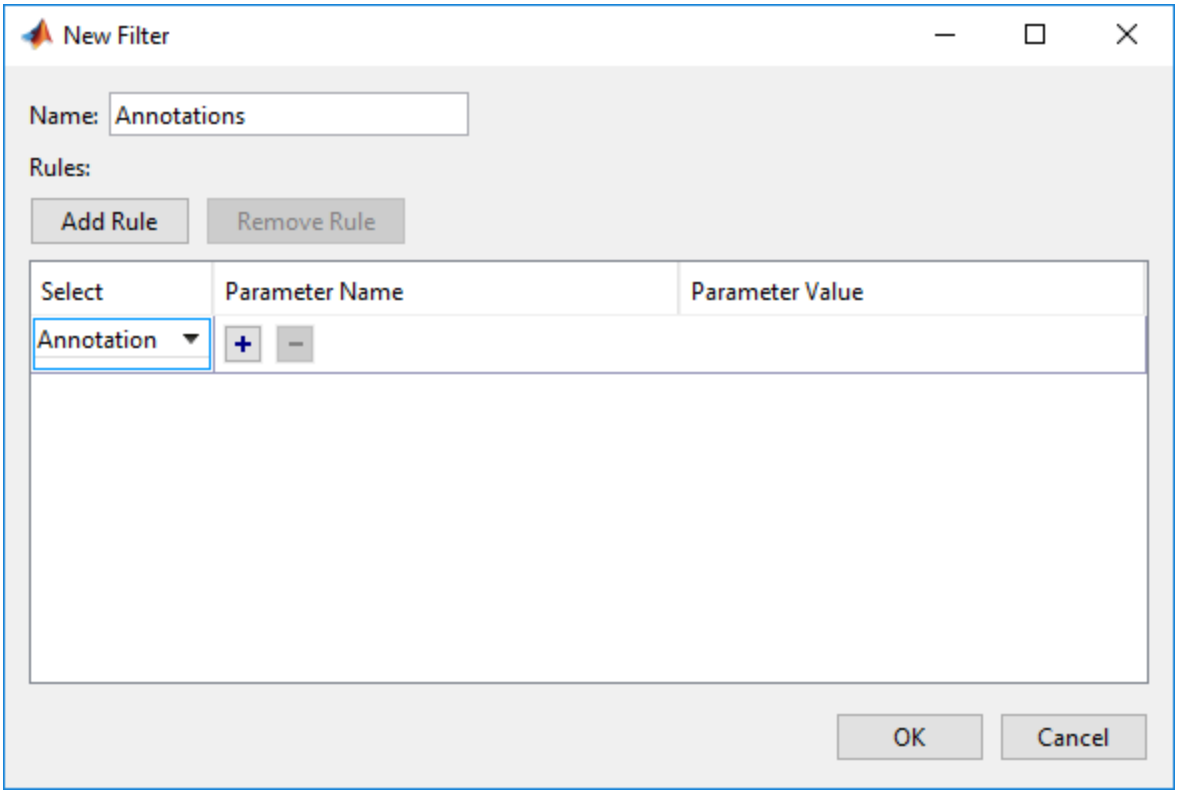
1. On the **Comparison** tab, in the **Filter** section, click **New Filter**.
2. In the New Filter dialog box, define one or more rules for your new filter. For example, you can select parameters or blocks of particular types or values. Suggested values depend on the items in your comparison report. Specify a unique name for your filter and click **Apply**.
3. To check if your filter applies to the items you expect, enable only the new filter in the filter gallery and then click **Show**. This is often easier then checking that a filter is hiding the changes you expect to see in the report.

Observe how these custom filters are defined.

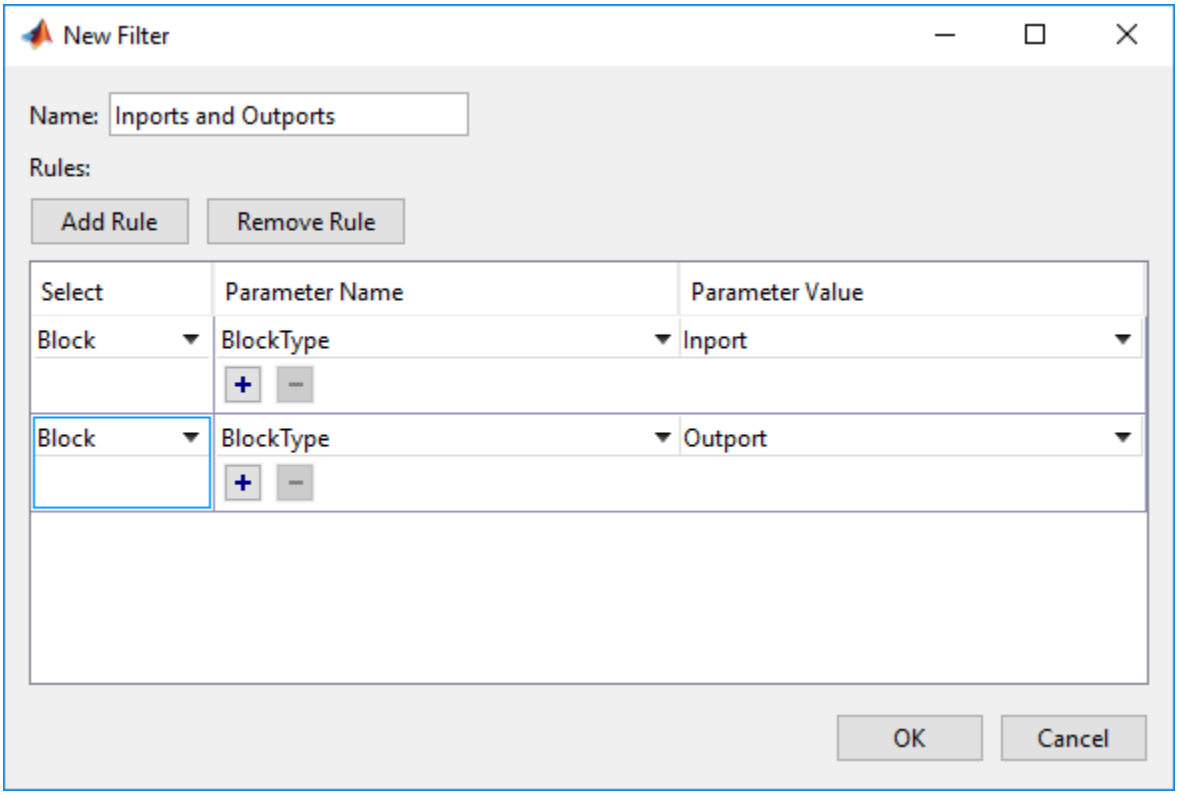
Purpose	Select Column	Parameter Name	Parameter Value
Filter out Annotation changes	`Annotation`	Click the minus button to clear the row.	Leave blank
Filter out Inport and Outport block changes	`Block`	`BlockType`	`Inport`
	To add a row, click **Add Rule**, then select `Block`	`BlockType`	`Outport`

Purpose	Select Column	Parameter Name	Parameter Value
Filter out sample time parameter changes	`Parameter`	`SampleTime`	`Any`

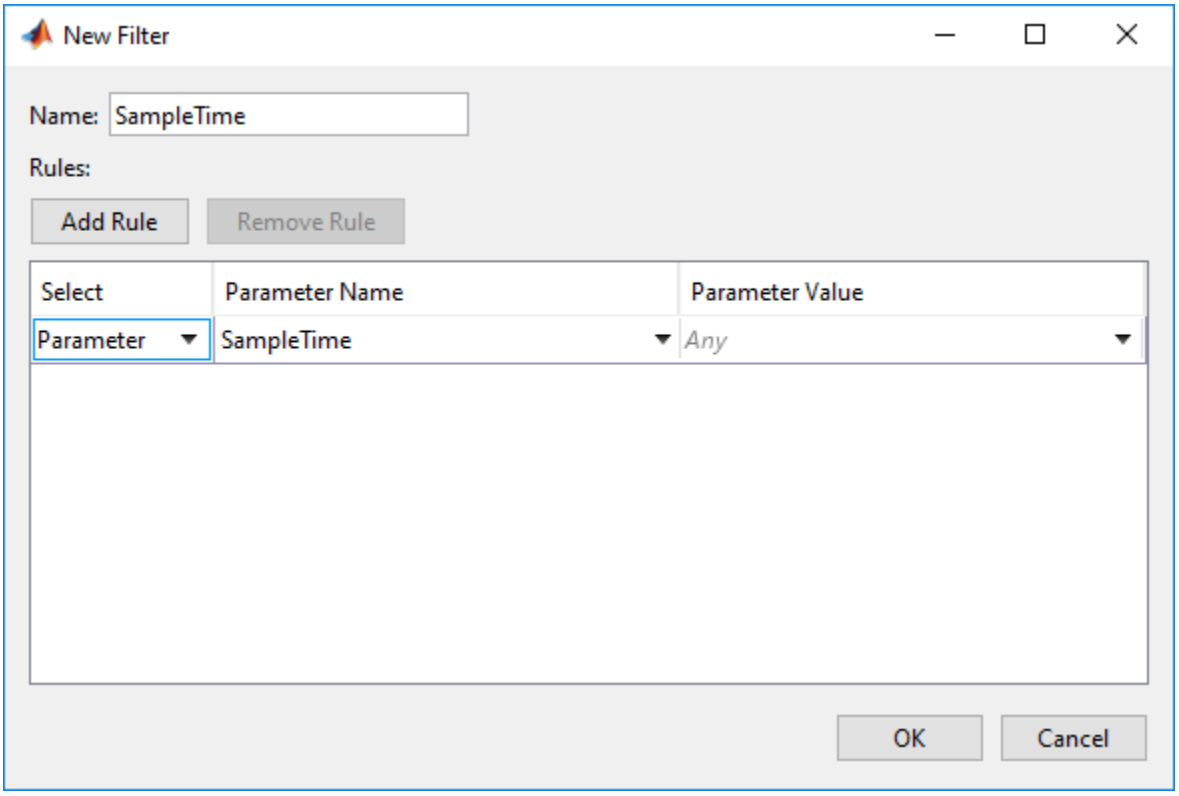
- Annotation changes:



- Inport and Outport block changes:



- Sample time parameter changes:



Exceptions

The report does *not* filter out changes to Block and System names, annotations, and Stateflow Notes as nonfunctional, even though changes to these items do not affect the outcome of simulation. The report always displays these changes to facilitate review of code changes, because they can contain important information about users' intentions.

In rare cases, the report filters out changes that can impact the behavior of the design. By default, moves are filtered as nonfunctional, but in these cases moves can change design behavior:

- Moving blocks can in some cases change the execution order.
- In a Stateflow chart, if you move states or junctions so that they intersect, the model fails to simulate.

To view these types of changes in the report, turn off the filter for nonfunctional changes.

Change Color Preferences

You can change and save your diff color preferences for the Comparison tool. You can apply your color preferences to all comparison types.

1. On the MATLAB Home tab, click **Preferences**.
2. In the Preferences dialog box, under **MATLAB**, click **Comparison**.
3. Edit color settings as desired for differences and merges. View the colors in the **Sample** pane.

   The **Active Settings** list displays **Default (modified)**.
4. To use your modified settings in the comparison, click **Apply** and refresh the comparison report.
5. To return to the default color settings, in the Preferences dialog box, click **Reset** and click **Apply**. Refresh the comparison report.
6. If you want to save your modified color preferences for use in future MATLAB sessions, click **Save As**. Enter a name for your color settings profile and click **OK**.

   After saving settings, you can select them in the **Active Settings** list.

Save Comparison Results

To save your comparison results, use these **Comparison** tab buttons:

- **Publish > HTML**, **Word**, or **PDF** — Open the Save dialog box, where you can choose to save a printable version of the comparison report. See “Save Printable Report” on page 21-22.
- **Publish > Workspace Variable** — Export comparison results to workspace. See “Export Results to the Workspace” on page 21-22.

Alternatively, you can publish a comparison report to a file using the `visdiff` function.

Examples of Model Comparison

For examples with instructions, see:

- “Compare and Merge Simulink Models”

- “Compare and Merge Simulink Models Containing Stateflow”
- Resolve Conflicts with Simulink Three-Way Merge

See Also

`visdiff`

Related Examples

- “Select Simulink Models to Compare” on page 21-3
- “Display Differences in Original Models” on page 21-14
- “Merge Simulink Models from the Comparison Report” on page 21-16
- “Compare Revisions” on page 19-39
- “Source Control in Projects”

More About

- “About Simulink Model Comparison” on page 21-2
- “Comparing Models with Identical Names” on page 21-24
- “Work with Referenced Models and Library Links” on page 21-25

Display Differences in Original Models

In this section...

Highlighting in Models

When you compare Simulink models, you can choose to display the corresponding items in the original models when you select report items. You can use this highlighting to explore the changes in the original models. When you select an item, the report highlights the corresponding item in the model.

Click a report entry to view the highlighted item (or its parent) in the model:

- If the item occurs in both models, they both appear with highlighting.
- When there is no match in the other model, the report highlights the first matched ancestor to show the context of the missing item.
- If the comparison tool cannot highlight an item directly (e.g., configuration parameters), then it highlights the nearest ancestor of the selected node.

Try highlighting items in original models using the example “Compare and Merge Simulink Models Containing Stateflow”.

Control Highlighting in Models

To control highlighting in models, in the Comparison Tool, select or clear the check box **Always Highlight**. You can click the **Highlight Now** button to highlight the currently selected report node at any time. This can be useful if you turn off automatic highlighting and only want to display specific nodes.

By default, models display to the right of the comparison report, with the model corresponding to the left side of the report on top, and the right below. If you move or resize the models your position settings are respected by subsequent model highlighting operations within the same session. The tool remembers your window positions.

If you want to preserve window positions across sessions, position the window, and then enter:

```
slxmlcomp.storeWindowPositions
```

This preserves the placement of any Simulink windows, Stateflow windows, and truth table windows.

To stop storing window positions and return to the defaults, enter:

```
slxmlcomp.clearWindowPositions
```

View Changes in Model Configuration Parameters

You can use the report to explore differences in the model Configuration Parameters. If you select a Configuration Parameter item, the report displays the appropriate root node pane, if possible, of both Configuration Parameters dialog boxes.

Parameters display the label text from the dialog controls (or the parameter name if it is command line only), and the parameter values. You can merge selected parameters using merge mode.

See Also

Related Examples

- “Select Simulink Models to Compare” on page 21-3
- “Compare Simulink Models” on page 21-6
- “Merge Simulink Models from the Comparison Report” on page 21-16
- “Compare Revisions” on page 19-39

More About

- “About Simulink Model Comparison” on page 21-2

Merge Simulink Models from the Comparison Report

In this section...

Merge tools enable you to:

- Resolve conflicts in model files under source control using three-way merge. Open by selecting **View Conflicts**.
- Merge any two model files using two-way merge. Open by selecting **Compare** context menu items.
- Merge MATLAB Function block code using text comparison reports.

Resolve Conflicts Using Three-Way Model Merge

If you have a conflicted model file under source control in a project or in the Current Folder browser, right-click and select **View Conflicts**. You can resolve the conflicts in the Three-Way Model Merge tool. Examine your local file compared to the conflicting revision and the base ancestor file, and decide which changes to keep. You can resolve the conflict and submit your changes.

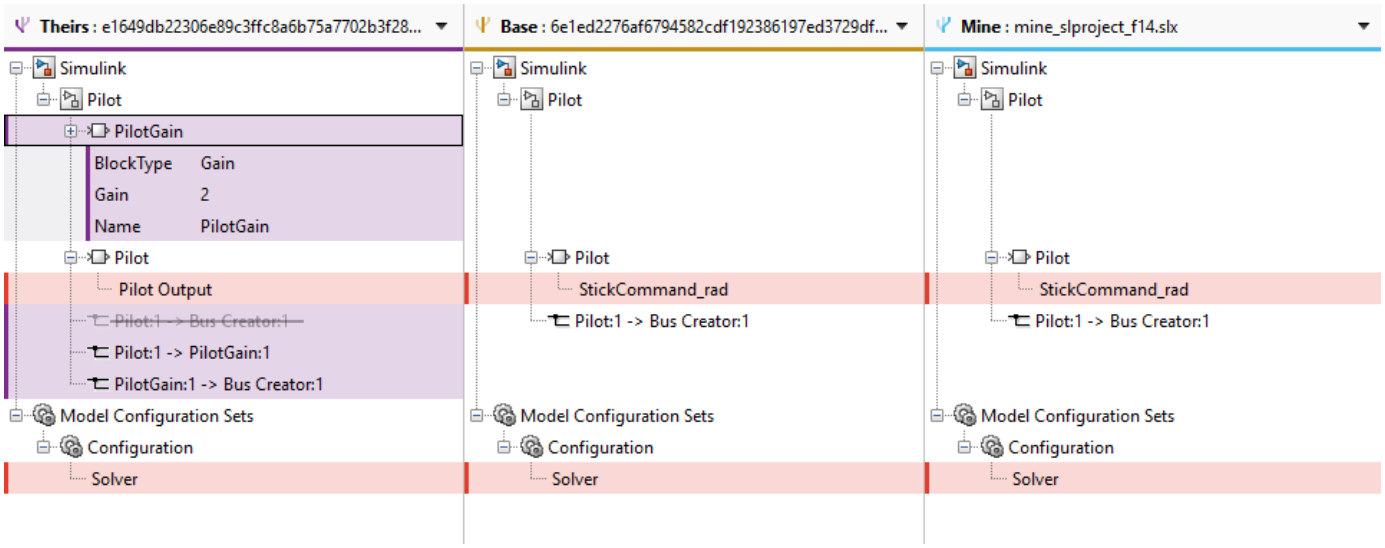
1 To try an example three-way merge, see Resolve Conflicts with Simulink Three-Way Merge.

2 In the project, locate the conflicted model file, right-click and select **View Conflicts**. You can only see **View Conflicts** in the context menu if your file is marked conflicted by the source control.

The Merge tool automatically resolves every difference that it can, and shows the results in the **Target** pane. Review the automerge choices, edit if desired, and decide how to resolve any remaining conflicts.

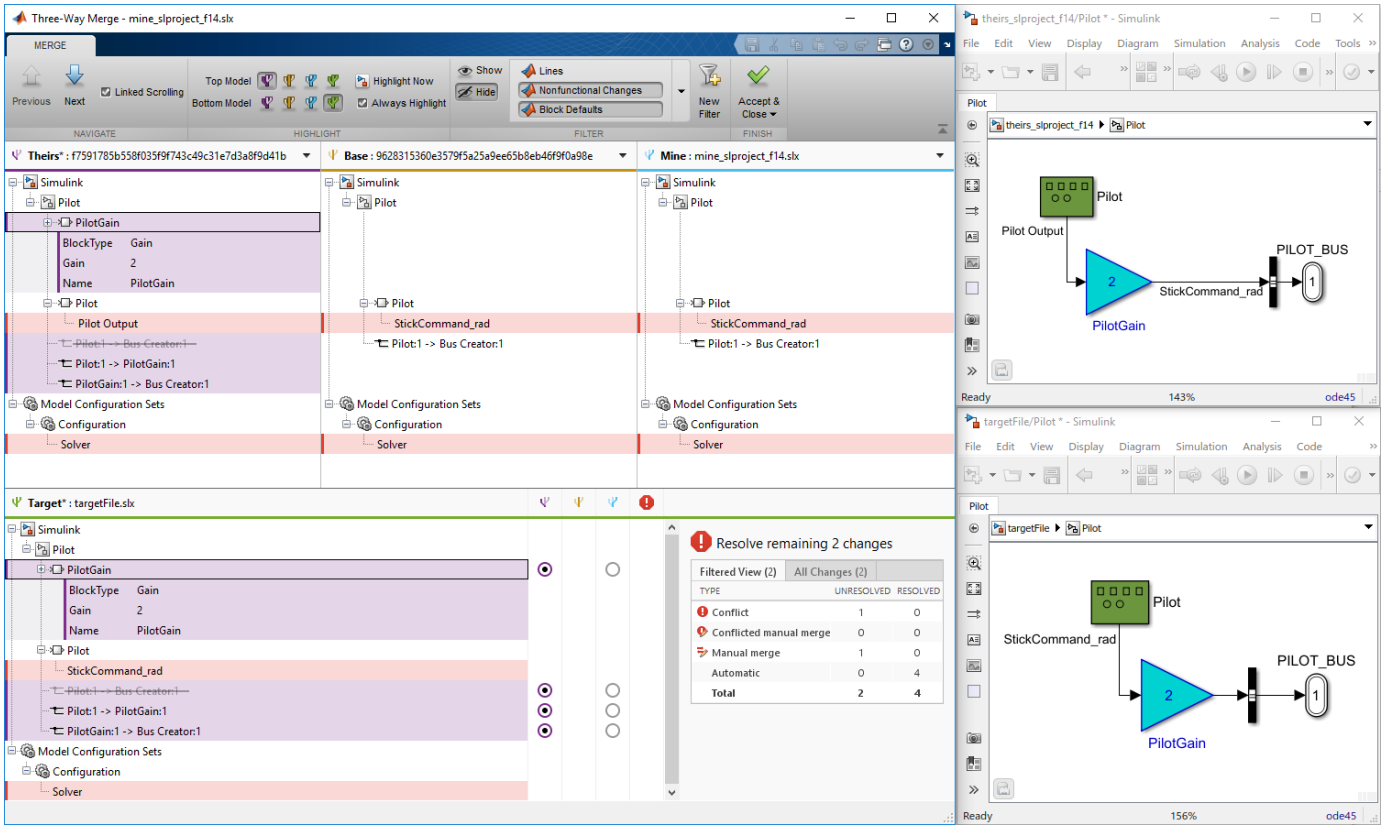
1 Examine the Merge report columns.

- At the top, **Theirs**, **Base**, and **Mine** columns show the differences in the conflicting revision, your revision, and the base ancestor of both files.
- Underneath, the **Target** shows the local file that you will merge changes into. The Merge tool already automerged the differences it can merge.

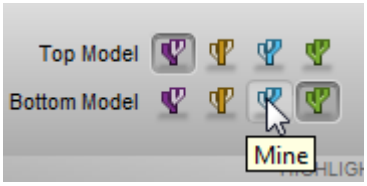


2 Examine a difference by clicking **Next** or by clicking a row in the **Theirs**, **Base**, and **Mine** columns.

The merge tool displays two models (or if you selected a configuration setting, you see two model Configuration Parameters dialog boxes). By default, you see **Theirs** and **Target** models.

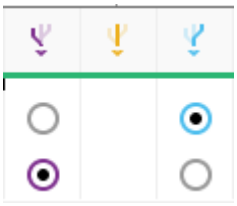


3 Choose the models to display with the toolstrip buttons on the **Merge** tab: **Top Model** or **Bottom Model**. View the models to help you decide what to merge.

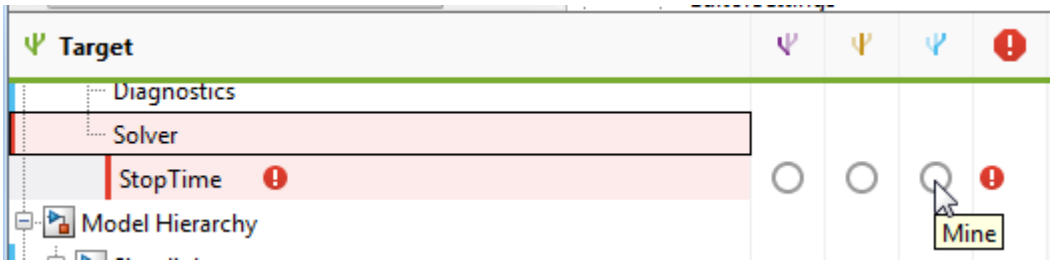


Note If you open the merge tool using **View Conflicts**, then the models **Theirs**, **Base**, and **Mine** are temporary files showing the conflicting revisions. Examine them to decide how to merge. The **Target** model is a copy of **Mine** containing the results of your merges in the report.

4 Select a version to keep for each change by clicking the buttons in the **Target** pane. You can merge modified, added, or deleted nodes, and you can merge individual parameters. The Merge tool selects a choice for every difference it could resolve automatically. Review the selections and change them if you want.

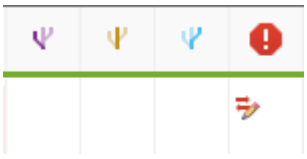


Look for warnings in the Conflicts column. Select a button to use **Theirs**, **Base**, or **Mine** for each conflicted item.



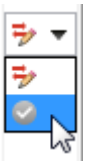
Tip Merge blocks before lines, and merge states and junctions before merging transitions. Merge tool then attempts to connect all lines to blocks for you.

5 Some differences you must merge manually. In the **Target** pane, look for the manual merge icon in the Conflicts column that shows you must take action.

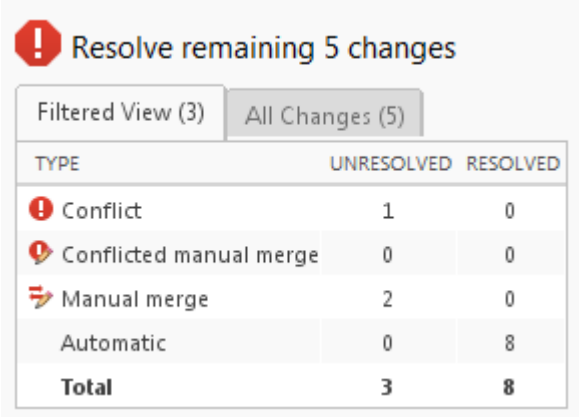


Make manual changes in the Editor. The comparison report cannot update to show any changes that you make in the Editor, so try to make manual changes after addressing all the simpler merges in the report.

After you have resolved the conflict using the Editor, in the **Target** pane, select the check option to mark the node as complete.



6 Examine the summary table to see the number of automatic merges and remaining conflicts you need to resolve.



Check for changes that are filtered out of the current view by looking at the summary table tab titles. The Filtered View and All Changes tab titles show the number of changes. By default the report hides all nonfunctional changes. Turn off active filters to view all identified changes.

7 When you are happy with your merge selections and any manual merges in the **Target** file, click **Accept and Close**. This action saves the target file with all your merges and marks the conflicted file resolved in the source control tool.

To save and not mark the conflict resolved, select **Accept and Close** > **Save and Close**.

To learn more about resolving conflicts in a change list of modified files in a project, see “Resolve Conflicts” on page 19-54.

Use Three-Way Merge with External Source Control Tools

If you are using source control outside of MATLAB, then you can customize external source control tools to open Three-Way Merge (or two-way merge for diffs).

For instructions, see “Customize External Source Control to Use MATLAB for Diff and Merge” on page 19-59.

Open Three-Way Merge Without Using Source Control

If you are not using source control or you want to choose three files to merge, then you can open Three-Way Merge using the function `slxmlcomp.slMerge`. Specify the files to merge, for example:

```
slxmlcomp.slMerge(baseFile, mineFile, theirsFile, targetFile);
```

Three-Way Merge opens, where you can merge the changes in `baseFile`, `mineFile`, and `theirsFile` into the `targetFile`.

Two-Way Model Merge

You can merge two Simulink models from a comparison report. The **Compare** context menu items from a project or the Current Folder browser open a two-way model merge. If you are using source control and want to resolve conflicts using a three-way model merge instead, see “Resolve Conflicts Using Three-Way Model Merge” on page 21-16.

The merge feature enables you to merge two versions of a design modeled in Simulink. You can merge individual parameters, blocks, or entire subsystems. Entire subsystems can only be merged as a whole if they are fully inserted or deleted subsystems.

1. On the **Comparison** tab, click **Merge Mode**. The Target pane appears at the bottom of the report.
2. Use the same workflow as three-way merge. Use the buttons to select the differences to keep in the target file.

   Tip Merge blocks before lines, and merge states and junctions before merging transitions. See “Merging Tips” on page 21-20.
3. View the results in the report and the models. Click **Save File**. **Save File** copies the temporary target file over the right file in the comparison and reruns the comparison.
4. (Optional) To revert all merge operations, click **Close Merge** without saving the file.
5. Inspect your merge changes in the Simulink Editor. If necessary, connect any lines that the software did not connect automatically. The comparison report does not update to show any changes that you make in the Editor.

Merging Tips

- You must merge blocks before lines in the Simulink part of the report. You must merge states and junctions before merging transitions, or the report cannot make the connections.

  For an example showing how to merge a change involving multiple nodes, see “Compare and Merge Simulink Models Containing Stateflow”.
- Not all parameters can be merged. In this case, only one radio button is shown in the target pane indicating the version that is in the target model.
- For information on merging between models with identical names, see “Comparing Models with Identical Names” on page 21-24.

Merge MATLAB Function Block Code

1. To merge differences in MATLAB Function block code, create a comparison report for the parent models.
2. Click the **Merge Mode** button.

   This creates and opens a third file called targetFile. It can contain the changes from either the left or right model.
3. Use the buttons in the right of the report to select changes you want in the target file.
4. Save these changes over the right model using the **Save File** toolstrip button.

See Also

Related Examples

- “Compare Simulink Models” on page 21-6
- “Display Differences in Original Models” on page 21-14
- “Source Control in Projects”
- “Resolve Conflicts” on page 19-54
- “Compare Revisions” on page 19-39
- “Customize External Source Control to Use MATLAB for Diff and Merge” on page 19-59

Export, Print, and Save Model Comparison Results

In this section...

Save Printable Report

To save a printable version of a model comparison report,

1. On the **Comparison** tab, select **Publish** > **HTML**, **Word**, or **PDF**.

   The Save dialog box opens, where you can choose to save a printable version of the model comparison report.
2. Select a file name and location to save the report.

The report is a non-interactive document of the differences detected by the algorithm for printing, sharing, or archiving a record of the comparison. If you have applied filters, your filtered results appear in the printable report.

Alternatively, you can publish a comparison report to a file using the `visdiff` function.

Export Results to the Workspace

To export the comparison results to the MATLAB base workspace,

1. On the **Comparison** tab, select **Publish** > **Workspace Variable**.

   The Input Variable Name dialog box appears.
2. Specify a name for the export object in the dialog and click **OK**. This action exports the results of the model comparison to an `xmlcomp.Edits` object in the workspace.

The `xmlcomp.Edits` object contains information about the comparison including file names, filters applied, and hierarchical nodes that differ between the two files.

To create an `xmlcomp.Edits` object at the command line without opening the Comparison Tool, enter:

```
Edits = slxmlcomp.compare(modelname_A,modelname_B)
```

Property of `xmlcomp.Edits`	Description
`Filters`	Array of filter structure arrays. Each structure has two fields, Name and Value.
`LeftFileName`	File name of left model.
`LeftRoot`	`xmlcomp.Node` object that references the root of the left tree.
`RightFileName`	File name of right model.
`RightRoot`	`xmlcomp.Node` object that references the root of the right tree.

Property of `xmlcomp.Edits`	Description
`TimeSaved`	Time when results exported to the workspace.
`Version`	MathWorks release-specific version number of `xmlcomp.Edits` object.

Property of `xmlcomp.Node`	Description
`Children`	Array of `xmlcomp.Node` references to child nodes, if any.
`Edited`	Boolean — If `Edited = true` then the node is either inserted or part of a modified matched pair.
`Name`	Name of node.
`Parameters`	Array of parameter structure arrays. Each structure has two fields, Name and Value.
`Parent`	`xmlcomp.Node` reference to parent node, if any.
`Partner`	If matched, `Partner` is an `xmlcomp.Node` reference to the matched partner node in the other tree. Otherwise empty `[]`.

See Also

`visdiff`

Comparing Models with Identical Names

You can compare model files of the same name. To complete the operation, the comparison tool copies one of the models to a temporary folder, because Simulink cannot have two models of the same name in memory at the same time. The comparison tool creates a read-only copy of one model named *modelname*_TEMPORARY_COPY, and compares the resulting XML files.

Warning When you use highlighting from the report, one of the models displayed is a temporary copy with a new name. The temporary copy is read-only, to avoid making changes that can be lost.

Alternatively, you can run the comparison by renaming or copying one of the files.

If one of the models is open when you try to compare them, a dialog box appears where you can click **Yes** to close the file and proceed, or **No** to abort. You must close open models before the comparison tool can compare two models with the same name. The problem requiring you to close the loaded model is called “shadowed files”. In some cases, another model with the same name might be in memory, but not visible. See “Shadowed Files” on page 15-3 for more information.

If you want to automatically close open models of the same name when comparing them and not see the dialog box again, run these commands:

```
opt = slxmlcomp.options
opt.setCloseSameNameModel(true)
```

This is persistent across MATLAB sessions. To revert to default behavior and be prompted whether or not to close the open model every time, enter:

```
opt = slxmlcomp.options
opt.setCloseSameNameModel(false)
```

If you open a comparison report from a project (for example, using **Compare to Revision**), the project handles files of the same name and does not prompt you to close models.

Work with Referenced Models and Library Links

The model comparison report applies only to the currently selected models, and does not include changes to any referenced models or linked libraries. The comparison report shows only changes in the files selected for comparison.

Tip If you want to examine your whole hierarchy instead, try using a project, where you can examine modified files and dependencies across your whole project, and compare to selected revisions. See “Project Management”.

If you are comparing models that contain referenced models with the same name, then your MATLAB path can affect the results. For example, this can happen if you generate a model comparison report for the current version of your model and a previous baseline. Make sure that your referenced models are not on your MATLAB path before you generate the report.

The reason why results can change is that Simulink records information in the top model about the interface between the top model and the child model. This interface information in the top model enables incremental loading and diagnostic checks without any need to load child models.

When you load a model (for example, to compare) then Simulink refreshes the interface information for referenced models if it can find the child model. Simulink can locate the child model if it is on the path. If another model of the same name is higher on the path, Simulink updates the interface information for that other model before comparing. This can produce entries for interface changes for model reference blocks in the comparison report. Make sure your referenced models are not on your path before you generate the report, to avoid these interface changes in the results. If both model versions are off the path, the interface information in the top model is not refreshed during the comparison process. Instead the cached information is used, resulting in a valid comparison report.

With library links, Simulink does not update the cached interface information when comparing, and so the report correctly captures library interfaces. However with both referenced models and library links, Simulink updates the information when displaying the model. When displaying report items in original models, you may see that Simulink finds another model or library that is higher in the path. To obtain the clearest results, make sure that the models and associated libraries are temporarily removed from the path. By removing the files from the path you will see unresolved library links and referenced models when you view the original models, but their interfaces will be correct and will correctly align with the comparison report.

Compare Project or Model Templates

In this section...

Compare Project Templates

You can compare project templates (SLTX files). If you select two project template files to compare, you see a comparison report showing differences in template properties and project metadata. You can open a new report to investigate project file and folder differences.

- Click **Template Properties** to view differences in the Parameters, such as the description or date modified.
- Expand the **Project Metadata** node to view metadata differences such as label changes.
- Next to **Project Files**, click **Compare** to open a folder comparison where you can investigate changed, added, or removed files and folders.

Compare Model Templates

You can compare model templates (SLTX files). If you select two model template files to compare, you see a comparison report showing differences in template properties. You can open a new comparison report to compare the Simulink models.

- Click **Template Properties** to view differences in the Parameters, such as the description or date modified.
- Next to **Model**, click **Compare** to open a Simulink model comparison report where you can investigate differences.

See Also

Related Examples

- “Compare Simulink Models” on page 21-6
- “Compare Folders and Zip Files”
- “Create Templates for Standard Project Settings” on page 16-32
- “Create a Template from a Model” on page 4-2

Large-Scale Modeling

Component-Based Modeling Guidelines

Componentization benefits organizations developing Simulink models that consist of many functional pieces. Using model components can enable:

- Team-based development — Reduce file contention and elaborate components independently through well-defined interfaces.
- Reduced design complexity — Each component solves smaller problems.
- Component reuse — Reuse algorithms and environment models within a project and across multiple projects.
- Unit testing — Eliminate retesting for unchanged components and reduce the cost of verification.
- Performance benefits that scale — Reduce memory usage and the time required to load and simulate models.
- Component variants — Choose among multiple implementations of a component.
- Intellectual property protection — Limit functionality and content visibility for components that you share with third parties.

Should You Create Model Components?

Considering the work required to define and manage components, you should use component-based modeling only when the benefits outweigh the cost.

Separating an existing Simulink model into components is analogous to taking a large piece of code (C, Java, or MATLAB code) and breaking it down into multiple functions. The conversion can require significant effort and extensive modifications if the design is not modular from the beginning.

Considering model scalability and potential requirements upfront makes separating a Simulink model into components easier. Identifying components upfront can help you avoid these difficulties:

- Poor component definition — The scope of subsystems that are grown over time can fail to meet component requirements. For example, they might contain too much or too little functionality to be reused, to generate code that integrates with legacy functionality, or to support hardware-in-the-loop tests.
- Merge conflicts — If additional engineers begin to work on a model that was originally designed for development by a single engineer, they can encounter time-consuming and error-prone merges.
- Algebraic loops — If a single engineer develops a model from the bottom up, they are likely to group blocks into subsystems as model complexity increases. The subsystems within the model are likely visual groupings that do not affect model execution. When you make these subsystems atomic, or convert them to referenced models, you can introduce unwanted algebraic loops that are difficult to diagnose and fix.

Components are also useful when a design becomes too complicated for one person to manage all of the details. For example, a complicated model can be a model that has:

- Thousands of blocks
- Hundreds of logical decisions
- Multiple variant configurations of the same functionality

Projects and source control can help you manage components. For more information, see “What Are Projects?” on page 16-3 and “Configuration Management” on page 22-21.

Define Model Components

1. “Choose Among Types of Model Components” on page 22-4	Identify Simulink components that align with your high-level modeling requirements.
2. “Compare Capabilities of Model Components” on page 22-8	Investigate which types of model components meet your low-level modeling requirements.
3. “Define Interfaces of Model Components” on page 22-17	Configure signal attributes at interfaces and manage data for model components.

Choose Among Types of Model Components

Useful model components have a well-defined scope, perform functionality defined by requirements, and form part of a larger system.

As you define a component, consider these potential requirements.

- File contention — You can have larger components if only one person is working on each. If you must share components between several people, you should divide the design into smaller logical pieces. If multiple people must edit the same file, see “Merge Simulink Models from the Comparison Report” on page 21-16.
- Reusability — If you expect to use a group of blocks multiple times in a model, define the group of blocks in a reusable component. By avoiding duplication, you make maintaining the model easier. To refactor an existing model with duplication, see “Refactor Models to Improve Component Reuse” (Simulink Check).
- Code generation — If you must generate standalone code for a physical component, such as a digital controller, you should have one component that represents the physical component and has a well-defined interface.
- Verification cost — If part of a model changes frequently and has high testing costs, you should manage this part of the model as a component in a separate file. When components are defined in separate files, you can control and trace changes using project source control. For more information on source control, see “Configuration Management” on page 22-21.
- Simulation speed — Using different solvers for components with different numerical properties can increase simulation speed. Similarly, grouping blocks based on their sample rate can increase simulation speed. For more information, see Solver Profiler and “Improve Simulation Performance Using Performance Advisor” on page 32-2.

Modeling requirements can influence the size of your components. For example, models with fewer than 500 blocks are easier to test than larger models. However, simulation can be faster for model hierarchies when referenced models contain more than 500 blocks.

Simulink Components

The different types of Simulink components serve a variety of modeling requirements.

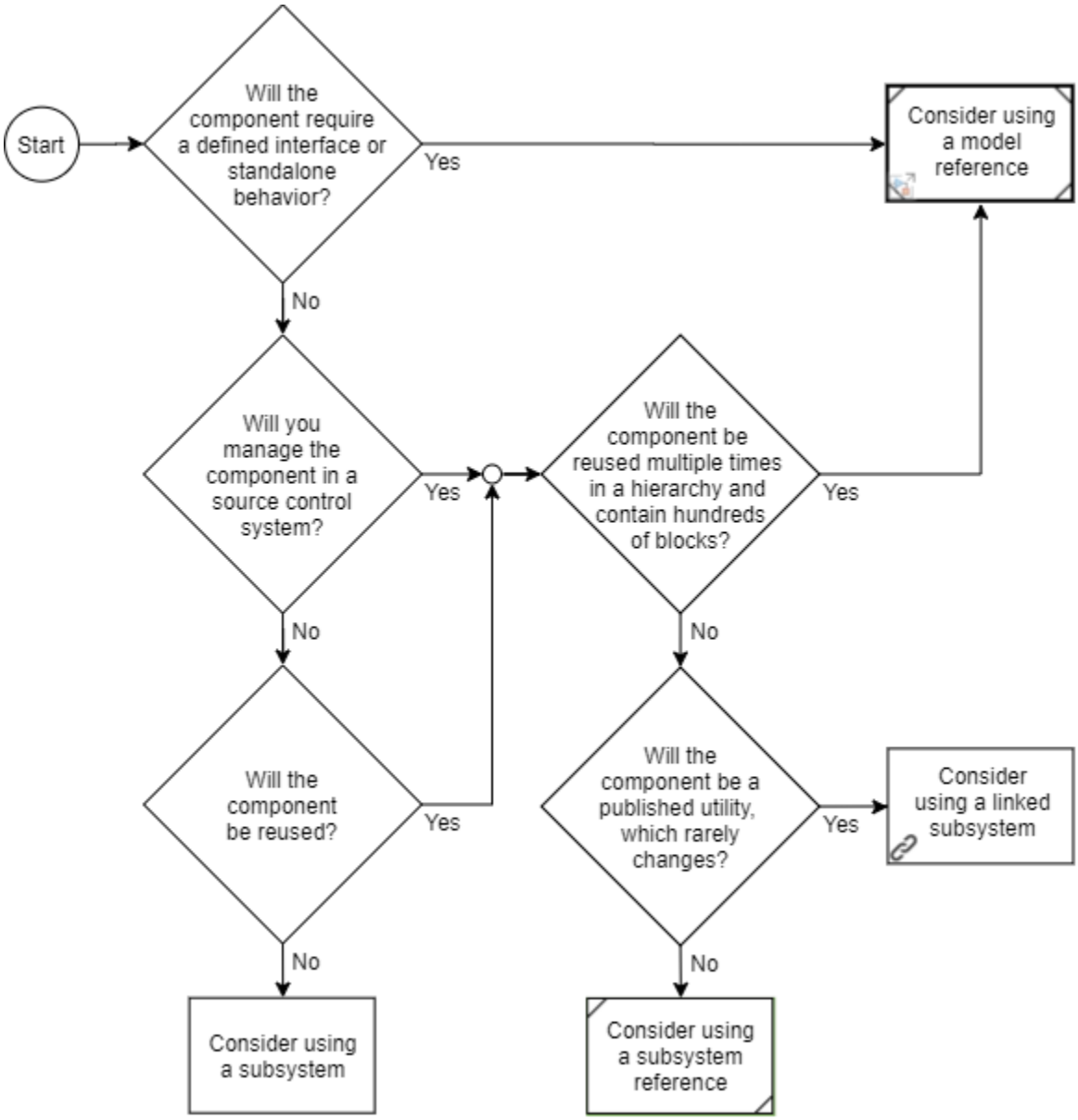
Type of Component	Definition	Source of Contents	Implementation in Model
Subsystem	Unique group of blocks with a dynamic interface, which can be visual or functional.	None — Contents must be manually added to each subsystem	Subsystem block
Subsystem reference	Reference to a reusable group of blocks with a dynamic interface, which can be visual or functional.	Subsystem file (`.slx`) that contains the referenced subsystem	Subsystem Reference block
Model reference	Reference to a model with a well-defined interface, which is functional and independent of the parent model.	Model file (`.slx`) that contains the referenced model	Model block

Type of Component	Definition	Source of Contents	Implementation in Model
Variant system	Multiple implementations of a component with only one active implementation. Variant systems allow you to address different sets of requirements within a single model. Variant choices can be any other component type, including a combination of component types.	None — Variant choices must be manually added to each variant system	Variant Subsystem block
Linked block, which can be linked to any component that is stored in a library	Linked instance of block that is stored in a library. If you disable the library link, each instance of a linked block can be unique. When you drag a subsystem reference or model reference from a library into a model, it directly references the subsystem file or model file that defines its contents. It has a library link only when the parent library block has a mask applied directly to it. Typically, you should use model masks, which are saved in the referenced file and do not require a library link.	Library file (`.slx`) that contains the parent library block, or prototype block	Block with a library link

Simulink models can use any combination of these components. For example, to minimize file contention for a large model, you can convert subsystems to referenced subsystems and models, both of which are saved in separate files.

High-Level Component Selection Guidelines

This flow chart provides a starting point for choosing a component type.



Before implementing a component based on the result from this flow chart, consider additional modeling requirements. For information on component compatibility with modeling requirements, see “Compare Capabilities of Model Components” on page 22-8.

If you expect a subsystem to grow, make it atomic so that it functionally groups the blocks and executes them together. Functionally grouping the blocks makes it easier to convert the subsystem to a referenced model.

See Also

Model | Subsystem | Variant Subsystem, Variant Model

More About

- “Create Subsystems” on page 4-15
- “Model Reference Basics” on page 8-2
- “Define, Configure, and Activate Variants” on page 12-42
- “Linked Blocks” on page 41-10

Compare Capabilities of Model Components

Before you implement a piece of your system using a specific component type, consider whether the component type satisfies your modeling requirements.

Component Consideration	Modeling Requirements
“Development Process” on page 22-9	• Component reuse • Shared data • Instance-specific edits • Version control and configuration management • Intellectual property protection • Unit testing
“Performance Requirements” on page 22-13	• Incremental model loading • Build artifact reuse • Reduced memory usage for large models • Artificial algebraic loop elimination
“Features” on page 22-14	• Compatible configuration parameter settings • Signal property specification at interfaces • Bus specification • State initialization • Code generation

Development Process

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Component Reuse	**Not supported** While you can copy a subsystem to reuse it in a model, the copies are independent of each other. When you edit a subsystem, the changes apply to the parent model file. To create or change a subsystem, you must open the parent model, which can lead to file contention when multiple people want to work in the model.	**Supported** You save the parent library block of a linked block in a separate file from the model that links to it. Using separate files helps to avoid file contention. You can link to the same parent library block multiple times in multiple models without creating copies. Managing library links adds some overhead, such as managing broken, disabled, or parameterized links.	**Supported** You save a referenced subsystem in a separate file from the model that references it. Using separate files helps to avoid file contention. You can reference the same subsystem multiple times in multiple models without creating copies.	**Supported** You save a referenced model in a separate file from the model that references it. Using separate files helps to avoid file contention. You can reference the same model multiple times in multiple models without creating copies. See “Model Reuse” on page 8-6.
Shared Data	**Supported** You can share data among instances by defining the data outside the component. For example, by using a data store in a common parent subsystem.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You can share data among instances of the referenced model by creating a data store inside the model. See “Share Data Among Referenced Model Instances” on page 8-34.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Instance-Specific Edits	**Supported** Subsystem copies are independent of each other.	**Supported** When you edit a parent library block, the changes apply to the library file and propagate to all blocks that link to that block. To edit an instance of the block, you can disable the library link. You cannot disable library links when the parent library block has restricted write access.	**Not supported** When you edit an instance of a referenced subsystem, the changes apply to the subsystem file and propagate to all other instances of the referenced subsystem.	**Not supported** When you edit an instance of a referenced model, the changes apply to the model file and propagate to all other instances of the referenced model.
Version Control and Configuration Management	**Not supported** You cannot directly place subsystems in a source control system. To reduce file contention and use separate version control for each subsystem, use a subsystem reference.	**Supported** You can place library files in a source control system. To provide individual version control for each library block, use subsystem references and model references in the library. When you drag these blocks from the library into your model, they reference the subsystem file or model file. Forwarding tables allow you to map old library blocks to new versions of the blocks.	**Supported** You can place subsystem files in a source control system.	**Supported** You can place model files in a source control system.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Intellectual Property Protection	**Not supported** Use model references instead.	**Not supported** *Same behavior as subsystems.*	**Not supported** *Same behavior as subsystems.*	**Supported** Protected models obscure model contents, which can be useful when distributing models. Creating a protected model requires a Simulink Coder license. Using a protected model does *not* require a Simulink Coder license.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Unit Testing	**Supported** Subsystems are dependent on their context in a model. If the context of a subsystem changes, such as the data type of an input signal, the related test harness must be updated. For subsystems that are not atomic, the test harness may use different block execution orders, due to virtual boundaries. For tools that support authoring, managing, and executing systematic, simulation-based tests of subsystems, see “Create Test Harnesses and Select Properties” (Simulink Test). To measure how thoroughly model components are tested, see “Model Coverage” (Simulink Coverage).	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You can test a referenced model independently to isolate behavior by simulating it as a top model. You can use a data-defined test harness, with MATLAB test vectors and direct coverage collection. For tools that support authoring, managing, and executing systematic, simulation-based tests of subsystems, see “Create Test Harnesses and Select Properties” (Simulink Test). To measure how thoroughly model components are tested, see “Model Coverage” (Simulink Coverage).

Performance Requirements

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Incremental Model Loading	**Not supported** Loading a model loads all subsystem contents that are saved in the model.	**Supported** Simulink incrementally loads a library at the point needed during editing, updating a diagram, or simulating a model.	**Supported** Simulink incrementally loads a referenced subsystem at the point needed during editing, updating a diagram, or simulating a model.	**Supported** Simulink incrementally loads a referenced model at the point needed during editing, updating a diagram, or simulating a model.
Build Artifact Reuse	**Not supported** Build artifacts, such as simulation targets, are not generated for subsystems.	**Not supported** *Same behavior as subsystems.*	**Not supported** *Same behavior as subsystems.*	**Supported** You can share and reuse build artifacts, such as simulation targets, using Simulink cache files. For more information, see “Share Simulink Cache Files for Faster Simulation” on page 8-54.
Reduced Memory Usage for Large Models	**Not supported** Subsystems do not reduce memory usage for simulation and code generation.	**Not supported** Linked subsystems do not reduce memory usage for simulation and code generation. Simulink duplicates library block instances during block update.	**Not supported** Subsystem references do not reduce memory usage for simulation and code generation. Simulink duplicates subsystem reference instances during block update.	**Supported** Models referenced in accelerator mode reduce memory usage for simulation and code generation because Simulink incrementally loads compiled versions of them.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Artificial Algebraic Loop Elimination	**Supported** Subsystems that are not atomic avoid artificial algebraic loops. If a subsystem is atomic, you can try to eliminate artificial algebraic loops by enabling the Subsystem block parameter **Minimize algebraic loop occurrences**.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You can try to eliminate artificial algebraic loops by enabling **Configuration Parameters > Model Referencing > Minimize algebraic loop occurrences**.

Features

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Compatible Configuration Parameter Settings	**Supported** Subsystems use the configuration parameter settings of the model that contains them.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** Configuration parameter settings can generally be different for a parent model and its referenced models. For compatibility information, see “Set Configuration Parameters for Model Hierarchies” on page 8-60.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
Signal Property Specification at Interfaces	**Supported** You can specify signal properties at a subsystem interface. For signal properties that you do not specify, subsystems inherit the signal properties from their context. Propagation of signal properties can lead to Simulink using signal properties that you do not anticipate.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You must specify most signal properties at a referenced model interface. Referenced models are context-independent with a defined boundary, so they do not inherit most signal properties. Referenced models can inherit discrete sample times when the referenced model is sample-time independent.
Bus Specification	**Supported** You can use a `Simulink.Bus` object to specify the data type of a bus that passes into a subsystem. Subsystems do not require the use of `Bus` objects for virtual buses.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You can use a `Simulink.Bus` object to specify the data type of a bus that passes into a referenced model. Model references do not require the use of `Bus` objects for virtual buses when you use In Bus Element and Out Bus Element blocks.

Modeling Requirement	Subsystems	Linked Subsystems	Subsystem References	Model References
State Initialization	**Supported** You can initialize states of subsystems.	**Supported** *Same behavior as subsystems.*	**Supported** *Same behavior as subsystems.*	**Supported** You can initialize states from the top model using either the structure format or structure-with-time format. For more information, see “State Information for Referenced Models” on page 72-79.
Code Generation	**Supported** For information on subsystem code generation, see “Control Generation of Functions for Subsystems” (Simulink Coder).	**Supported** For information on linked subsystem code generation, see “Control Generation of Functions for Subsystems” (Simulink Coder).	**Supported** *Same behavior as subsystems.*	**Supported** For information on referenced model code generation, see “Generate Code for Model Reference Hierarchy” (Simulink Coder).

See Also

More About

- “Model Reference Requirements and Limitations” on page 8-6
- “Choose Simulation Modes for Model Hierarchies” on page 8-39

External Websites

- 11 Best Practices for Developing ISO 26262 Applications with Simulink

Define Interfaces of Model Components

Defining the interface of a software component, such as a C or MATLAB code function or a Simulink subsystem, is a key first step before others can use it, for these reasons:

- Agreeing on an interface is a useful first step in deciding how to break down the functionality of a large system into subcomponents.
- After you define interfaces between components, you can develop the components in parallel. If the interface remains stable, then it is easy to integrate those components into a larger system.
- Changing the interface between components is expensive. It requires changes to at least two components (the source and any sinks) and to any test harnesses. It also makes all previous versions of those components incompatible with the current and future versions.

When you must change an interface, doing so is much easier if the components are stored under configuration management. You can track configurations of compatible component versions to prevent incompatible combinations of components.

Guidelines for Interface Design

Suggestions for defining the interfaces of components for a new project:

- Base the boundaries of the components upon the boundaries of the corresponding real systems. This guideline is especially useful when the model contains:
  - Both physical (plant and environment) and control systems
  - Algorithms that run at different rates
- Consider future model elaboration. If you intend to add models of sensors, then put them in from the start as an empty subsystem that passes signals straight through or performs a unit delay or name conversion.
- Consider future component reuse.
- Consider using a signal naming convention.
- Use data objects for:
  - Defining component interfaces
  - Precise control over data attributes
- Simplify interface design by grouping signals into buses. Buses are well suited for use at the high levels of models, where components often have many signals going in and out, and do not use all the signals available. Using buses can simplify modifying the interface to a component. For example, if you must add or remove signals used by a component, it can be simpler to modify a bus than to add or remove input or output ports to that component. However, using a bus that crosses model reference boundaries requires using a bus object.

  Best practices for using Simulink buses and bus objects:

  - Make buses virtual, except for at model reference component boundaries.
  - Use nonvirtual buses when defining interfaces between components. A bus object must define each nonvirtual bus. Bus objects completely define the properties of the signals on a bus, giving an unambiguous interface definition.

Include bus objects in a data dictionary, or save bus objects as a `.mat` or `.m` file, in order to place them under revision control.

- Pass only required signals to each component to reduce costly passing of unnecessary data. Signal buses allow the full set of input and output signals to be defined, but not necessarily used or created.
- Make sure that the interface specifies exactly what the component uses.
- Use a rigorous naming convention for bus objects. Unless you use a data dictionary, bus objects are stored in the base workspace.
- At the lower levels of a model, consider using input and output ports for each signal. At lower levels of a model, where components typically implement algorithms rather than serve as containers for other components, it can increase readability if you use individual input and output ports for components, instead of using signal buses. However, creating interfaces in this way has a greater risk of connection problems, because it is difficult to check the validity of connections, other than their data type, size, etc.
- To package signals or parameters into structures that correspond to a `struct` type definition that your external C code defines, import the type as a bus object and use the object as a data type for buses and MATLAB structures. To create the object, use the `Simulink.importExternalCTypes` function.

Partitioning Data

Explicitly control the scope of data for your components. Some techniques:

- Global parameters — A common approach in the automotive world is to completely separate the problem of parameter storage from model storage. The parameters for a model come from a database of calibration data, and the specific calibration file used becomes part of the configuration. The calibration data is treated as global data, and resides in the base MATLAB workspace. You can migrate base workspace data to a data dictionary for more control.
- Nonglobal parameters — Combining components that store their own parameter data has the risk of parameter name collisions. If you do not use a naming convention for parameters or, alternatively, a list of unique parameter names and definitions, then there is the risk that two components use a parameter with the same name but with different meanings.

  Methods for storing local parameter data include:

  - Partition data into reference dictionaries for each component.
  - For referenced models, you can use model workspaces.
  - Use parameter files (`.m` or `.mat`) and callbacks of the individual Simulink models (e.g., `preload` function).

    You can also automatically load required data using project shortcuts.
  - Mask workspaces, with or without the use of mask initialization functions.
  - For subsystems, you can control the scope of data for a subsystem using the Subsystem Parameters, Permit Hierarchical Resolution dialog box.

Configure Data Interface for Component

Whether you use referenced models or subsystems to break a large system into components, the components can exchange signal data through Inport and Outport blocks. You can explicitly configure

design attributes (such as data type and numeric complexity) of the interface to prevent modeling errors and make integrating the components easier.

After you create the Inport and Outport blocks, you can use the Model Data Editor and the interface display to configure the design attributes (such as data type and numeric complexity) of the blocks. Use this technique to view the component interface in its entirety at once and to trace the pieces of the interface to usage points in the internal block algorithm. You can also use this technique to configure the interface of a component before you develop the internal algorithm, in which case the component contains unconnected Inport and Outport blocks.

The example model `sldemo_fuelsys_dd` contains two components which are referenced models:

- A plant component, `sldemo_fuelsys_dd_plant`.
- A controller component, `sldemo_fuelsys_dd_controller`.

Use the Model Data Editor and the interface display to examine and configure the interface of the plant component.

1. Open the plant component.

   ```
   sldemo_fuelsys_dd_plant
   ```

2. On the **Modeling** tab, under **Design**, click **Model Interface**.
3. On the **Modeling** tab, click **Model Data Editor**.

   By default, in the Model Data Editor, the **Inports/Outports** tab is selected. Each row in the table represents an Inport or Outport block. By default, the **Change view** drop-down list is set to `Design`.

   Tip To view only the Inport and Outport blocks at the root level of the model (by excluding the blocks inside the subsystems), deactivate the **Change Scope** button.

4. Use the columns in the Model Data Editor to explicitly configure the design attributes of the interface. For example, specify minimum and maximum values for each Inport and Outport block by using the **Min** and **Max** columns.

To configure code generation settings for the interface of a controller component, in the Model Data Editor, set the **Change view** drop-down list to `Code`.

For more information about using the interface display, see “Trace Connections Using Interface Display” on page 76-110. For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

See Also

More About

- “Types of Composite Signals” on page 76-2
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Trace Connections Using Interface Display” on page 76-110
- “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27

- “Parameter Interfaces for Reusable Components” on page 37-17
- “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder)
- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100

Configuration Management

In this section...

Manage Designs Using Source Control

Projects can help you work with configuration management tools for team collaboration. You can use projects to help you manage all the models and associated files for model-based design.

You can control and trace the changes in each component using project source control. Using source control directly from a project provides these benefits:

- Engineers do not have to remember to use two separate tools, avoiding the common mistake of beginning work in Simulink without checking out the required files first.
- You can perform analysis within MATLAB and Simulink to determine the dependencies of files upon each other. Third-party tools are unlikely to understand such dependencies.
- You can compare revisions and use tools to merge models.

If each component is a single file, you can achieve efficient parallel development, where different engineers can work on the different components of a larger system in parallel. Using model components allows you to avoid or minimize time-consuming merging. One file per component is not strictly necessary to perform configuration management, but it makes parallel development much easier.

If you break down a model into components, it is easier to reuse those components in different projects. If the components are kept under revision control and configuration management, then you can reuse components in multiple projects simultaneously.

To find out about source control support, see “Source Control in Projects”.

Determine the Files Used by a Component

You can use a project to determine the set of files you must place under configuration management. You can analyze the set of files that are required for the model to run, such as model references, library links, block and model callbacks (`preload` functions, `init` functions, etc.), S-functions, From Workspace blocks, etc. Any MATLAB code found is also analyzed to determine additional file dependencies. You can use the Dependency Analyzer to report which toolboxes are required by a model, which can be a useful artifact to store.

You can also perform a file dependency analysis of a model programmatically from MATLAB using `dependencies.fileDependencyAnalysis` to get a cell array of paths to required files.

For more information, see “Dependency Analysis”.

Manage Model Versions

Simulink can help you to manage multiple versions of a model.

- Use a project to manage your project files, connect to source control, review modified files, and compare revisions. See “Project Management”.
- Simulink notifies you if a model has changed on disk when updating, simulating, editing, or saving the model. Models can change on disk, for example, with source control operations and multiple users. Control this notification with the Model File Change Notification preference. See “Model File Change Notification” on page 4-57.
- As you edit a model, Simulink generates version information about the model, including a version number, who created and last updated the model, and an optional comments history log. Simulink saves these version properties with the model.
  - Use the Model Properties dialog box to view and edit some of the version information stored in the model and specify history logging.
  - The Model Info block lets you display version information as an annotation block in a model diagram.
- Use `Simulink.MDLInfo` to extract information from a model file without loading the block diagram into memory. You can use `MDLInfo` to query model version and Simulink version, find the names of referenced models without loading the model into memory, and attach arbitrary metadata to your model file.

Create Configurations

You can use a project to work with the revision control parts of the workflow: retrieving files, adding files to source control, checking out files, and committing edited files to source control.

To define configurations of files, you can label several files as a new mutually consistent configuration. Team members can get this set of files from the revision control system.

Configurations are different from revisions. Individual components can have revisions that work together only in particular configurations.

Tools for creating configurations in Simulink:

- Variant modeling. See “Variant Systems”.
- Project tools:
  - Label — Label project files. Use labels to apply metadata to files. You can group and sort by labels, label folders for adding to the path using shortcut functions, or create batch jobs to export files by label, for example, to manage files with the label `Diesel`. You cannot retrieve from source control by label, and labels persist across revisions.
  - Revision Log — Use Revert Project to choose a revision to revert to (SVN source control only).
  - Branch — Create branches of file versions, and switch to any branch in the repository (Git source control only).
  - Tag — You can tag all project files (SVN source control only) to identify a particular configuration of a project, and retrieve tagged versions from source control. However, continued development is limited. That is, you cannot tag again, and you must check out from `trunk` to apply tags.

- Archive — Package all project files in a zip file that you can create a project from. However, this packaging removes all source control information, because archiving is for exporting, sharing, and changing to another source control. You can commit the new zip file to source control.

See Also

More About

- “What Are Projects?” on page 16-3
- “Source Control in Projects”
- “Dependency Analysis”
- “Model Comparison”
- “Project Management”
- “Component-Based Modeling Guidelines” on page 22-2
- “Define Interfaces of Model Components” on page 22-17

Power Window Example

Power Window

In this section...

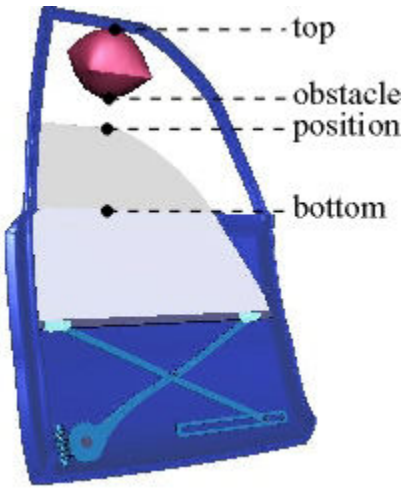
Study Power Windows

Automobiles use electronics for control operations such as:

- Opening and closing windows and sunroof
- Adjusting mirrors and headlights
- Locking and unlocking doors

These systems are subject to stringent operation constraints. Failures can cause dangerous and possibly life-threatening situations. As a result, careful design and analysis are needed before deployment.

This example focuses on the design of a power window system of an automobile, in particular, the passenger-side window. A critical aspect of this system is that it cannot exert a force of more than 100 N on an object when the window closes. When the system detects such an object, it must lower the window by about 10 cm.



As part of the design process, the example considers:

- Quantitative requirements for the window control system, such as timing and force requirements

- System requirements, captured in activity diagrams
- Data definitions for the signals used in activity diagrams

Other aspects of the design process that this example contains are:

- Managing the components of the system
- Building the model
- Validating the results of system simulation
- Generating code

MathWorks Software Used in This Example

In addition to Simulink, this example uses these additional MathWorks products:

- DSP System Toolbox
- Fixed-Point Designer
- Simscape Multibody
- Simscape Electrical™
- Simscape
- Simulink 3D Animation™
- Simulink Real-Time
- Simulink Coverage
- Stateflow

Quantitative Requirements

Quantitative requirements for the control are:

- The window must fully open and fully close within 4 s.
- If the up is issued for between 200 ms and 1 s, the window must fully open. If the down command is issued for between 200 ms and 1 s, the window must fully close.
- The window must start moving 200 ms after the command is issued.
- The force to detect when an object is present is less than 100 N.
- When closing the window, if an object is in the way, stop closing the window and lower the window by approximately 10 cm.

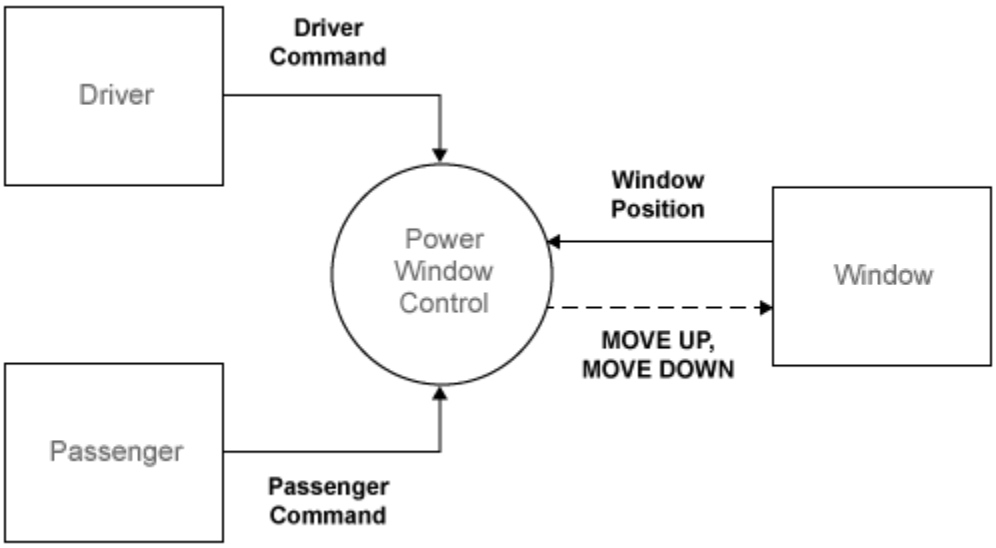
Capturing Requirements in Activity and Context Diagrams

Activity diagrams help you graphically capture the specification and understand how the system operates. A hierarchical structure helps with analyzing even large systems. At the top level, a context diagram describes the system environment and its interaction with the system under study in terms of data exchange and control operations. Then you can decompose the system into an activity diagram with processes and control specifications (CSPEC).

The processes guide the hierarchical decomposition. You specify each process using another activity diagram or a primitive specification (PSPEC). You can specify a PSPEC in a number of representations with a formal semantic, such as a Simulink block diagram. In addition, context diagrams graphically capture the context of system operation.

Context Diagram: Power Window System

The figure represents the context diagram of a power window system. The square boxes capture the environment, in this case, the driver, passenger, and window. Both the driver and passenger can send commands to the window to move it up and down. The controller infers the correct command to send to the window actuator (e.g., the driver command has priority over the passenger command). In addition, diagram monitors the state of the window system to establish when the window is fully opened and closed and to detect if there is an object between the window and frame.

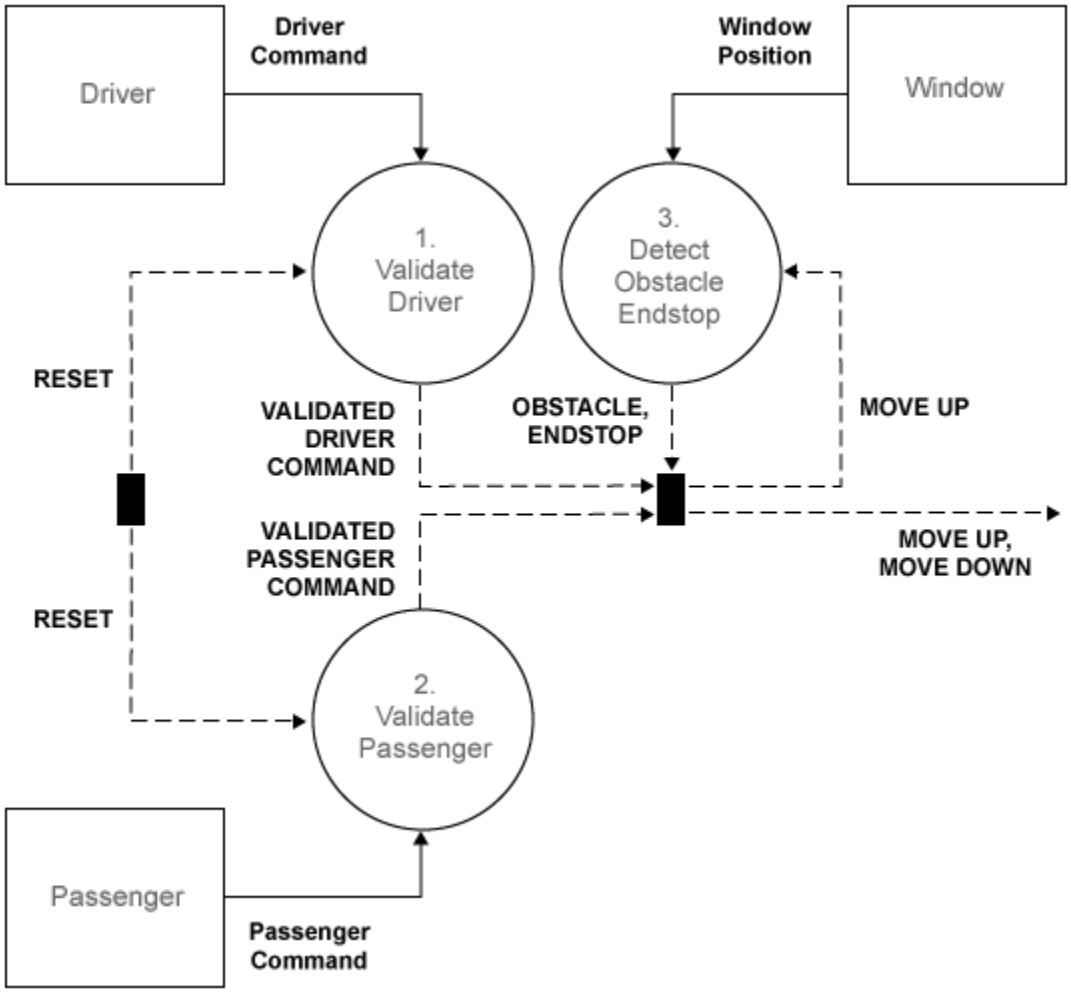


The circle (also known as a bubble) represents the power window controller. The circle is the graphical notation for a process. Processes capture the transformation of input data into output data. Primitive process might also generate. CSPECs typically consist of combinational or sequential logic to infer output control signals from input control.

For implementation in the Simulink environment, see “Implementation of Context Diagram: Power Window System” on page 23-26.

Activity Diagram: Power Window Control

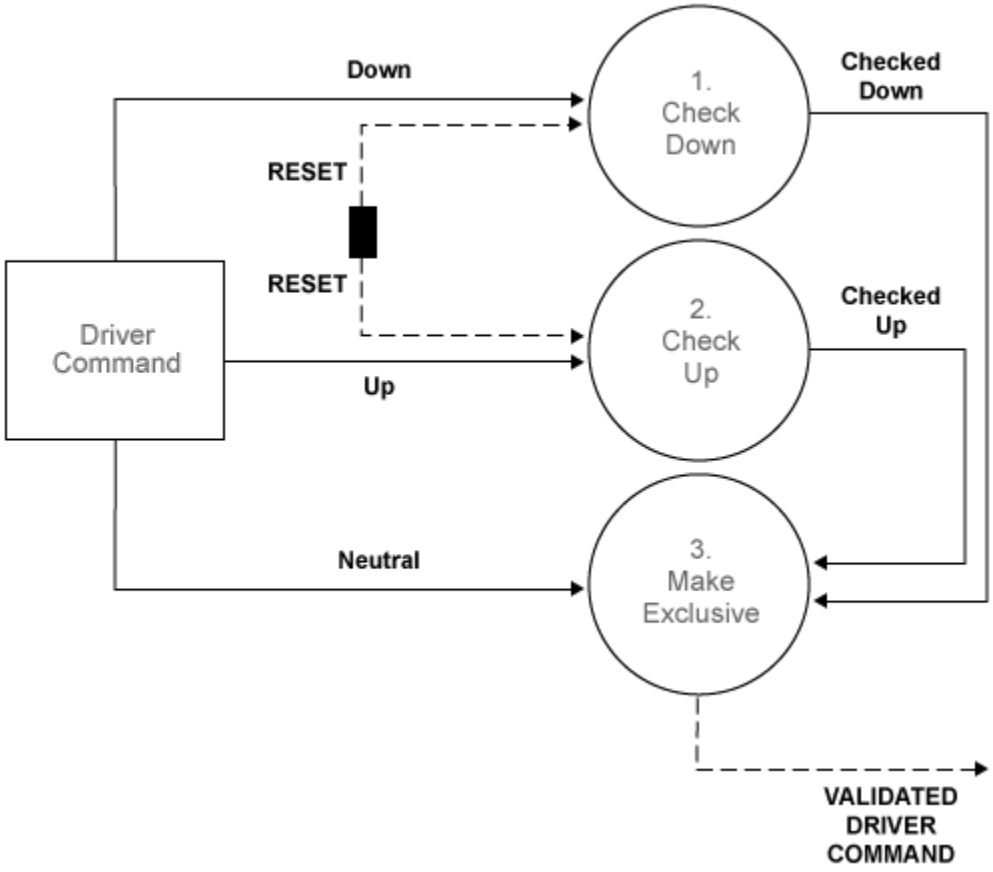
The power window control consists of three processes and a CSPEC. Two processes validate the driver and passenger input to ensure that their input is meaningful given the state of the system. For example, if the window is completely opened, the `MOVE DOWN` command does not make sense. The remaining process detects if the window is completely opened or completely closed and if an object is present. The CSPEC takes the control signals and infers whether to move the window up or down (e.g., if an object is present, the window moves down for about one second or until it reaches an endstop).



For implementation in the Simulink environment, see “Implementation of Activity Diagram: Power Window Control” on page 23-11.

Activity Diagram: Validate Driver

Each process in the VALIDATE DRIVER activity chart is primitive and specified by the following PSPEC. In the MAKE EXCLUSIVE PSPEC, for safety reasons the `DOWN` command takes precedence over the `UP` command.

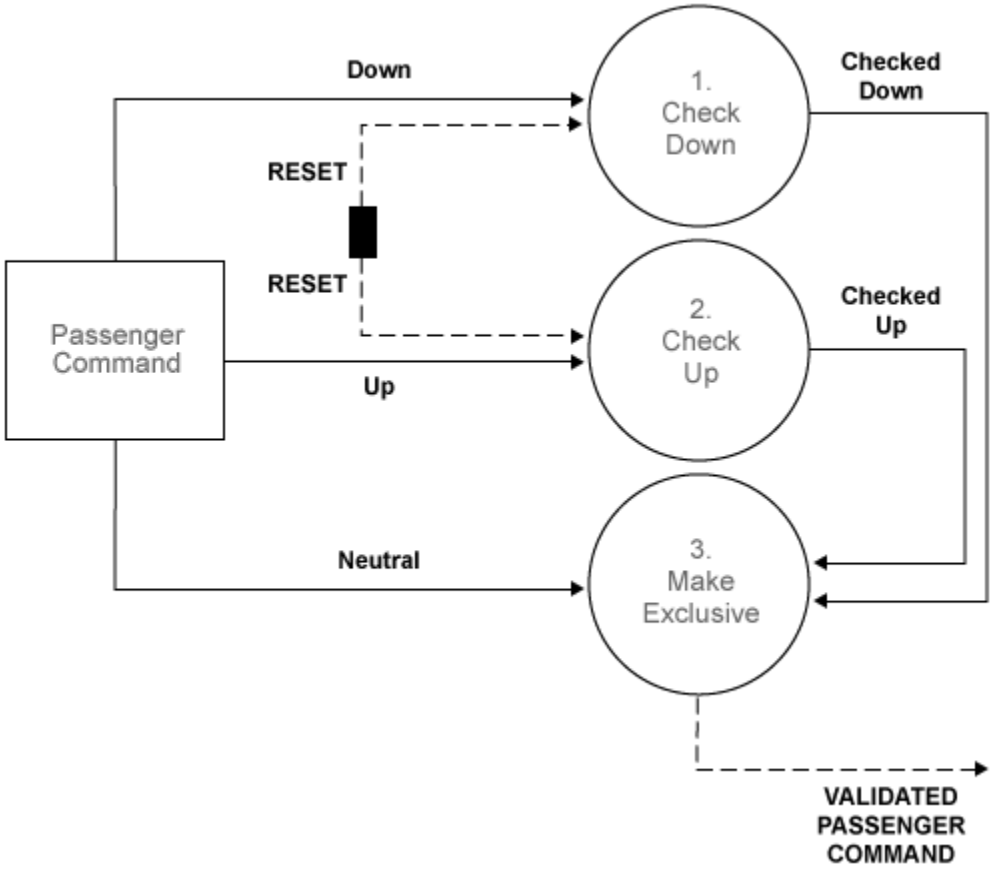



```
PSPEC 1.1.1: CHECK DOWN
  CHECKED_DOWN = DOWN and not RESET

PSPEC 1.1.2: CHECK UP
  CHECKED_UP = UP and not RESET

PSPEC 1.1.3: MAKE EXCLUSIVE
  VALIDATED_DOWN    = CHECKED_DOWN
  VALIDATED_UP      = CHECKED_UP and not CHECKED_DOWN
  VALIDATED_NEUTRAL = (NEUTRAL and not (CHECKED_UP and not CHECKED_DOWN))
                        or not (CHECKED_UP or CHECKED_DOWN)
```

For implementation in the Simulink environment, see “Implementation of Activity Diagram: Validate” on page 23-28.

Activity Diagram: Validate Passenger

The internals of the VALIDATE PASSENGER process are the same as the VALIDATE DRIVER process. The only difference is the different input and output.



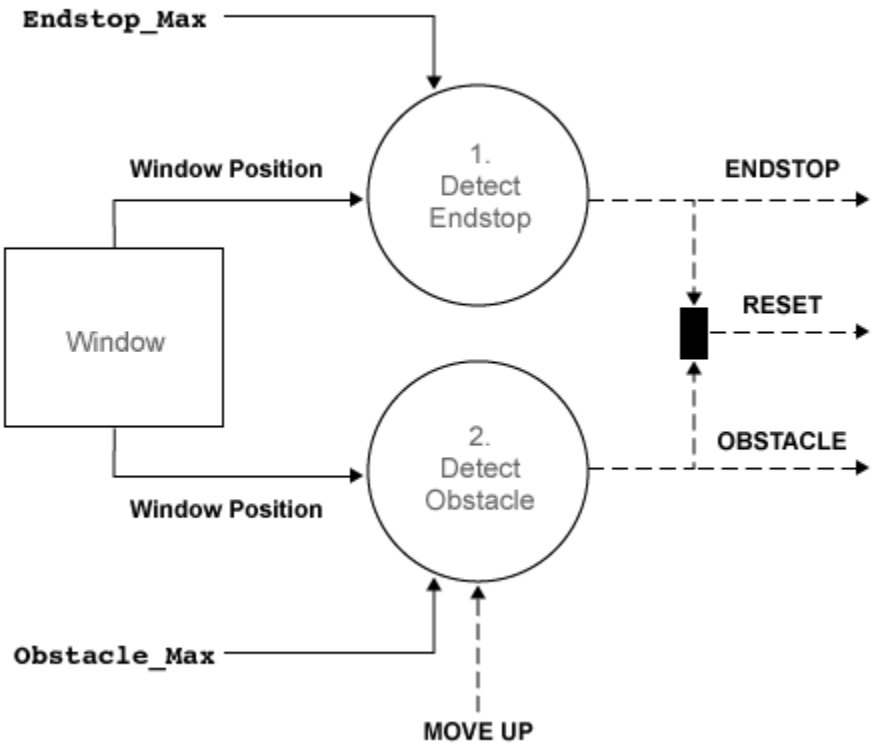

```
PSPEC 1.2.1: CHECK DOWN
  CHECKED_DOWN = DOWN and not RESET

PSPEC 1.2.2: CHECK UP
  CHECKED_UP = UP and not RESET

PSPEC 1.2.3: MAKE EXCLUSIVE
  VALIDATED_DOWN    =  CHECKED_DOWN
  VALIDATED_UP      =  CHECKED_UP and not CHECKED_DOWN
  VALIDATED_NEUTRAL = (NEUTRAL and not (CHECKED_UP and not CHECKED_DOWN))
                        or not (CHECKED_UP or CHECKED_DOWN)
```

For implementation in the Simulink environment, see “Activity Diagram: Validate Passenger” on page 23-6.

Activity Diagram: Detect Obstacle Endstop

The third process in the POWER WINDOW CONTROL activity diagram detects the presence of an obstacle or when the window reaches its top or bottom (ENDSTOP). The detection mechanism is based on the armature current of the window actuator. During normal operation, this current is within certain bounds. When the window reaches its top or bottom, the electromotor draws a large current (more than 15 A or less than –15 A) to try and sustain its angular velocity. Similarly, during normal operation the current is about 2 A or –2 A (depending on whether the window is opening or closing). When there is an object, there is a slight deviation from this value. To keep the window force on the object less than 100 N, the control switches to its emergency operation when it detects a current that is less than –2.5 A. This operations is necessary only when the window is rolling up, which

corresponds to a negative current in the particular wiring of this model. The DETECT OBSTACLE ENDSTOP activity diagram embodies this functionality.




```
CSPEC 1.3: DETECT OBSTACLE ENDSTOP
  RESET = OBSTACLE or ENDSTOP

PSPEC 1.3.1: DETECT ENDSTOP
  ENDSTOP = WINDOW_POSITION > ENDSTOP_MAX

PSPEC 1.3.2: DETECT OBSTACLE
  OBSTACLE = (WINDOW_POSITION > OBSTACLE_MAX) and MOVE_UP for 500 ms
```

For implementation in the Simulink environment, see “Activity Diagram: Detect Obstacle Endstop” on page 23-7.

Data Definitions

The functional decomposition unambiguously specifies each process by its decomposition or primitive specification (PSPEC). In addition, it must also formally specify the signals in the activity diagrams. Use data definitions for these specifications.

The following tables are data definitions for the signals used in the activity diagrams.

For the associated activity diagram, see “Context Diagram: Power Window System” on page 23-4.

Context Diagram: Power Window System Data Definitions

Signal	Information Type	Continuous/ Discrete	Data Type	Values
DRIVER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
PASSENGER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
WINDOW_POSITION	Data	Continuous	Real	0 to 0.4 m
MOVE_UP	Control	Discrete	Boolean	`'True'`, `'False'`
MOVE_DOWN	Control	Discrete	Boolean	`'True'`, `'False'`

For the associated activity diagram, see “Activity Diagram: Power Window Control” on page 23-4.

Activity Diagram: Power Window Control Data Definitions

Signal	Information Type	Continuous/ Discrete	Data Type	Values
DRIVER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
PASSENGER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
WINDOW_POSITION	Data	Continuous	Real	0 to 0.4 m
MOVE_UP	Control	Discrete	Boolean	`'True'`, `'False'`
MOVE_DOWN	Control	Discrete	Boolean	`'True'`, `'False'`

For the associated activity diagram, see “Activity Diagram: Validate Driver” on page 23-5.

Activity Diagram: Validate Driver Data Definitions

Signal	Information Type	Continuous/ Discrete	Data Type	Values
DRIVER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
PASSENGER_COMMAND	Data	Discrete	Aggregate	`Neutral`, `up`, `down`
WINDOW_POSITION	Data	Continuous	Real	0 to 0.4 m
MOVE_UP	Control	Discrete	Boolean	`'True'`, `'False'`
MOVE_DOWN	Control	Discrete	Boolean	`'True'`, `'False'`

For the associated activity diagram, see “Activity Diagram: Validate Passenger” on page 23-6.

Activity Diagram: Validate Passenger Data Definitions

Signal	Information Type	Continuous/ Discrete	Data Type	Values
NEUTRAL	Data	Discrete	Boolean	`'True'`, `'False'`
UP	Data	Discrete	Boolean	`'True'`, `'False'`
DOWN	Data	Discrete	Boolean	`'True'`, `'False'`
CHECKED_UP	Data	Discrete	Boolean	`'True'`, `'False'`
CHECKED_DOWN	Data	Discrete	Boolean	`'True'`, `'False'`

For the associated activity diagram, see “Activity Diagram: Detect Obstacle Endstop” on page 23-7.

Activity Diagram: Detect Obstacle Endstop Data Definitions

Signal	Information Type	Continuous/ Discrete	Data Type	Values
ENDSTOP_MIN	Data	Constant	Real	0.0 m
ENDSTOP_MAX	Data	Constant	Real	0.4 m
OBSTACLE_MAX	Data	Constant	Real	0.3 m

The model design iterates as we examine more detailed implementations. For information about how the model design iterates as you introduce more detail, see “Iterate on the Design” on page 23-35.

Simulink Power Window Controller Project

MATLAB and Simulink support Model-Based Design for embedded control design, from initial specification to code generation. To organize large projects and share your work with others, use “Project Management”.

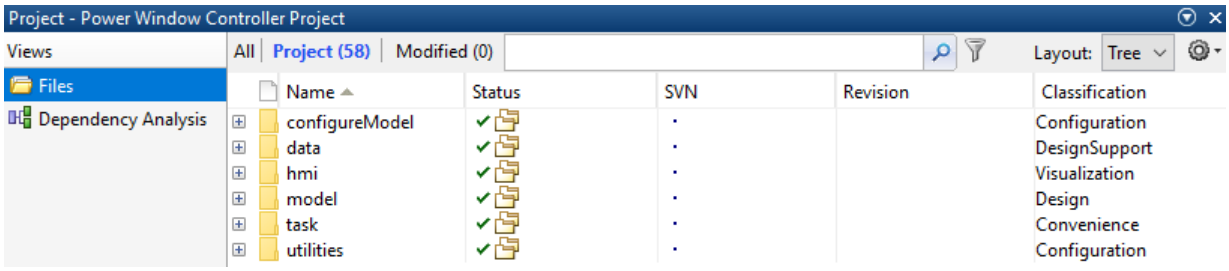
“Power Window Control Project” shows how you can use MathWorks tools and the Model-Based Design process to go from concept through implementation for an automobile power window system. It uses projects to organize the files and other model components.

In addition, this example shows how to link models to system documentation.

Explore the Power Window Controller Project

1 To open the Power Window Controller project, in the MATLAB Command Window, type:

```
slexPowerWindowStart
```



2 Explore the project folders. In particular, note the **task** folders. This folder contains scripts that run frequent tasks for a model. For the Power Window Controller Project, these scripts:

- Set up the model to control window movement on a controller area network (CAN).
- Set up the model to use the Stateflow and Simulink software to model discrete-event reactive behavior and continuous time behavior, with a low-order plant model.
- Set up the model with a more detailed plant model that includes power effects in the electrical and mechanical domains. The plant model validates the force exerted by the window on a trapped object.
- Set up the model with a model that includes other effects that may change the model, such as quantization of the measurements.

  Note These scripts also simulate the model. To only configure the model, see the scripts in the **configureModel** folder.
- Use the increase coverage model to generate the model coverage report.

3 The **Project Shortcuts** section contains quick-access commands that you can double-click to perform common tasks such as:

- Add projects to MATLAB paths.
- Perform interactive testing.
- Validate model testing with model coverage.
- Open the main model.
- Simulate the model with various configurations.
- Generate a model coverage report for increased coverage of the model.
- Open the model used for increasing model coverage.

Simulink Power Window Controller

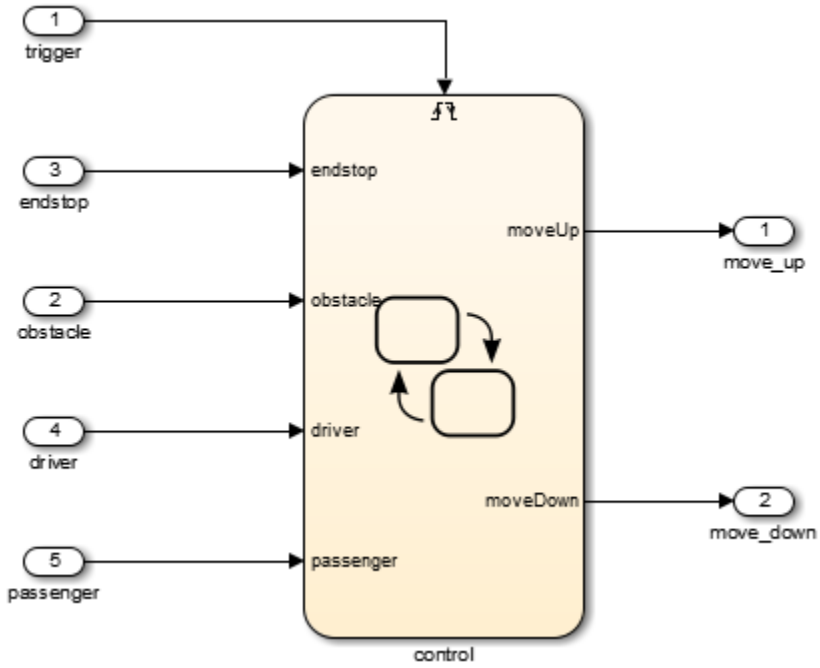
Implementation of Activity Diagram: Power Window Control

This topic describes the high-level discrete-event control specification for a power window control.

You can model the discrete-event control of the window with a Stateflow chart. A Stateflow chart is a finite state machine with hierarchy and parallelism. This state machine contains the basic states of the power window system: up, auto-up, down, auto-down, rest, and emergency. It models the state transitions and accounts for the precedence of driver commands over the passenger commands. It also includes emergency behavior that activates when the software detects an object between the window and the frame while moving up.

The initial Simulink model for the power window control, `slexPowerWindowControl`, is a discrete-event controller that runs at a given sample rate.

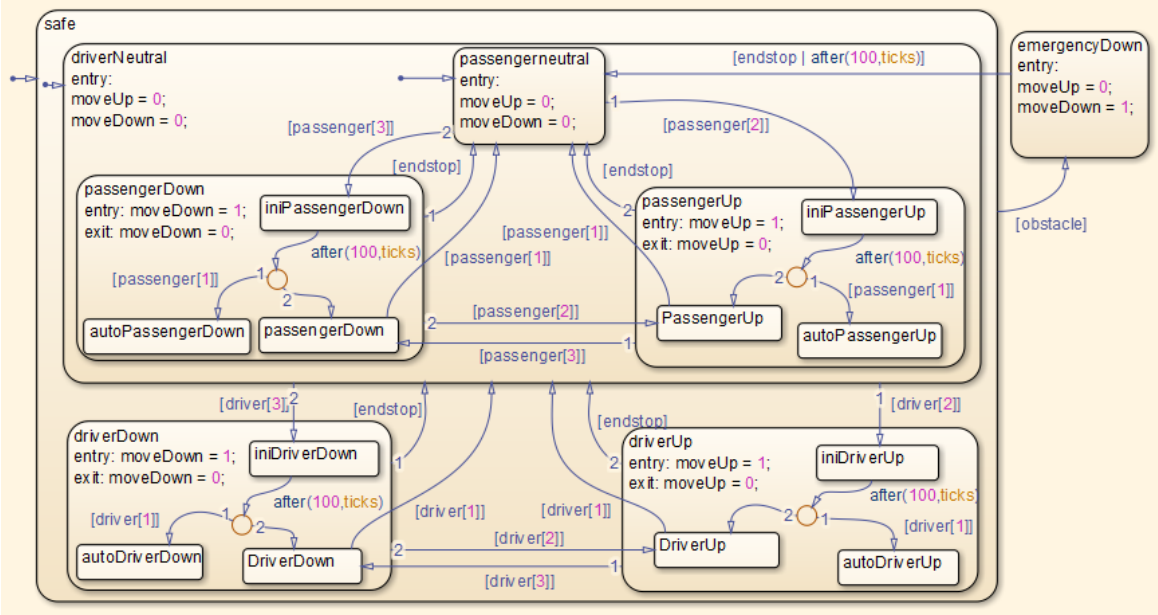
In this implementation, open the power window control subsystem and observe that the Stateflow chart with the discrete-event control forms the CSPEC, represented by the tilted thick bar in the bottom right corner. The detect_obstacle_endstop subsystem encapsulate the threshold detection mechanisms.



The discrete-event control is a Stateflow model that extends the state transition diagram notion with hierarchy and parallelism. State changes because of passenger commands are encapsulated in a *super state* that does not correspond to an active driver command.

Consider the control of the passenger window. The passenger or driver can move this window up and down.

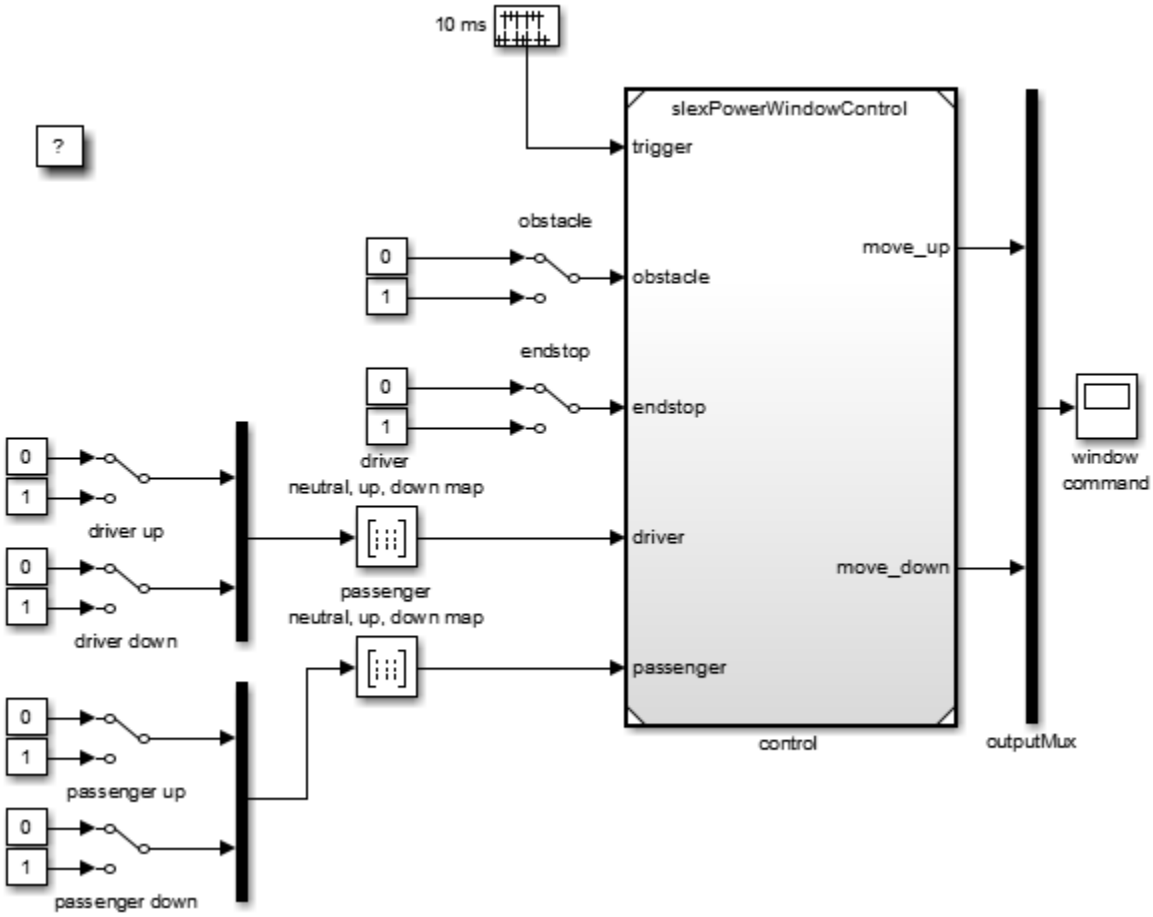
This state machine contains the basic states of the power window system: up, auto-up, down, auto-down, rest, and emergency.



Interactive Testing

Control Input

The `slexPowerWindowCntlInteract` model includes this control input as switches. Double-click these switches to manually operate them.



Test the state machine that controls a power window by running the input test vectors and checking that it reaches the desired internal state and generates output. The power window has the following external inputs:

- Passenger input
- Driver input
- Window up or down
- Obstacle in window

Each input consists of a vector with these inputs.

Passenger Input

Element	Description
`neutral`	Passenger control switch is not depressed.
`up`	Passenger control switch generates the up signal.
`down`	Passenger control switch generates the down signal.

Driver Input

Element	Description
`neutral`	Driver control switch is not depressed.
`up`	Driver control switch generates the up signal.
`down`	Driver control switch generates the down signal.

Window Up or Down

Element	Description
`0`	Window moves freely between top or bottom.
`1`	Window is stuck at the top or bottom because of physical limitations.

Obstacle in Window

Element	Description
`0`	Window moves freely between top or bottom.
`1`	Window has obstacle in the frame.

Generate the passenger and driver input signals by mapping the up and down signals according to this table:

Inputs		Outputs		
up	**down**	**up**	**down**	**neutral**
`0`	`0`	`0`	`0`	`1`
`0`	`1`	`0`	`1`	`0`
`1`	`0`	`1`	`0`	`0`
`1`	`1`	`0`	`0`	`1`

The inputs explicitly generate the `neutral` event from the `up` and `down` events, generated by pressing a power window control switch. The inputs are entered as a truth table in the passenger neutral, up, down map and the driver neutral, up, down map.

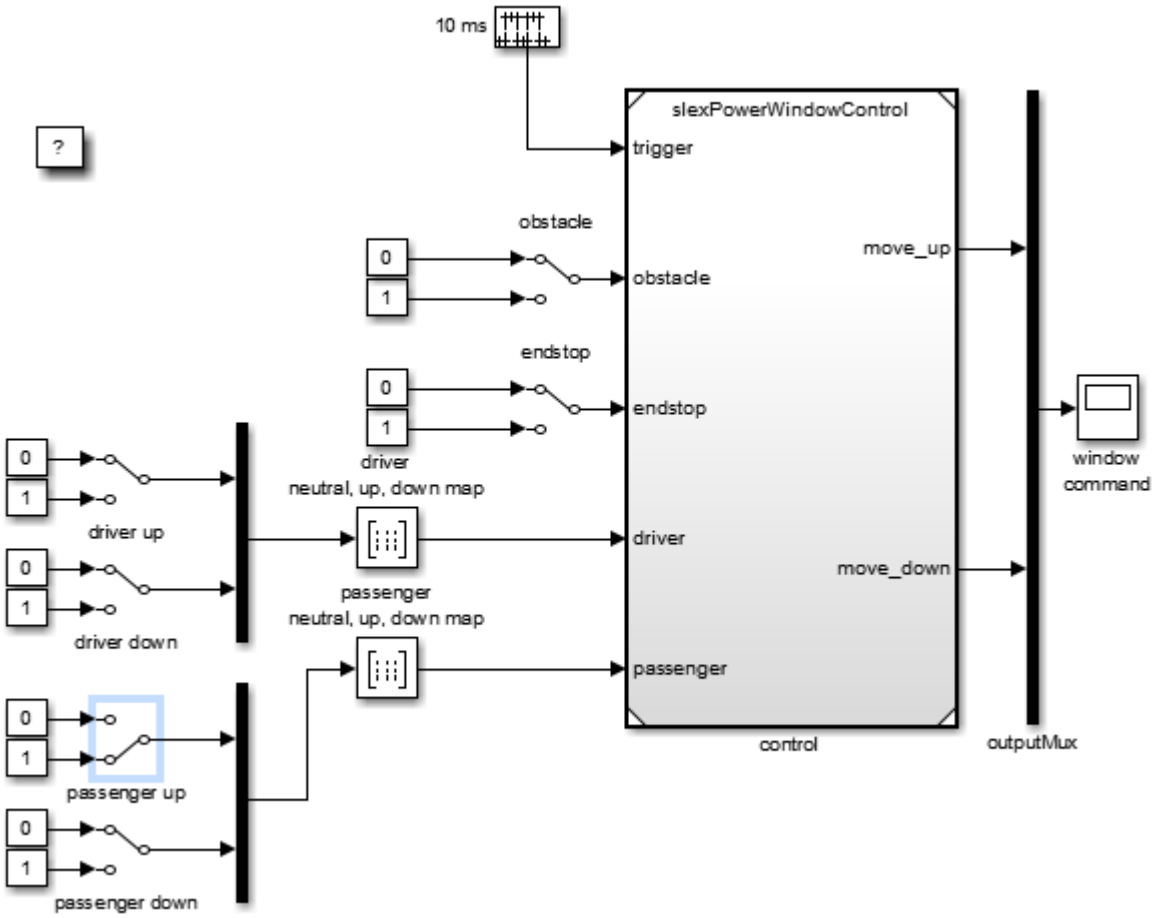
Experimental Results from Interactive Testing

Case 1: Window Up

To observe the state machine behavior:

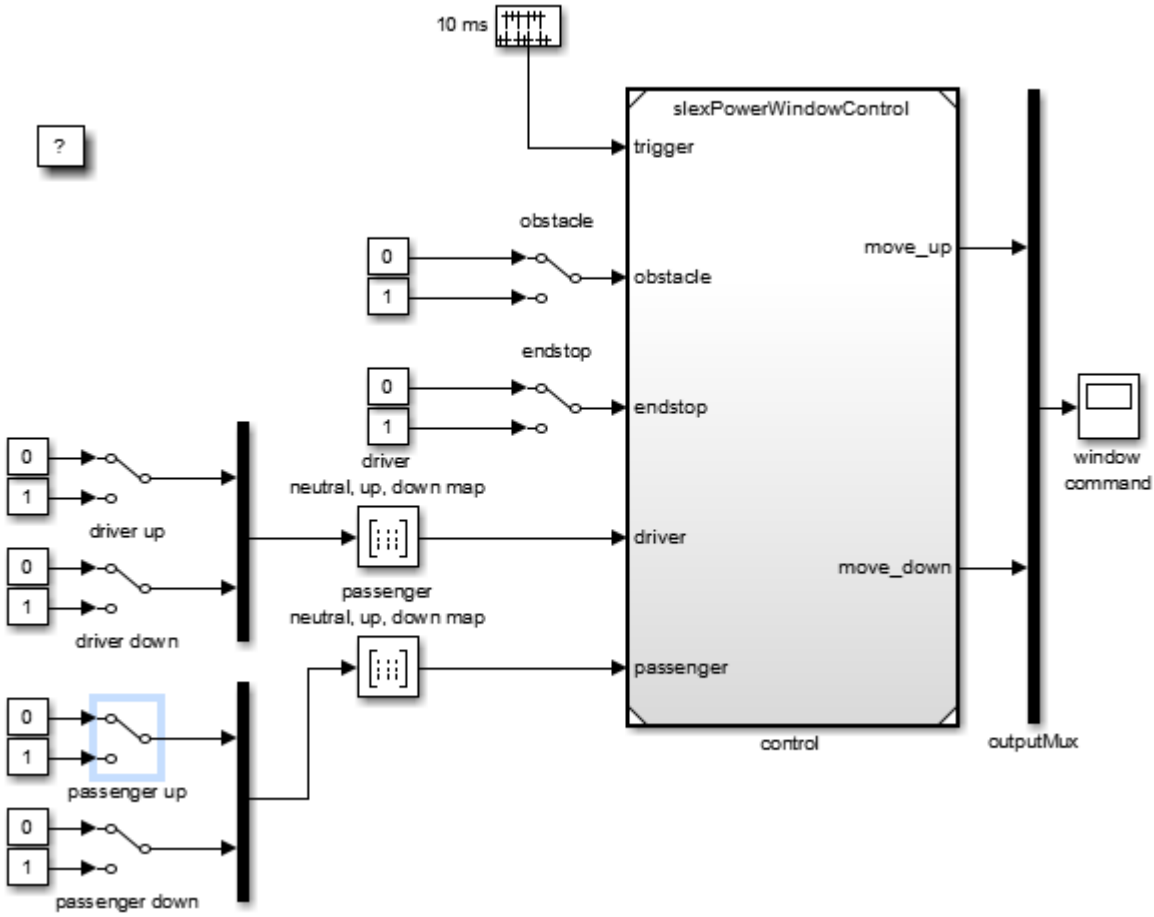
1. Open the `slexPowerWindowCntlInteract` model.

2 Run the simulation and then double-click the passenger up switch.



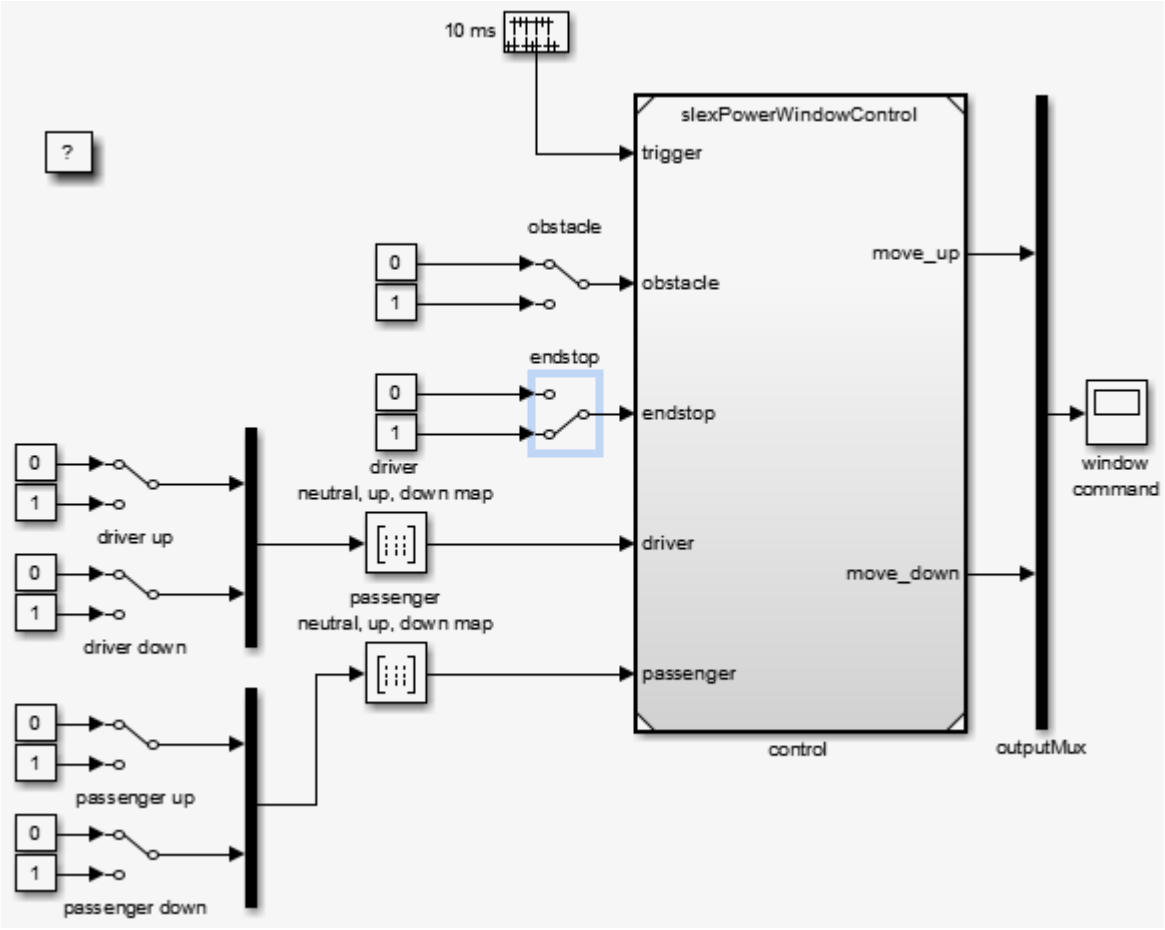
If you press the physical window switch for more than one second, the window moves up until the up switch is released (or the top of the window frame is reached and the `endstop` event is generated).

3 Double-click the selected passenger up switch to release it.



4 Simulate the model.

Setting the endstop switch generates the endstop event.

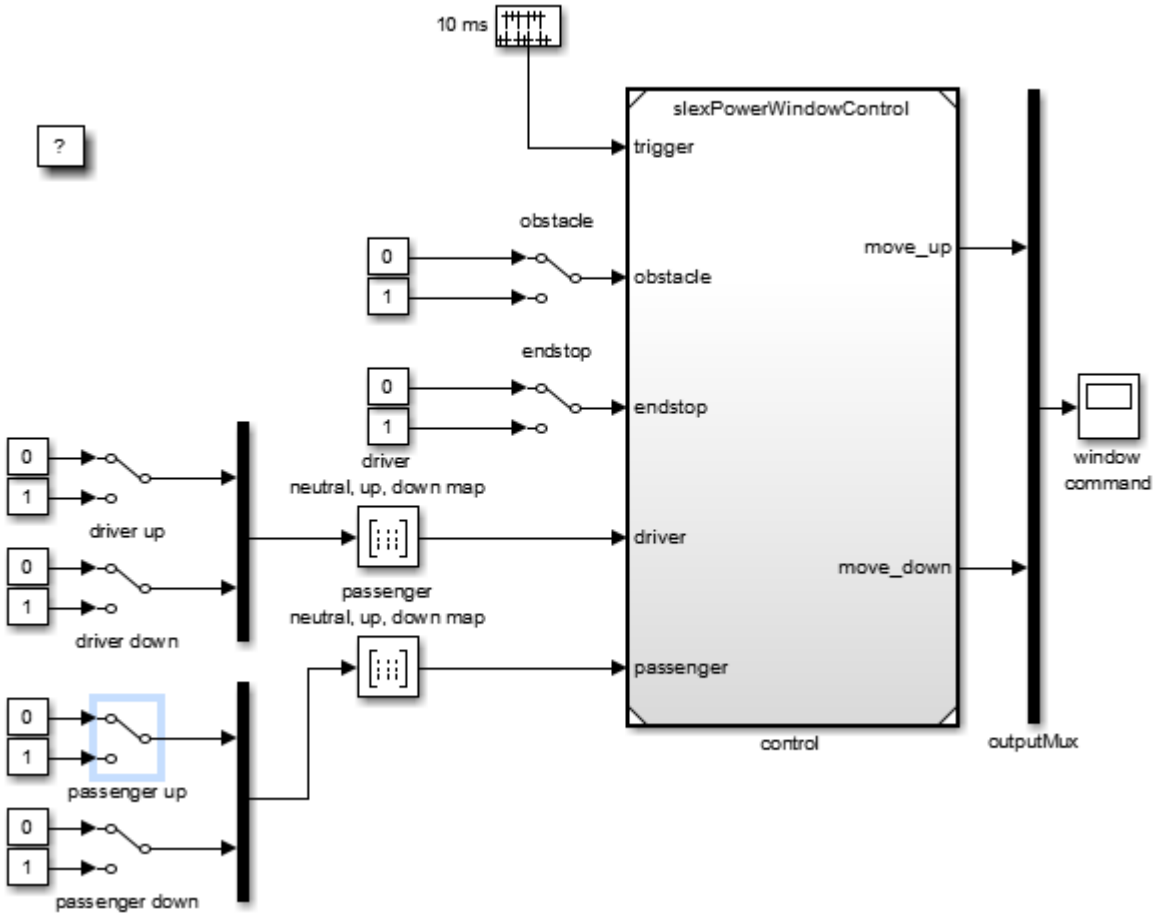


Case 2: Window Auto-Up

If you press the physical passenger window up switch for a short period of time (less than a second), the software activates auto-up behavior and the window continues to move up.

1 Press the physical passenger window up switch for a short period of time (less than a second).

Ultimately, the window reaches the top of the frame and the software generates the `endstop` event. This event moves the state machine back to its neutral state.

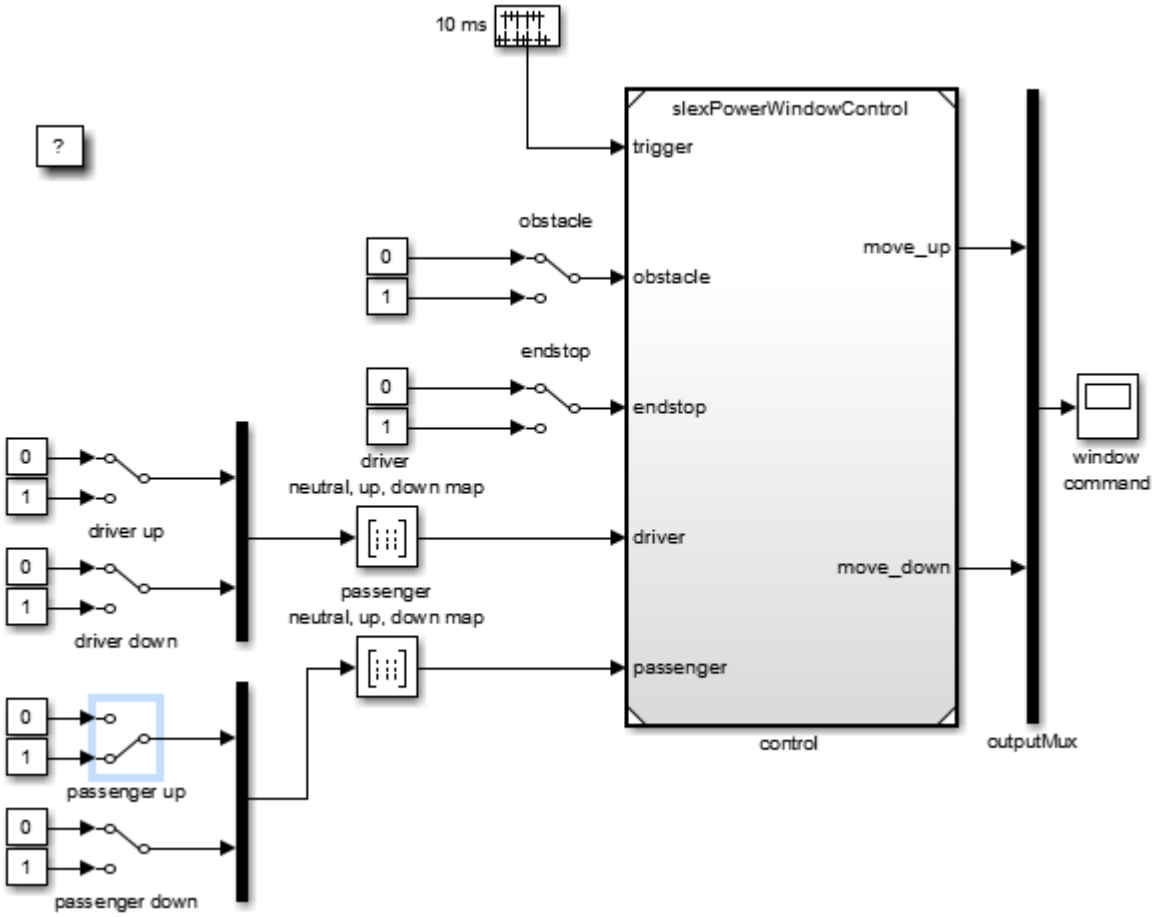


2 Simulate the model.

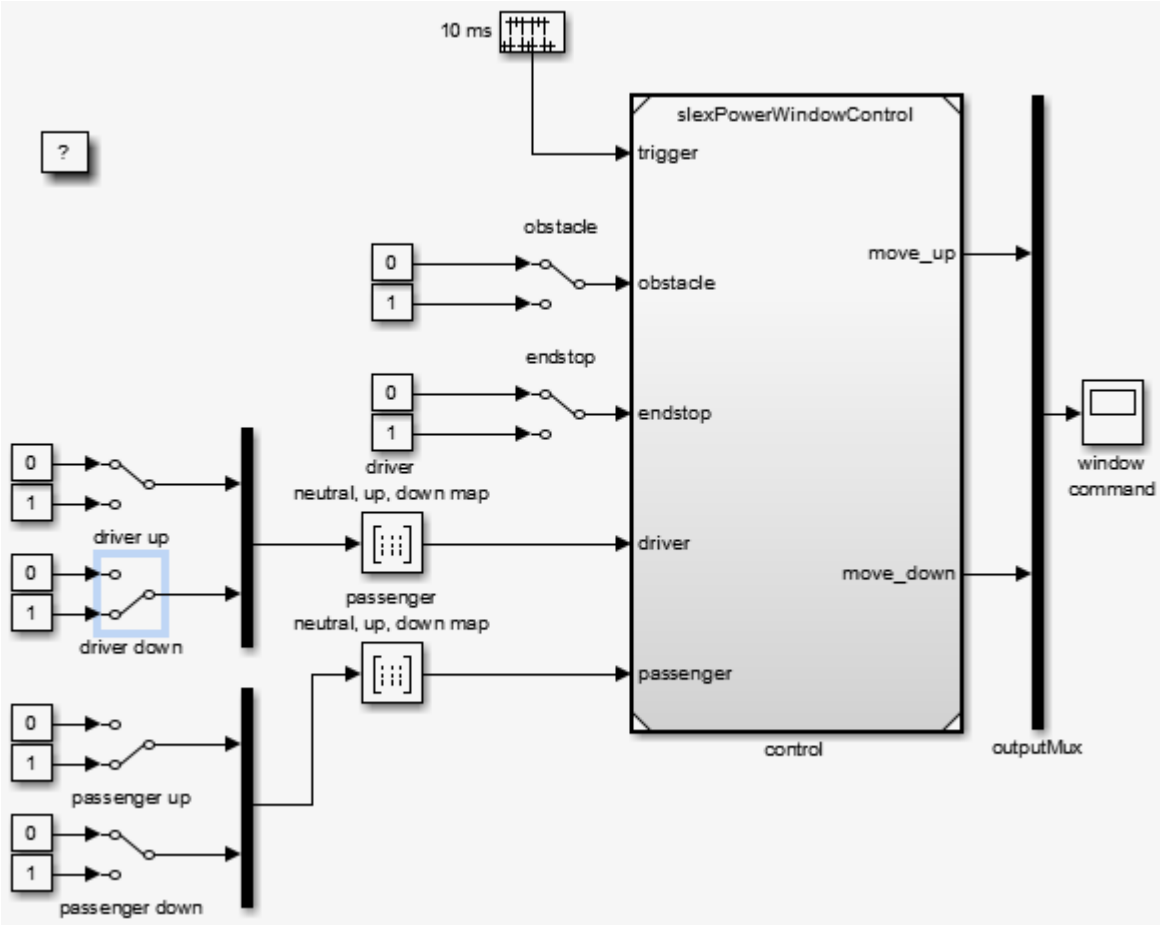
Case 3: Driver-Side Precedence

The driver switch for the passenger window takes precedence over the driver commands. To observe the state machine behavior in this case:

1 Run the simulation, and then move the system to the `passenger up` state by double-clicking the passenger window up switch.



2 Double-click the driver down switch.

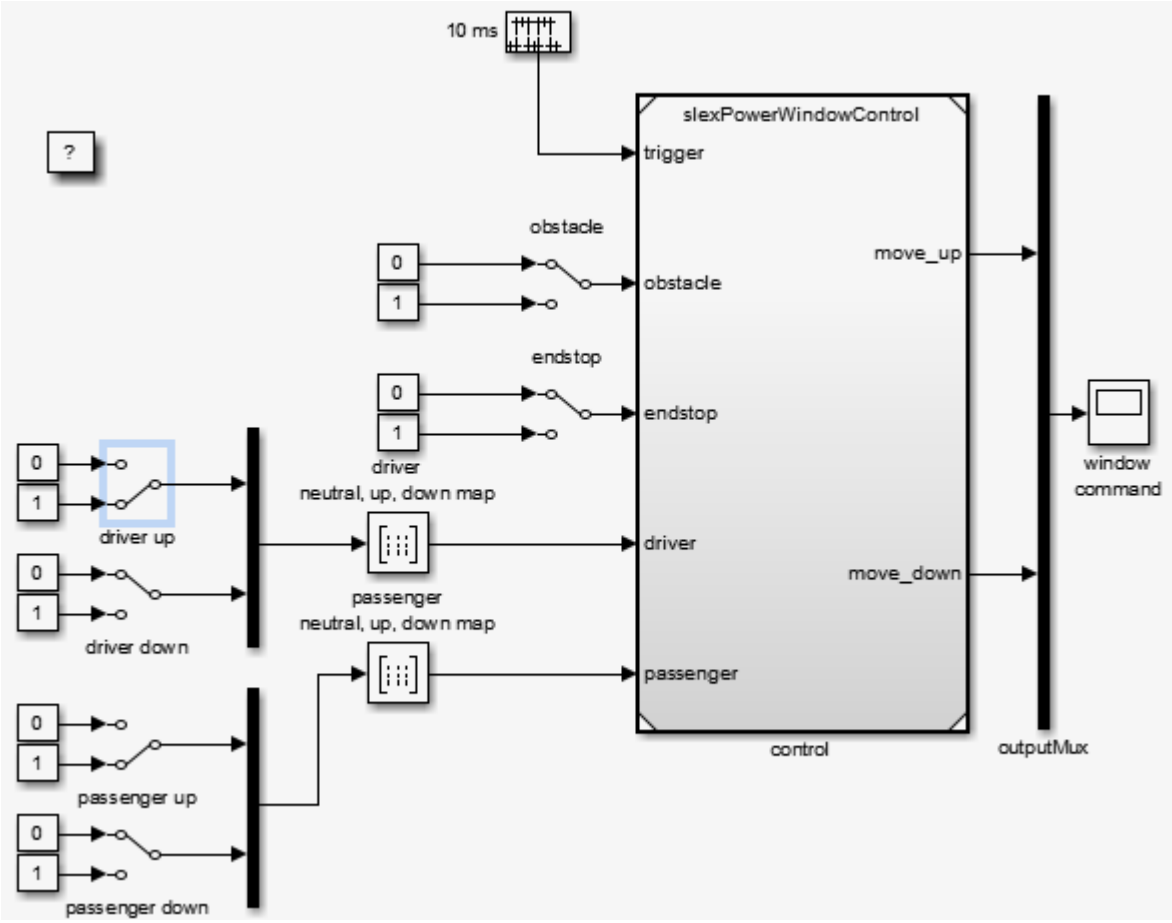


3 Simulate the model.

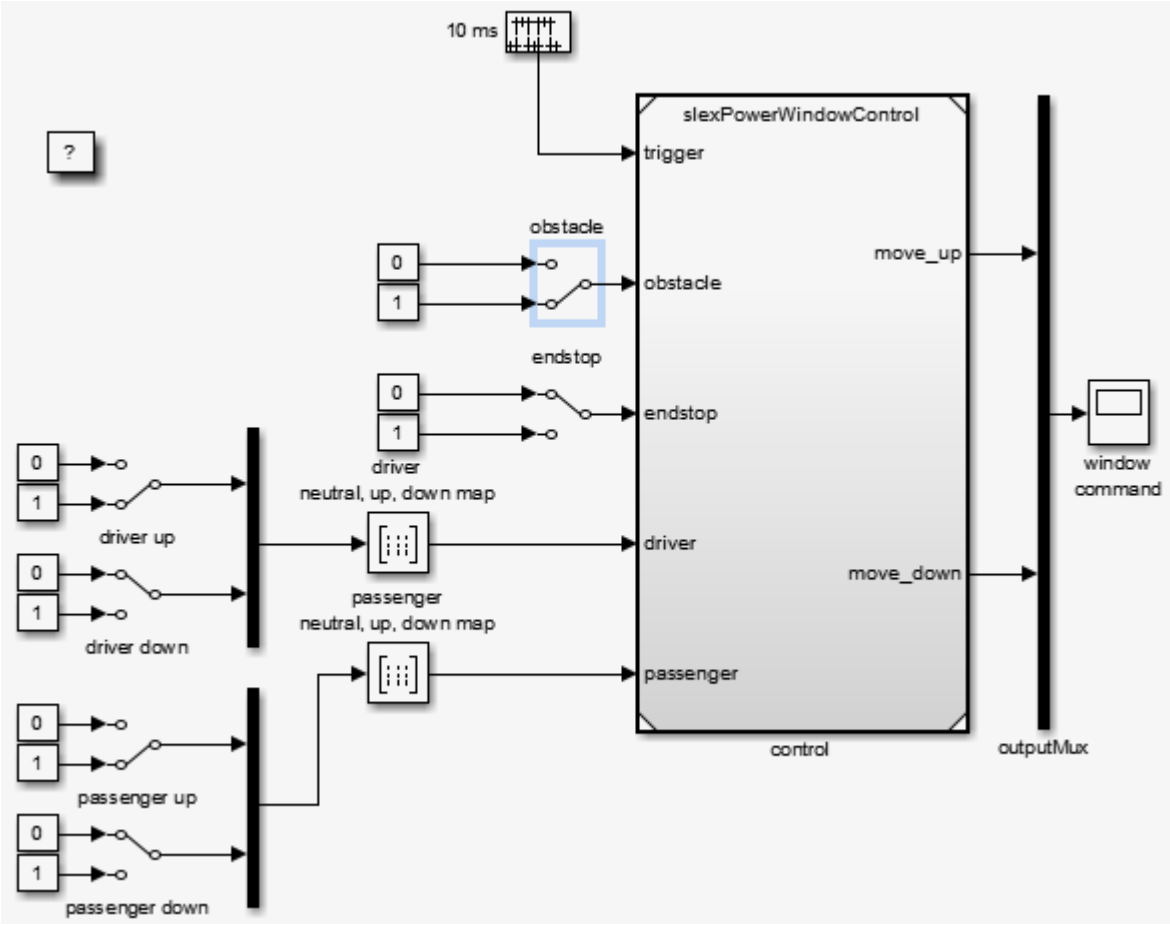
4 Notice how the state machine moves to the driver control part to generate the window down output instead of the window up output.

5 Double-click the driver control to driver up. Double-click the driver down switch.

The driver window up state is reached, which generates the window up output again, i.e., `windowUp = 1`.



6 To observe state behavior when an object is between the window and the frame, double-click the obstacle switch.



7 Simulate the model.

On the next sample time, the state machine moves to its `emergencyDown` state to lower the window a few inches. How far the software lowers the window depends on how long the state machine is in the `emergencyDown` state. This behavior is part of the next analysis phase.

If a driver or passenger window switch is still active, the state machine moves into the up or down states upon the next sample time after leaving the emergency state. If the obstacle switch is also still active, the software again activates the emergency state at the next sample time.

Model Coverage

Validation of the Control Subsystem

Validate the discrete-event control of the window using the model coverage tool. This tool helps you determine the extent to which a model test case exercises the conditional branches of the controller. It helps evaluate whether all transitions in the discrete-event control are taken, given the test case, and whether all clauses in a condition that enables a particular transition have become true. Multiple clauses can enable one transition, e.g., the transition from emergency back to neutral occurs when either 100 ticks have occurred or if the endstop is reached.

To achieve full coverage, each clause evaluates to true and false for the test cases used. The percentage of transitions that a test case exercises is called its model coverage. Model coverage is a measure of how thoroughly a test exercises a model.

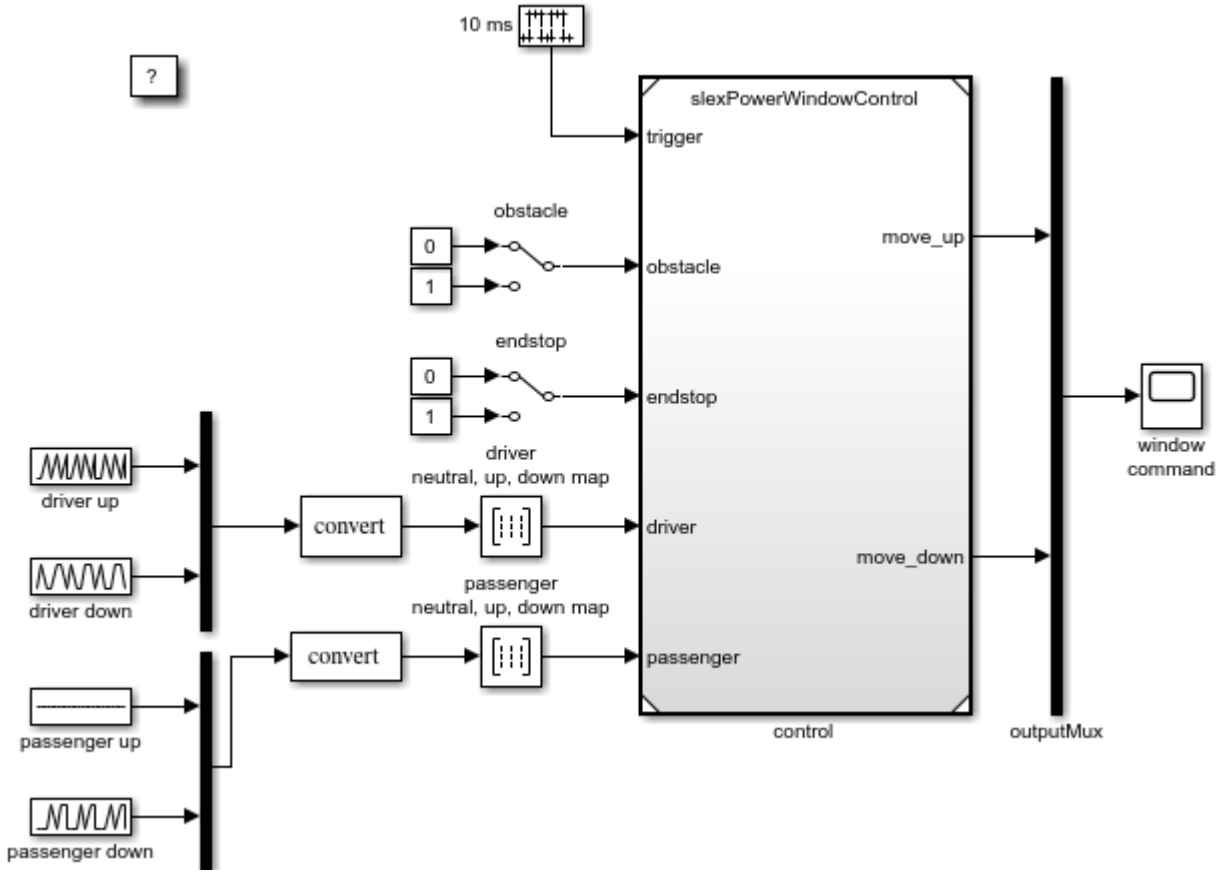
Using Simulink Coverage software, you can apply the following test to the power window controller.

Position	Step						
	0	1	2	3	4	5	6
Passenger up	0	0	0	0	0	0	0
Passenger down	0	0	0	1	0	1	1
Driver up	0	0	1	0	1	0	1
Driver down	0	1	0	0	1	1	0

With this test, all switches are inactive at time 0. At regular 1 s steps, the state of one or more switches changes. For example, after 1 s, the driver down switch becomes active. To automatically run these input vectors, replace the manual switches by prescribed sequences of input. To see the preconstructed model:

1 In the MATLAB Command Window, type:

```
slexPowerWindowCntlCoverage
```



2 Simulate the model to generate the Simulink Coverage coverage report.

For the `slexPowerWindowCntlCoverage` model, the report reveals that this test handles 100% of the decision outcomes from the driver neutral, up, down map block. However, the test achieves only 50% coverage for the passenger neutral, up, down map block. This coverage means the overall

coverage for `slexPowerWindowCntlCoverage` is 45% while the overall coverage for the `slexPowerWindowControl` model is 42%. A few of the contributing factors for the coverage levels are:

- Passenger up block does not change.
- Endstop and obstacle blocks do not change.

Increase Model Coverage

To increase total coverage to 100%, you need to take into account all possible combinations of driver, passenger, obstacle, and endstop settings. When you are satisfied with the control behavior, you can create the power window system. For more information, see “Create Model Using Model-Based Design” on page 23-26.

This example increases the model coverage for the validation of the discrete-event control of the window. To start, the example uses inputs from `slexPowerWindowCntlCoverage` as a baseline for the model coverage. Next, to further exercise the discrete-event control of the window, it creates more input sets. The spreadsheet file, `inputCntlCoverageIncrease.xlsx`, contains these input sets using one input set per sheet.

In the example, the `slexPowerWindowSpreadsheetGeneration` utility function, which creates a spreadsheet template from the controller model, `slexPowerWindowControl`, creates the `inputCntlCoverageIncrease.xlsx`. In `inputCntlCoverageIncrease.xlsx`, the function uses the block names in the controller model as signal names. `slexPowerWindowSpreadsheetGeneration` defines the sheet names. The `slexWindowSpreadsheetAddInput` utility function populates `inputCntlCoverageIncrease.xlsx` with signal data.

The sheet names of these input sets and their descriptions are:

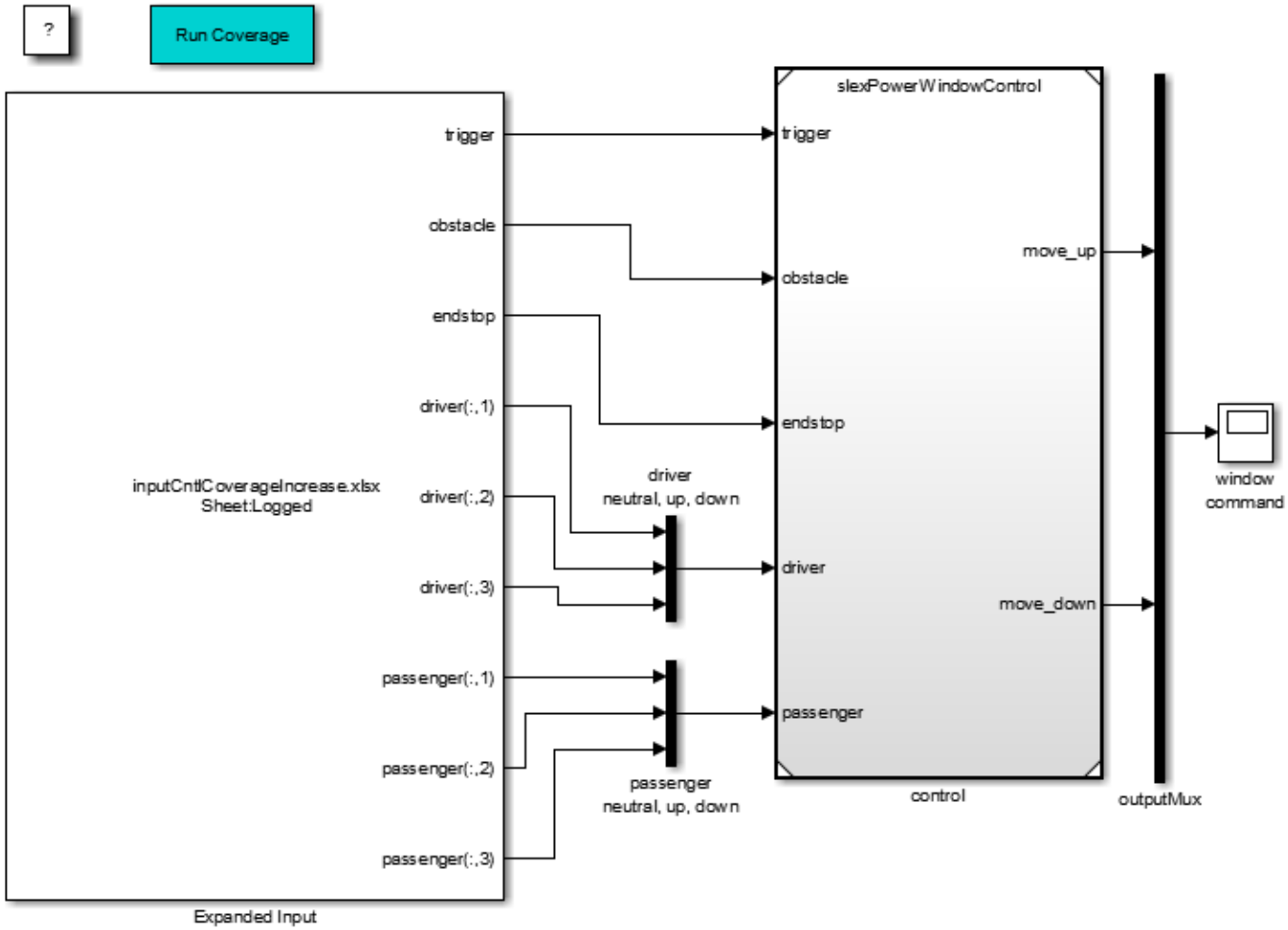
Sheet Name	Description
`Logged`	Inputs logged from `slexPowerWindowCntlCoverage`
`LoggedObstacleOffEndStopOn`	Inputs logged from `slexPowerWindowCntlCoverage` with ability to hit endstop
`LoggedObstacleOnEndStopOff`	Inputs logged from `slexPowerWindowCntlCoverage` with obstacle in window
`LoggedObstacleOnEndStopOn`	Inputs logged from `slexPowerWindowCntlCoverage` with obstacle in window and ability to hit endstop
`DriverLoggedPassengerNeutral`	Inputs logged from `slexPowerWindowCntlCoverage` for driver only and passenger takes no action
`DriverDownPassengerNeutral`	Driver is putting down window and passenger takes no action
`DriverUpPassengerNeutral`	Driver is putting up window and passenger takes no action
`DriverAutoDownPassengerNeutral`	Driver is putting down window for one second (auto-down) and passenger takes no action
`DriverAutoUpPassengerNeutral`	Driver is putting up window for one second (auto-up) and passenger takes no action
`PassengerAutoDownDriverNeutral`	Passenger is putting down window for one second (auto-down) and driver takes no action

Sheet Name	Description
`PassengerAutoUpDriverNeutral`	Passenger is putting up window for one second (auto-up) and driver takes no action

To automatically run these input vectors, replace the inputs to the discrete-event control with the From Spreadsheet block using the file, `inputCntlCoverageIncrease.xlsx`. This file contains the multiple input sets. To see the preconstructed model:

1. In the MATLAB Command Window, type:

   ```
   slexPowerWindowCntlCoverageIncrease
   ```



2. To generate the Simulink Coverage coverage report for multiple input set, double click the Run Coverage subsystem in the model.

   For the `slexPowerWindowCntlCoverageIncrease` model, the report reveals that using multiple input sets has successfully raised the overall coverage for the `slexPowerWindowControl` model from 42% to 78%. Coverage levels are less than 100% because of missing input sets for:

   - Passenger up state

- Driver up and down states
- Passenger automatic down and automatic up states

Create Model Using Model-Based Design

Why Use Model-Based Design?

In Model-Based Design, a system model is at the center of the development process, from requirements development, through design implementation, and testing. Use Model-Based Design to:

- Use a common design environment across project teams.
- Link designs directly to requirements.
- Integrate testing with design to continuously identify and correct errors.
- Refine algorithms through multidomain simulation.
- Automatically generate embedded software code.
- Develop and reuse test suites.
- Automatically generate documentation for the model.
- Reuse designs to deploy systems across multiple processors and hardware targets.

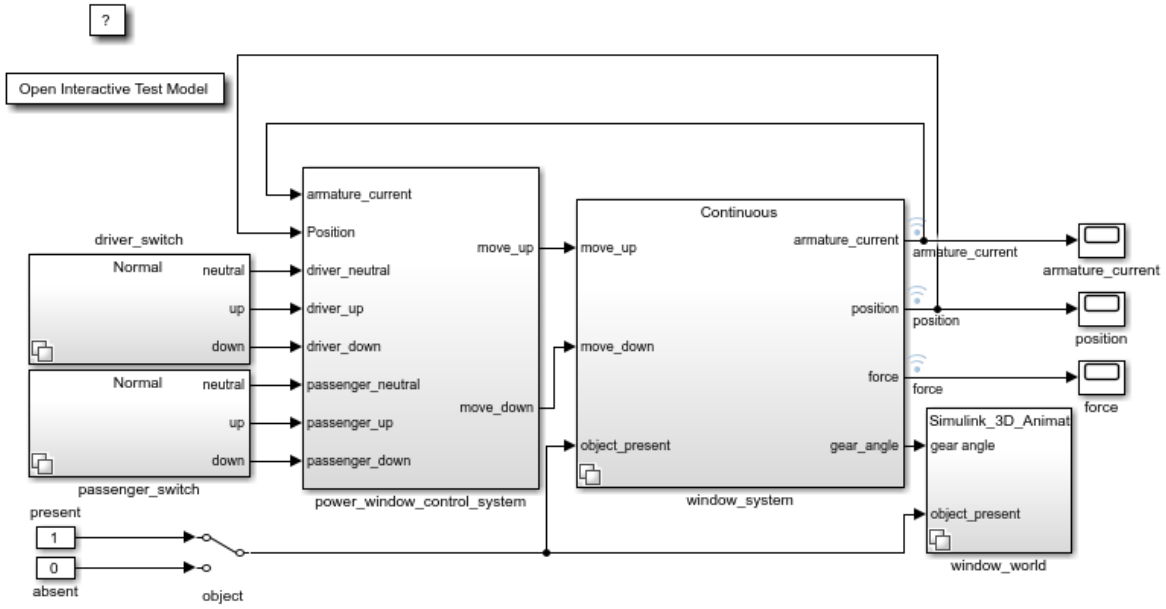
Implementation of Context Diagram: Power Window System

For requirements presented as a context diagram, see “Context Diagram: Power Window System” on page 23-4.

Create a Simulink model to resemble the context diagram.

1. Place the plant behavior into one subsystem.
2. Create two subsystems that contain the driver and passenger switches.
3. Add a control mechanism to conveniently switch between the presence and absence of the object.
4. Put the control in one subsystem.
5. Connect the new subsystems.
6. To see an implementation of this model, in the MATLAB Command Window, type:

```
slexPowerWindowStart
```



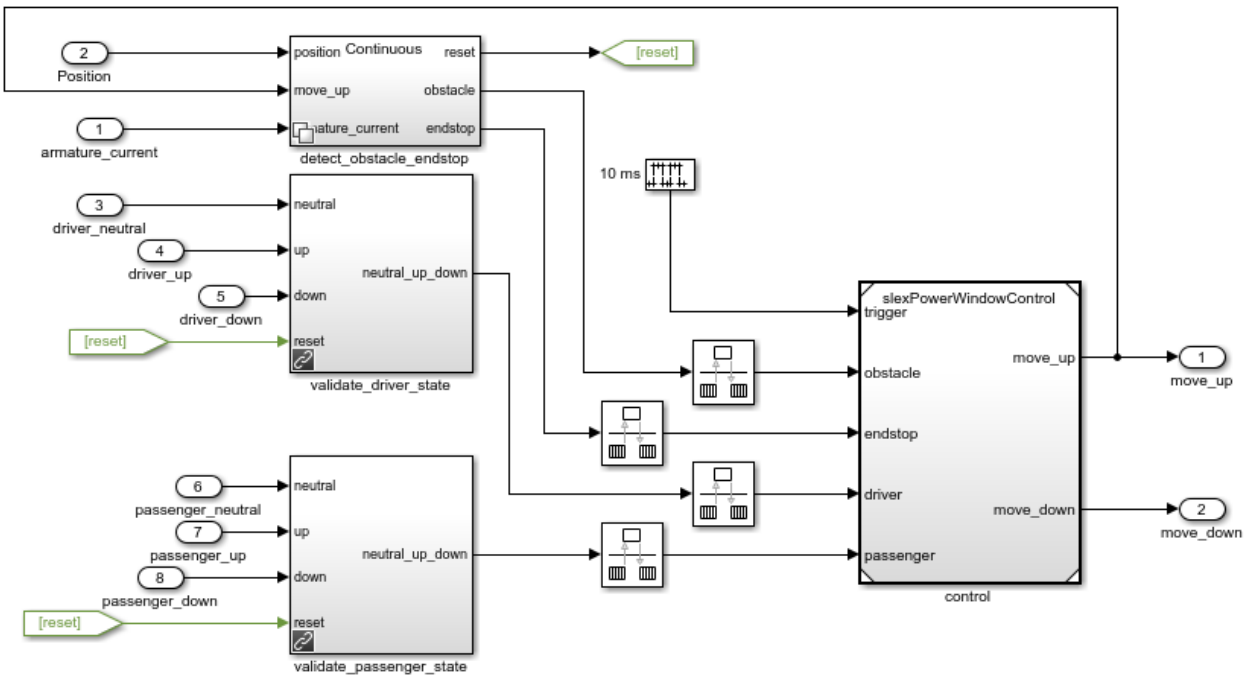
You can use the power window control activity diagram (“Activity Diagram: Power Window Control” on page 23-4) to decompose the power window controller of the context diagram into parts. This diagram shows the input and output signals present in the context diagram for easier tracing to their origins.

Implement Power Window Control System

To satisfy full requirements, the power window control must work with the validation of the driver and passenger inputs and detect the endstop.

For requirements presented as an activity diagram, see “Activity Diagram: Power Window Control” on page 23-4.

Double-click the slexPowerWindowExample/power_window_control_system block to open the following subsystem:



Implementation of Activity Diagram: Validate

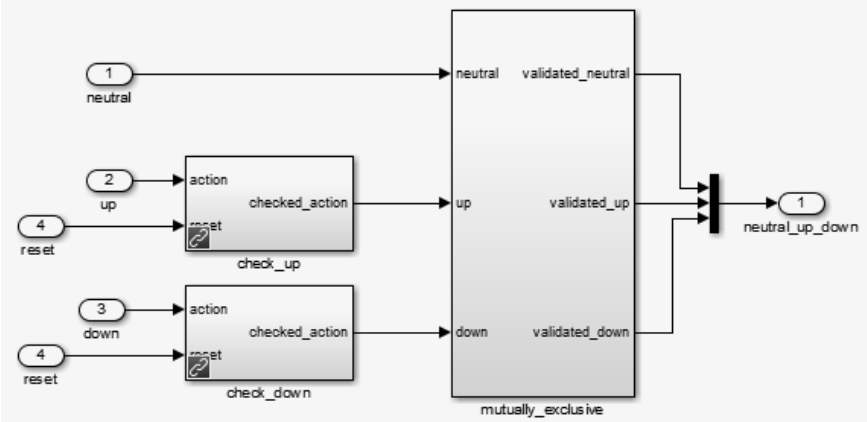
For requirements presented as activity diagrams, see “Activity Diagram: Validate Driver” on page 23-5 and “Activity Diagram: Validate Passenger” on page 23-6.

The activity diagram adds data validation functionality for the driver and passenger commands to ensure correct operation. For example, when the window reaches the top, the software blocks the `up` command. The implementation decomposes each validation process in new subsystems. Consider the validation of the driver commands (validation of the passenger commands is similar). Check if the model can execute the `up` or `down` commands, according to the following:

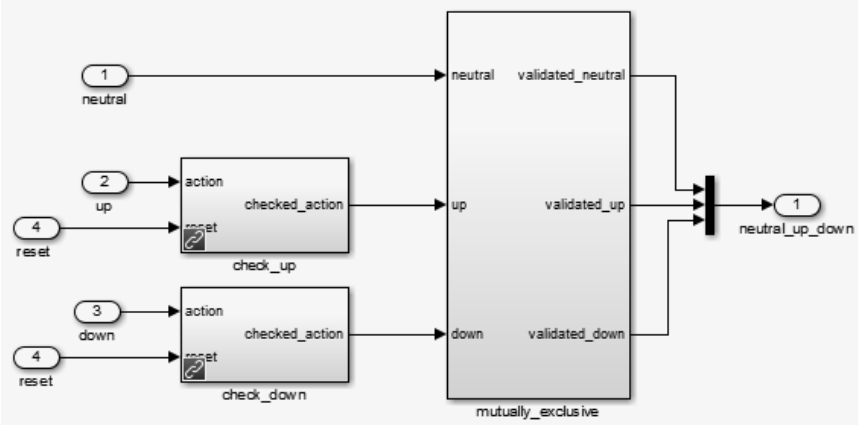
- The model allows the `down` command only when the window is not completely opened.
- The model allows the `up` command only when the window is not completely closed and no object is detected.

The third activity diagram process checks that the software sends only one of the three commands (`neutral`, `up`, `down`) to the controller. In an actual implementation, both `up` and `down` can be simultaneously true (for example, because of switch bouncing effects).

From the power_window_control_system subsystem, this is the validate_driver_state subsystem:



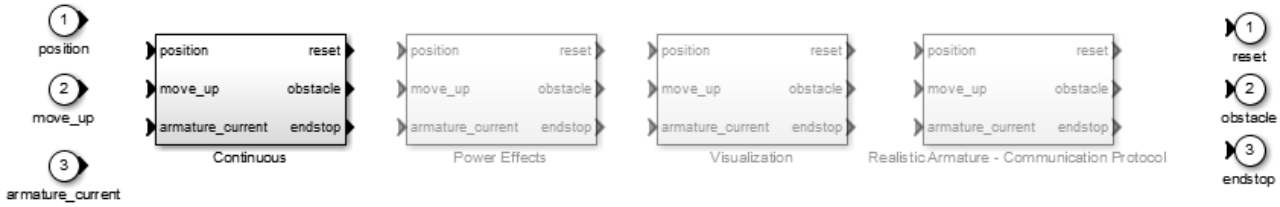
From the power_window_control_system subsystem, this is the validate_passenger_state subsystem:



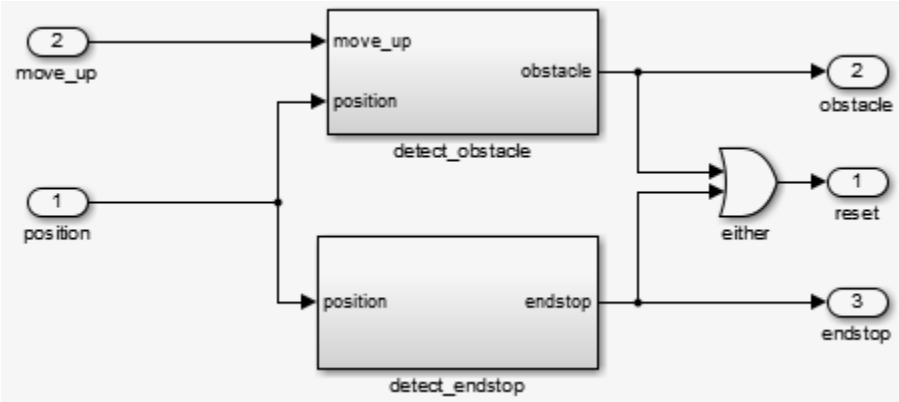
Implementation of Activity Diagram: Detect Obstacle Endstop

For requirements presented as an activity diagram, see “Activity Diagram: Detect Obstacle Endstop” on page 23-7.

In the `slexPowerWindowExample` model, the power_window_control_system/detect_obstacle_endstop block implements this activity diagram in the continuous variant of the Variant Subsystem block. During design iterations, you can add additional variants.



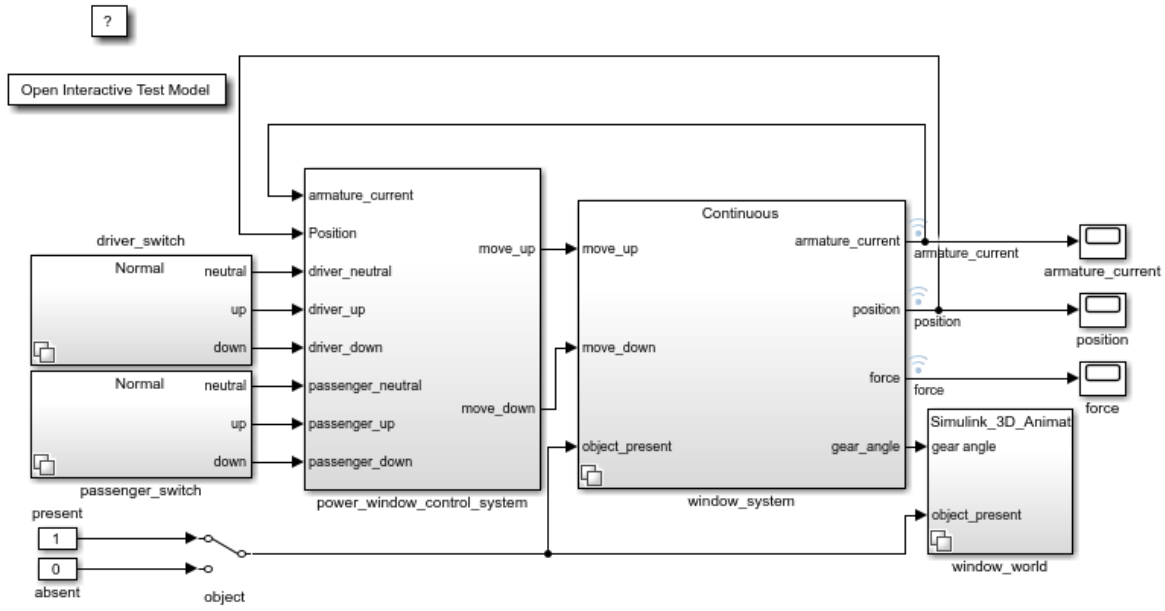
Double-click the `slexPowerWindowExample` model power_window_control_system/detect_obstacle_endstop/Continuous/verify_position block:



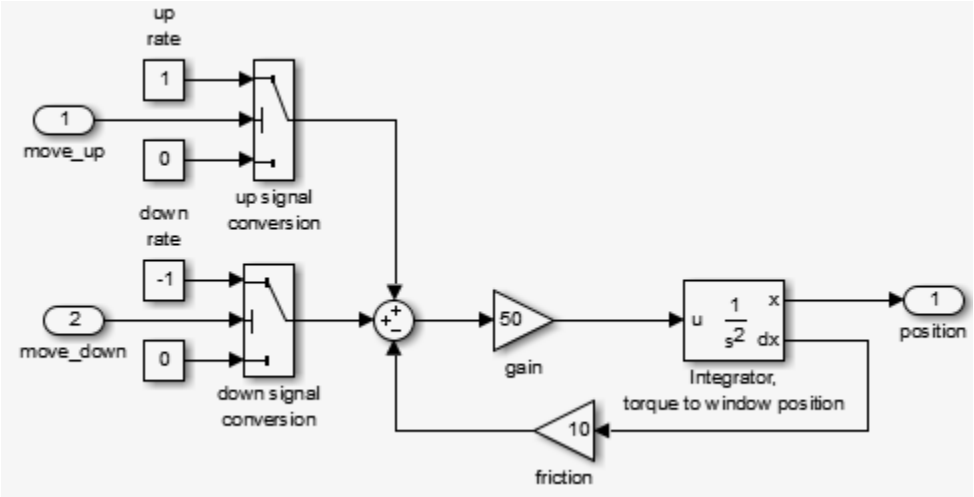
Hybrid Dynamic System: Combine Discrete-Event Control and Continuous Plant

After you have designed and verified the discrete-event control, integrate it with the continuous-time plant behavior. This step is the first iteration of the design with the simplest version of the plant.

In the project, navigate to **Files** and click **Project**. In the **configureModel** folder, run the `slexPowerWindowContinuous` utility to open and initialize the model.



The window_system block uses the Variant Subsystem block to allow for varying levels of fidelity in the plant modeling. Double-click the window_system/Continuous/2nd_order_window_system block to see the continuous variant.



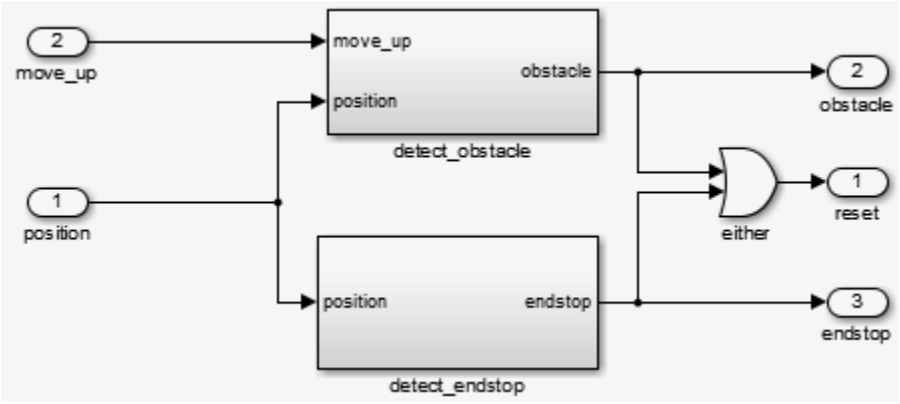
The plant is modeled as a second-order differential equation with step-wise changes in its input:

- When the Stateflow chart generates `windowUp`, the input is 1.
- When the Stateflow chart generates `windowDown`, the input is –1.
- Otherwise, the input is 0.

This phase allows analysis of the interaction between the discrete-event state behavior, its sample rate, and the continuous behavior of the window movement. There are threshold values to generate the window frame top and bottom:

- `endStop`
- Event when an obstacle is present, that is, `obstacle`
- Other events

Double-click the `slexPowerWindowExample` model power_window_control_system/-detect_obstacle_endstop/Continuous/verify_position block to see the continuous variant.



When you run the `slexPowerWindowContinuous` configureModel utility, the model uses the continuous time solver ode23 (Bogacki-Shampine).

A structure analysis of a system results in:

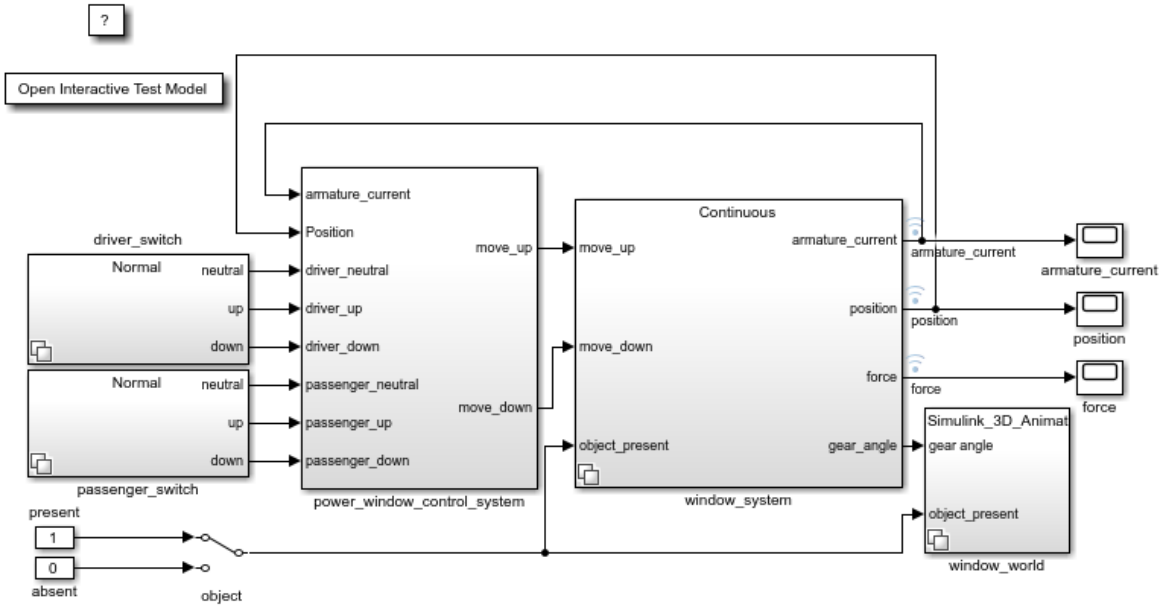
- A functional decomposition of the system
- Data definitions with the specifics of the system signals
- Timing constraints

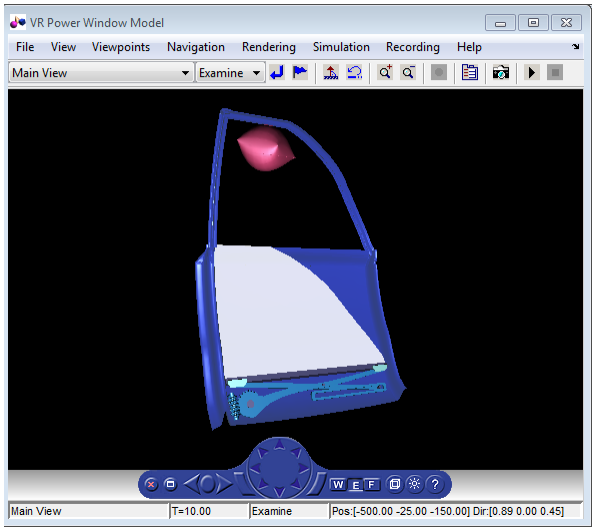
A structure analysis can also include the implementation architecture (outside the scope of this discussion).

The implementation also adds a control mechanism to conveniently switch between the presence and absence of the object.

Expected Controller Response

To view the window movement, in **Project Shortcuts**, double-click `SimHybridPlantLowOrder`. Alternatively, you can run the task `slexPowerWindowContinuousSim`.





The position scope shows the expected result from the controller. After 30 cm, the model generates the `obstacle` event and the Stateflow chart moves into its `emergencyDown` state. In this state, `windowDown` is output until the window lowers by about 10 cm. Because the passenger window up switch is still on, the window starts moving up again and this process repeats. Stop the simulation and open the position scope to observe the oscillating process. In case of an emergency, the discrete-event control rolls down the window approximately 10 cm.

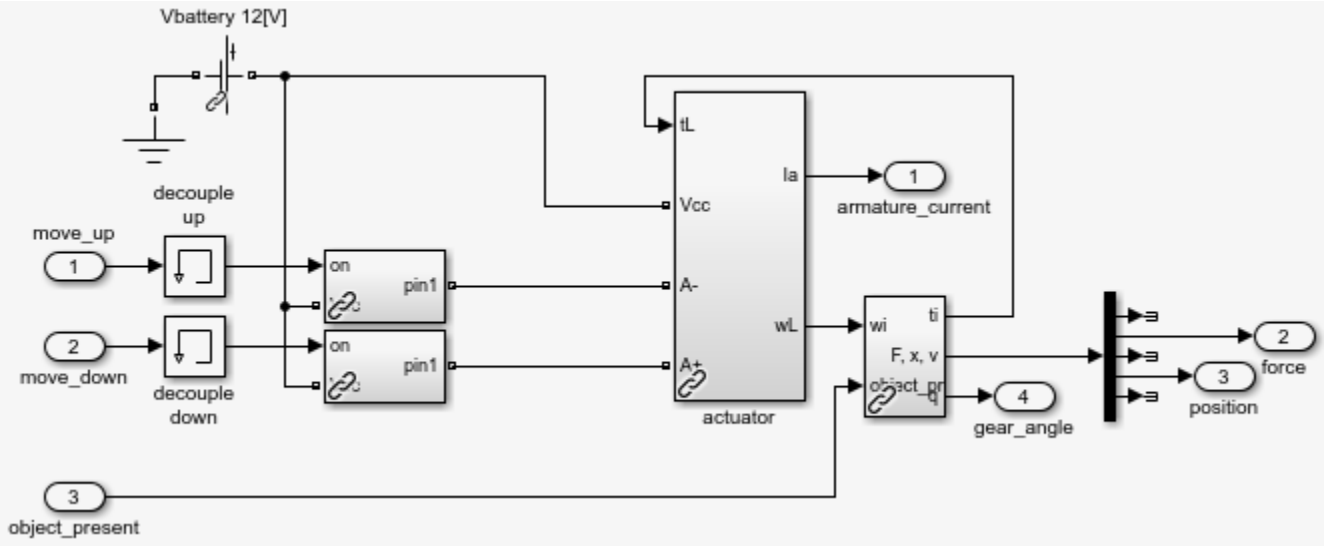
Detailed Modeling of Power Effects

After an initial analysis of the discrete-event control and continuous dynamics, you can use a detailed plant model to evaluate performance in a more realistic situation. It is best to design models at such a level of detail in the power domain, in other words, as energy flows. Several domain-specific MathWorks blocksets can help with this.

To take into account energy flows, add a more detailed variant consisting of power electronics and a multibody system to the window_system variant subsystem.

To open the model and explore the more detailed plant variant, in the project, run configureModel `slexPowerWindowPowerEffects`.

Double-click the `slexPowerWindowExample` model window_system/Power Effects - Visualization/ detailed_window_system block.

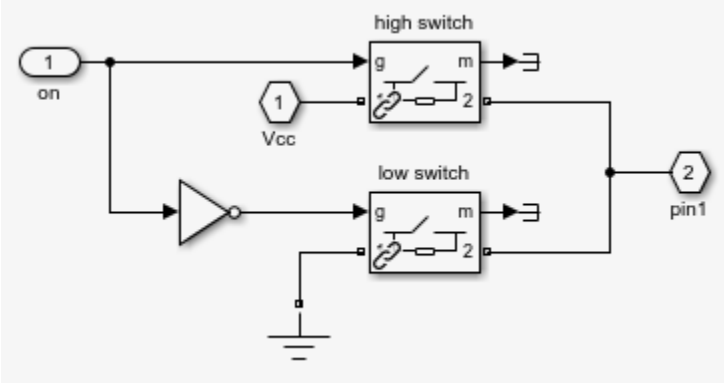


Power Electronics Subsystem

The model must amplify the control signals generated by the discrete-event controller to be powerful enough to drive the DC motor that moves the window.

The amplification modules model this behavior. They show that a switch either connects the DC motor to the battery voltage or ground. By connecting the battery in reverse, the system generates a negative voltage and the window can move up, down, or remain still. The window is always driven at maximum power. In other words, no DC motor controller applies a prescribed velocity.

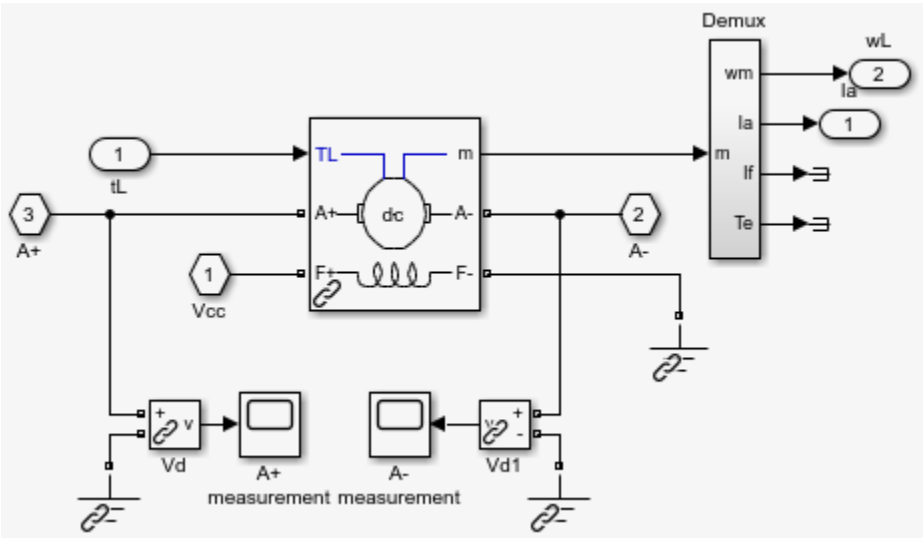
To see the implementation, double-click the `slexPowerWindowExample` model window_system/ Power Effects - Visualization/detailed_window_system/amplification_up block.



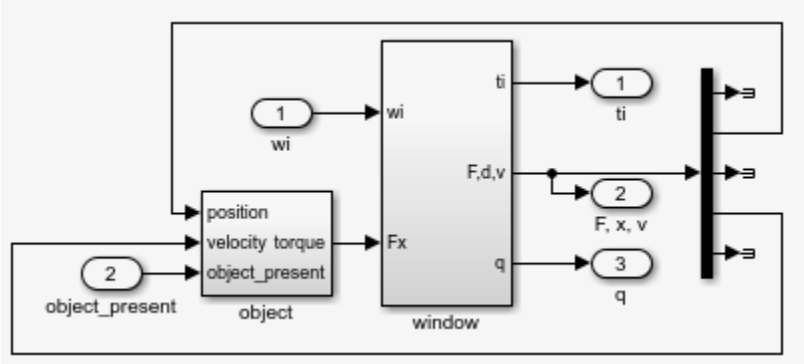
Multibody System

This implementation models the window using Simscape Multibody blocks.

To see the actuator implementation, double-click the `slexPowerWindowExample` model window_system/Power Effects - Visualization/detailed_window_system/actuator block.



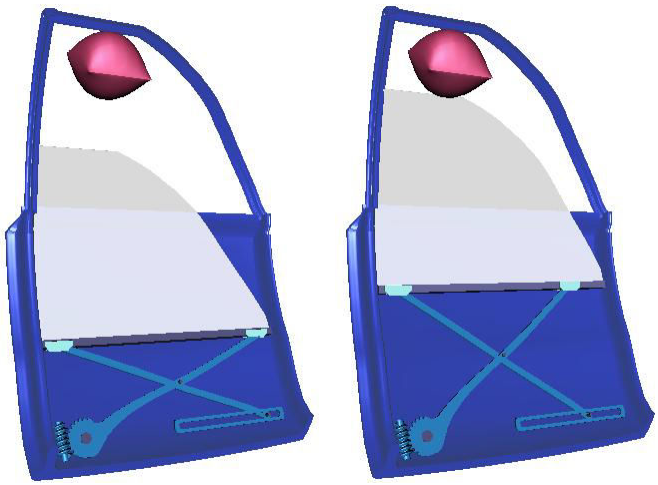
To see the window implementation, double-click the `slexPowerWindowExample` model window_system/Power Effects - Visualization/detailed_window_system/plant block.



This implementation uses Simscape Multibody blocks for bodies, joints, and actuators. The window model consists of:

- A worm gear
- A lever to move the window holder in the vertical direction

The figure shows how the mechanical parts move.



Iterate on the Design

An important effect of the more detailed implementation is that there is no window position measurement available. Instead, the model measures the DC motor current and uses it to detect the endstops and to see if an obstacle is present. The next stage of the system design analyzes the control to make sure that it does not cause excessive force when an obstacle is present.

In the original system, the design removes the obstacle and endstop detection based on the window position and replaces it with a current-based implementation. It also connects the process to the controller and position and force measurements. To reflect the different signals used, you must modify the data definition. In addition, observe that, because of power effects, the units are now amps.

```
PSPEC 1.3.1: DETECT ENDSTOP
  ENDSTOP = ARMATURE_CURRENT > ENDSTOP_MAX

PSPEC 1.3.2: DETECT OBSTACLE
  OBSTACLE = (ARMATURE_CURRENT > OBSTACLE_MAX) and MOVE_UP for 500 ms

PSPEC 1.3.3: ABSOLUTE VALUE
  ABSOLUTE_ARMATURE_CURRENT = abs(ARMATURE_CURRENT)
```

This table lists the additional signal for the Context Diagram: Power Window System data definitions.

Context Diagram: Power Window System Data Definition Changes

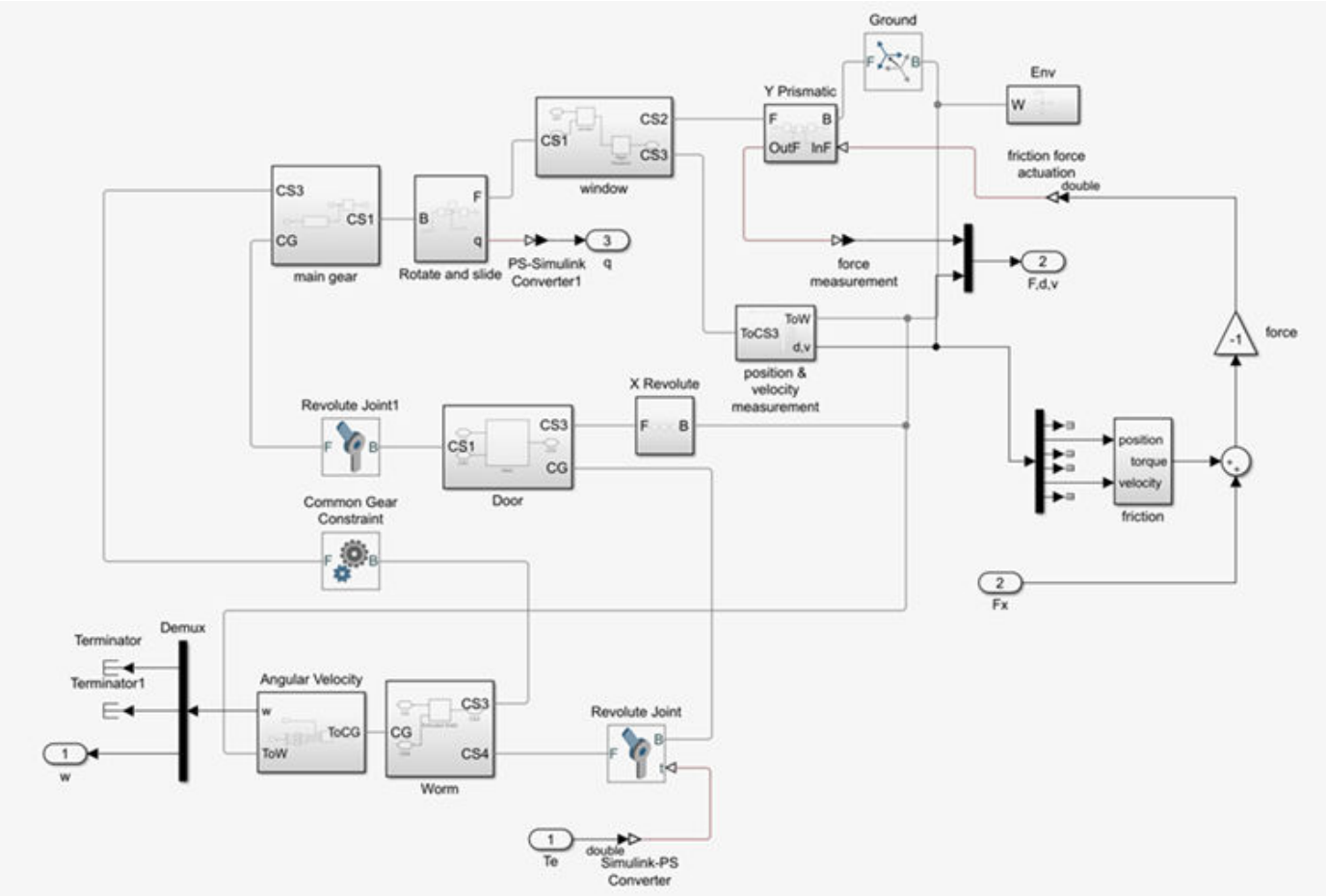
Signal	Information Type	Continuous/ Discrete	Data Type	Values
ARMATURE_CURRENT	Data	Continuous	Real	–20 to 20 A

This table lists the changed signals for the Activity Diagram: Detect Obstacle Endstop data definitions.

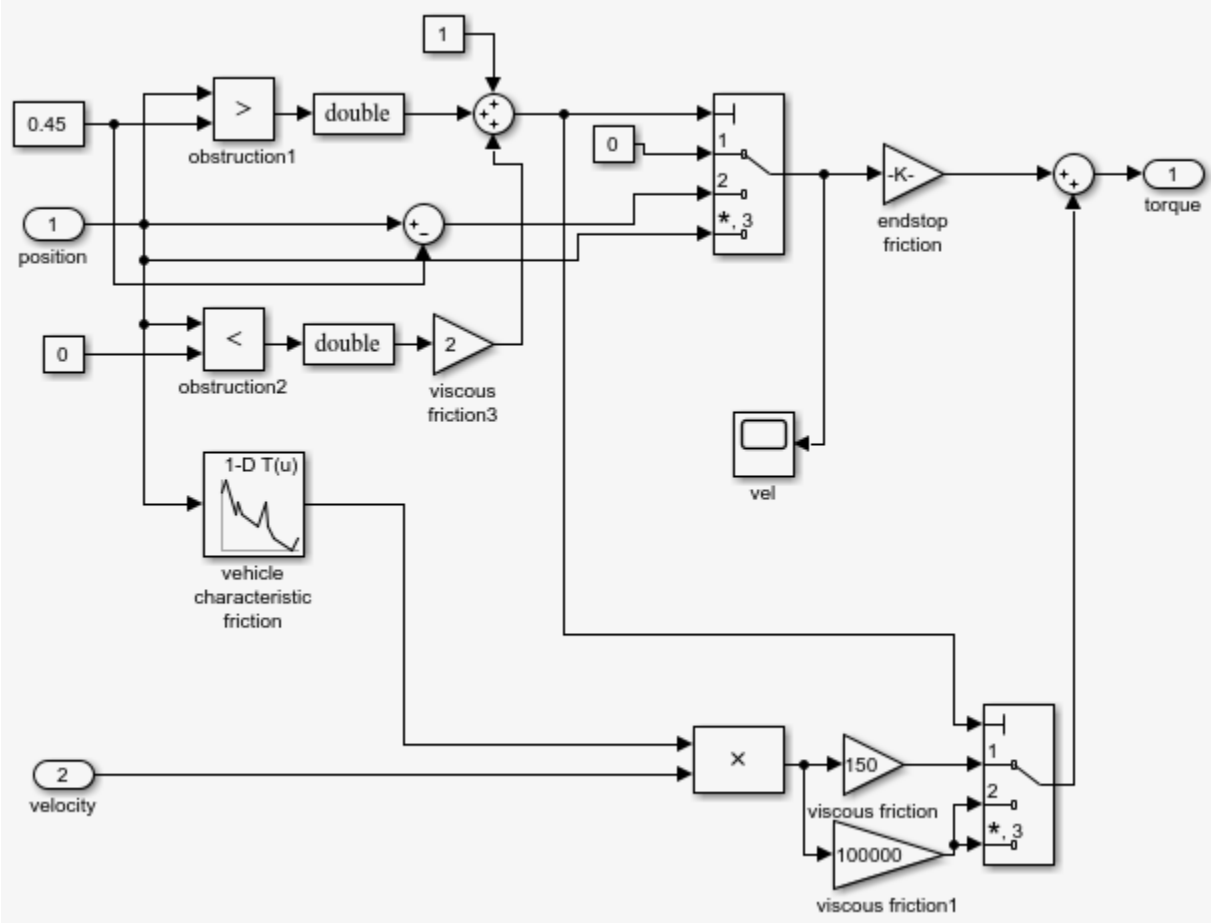
Activity Diagram: Detect Obstacle Endstop Data Definition Changes

Signal	Information Type	Continuous/ Constant	Data Type	Values
ABSOLUTE_ARMATURE_-CURRENT	Data	Continuous	Real	0 to 20 A
ENDSTOP_MAX	Data	Constant	Real	15 A
OBSTACLE_MAX	Data	Constant	Real	2.5 A

To see the window subsystem, double-click the `slexPowerWindowExample` model window_system/Power Effects - Visualization/detailed_window_system/plant/window block.



The implementation uses a lookup table and adds noise to allow evaluation of the control robustness. To see the implementation of the friction subsystem, double-click the `slexPowerWindowExample` model window_system/Power Effects - Visualization/detailed_window_system/plant/window/friction block.



Control Law Evaluation

The idealized continuous plant allows access to the window position for `endStop` and `obstacle` event generation. In the more realistic implementation, the model must generate these events from accessible physical variables. For power window systems, this physical variable is typically the armature current, `Ia`, of the DC motor that drives the worm gear.

When the window is moving, this current has an approximate value of 2 A. When you switch the window on, the model draws a transient current that can reach a value of approximately 10 A. When the current exceeds 15 A, the model activates endstop detection. The model draws this current when the angular velocity of the motor is kept at almost 0 despite a positive or negative input voltage.

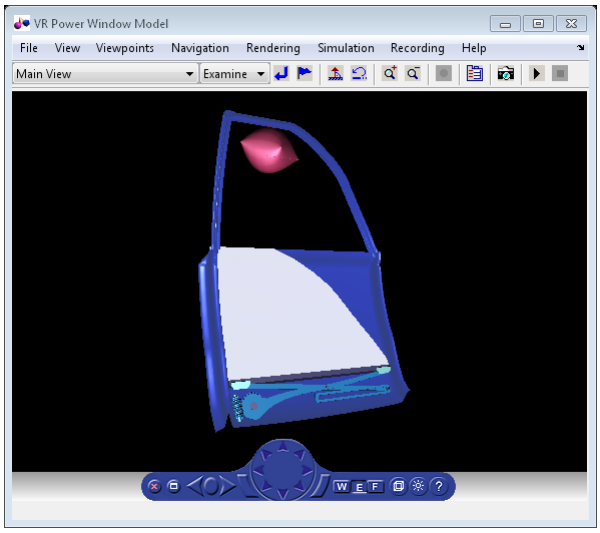
Detecting the presence of an object is more difficult in this setup. Because safety concerns restrict the window force to no more than 100 N, an armature current much less than 10 A should detect an object. However, this behavior conflicts with the transient values achieved during normal operation.

Implement a control law that disables object detection during achieved transient values. Now, when the system detects an armature current more than 2 A, it considers an object to be present and enters the `emergencyDown` state of the discrete-event control. Open the force scope window (measurements are in newtons) to check that the force exerted remains less than 100 N when an object is present and the window reverses its velocity.

In reality, far more sophisticated control laws are possible and implemented. For example, you can implement neural-network-based learning feedforward control techniques to emulate the friction characteristic of each individual vehicle and changes over time.

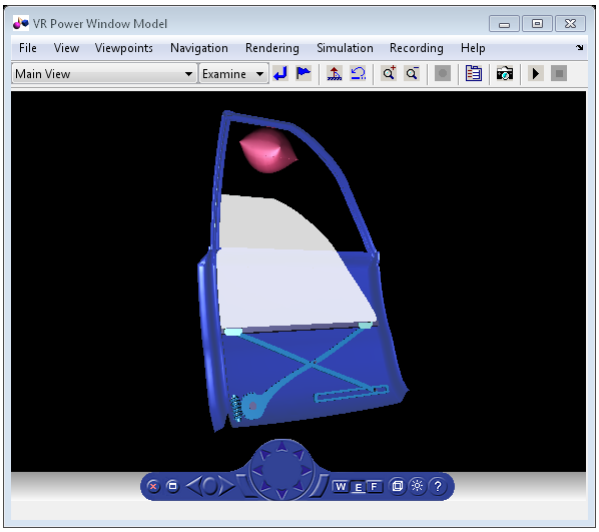
Visualization of the System in Motion

If you have Simulink 3D Animation software installed, you can view the geometrics of the system in motion via a virtual reality world. If the VR Sink block is not yet open, in the `slexPowerWindowExample/window_world/Simulink_3D_Animation View` model, double-click the VR Sink block.



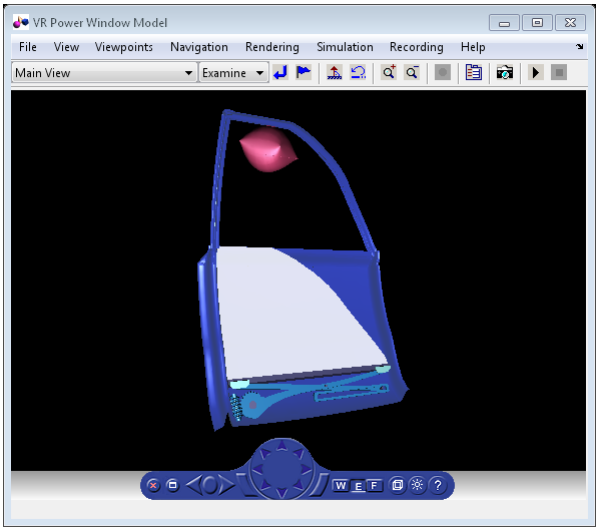
To simulate the model with a stiff solver:

1. In a project, run the task, `slexPowerWindowPowerEffectsSim`. This batch job sets the solver to ode23tb (stiff/TR-BDF2).
2. In the `slexPowerWindowExample` model passenger_switch/Normal block, set the passenger up switch to on.
3. In the `slexPowerWindowExample` model driver_switch/Normal block, set the driver up switch to off.
4. Simulate the model.
5. Between 10 ms and 1 s in simulation time, switch off the slexPowerWindowExample/passenger_switch/Normal block passenger up switch to initiate the auto-up behavior.



6 Observe how the window holder starts to move vertically to close the window. When the model encounters the object, it rolls the window down.

7 Double-click the `slexPowerWindowExample` model passenger_switch/Normal block driver down switch to roll down the window completely and then simulate the model. In this block, at less than one second simulation time, switch off the driver down switch to initiate the auto-down behavior.



8 When the window reaches the bottom of the frame, stop the simulation.

9 Look at the position measurement (in meters) and at the armature current (Ia) measurement (in amps).

Note The absolute value of the armature current transient during normal behavior does not exceed 10 A. The model detects the obstacle when the absolute value of the armature current required to move the window up exceeds 2.5 A (in fact, it is less than –2.5 A). During normal operation, this is about 2 A. You might have to zoom into the scope to see this measurement. The model detects the window endstop when the absolute value of the armature current exceeds 15 A.

Variation in the armature current during normal operation is due to friction that is included by sensing joint velocities and positions and applying window specific coefficients.

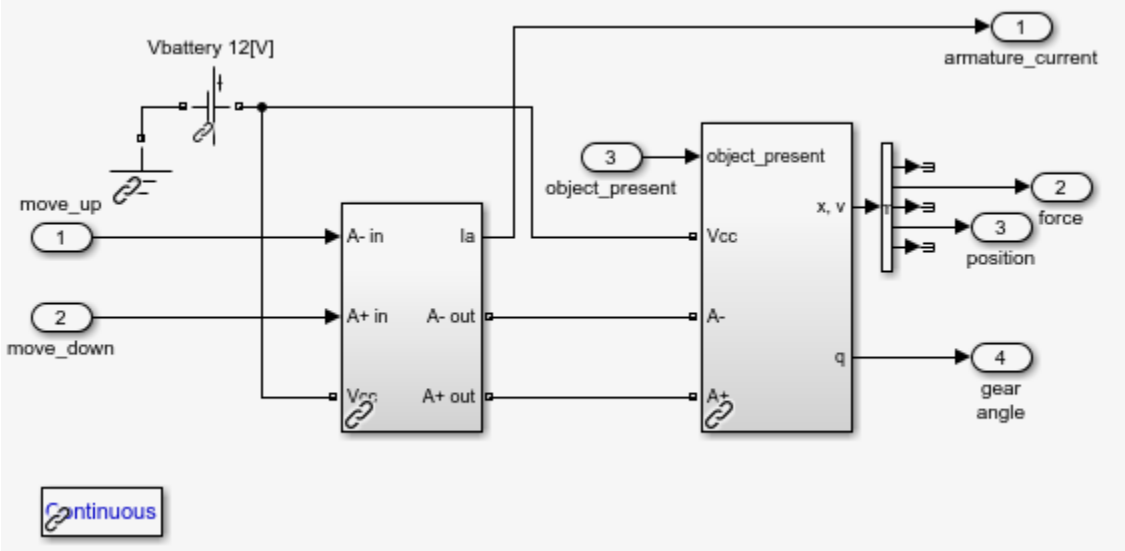
Realistic Armature Measurement

The armature current as used in the power window control is an ideal value that is accessible because of the use of an actuator model. In a more realistic situation, data acquisition components must measure this current value.

To include data acquisition components, add the more realistic measurement variant to the window_system variant subsystem. This realistic measurement variant contains a signal conditioning block in which the current is derived based on a voltage measurement.

To open a model and configure the realistic measurement, in the project, run the configureModel task `slexPowerWindowRealisticArmature`.

To view the contents of the Realistic Armature - Communications Protocol block, double-click the `SlexPowerWindowExample` model window_system/Realistic Armature - Communications Protocol/ detailed_window_system_with_DAQ.



The measurement voltage is within the range of an analog-to-digital converter (ADC) that discretizes based on a given number of bits. You must scale the resulting value based on the value of the resistor and the range of the ADC.

Include these operations as fixed-point computations. To achieve the necessary resolution with the given range, 16 bits are required instead of 8.

Study the same scenario:

1. In the slexPowerWindowExample/passenger_switch/Normal block, set the passenger up switch.
2. Run the simulation.
3. After some time, in the slexPowerWindowExample/passenger_switch/Normal block, switch off the passenger up switch.
4. When the window has been rolled down, click the slexPowerWindowExample/passenger_switch/ Normal block driver down switch.

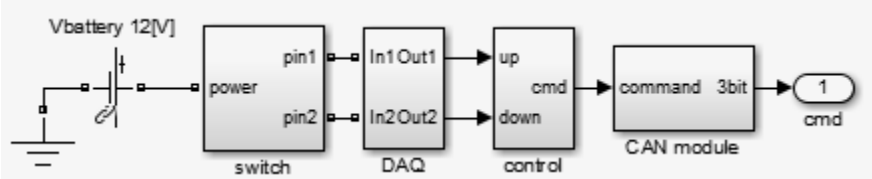
5. After some time, switch off the slexPowerWindowExample/passenger_switch/Normal block driver down switch.
6. When the window reaches the bottom of the frame, stop the simulation.
7. Zoom into the armature_current scope window and notice the discretized appearance.

Communication Protocols

Similar to the power window output control, hardware must generate the input events. In this case, the hardware is the window control switches in the door and center control panels. Local processors generate these events and then communicate them to the window controller via a CAN bus.

To include these events, add a variant containing input from a CAN bus and switch components that generate the events delivered on the CAN bus to the driver switch and passenger switch variant subsystems. To open the model and configure the CAN communication protocols, run the configureModel task, `slexPowerWindowCommunicationProtocolSim`.

To see the implementation of the switch subsystem, double-click the slexPowerWindowExample/driver_switch/Communication Protocol/driver window control switch block.



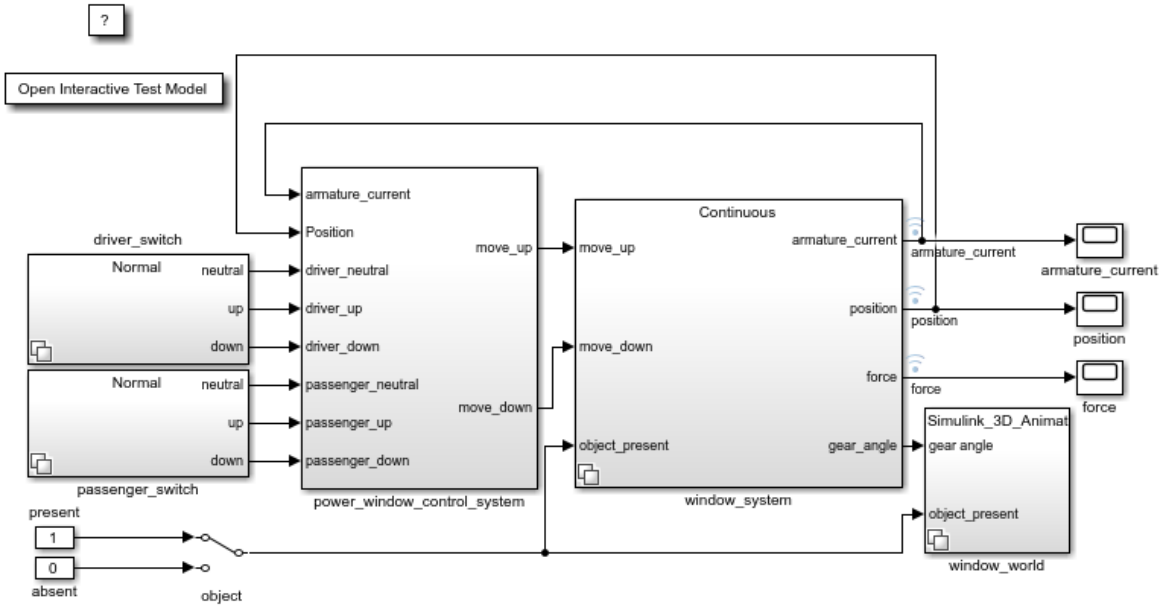
Observe a structure that is very similar to the window control system. This structure contains a:

- Plant model that represents the control switch
- Data acquisition subsystem that includes, among other things, signal conditioning components
- Control module to map the commands from the physical switch to logical commands
- CAN module to post the events to the vehicle data bus

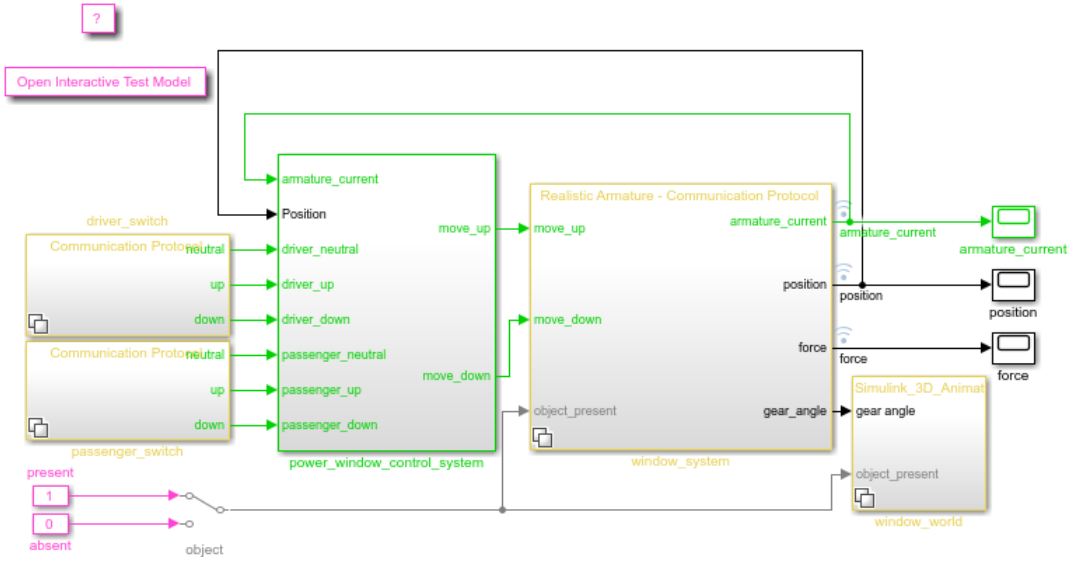
You can add communication effects, such as other systems using the CAN bus, and more realism similar to the described phases. Each phase allows analysis of the discrete-event controller in an increasingly realistic situation. When you have enough detail, you can automatically generate controller code for any specific target platform.

Automatic Code Generation for Control Subsystem

You can generate code for the designed control model, `slexPowerWindowExample`.



1 Display the sample rates of the controller. In the Simulink Editor, from the **Debug** tab, select **Information Overlays > Sample TIme > Colors**. Observe that the controller runs at a uniform sample rate.



2 Right-click the power_window_control_system block and select **C/C++ Code > Build This Subsystem**.

References

Mosterman, Pieter J., Janos Sztipanovits, and Sebastian Engell, “Computer-Automated Multiparadigm Modeling in Control Systems Technology,” *IEEE Transactions on Control Systems Technology*, Vol. 12, Number 2, 2004, pp. 223–234.

See Also

Related Examples

- “Power Window Control Project”

More About

- “Project Management”

Schedule Editor

What are Partitions?

Partitions are components of a model that execute independently as atomic tasks. In multi-tasking models, partitions are created from model components. Periodic partitions are scheduled as a function of sample times in a model. In export-function models, the root function-call inputs are defined as partitions. These partitions have a schedule associated with it, which tells order in which the partitions execute.

With partitions, you can separate parts of the model which you can explicitly control. You can think of periodic partitions as components that run at specific rates in a model. Aperiodic partitions are components that run at specified hit times. The schedule of these partitions have an impact on simulation and code generation.

The Schedule Editor enables you to partition the model and interact with those partitions. The Schedule Editor shows partitions, the connections between them, and the ordering of the partitions

There are three types of partitions:

Types of Partitions	Image	Description
Implicit		Automatically created by Simulink. Blocks running at the base rate show up as an implicit partition.
Periodic		User-defined partitions from the atomic subsystems and/or Model blocks. Periodic partitions can also be defined by export-functions. These partitions execute based on their sample time and thus their execution is periodic.
Aperiodic		Aperiodic partitions are partitions which have no constraints and can be made to execute at any time. Specify the `Trigger` in the **Property Inspector** of the Schedule Editor, at which you want to run the aperiodic partition. You can also use events in the Schedule Editor to schedule execution of the aperiodic partitions.

The blocks running at the base rate in the model is shown as an implicit partition in the Schedule Editor. The base rate is the fastest discrete rate in the model. *D1* denotes the base rate. *D1* annotation also appears in the Timing Legend. The *D1* partition or implicit partition always remains the first discrete partition in the order.

The default partitions that are already present in the model are also implicit partitions.

The partition colors match their rate.

See Also

More About

- “Create Partitions” on page 24-4
- “Using the Schedule Editor” on page 24-11
- “Using the Schedule Editor” on page 24-11
- **Schedule Editor**

Create Partitions

Partitioning a Model

Partitions are components of a model that execute independently as atomic tasks. In multi-tasking models, partitions are created from model components. Periodic partitions are scheduled as a function of sample times in a model. In export-function models, the root function-call inputs are defined as partitions. These partitions have a schedule associated with them, which tells what order the partitions execute.

With partitions, you can separate parts of the model which you can explicitly control. You can think of periodic partitions as components that run at specific rates in a model. Aperiodic partitions are components that run at specified hit times or specified events. The schedule of these partitions have an impact on simulation and code generation.

The Schedule Editor enables you to partition the model and interact with those partitions. The Schedule Editor shows partitions, the connections between them, and the order of the partitions.

There are three types of partitions:

Types of Partitions	Image	Description
Implicit		Automatically created by Simulink. Blocks running at the base rate show up as an implicit partition.
Periodic		User-defined partitions from the atomic subsystems and/or Model blocks. Periodic partitions can also be defined by export-functions. These partitions execute based on their sample time and thus their execution is periodic.
Aperiodic		Aperiodic partitions are partitions which have no constraints and can be made to execute at any time. Specify the `hit times` in the **Trigger** field of the **Property Inspector** of the Schedule Editor, at which you want to run the aperiodic partition. You can also use events in the Schedule Editor to schedule execution of the aperiodic partitions.

The blocks running at the base rate in the model is shown as an implicit partition in the Schedule Editor. The base rate is the fastest discrete rate in the model. *D1* denotes the base rate. *D1* annotation also appears in the Timing Legend. The *D1* partition or implicit partition always remains the first discrete partition in the order.

The default partitions that are already present in the model are also implicit partitions.

The partition colors match their rate.

Create Partitions from a Rate-Based Model

Partitioning is enabled only on multitasking, fixed-step and variable-step solver models. To choose multitasking execution mode, In Solver selection on the **Solver** pane, select the **Type** to be `Fixed-step` or `Variable-step`. Select the **Treat each discrete rate as a separate task** check box on the **Solver** pane of the Configuration Parameters dialog box. For more information on multitasking execution mode, see “Time-Based Scheduling and Code Generation” (Embedded Coder).

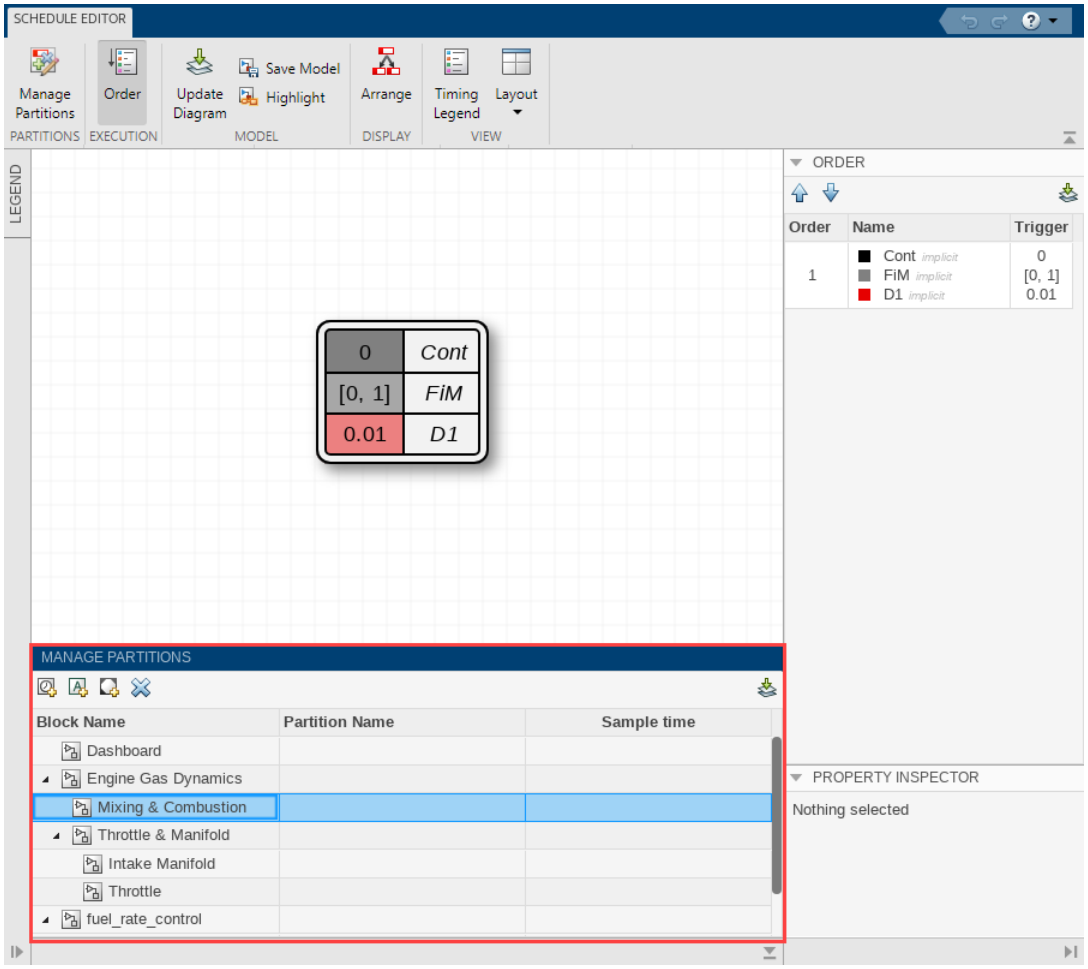
As a best practice, enable the **Automatically handle rate transition for data transfer** setting in the **Solver** pane. When you check **Automatically handle rate transition for data transfer**, Simulink inserts Rate Transition blocks between blocks when rate transitions are detected. Simulink handles rate transitions for asynchronous and periodic tasks. Simulink adds the hidden blocks configured to ensure data integrity and determinism for data transfers. When you check **Treat each discrete rate as a separate task**, Simulink selects multitasking execution for models operating at different rates. It also specifies that groups of blocks with the same execution priority are processed through each stage of simulation (for example, calculating output and updating states) based on task priority.

To see default partitions in the Schedule Editor, open the Schedule Editor. On the **Modeling** tab, click **Schedule Editor**. Update the diagram by clicking the  icon on the toolstrip. The default partitions in the model are called implicit partitions. You can also create partitions in the model through an atomic subsystem or a model block.

Create Partitions Using Manage Partitions

In the Schedule Editor, use the **Manage Partitions** panel to create partitions. The changes made in **Manage Partitions** are applied to the model to create partitions after updating the diagram. **Manage Partitions** shows the model hierarchy with the Subsystem blocks and Model blocks which can be explicitly partitioned.

To create partitions from Subsystem blocks, select the subsystems and click the  . To create partitions from Model blocks, select the Model blocks and click  . Enter partition names in the column **Partition Name**, and sample times in the column **Sample Time**. Repeat the steps for all the subsystems and Model blocks in the model that you want to partition. Update the diagram to see the created partitions in the Schedule Editor. The partitions appear in the graph with their deduced data dependencies and order.



Note Creating partitions using the **Manage Partitions** panel makes changes to the subsystem or model block parameters.

The following example shows how to configure a model for partitioning and create partitions by using **Manage Partitions** panel in the Schedule Editor.

1. Open the model.

   ```
   sldemo_fuelsys
   ```

2. Open the Schedule Editor.

   Open the Schedule Editor from the Simulink **View** menu. To see the default partitions, click **Update Diagram** in the Schedule Editor. Two implicit partitions, created automatically by Simulink, are seen in the Schedule Editor.

3. Create partitions.

   Open the **Manage Partitions** panel. In the panel, expand the `fuel_rate_control` subsystem.

   Select the `airflow_calc` subsystem and click  . To change the default partition name and sample time, click the default name and sample time.

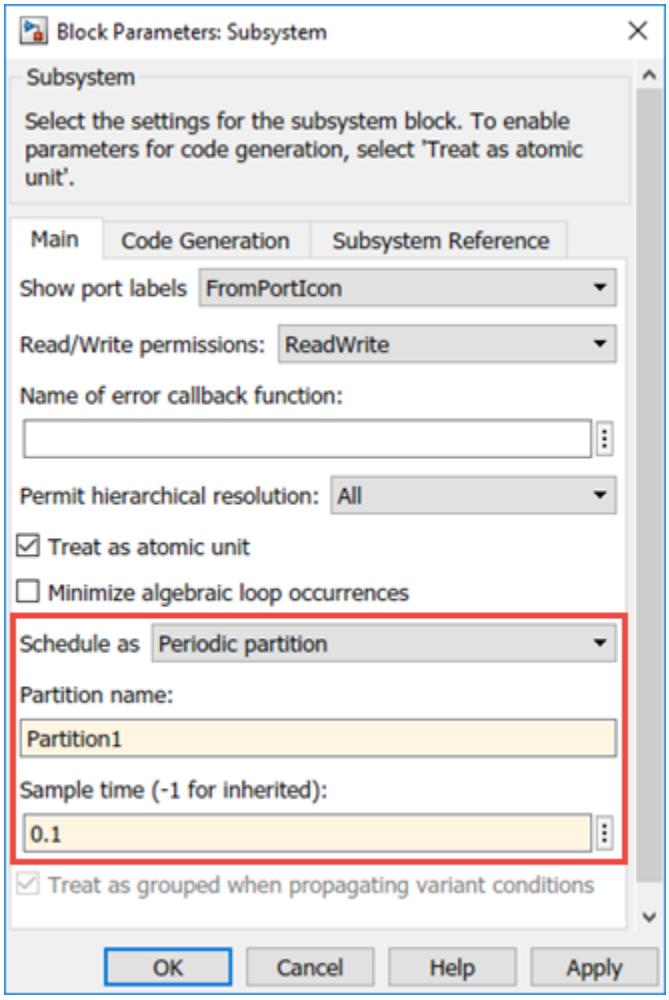
To create a partition for the `fuel_calc` subsystem, select the `fuel_calc` subsystem and click  .

The **Manage Partitions** panel gives you the default partition names and sample times.

Update the diagram to see the newly created partitions.

Create Partitions from Atomic Subsystem Blocks

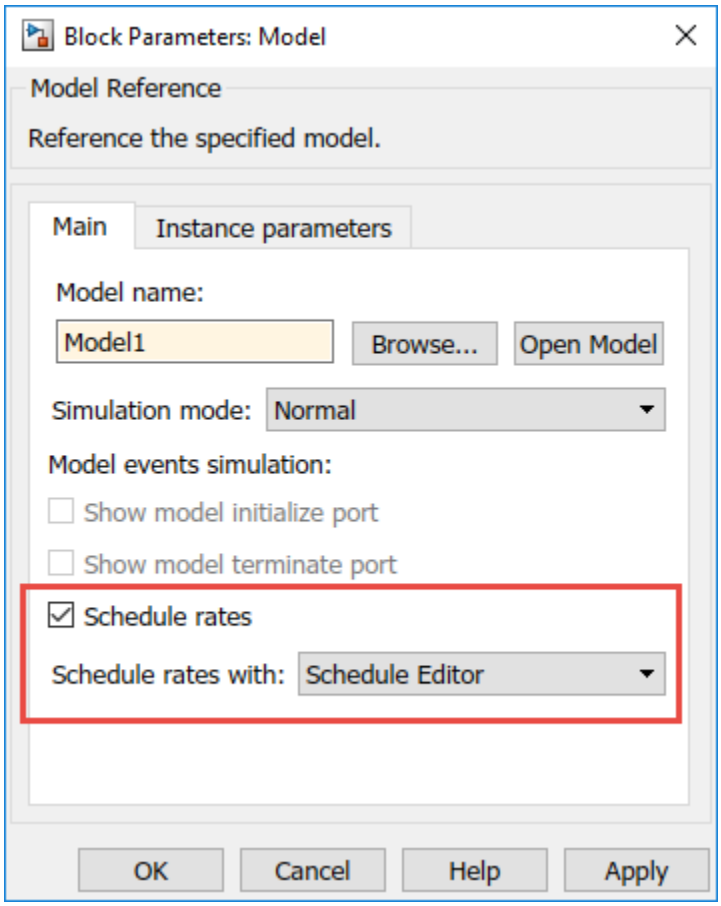
You can partition only an atomic subsystem. An atomic subsystem is treated by Simulink as a unit when determining the execution order of block methods. To create partitions from an atomic subsystem block, go to the Block Parameters dialog box. Select `Periodic partition` from the **Schedule as** drop-down. Give the partition a name and a discrete sample time and click OK. This creates an explicit partition for this block. To see this partition in the Schedule Editor, update the diagram. The partition appears in the graph and in the **Order** with the connections based on the signals in the model.



Create Partitions from Model Blocks

To create partitions from a Model block, in the **Model events simulation**, select **Schedule rates** and `Schedule Editor` from the **Schedule rates with** drop-down. When you enable partitioning

from a referenced model, partitions are created from all the Model blocks present in the referenced model. These partitions are scoped by the model block name. To see this partition in the Schedule Editor, update the diagram. The partitions appear in the graph and in the **Order** column with the connections based on the design of your model.



Export-Function Partitions

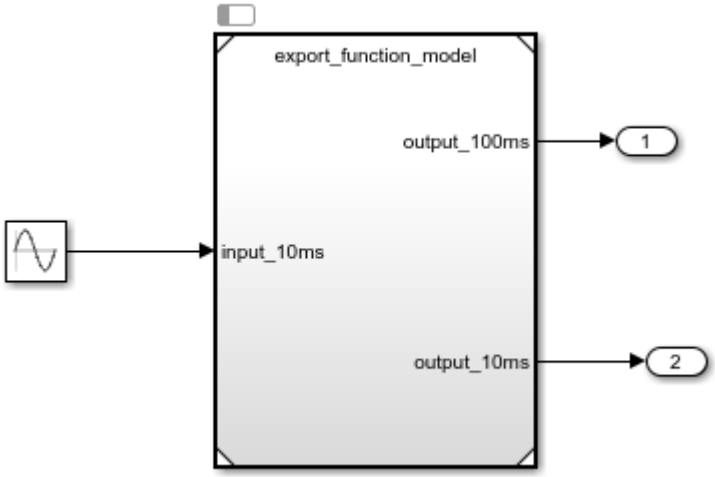
In export-function models, partitions are created from the function calls present in the model. To create partitions from the function calls in the export-function models, reference the export-function model in a top model. Schedule the Model block using the Schedule Editor through the block parameters. Partitions are then created from the function calls and their order is determined by the Schedule Editor.

1. Create an export-function model.
2. Add a Model block that references the export-function model.
3. Set the parameter to partition the export-function model.

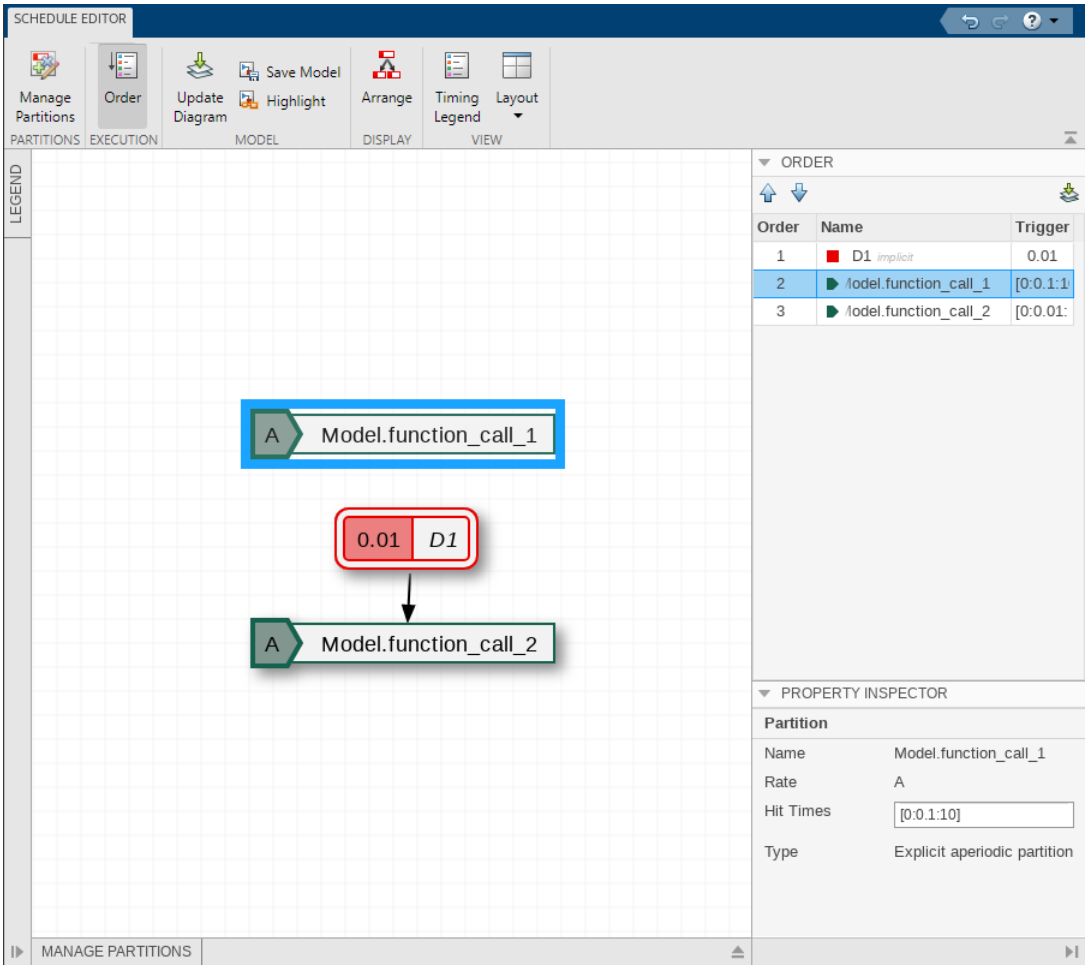
Referencing an export-function model from a Model block allows you to partition the function calls without changing the model itself. To create the model for this example, see “Create an Export-Function Model” on page 10-72. Simulink functions in an export-functions model cannot be scheduled, and do not appear as partitions in the Schedule Editor.

1. Add a model block to a new Simulink model. In the **Model name** box, enter `export_function_model`. To enable the use of the Schedule Editor, configure the model to be multitasking. Open the Model Configuration Parameters. In **Solver selection** on the **Solver** pane, set the **Type** to `Fixed-step`. Check the **Treat each discrete rate as a separate task** and **Automatically handle rate transition for data transfer** parameters. Click **OK**. The model is enabled for partitioning.

   Note Do not convert a subsystem to create a model because it automatically creates Inport blocks.



2. Add Outport blocks to the **output_100ms** and **output_10ms** ports for saving simulation data to MATLAB. For this example, in the `export_function_model`, set the sample time of both the function calls to -1.
3. Add a Sine Wave block to provide data input. Set **Amplitude** to 2 and **Sample time** to `0.01`.
4. To partition the function calls of the export-function model, in the Block Parameters of the model block, select `Schedule Editor` option from the **Schedule Rates with** drop-down menu.
5. Open the Schedule Editor and update the diagram to see the function calls as partitions.



See Also

Schedule Editor

More About

- “Schedule the Partitions” on page 24-15
- “Generate Code from a Partitioned Model” on page 24-22

Using the Schedule Editor

The Schedule Editor is a scheduling tool that represents the components in the model known as partitions, the data connections between them, and the order of those partitions.

Partitions are the components of the model that execute independently as tasks. The data connections between the partitions show the flow of the data between those partitions. The scheduling of these partitions is based on the rates and the events in the model. This schedule is shown in the **Order** table in the Schedule Editor.

Using the Schedule Editor you can:

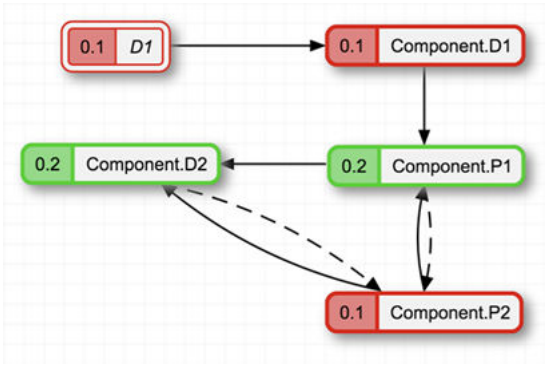
- Create partitions and specify their order
- Edit and analyze the schedule of the executable partitions without disturbing the structure of the model.
- Visualize how Simulink executes partitions

Changes made in the Schedule Editor affect both, simulation and code generation.

Using the Schedule Editor

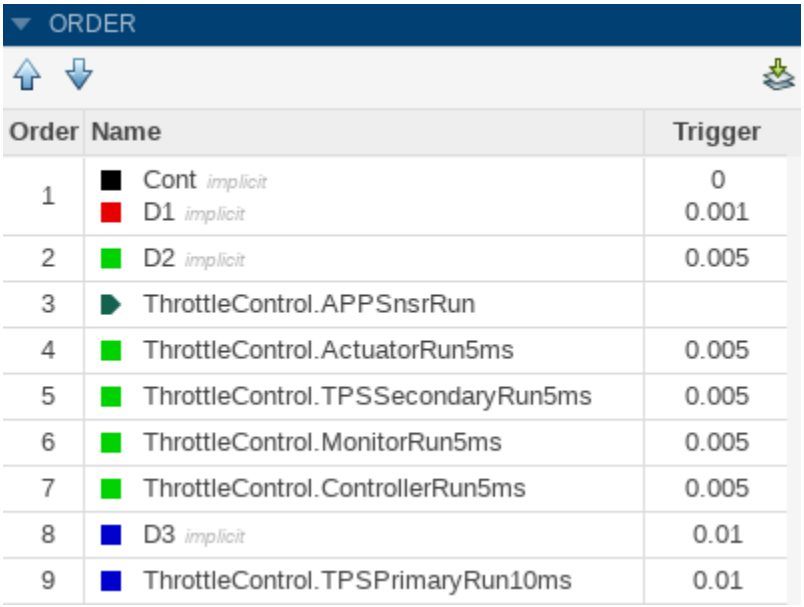
The Schedule Editor consists of two parts representing two different views of partitions in the model.

- A graph that shows the partitions and the data connections between them.



- A table that shows the order in which the partitions execute.

Note Aperiodic partitions are listed in the order of their priority.



Changing one of the views impacts the other.

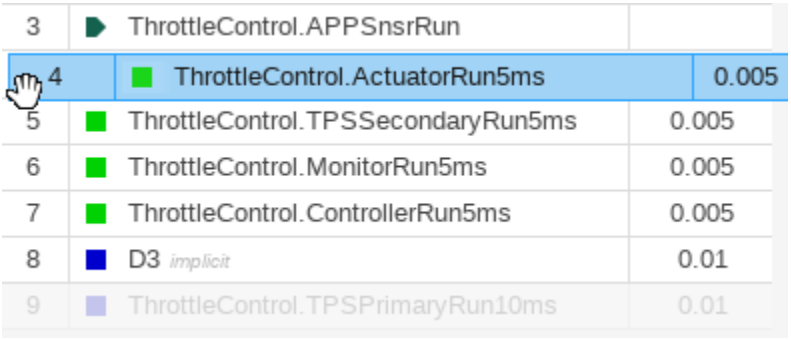
To use the Schedule Editor, on the **Modeling** tab, click **Schedule Editor**. If the model is already partitioned, you can open the Schedule Editor by clicking the badge,  , which appears above the blocks. To see the default partitions present in the model in the Schedule Editor, update the diagram with  icon, on the toolstrip or by selecting **Ctrl+D**. As you create partitions in the model and update the diagram, partitions appear in the Schedule Editor.

To check how the partitions map to the model, right-click the partitions and select **Show Source**. The Simulink model window appears with every block corresponding to the partition highlighted.

Order

The **Order** pane shows the order in which the partitions execute. To change the order, you can drag and drop the partitions. You can also use the **Up** and **Down** arrows on the toolstrip. Partitions are sorted based on their rates. You can only reorder the partitions with the same rate. Clicking a partition in the **Order**, highlights the corresponding partition in the graph. On changing the order, the connections that are affected by this specified change get highlighted.

Note Aperiodic partitions are listed in the order of their priority.



Connections

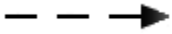
Connections between the partitions show data dependencies. You can right-click the connections between the partitions to change the constraints on data connections. The different types of connections illustrate how the partitions behave with each other.

The types of connections are:

- **Dependency** – Indicates that the source always runs before the destination. The dependency connection is a solid line.

- **Delay** – Indicates that the destination runs before the source. When the destination runs before the source, a scheduling delay is introduced. The delay connection is a dashed line.

You can put these types of constraints on connections:

- **Allow Delay** – Inserts a delay when required. When you specify this constraint for a connection, Simulink inserts a delay for that connection only when necessary. The unlock icon on the connections signifies an allowed delay. When you select this constraint on a connection, Simulink prefers these connections to be turned into a delay if necessary over other connections.

  This constraint is displayed as one of these options.

- **Prevent Delay** – Prevents delay from being inserted in the connection. When you specify this constraint for a connection, Simulink ensures that the connection always remains as a dependency. The lock icon on the connection indicates that the connection is locked to be a dependency and is not changed to a delay.

See Also

Schedule Editor

More About

- “Create Partitions” on page 24-4
- “Schedule the Partitions” on page 24-15
- “Generate Code from a Partitioned Model” on page 24-22

Schedule the Partitions

These two examples walk through the workflow of partitioning a model, scheduling the partitions and analyzing the simulations before and after editing the schedule of the partitions.

Schedule an Export-Function Model Using the Schedule Editor

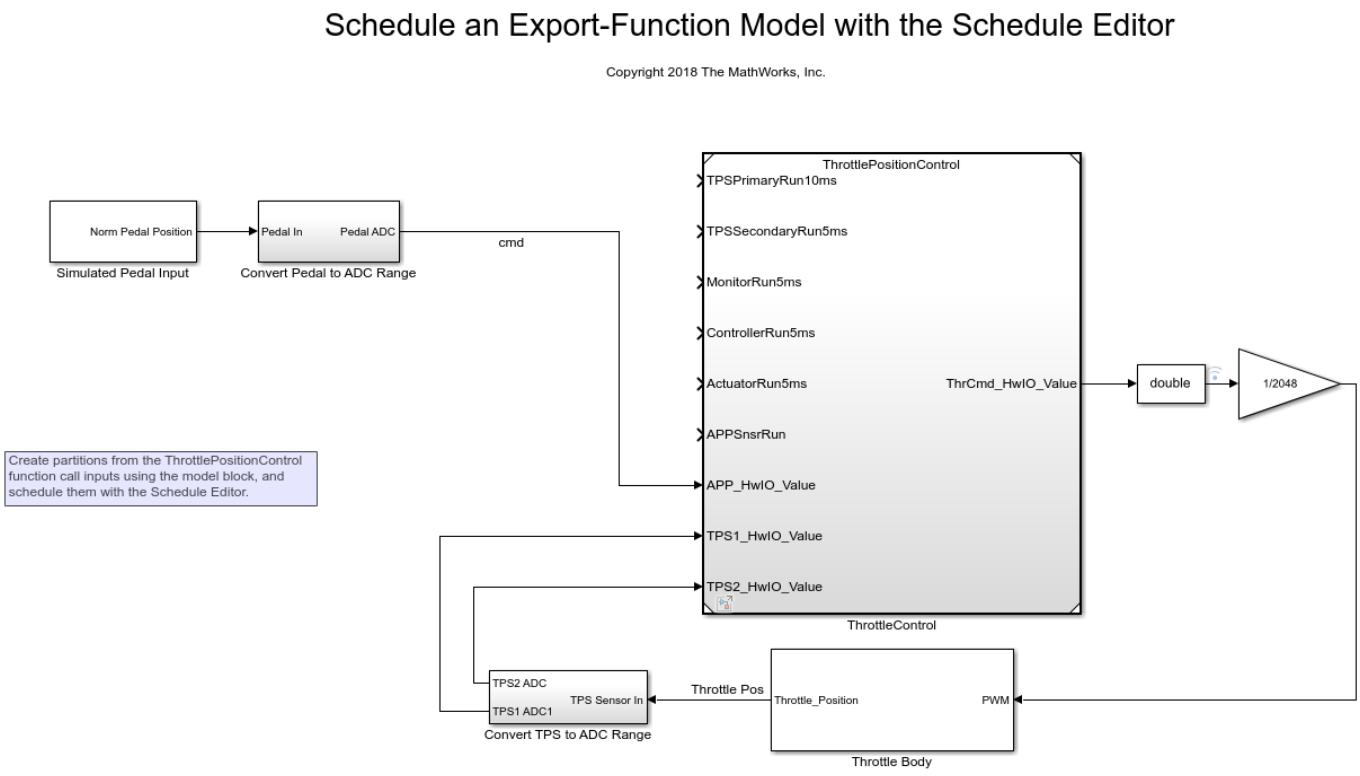
This example shows how to view and edit the order of function-calls in an export-function model using the Schedule Editor. As in all export-function models, the desired functionality is modeled as function-call subsystems. These function-call subsystems define the partitions that are scheduled by the Schedule Editor.

With the Schedule Editor, you can easily view and edit the schedule of the function-calls. The behavior of the system depends on the order of these partitions. In this example, we change the order and observe its effects on the behavior of the system by simulating the model. To see the impact of editing the schedule on the simulation, we compare the model simulations before and after scheduling.

Create Partitions from Referenced Export-Function Model

To view and edit the schedule of the export-function model, reference the model.

```
open_system('ThrottlePositionControlTop.slx');
```



`ThrottlePositionControl` is the referenced export-function model. By default, each function has an input port that can be used to trigger these functions. The Schedule Editor automatically handles these ports. To use the Schedule Editor, set the Schedule Rates With parameter to Schedule Editor.

```
set_param('ThrottlePositionControlTop/ThrottleControl','ScheduleRatesWith','Schedule Editor');
```

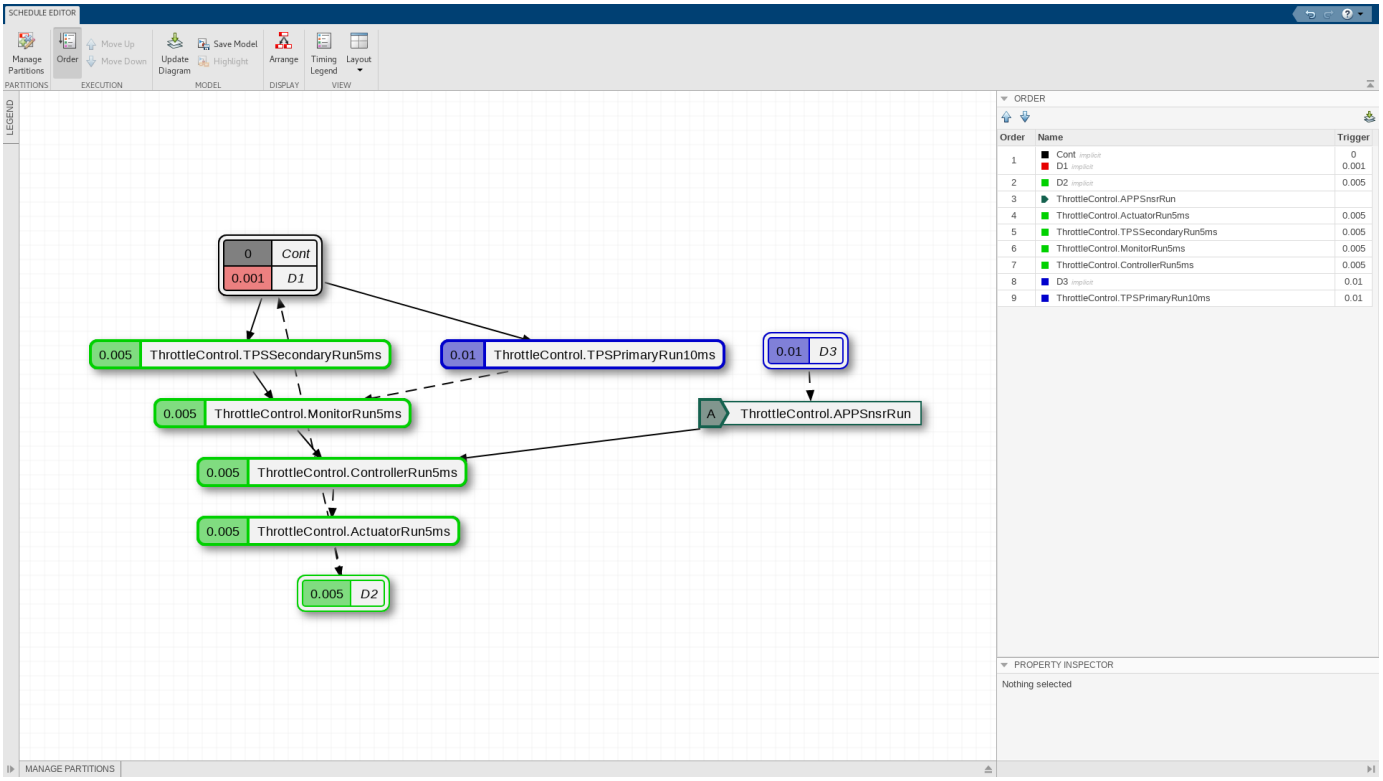
Establish a Simulation Baseline

To observe the impact of scheduling on the model behavior, establish a baseline by simulating the model before editing the schedule. Simulate the model.

```
sim('ThrottlePositionControlTop');
```

Open the Schedule Editor

To open the Schedule Editor, click **Schedule Editor** in the **Design** section of the **Modeling** tab. In the Schedule Editor, different components of the model are represented as partitions. Update the diagram to see the partitions. Partitions are the entry-points in the model. The Schedule Editor shows the order and data communications of these partitions. The arrows are data connections between the partitions that show the data flow. The dashed lines indicate that there is a delay because the source runs after the destination. The solid lines indicate that there is no delay as the source runs before the destination.

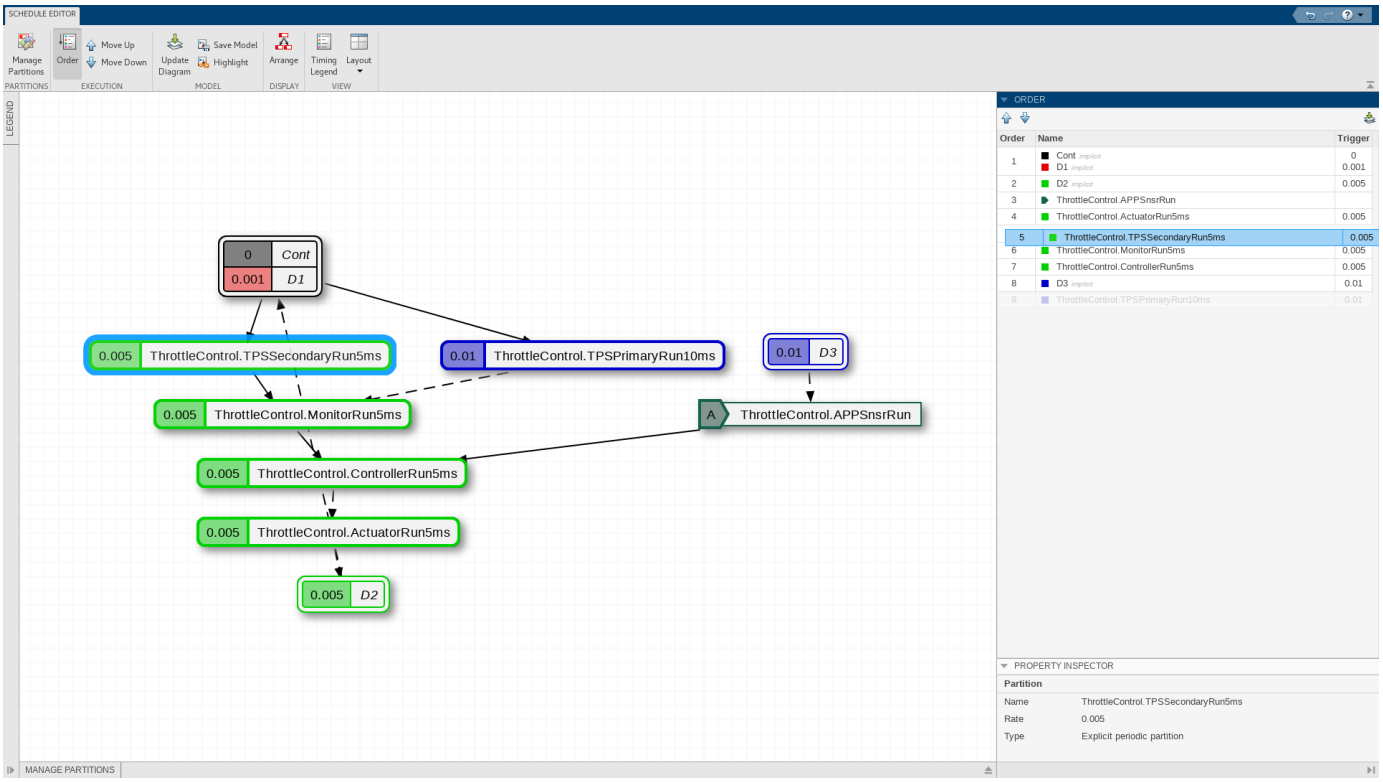


Edit Partition Schedule

The Order pane shows the order the partitions run in at a given time step. Assume that the order of the partitions is in an imperfect state. In this case, to remove the delay, you want to run the `ThrottleControl.ActuatorRun5ms` partition after the `ThrottleControl.ControllerRun5ms` partition.

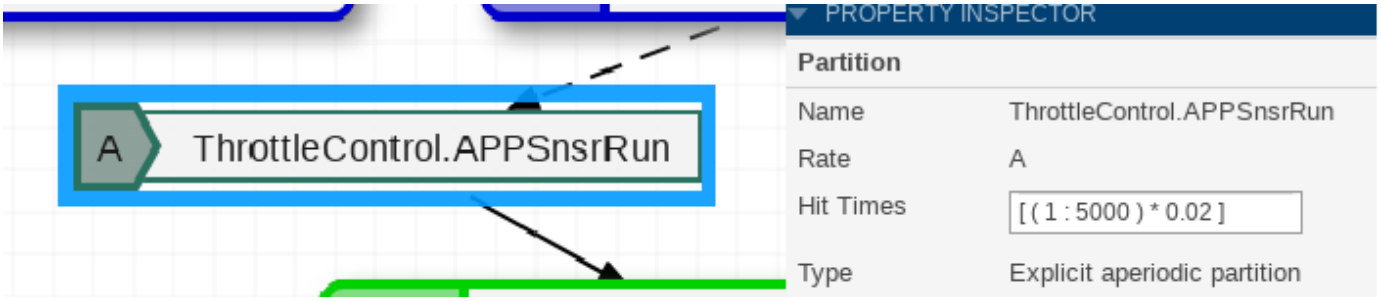
Drag `ThrottleControl.ActuatorRun5ms` after the `ThrottleControl.ControllerRun5ms` in the Order pane. Observe that the delay between the `ThrottleControl.ControllerRun5ms` and the `ThrottleControl.ActuatorRun5ms` partitions changes to a dependency. Observe that now

there is no delay between the executions of `ThrottleControl.ControllerRun5ms` and `ThrottleControl.ActuatorRun5ms`.



Schedule the Execution of Aperiodic Partitions

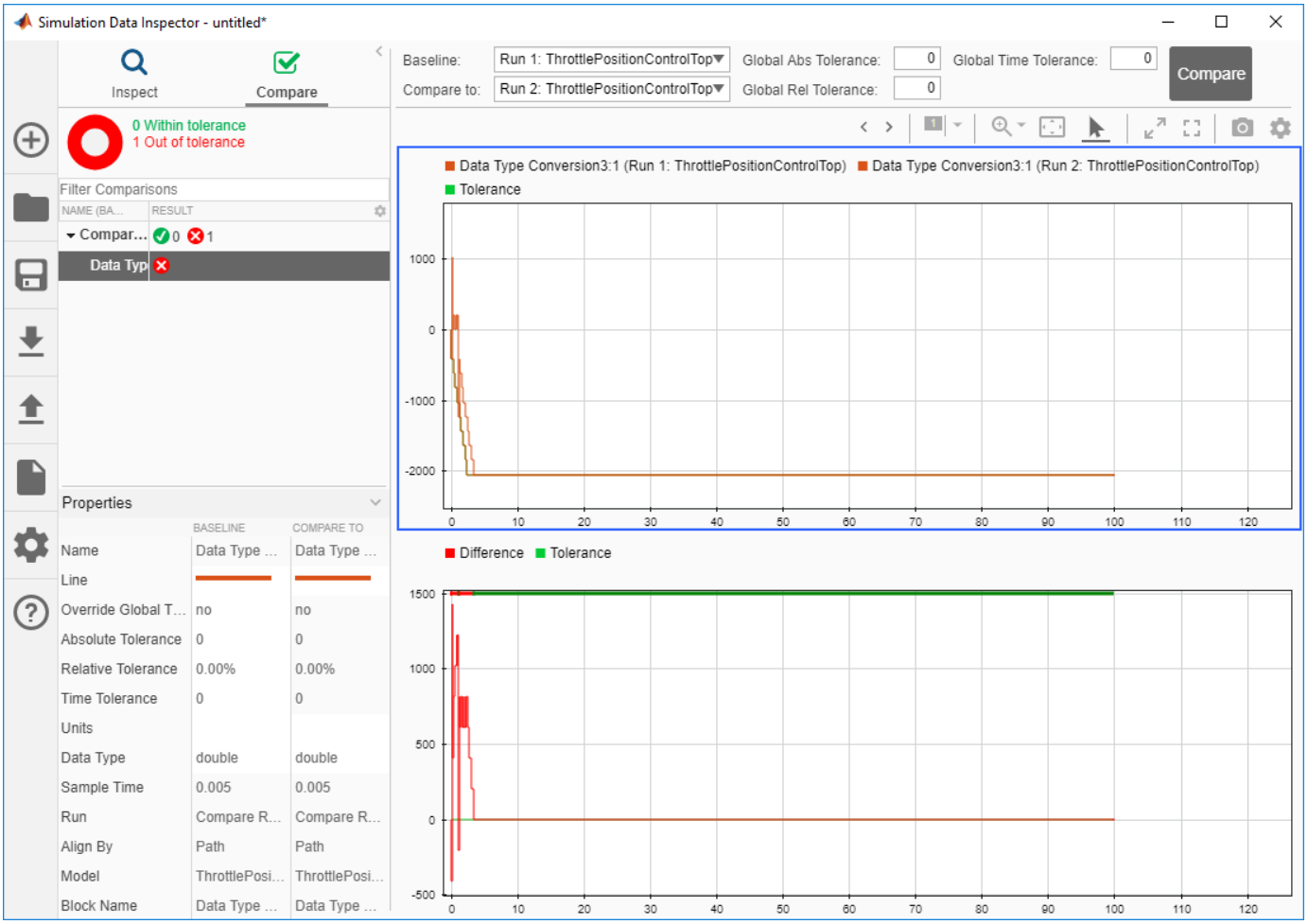
The export-function model contains an unconstrained partition, `AccelerationPedalPositionSensor`. Suppose you want to schedule an unconstrained partition to simulate as if it were discrete. Schedule `ThrottleControl.AppSnsrRun` partition to run at [0:0.02:100] to observe its behavior at different instances of time. Click the unconstrained partition and enter [(1:5000)*.02] for Hit Times in the Property Inspector.



Compare the Runs in Simulation Data Inspector

Now, simulate the model with the changed schedule.

Open the Simulation Data Inspector. Select the two runs and compare. You can see how changing the schedule impacts the model behavior.



Schedule a Rate-Based Model Using the Schedule Editor

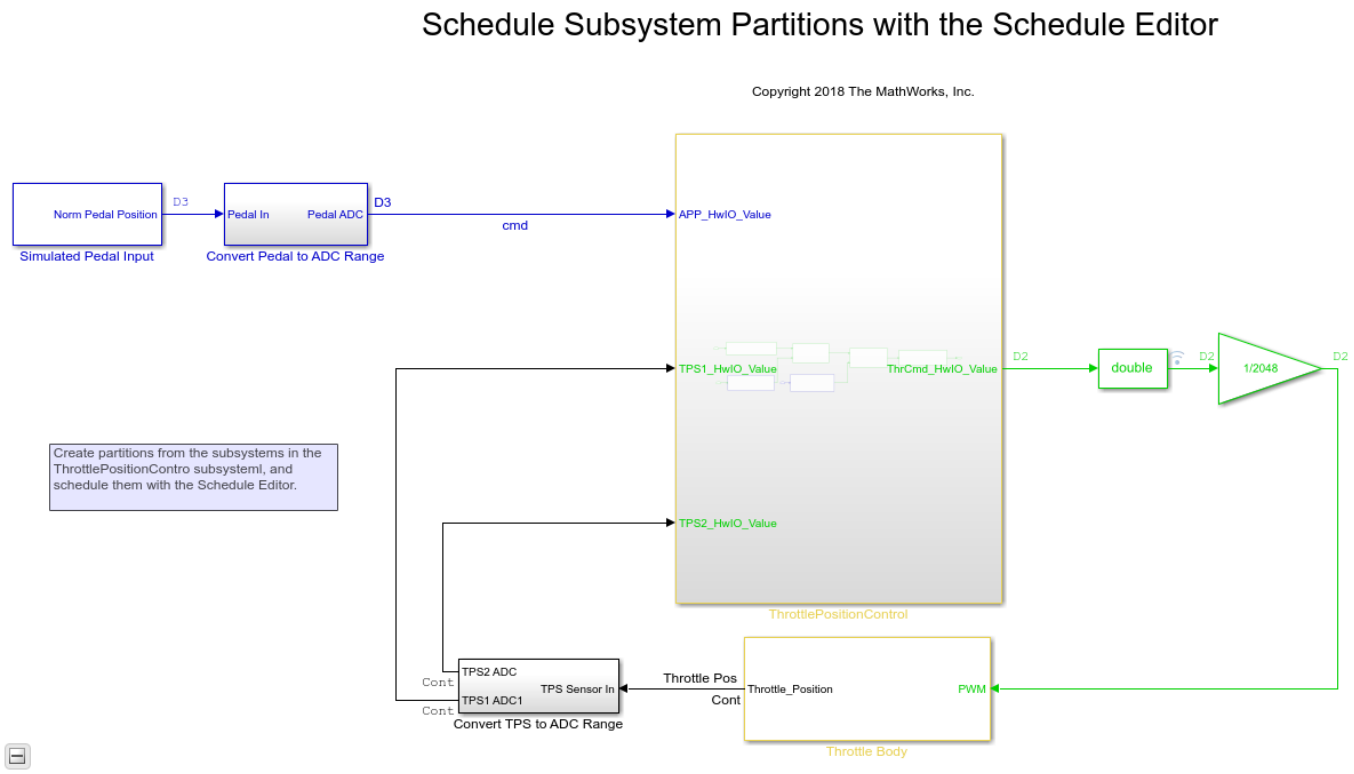
This example shows how to partition a rate-based model using the Schedule Editor. Partitions are the components of the model that can execute independently. In this example, we convert the subsystems into partitions and view and edit their schedule.

With the Schedule Editor, you can easily view and edit the schedule of the partitions. The behavior of the system depends on the order of these partitions. In this example, we observe the effects of scheduling this model on the simulation. To see the impact of partitioning and scheduling the model, we compare the model simulations before and after creating partitions and scheduling them.

Open the Model and Establish a Simulation Baseline

Open the model of a Throttle Position Control system and simulate it to establish a baseline for comparison

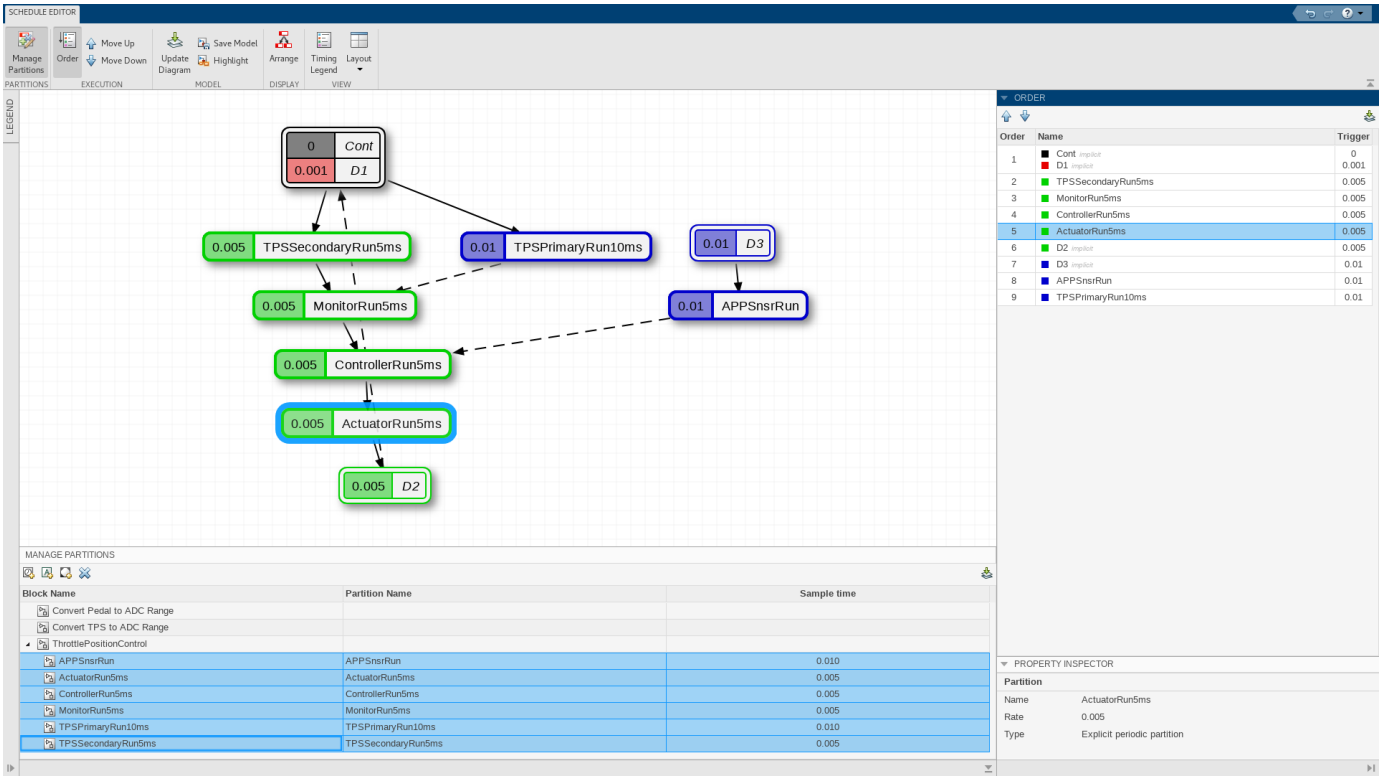
```
open_system('ScheduleEditorWithSubsystemPartitions');
sim('ScheduleEditorWithSubsystemPartitions');
```



Open the Schedule Editor and Create Partitions

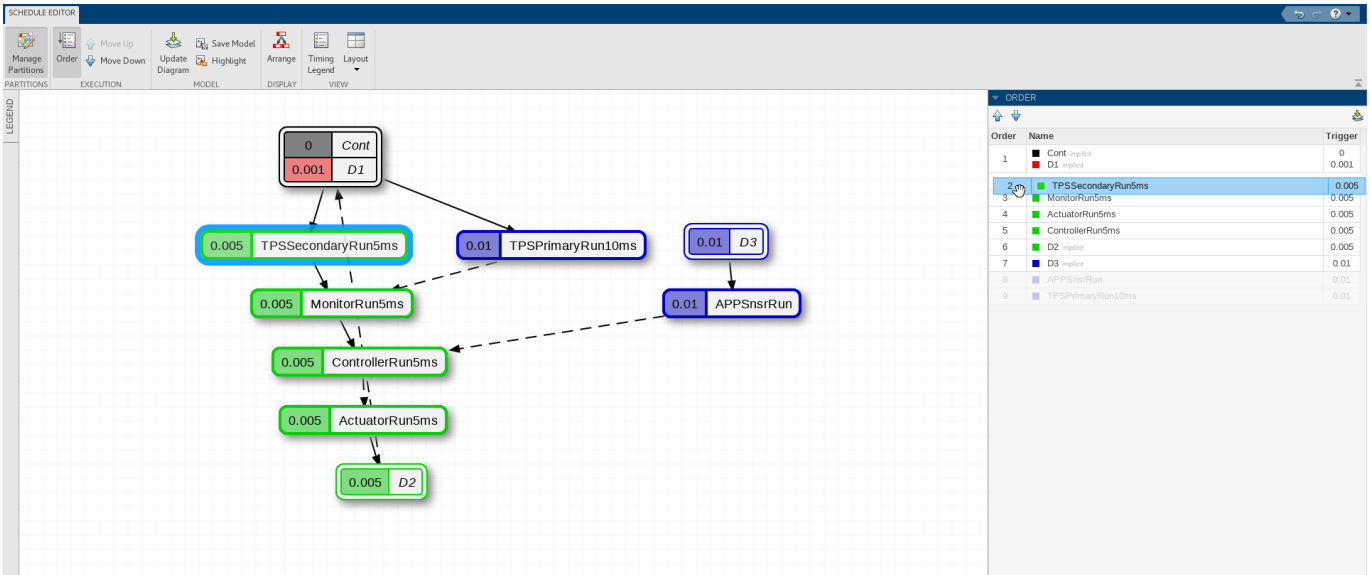
To open the Schedule Editor, click **Schedule Editor** in the **Design** section of the **Modeling** tab. Use **Manage Partitions** to create partitions from the subsystems in your model. Select all the subsystems in `ThrottlePositionControl`, and click the **Create Partitions** icon on the top of the **Manage Partitions** panel. Specify the names for the partitions and their sample time. Update the diagram to see the partitions in the Schedule Editor.

The arrows are data connections between the partitions that show the data flow. The dashed lines always indicate that there is a delay as the source runs after the destination. The solid lines indicate that there is no delay as the source runs before the destination.



Edit Partition Schedule

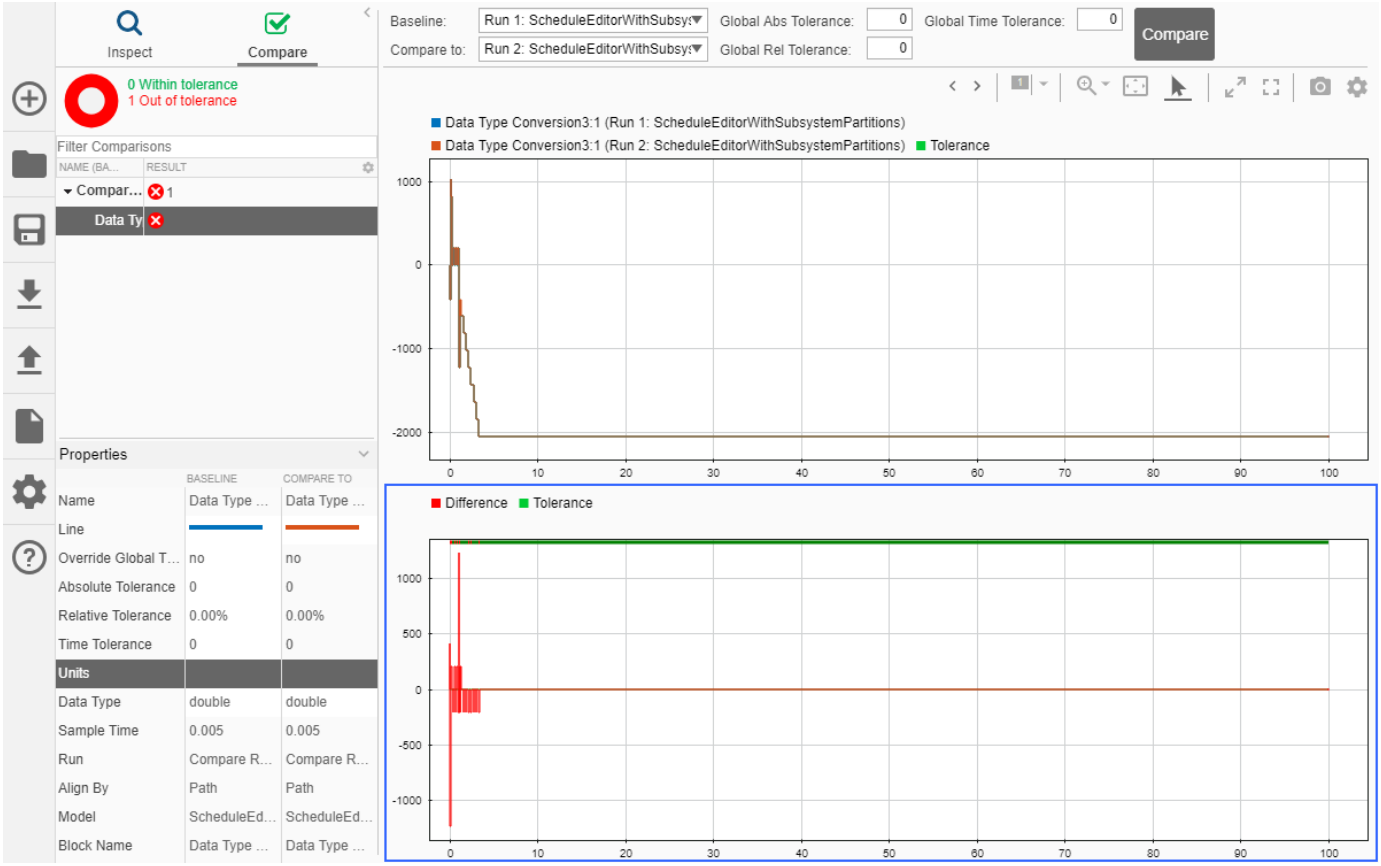
The Order shows the order the partitions run at a given time step. Assume that the order of the partitions is in an imperfect state. In this case, you want to run the `ActuatorRun5ms` partition before the `ControllerRun5ms` partition. Drag `ActuatorRun5ms` before the `ControllerRun5ms` in the order. Observe that the dependency between the `ControllerRun5ms` and the `ActuatorRun5ms` partitions changes to a delay.



Compare Runs in Simulation Data Inspector

Now, simulate the model with the changed schedule.

Open the Simulation Data Inspector. Select the two runs and compare. You can see how changing the schedule impacts the model behavior.



See Also

Schedule Editor

More About

- “Create Partitions” on page 24-4
- “Generate Code from a Partitioned Model” on page 24-22

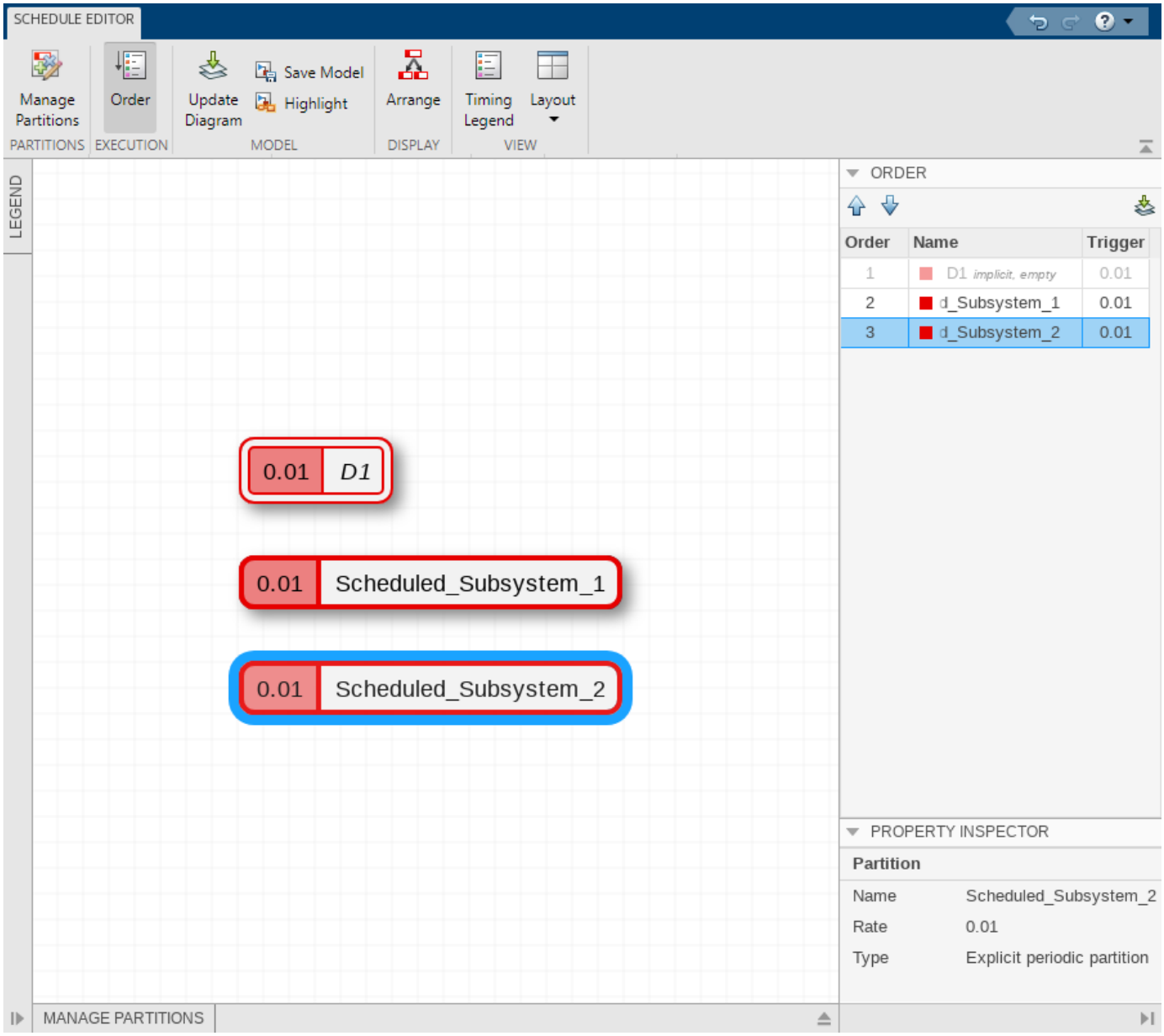
Generate Code from a Partitioned Model

Partitioning and scheduling a model has an impact on the order of the function calls in the generated code. Using the Schedule Editor to edit the schedule of a rate-based model or a referenced export-function model, the order of the functions in the generated code depends on the specified schedule in the Schedule Editor. The resulting code shows every partition as an entry point.

Note To use the code generation functionality, Embedded Coder and Simulink Coder are required.

To see the impact of the Schedule Editor on the generated code, use the model that is created in “Create A Rate-Based Model” on page 10-40.

1. Open the Schedule Editor from the **Design** section of the **Modelling** tab. Use the **Manage Partitions** panel to create partitions for Scheduled Subsystem 1 and Scheduled Subsystem 2 with 0.01 sample time. Update the diagram.
2. Change the order of the partitions by dragging `Scheduled_Subsystem_2` above `Scheduled_Subsystem_1`.



3 Generate code for the component model. From the **Apps** tab, select **C/C++ Code** > **Build Model**.

In the generated code, the order of the functions depends on the schedule specified in the Schedule Editor.

```c
#include "rate_based_model.h"
#include "rate_based_model_private.h"

/* External inputs (root inport signals with default storage) */
ExtU_rate_based_model_T rate_based_model_U;

/* External outputs (root outports fed by signals with default storage) */
ExtY_rate_based_model_T rate_based_model_Y;

/* Real-time model */
RT_MODEL_rate_based_model_T rate_based_model_M_;
RT_MODEL_rate_based_model_T *const rate_based_model_M = &rate_based_model_M_;

/* Model step function for TID0 */
void rate_based_model_step0(void)      /* Sample time: [0.01s, 0.0s] */
{
  /* (no output/update code required) */
}

/* Model step function for TID1 */
void rate_based_model_step1(void)     /* Explicit Task: Scheduled_Subsystem_2 */
{
  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In2_100ms'
   */
  rate_based_model_Y.Out2 = rate_based_model_U.In2_100ms;
}

/* Model step function for TID2 */
void rate_based_model_step2(void)     /* Explicit Task: Scheduled_Subsystem_1 */
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1_10ms'
   */
  rate_based_model_Y.Out1 = rate_based_model_U.In1_10ms;
}
```

Note Changing the constraints on the connections does impact the generated code. The changes with respect to connections are useful to set preferences about delays and dependencies.

See Also

Schedule Editor

More About

- “Create Partitions” on page 24-4
- “Schedule the Partitions” on page 24-15

Export-Function Conversion

To use the **Schedule rates with** option with the Schedule Editor, the model can not be an export-function model. To enable **Schedule rates with** option, convert your model to a rate-based model. To convert your model to a rate-based model, remove the function call input ports. For more information, see “Export-Function Models” on page 10-47.

Create and Analyze Random Schedules for a Model Using the Schedule Editor API

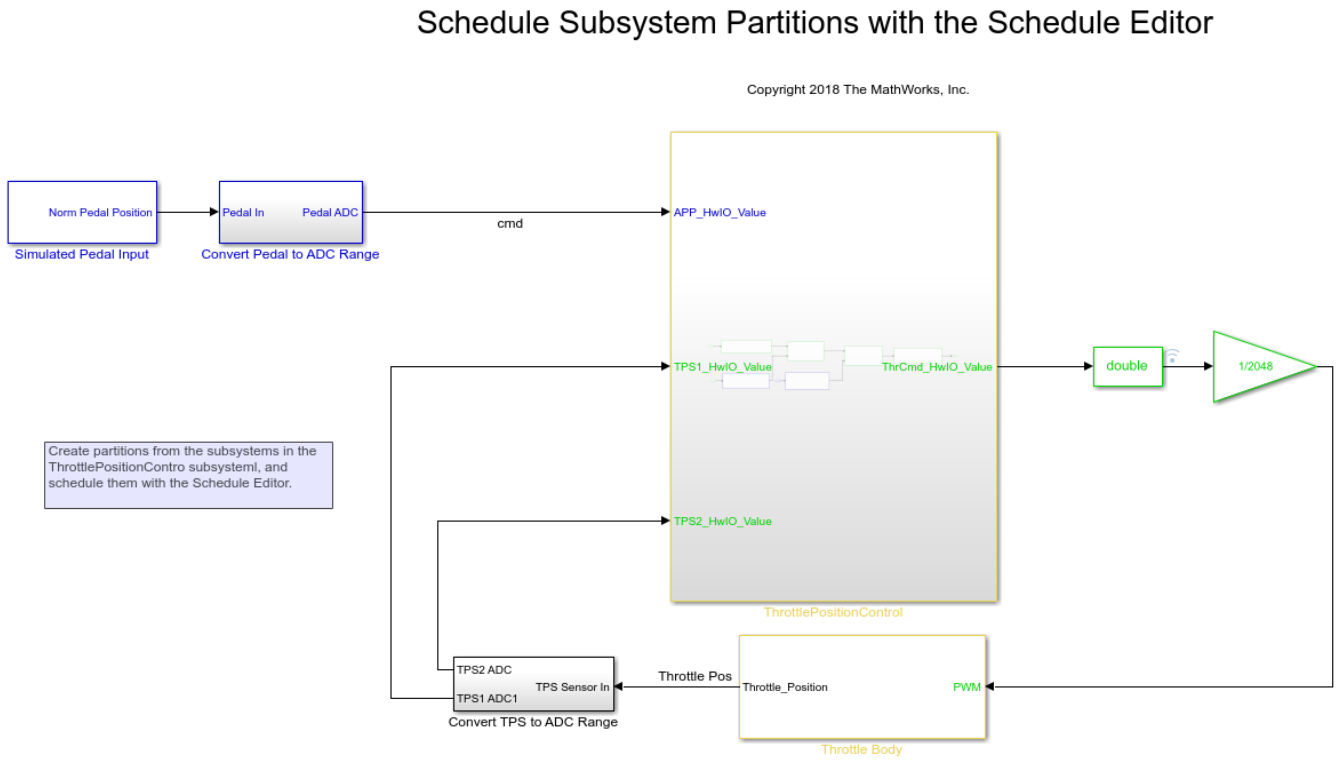
This example uses the Schedule Editor API to perform operations on the schedule. Then it uses a function to generate random schedules and analyze them in Simulation Data Inspector

Open the Model and get the Schedule Object

Open a model of a Throttle Position Control system and use get_param to obtain the `simulink.schedule.OrderedSchedule` object. This object contains the current schedule.

```
model = 'ScheduleEditorAPIWithSubsystemPartitions';
open_system(model);
schedule = get_param(model, 'Schedule')
```

```
schedule =

  OrderedSchedule with properties:

          Order: [9x3 table]
   RateSections: [3x1 simulink.schedule.RateSection]
         Events: [0x1 simulink.schedule.Event]
    Description: ''
```



Examine the Schedule Object

The schedule object has an `Order` property that contains the execution order of the partitions in the model. The `Order` property displays a table that contains partition names, their index, type, and their trigger.

```
schedule.Order
```

```
ans =

  9x3 table

                          Index      Type       Trigger
                          _____    ________    _______

    Cont                    1      Periodic    "0"
    TPSSecondaryRun5ms      2      Periodic    "0.005"
    MonitorRun5ms           3      Periodic    "0.005"
    ControllerRun5ms        4      Periodic    "0.005"
    ActuatorRun5ms          5      Periodic    "0.005"
    D2                      6      Periodic    "0.005"
    D3                      7      Periodic    "0.01"
    APPSnsrRun              8      Periodic    "0.01"
    TPSPrimaryRun10ms       9      Periodic    "0.01"
```

Use the index variable in the `Order` table to change the execution order of the model

```
schedule.Order.Index('ActuatorRun5ms') = 2;
schedule.Order
```

```
ans =

  9x3 table

                          Index      Type       Trigger
                          _____    ________    _______

    Cont                    1      Periodic    "0"
    ActuatorRun5ms          2      Periodic    "0.005"
    TPSSecondaryRun5ms      3      Periodic    "0.005"
    MonitorRun5ms           4      Periodic    "0.005"
    ControllerRun5ms        5      Periodic    "0.005"
    D2                      6      Periodic    "0.005"
    D3                      7      Periodic    "0.01"
    APPSnsrRun              8      Periodic    "0.01"
    TPSPrimaryRun10ms       9      Periodic    "0.01"
```

Any moves within the `Order` property that are made to modify the schedule should result in valid schedule. To perform the schedule modifications and valid moves easier, each partition is grouped with partitions of the same rate in the `RateSections` property. Each element of the `RateSection` property contains an order table with partitions of the same rate.

```
schedule.RateSections(2)
schedule.RateSections(2).Order
```

```
ans =

  RateSection with properties:

     Rate: "0.005"
    Order: [5x3 table]


ans =

  5x3 table

                          Index      Type      Trigger
                          _____    _________    _______

    ActuatorRun5ms          2      Periodic    "0.005"
    TPSSecondaryRun5ms      3      Periodic    "0.005"
    MonitorRun5ms           4      Periodic    "0.005"
    ControllerRun5ms        5      Periodic    "0.005"
    D2                      6      Periodic    "0.005"
```

Use the index variable to move the partitions within `RateSections`.

```
schedule.RateSections(2).Order.Index('ActuatorRun5ms') = 5;
schedule.Order
```

```
ans =

  9x3 table

                          Index      Type      Trigger
                          _____    _________    _______

    Cont                    1      Periodic    "0"
    TPSSecondaryRun5ms      2      Periodic    "0.005"
    MonitorRun5ms           3      Periodic    "0.005"
    ControllerRun5ms        4      Periodic    "0.005"
    ActuatorRun5ms          5      Periodic    "0.005"
    D2                      6      Periodic    "0.005"
    D3                      7      Periodic    "0.01"
    APPSnsrRun              8      Periodic    "0.01"
    TPSPrimaryRun10ms       9      Periodic    "0.01"
```

Create a Function to Generate Random Schedules

In this section, we create three different functions: `randomSchedule`, `generateSimulationInputs` and `simulateRandomSchedules`

`randomSchedule` function is used to create random schedules by using random permutations of index modifications in the `schedule` object. Using the `Order` and the `RateSections` properties of the `schedule` object, partitions in the schedules are moved around in different, random combinations. With these randomly created schedules, models are simulated and compared to study the effect of different schedules on simulation. In the function `randomSchedule`, the input is the model name. Then use `get_param` to obtain the `simulink.schedule.OrderedSchedule` object of

the model. The schedule object and its properties are used to modify and randomize the schedules. Create a variable `firstExecutionOrder` for the first rate section of the model. The `rateSections(1).ExecutionOrder = [firstExecutionOrder(1,:); reSchedule(firstExecutionOrder(2:end,:))]` line of code calls the function `reSchedule` which creates random permutations of the indexes.

```
type randomSchedule
```

```
function schedule = randomSchedule(model)
    % schedule = randomSchedule(model) Produces a
    % simulink.schedule.OrderedSchedule that has a randomized permutation
    % of the model's original execution order schedule

    arguments
        model char = bdroot
    end

    schedule = get_param(model, 'Schedule');

    rateSections = schedule.RateSections;
    firstOrder = rateSections(1).Order;

    % This assumes that the slowest discrete rate is at index 1. This may
    % not be the case for all models (ex. JMAAB-B).
    rateSections(1).Order = [firstOrder(1,:); reSchedule(firstOrder(2:end,:))];

    for i=2:length(rateSections)
        rateSections(i).Order = reSchedule(rateSections(i).Order);
    end

    schedule.RateSections = rateSections;
end

function out = reSchedule(in)
    numPartitions = height(in);
    in.Index = in.Index(randperm(numPartitions));
    out = in;
end
```

To analyze the effects of different schedules on the model, simulate the model with the different schedules. In this function, create an array of `Simulink.SimulationInput` objects. Through this array of `Simulink.SimulationInput` objects, you can apply the schedules to the model with the `setModelParameters` method of the `Simulink.SimulationInput` object.

```
type generateSimulationInputs
```

```
function in = generateSimulationInputs(model, numSimulations)
    % in = generateSimulationInputs(model, numSimulations) Generates
    % numSimulations Simulink.SimulationInput objects each containing a
    % different, randomized execution order schedule
    arguments
        model char = bdroot
        numSimulations double = 10
    end

    in(numSimulations) = Simulink.SimulationInput();
```

```
    in = in.setModelName(model);
    for idx = 1:numSimulations
        in(idx) = in(idx).setModelParameter('Schedule', randomSchedule(model));
    end
end
```

In the last function, use the array of `Simulink.SimulationInput` objects to run multiple simulations. Once the simulations are complete, you can plot the output of all the simulations in Simulation Data Inspector.

```
type simulateRandomSchedules
```

```
function out = simulateRandomSchedules(model, numSimulations)
    % out = simulateRandomSchedules(model, numSimulations) Simulates a
    % model numSimulations number of times.  Each simulation has a
    % randomized execution order schedule.
    arguments
        model char = bdroot
        numSimulations double = 10
    end

    in = generateSimulationInputs(model, numSimulations);
    out = sim(in);
    plot(out);
end
```

Execute the Functions

Now run the above functions for the `ScheduleEditorAPIWithSubsystemPartitions` model. First, use the `randomSchedule` function to create randomly generated schedules, then, use the `generateSimulationInputs` function to generate an array of `Simulink.SimulationInput` objects, and use the `simulateRandomSchedule` function to simulate the model with different schedules and plot their results for comparison. Let's run simulations with 15 randomly generated schedules.

```
simulateRandomSchedules(model,15)
```

```
[26-Aug-2020 08:56:53] Running simulations...
[26-Aug-2020 08:57:02] Completed 1 of 15 simulation runs
[26-Aug-2020 08:57:07] Completed 2 of 15 simulation runs
[26-Aug-2020 08:57:13] Completed 3 of 15 simulation runs
[26-Aug-2020 08:57:18] Completed 4 of 15 simulation runs
[26-Aug-2020 08:57:24] Completed 5 of 15 simulation runs
[26-Aug-2020 08:57:29] Completed 6 of 15 simulation runs
[26-Aug-2020 08:57:35] Completed 7 of 15 simulation runs
[26-Aug-2020 08:57:41] Completed 8 of 15 simulation runs
[26-Aug-2020 08:57:47] Completed 9 of 15 simulation runs
[26-Aug-2020 08:57:52] Completed 10 of 15 simulation runs
[26-Aug-2020 08:57:57] Completed 11 of 15 simulation runs
[26-Aug-2020 08:58:02] Completed 12 of 15 simulation runs
[26-Aug-2020 08:58:08] Completed 13 of 15 simulation runs
[26-Aug-2020 08:58:13] Completed 14 of 15 simulation runs
[26-Aug-2020 08:58:18] Completed 15 of 15 simulation runs

ans =

1x15 Simulink.SimulationOutput array
```

Events in Schedule Editor

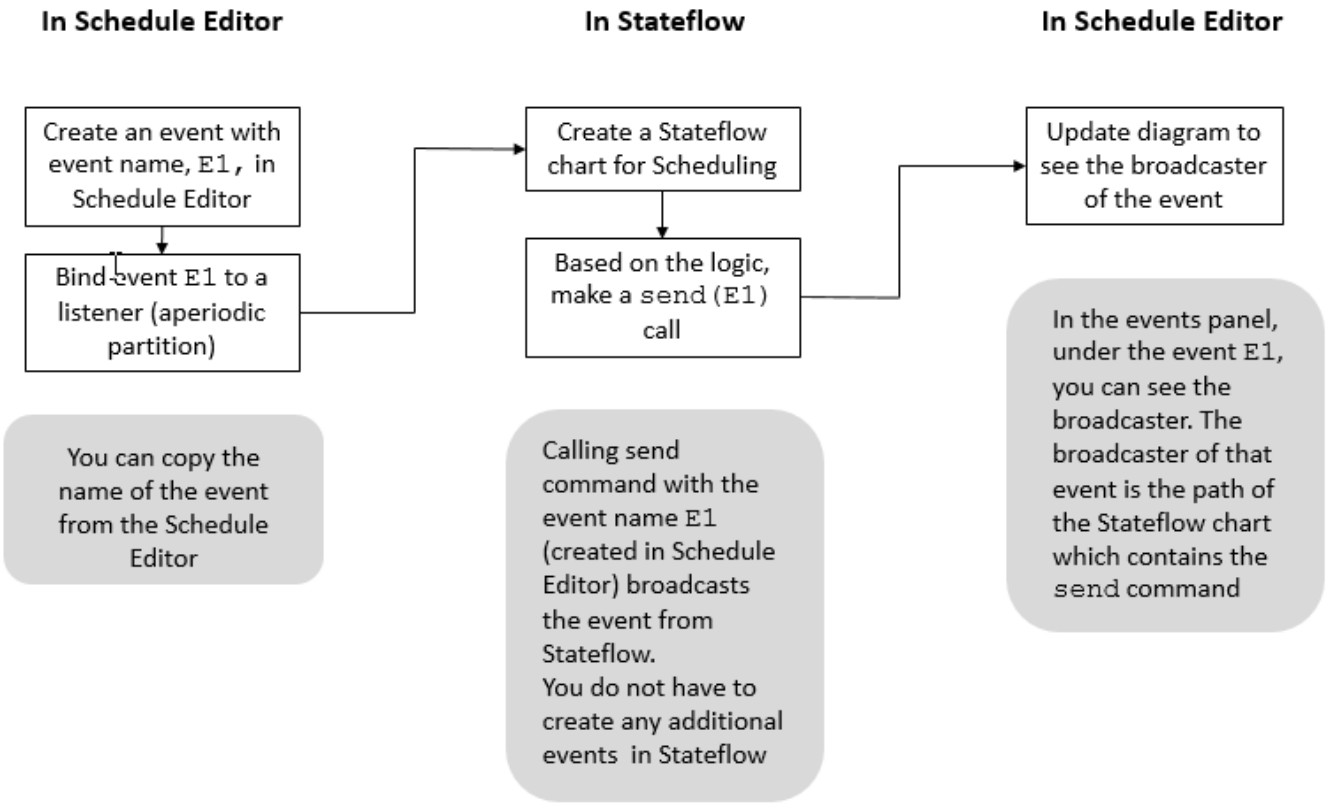
An event is a construct that represents an action, transition, or condition. You can broadcast events from within a model hierarchy. Events can connect blocks that detect important conditions with a partition to schedule the execution of the partition when the conditions occur. The Schedule Editor allows you to create and manage partitions and schedule the execution of your model. You can bind an event to an aperiodic partition that is scheduled, based on priority, for execution when the event is broadcast. Starting in R2020b, Schedule Editor events can be sent from Stateflow charts by using the `send` keyword. Events simplify system creation and provides more flexibility during scheduling.

Event Management in the Schedule Editor

Each model in a model hierarchy has its own **Events** panel in the Schedule Editor, that contains all the events present in the model and the model's children. When you open the Schedule Editor and update the diagram, all partitions and events present in the model appear in the Schedule Editor. Through the **Events** panel, you can:

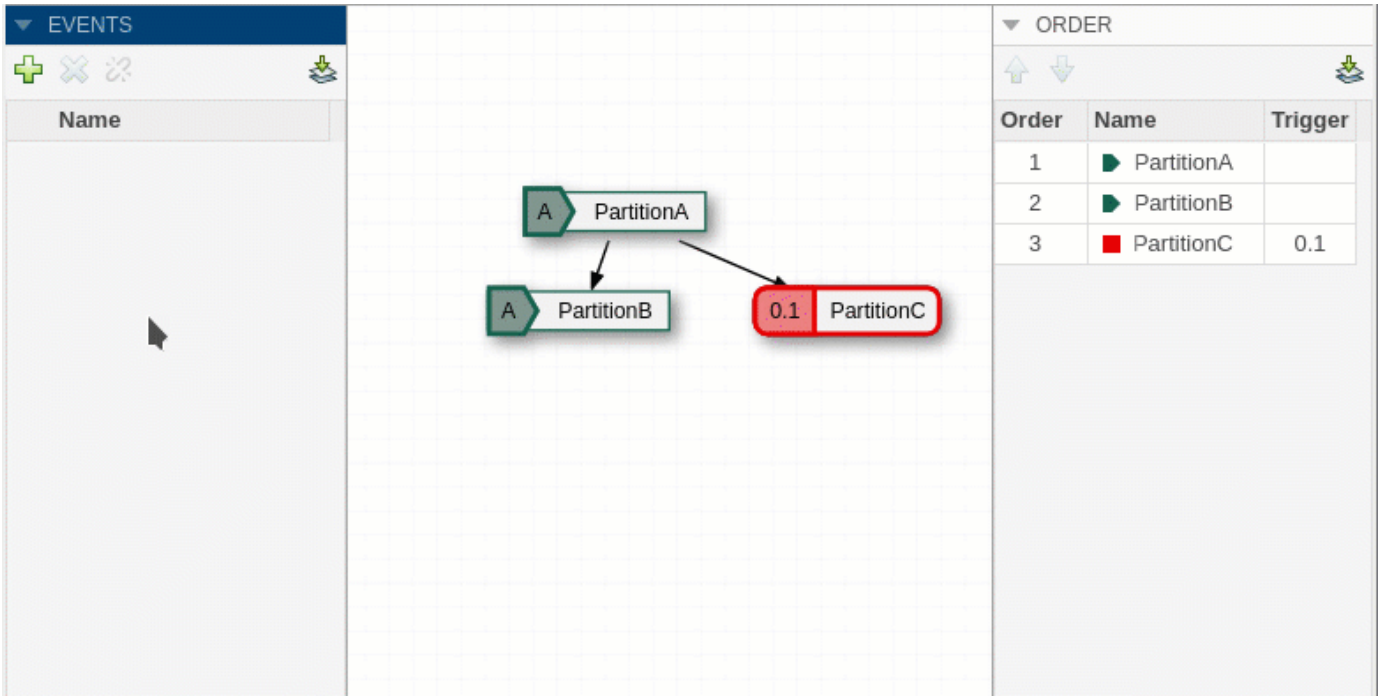
- Create an event.
- Delete an event.
- Rename an event.
- Get an event.
- Bind an event to a partition.
- Unbind an event from a partition.

You can schedule the execution of an aperiodic partition based on broadcasting of a particular event in the Stateflow Chart as shown below:



The **Events** panel in the Schedule Editor shows you the event tree. Under every event, you can see the broadcasters and listeners of that event.

When an event is bound to a partition, the event name appears on the left side of the partition and in, the **Trigger** column of the **Order** table.



Naming

Unique names are enforced for events within a model and within a model hierarchy. Within a model hierarchy, the model name prepends the event name. For example, if the model reference named `ModelName` contains an event, `E1`, then that event appears as `ModelName.E1` in the Schedule Editor.

Execution of Aperiodic Partitions with Events

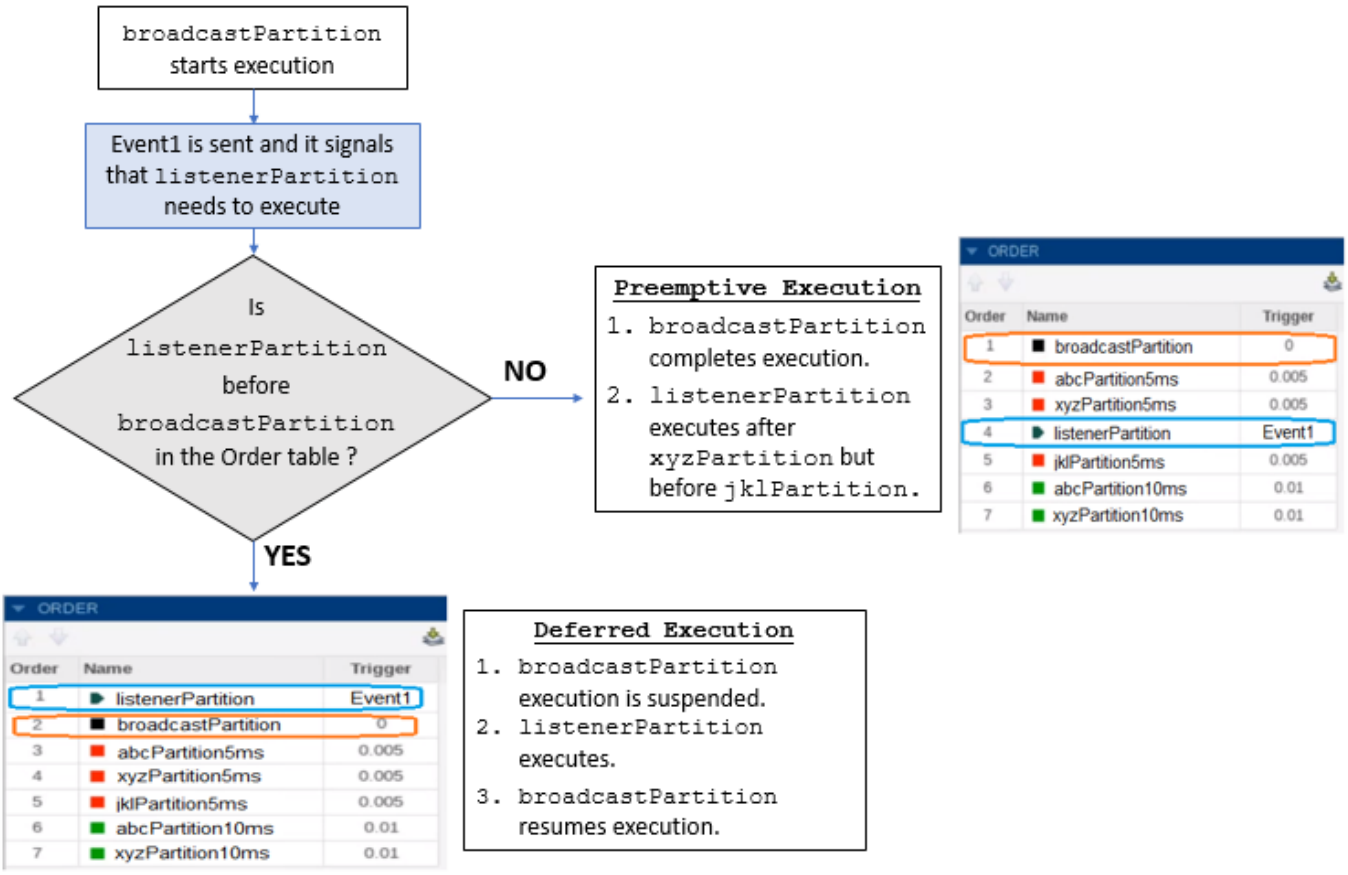
Bind Events

An event and a partition can be bound together to indicate that the partition executes when the event is broadcast. A partition may only be bound to a single event.

Priority Based Execution

If the event-bound partition is scheduled to run before the partition driving the event, then the event-bound partition execution preempts the execution of the partition driving the event. If the triggered partition is scheduled to run after the partition driving the event, then the triggered partition executes when the scheduler reaches its position in the execution order.

`listenerPartition` is an aperiodic partition and a listener to the event, `Event1`. Suppose that `Event1` comes from a Stateflow Chart present in the model and is a part of the partition called, `broadcastPartition`. When `broadcastPartition` starts executing, `Event1` occurs, which then triggers the execution of `listenerPartition`.



Hit Times

You can trigger an aperiodic partition at specified hit times. If a partition has hit times and gets bound to an event, the hit times are replaced with the specified event. Likewise, if a partition is bound to an event and hit times are added, then the bound event is removed from the partition. Variables can also specify hit times for the aperiodic partitions. In case of ambiguous variable and events names, events are given precedence.

Schedule Partitions with Events

This example shows how you can use events to schedule the execution of aperiodic partitions via a Stateflow chart. You can send events from Stateflow and use those events to control the execution of the partitions in the model by using events in Schedule Editor.

Open the Model

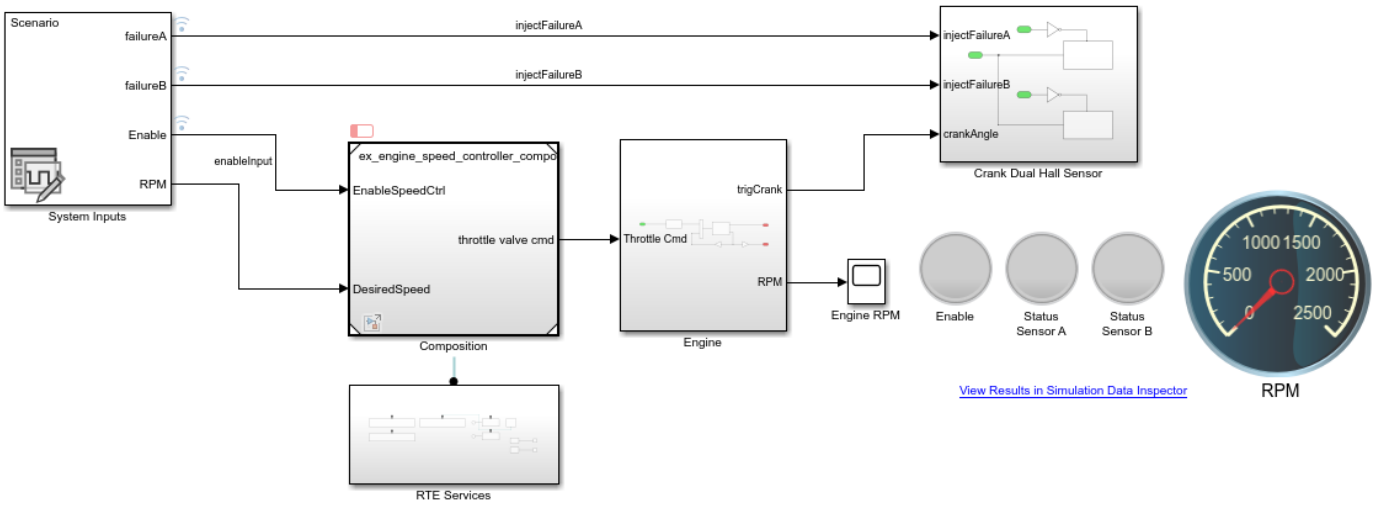
In this example, we simulate the digital speed control of an internal combustion engine. In this model, the angular velocity of the engine is measured using two redundant Hall sensors, and we simulate the failure of the Hall sensors.

The model primarily contains the following main components:

- **System Inputs**: Inputs and signals to exercise the system.

- **Engine:** Simplified representation of internal combustion engine.
- **Composition:** Digital controller intended to deploy on an Engine Control Unit (ECU), with a Run-time Environment (RTE).
- **Crank Dual Hall Sensor**: Emulation of two redundant sensors, `sensor A` and `sensor B.`
- **RTE Services**: Emulation of the services provided by Engine Control Unit.

```
open_system("ex_engine_speed_control_system");
```



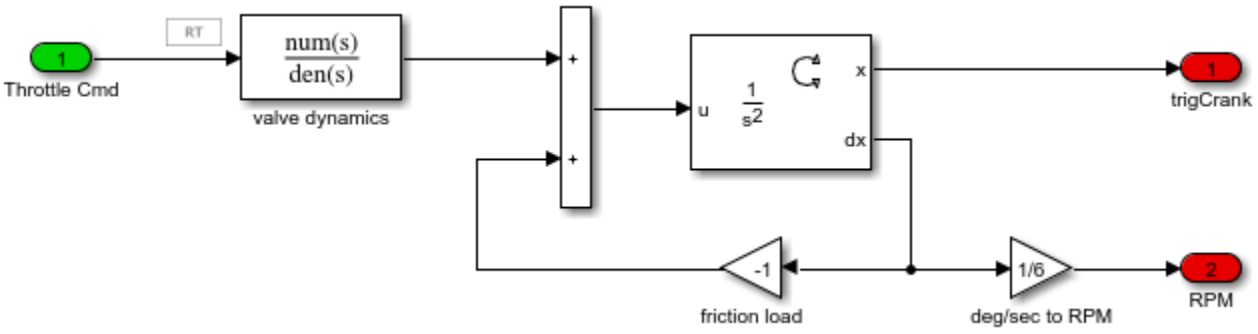
System Inputs

The system inputs for this model are designed using the Signal Editor block. The `ex_engine_speed_control_external_inputs.mat` file contains the following scenario:

- The desired angular velocity is set to `2000 rpm` for the entire simulation.
- At `t = 01 sec`, the speed controller is enabled. As a result, `Hall sensor A` is being used to compute the angular velocity and the engine speed increases to `2000 rpm.`
- At `t = 06 sec`, a failure of the first Hall sensor is injected. As a result, `Hall sensor B` is now being used to compute the angular velocity, the engine speed remains at `2000 rpm.`
- At `t = 11 sec`, a failure of the second sensor is injected. As a result, the speed controller is disabled, the engine speed falls toward zero.
- At `t = 15 sec`, the failure of `sensors A` and `B` gets resolved.
- At `t = 21 sec`, the command to enable speed control is cycled back to zero for one second and back to one. As a result, the engine speed increases to `2000 rpm.`

Engine

The engine dynamics are represented by a Transfer Function that receives a throttle command and converts it into torque, and a Second Order Integrator to integrate the motion.

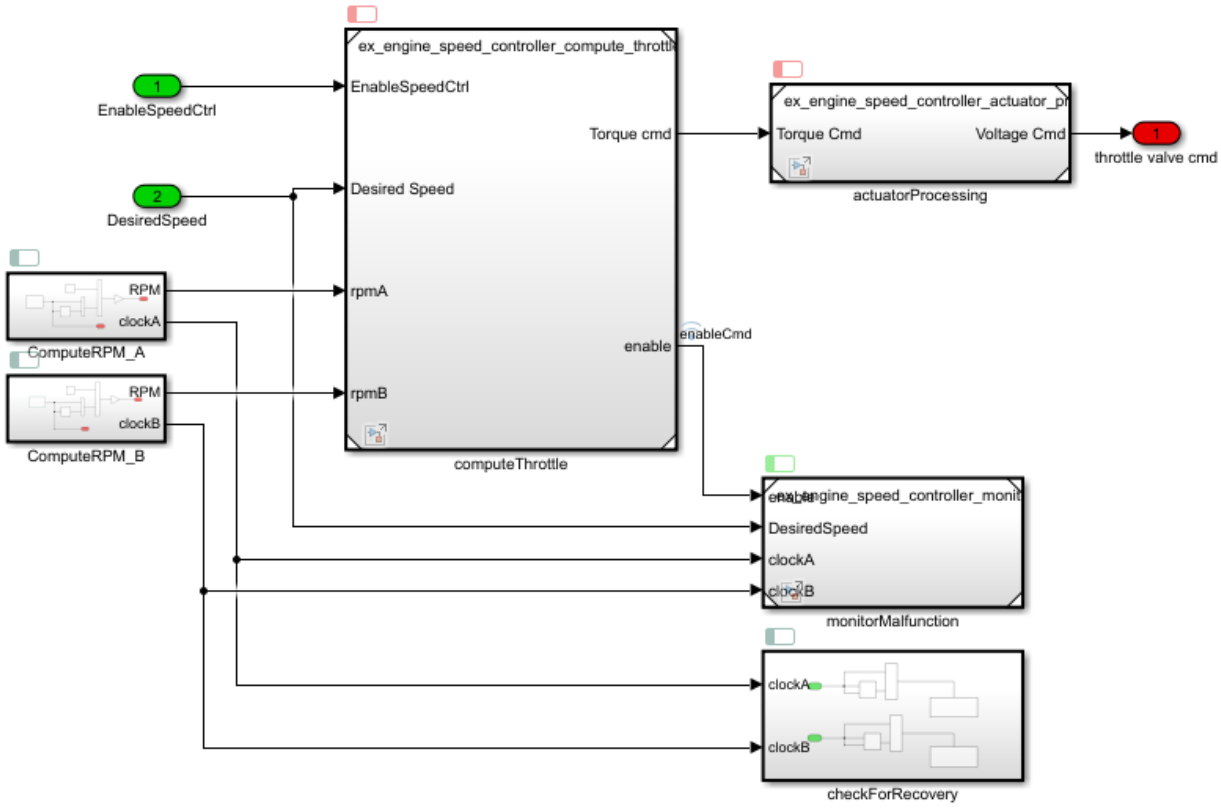


Composition

`ex_engine_speed_controller_composition` implements the digital control algorithm. It contains the following components:

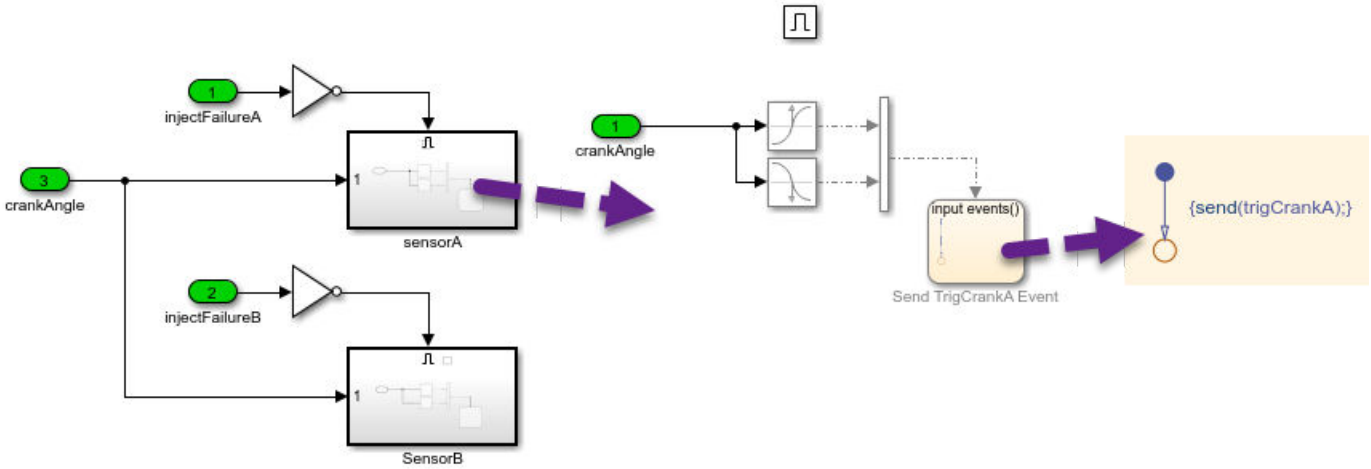
- **`ComputeRPM_A and ComputeRPM_B:`** Aperiodic partitions registered as hardware interrupts. `Hall sensors A` and `B` trigger these partitions when the engine shaft rotates by every increment of 10 degrees.
- **`computeThrottle and actuatorProcessing:`** Periodic partitions executing at 0.001 sec. `computeThrottle` interrogates the RTE at every time step.
- **`monitorMalfunction:`** Periodic partition executing at 0.01 sec. Monitors output signals of `ComputeRPM_A` and `ComputeRPM_B` to identify potential hardware failures. If a failure is detected, it calls a function provided by the RTE to register the failure.
- **`checkForRecovery:`** Aperiodic partition that is triggered by RTE once a failure has been detected. Upon detection, `checkForRecovery` is called by the RTE at a rate of 1 sec**.** It calls a function provided by the RTE if a recovery is detected.

Using the Schedule Editor, the events `checkForRecovery`, `tringCrankA` and `trigCrankB` are created and bound to aperiodic partitions `checkForRecovery, ComputeRPM_A` and `ComputeRPM_B` respectively. These events are broadcast from the top level model.



Crank Dual Hall Sensor

The Hall sensors are modeled using Hit Crossing blocks that generate a function-call every time the engine shaft rotates by increments of 10 degrees. When the Stateflow chart gets triggered, it sends events `trigCrankA` and `trigCrankB`, which have been bound to execute aperiodic partitions, `ComputeRPM_A` and `ComputeRPM_B` respectively.



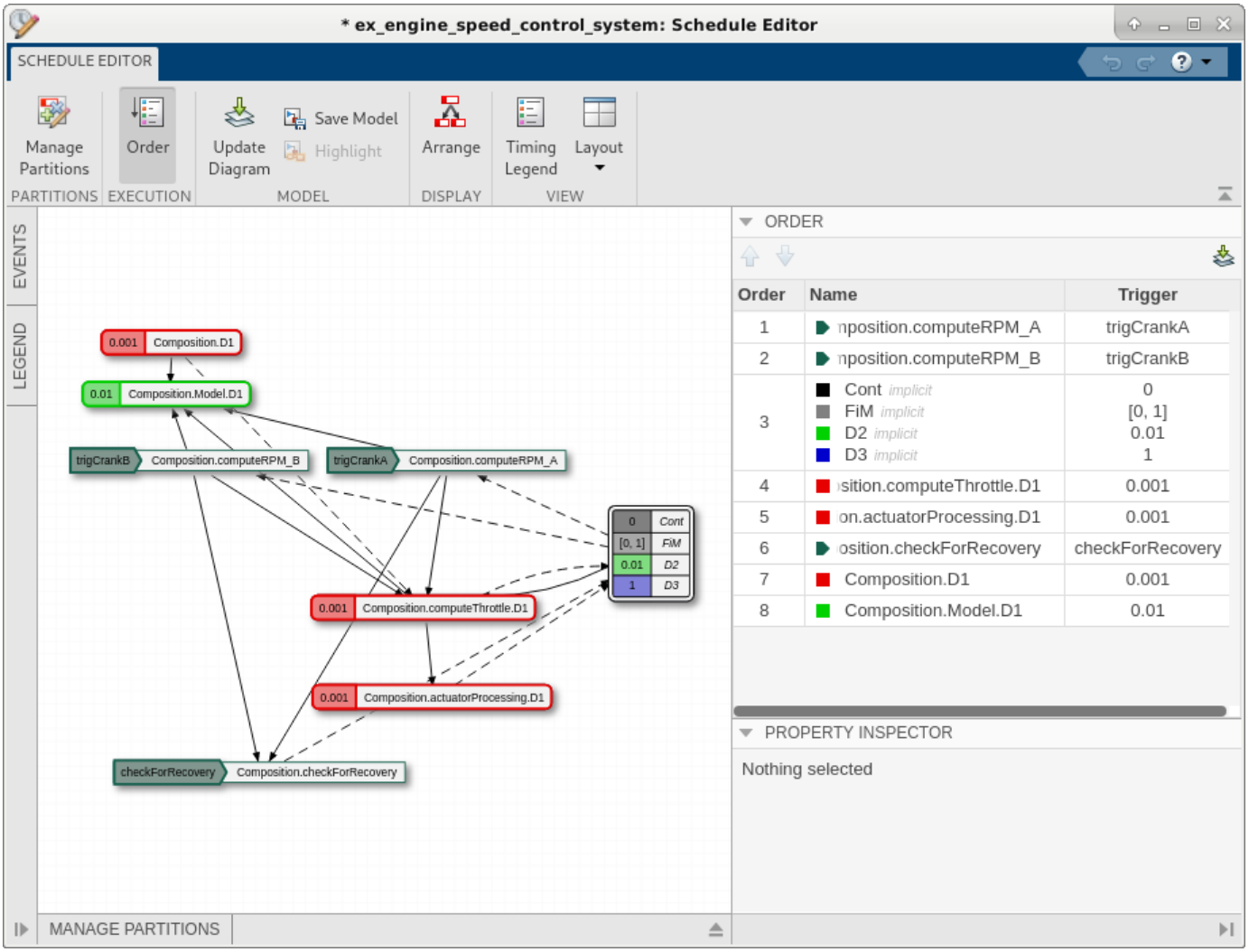
RTE Services

The `RTE Services` subsystem emulates functionalities available on the Run-time Environment onto which the code generated form the digital controller is deployed. For simulation purposes, those services are emulated using Simulink Functions.

- **`sendFailureStatus`** and **`recoverFailureStatus`** are called respectively by the `monitorMalfunctions` and `checkForRecovery` partitions when a failure or a recovery is detected. The global failure statuses are stored using Data Store Memory blocks for simulation purposes.
- **`getFailureMode`** is called by the `computeThrottle` to verify if a failure has been detected in one of the sensors.
- **`getTimeA`** and **`getTimeB`** simulate the RTE clock.
- **`Check for Recovery`** simulates the logic to determine when the `checkForRecovery` aperiodic partition of the digital controller should be triggered. Triggering is done by broadcasting the event `checkForRecovery`.

Open the Schedule Editor

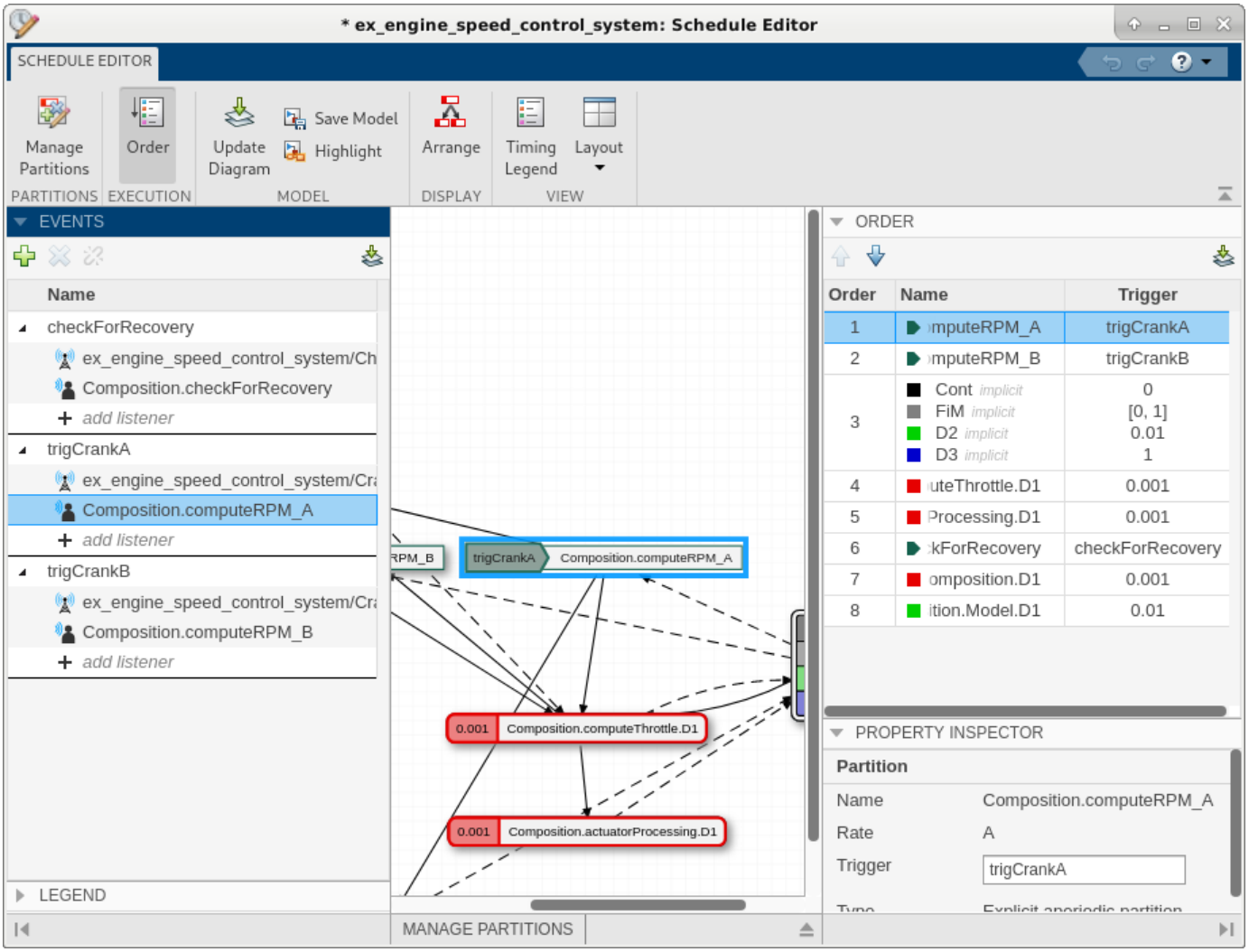
To open the Schedule Editor, on the **Modeling** tab in the **Design Section**, click **Schedule Editor**. Doing an Update Diagram compiles the models and shows you the existing partitions in the Schedule Editor.



Events Panel in the Schedule Editor

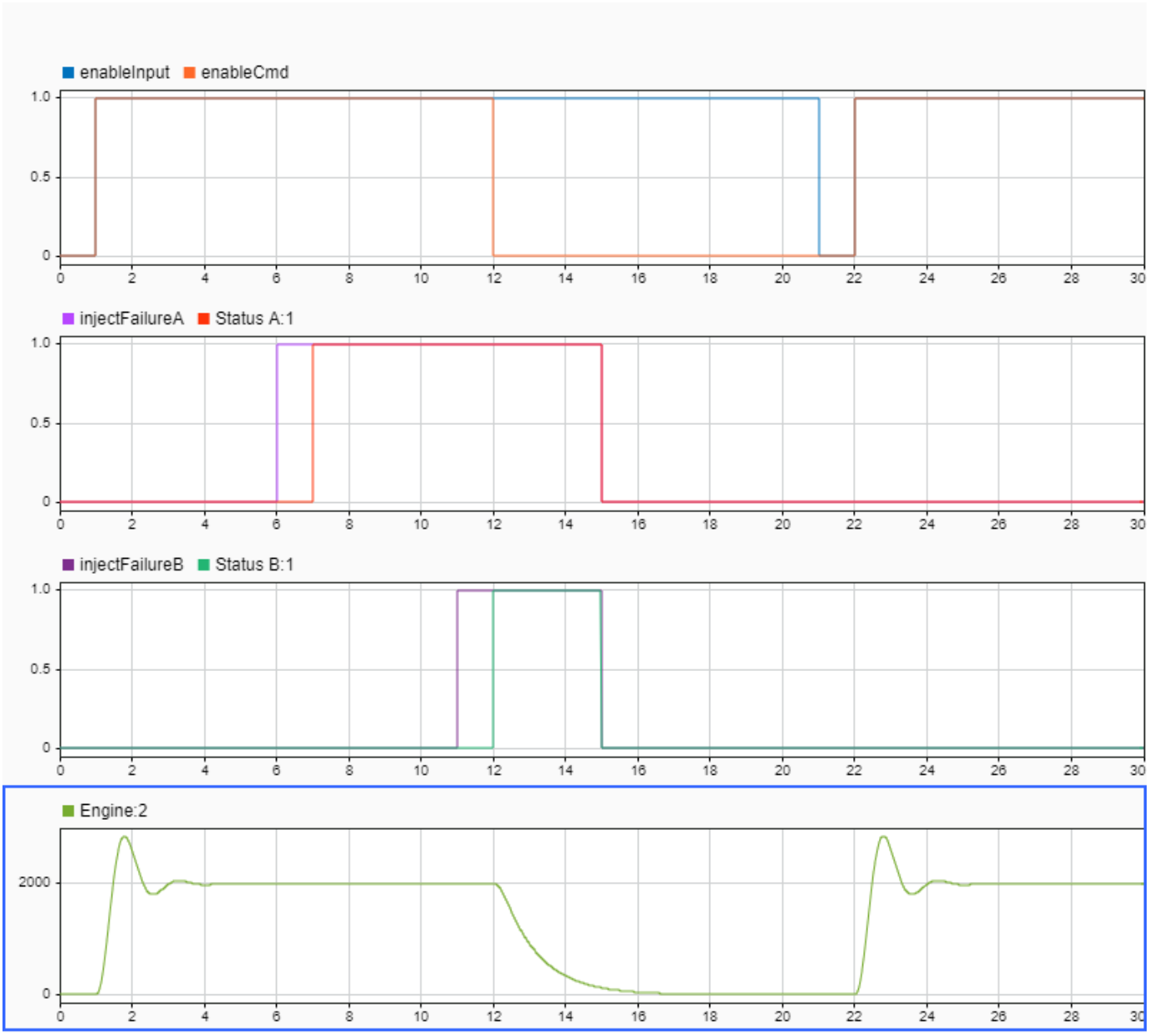
The events broadcast by the `send` keyword in Stateflow are shown in the **Events** panel of the Schedule Editor, these events are bound to aperiodic paritions, so that these partitions can be triggered from the top model. In the **Events** panel, you can expand each event to see the listeners and the broadcasters of that event. The icon,  indicates broadcaster of the event and the icon,  indicates the listner. In this example, the sender of the event `checkForRecovery` is the Stateflow Chart in `RTE Services` subsystem, the sender of the events `trigCrankA` and `trigCrankB` is the Stateflow chart in the Crank Dual Hall -> Sensor A and Sensor B.

In the **Order** panel, the partitions are arranged in the order of priorty of execution. Since `ComputeRPM_A` and `ComputeRPM_B` are time sensitive, their priority is the highest. Therefore, when the events `trigCranA` and `trigCrankB` are broadcast, the corresponding partitions `ComputeRPM_A` and `ComputeRPM_B` are executed immediately. In contrast, the aperiodic partition `checkForRecovery` is less time sensitive, and is lower in the priority order. Therefore, when the event `checkForRecovery` is broadcast, the execution of the corresponding partition `checkForRecovery` is deferred until all the partitions with higher priority complete execution.



Simulation Results

Click on **View Results in Simulation Data Inspector** in the model to view the results of the simulation.



Test Harness Generation

You can use the Schedule Editor to generate a test harness for a model with events. Using events with test harnesses helps you avoid complex wiring between the test sequence block and the entire system.

The generated test harness gets its own Schedule Editor, which enables you to easily send events through the test harness. Through the test harness scheduler, you can test the model under different scenarios by triggering events at specific times.

The Events panel also allows you to bind existing events to other aperiodic partitions. You can do this by dragging and dropping the event over a valid aperiodic partition, or by adding the partition

directly using the dropdown. You can order the partitions that have events as their Trigger in the Order table relative to the other partitions.You can also create events in Schedule Editor. Click the plus icon. Double click Add Row to create a new event. You can use this event to send from Stateflow to schedule execution of an aperiodic partition.

Limitations and Error Conditions

- Events in the Schedule Editor cannot be used in models with export-functions.
- Events do not support code generation and do not impact the generated code.

Events in the Schedule Editor use the following guidelines:

- An event cannot be raised before that event has processed.
- Duplicate event names in the parent model caused by giving two model references the same name are not allowed.
- Infinite looping is not allowed.
- A partition cannot raise an event that triggers itself.

See Also

Schedule Editor

More About

- “Create Partitions” on page 24-4
- “Generate Code from a Partitioned Model” on page 24-22

Simulating Dynamic Systems

Running Simulations

Simulate a Model Interactively

Simulation Basics

You can simulate a model in the Simulink Editor using the **Run** button on the toolstrip. The **Run** button also appears in tools within the Simulink Editor. You can simulate from any tool that includes the button, such as the Scope viewer.

Before you start a simulation, you can specify options like simulation start time, stop time, and the solver for solving the model. (See “Solver Selection Criteria” on page 25-5) You specify these options in the Configuration Parameters dialog box, which you can open by clicking **Model Settings** on the **Modeling** tab. These settings are saved with the model in a configuration set. You can create multiple configuration sets for each model and switch between them to see the effects of different settings. See “Model Configuration Sets”.

By default, simulations start at 0.0 s and end at 10.0 s.

Note In the Simulink software, time and all related parameters (such as sample times) are implicitly in seconds. If you choose to use a different time unit, scale parameters accordingly.

The **Solver** configuration pane allows you to specify other start and stop times for the currently selected simulation configuration. See “Solver Pane” for more information.

Note Simulation time and actual clock time are not the same. For example, if running a simulation for 10 s usually does not take 10 s as measured on a clock. The amount of time it actually takes to run a simulation depends on many factors including the complexity of the model, the step sizes, and the computer speed.

After you set your model configuration parameters, you can start the simulation. You can pause, resume, and stop simulation using toolstrip controls. You can also simulate more than one model at a time, so you can start another simulation while one is running.

During simulation, you cannot make changes to the structure of the model, such as adding or deleting lines or blocks. However, you can make these changes while a simulation is running:

- Modify some configuration parameters, including the stop time and the maximum step size.
- Modify the parameters of a block, as long as you do not cause a change in:
  - Number of states, inputs, or outputs
  - Sample time
  - Number of zero crossings
  - Vector length of any block parameters
  - Length of the internal block work vectors
  - Dimension of any signals

You can also examine the model visually as it simulates. For example, you can click a line to see the signal carried on that line on a Floating Scope or Display block. You can also display port values as a model simulates. See “Display Port Values for Debugging” on page 36-16.

Run, Pause, and Stop a Simulation

To start simulating your model, click the **Run** button  . You can pause, resume, or stop a simulation using the corresponding controls on the toolstrip.

The model starts simulating at the specified start time and runs until the specified end time. While the simulation is running, information at the bottom of the editor shows the percentage of simulation completed and the current simulation time.

- If an error occurs, simulation stops and a message appears. If a warning condition occurs, simulation completes. In both cases, click the diagnostics link at the bottom of the editor to see the message, which helps you to locate errors.
- Pausing takes effect after the current time step finishes executing. Resuming a paused simulation occurs at the next time step.
- If you stop a simulation, the current time step completes, and then simulation stops.
- If the model outputs to a file or to the workspace, stopping or pausing simulation writes the data.

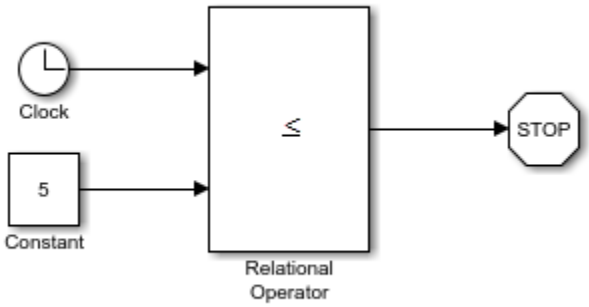
Use Blocks to Stop or Pause a Simulation

Stop Simulation Using Stop Simulation Blocks

You can use the Stop Simulation block to stop a simulation when the input to the block is nonzero. If the block input is a vector, any nonzero element stops the simulation.

1. Add a Stop Simulation block to your model.
2. Connect the Stop Simulation block to a signal whose value becomes nonzero at the specified stop time.

For example, this model stops the simulation when the simulation time reaches 10.



Pause Simulation Using Assertion Blocks

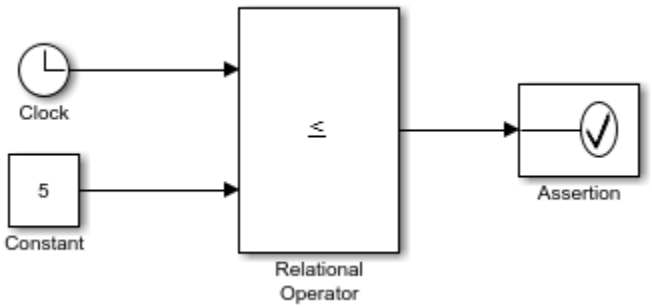
You can use an Assertion block to pause the simulation when the input signal to the block is zero. The Assertion block uses the `set_param` command to pause the simulation. See “Run Simulations

Programmatically” on page 26-2 for more information on using the `set_param` command to control the execution of a Simulink model.

1. Add an Assertion block to your model.
2. Connect the Assertion block to a signal whose value becomes zero at the desired pause time.
3. In the Assertion block dialog box, clear the **Stop simulation when assertion fails** check box. Enter this command as the value of **Simulation callback when assertion fails**:

```
set_param(bdroot,'SimulationCommand','pause'),
disp(sprintf('\nSimulation paused.'))
```

This model uses an Assertion block with these settings to pause the simulation when the simulation time reaches 5.



When the simulation pauses, a message appears that shows the time the block paused the simulation.

You can resume the simulation using **Continue** as you can for any paused simulation.

See Also

Assertion | Stop Simulation | `sim`

Related Examples

- “Systematic Diagnosis of Errors and Warnings” on page 25-53

More About

- “Run Individual Simulations”
- “Solver Selection Criteria” on page 25-5

Choose a Solver

To simulate a dynamic system, you compute its states at successive time steps over a specified time span. This computation uses information provided by a model of the system. Time steps are time intervals when the computation happens. The size of this time interval is called step size. The process of computing the states of a model in this manner is known as solving the model. No single method of solving a model applies to all systems. Simulink provides a set of programs called solvers. Each solver embodies a particular approach to solving a model.

A solver applies a numerical method to solve the set of ordinary differential equations that represent the model. Through this computation, it determines the time of the next simulation step. In the process of solving this initial value problem, the solver also satisfies the accuracy requirements that you specify.

Mathematicians have developed a wide variety of numerical integration techniques for solving the ordinary differential equations (ODEs) that represent the continuous states of dynamic systems. An extensive set of fixed-step and variable-step continuous solvers are provided, each of which implements a specific ODE solution method (see “Compare Solvers” on page 3-6). Select solvers in the **Solver** pane of model configuration parameters.

All solvers provided by MATLAB and Simulink follow a similar naming convention: `ode`, followed by two or three numerals indicating the orders of the solver. Some solvers can solve stiff differential equations and the methods used by them are expressed by the `s`, `t`, or `tb` suffixes.

Solver Selection Criteria

The appropriate solver for simulating a model depends on these characteristics:

- System dynamics
- Solution stability
- Computation speed
- Solver robustness

As such, the numerical solvers provided by Simulink can be broadly classified by two properties.

Computation Step Size Type

- Fixed-step solvers, as the name suggests, solve the model using the same step size from the beginning to the end of the simulation. You can specify the step size or let the solver choose it. Generally, decreasing the step size increases the accuracy of the results and the time required to simulate the system.
- Variable-step solvers vary the step size during the simulation. These solvers reduce the step size to increase accuracy at certain events during the simulation of the model, such as rapid state changes, zero-crossing events, etc. Also, they increase the step size to avoid taking unnecessary steps when the states of a model change slowly. Computing the step size adds to the computational overhead at each step. However, it can reduce the total number of steps, and hence the simulation time required to maintain a specified level of accuracy for models with zero-crossings, rapidly changing states, and other events requiring extra computation.

Model States

- Continuous solvers use numerical integration to compute continuous states of a model at the current time step based on the states at previous time steps and the state derivatives. Continuous solvers rely on individual blocks to compute the values of the discrete states of the model at each time step.
- Discrete solvers are primarily used for solving purely discrete models. They compute only the next simulation time step for a model. When they perform this computation, they rely on each block in the model to update its individual discrete state. They do not compute continuous states.

Use an iterative approach to choose a solver for your requirements. Compare simulation results from several solvers and select a solver that offers the best performance with minimal tradeoffs.

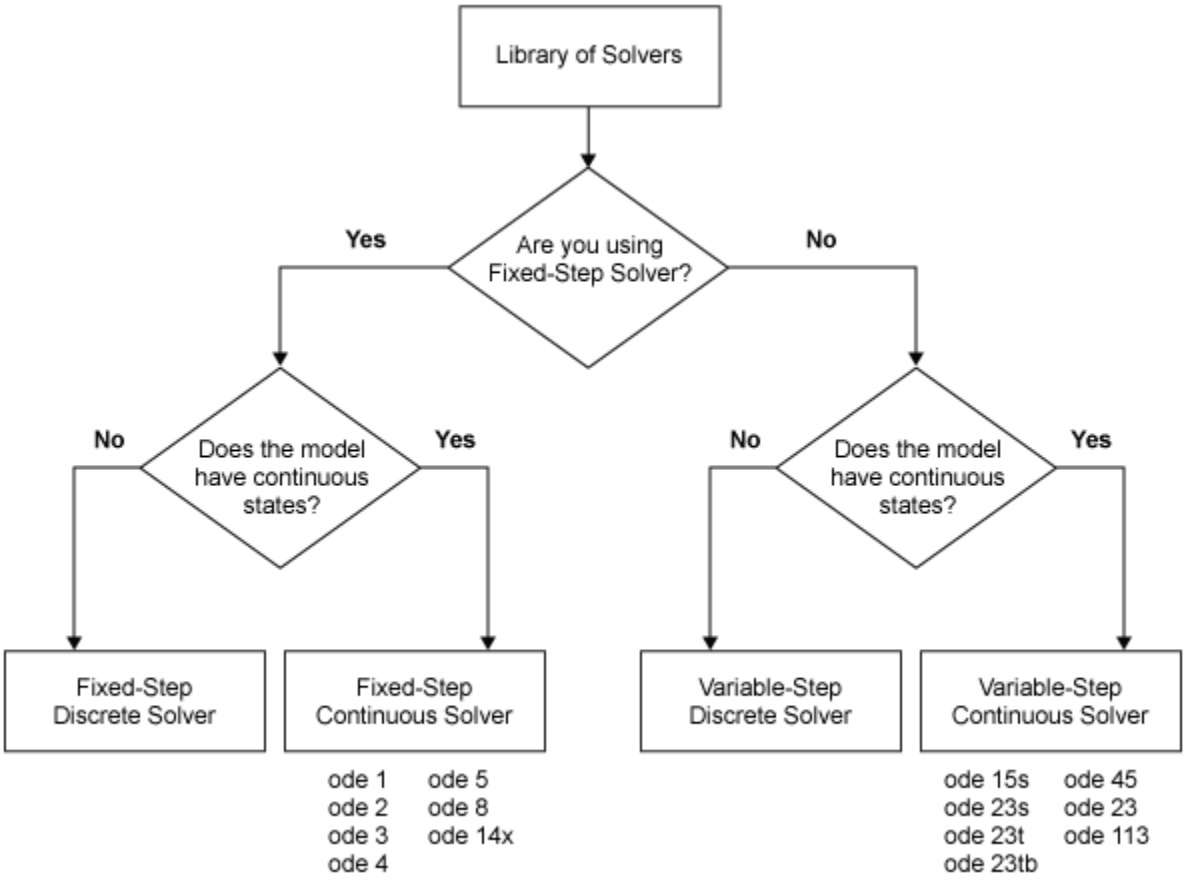
Select a solver for your model in these ways:

- Use auto solver. New models have their solver selection set to auto solver by default. Auto solver recommends a fixed-step or variable-step solver for your model as well as the maximum step size. For more information, see “Select Solver Using Auto Solver” on page 25-38.
- If you are not satisfied with the simulation results using auto solver, select a solver in the **Solver** pane in the model configuration parameters.

When you build and simulate a model, you can choose the solver based on the dynamics of your model. A variable-step solver is better suited for purely continuous models, like the dynamics of a mass spring damper system. A fixed-step solver is recommended for a model that contains several switches, like an inverter power system, due to the number of solver resets that would cause a variable-step solver to behave like a fixed-step solver.

Note When you deploy a model as generated code, you can use only a fixed-step solver. If you select a variable-step solver during simulation, use it to calculate the step size required for the fixed-step solver that you need at deployment.

This chart provides a broad classification of solvers in the Simulink library.



To tailor the selected solver to your model, see “Check and Improve Simulation Accuracy” on page 31-11.

Ideally, the solver you select should:

- Solve the model successfully.
- For variable-step solvers, provide a solution within the tolerance limits you specify.
- Solve the model in a reasonable duration.

A single solver might not meet all of these goals. Try simulating using different solvers before making a selection.

The Simulink library provides several solvers, all of which can work with the algebraic loop solver. For more information, see “How the Algebraic Loop Solver Works” on page 3-30.

Solver Type	Explicit/ Implicit	Discrete	Continuous	Variable-Order
Fixed-Step	Explicit	Not Applicable	“Fixed-Step Continuous Explicit Solvers” on page 25-22	Not Applicable

Solver Type	Explicit/ Implicit	Discrete	Continuous	Variable-Order
	Implicit	Not Applicable	“Fixed-Step Continuous Implicit Solvers” on page 25-23	Not Applicable
Variable-Step	Explicit	“Variable Step Solvers in Simulink” on page 25-14	“Variable-Step Continuous Explicit Solvers” on page 25-16	“Single-Order Versus Variable-Order Continuous Solvers” on page 3-9
	Implicit		“Variable-Step Continuous Implicit Solvers” on page 25-17	“Single-Order Versus Variable-Order Continuous Solvers” on page 3-9

In the **Solver** pane of model configuration parameters, the Simulink library of solvers is divided into two major types. See “Fixed-Step Versus Variable-Step Solvers” on page 3-6.

For other ways to compare solvers, see:

- “Continuous Versus Discrete Solvers” on page 3-7
- “Explicit Versus Implicit Continuous Solvers” on page 3-8
- “One-Step Versus Multistep Continuous Solvers” on page 3-8
- “Single-Order Versus Variable-Order Continuous Solvers” on page 3-9

See Also

Related Examples

- “Select Solver Using Auto Solver” on page 25-38
- “Examine Model Dynamics Using Solver Profiler” on page 33-2

More About

- “Compare Solvers” on page 3-6
- “Solver Pane”
- “Algebraic Loop Concepts” on page 3-27
- “Understand Profiling Results” on page 33-5

Choose a Jacobian Method for an Implicit Solver

For implicit solvers, Simulink must compute the *solver Jacobian*, which is a submatrix of the Jacobian matrix associated with the continuous representation of a Simulink model. In general, this continuous representation is of the form:

$$\dot{x} = f(x, t, u)$$
$$y = g(x, t, u)\,.$$

The Jacobian, J, formed from this system of equations is:

$$J = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial u} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial u} \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

In turn, the solver Jacobian is the submatrix, J_x.

$$J_x = A = \frac{\partial f}{\partial x}\,.$$

Sparsity of Jacobian

For many physical systems, the solver Jacobian J_x is *sparse*, meaning that many of the elements of J_x are zero.

Consider the following system of equations:

$$\dot{x}_1 = f_1(x_1, x_3)$$
$$\dot{x}_2 = f_2(x_2)$$
$$\dot{x}_3 = f_3(x_2)\,.$$

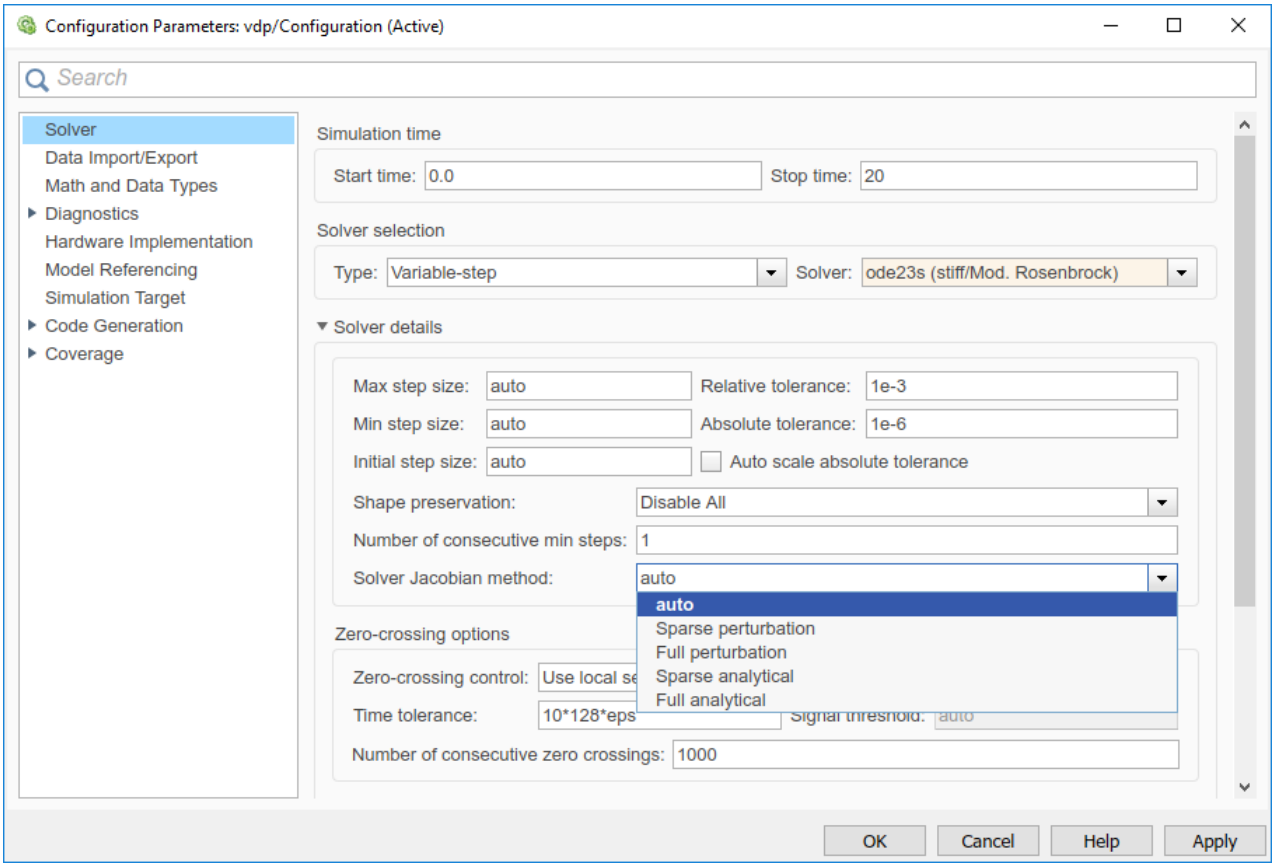
From this system, you can derive a sparsity pattern that reflects the structure of the equations. The pattern, a Boolean matrix, has a 1 for eachx_i that appears explicitly on the right-hand side of an equation. Therefore, you obtain:

$$J_{x,\,pattern} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

The sparse perturbation and the sparse analytical methods may be able to take advantage of this sparsity pattern to reduce the number of computations necessary and improve performance.

Solver Jacobian Methods

When you choose an implicit solver from the **Solver** pane of the configuration parameters dialog box, a parameter called `Solver Jacobian method` and a drop-down menu appear. This menu has five options for computing the solver Jacobian.



Note If you set **Automatic solver parameter selection** to `error` in the Solver Diagnostics pane, and you choose a different solver than that suggested by Simulink, you may receive an error.

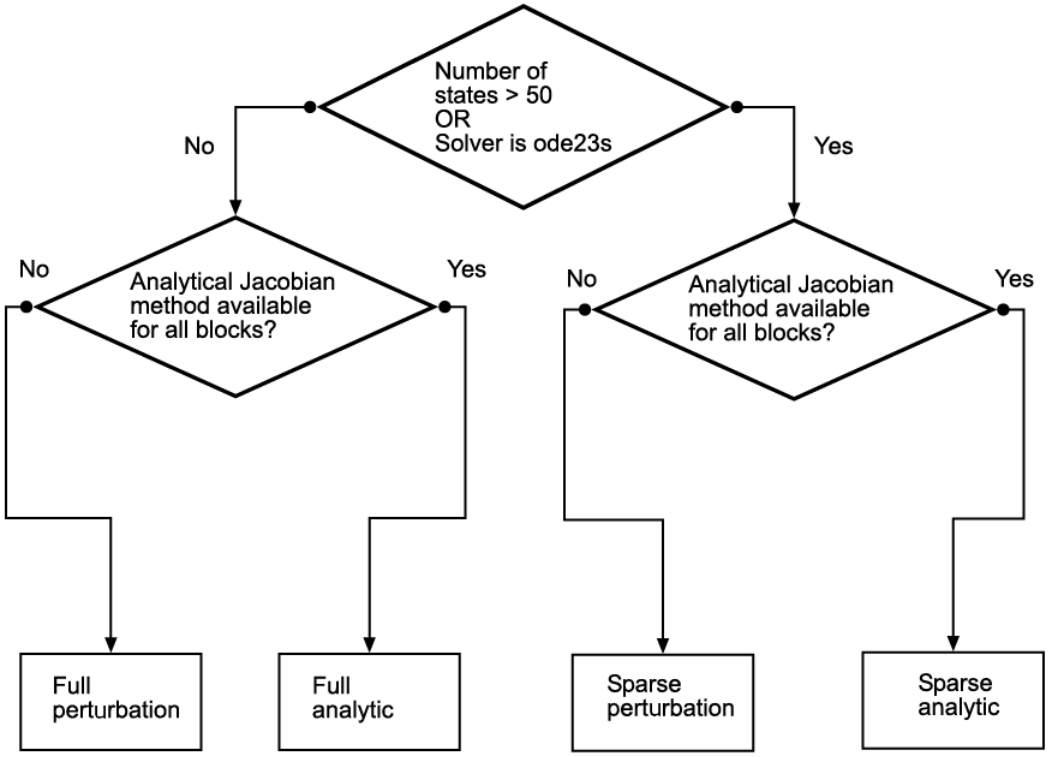
Limitations

The solver Jacobian methods have these limitations associated with them.

- If you select an analytical Jacobian method, but one or more blocks in the model do not have an analytical Jacobian, then Simulink applies a perturbation method.
- If you select sparse perturbation and your model contains data store blocks, Simulink applies the full perturbation method.

Heuristic 'auto' Method

The default setting for the `Solver Jacobian method` is `auto`. Selecting this choice causes Simulink to determine which of the remaining four methods best suits your model. This flowchart depicts the algorithm.



Sparse methods are beneficial for models that have a large number of states. If 50 or more states exist in your model, `auto` chooses a sparse method. Unlike other implicit solvers, `ode23s` is a sparse method because it generates a new Jacobian at every time step. A sparse analytical or a sparse perturbation method is, therefore, advantageous. Selecting `auto` also ensures that the analytical methods are used only if every block in your model can generate an analytical Jacobian.

Full and Sparse Perturbation Methods

The full perturbation method solves the full set of perturbation equations and uses LAPACK for linear algebraic operations. This method is costly from a computational standpoint, but remains a recommended method for establishing baseline results.

The sparse perturbation method attempts to improve the run-time performance by taking mathematical advantage of the sparse Jacobian pattern. Returning to the sample system of three equations and three states,

$$\dot{x}_1 = f_1(x_1, x_3)$$
$$\dot{x}_2 = f_2(x_2)$$
$$\dot{x}_3 = f_3(x_2)\,.$$

The solver Jacobian is:

$$J_x = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{f_1(x_1+\Delta x_1, x_2, x_3) - f_1}{\Delta x_1} & \frac{f_1(x_1, x_2+\Delta x_2, x_3) - f_1}{\Delta x_2} & \frac{f_1(x_1, x_2, x_3+\Delta x_3) - f_1}{\Delta x_3} \\ \frac{f_2(x_1+\Delta x_1, x_2, x_3) - f_2}{\Delta x_1} & \frac{f_2(x_1, x_2+\Delta x_2, x_3) - f_2}{\Delta x_2} & \frac{f_2(x_1, x_2, x_3+\Delta x_3) - f_2}{\Delta x_3} \\ \frac{f_3(x_1+\Delta x_1, x_2, x_3) - f_3}{\Delta x_1} & \frac{f_3(x_1, x_2+\Delta x_2, x_3) - f_3}{\Delta x_2} & \frac{f_3(x_1, x_2, x_3+\Delta x_3) - f_3}{\Delta x_3} \end{pmatrix}$$

It is, therefore, necessary to perturb each of the three states three times and to evaluate the derivative function three times. For a system with n states, this method perturbs the states n times.

By applying the sparsity pattern and perturbing states x_1 and x_2 together, this matrix reduces to:

$$J_x = \begin{pmatrix} \frac{f_1(x_1+\Delta x_1, x_2+\Delta x_2, x_3) - f_1}{\Delta x_1} & 0 & \frac{f_1(x_1, x_2, x_3+\Delta x_3) - f_1}{\Delta x_3} \\ 0 & \frac{f_2(x_1+\Delta x_1, x_2+\Delta x_2, x_3) - f_2}{\Delta x_2} & 0 \\ 0 & \frac{f_3(x_1+\Delta x_1, x_2+\Delta x_2, x_3) - f_3}{\Delta x_2} & 0 \end{pmatrix}$$

The solver can now solve columns 1 and 2 in one sweep. While the sparse perturbation method saves significant computation, it also adds overhead to compilation. It might even slow down the simulation if the system does not have a large number of continuous states. A tipping point exists for which you obtain increased performance by applying this method. In general, systems having a large number of continuous states are usually sparse and benefit from the sparse method.

The sparse perturbation method, like the sparse analytical method, uses UMFPACK to perform linear algebraic operations. Also, the sparse perturbation method supports both RSim and rapid accelerator mode.

Full and Sparse Analytical Methods

The full and sparse analytical methods attempt to improve performance by calculating the Jacobian using analytical equations rather than the perturbation equations. The sparse analytical method, also uses the sparsity information to accelerate the linear algebraic operations required to solve the ordinary differential equations.

For details on how to access and interpret the sparsity pattern in MATLAB, see “Exploring the Solver Jacobian Structure of a Model”.

Code Generation Support

While the sparse perturbation method supports RSim, the sparse analytical method does not. Consequently, regardless of which sparse method you select, any generated code uses the sparse perturbation method. This limitation applies to rapid accelerator mode as well.

Variable Step Solvers in Simulink

Variable-step solvers vary the step size during the simulation, reducing the step size to increase accuracy when model states are changing rapidly and increasing the step size to avoid taking unnecessary steps when model states are changing slowly. Computing the step size adds to the computational overhead at each step but can reduce the total number of steps, and hence simulation time, required to maintain a specified level of accuracy for models with rapidly changing or piecewise continuous states.

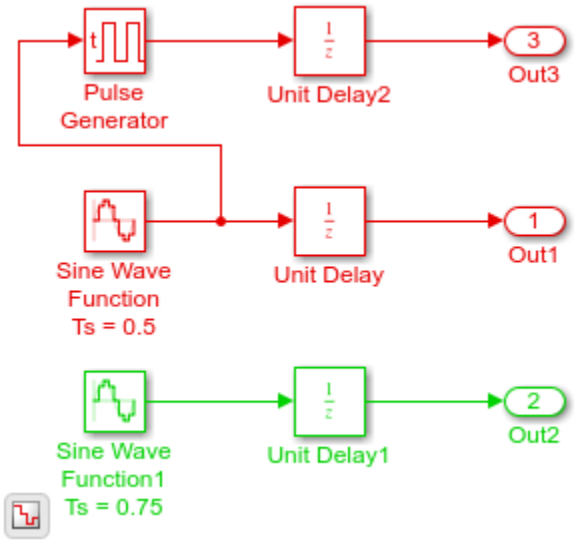
When you set the **Type** control of the **Solver** configuration pane to `Variable-step`, the **Solver** control allows you to choose one of the variable-step solvers. As with fixed-step solvers, the set of variable-step solvers comprises a discrete solver and a collection of continuous solvers. However, unlike the fixed-step solvers, the step size varies dynamically based on the local error.

The choice between the two types of variable-step solvers depends on whether the blocks in your model define states and, if so, the type of states that they define. If your model defines no states or defines only discrete states, select the discrete solver. If the model has continuous states, the continuous solvers use numerical integration to compute the values of the continuous states at the next time step.

Note If a model has no states or only discrete states, Simulink uses the discrete solver to simulate the model even if you specify a continuous solver.

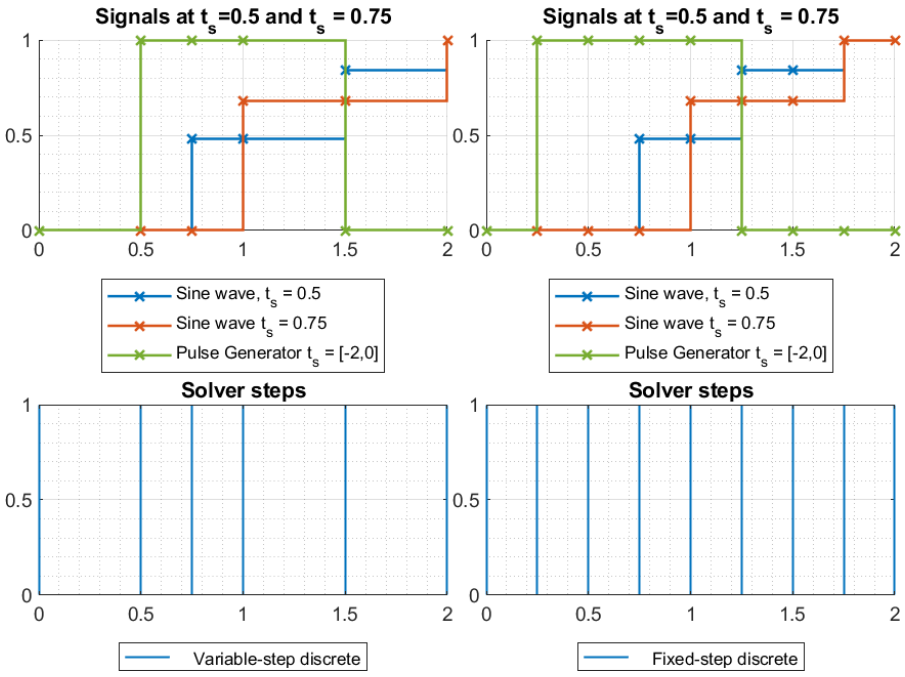
Variable-Step Discrete Solver

Use the variable-step discrete solver when your model does not contain continuous states. For such models, the variable-step discrete solver reduces its step size in order to capture model events such as zero-crossings, and increases the step size when it is possible to improve simulation performance.



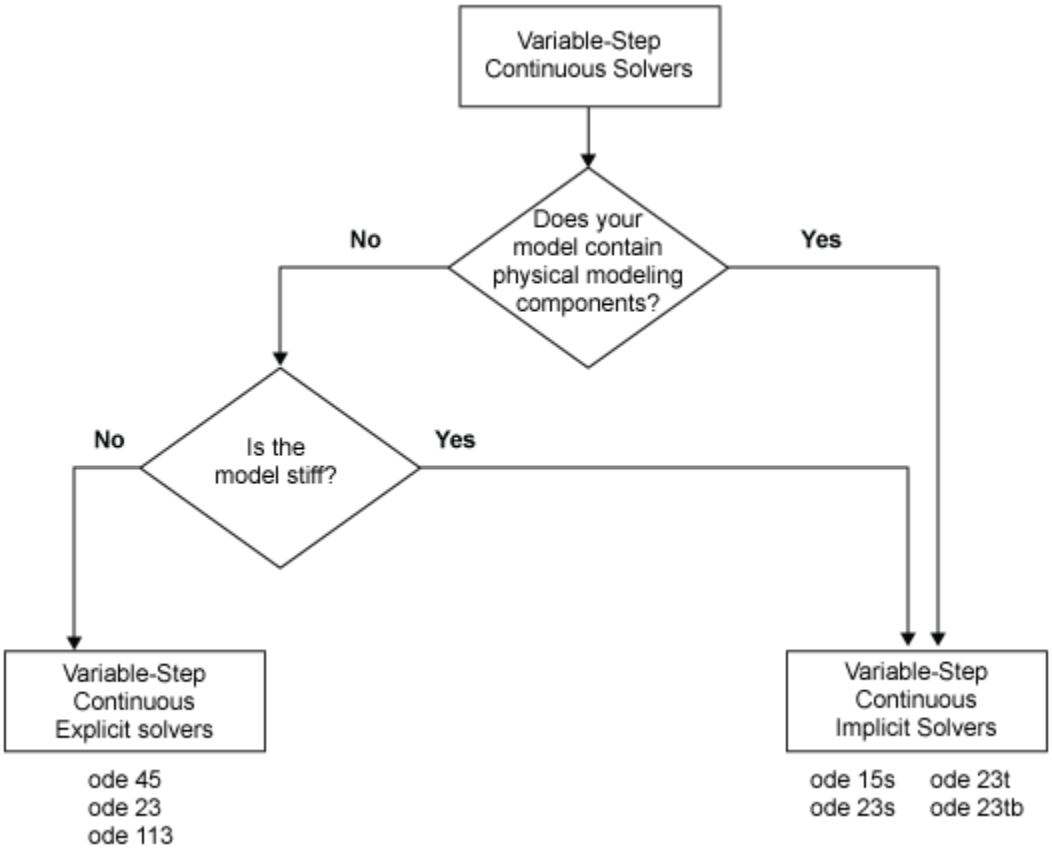
The model shown in the figure contains two discrete sine wave signals at 0.5 and 0.75 sample times. The graphs below show the signals in the model along with the solver steps for the variable-step discrete and the fixed-step discrete solvers respectively. You can see that the variable-step solver only

takes the steps needed to record the output signal from each block. On the other hand, the fixed-step solver will need to simulate with a fixed-step size—or fundamental sample time—of 0.25 to record all the signals, thus taking more steps overall.



Variable-Step Continuous Solvers

Variable-step solvers dynamically vary the step size during the simulation. Each of these solvers increases or reduces the step size using its local error control to achieve the tolerances that you specify. Computing the step size at each time step adds to the computational overhead. However, it can reduce the total number of steps, and the simulation time required to maintain a specified level of accuracy.



You can further categorize the variable-step continuous solvers as one-step or multistep, single-order or variable-order, and explicit or implicit. See “One-Step Versus Multistep Continuous Solvers” on page 3-8 for more information.

Variable-Step Continuous Explicit Solvers

The variable-step explicit solvers are designed for nonstiff problems. Simulink provides four such solvers:

- `ode45`
- `ode23`
- `ode113`
- `odeN`

ODE Solver	One-Step Method	Multistep Method	Order of Accuracy	Method
`ode45`	X		Medium	Runge-Kutta, Dormand-Prince (4,5) pair
`ode23`	X		Low	Runge-Kutta (2,3) pair of Bogacki & Shampine

ODE Solver	One-Step Method	Multistep Method	Order of Accuracy	Method
ode113		X	Variable, Low to High	PECE Implementation of Adams-Bashforth-Moulton
odeN	X		See **Order of Accuracy** in "Fixed-Step Continuous Explicit Solvers" on page 25-22	See **Integration Technique** in "Fixed-Step Continuous Explicit Solvers" on page 25-22

ODE Solver	When to Use
ode45	In general, the ode45 solver is the best to apply as a first try for most problems. The Runge-Kutta (4,5) solver is a fifth-order method that performs a fourth-order estimate of the error. This solver also uses a fourth-order interpolant, which allows for event location and smoother plots. If the ode45 is computationally slow, the problem may be stiff and thus require an implicit solver.
ode113	For problems with stringent error tolerances or for computationally intensive problems, the Adams-Bashforth-Moulton PECE solver can be more efficient than ode45.
ode23	The ode23 can be more efficient than the ode45 solver at crude error tolerances and in the presence of mild stiffness. This solver provides accurate solutions by applying a cubic Hermite interpolation to the values and slopes computed at the ends of a step.
odeN	The odeN solver uses a nonadaptive Runge-Kutta integration whose order is determined by the **Solver order** parameter. odeN uses a fixed step size determined by the **Max step size** parameter, but the step size can be reduced to capture certain solver events, such as zero-crossings and discrete sample hits..

Note Select the odeN solver when simulation speed is important, for example, when

- The model contains lots of zero-crossings and/or solver resets
- The **Solver Profiler** does not detect any failed steps when profiling the model

Variable-Step Continuous Implicit Solvers

If your problem is stiff, try using one of the variable-step implicit solvers:

- ode15s
- ode23s
- ode23t
- ode23tb

ODE Solver	One-Step Method	Multistep Method	Order of Accuracy	Solver Reset Method	Max. Order	Method
ode15s		X	Variable, low to medium	X	X	Numerical Differentiation Formulas (NDFs)
ode23s	X		Low			Second-order, modified Rosenbrock formula
ode23t	X		Low	X		Trapezoidal rule using interpolant
ode23tb	X		Low	X		TR-BDF2

Solver Reset Method

For `ode15s`, `ode23t`, and `ode23tb` a drop-down menu for the **Solver reset method** appears in the **Solver details** section of the Configuration pane. This parameter controls how the solver treats a reset caused, for example, by a zero-crossing detection. The options allowed are `Fast` and `Robust`. `Fast` specifies that the solver does not recompute the Jacobian for a solver reset, whereas `Robust` specifies that the solver does. Consequently, the `Fast` setting is computationally faster but it may use a small step size in certain cases. To test for such cases, run the simulation with each setting and compare the results. If there is no difference in the results, you can safely use the `Fast` setting and save time. If the results differ significantly, try reducing the step size for the fast simulation.

Maximum Order

For the `ode15s` solver, you can choose the maximum order of the numerical differentiation formulas (NDFs) that the solver applies. Since the `ode15s` uses first- through fifth-order formulas, the `Maximum order` parameter allows you to choose orders 1 through 5. For a stiff problem, you may want to start with order 2.

Tips for Choosing a Variable-Step Implicit Solver

The following table provides tips for the application of variable-step implicit solvers. For an example comparing the behavior of these solvers, see “Exploring Variable-Step Solvers Using a Stiff Model”.

ODE Solver	Tips on When to Use
ode15s	`ode15s` is a variable-order solver based on the numerical differentiation formulas (NDFs). NDFs are related to, but are more efficient than the backward differentiation formulas (BDFs), which are also known as Gear's method. The `ode15s` solver numerically generates the Jacobian matrices. If you suspect that a problem is stiff, or if `ode45` failed or was highly inefficient, try `ode15s`. As a rule, start by limiting the maximum order of the NDFs to 2.
ode23s	`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it can be more efficient than `ode15s` at crude tolerances. Like `ode15s`, `ode23s` numerically generates the Jacobian matrix for you. However, it can solve certain kinds of stiff problems for which `ode15s` is not effective.
ode23t	The `ode23t` solver is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if your model is only moderately stiff and you need a solution without numerical damping. (Energy is not dissipated when you model oscillatory motion.)

ODE Solver	Tips on When to Use
ode23tb	`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with two stages. The first stage is a trapezoidal rule step while the second stage uses a backward differentiation formula of order 2. By construction, the method uses the same iteration matrix in evaluating both stages. Like `ode23s`, this solver can be more efficient than `ode15s` at crude tolerances.

Note For a stiff problem, solutions can change on a time scale that is very small as compared to the interval of integration, while the solution of interest changes on a much longer time scale. Methods that are not designed for stiff problems are ineffective on intervals where the solution changes slowly because these methods use time steps small enough to resolve the fastest possible change. For more information, see Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, 1994.

Error Tolerances for Variable-Step Solvers

Local Error

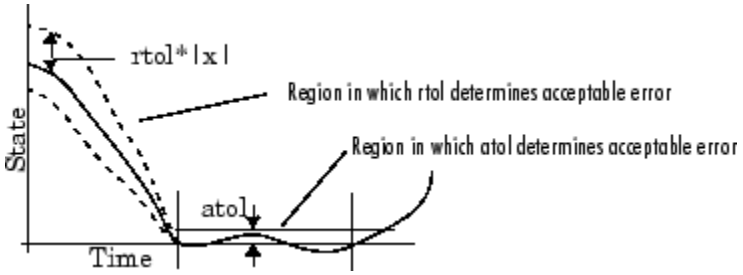
The variable-step solvers use standard control techniques to monitor the local error at each time step. During each time step, the solvers compute the state values at the end of the step and determine the *local error*—the estimated error of these state values. They then compare the local error to the *acceptable error*, which is a function of both the relative tolerance (*rtol*) and the absolute tolerance (*atol*). If the local error is greater than the acceptable error for *any one* state, the solver reduces the step size and tries again.

- Relative tolerance measures the error relative to the size of each state. The relative tolerance represents a percentage of the state value. The default, 1e-3, means that the computed state is accurate to within 0.1%.
- Absolute tolerance is a threshold error value. This tolerance represents the acceptable error as the value of the measured state approaches zero.

  The solvers require the error for the `i`th state, e_i, to satisfy:

  $$e_i \le \max(rtol \times |x_i|, atol_i) .$$

The following figure shows a plot of a state and the regions in which the relative tolerance and the absolute tolerance determine the acceptable error.



Absolute Tolerances

Your model has a global absolute tolerance that you can set on the Solver pane of the Configuration Parameters dialog box. This tolerance applies to all states in the model. You can specify `auto` or a

real scalar. If you specify `auto` (the default), Simulink initially sets the absolute tolerance for each state based on the relative tolerance. If the relative tolerance is larger 1e-3, `abstol` is initialized at 1e-6. However, for `reltol` smaller than 1e-3, `abstol` for the state is initialized at `reltol * 1e-3`. As the simulation progresses, the absolute tolerance for each state resets to the maximum value that the state has assumed so far, times the relative tolerance for that state. Thus, if a state changes from 0 to 1 and `reltol` is 1e-3, `abstol` initializes at 1e-6 and by the end of the simulation reaches 1e-3 also. If a state goes from 0 to 1000, then `abstol` changes to 1.

Now, if the state changes from 0 to 1 and `reltol` is set at 1e-4, then `abstol` initializes at 1e-7 and by the end of the simulation reaches a value of 1e-4.

If the computed initial value for the absolute tolerance is not suitable, you can determine an appropriate value yourself. You might have to run a simulation more than once to determine an appropriate value for the absolute tolerance. You can also specify if the absolute tolerance should adapt similarly to its `auto` setting by enabling or disabling the `AutoScaleAbsTol` parameter. For more information, see “Auto scale absolute tolerance”.

Several blocks allow you to specify absolute tolerance values for solving the model states that they compute or that determine their output:

- Integrator
- Second-Order Integrator
- Variable Transport Delay
- Transfer Fcn
- State-Space
- Zero-Pole

The absolute tolerance values that you specify for these blocks override the global settings in the Configuration Parameters dialog box. You might want to override the global setting if, for example, the global setting does not provide sufficient error control for all of your model states because they vary widely in magnitude. You can set the block absolute tolerance to:

- `auto`
- `-1` (same as `auto`)
- `positive scalar`
- `real vector` (having a dimension equal to the number of corresponding continuous states in the block)

Tips

If you do choose to set the absolute tolerance, keep in mind that too low of a value causes the solver to take too many steps in the vicinity of near-zero state values. As a result, the simulation is slower.

On the other hand, if you set the absolute tolerance too high, your results can be inaccurate as one or more continuous states in your model approach zero.

Once the simulation is complete, you can verify the accuracy of your results by reducing the absolute tolerance and running the simulation again. If the results of these two simulations are satisfactorily close, then you can feel confident about their accuracy.

Fixed Step Solvers in Simulink

Fixed-step solvers solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as the step size. You can specify the step size or let the solver choose the step size. Generally, a smaller the step size increases the accuracy of the results but also increases the time required to simulate the system.

Fixed-Step Discrete Solver

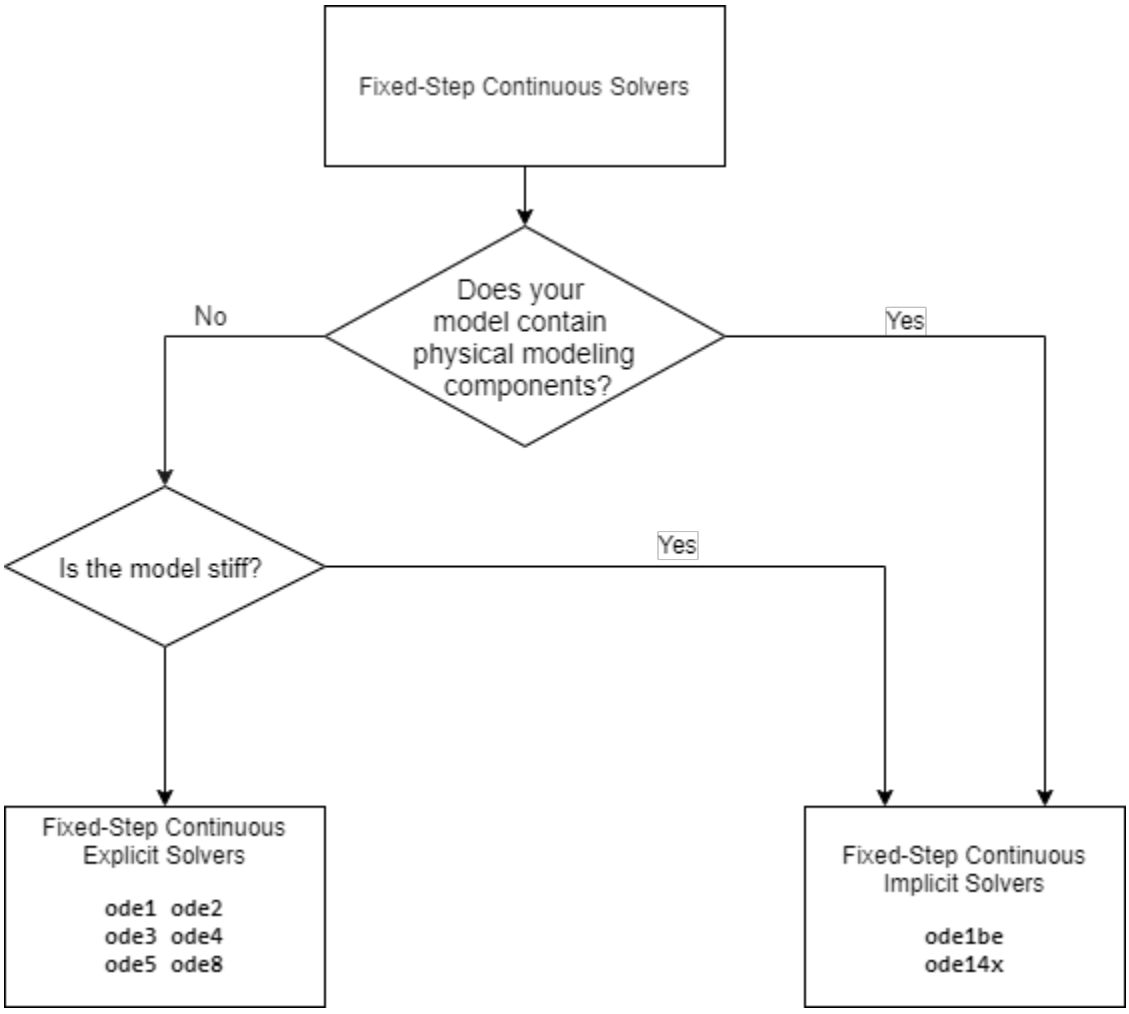
The fixed-step discrete solver computes the time of the next simulation step by adding a fixed step size to the current time. The accuracy and the length of time of the resulting simulation depends on the size of the steps taken by the simulation: the smaller the step size, the more accurate the results are but the longer the simulation takes. By default, Simulink chooses the step size or you can choose the step size yourself. If you choose the default setting of `auto`, and if the model has discrete sample times, then Simulink sets the step size to the fundamental sample time of the model. Otherwise, if no discrete rates exist, Simulink sets the size to the result of dividing the difference between the simulation start and stop times by 50.

Fixed-Step Continuous Solvers

The fixed-step continuous solvers, like the fixed-step discrete solver, compute the next simulation time by adding a fixed-size time step to the current time. For each of these steps, the continuous solvers use numerical integration to compute the values of the continuous states for the model. These values are calculated using the continuous states at the previous time step and the state derivatives at intermediate points (minor steps) between the current and the previous time step.

Note Simulink uses the fixed-step discrete solver for a model that contains no states or only discrete states, even if you specify a fixed-step continuous solver for the model.

Simulink provides two types of fixed-step continuous solvers — explicit and implicit.



The difference between these two types lies in the speed and the stability. An implicit solver requires more computation per step than an explicit solver but is more stable. Therefore, the implicit fixed-step solver that Simulink provides is more adept at solving a stiff system than the fixed-step explicit solvers. For a comparison of explicit and implicit solvers, see “Explicit Versus Implicit Continuous Solvers” on page 3-8.

Fixed-Step Continuous Explicit Solvers

Explicit solvers compute the value of a state at the next time step as an explicit function of the current values of both the state and the state derivative. A fixed-step explicit solver is expressed mathematically as:

$$x(n+1) = x(n) + h * Dx(n)$$

where

- x is the state.
- Dx is a solver-dependent function that estimates the state derivative.
- h is the step size.

- *n* indicates the current time step.

Simulink provides a set of fixed-step continuous explicit solvers. The solvers differ in the specific numerical integration technique that they use to compute the state derivatives of the model. This table lists each solver and the integration technique it uses. The table lists the solvers in the order of the computational complexity of the integration methods they use, from the least complex (`ode1`) to the most complex (`ode8`).

Solver	Integration Technique	Order of Accuracy
`ode1`	Euler's method	First
`ode2`	Heun's method	Second
`ode3`	Bogacki-Shampine formula	Third
`ode4`	Fourth-order Runge-Kutta (RK4) formula	Fourth
`ode5`	Dormand-Prince (RK5) formula	Fifth
`ode8`	Dormand-Prince RK8(7) formula	Eighth

None of these solvers have an error control mechanism. Therefore, the accuracy and the duration of a simulation depend directly on the size of the steps taken by the solver. As you decrease the step size, the results become more accurate, but the simulation takes longer. Also, for any given step size, the higher the order of the solver, the more accurate the simulation results.

If you specify a fixed-step solver type for a model, then by default, Simulink selects the `FixedStepAuto` solver. Auto solver then selects an appropriate fixed-step solver that can handle both continuous and discrete states with moderate computational effort. As with the discrete solver, if the model has discrete rates (sample times), then Simulink sets the step size to the fundamental sample time of the model by default. If the model has no discrete rates, Simulink automatically uses the result of dividing the simulation total duration by 50. Consequently, the solver takes a step at each simulation time at which Simulink must update the discrete states of the model at its specified sample rates. However, it does not guarantee that the default solver accurately computes the continuous states of a model. Therefore, you may need to choose another solver, a different fixed step size, or both to achieve acceptable accuracy and an acceptable simulation time.

Fixed-Step Continuous Implicit Solvers

An implicit solver computes the state at the next time step as an implicit function of the state at the current time step and the state derivative at the next time step, as described by the following expression.

$$x(n+1) - x(n) - h * Dx(n+1) = 0$$

Simulink provides one fixed-step implicit solver: `ode14x`. This solver uses a combination of Newton's method and extrapolation from the current value to compute the value of a state at the next time step. You can specify the number of Newton's method iterations and the extrapolation order that the solver uses to compute the next value of a model state. See “Fixed-step size (fundamental sample time)”. The more iterations and the higher the extrapolation order that you select, the greater the accuracy you obtain. However, you simultaneously create a greater computational burden per step size.

See Also

“Choose a Fixed-Step Solver” on page 25-25

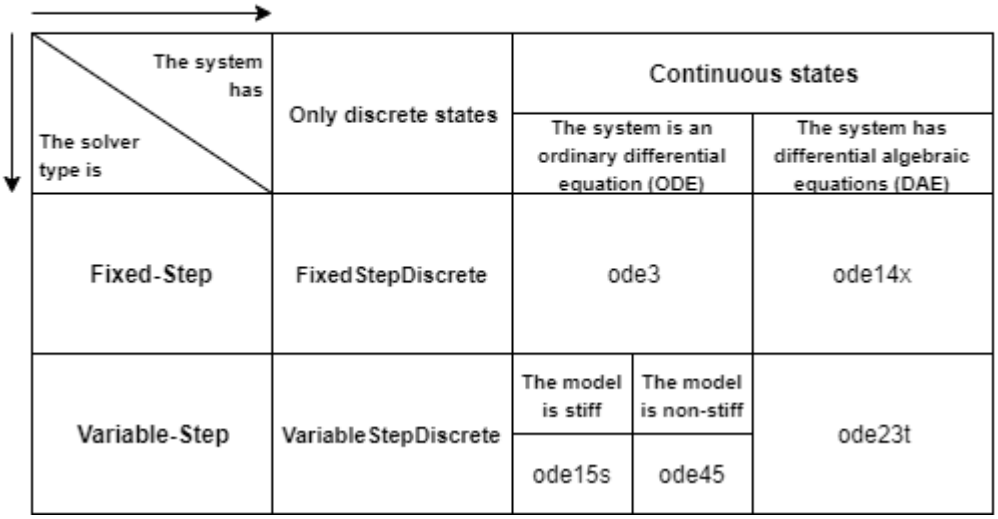
More About

- “Variable Step Solvers in Simulink” on page 25-14
- “Compare Solvers” on page 3-6

Choose a Fixed-Step Solver

This example shows an algorithmic method of selecting an appropriate fixed-step solver for your model. For simulation workflows in Simulink, the default setting for the **Solver** parameter in the Model Configuration Parameters is `auto`. The heuristics used by Simulink to select a variable-step solver is shown in the figure below.

Auto Solver Heuristics



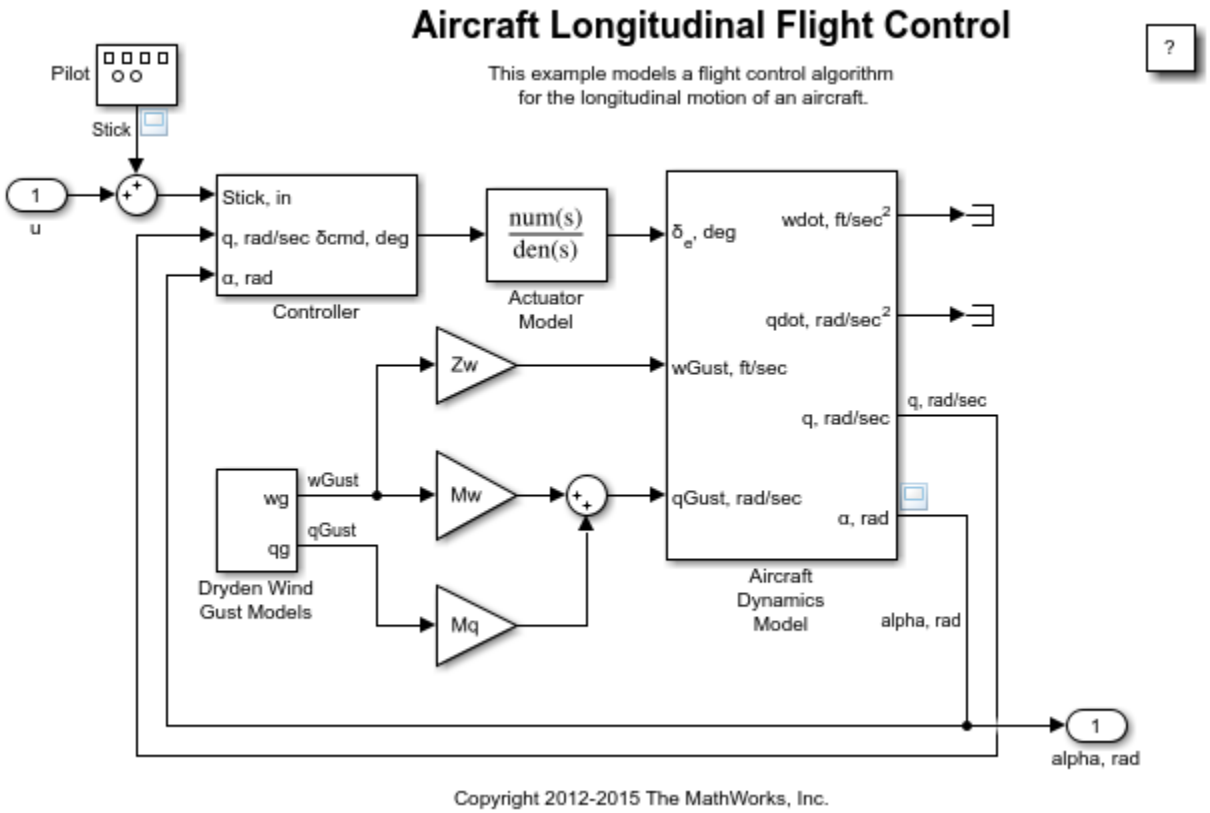
When to Use a Fixed-Step Solver

One common case to use a fixed-step solver is for workflows where you plan to generate code from your model and run the code on a real-time system.

With a variable-step solver, the step size can vary from step to step, depending on the model dynamics. In particular, a variable-step solver increases or reduces the step size to meet the error tolerances that you specify and as such, the variable step sizes cannot be mapped to the real-time clock of a target system.

Any of the fixed-step continuous solvers in the Simulink product can simulate a model to any desired level of accuracy, given a small enough step size. Unfortunately, it is not possible or practical to decide without trial, the combination of solver and step size that will yield acceptable results for the continuous states in the shortest time. Determining the best solver for a particular model generally requires experimentation.

The example model represents the flight control algorithm for the longitudinal flight of an aircraft.



Establish Baseline Results Using a Variable-Step Solver

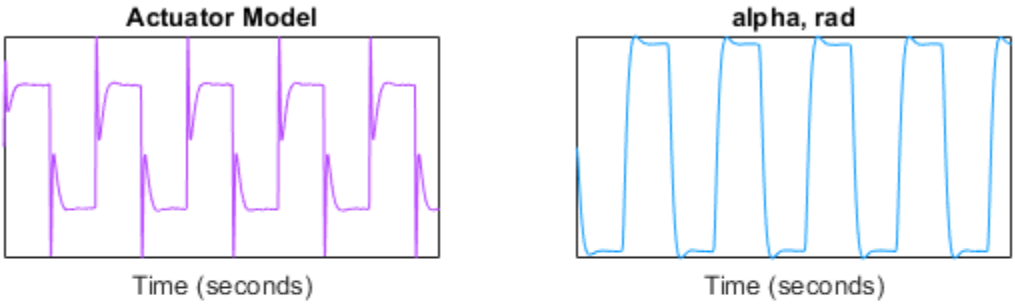
Before you begin simulation, determine acceptable error tolerances for your variable-step solver. The model is currently set up with the default values of absolute and relative tolerances of `1e-6` and `1e-4` respectively.

If these values are acceptable, continue with the example. Otherwise, you can change them to your specification using the Model Configuration Parameters.

Select a variable-step solver from the list of solvers in the **Solver** dropdown in the **Solver** pane of the Model Configuration Parameters to simulate the model. The default setting in Simulink for the **Solver** parameter is `VariableStepAuto`. Simulink selects a solver and the maximum step size of the simulation based on the contents and dynamics of the model.

Simulate the model using the `auto` solver, or pick another solver. Additionally, enable the **Save states**, **Save time,** and **Save outputs** parameters in the **Data Import/Export** pane of the Model Configuration Parameters. Set the logging format for your model to `Dataset` to allow the Simulation Data Inspector to log the signals.

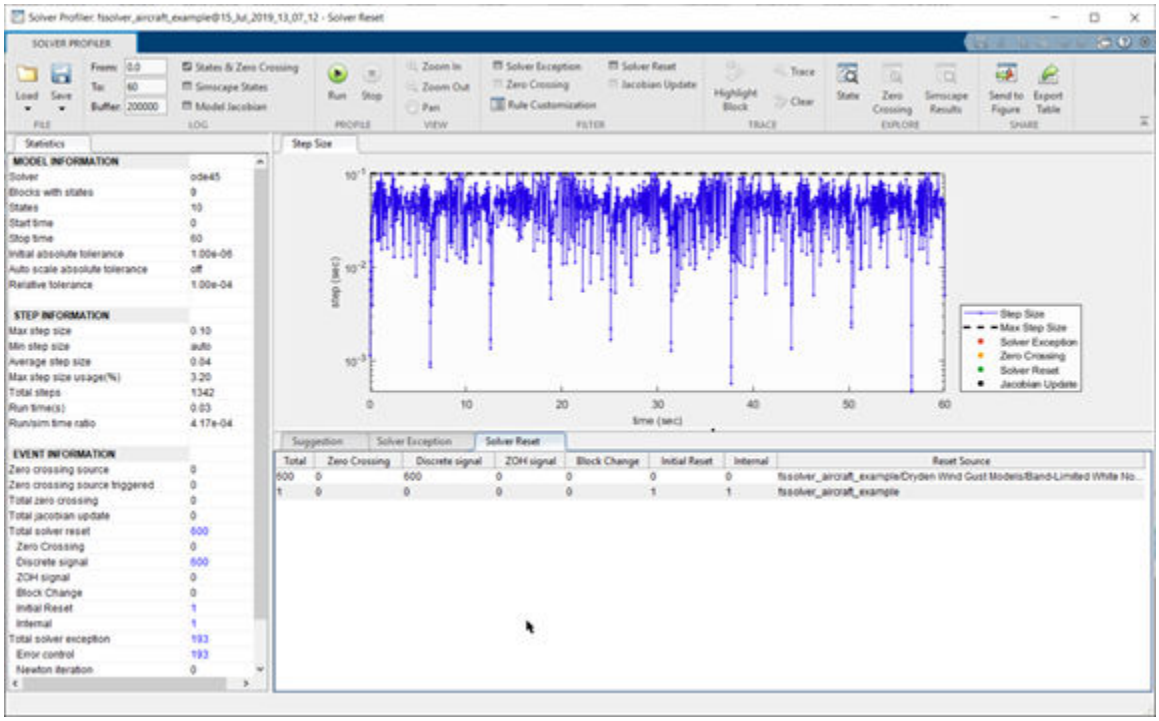
The simulation results from this run will be designated the baseline results for this task. The model contains 13 signals, but this example focuses on only a few signals plotted below.



Profile the model using the **Solver Profiler** to find an appropriate step size for the candidate fixed-step simulations of the model. See Solver Profiler for information on how to launch and use the tool. For command-line usage, see `solverprofiler.profileModel`.

Note the maximum and average step sizes returned by the **Solver Profiler.**

```
            solver: 'ode45'
            tStart: 0
             tStop: 60
            absTol: 1.0000e-06
            relTol: 1.0000e-04
              hMax: 0.1000
          hAverage: 0.0447
             steps: 1342
       profileTime: 0.0665
          zcNumber: 0
       resetNumber: 600
    jacobianNumber: 0
  exceptionNumber: 193
```



Run Fixed-Step Simulations of the Model

Once you obtain the results of the variable-step simulation of the model, simulate it using one or more of the fixed-step solvers. In this example, the model is simulated using all the non-stiff fixed-step solvers: `ode1`, `ode2`, `ode3`, `ode4`, `ode5`, and `ode8`. You can also select a specific solver from the **Solver** dropdown in the Model Configuration Parameters to run against the variable-step baseline.

Considerations for Selecting a Fixed Step Size

The optimal step size for a fixed-step simulation of your model strikes a balance between speed and accuracy, given constraints such as code-generation objectives, physics or dynamics of the model, and modeling patterns used. For example, code generation would dictate the step size must be greater than or equal to the clock speed of the processor (the reciprocal of the CPU frequency). For pure simulation objectives, the step size must be less than the discrete sample times specified by individual blocks in the model. For models with periodic signals, the step size must be such that the signal is sampled at twice its highest frequency; this is known as the Nyquist frequency.

For this specific example, set the fixed-step size of the solver to 0.1 (the maximum step size detected by the **Solver Profiler**). This takes into account the discrete sample time 0.1 of the Dryden Wind-Gust block, as well as the periodic nature of the stick movements and the aircraft response.

Make sure that the model states, outputs, and simulation time are enabled for logging and that the logging format is set to `Dataset` in the Model Configuration Parameters.

Simulate the model by selecting any one or all the non-stiff fixed-step solvers from the **Solver** dropdown of the Model Configuration Parameters when the solver **Type** is set to `Fixed-step`.

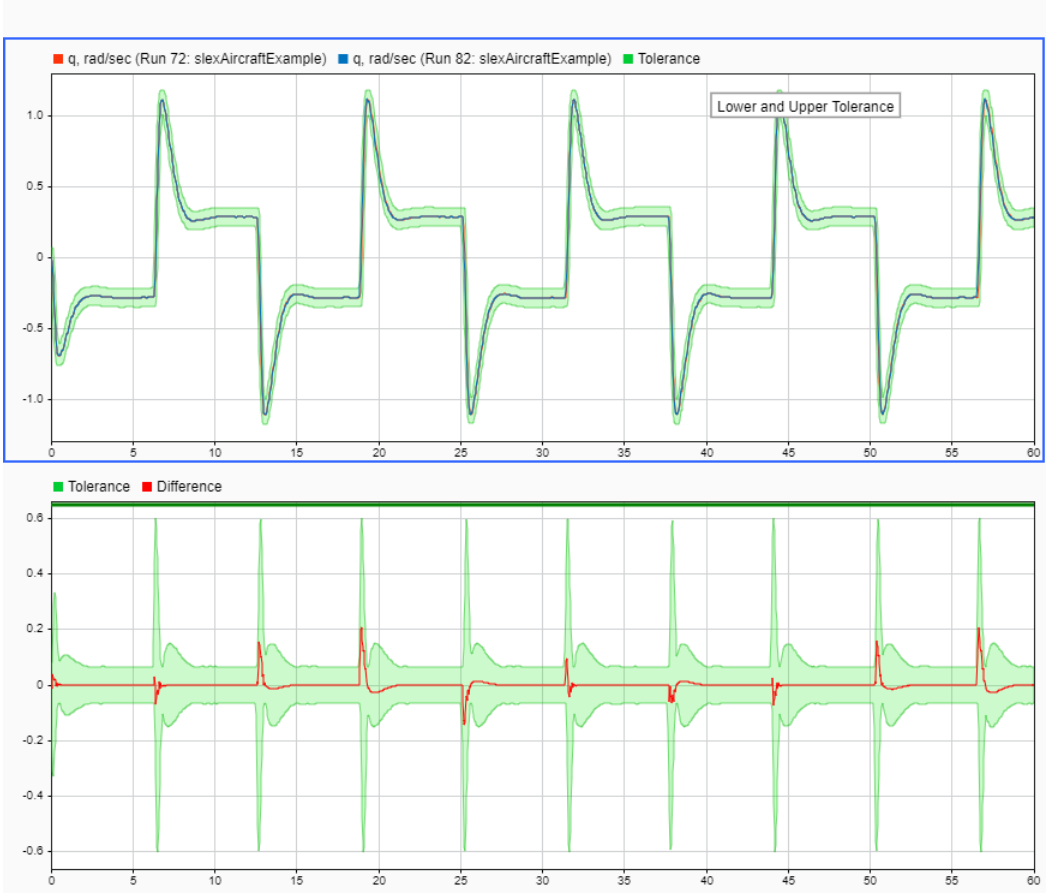
A `Simulink.sdi.Run` object is created for the fixed-step solver simulation(s) and stored in the `fsRuns` struct in the base workspace.

Compare Fixed-Step Simulations with the Variable-Step Baseline

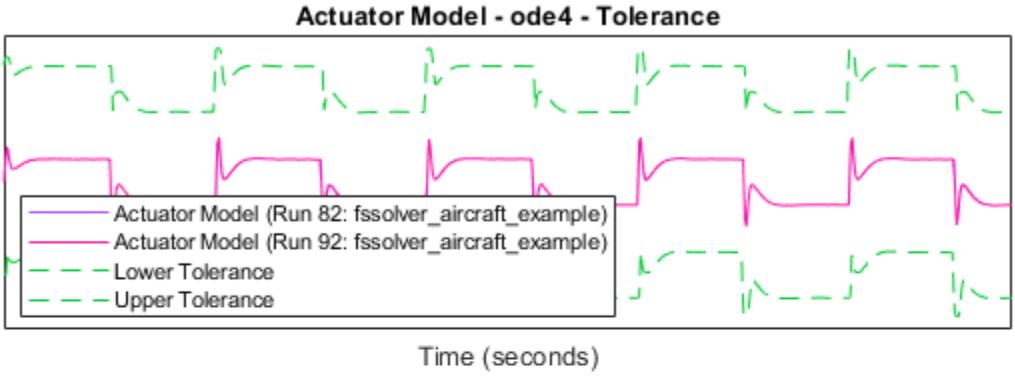
Use the **Simulation Data Inspector** to visualize and inspect logged signals in your model. You can also compare signals across simulations, or runs, using the **Compare** feature. For more information on using the **Simulation Data Inspector**, see Simulation Data Inspector. For more information on how to compare simulations using the Simulation Data Inspector, see “Compare Simulation Data” on page 29-130.

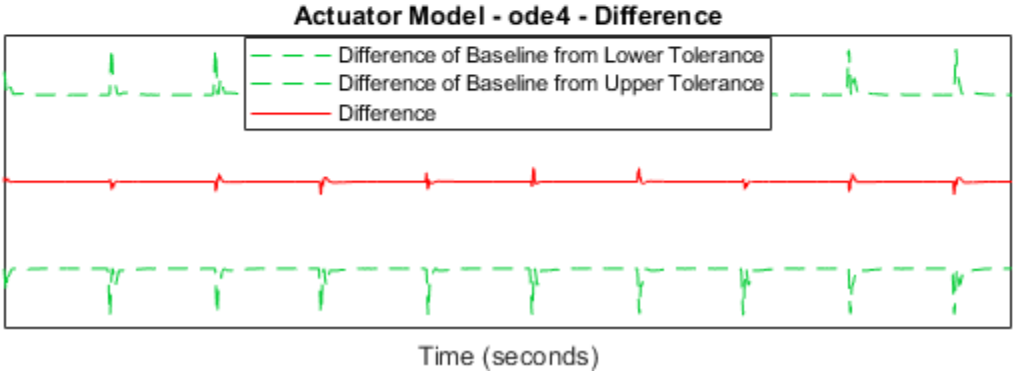
To compare signals, switch to the **Compare** tab in the **Simulation Data Inspector**. Set the **Baseline** run to the variable-step simulation and select a fixed-step simulation from the **Compare to** dropdown. Set the **Global Abs Tolerance**, **Global Rel Tolerance**, and **Global Time Tolerance** based on your requirements.

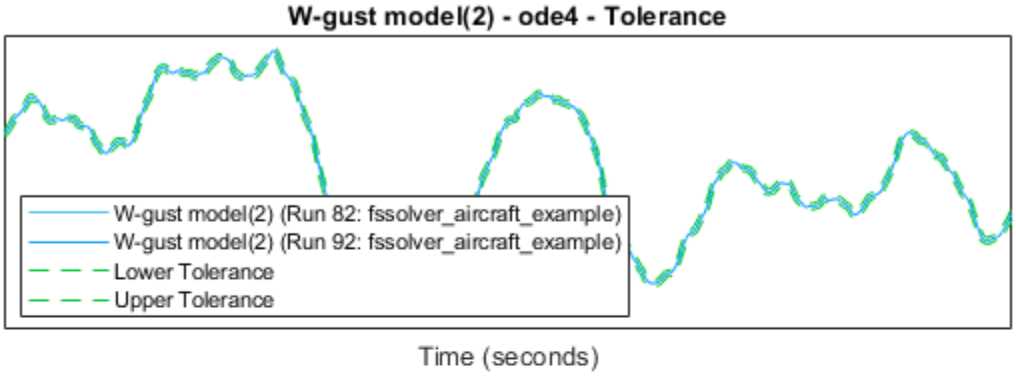
For this example, **Global Abs Tolerance** is set to `0.065`, **Global Rel Tolerance** is set to 0.005, and **Global Time Tolerance** is set to `0.1`.

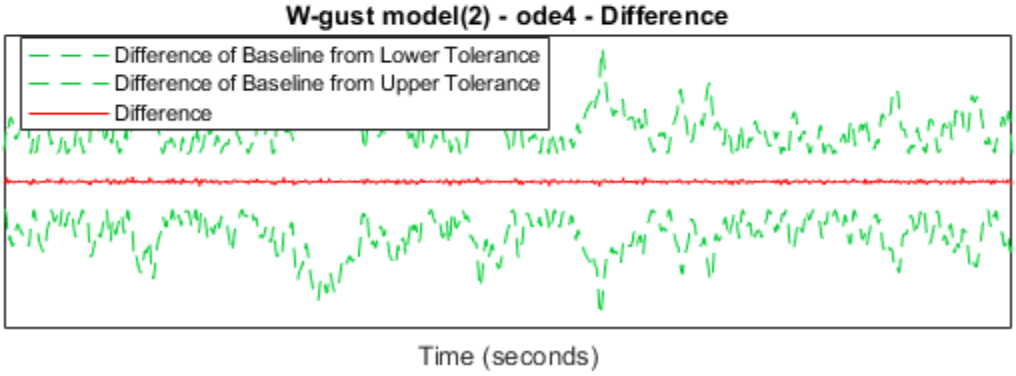


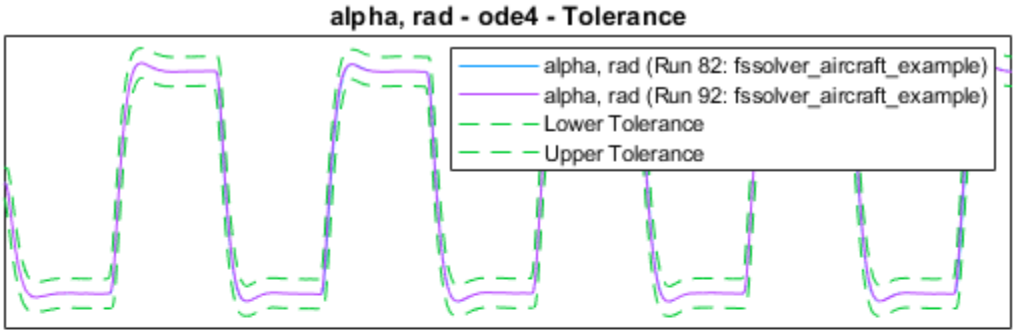
The comparison plots display the results for the lowest order fixed-step solver simulation where all signals fell within tolerance, when compared to the baseline variable-step simulation. For the selected solver, comparison results of a few of the signals are plotted below.

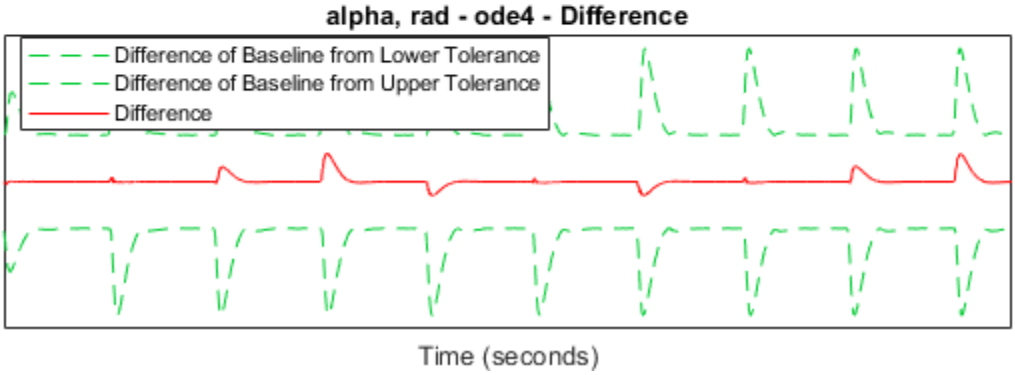

Actuator Model - ode4 - Tolerance
Actuator Model (Run 82: fssolver_aircraft_example)
Actuator Model (Run 92: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)

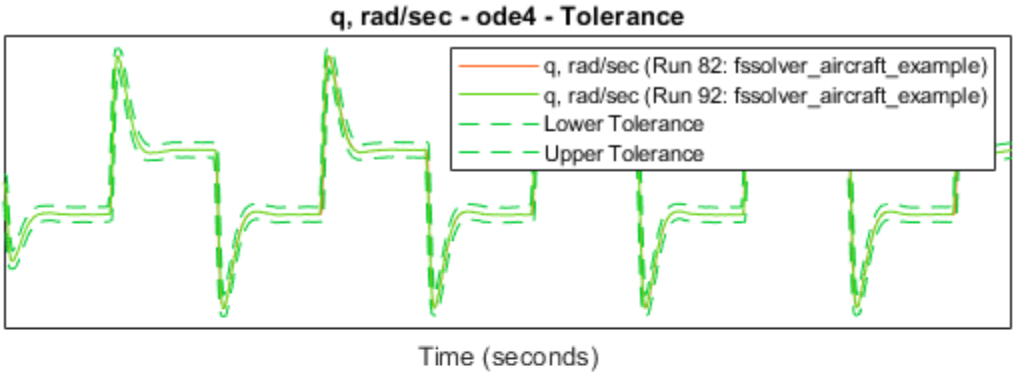

Actuator Model - ode4 - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)

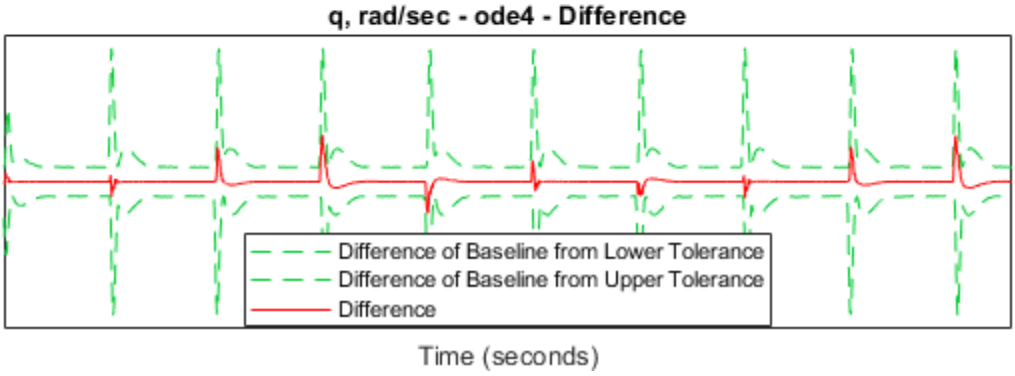

W-gust model(2) - ode4 - Tolerance
W-gust model(2) (Run 82: fssolver_aircraft_example)
W-gust model(2) (Run 92: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)


W-gust model(2) - ode4 - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)


alpha, rad - ode4 - Tolerance
alpha, rad (Run 82: fssolver_aircraft_example)
alpha, rad (Run 92: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)

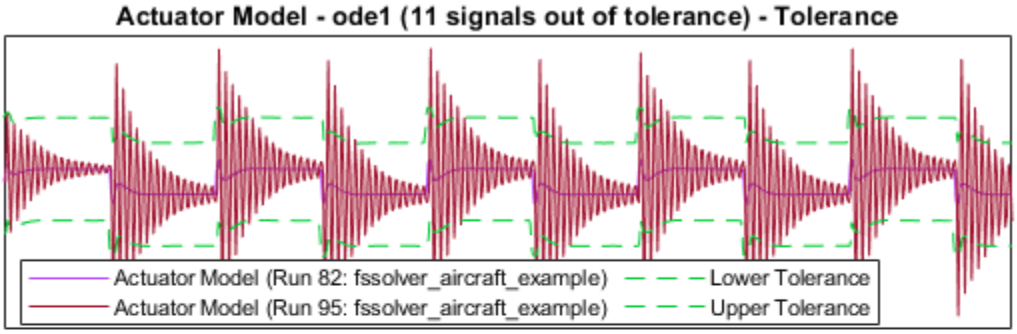

alpha, rad - ode4 - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)

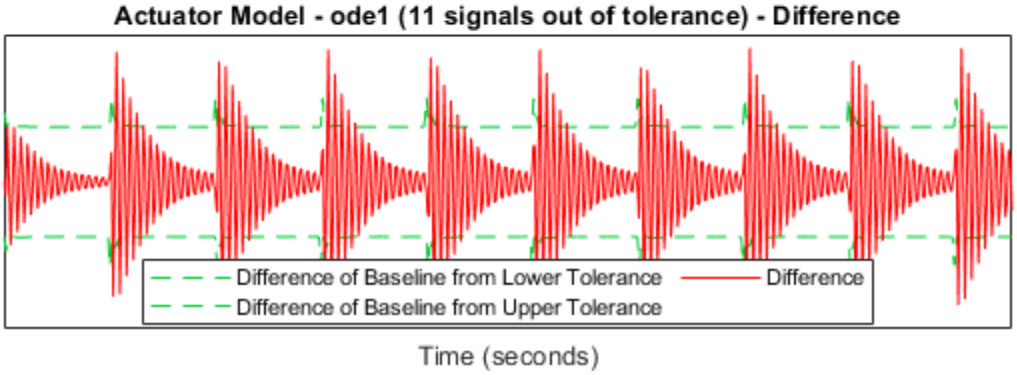


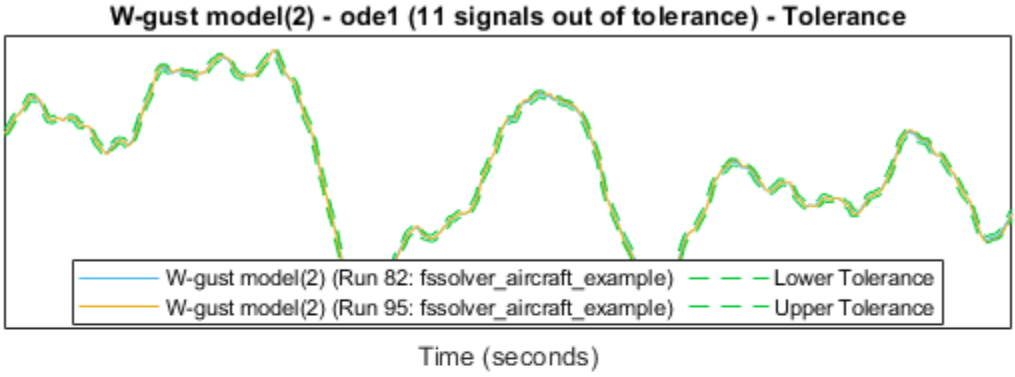


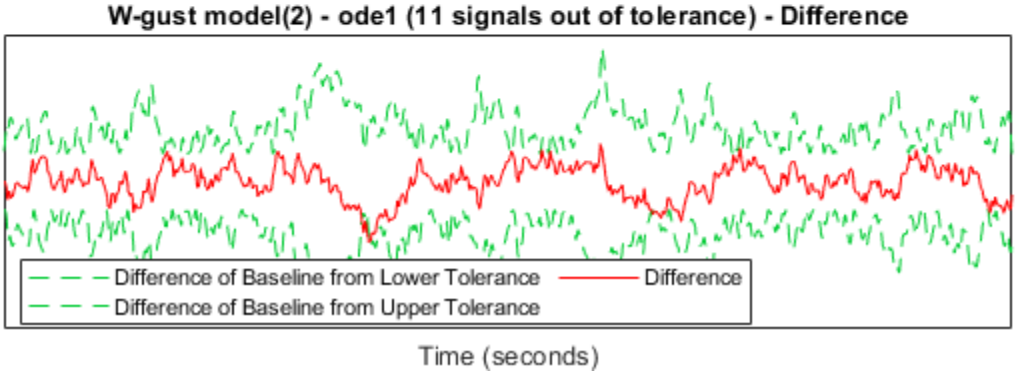
The lowest order with all signals within tolerance is determined to be `ode4`. Consider the results for the ode1 fixed-step solver, where the comparison results showed 11 signals out of tolerance. Observe that there are 11 signals out of tolerance when the signal comparison parameters are set as:

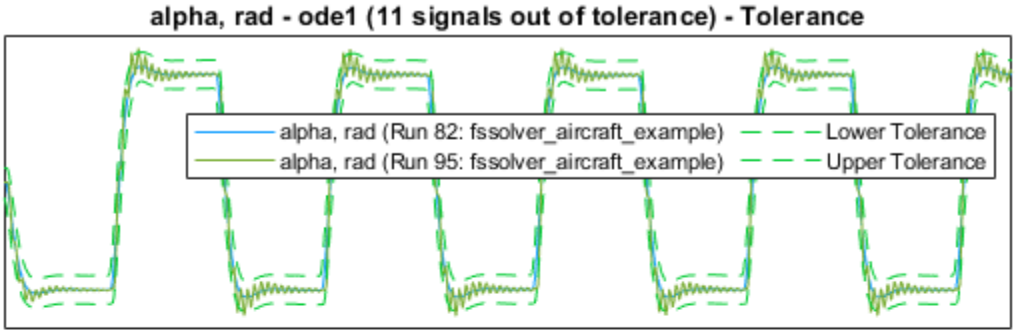
- **Signal Abs Tolerance**: 0.065
- **Signal Rel Tolerance**: 0.065
- **Signal Time Tolerance**: 0.1

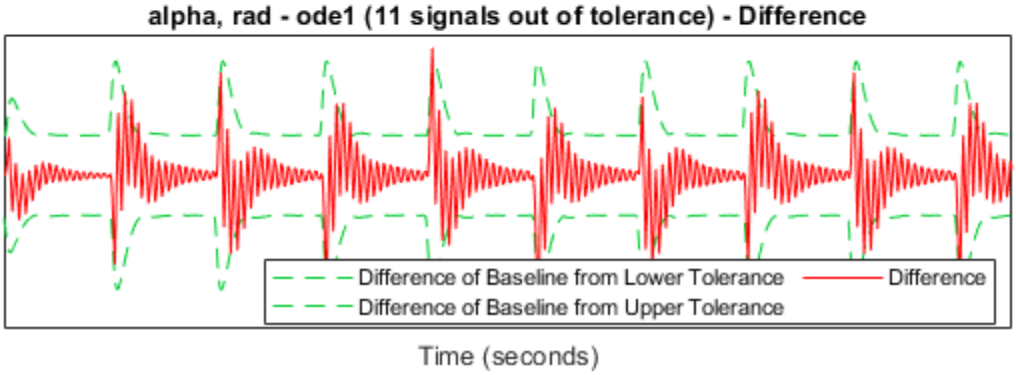

Actuator Model - ode1 (11 signals out of tolerance) - Tolerance
Actuator Model (Run 82: fssolver_aircraft_example)
Actuator Model (Run 95: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)

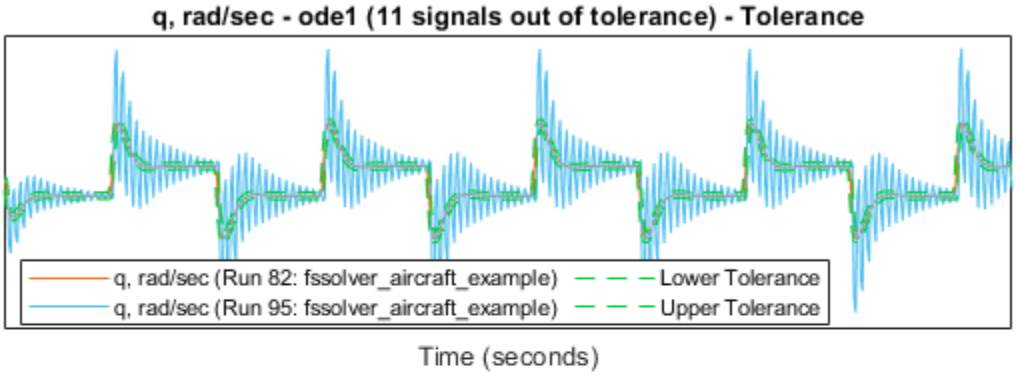

Actuator Model - ode1 (11 signals out of tolerance) - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)

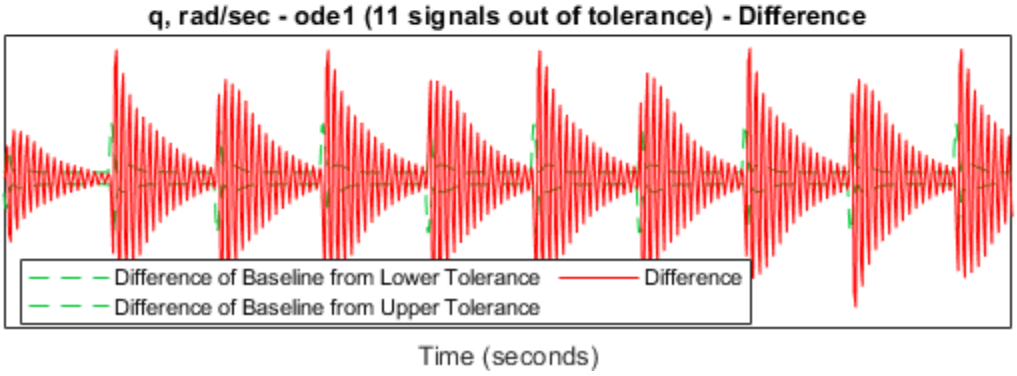

W-gust model(2) - ode1 (11 signals out of tolerance) - Tolerance
W-gust model(2) (Run 82: fssolver_aircraft_example)
W-gust model(2) (Run 95: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)


W-gust model(2) - ode1 (11 signals out of tolerance) - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)


alpha, rad - ode1 (11 signals out of tolerance) - Tolerance
alpha, rad (Run 82: fssolver_aircraft_example)
alpha, rad (Run 95: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)


alpha, rad - ode1 (11 signals out of tolerance) - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)


q, rad/sec - ode1 (11 signals out of tolerance) - Tolerance
q, rad/sec (Run 82: fssolver_aircraft_example)
q, rad/sec (Run 95: fssolver_aircraft_example)
Lower Tolerance
Upper Tolerance
Time (seconds)

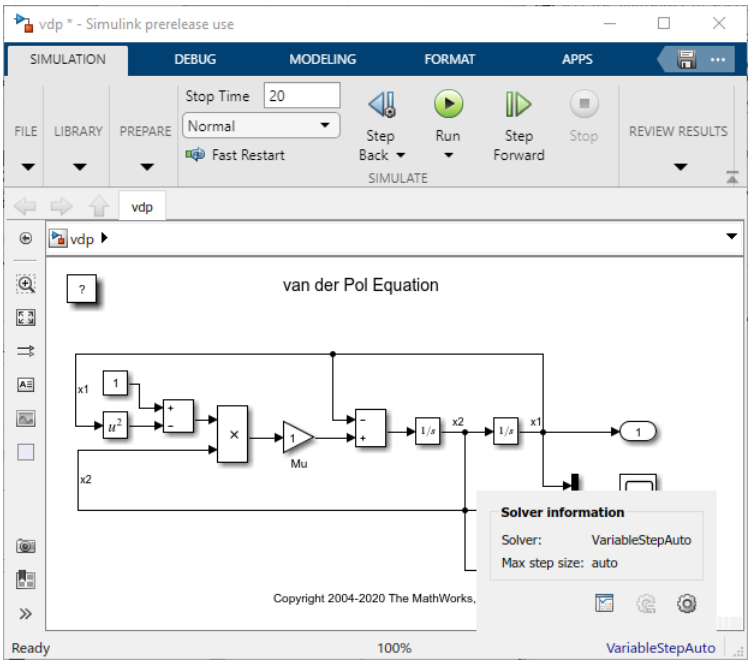

q, rad/sec - ode1 (11 signals out of tolerance) - Difference
Difference of Baseline from Lower Tolerance
Difference of Baseline from Upper Tolerance
Difference
Time (seconds)

Select Solver Using Auto Solver

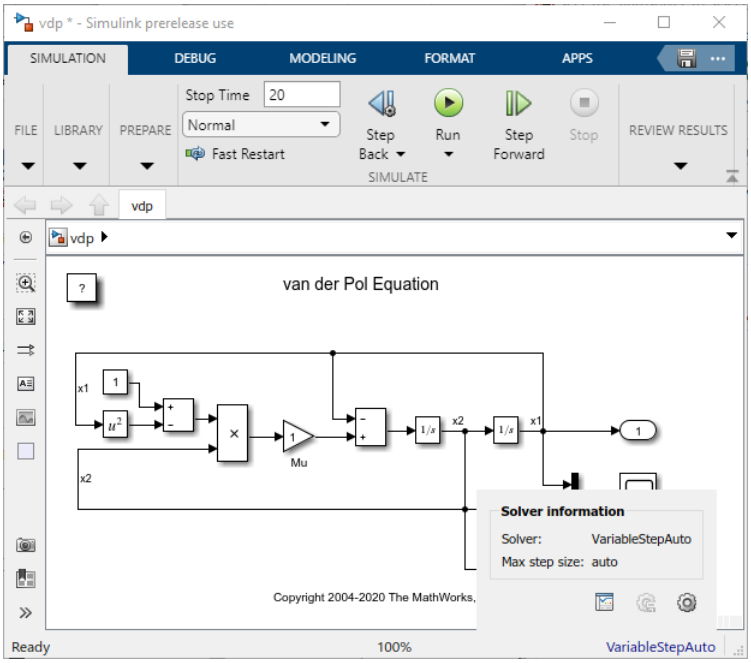
When you want Simulink to select a solver for simulating the model, use auto solver. Auto solver chooses a suitable solver and sets the maximum step size of the simulation.

For new models, Simulink selects auto solver and sets the type to variable-step by default. For an existing model, you can use auto solver to select a solver.

1. Open `vdp` and click the solver link in the lower-right corner.



2. In the **Solver Information** pane, click the **View solver settings** button to open the **Solver** pane of the model configuration parameters.
3. Under **Solver selection**, set **Type** to fixed or variable-step according to your preference and set **Solver** to `auto`.
4. When you simulate the model, auto solver selects a fixed-step or variable-step solver according to your preference and calculates the maximum step size it recommends. To see the results, open the **Solver information** pane.



5 Click the **Accept suggested settings** button to apply the recommendations of auto solver. To select different settings, click the **View solver settings** button and make changes in the configuration parameters **Solver** pane.

This chart describes the selection process of the auto solver.

Auto Solver Heuristics

The system has / The solver type is	Only discrete states	Continuous states	
		The system is an ordinary differential equation (ODE)	The system has differential algebraic equations (DAE)
Fixed-Step	FixedStepDiscrete	ode3	ode14x
Variable-Step	VariableStepDiscrete	The model is stiff: ode15s	The model is non-stiff: ode45
			ode23t

- For Simscape Electrical models, auto solver selects `ode23tb`. These systems can have circuits with nonlinear models, especially circuit breakers and power electronics. Such nonlinear models require a stiff solver.

- If the number of continuous states in the model exceeds the `NumStatesForStiffnessChecking` value, auto solver uses `ode15s`. It does not calculate the stiffness of the model. The default value for this parameter is 1000. You can change this value using `set_param`.
- If the number of continuous states in the model is less than the `NumStatesForStiffnessChecking` value, auto solver calculates the stiffness of the model. A model is stiff if the stiffness exceeds the `StiffnessThreshold` value. The default value for this parameter is 1000. You can change this value using `set_param`.

See Also

More About

- “Choose a Solver” on page 25-5
- “Compare Solvers” on page 3-6

Save and Restore Simulation Operating Point

In this section...

Note In R2019a, the `SimState` object was renamed to `ModelOperatingPoint`.

To effectively design a system, you simulate a model iteratively, so you can analyze the system based on different inputs, boundary conditions, or operating conditions. In many applications, when performing multiple simulations, a startup phase with significant dynamic behavior is common. For example, the cold start takeoff of a gas turbine engine occurs before each set of aircraft maneuvers. In multiple simulations, you ideally:

1. Simulate the startup phase once.
2. Save the simulation snapshot at the end of the startup phase.
3. Use this snapshot as the initial state for each set of conditions or maneuvers.

Use the `ModelOperatingPoint` object to save the snapshot of a simulation. Once you save the snapshot, in future simulations, restore the `ModelOperatingPoint` object and use it to set initial conditions.

The `ModelOperatingPoint` object contains information about:

- Logged states
- State of the solver and execution engine
- Zero-crossing signals for blocks that register zero crossings
- Output values of certain blocks in the model

  Simulink analyzes block connections and other information to determine whether it is using the output values effectively as state information.

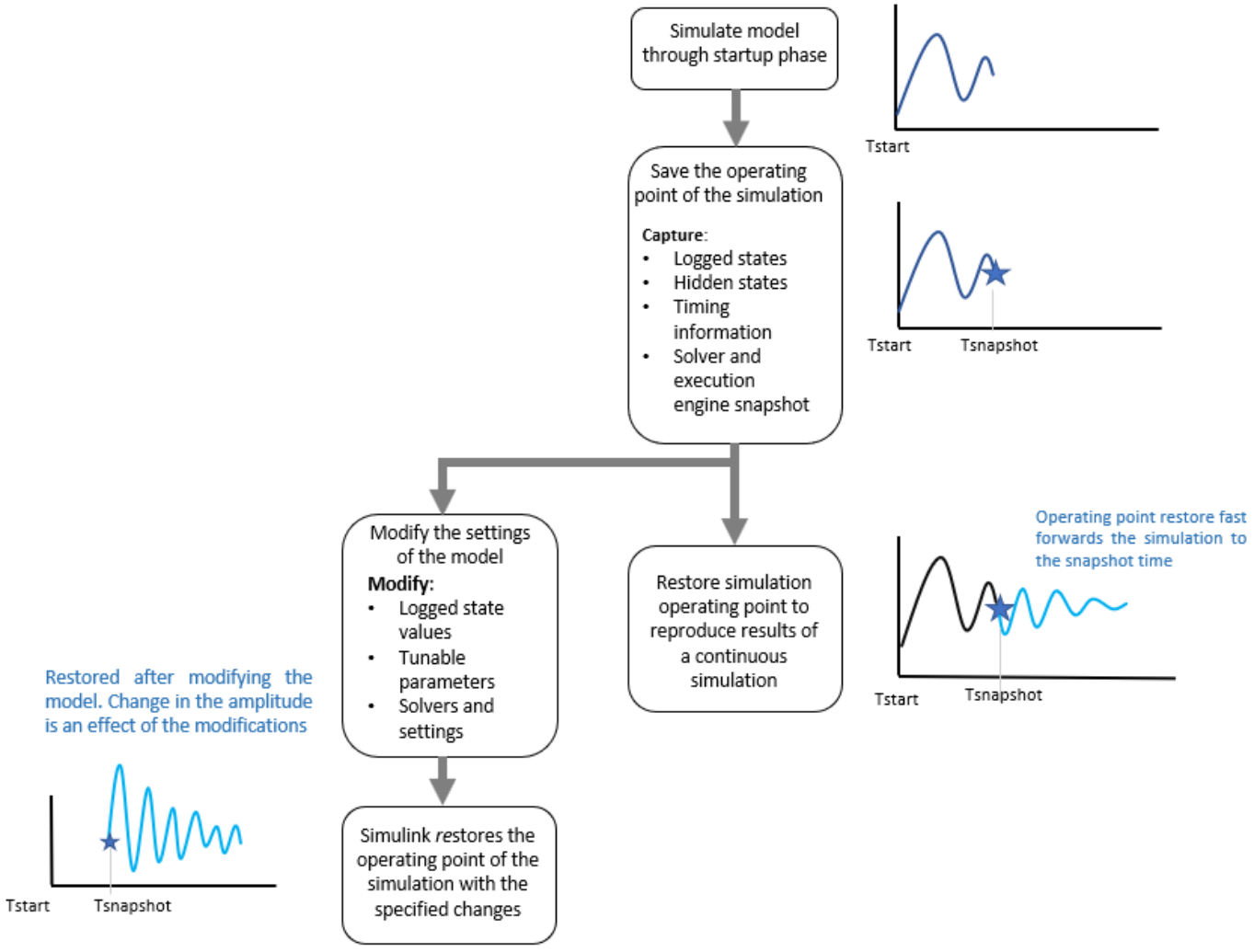
The `ModelOperatingPoint` object also stores the hidden states of these blocks:

- Transport Delay
- Variable Transport Delay
- From Workspace
- For Each subsystem
- Conditionally executed subsystems
- Stateflow
- MATLAB System
- Simscape Multibody Second Generation

By storing this information, the `ModelOperatingPoint` object ensures that the result of a simulation that starts from the operating point is the same as a simulation that runs from the beginning.

Benefits of Using Operating Point

- When the `ModelOperatingPoint` object saves the snapshot of a simulation, it saves information in addition to the logged states in the model. Restore all of this information to ensure that the simulation matches the uninterrupted simulation. For example, if solver information affected the simulation, then changing the state of a block without using `ModelOperatingPoint` can produce different results.
- You can save several operating points during a simulation, then resume the simulation from any of those operating points.
- The `ModelOperatingPoint` object restores the state of blocks that are typically difficult to restore to a particular state, for example, the Transport delay block. The state of the Transport Delay block is not saved in the structure format or the array format when you log data using the **Final states** configuration parameter.



You can also use the **Final states** option in the Configuration Parameters **Data Import/Export** pane to save a simulation state. However, this option saves only *logged* states—the continuous and discrete states of blocks. These states are only subsets of the complete simulation state of the model. They do

not include information about hidden states of blocks, which required for the proper execution of a block.

Save an Operating Point

Save an operating point at the beginning of the final step using one of these options:

- At the final **Stop time**.
- When you interrupt a simulation with the **Pause** or **Stop** button. You can also save an operating point when you pause a simulation using `get_param('modelName','CurrentOperatingPoint')`.
- When you use `set_param` or a block, like the Stop block, to stop a simulation.

Interactive Save

1. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select the **Final states** check box. The **Save final Operating Point** check box becomes available.
2. Select the **Save final Operating Point** check box.
3. In the **Final states** text box, enter a variable name for the `ModelOperatingPoint` object.
4. Simulate the model.

Programmatic Save

Use the `sim` command with `set_param`. Set the `SaveOperatingPoint` parameter to `'on'`.

```
fuelsys
set_param('fuelsys','SaveFinalState','on','FinalStateName',...
'myOperPoint','SaveOperatingPoint','on');
simOut = sim('fuelsys','StopTime','10')
myOperPoint = simOut.myOperPoint
```

Tip Before you save the operating point, disable the **Block Reduction** parameter in **Configuration Settings** > **Simulation Target** > **Advanced Parameters**.

Restore Operating Point

Restore the simulation snapshot using the `ModelOperatingPoint` object after modifying the model. The **Start time** does not change from the value in the simulation that generated the operating point. It is a reference value for all time and time-dependent variables in both the original and the current simulation. For example, a block can save and restore the number of sample time hits that occurred since the beginning of simulation as its `ModelOperatingPoint` object.

Consider a model that you ran from 0 to100 s and that you now want to run from 100 to 200 s. The **Start time** is 0 s for both the original simulation and for the current simulation. The initial time of the current simulation is 100 s. Also, if the block had 10 sample time hits during the original simulation, Simulink recognizes that the next sample time hit is the 11th, relative to 0, not 100 s.

Note If you change the **Start time** before restoring the `ModelOperatingPoint`, Simulink overwrites the **Start time** with the value saved in the `ModelOperatingPoint`.

Interactive Restore

1. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, under **Load from workspace**, select the **Initial state** check box. The text box becomes available.
2. Enter the name of the variable containing the `ModelOperatingPoint` in the text box.
3. Set the **Stop time** to a value greater than the time at which the operating point was saved.

Programmatic Restore

To configure simulation to restore the operating point that you saved in the example above:

```
set_param('fuelsys','LoadInitialState','on','InitialState',...
'myOperPoint');
myOperPoint = simOut.myOperPoint
```

The `ModelOperatingPoint` object is restored when you simulate the model.

Restore from Different Simulink Versions

You can use `ModelOperatingPoint` objects saved in releases starting with R2010a to restore the `ModelOperatingPoint` of a model. However, this option restores only the logged states of the model. To see the version of Simulink used to save the `ModelOperatingPoint`, examine the **version** parameter of the `ModelOperatingPoint` object.

Simulink detects if the `ModelOperatingPoint` object you provided as the initial state was saved in the current release. By default, Simulink displays an error message if the `ModelOperatingPoint` was not saved in the current release. You can configure the diagnostic to allow Simulink to display the message as a warning and try to restore as many of the values as possible. To enable this best-effort restoration, in the Configuration Parameters dialog box set the message for **Operating Point object from earlier release** to `warning`. Previously named `SimState` objects are loaded as `ModelOperatingPoint` objects in 19a.

See Also

More About

- “Operating Point Behavior” on page 25-45

Operating Point Behavior

Learn how you can use operating point with a block, S-functions and model changes.

Change the States of a Block Within Operating Point

- Use `loggedStates` to get or set the states of blocks. If `xout` is the state log that Simulink exports to the workspace, then the `loggedStates` field has the same structure as `xout.signals`.
- You cannot change the states that are not logged. Simulink does not allow this modification as it could cause the state to be inconsistent with the simulation.

S-Functions

You can use APIs for C-MEX and Level-2 MATLAB S-functions to enable S-functions to work with the `ModelOperatingPoint` object. For information on how to implement these APIs within S-functions, see “S-Function Compliance with the ModelOperatingPoint”.

S-functions that have PWork vectors, which store pointers to data structures but do not declare their operating point compliance level or declare it to be unknown or disallowed do not support operating point. For more information, see “S-Function Compliance with the ModelOperatingPoint”.

Model Changes and Operating Point Restore

After saving the operating point of a model, you can change the model and restore an operating point with those changes.

- You can rename a model between saving and restoring its operating point.
- The operating point interface checksum is primarily based on the configuration settings in a model and sample times used in the model. You can make some nonstructural changes to the model between saving and restoring an operating point. In the Configuration Parameters dialog box, in the **Diagnostics** pane, use the **Operating point interface checksum mismatch** diagnostic to track such changes. You can then find out if the interface checksum of the restored operating point matches the current interface checksum. See “Operating point interface checksum mismatch”.

  You can make the following nonstructural changes to the model without affecting your ability to restore a previously saved operating point:

  - Changes to model-level signal logging settings in the “Model Configuration Parameters: Data Import/Export”.
  - Logging of specific signals.
  - Addition and removal of Scope, Floating Scope and Scope Viewer, To Workspace, To File, and Display blocks.
  - Addition and removal of Level-2 MATLAB or C S-Functions that are configured as simulation viewing devices and do not set the operating point compliance to Custom or Disallowed. See “S-Function Compliance with the ModelOperatingPoint” for more information.

Note These modifications may change the number of sample times in the model. This can cause the model's interface checksum to be different from the operating point save and restore

checksum. Configure the **Operating point interface checksum mismatch diagnostic** to display a `warning` (default), `error`, or `none` to not compare the checksums.

- You cannot make structural changes to the model between the time you save the operating point and the time you restore the simulation using the operating point. Examples include, adding or removing a block after saving the operating point, changing the sample time of a model, and changing the type of solver from variable-step to fixed-step.
- Mismatches can occur when you try to simulate using a solver that is different from the one that generated the saved operating point. Simulink permits solver changes. For example, you can use the `ode15s` solver to solve the initial stiff portion of a simulation and save the final operating point. You can then continue the simulation with the restored operating point using `ode45`. In other words, this diagnostic helps you see the solver changes but does not signal a problem with the simulation.

Note When you use a variable-step solver with the maximum step size set to `auto`, Simulink uses the maximum step size from the restored `ModelOperatingPoint` object for the new simulation. To ensure that the concatenated operating point trajectory of two simulations matches that of an uninterrupted simulation, specify a value for the maximum step size.

Limitations of Saving and Restoring Operating Point

Note In some cases, saving partial state information avoids some of the limitations of using operating point. For a comparison of the two ways to save state data, see “Comparison of Operating Point and Final State Logging” on page 72-77.

Block Support

The following blocks do not support operating point:

- In the Stack and Queue blocks, the default setting for the **Push full stack** option is **Dynamic reallocation**. This default setting does not support `ModelOperatingPoint` object. Other settings (**Ignore**, **Warning** and **Error**) support `ModelOperatingPoint` object.
- Simscape Multibody First Generation blocks

Simulink tries to save the output of a block as part of an operating point. For S-functions, this happens even if the functions declare that no operating point is required. If the block output is of custom type, Simulink cannot save the operating point and displays an error. For more information about operating point use with S-functions, see “S-Functions” on page 25-45.

Model reference offers partial support for operating point. For details, see “Model Referencing” on page 25-47.

Simulation

- You can use only the normal or the accelerator simulation mode.
- You cannot save operating point in normal mode and restore it in accelerator mode, or vice versa.
- You cannot change the states of certain blocks that are not logged. For more information, see “Change the States of a Block Within Operating Point” on page 25-45.

Code Generation

The operating point feature does not support Simulink Coder or Embedded Coder code generation.

Model Referencing

- You cannot modify logged states of blocks that are inside a referenced model in accelerator mode.
- The following blocks do not support operating point when included in a model referenced in accelerator mode:
  - Level 2 MATLAB S-Function
  - MATLAB System
  - n-D Lookup Table
  - S-Function (with custom operating point or `PWork` vectors)
  - To File
  - Simscape blocks

For additional information, see “State Information for Referenced Models” on page 72-79.

Model Function

You cannot input the operating point to the `model` function.

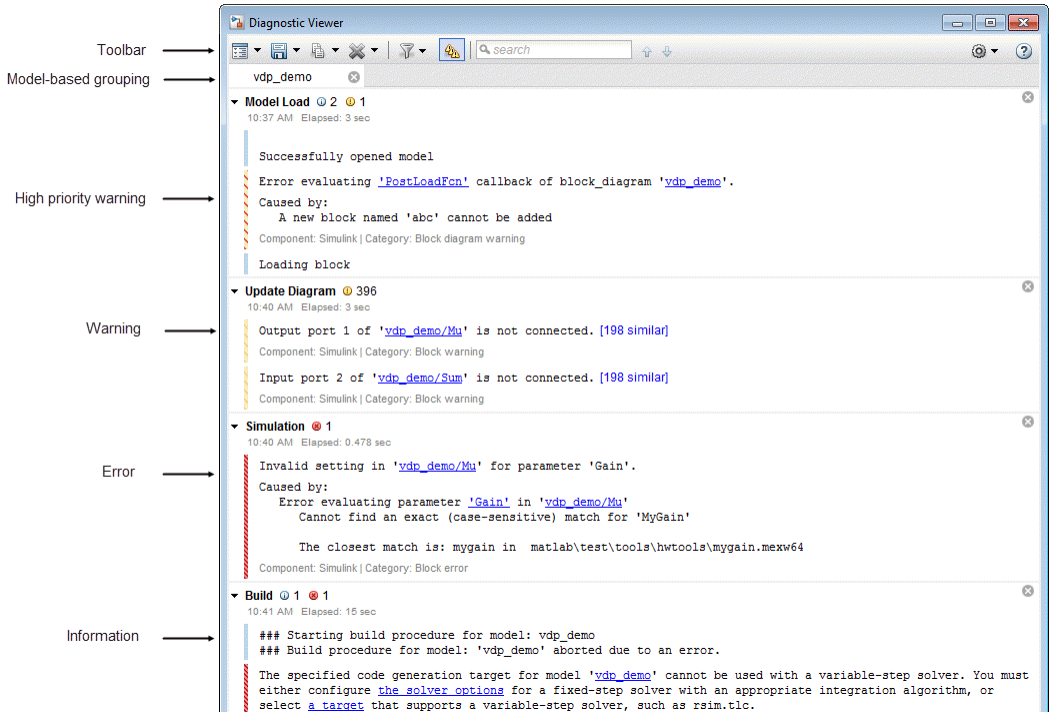
See Also

More About

- “Save and Restore Simulation Operating Point” on page 25-41

View Diagnostics

You can view and diagnose errors and warnings generated by your model using the **Diagnostic Viewer**. The **Diagnostic Viewer** displays three types of diagnostic messages: errors, warnings, and information. A model generates these messages during a runtime operation, like model load, simulation, or update diagram.



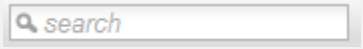
The diagnostic viewer window is divided into:

- Toolbar menu: Displays various commands to help you manage the diagnostic messages. For more information, see “Toolbar” on page 25-48.
- Diagnostic Message pane: Displays the error, warning, and information messages. For more information, see “Diagnostic Message Pane” on page 25-49.
- Suggested Actions: Displays suggestions and fixes to correct the diagnostic errors and warnings. For more information, see “Suggested Actions” on page 25-51.

Toolbar

To manage the diagnostic messages, use the **Diagnostic Viewer** toolbar.

Button	Action
	Expand or collapse messages
	Save all or latest messages in a log file
	Copy all or latest messages

Button	Action
	Clear all or all but latest messages
	Filter out errors, warning, and information messages
	Group similar type of messages
	Search messages for specific keywords and navigate between messages
	Set maximum number of models to display in tabbed panes and the maximum number of events to display per model

Diagnostic Message Pane

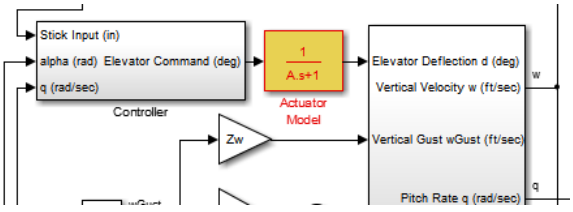
The diagnostic message pane displays the error, warning, and information messages in a tabbed format. These messages are color-coded for distinction and are hierarchical.

A new stage is generated for each successive event, you can save or clear stage. Each stage represents a single event such as model load, update diagram, or simulation.

Different types of diagnostic messages are:

- Information message: Displays the information related to a model load. Information messages are marked as  .
- High priority warning: Displays the errors encountered during model load as a high priority warning. Any subsequent operation, like update on the model without rectifying the high priority warning messages are marked as errors. High priority warnings are marked as  .
- Warning: Displays the warnings associated during an operation on a model. Warnings are marked as  .
- Error: Displays the errors associated during an operation on a model. Errors are marked as  .

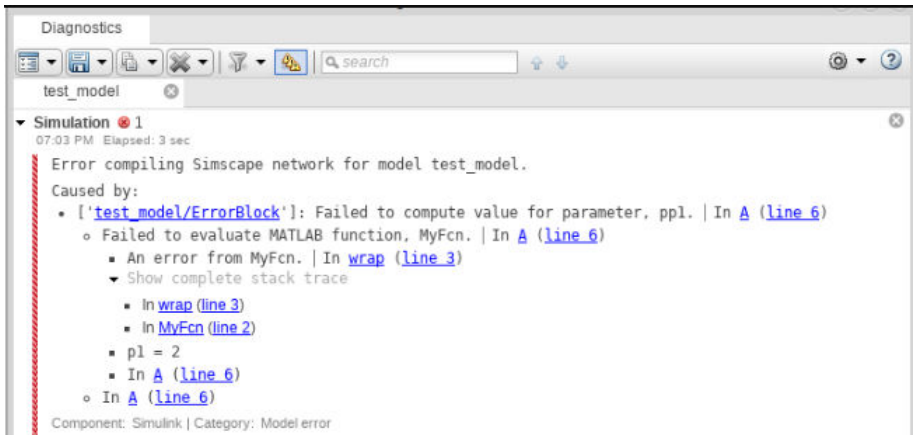
Tip To locate the source of error, click the hyperlink in the message. The source of error in the model is highlighted.



Trace Diagnostics Location

Diagnostic Viewer can trace the location of an error so that you can investigate the errors in your model easily. If the error is in a file that is being called from another file, the diagnostic shows up as

an expandable stack. You can expand or collapse the stack, as required. Expanding the stack displays information about the file and the line in which the error or warning is located. You can click any of the links to go the error or warning. You can also see the same diagnostic message with stack trace enabled, while using the `Sim` command in MATLAB.



Note Tracing the exact location of an error is not applicable for protected files.

Identify Diagnostics from Custom Compilers

Diagnostic viewer can recognize errors and warnings from builds generated by custom compilers. You can specify compiler-specific patterns using the following directives:

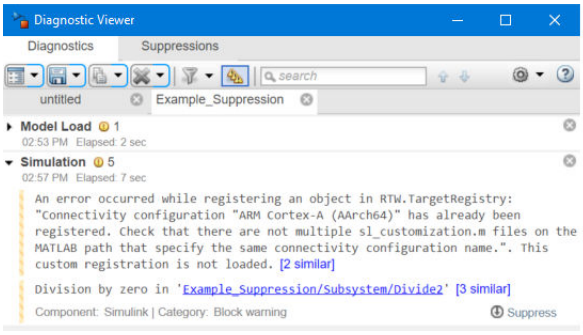
```
% Here tool is the buildtool obtained from the toolchain
tool.setDirective ('WarningPattern','warning #(\d+):'); %Specifies warning patterns
tool.setDirective ('ErrorPattern','error:'); %Specifies error pattern
tool.setDirective ('FileNamePattern','[^\s]*\w+\.(c|h)'); %Specifies file name pattern
tool.setDirective ('LineNumberPattern','\(\d+\)'); %Specifies line number pattern
```

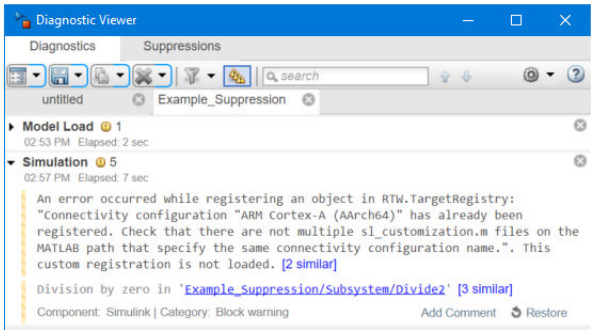
For more information about creating a `ToolchainInfo` object, see “Register Custom Toolchain and Build Executable” (Simulink Coder).

Suppress Diagnostics

The Diagnostic Viewer provides a **Suppress** button for certain diagnostics. This button allows you to suppress certain numerical diagnostics (for example, overflow, saturation, precision loss) for specific objects in your model. You can also suppress certain errors that have diagnostic level set to `error` in the **Diagnostics** section of Model Configuration Parameters. You can add a comment for the suppressed diagnostics.

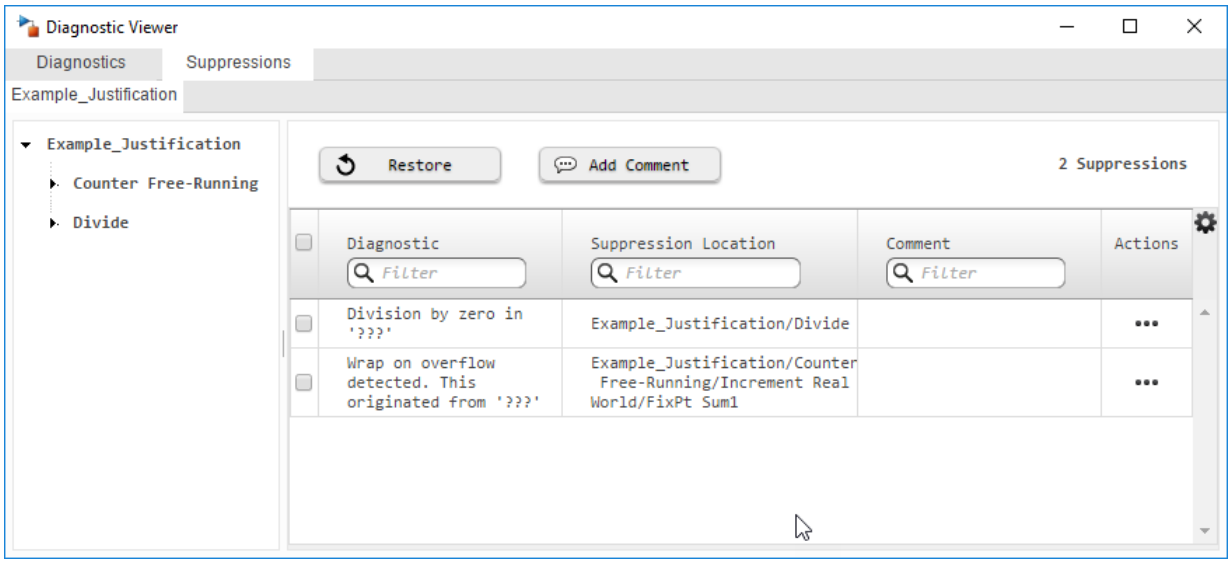
To suppress the diagnostic from the specified source, click the **Suppress** button next to the diagnostic in the Diagnostic Viewer. You can restore the diagnostic from the source by clicking **Restore**. Diagnostic suppressions are saved with the model and persist across sessions.





The suppressed diagnostics are displayed in the **Suppressions** tab. You can restore the suppressed diagnostics, add or edit comments to the suppressed diagnostic by using the **Restore** and **Add Comment** buttons respectively. Alternatively, you can perform these actions on the suppressed diagnostic by selecting one of the options from the **Actions** menu.

The **Suppression** tab of the Diagnostic Viewer displays the model name in the left pane of the suppressed diagnostics in the right pane in a tabular format. You can use the filter options available in the **Diagnostic**, **Suppression Location**, and the **Comment** columns to filter the diagnostics.



You can move the suppressed diagnostics from block level to subsystem level. You can also control the suppression of diagnostics from the command line. For more information, see “Suppress Diagnostic Messages Programmatically” on page 25-56.

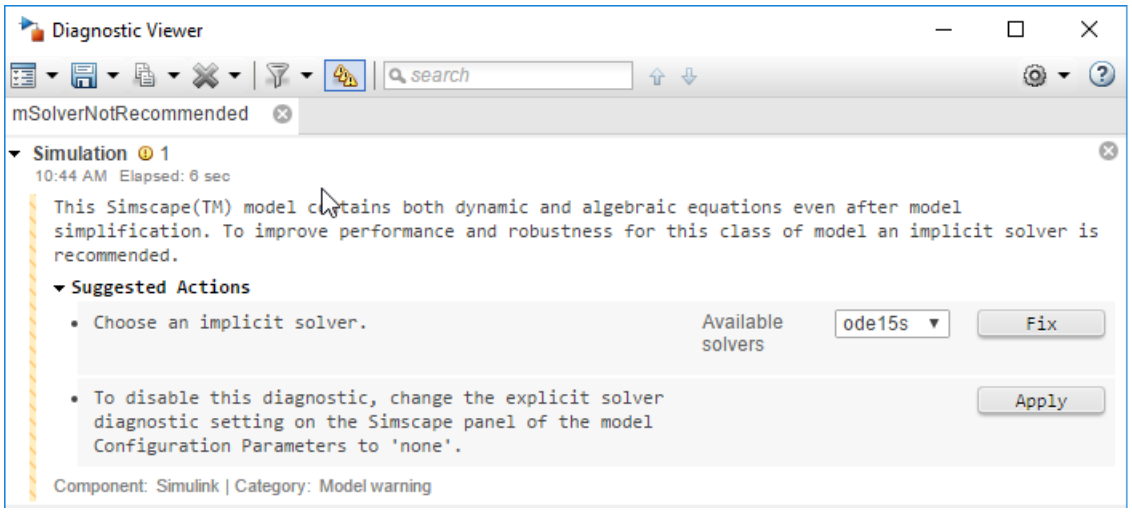
Suggested Actions

Diagnostic viewer provides suggestions and fixes for diagnostic error and warning messages. These suggestions and fixes are provided in the **Suggested Actions** section of diagnostic message pane.

A diagnostic error or warning can have multiple fixes and suggestions. Each fix is associated with a **Fix** button.

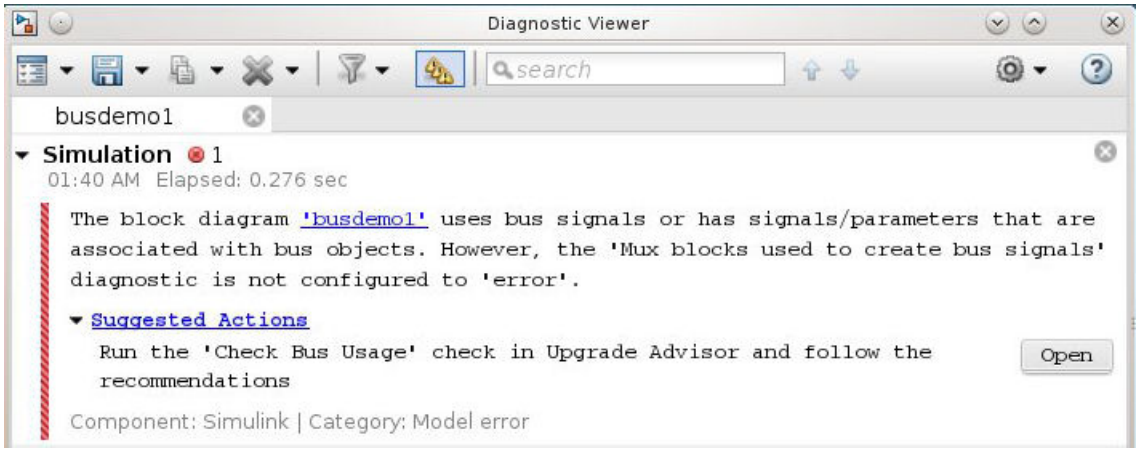
You can click the **Fix** button for the most suitable fix to rectify the error automatically. In some cases, you can provide the fix by one of these ways:

- Enter the values in the available edit boxes.
- Select a value from one of the listed values from a combo box.



The **Fix** buttons for a diagnostic error or warning are no longer available after a fix is successfully applied. If a fix was unsuccessful, a failure message is displayed in the **Suggested Actions** section.

Suggestions are provided for errors and warnings that cannot be fixed automatically.



Note The **Suggested Actions** section is available only for the diagnostic errors or warnings that have a predefined fix.

See Also

Related Examples

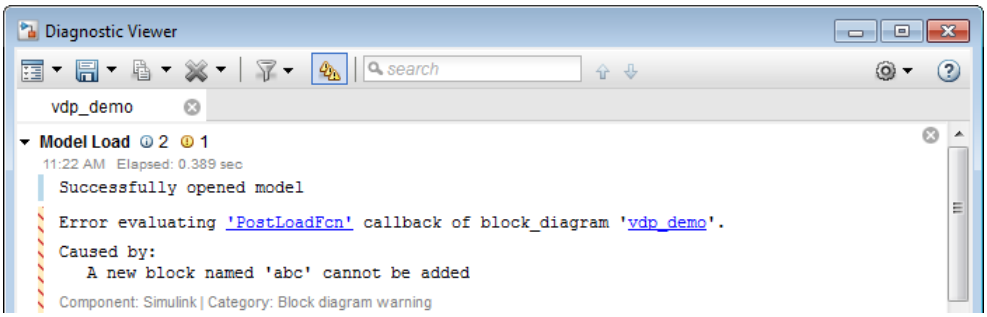
- “Systematic Diagnosis of Errors and Warnings” on page 25-53
- “Customize Diagnostic Messages” on page 25-63
- “Report Diagnostic Messages Programmatically” on page 25-65

Systematic Diagnosis of Errors and Warnings

This example shows how to use the Diagnostic Viewer to identify and locate simulation errors and warnings systematically.

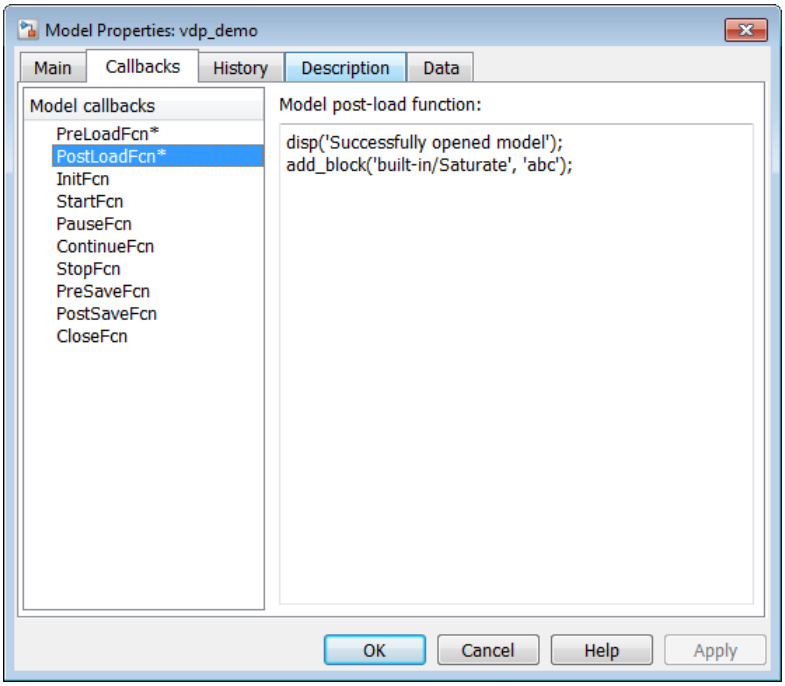
1 Open your model.

If your model contains errors related to callback functions, the **Diagnostic Viewer** opens and displays the following errors in **Model Load** stage.



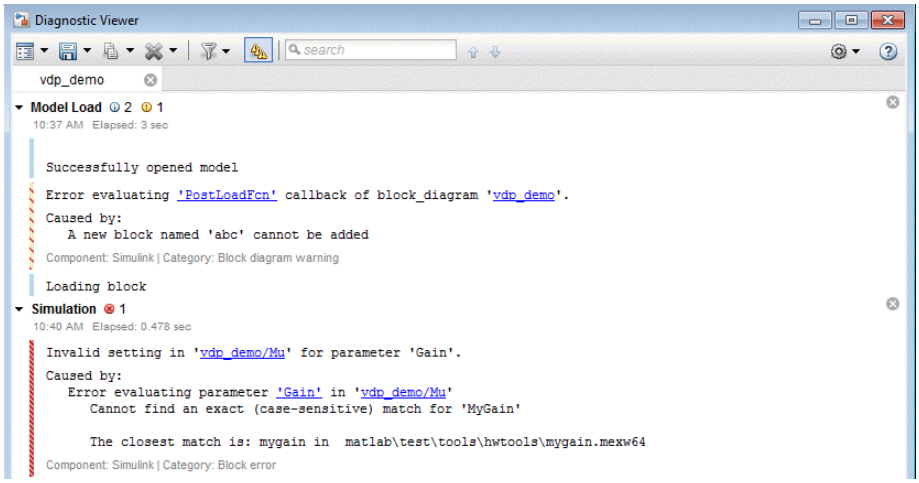
Tip To open the Diagnostic Viewer window, in the **Debug** tab, click **Diagnostics** or click the view errors or warnings link displayed at the bottom of the Simulink Editor window.

2 In the Simulink Editor, in the **Modeling** tab, select **Model Settings > Model Properties**, and examine the callback error.



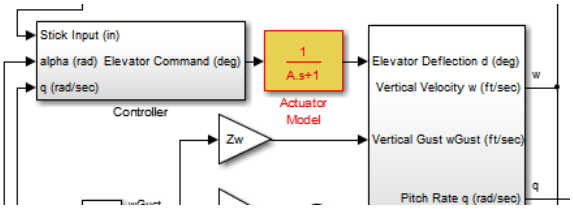
3 After fixing any callback errors, simulate the model to diagnose simulation errors and warnings.

Diagnostic Viewer lists errors and warnings in stages. Each stage in **Diagnostic Viewer** represents a single event such as model load, update diagram, simulation, or build.



4 Filter out warnings by clicking so that you can address errors first.

5 To locate the source of the error, click the hyperlink in the message. The model in the source is highlighted. If a block has multiple ports, you can hover over each port to see the port number.



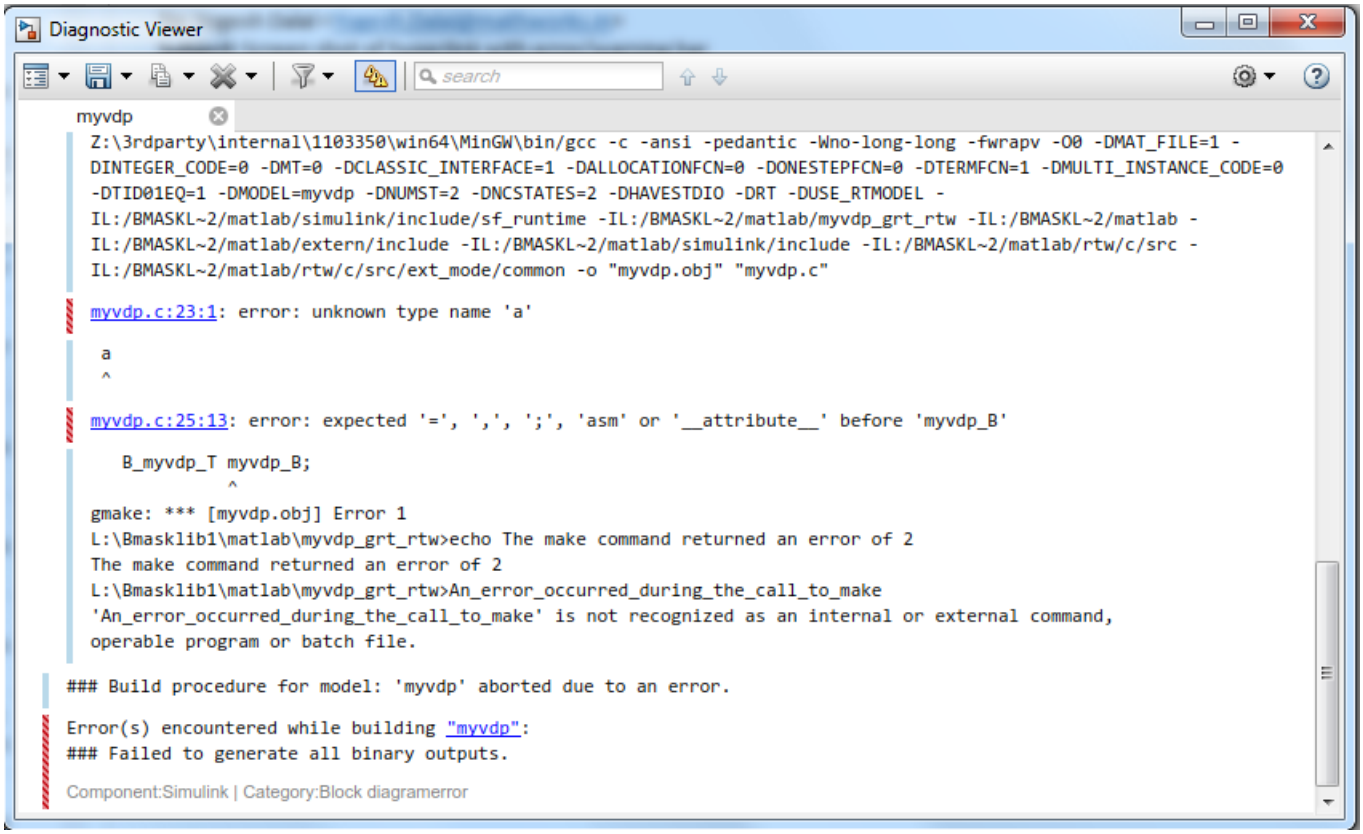
6 After fixing all errors, simulate your model again and view the **Diagnostic Viewer** to identify remaining problems.

Note If an error or warning has a predefined fix, the diagnostic message pane displays a **Suggested Actions** section. You can use the **Fix** button provided in this section to rectify the related error or warning. For more information see, “Suggested Actions” on page 25-51.

7 If an object in your model generates a warning that you do not want to be notified of, sometimes, you can suppress the warning from the specified source using the **Suppress** button. You can restore the warning from that source using the **Restore** button. For example, if a Counter Free-Running block generates an overflow warning that is intentional in your design, you can suppress only overflow warnings from this particular block, without sacrificing notification of other overflows in your model.

8 To generate code for your model, in the **C Code** tab, click **Build**.

Note If there is a failure during code generation, Diagnostic Viewer provides hyperlinks for easy navigation to the source of the error or warning message.



See Also

Related Examples

- “Customize Diagnostic Messages” on page 25-63
- “Report Diagnostic Messages Programmatically” on page 25-65

Suppress Diagnostic Messages Programmatically

The following examples show how to manage diagnostic suppressions programmatically.

In this section...

Suppress Diagnostic Messages Programmatically

This example shows how to access simulation metadata to manage diagnostic suppressions and to restore diagnostic messages programmatically.

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\diagnostic_suppressor_demo`.
2. Change to the `docroot\toolbox\simulink\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'simulink', 'examples'))
   ```
3. Copy the `getDiagnosticObjects.m`, `suppressor_script.m`, and `Suppressor_CLI_Demo.slx` files to your local working folder.

   The `getDiagnosticObjects.m` function queries the simulation metadata to access diagnostics that were thrown during simulation. The `suppressor_script.m` script contains the commands for suppressing and restoring diagnostics to the `Suppressor_CLI_Demo` model.

   `getDiagnosticObjects.m`

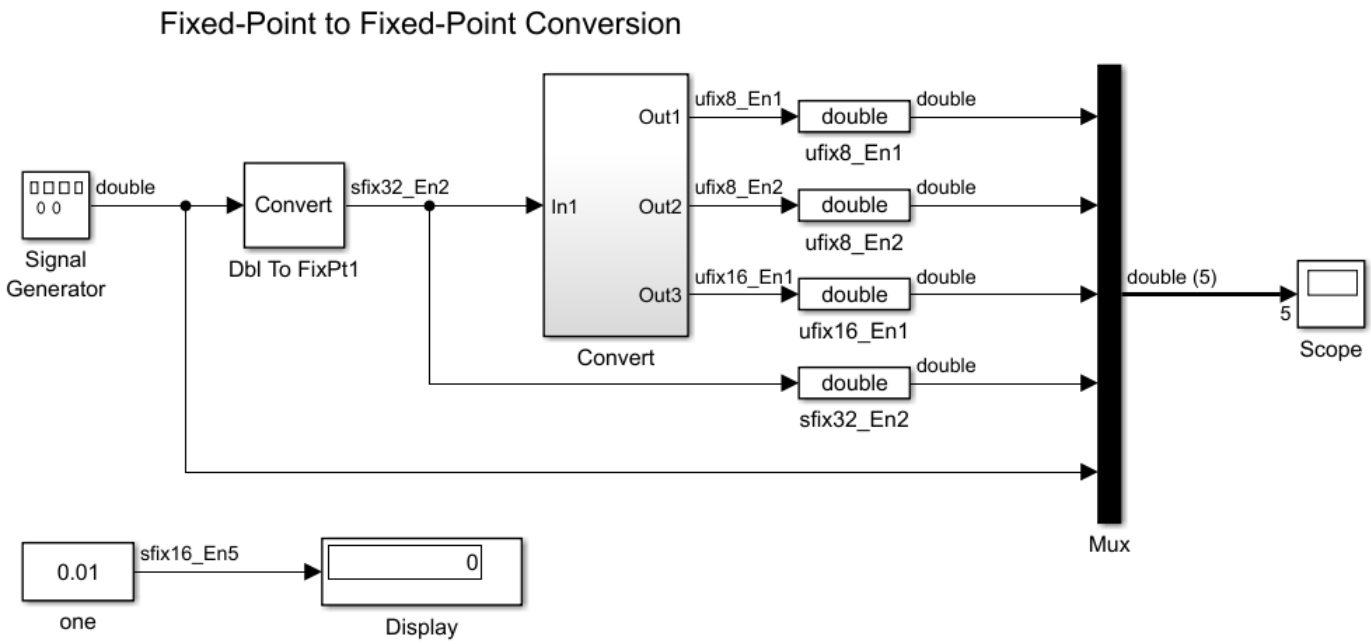
   ```
   function y = getDiagnosticObjects(in)
   Warningdata = in.getSimulationMetadata.ExecutionInfo.WarningDiagnostics;
   Errordata = in.getSimulationMetadata.ExecutionInfo.ErrorDiagnostic;

   index = 1;

   for i = 1 : numel(Warningdata)
       y(index) = Warningdata(i).Diagnostic;
       index = index + 1;
   end


   for i = 1 : numel(Errordata)
       y(index) = Errordata(i).Diagnostic;
       index = index + 1;
   end
   ```

Open and Simulate the Model

Open the model. To access `Simulink.SimulationMetadata`, set the `ReturnWorkspaceOutputs` parameter value to `'on'`. Simulate the model.

```
model = 'Suppressor_CLI_Demo';
open_system(model);
```

```
set_param(model,'ReturnWorkspaceOutputs','on');
out = sim(model);
```



Get Message Identifiers from Simulation Metadata

Find the names of diagnostic message identifiers using the simulation metadata stored in the `MSLDiagnostic` object.

```
if (exist('out', 'var'))
    diag_objects = getDiagnosticObjects(out);
end
```

Several warnings were generated during simulation, including a saturation of the Data Type Conversion block. Query the `diag_objects` variable to get more information on the identifiers.

```
diag_objects(5)
```

```
ans =

  MSLDiagnostic with properties:

    identifier: 'SimulinkFixedPoint:util:Saturationoccurred'
       message: 'Saturation occurred. This originated from 'Suppressor_CLI_Demo/Con…'
         paths: {'Suppressor_CLI_Demo/Convert/FixPt To FixPt3'}
         cause: {}
         stack: [0×1 struct]
```

Suppress Saturation Diagnostic on a Block

Use the `Simulink.suppressDiagnostic` function to suppress the saturation diagnostic on the data type conversion block only. Simulate the model.

```
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/Convert/FixPt To FixPt3', ...
'SimulinkFixedPoint:util:Saturationoccurred');
set_param(model,'SimulationCommand','start');
```

Restore the Saturation Diagnostic

Use the `Simulink.restoreDiagnostic` function to restore the saturation diagnostic of the same block.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/Convert/FixPt To FixPt3',...
 'SimulinkFixedPoint:util:Saturationoccurred');
set_param(model,'SimulationCommand','start');
```

Suppress Multiple Diagnostics on a Source

You can suppress multiple warnings on a single source by creating a cell array of message identifiers. Suppress the precision loss and parameter underflow warnings of the Constant block, `one`, in the model.

```
diags = {'SimulinkFixedPoint:util:fxpParameterPrecisionLoss',...
 'SimulinkFixedPoint:util:fxpParameterUnderflow'};
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one',diags);
set_param(model,'SimulationCommand','start');
```

Restore All Diagnostics on a Block

Restore all diagnostics on a specified block using the `Simulink.restoreDiagnostic` function.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/one');
set_param(model,'SimulationCommand','start');
```

Suppress a Diagnostic on Many Blocks

You can suppress one or more diagnostics on many blocks. For example, use the `find_system` function to create a cell array of all Data Type Conversion blocks in a system, and suppress all saturation warnings on the specified blocks.

```
dtc_blocks = find_system('Suppressor_CLI_Demo/Convert',...
 'BlockType', 'DataTypeConversion');
Simulink.suppressDiagnostic(dtc_blocks, 'SimulinkFixedPoint:util:Saturationoccurred');
set_param(model,'SimulationCommand','start');
```

Restore All Diagnostics Inside a Subsystem

You can also use the `Simulink.restoreDiagnostic` function to restore all diagnostics inside a specified subsystem.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/Convert',...
 'FindAll', 'On');
set_param(model,'SimulationCommand','start');
```

Add Comments and User Information to a Suppression

A `SuppressedDiagnostic` object contains information on the source of the suppression and the suppressed diagnostic message identifier. You can also include comments, and the name of the user who last modified the suppression.

```
Object = Simulink.SuppressedDiagnostic('Suppressor_CLI_Demo/Convert/FixPt To FixPt1',...
 'SimulinkFixedPoint:util:Saturationoccurred');
```

```
Object.Comments = 'Reviewed: John Doe';
Object.LastModifiedBy = 'Joe Schmoe'
set_param(model,'SimulationCommand','start');
```

```
Object =

  SuppressedDiagnostic with properties:

           Source: 'Suppressor_CLI_Demo/Convert/FixPt To FixPt1'
               Id: 'SimulinkFixedPoint:util:Saturationoccurred'
    LastModifiedBy: 'Joe Schmoe'
         Comments: 'Reviewed: John Doe'
     LastModified: '2016-Jun-21 18:23:01'
```

Get Suppression Data

To get suppression data for a certain subsystem or block, use the `Simulink.getSuppressedDiagnostics` function.

```
Object = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo/Convert/FixPt To FixPt1');
set_param(model,'SimulationCommand','start');
```

Restore All Diagnostics on a Model

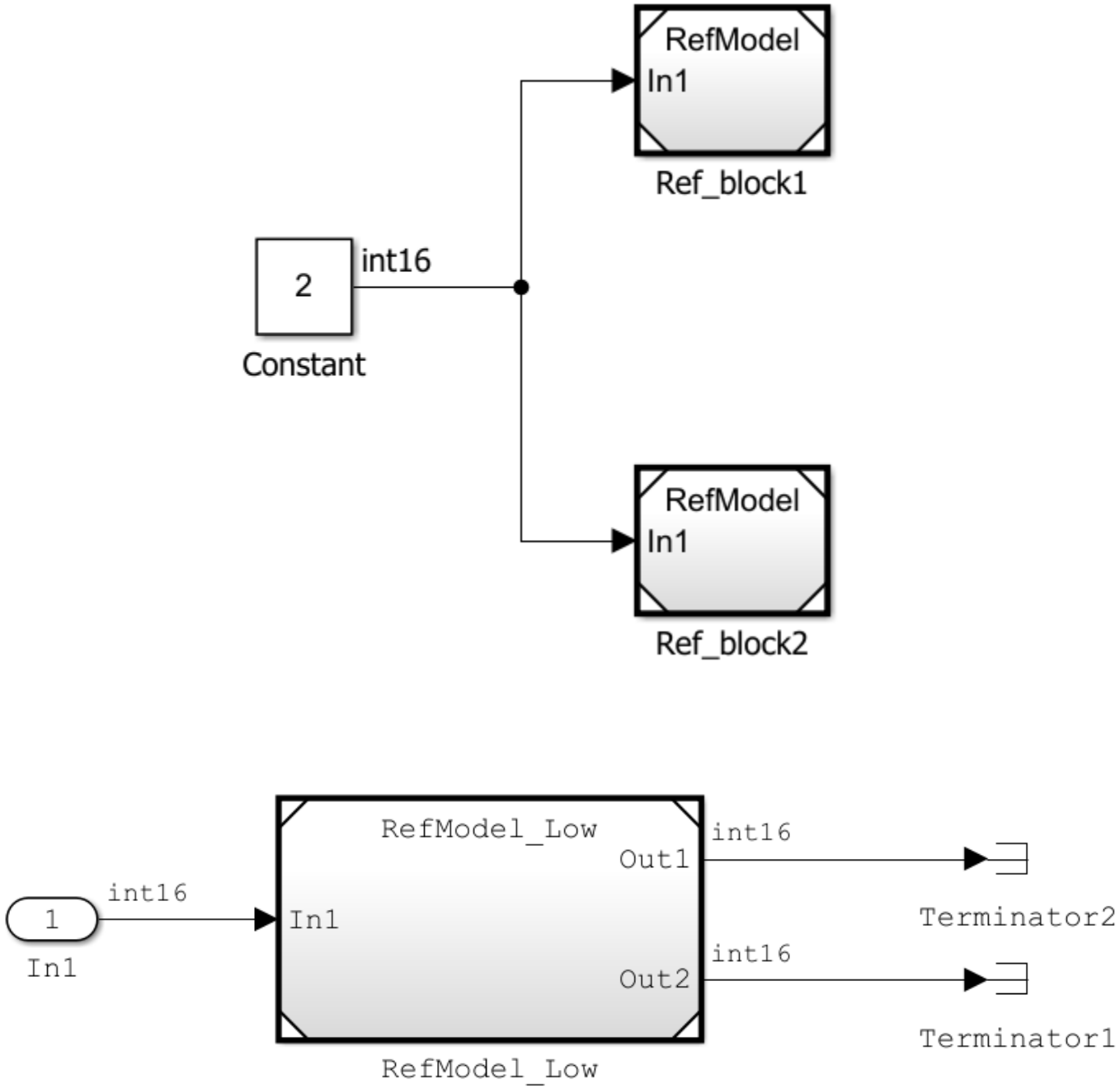
When a model contains many diagnostic suppressions, and you want to restore all diagnostics to a model, use the `Simulink.getSuppressedDiagnostics` function to return an array of `Simulink.SuppressedDiagnostic` objects. Then use the `restore` method as you iterate through the array.

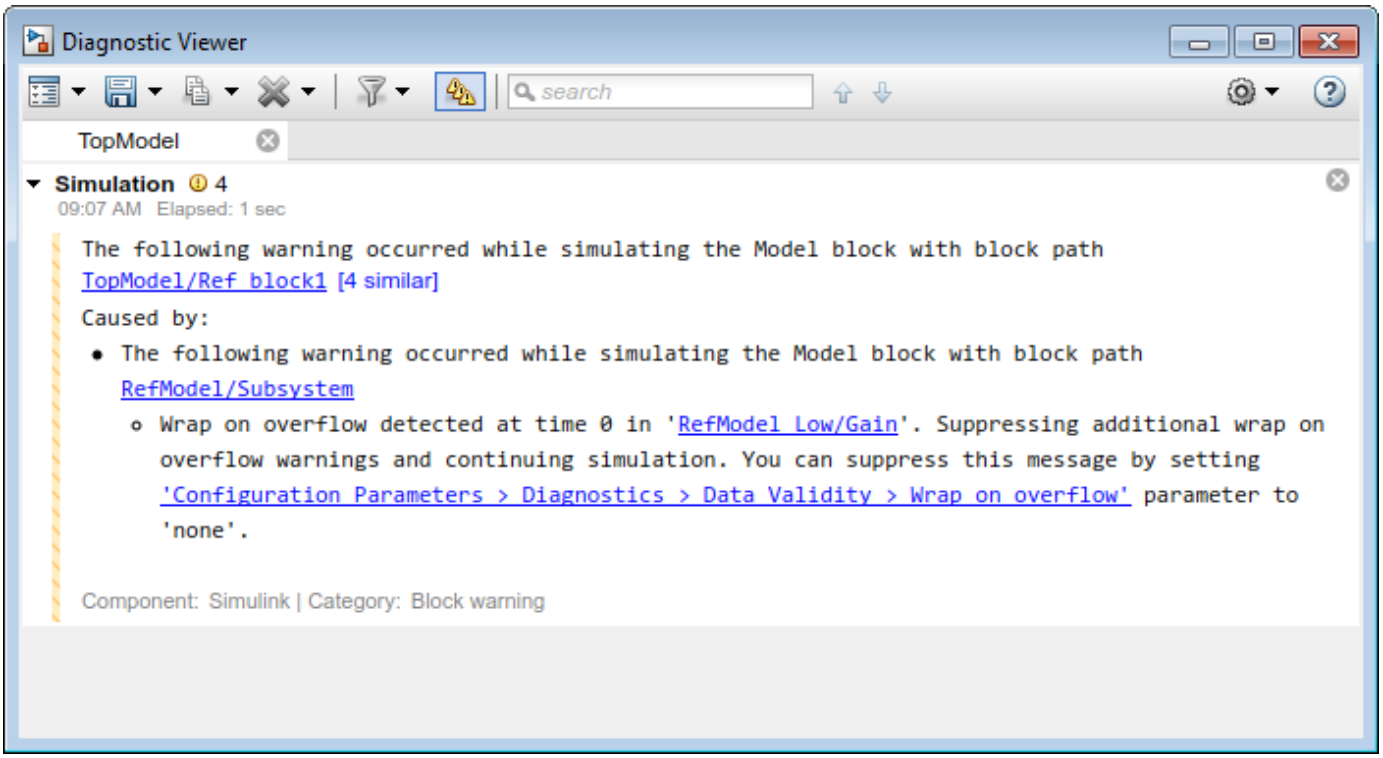
```
Objects = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo');
for iter = 1:numel(Objects)
    restore(Objects(iter));
end
set_param(model,'SimulationCommand','start');
```

Suppress Diagnostic Messages of a Referenced Model

This example shows how to suppress a diagnostic when the diagnostic originates from a referenced model. By accessing the `MSLDiagnostic` object of the specific instance of the warning, you can suppress the warning only for instances when the referenced model is simulated from the specified top model.

This example model contains two instances of the same referenced model, `RefModel`. The model `RefModel` references yet another model, `RefModel_Low`. `RefModel_Low` contains two Gain blocks that each produce a wrap on overflow warning during simulation. Suppress one of the four instances of this warning in the model by accessing the `MSLDiagnostic` object associated with the wrap on overflow warning produced by one of the Gain blocks in the `RefModel_Low` model only when it is referenced by `Ref_block1`.


RefModel
In1
Ref_block1
2
int16
Constant
RefModel
In1
Ref_block2
1
In1
int16
RefModel_Low
In1
Out1
Out2
int16
int16
Terminator2
Terminator1
RefModel_Low



Open the top model. Simulate the model and store the output in a variable, `out`.

```
out = sim('TopModel');
```

Access the simulation metadata stored in the `MSLDiagnostic` object.

```
diag = getDiagnosticObjects(out)
```

```
diag =

  1×4 MSLDiagnostic array with properties:

    identifier
    message
    paths
    cause
    stack
```

You can view the diagnostics and their causes in the Diagnostic Viewer or at the command-line.

```
for i = 1 : numel(diag)
    disp(diag(i));
    disp(diag(i).cause{1});
end
```

Suppress one of the wrap on overflow warnings from `RefModel_Low` only when it is simulated from `TopModel/Ref_block1` by accessing the specific diagnostic. Simulate the model.

```
Simulink.suppressDiagnostic(diag(1));
out = sim('TopModel')
```

Access the simulation metadata. This simulation produced only three warnings.

```
diag = getDiagnosticObjects(out)
```

```
diag =

  1×3 MSLDiagnostic array with properties:

    identifier
    message
    paths
    cause
    stack
```

Restore the diagnostic to the model.

```
Simulink.restoreDiagnostic(diag(1));
```

See Also

`Simulink.SimulationMetadata` | `Simulink.SuppressedDiagnostic` | `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic` | `Simulink.suppressDiagnostic` | `restore`

Customize Diagnostic Messages

In this section...

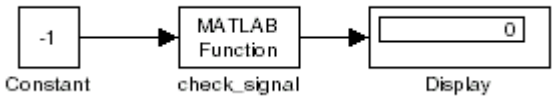
The **Diagnostic Viewer** displays the output of MATLAB error functions executed during simulation.

You can customize simulation error messages in the following ways by using MATLAB error functions in callbacks, S-functions, or MATLAB Function blocks.

Display Custom Text

This example shows how to can customize the MATLAB function `check_signal` to display the text `Signal is negative`.

1 Open the MATLAB Function for editing.



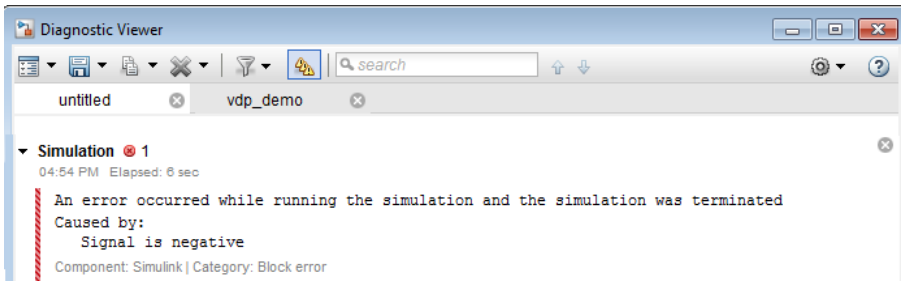
2 In the MATLAB Editor, create a function to display custom text.

```
function y = check_signal(x)
    if x < 0
        error('Signal is negative');
    else
        y = x;
    end
```

3 Simulate the model.

A runtime error appears and you are prompted to start the debugger. Click **OK**.

4 To view the following error in **Diagnostic Viewer**, close the debugger.



Create Hyperlinks to Files, Folders, or Blocks

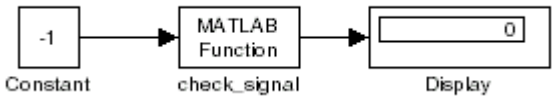
To create hyperlinks to files, folders, or blocks in an error message, include the path to these items within customized text.

Example error message	Hyperlink
`error ('Error reading data from "C:/Work/designData.mat"')`	Opens `designData.data` in the MATLAB Editor.
`error ('Could not find data in folder "C:/Work"')`	Opens a Command Window and sets `C:\Work` as the working folder.
`error ('Error evaluating parameter in block "myModel/Mu"')`	Displays the block `Mu` in model `myModel`.

Create Programmatic Hyperlinks

This example shows how to can customize the MATLAB function `check_signal` to display a hyperlink to the documentation for `find_system`.

1 Open the MATLAB Function for editing.



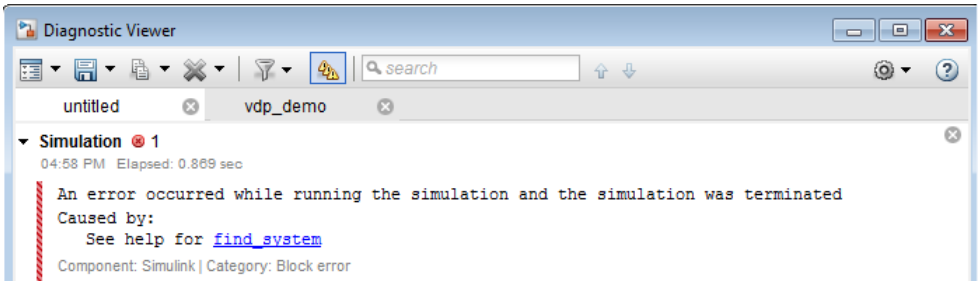
2 In the MATLAB Editor, create a function to display custom text.

```
function y = check_signal(x)
    if x < 0
        error('See help for <a href="matlab:doc find_system">find_system</a>');
    else
        y = x;
    end
```

3 Simulate the model.

A runtime error appears and you are prompted to start the debugger. Click **OK**.

4 To view the following error in **Diagnostic Viewer**, close the debugger.



See Also

Related Examples

- “Systematic Diagnosis of Errors and Warnings” on page 25-53
- “Report Diagnostic Messages Programmatically” on page 25-65

Report Diagnostic Messages Programmatically

The `sldiagviewer` functions enable you to generate, display, and log diagnostic messages in the Diagnostic Viewer.

You can use these functions to report the diagnostic messages programmatically:

- Function to create a stage: `sldiagviewer.createStage`
- Functions to report diagnostic messages:
  - `sldiagviewer.reportError`
  - `sldiagviewer.reportWarning`
  - `sldiagviewer.reportInfo`
- Function to log the diagnostics: `sldiagviewer.diary`

Create Diagnostic Stages

In the Diagnostic Viewer, errors, warnings, and information messages are displayed in groups based on the operation, such as model load, simulation, and build. These groups are called stages. The `sldiagviewer.createStage` function enables you to create stages. You can also create child stages for a stage object. A parent stage object must be active to create a child stage. When you create a stage object, Simulink initializes a stage. When you close the stage object, Simulink ends the stage. If you delete a parent stage object, the corresponding parent and its child stage close in the Diagnostic Viewer. The syntax for creating a stage is:

```
stageObject = sldiagviewer.createStage(StageName,'ModelName',ModelNameValue)
```

In this syntax,

- `StageName` specifies the name of a stage and is a required argument, for example, `'Analysis'`.
- Use the `'ModelName'`, `ModelNameValue` pair to specify the model name of a stage, for example `'ModelName'`, `'vdp'`. All the child stages inherit the model name from their parent.

Example to Create Stage

```
my_stage = sldiagviewer.createStage('Analysis','ModelName','vdp');
```

Report Diagnostic Messages

You can use the `sldiagviewer` functions to report error, warning, or information messages in the Diagnostic Viewer. The syntaxes for reporting diagnostic messages are:

- `sldiagviewer.reportError(Message)`: Reports the error messages.
- `sldiagviewer.reportWarning(Message)`: Reports the warnings.
- `sldiagviewer.reportInfo(Message)`: Reports the information messages.

`Message` describes the error, warning, or build information and is a required argument. `Message` can have values in these formats:

- String

- `MSLException` or `MException` object

Optionally, you can use the `'Component'` argument and its corresponding value in the syntax to specify the component or product that generates the message, for example, `'Simulink'` and `'Stateflow'`.

Example to Report Diagnostics

```
% Create a Stage to display all the messages

my_stage = sldiagviewer.createStage('Analysis', 'ModelName', 'vdp');

% Catch the error introduced in vdp as an exception.

try
sim('vdp');
catch error
% Report the caught exception as warning

sldiagviewer.reportWarning(error);
end

% Report a custom info message to Diagnostic Viewer

sldiagviewer.reportInfo('My Info message');
```

Log Diagnostic Messages

You can use the `sldiagviewer.diary` function to log the simulation warning, error, and build information to a file. The syntaxes for generating log files are:

- `sldiagviewer.diary`: Intercepts the build information, warnings, and errors transmitted to the Diagnostic Viewer and logs them to a text file `diary.txt` in the current directory.
- `sldiagviewer.diary(filename)`: Toggles the logging state of the text file specified by `filename`.
- `sldiagviewer.diary(toggle)`: Toggles the logging ability. Valid values are `'on'` and `'off'`. If you have not specified a log file name, the toggle setting applies to the last file name that you specified for logging or to the `diary.txt` file.
- `sldiagviewer.diary(filename,'UTF-8')`: Specifies the character encoding for the log file.

In this syntax,

- `filename` specifies the file to log the data to.
- `toggle` specifies the logging state `'on'` or `'off'`.

Log Diagnostic Messages

```
% Start logging build information and simulation warnings and errors to diary.txt

sldiagviewer.diary
open_system('vdp')
set_param('vdp/Mu','Gain', 'xyz')
set_param('vdp', 'SimulationCommand', 'Start')
% This introduces an error and do UI simulation which you can see in the diary log
```

```
% Open diary.txt to view logs.

%### Starting build procedure for model: vdp
%### Build procedure for model: 'vdp' aborted due to an error.
%...

% Set up logging to a specific file

sldiagviewer.diary('C:\MyLogs\log1.txt') % Make sure you have write permission for this location

% Switch the logging state of a file

sldiagviewer.diary('C:\MyLogs\log2.txt') % Switch on logging and specify a log file.
open_system('vdp')
set_param('vdp/Mu', 'Gain', 'xyz')
set_param('vdp', 'SimulationCommand', 'Start')

sldiagviewer.diary('off') % Switch off logging.
open_system('sldemo_fuelsys') % Any operation you do after the previous command will not be logge
rtwbuild('sldemo_fuelsys')

sldiagviewer.diary('on') % Resume logging in the previously specified log file.

% Specify the filename to log to and character encoding to be used
sldiagviewer.diary('C:\MyLogs\log3.txt','UTF-8')
```

See Also

More About

- “View Diagnostics” on page 25-48
- “Systematic Diagnosis of Errors and Warnings” on page 25-53

Running a Simulation Programmatically

Run Simulations Programmatically

You can programmatically simulate a model with the `sim` function, using various techniques to specify parameter values. In addition to simulating a model, you can use the `sim` to enable simulation timeouts, capture simulation errors, and access simulation metadata when your simulation is complete

For an interactive simulation, you can use `set_param` and `get_param`. With `set_param` and `get_param`, you can check the status of a running simulation, control how the simulation works by using block callbacks.

Specify Parameter Name-Value Pairs

This example shows how to programmatically simulate a model, specifying parameters as name-value pairs.

Simulate the `vdp` model with parameter values specified as consecutive name-value pairs.

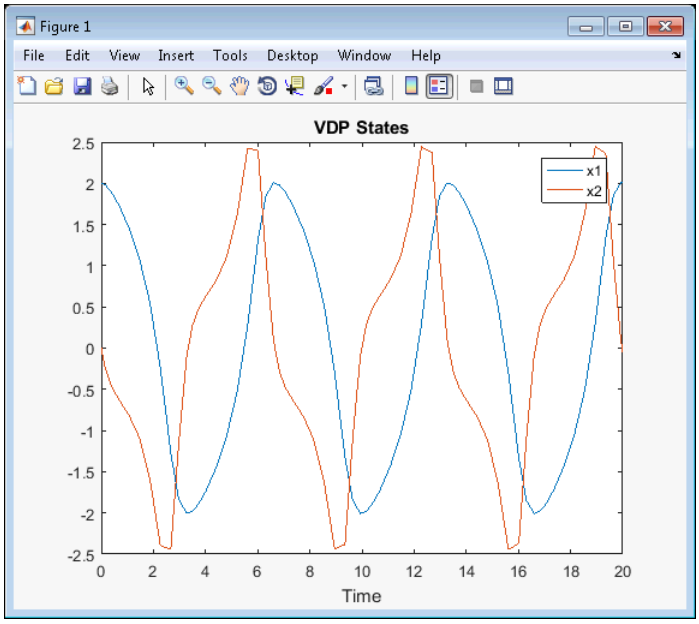
```
simOut = sim('vdp','SimulationMode','normal','AbsTol','1e-5',...
             'SaveState','on','StateSaveName','xout',...
             'SaveOutput','on','OutputSaveName','yout',...
 'SaveFormat', 'Dataset');
outputs = simOut.get('yout')
```

```
outputs =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'yout'
    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.
```

You simulate the model in `Normal` mode, specifying an absolute tolerance for solver error. The `sim` function returns `SimOut`, a single `Simulink.SimulationOutput` object that contains all of the simulation outputs (logged time, states, and signals). The `sim` function *does not* return simulation values to the workspace.

Plot the output signal values against time.

```
x1=(outputs.get('x1').Values);
x2=(outputs.get('x2').Values);
plot(x1); hold on;
plot(x2);
title('VDP States')
xlabel('Time'); legend('x1','x2')
```



Enable Simulation Timeouts

If you are running multiple simulations in a loop and are using a variable-step solver, consider using `sim` with the `timeout` parameter. If, for some reason, a simulation hangs or begins to take unexpectedly small time steps, it will time out. Then, the next simulation can run. Example syntax is shown below.

```
N = 100;
simOut = repmat(Simulink.SimulationOutput, N, 1);
for i = 1:N
        simOut(i) = sim('vdp', 'timeout', 1000);
end
```

Capture Simulation Errors

If an error causes your simulation to stop, you can see the error in the simulation metadata. In this case, `sim` captures simulation data in the simulation output object up to the time it encounters the error, enabling you to do some debugging of the simulation without rerunning it. To enable this feature, use the `CaptureErrors` parameter with the `sim` function.

Example syntax and resulting output for capturing errors with `sim` is:

```
simOut = sim('my_model', 'CaptureErrors', 'on');
simOut.getSimulationMetadata.ExecutionInfo

ans =

  struct with fields:

              StopEvent: 'DiagnosticError'
        StopEventSource: []
   StopEventDescription: 'Division by zero in 'my_model/Divide''
        ErrorDiagnostic: [1×1 struct]
     WarningDiagnostics: [0×1 struct]
```

Another advantage of this approach is that the simulation error does not also cause `sim` to stop. Therefore, if you are using `sim` in a `for` loop for example, subsequent iterations of the loop will still run.

Access Simulation Metadata

This example shows you how to access simulation metadata once your simulation is complete. You can run any kind of simulation and access it's metadata.

This example simulates the model with parameter values specifies as name-value pairs. Run the simulation.

```
simOut = sim('vdp','SimulationMode','normal','AbsTol','1e-5',...
             'SaveState','on','StateSaveName','xoutNew',...
             'SaveOutput','on','OutputSaveName','youtNew',...
 'SaveFormat', 'StructureWithTime');
```

Access the `ModelInfo` property, which has some basic information about the model and solver.

```
simOut.getSimulationMetadata.ModelInfo
```

```
ans =

  struct with fields:

                 ModelName: 'vdp'
              ModelVersion: '1.6'
             ModelFilePath: 'C:\MyWork'
                    UserID: 'User'
               MachineName: 'MyMachine'
                  Platform: 'PCWIN64'
    ModelStructuralChecksum: [4×1 uint32]
            SimulationMode: 'normal'
                 StartTime: 0
                  StopTime: 20
                SolverInfo: [1×1 struct]
             SimulinkVersion: [1×1 struct]
                LoggingInfo: [1×1 struct]
```

Inspect the solver information.

```
simOut.getSimulationMetadata.ModelInfo.SolverInfo
```

```
ans =

  struct with fields:

           Type: 'Variable-Step'
         Solver: 'ode45'
    MaxStepSize: 0.4000
```

Review timing information for your simulation, such as when your simulation started and finished, and the time the simulation took to initialize, execute, and terminate.

```
simOut.getSimulationMetadata.TimingInfo
```

```
ans =
```

```
  struct with fields:

         WallClockTimestampStart: '2016-06-17 10:26:58.433686'
          WallClockTimestampStop: '2016-06-17 10:26:58.620687'
    InitializationElapsedWallTime: 0.1830
         ExecutionElapsedWallTime: 1.0000e-03
       TerminationElapsedWallTime: 0.0030
            TotalElapsedWallTime: 0.1870
```

Add notes to your simulation.

```
simOut=simOut.setUserString('Results from simulation 1 of 10');
simOut.getSimulationMetadata

ans =

  SimulationMetadata with properties:

        ModelInfo: [1×1 struct]
       TimingInfo: [1×1 struct]
    ExecutionInfo: [1×1 struct]
       UserString: 'Results from simulation 1 of 10'
         UserData: []
```

You can also add your own custom data using the `UserData` property.

Control and Check Status of Simulation

This example shows how to use `set_param` to control and check the status of your simulation. `set_param` allows you to update the variables dynamically as well as write data-logging variables to the workspace.

Start a simulation.

```
set_param('vdp','SimulationCommand','start')
```

When you start a simulation using `set_param` and the `'start'` argument, you must use the `'stop'` argument to stop it.

Pause, continue, and stop a simulation.

```
set_param('vdp','SimulationCommand','pause')
set_param('vdp','SimulationCommand','continue')
set_param('vdp','SimulationCommand','stop')
```

When you use `set_param` to pause or stop a simulation, the commands are requests for such actions and the simulation doesn't execute them immediately. You can use `set_param` to start a simulation after the stop command and to continue a simulation after the pause command. Simulink first completes uninterruptable work, such as solver steps and other commands that preceded the `set_param` command. Then, simulation starts, pauses, continues or stops as specified by the `set_param` command.

Check the status of a simulation.

```
get_param('vdp','SimulationStatus')
```

The software returns `'stopped'`, `'initializing'`, `'running'`, `'paused'`, `'compiled'`, `'updating'`, `'terminating'`, or `'external'` (used with the Simulink Coder product).

To update the changed workspace variables dynamically while a simulation is running, use the `update` command.

```
set_param('vdp','SimulationCommand','update')
```

Write all data-logging variables to the base workspace.

```
set_param('vdp','SimulationCommand','WriteDataLogs')
```

Automate Simulation Tasks Using Callbacks

A callback executes when you perform various actions on your model, such as starting, pausing, or stopping a simulation. You can use callbacks to execute a MATLAB script or other MATLAB commands. For more information, see “Callbacks for Customized Model Behavior” on page 4-44 and “Block Callback Parameters” on page 4-49.

This example shows how to use the model `StartFcn` callback to automatically execute MATLAB code before the simulation starts.

Write a MATLAB script that finds Scope blocks in your model and opens them in the foreground when you simulate the model. Save the script in the current folder.

```
% openscopes.m
% Brings scopes to forefront at beginning of simulation.

blocks = find_system(bdroot,'BlockType','Scope');

% Finds all of the scope blocks in the top level of your
    % model. To find scopes in subsystems, provide the subsystem
    % names. Type help find_system for more on this command.

for i = 1:length(blocks)
  set_param(blocks{i},'Open','on')
end

% Loops through all of the scope blocks and brings them
    % to the forefront.
```

Set the `StartFcn` parameter for the model to call the `openscopes` script.

```
set_param('my_model','StartFcn','openscopes')
```

See Also

`Simulink.SimulationMetadata` | `Simulink.SimulationOutput` | `getSimulationMetadata` | `setUserData` | `setUserString`

Related Examples

- “Run Multiple Simulations” on page 27-2

Run Parallel Simulations

The `parsim` command allows you to run parallel (simultaneous) Simulink® simulations of your model (design). In this context, parallel runs mean multiple simulations at the same time on different workers. `parsim` makes it easy for you to run the same model with different inputs or different parameter settings in scenarios such as Monte Carlo analyses, parameter sweeps, model testing, experiment design, and model optimization. Running a single simulation in parallel by decomposing the model into smaller components and running those individual pieces simultaneously on multiple workers is currently not supported.

To run the simulations in parallel with `parsim`, you need a Parallel Computing Toolbox for local workers. In addition, you can use MATLAB Parallel Server for multiple computer clusters, clouds, and grids. In the absence of Parallel Computing Toolbox and MATLAB Parallel Server, `parsim` runs the simulations in serial. For more information, see “Parallel Computing Toolbox” and “MATLAB Parallel Server”.

If no parallel pool exists, `parsim` creates a pool from the default cluster profile . To use a cluster other than the default, create a pool with that cluster profile before calling `parsim`.

This example runs multiple simulations in parallel for a set of sweep parameters.

```
% 1) Load model
model = 'sldemo_suspn_3dof';
load_system(model);

% 2) Set up the sweep parameters
Cf_sweep  = 2500*(0.05:0.1:0.95);
numSims   = numel(Cf_sweep);

% 3) Create an array of SimulationInput objects and specify the sweep value for each simulation
simIn(1:numSims) = Simulink.SimulationInput(model);
for idx = 1:numSims
    simIn(idx) = simIn(idx).setBlockParameter([model '/Road-Suspension Interaction'], 'Cf', num2str(Cf_sweep(idx)))
end

% 4) Simulate the model
simOut = parsim(simIn)
```

How parsim works

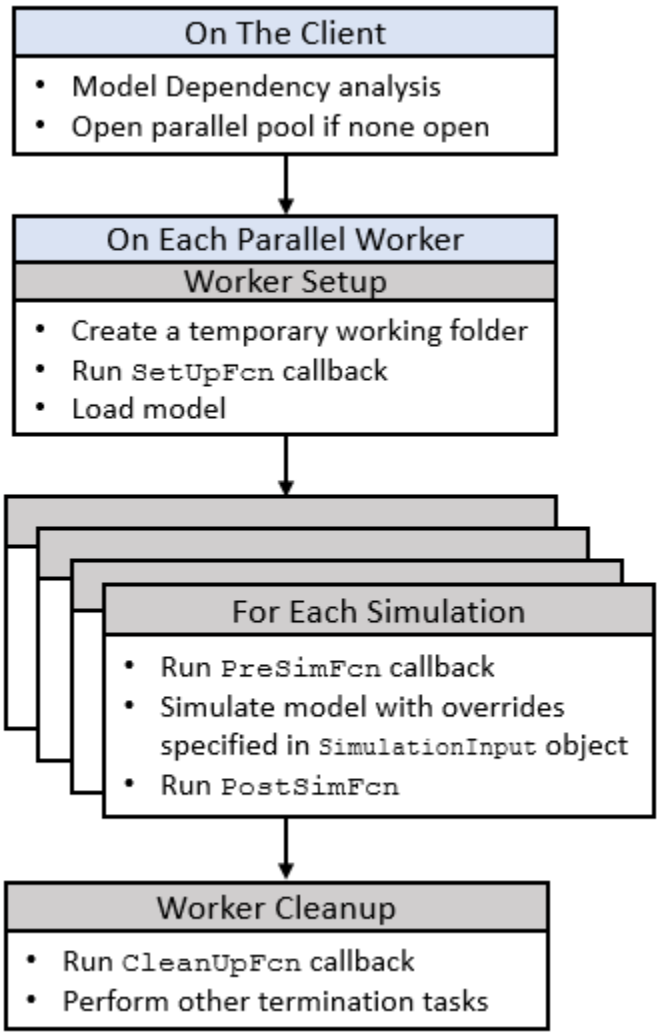
`parsim` runs simulations with different parameters and values based on the `Simulink.SimulationInput` object. Each `SimulationInput` object specifies one simulation of the model. An array of these objects can be created for multiple simulations. For more information, see “Run Multiple Simulations” on page 27-2.

You can use the following methods and properties on `Simulink.SimulationInput` object:

- `setVariables` - Change variables in base workspace, data dictionary, or model workspace
- `setBlockParameters` - Change block parameters
- `setModelParameters` - Change model parameters
- `setPreSimFcn` - Specify MATLAB functions to run before each simulation for customization and post-processing results on the cluster
- `setPostSimFcn` - Specify MATLAB functions to run after each simulation for customization and post-processing results on the cluster
- `InitialState` - Change the Initial State

- `ExternalInput` - Specify a numerical array, timeseries, or Dataset object as external inputs to the model

This flowchart shows a general sequence of events that occur when `parsim` is executed



Changes to model library blocks can be overwritten when using `parsim`. When models are set up on new workers, model inherits properties directly from the worker library. Use `SetUpFcn` with `parsim` to transfer the model library block changes to the workers.

See Also

`ExternalInput` | `Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Related Examples

- “Run Parallel Simulations Using parsim” on page 27-5

- “Run Multiple Simulations” on page 27-2
- “Multiple Simulation Workflows” on page 27-9
- “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38

Using sim function within parfor

Note Using `sim` function within `parfor` loop is no longer recommended. For running parallel simulations, use `parsim`. Please see “Run Parallel Simulations” on page 26-7.

Overview of Calling sim from Within parfor

The `parfor` command allows you to run parallel (simultaneous) Simulink simulations of your model (design). In this context, parallel runs mean multiple model simulations at the same time on different workers. Calling `sim` from within a `parfor` loop often helps for performing multiple simulation runs of the same model for different inputs or for different parameter settings. For example, you can save simulation time performing parameter sweeps and Monte Carlo analyses by running them in parallel. Note that running parallel simulations using `parfor` does not currently support decomposing your model into smaller connected pieces and running the individual pieces simultaneously on multiple workers.

Normal, Accelerator, and Rapid Accelerator simulation modes are supported by `sim` in `parfor`. (See “Choosing a Simulation Mode” on page 35-10 for details on selecting a simulation mode and “Design Your Model for Effective Acceleration” on page 35-14 for optimizing simulation run times.) For other simulation modes, you need to address any workspace access issues and data concurrency issues to produce useful results. Specifically, the simulations need to create separately named output files and workspace variables. Otherwise, each simulation overwrites the same workspace variables and files, or can have collisions trying to write variables and files simultaneously.

For information on code regeneration and parameter handling in Rapid Accelerator mode, see “Parameter Tuning in Rapid Accelerator Mode” on page 35-7.

Also, see `parfor`.

Note If you open models inside a `parfor` statement, close them again using `bdclose all` to avoid leaving temporary files behind.

sim in parfor with Normal Mode

This code fragment shows how you can use `sim` and `parfor` in Normal mode. Save changes to your model before simulating in `parfor`. The saved copy of your model is distributed to parallel workers when simulating in `parfor`.

```
% 1) Load model and initialize the pool.
model = 'sldemo_suspn_3dof';
load_system(model);
parpool;

% 2) Set up the iterations that we want to compute.
Cf                  = evalin('base', 'Cf');
Cf_sweep            = Cf*(0.05:0.1:0.95);
iterations          = length(Cf_sweep);
simout(iterations)  = Simulink.SimulationOutput;

% 3) Need to switch all workers to a separate tempdir in case
% any code is generated for instance for StateFlow, or any other
```

```
% file artifacts are  created by the model.
spmd
    % Setup tempdir and cd into it
    currDir = pwd;
    addpath(currDir);
    tmpDir = tempname;
    mkdir(tmpDir);
    cd(tmpDir);
    % Load the model on the worker
    load_system(model);
end

% 4) Loop over the number of iterations and perform the
% computation for different parameter values.
parfor idx=1:iterations
    set_param([model '/Road-Suspension Interaction'],'MaskValues',...
        {'Kf',num2str(Cf_sweep(idx)),'Kr','Cr'});
    simout(idx) = sim(model, 'SimulationMode', 'normal');
end

% 5) Switch all of the workers back to their original folder.
spmd
    cd(currDir);
    rmdir(tmpDir,'s');
    rmpath(currDir);
    close_system(model, 0);
end

close_system(model, 0);
delete(gcp('nocreate'));
```

sim in parfor with Normal Mode and MATLAB Parallel Server Software

This code fragment is identical to the one in “sim in parfor with Normal Mode” on page 26-10

. Modify it as follows for using `sim` and `parfor` in Normal mode:

- In item 1, modify the `parpool` command as follows to create an object and use it to call a cluster name.

  ```
  p = parpool('clusterProfile');
  % 'clusterProfile' is the name of the distributed cluster
  ```

- In item 1, find files on which the model depends and attach those files to the model for distribution to cluster workers on remote machines.

  ```
  files = dependencies.fileDependencyAnalysis(modelName);
  p.addAttachedFiles(files);
  ```

- If you do not have a MATLAB Parallel Server cluster, use your local cluster. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

Start your cluster before running the code.

```
% 1) Load model and initialize the pool.
model = 'sldemo_suspn_3dof';
load_system(model);
parpool;
```

```
% 2) Set up the iterations that we want to compute.
Cf                  = evalin('base', 'Cf');
Cf_sweep            = Cf*(0.05:0.1:0.95);
iterations          = length(Cf_sweep);
simout(iterations)  = Simulink.SimulationOutput;

% 3) Need to switch all workers to a separate tempdir in case
% any code is generated for instance for StateFlow, or any other
% file artifacts are  created by the model.
spmd
    % Setup tempdir and cd into it
    addpath(pwd);
    currDir = pwd;
    addpath(currDir);
    tmpDir = tempname;
    mkdir(tmpDir);
    cd(tmpDir);
    % Load the model on the worker
    load_system(model);
end

% 4) Loop over the number of iterations and perform the
% computation for different parameter values.
parfor idx=1:iterations
    set_param([model '/Road-Suspension Interaction'],'MaskValues',...
        {'Kf',num2str(Cf_sweep(idx)),'Kr','Cr'});
    simout(idx) = sim(model, 'SimulationMode', 'normal');
end

% 5) Switch all of the workers back to their original folder.
spmd
    cd(currDir);
    rmdir(tmpDir,'s');
    rmpath(currDir);
    close_system(model, 0);
end

close_system(model, 0);
delete(gcp('nocreate'));
```

sim in parfor with Rapid Accelerator Mode

Running Rapid Accelerator simulations in `parfor` combines speed with automatic distribution of a prebuilt executable to the `parfor` workers. As a result, this mode eliminates duplication of the update diagram phase.

To run parallel simulations in Rapid Accelerator simulation mode using the `sim` and `parfor` commands:

- Configure the model to run in Rapid Accelerator simulation mode.
- Save changes to your model before simulating in `parfor`. The saved copy of your model is distributed to parallel workers when simulating in `parfor`.
- Ensure that the Rapid Accelerator target is already built and up to date.
- Disable the Rapid Accelerator target up-to-date check by setting the `sim` command option `RapidAcceleratorUpToDateCheck` to `'off'`.

To satisfy the second condition, you can change parameters only between simulations that do not require a model rebuild. In other words, the structural checksum of the model must remain the same. Hence, you can change only tunable block diagram parameters and tunable run-time block parameters between simulations. For a discussion on tunable parameters that do not require a rebuild subsequent to their modifications, see “Determine If the Simulation Will Rebuild” on page 35-7.

To disable the Rapid Accelerator target up-to-date check, use the `sim` command, as shown in this sample.

```
parpool;
% Load the model and set parameters
model = 'vdp';
load_system(model);
% Build the Rapid Accelerator target
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(model);
% Run parallel simulations
parfor i=1:4
   simOut{i} = sim(model,'SimulationMode', 'rapid',...
               'RapidAcceleratorUpToDateCheck', 'off',...
               'SaveTime', 'on',...
               'StopTime', num2str(10*i));
   close_system(model, 0);
end

close_system(model, 0);
delete(gcp('nocreate'));
```

In this example, the call to the `buildRapidAcceleratorTarget` function generates code once. Subsequent calls to `sim` with the `RapidAcceleratorUpToDateCheck` option `off` guarantees that code is not regenerated. Data concurrency issues are thus resolved.

When you set `RapidAcceleratorUpToDateCheck` to `'off'`, changes that you make to block parameter values in the model (for example, by using block dialog boxes, by using the `set_param` function, or by changing the values of MATLAB variables) do not affect the simulation. Instead, use `RapidAcceleratorParameterSets` to pass new parameter values directly to the simulation.

Workspace Access Issues

Workspace Access for MATLAB worker sessions

By default, to run `sim` in `parfor`, a parallel pool opens automatically, enabling the code to run in parallel. Alternatively, you can also first open MATLAB workers using the `parpool` command. The `parfor` command then runs the code within the `parfor` loop in these MATLAB worker sessions. The MATLAB workers, however, do not have access to the workspace of the MATLAB client session where the model and its associated workspace variables have been loaded. Hence, if you load a model and define its associated workspace variables outside of and before a `parfor` loop, then neither is the model loaded, nor are the workspace variables defined in the MATLAB worker sessions where the `parfor` iterations are executed. This is typically the case when you define model parameters or external inputs in the base workspace of the client session. These scenarios constitute workspace access issues.

Transparency Violation

When you run `sim` in `parfor` with `srcWorkspace` set to `current`, Simulink uses the `parfor` workspace, which is a transparent workspace. Simulink then displays an error for transparency

violation. For more information on transparent workspaces, see “Ensure Transparency in parfor-Loops or spmd Statements” (Parallel Computing Toolbox) .

Data Dictionary Access

When a model is linked to a data dictionary (see “What Is a Data Dictionary?” on page 74-2), to write code in `parfor` that accesses a variable or object that you store in the dictionary, you must use the functions `Simulink.data.dictionary.setupWorkerCache` and `Simulink.data.dictionary.cleanupWorkerCache` to prevent access issues. For an example, see “Sweep Variant Control Using Parallel Simulation” on page 26-15.

Resolving Workspace Access Issues

When a Simulink model is loaded into memory in a MATLAB client session, it is only visible and accessible in that MATLAB session; it is not accessible in the memory of the MATLAB worker sessions. Similarly, the workspace variables associated with a model that are defined in a MATLAB client session (such as parameters and external inputs) are not automatically available in the worker sessions. You must therefore ensure that the model is loaded and that the workspace variables referenced in the model are defined in the MATLAB worker session by using the following two methods.

- In the `parfor` loop, use the `sim` command to load the model and to set parameters that change with each iteration. (Alternative: load the model and then use the `g(s)et_param` command(s) to set the parameters in the `parfor` loop)
- In the `parfor` loop, use the MATLAB `evalin` and `assignin` commands to assign data values to variables.

Alternatively, you can simplify the management of workspace variables by defining them in the model workspace. These variables will then be automatically loaded when the model is loaded into the worker sessions. There are, however, limitations to this method. For example, you cannot store signal objects that use a storage class other than `Auto` in a model workspace. For a detailed discussion on the model workspace, see “Model Workspaces” on page 67-119.

Specifying Parameter Values Using the sim Command

Use the `sim` command in the `parfor` loop to set parameters that change with each iteration.

```
%Specifying Parameter Values Using the sim Command

model = 'vdp';
load_system(model)

%Specifying parameter values.
paramName = 'StopTime';
paramValue = {'10', '20', '30', '40'};

% Run parallel simulations
parfor i=1:4
    simOut{i} = sim(model, ...
                    paramName, paramValue{i}, ...
                    'SaveTime', 'on'); %#ok
end

close_system(model, 0);
```

An equivalent method is to load the model and then use the `set_param` command to set the `paramName` in the `parfor` loop.

Specifying Variable Values Using the assignin Command

You can pass the values of model or simulation variables to the MATLAB workers by using the `assignin` or the `evalin` command. The following example illustrates how to use this method to load variable values into the appropriate workspace of the MATLAB workers.

```
parfor i = 1:4
    assignin('base', 'extInp', paramValue{i})%#ok
    % 'extInp' is the name of the variable in the base
    % workspace which contains the External Input data
    simOut{i} = sim(model, 'ExternalInput', 'extInp'); %#ok
end
```

Sweep Variant Control Using Parallel Simulation

To use parallel simulation to sweep a variant control (a `Simulink.Parameter` object whose value influences the variant condition of a `Simulink.Variant` object) that you store in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and variant control to match your application.

To sweep block parameter values or the values of workspace variables that you use to set block parameters, use `Simulink.SimulationInput` objects instead of the programmatic interface to the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

```
% For convenience, define names of model and data dictionary
model = 'mySweepMdl';
dd = 'mySweepDD.sldd';

% Define the sweeping values for the variant control
CtrlValues = [1 2 3 4];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(CtrlValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
    dictObj = Simulink.data.dictionary.open(dd);
    sectObj = getSection(dictObj,'Design Data');
    entryObj = getEntry(sectObj,'MODE');
    % Suppose MODE is a Simulink.Parameter object stored in the data dictionary

    % Modify the value of MODE
    temp = getValue(entryObj);
    temp.Value = CtrlValues(index);
    setValue(entryObj,temp);
```

```
    % Simulate and store simulation output in the nondistributed array
    simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and
    % close the dictionary when finished with an iteration of the parfor-loop
    discardChanges(dictObj);
    close(dictObj);
end

% Restore default settings that were changed by the function
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model

spmd
    bdclose(model)
    Simulink.data.dictionary.cleanupWorkerCache
end
```

Note If data dictionaries are open, you cannot use the command `Simulink.data.dictionary.cleanupWorkerCache`. To identify open data dictionaries, use `Simulink.data.dictionary.getOpenDictionaryPaths`.

Data Concurrency Issues

Data concurrency issues refer to scenarios for which software makes simultaneous attempts to access the same file for data input or output. In Simulink, they primarily occur as a result of the nonsequential nature of the `parfor` loop during simultaneous execution of Simulink models. The most common incidences arise when code is generated or updated for a simulation target of a Stateflow, Model block or MATLAB Function block during parallel computing. The cause, in this case, is that Simulink tries to concurrently access target data from multiple worker sessions. Similarly, To File blocks may simultaneously attempt to log data to the same files during parallel simulations and thus cause I/O errors. Or a third-party blockset or user-written S-function may cause a data concurrency issue while simultaneously generating code or files.

A secondary cause of data concurrency is due to the unprotected access of network ports. This type of error occurs, for example, when a Simulink product provides blocks that communicate via TCP/IP with other applications during simulation. One such product is the HDL Verifier™ for use with the Mentor Graphics® ModelSim® HDL simulator.

Resolving Data Concurrency Issues

The core requirement of `parfor` is the independence of the different iterations of the `parfor` body. This restriction is not compatible with the core requirement of simulation via incremental code generation, for which the simulation target from a prior simulation is reused or updated for the current simulation. Hence during the parallel simulation of a model that involves code generation (such as Accelerator mode simulation), Simulink makes concurrent attempts to access (update) the simulation target. However, you can avoid such data concurrency issues by creating a temporary folder within the `parfor` loop and then adding several lines of MATLAB code to the loop to perform the following steps:

1. Change the current folder to the temporary, writable folder.
2. In the temporary folder, load the model, set parameters and input vectors, and simulate the model.

3 Return to the original, current folder.

4 Remove the temporary folder and temporary path.

In this manner, you avoid concurrency issues by loading and simulating the model within a separate temporary folder. Following are examples that use this method to resolve common concurrency issues.

A Model with Stateflow, MATLAB Function Block, or Model Block

In this example, either the model is configured to simulate in Accelerator mode or it contains a Stateflow, a MATLAB Function block, or a Model block (for example, `sf_bounce`, `sldemo_autotrans`, or `sldemo_mdlref_basic`). For these cases, Simulink generates code during the initialization phase of simulation. Simulating such a model in `parfor` would cause code to be generated to the same files, while the initialization phase is running on the worker sessions. As illustrated below, you can avoid such data concurrency issues by running each iteration of the `parfor` body in a different temporary folder.

```
parfor i=1:4
   cwd = pwd;
   addpath(cwd)
   tmpdir = tempname;
   mkdir(tmpdir)
   cd(tmpdir)
   load_system(model)
   % set the block parameters, e.g., filename of To File block
   set_param(someBlkInMdl, blkParamName, blkParamValue{i})
   % set the model parameters by passing them to the sim command
   out{i} = sim(model, mdlParamName, mdlParamValue{i});
   close_system(model,0);
   cd(cwd)
   rmdir(tmpdir,'s')
   rmpath(cwd)
end
```

Note the following:

- You can also avoid other concurrency issues due to file I/O errors by using a temporary folder for each iteration of the `parfor` body.
- On Windows platforms, consider inserting the `evalin('base', 'clear mex');` command before `rmdir(tmpdir, 's')`. This sequence closes MEX-files first before calling `rmdir` to remove `tmpdir`.

```
evalin('base', 'clear mex');
rmdir(tmpdir, 's')
```

A Model with To File Blocks

If you simulate a model with To File blocks from inside of a `parfor` loop, the nonsequential nature of the loop may cause file I/O errors. To avoid such errors during parallel simulations, you can either use the temporary folder idea above or use the `sim` command in Rapid Accelerator mode with the option to append a suffix to the file names specified in the model To File blocks. By providing a unique suffix for each iteration of the `parfor` body, you can avoid the concurrency issue.

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(model);
       parfor idx=1:4
       sim(model, ...
```

```
    'ConcurrencyResolvingToFileSuffix', num2str(idx),...
    'SimulationMode', 'rapid',...
    'RapidAcceleratorUpToDateCheck', 'off');
end
```

See Also

Related Examples

- “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38
- “Sweep Variant Control Using Parallel Simulation” on page 26-15

Error Handling in Simulink Using MSLException

Error Reporting in a Simulink Application

Simulink allows you to report an error by throwing an exception using the `MSLException` object , which is a subclass of the MATLAB `MException` class. As with the MATLAB `MException` object, you can use a `try-catch` block with a `MSLException` object construct to capture information about the error. The primary distinction between the `MSLException` and the `MException` objects is that the `MSLException` object has the additional property of handles. These handles allow you to identify the object associated with the error.

The MSLException Class

The `MSLException` class has five properties: `identifier`, `message`, `stack`, `cause`, and `handles`. The first four of these properties are identical to those of `MException`. For detailed information about them, see `MException`. The fifth property, `handles`, is a cell array with elements that are double array. These elements contain the handles to the Simulink objects (blocks or block diagrams) associated with the error.

Methods of the MSLException Class

The methods for the `MSLException` class are identical to those of the `MException` class. For details of these methods, see `MException`.

Capturing Information about the Error

The structure of the Simulink `try-catch` block for capturing an exception is:

```
try
        Perform one or more operations
catch E
        if isa(E, 'MSLException')
...
end
```

If an operation within the `try` statement causes an error, the `catch` statement catches the exception (*E*). Next, an `if isa` conditional statement tests to determine if the exception is Simulink specific, i.e., an `MSLException`. In other words, an `MSLException` is a type of `MException`.

The following code example shows how to get the handles associated with an error.

```
errHndls = [];
try
    sim('ModelName', ParamStruct);
catch e
    if isa(e,'MSLException')
            errHndls = e.handles{1}
    end
end
```

You can see the results by examining *e*. They will be similar to the following output:

```
e =
```

```
  MSLException

  Properties:
       handles: {[7.0010]}
    identifier: 'Simulink:Parameters:BlkParamUndefined'
       message: [1x87 char]
         cause: {0x1 cell}
         stack: [0x1 struct]

  Methods, Superclasses
```

To identify the name of the block that threw the error, use the `getfullname` command. For the present example, enter the following command at the MATLAB command line:

```
getfullname(errHndls)
```

If a block named Mu threw an error from a model named *vdp*, MATLAB would respond to the `getfullname` command with:

```
ans =
vdp/Mu
```

See Also

Related Examples

- “Run Simulations Programmatically” on page 26-2
- “Run Multiple Simulations” on page 27-2

Multiple Simulations

Run Multiple Simulations

For workflows that involve multiple parallel simulations and logging of large amounts of data, you can create simulation sets by using an array of `Simulink.SimulationInput` objects. This is useful in scenarios like model testing, experiment design, Monte Carlo analysis, and model optimization.

Using arrays of `Simulink.SimulationInput` objects simplify the running of multiple simulations and running them in parallel. With the Parallel Computing Toolbox, you can use the `parsim` and `batchsim` commands to run the simulations in parallel.

The `parsim` command distributes each simulation to your workers to decrease your overall simulation time. The `parsim` command automates the creation of a parallel pool, identifying file dependencies and managing build artifacts for accelerator and rapid accelerator simulations.

The `batchsim` command offloads the simulations to the compute cluster. The execution of the simulations takes place on the cluster, giving you the ability to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

In the absence of a Parallel Computing Toolbox license, the `parsim` behaves like the `sim` command. The simulations then run in serial.

The `batchsim` command uses the Parallel Computing Toolbox™ license to run the simulations on compute cluster. `batchsim` runs the simulations in serial if a parallel pool cannot be created. If Parallel Computing Toolbox license is not used, `batchsim` errors out.

You can make changes to your model using the `Simulink.SimulationInput` object and run a simulation in parallel with those changes. Changing the `Simulink.SimulationInput` object, overrides the values in the model. The simulation uses the values in the `Simulink.SimulationInput` object rather than the values defined in the model. This way, you can change the model without dirtying it. The `Simulink.SimulationInput` object allows you to change these settings in your model:

- Initial state
- External inputs
- Model parameters
- Block parameters
- Variables

Through the `Simulink.SimulationInput` object, you can also specify MATLAB functions to run at the start and the end of each simulation by using `PreSimFcn` and `PostSimFcn` respectively.

When you use `Simulink.SimulationInput` objects, the model parameters are restored after the simulation ends. See “Run Parallel Simulations Using parsim” on page 27-5.

Note When the pool is not already open and simulations are run for the first time, simulations take an additional time to start. Subsequent parallel simulations are faster.

Other Advantages

- Outputs errors in the simulation output object for easier debugging

- Compatible with rapid accelerator and fast restart
- Compatible with file logging (to facilitate big data)
- Compatible with MATLAB Parallel Server in addition to local parallel pools
- Capable of transferring base workspace variables to workers
- Avoids transparency errors

Simulation Manager

The Simulation Manager allows you to monitor multiple parallel simulations. It shows the progress of the runs as they are running in parallel. You can view the details of every run such as parameters, elapsed time, and diagnostics. The Simulation Manager acts as a useful tool by giving you the option to analyze and compare your results in the Simulation Data Inspector. You can also select a run and apply its values to the model. For more information, see `Simulation Manager`.

Data Logging for Multiple Simulations

The resulting `Simulink.SimulationOutput` object, which contains the simulation outputs, captures error messages and the simulation metadata. When you select the **Data Import/Export > Log Dataset data to file** configuration parameter, Simulink creates a `Simulink.SimulationData.DatasetRef` object for each `Dataset` stored in the resulting MAT file. You can use the `DatasetRef` object to access the data for a `Dataset` element. For simulations that are run using the `Simulink.SimulationInput` objects, the `DatasetRef` object is returned as part of the `SimulationOutput` object. As a result, you have quicker access to and do not need to create them.

Parallel simulations can produce more logged data than the MATLAB memory can hold. Consider logging to persistent storage for parallel simulations to reduce the memory requirement. When you select the **Data Import/Export > Log Dataset data to file** configuration parameter (`LoggingToFile`), for parallel simulations in Simulink:

- Data is logged in Dataset format in a MAT-file
- A `Simulink.SimulationData.DatasetRef` object is created for each Dataset element (for example, `logsout`) for each simulation

You can use `DatasetRef` objects to access data for a specific signal. You can create `matlab.io.datasetore.SimulationDatastore` objects to use for streaming logged data from persistent storage in to a model.

See Also

`Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

More About

- “Run Parallel Simulations” on page 26-7
- “Run Parallel Simulations Using parsim” on page 27-5
- “Multiple Simulation Workflows” on page 27-9

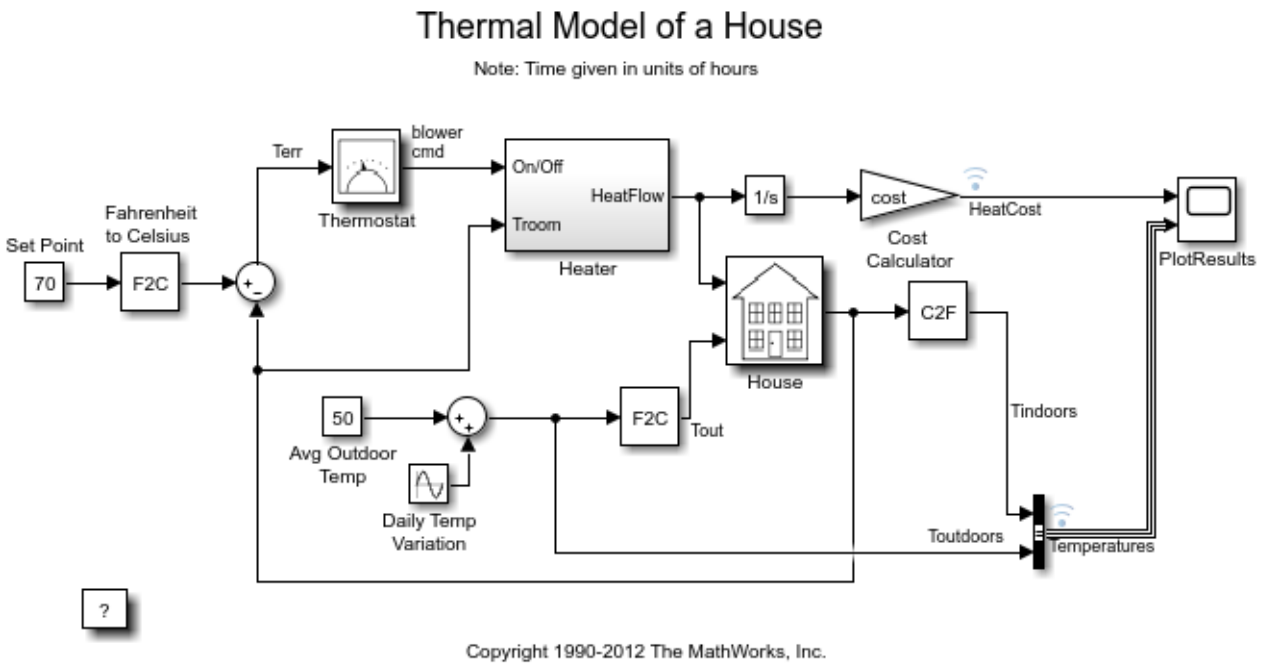
- “Work with Big Data for Simulations” on page 72-29

Run Parallel Simulations Using parsim

This example shows how to use a `Simulink.SimulationInput` object to change block and model parameters and run simulations in parallel with those changes.

The example model `sldemo_househeat` is a system that models the thermal characteristics of a house, its outdoor environment, and a house heating system. This model calculates heating costs for a generic house.

Set Point block, Thermostat subsystem, Heater subsystem, House subsystem, and Cost Calculator component are the main components. For a detailed explanation of the model, see “Thermal Model of a House”.



Run Multiple Parallel Simulations with Different Set Points

This model uses a Constant block to specify a temperature set point that must be maintained indoors. The default value of set point value is 70 degrees Fahrenheit. This example shows you how to simulate the model in parallel using different values of `Set Point`.

Open the example model.

```
open_system('sldemo_househeat');
```

Define a set of values for `Set Point`.

```
SetPointValues = 65:2:85;
spv_length = length(SetPointValues);
```

Using the defined values, initialize an array of `Simulink.SimulationInput` objects. Use these `Simulink.SimulationInput` objects to specify the `Set Point` values. In this step, to preallocate the array, the loop index is made to start from the largest value.

```
for i = spv_length:-1:1
    in(i) = Simulink.SimulationInput('sldemo_househeat');
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(SetPointValues(i)));
end
```

This example produces an array of 10 `Simulink.SimulationInput` objects, each corresponding to a different value of `Set Point`.

Now, run these multiple simulations in parallel using the command `parsim`. To monitor and analyze the runs, open the `Simulation Manager` by setting the `ShowSimulationManager` argument to `on`. The `ShowProgress` argument when set to `on` shows the progress of the simulations.

```
out = parsim(in,'ShowSimulationManager','on','ShowProgress','on')
```

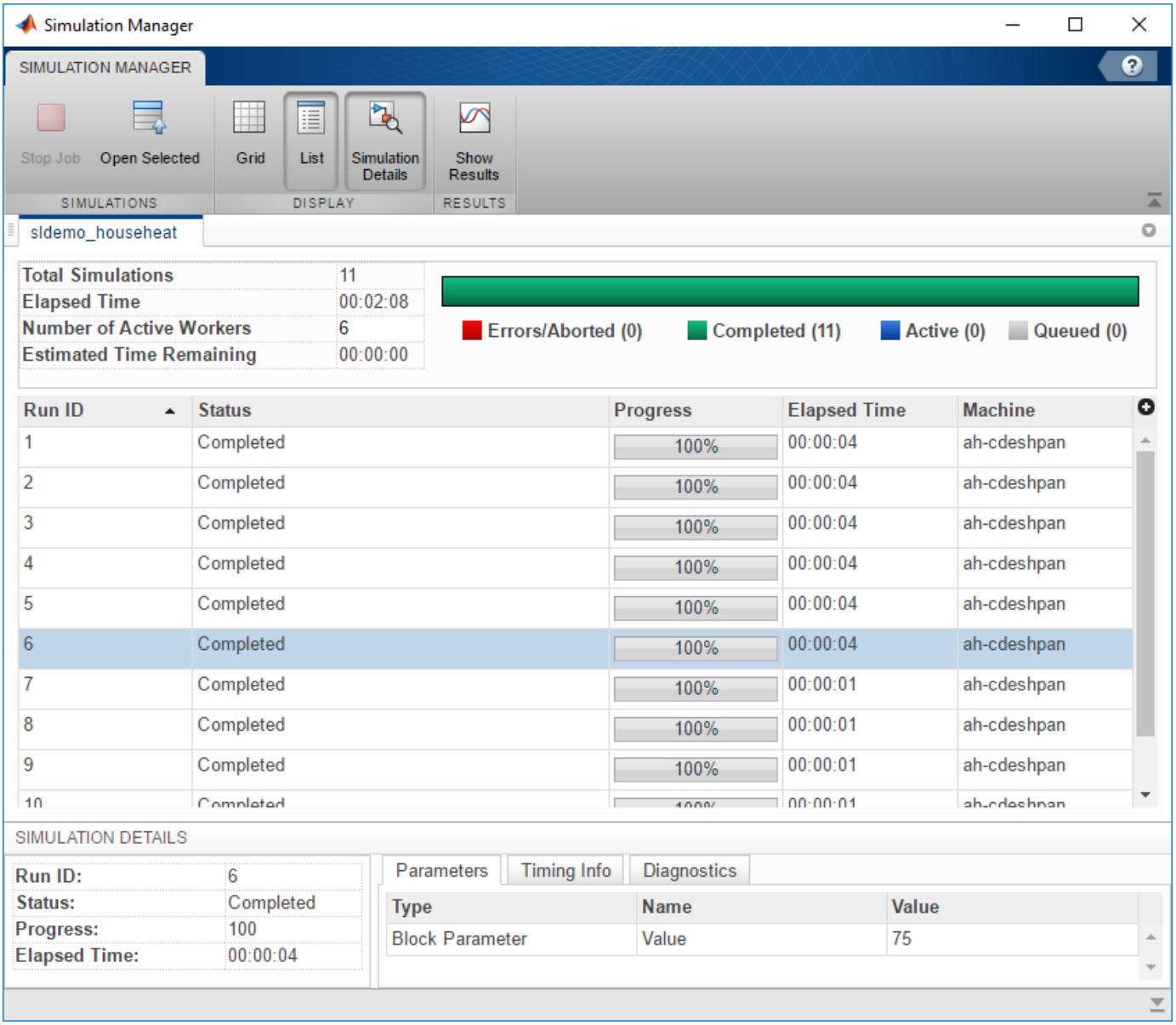
The output is generated as a `Simulink.SimulationOutput` object. To see all of the different set point values, open the plot of the Temperatures (Indoor and Outdoor) and the Heat Cost component. The constant block Avg Outdoor Temp specifies the average air temperature outdoors. The Daily Temp Variation Sine Wave block generates daily temperature fluctuations of outdoor temperature. The indoor temperature is derived from the House subsystem. The temperature outdoor varies sinusoidally, whereas the temperature indoors is maintained within 5 degrees Fahrenheit of the set point.

In the absence of the Parallel Computing Toolbox licenses, the `parsim` command behaves like the `sim` command. The simulations run in serial.

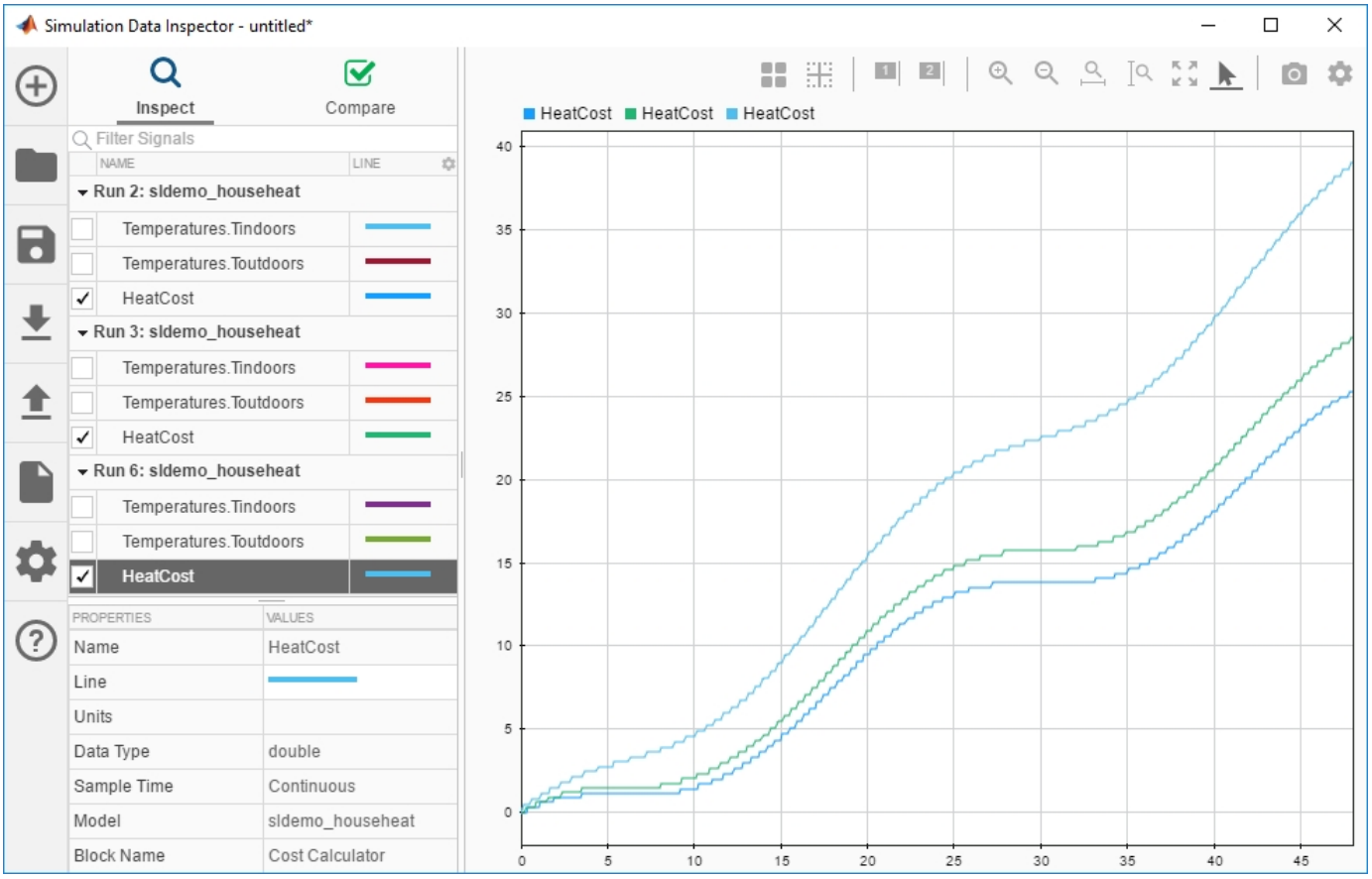
View the Runs in the Simulation Manager

Setting the `ShowSimulationManager` argument to `on` enables the Simulation Manager. For more information, see `Simulation Manager`.

You can view the status of all the runs and detailed information about them.



The Simulation Manager enables you to view your results in the Simulation Data Inspector, which in turn allows you to analyze and compare your data. You can view the plot of the Temperatures (Indoor and Outdoor) and the Heat Cost in Simulation Data Inspector. Select the runs for which you want to view the plot and click  icon.



You can now see the heat cost for three simulations.

Using the Simulation Manager, you can apply the parameters of any run to your model. Now, suppose that you want to apply the parameters of Run 3 to your model. Select Run 3 and click the  icon. Your parameters are applied to the model.

See Also

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

More About

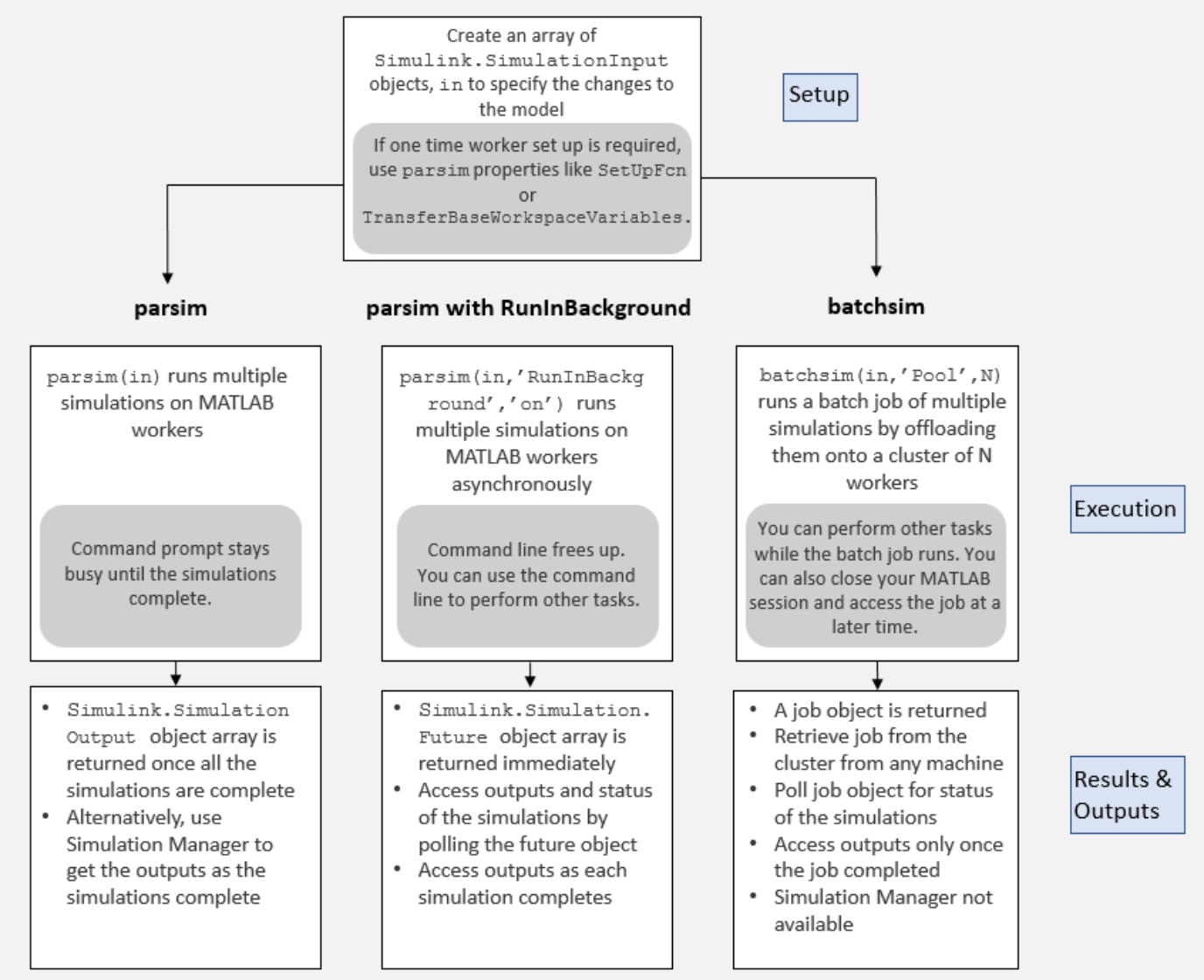
- “Run Parallel Simulations” on page 26-7
- “Multiple Simulation Workflows” on page 27-9
- “Run Multiple Simulations” on page 27-2

Multiple Simulation Workflows

When running a set of multiple simulations, you can run them in parallel on multiple MATLAB workers in a parallel pool. To run multiple simulations, you can use `parsim`, `parsim` with `'RunInBackground'` option turned on, or `batchsim`.

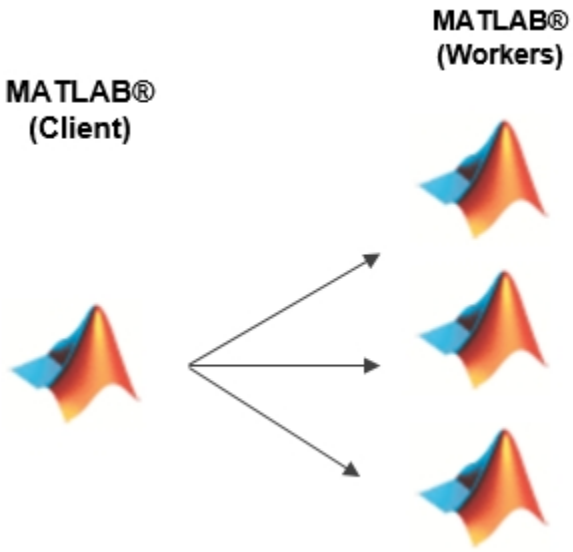
The flow chart shows how running multiple simulations with `parsim`, `parsim` with `RunInBackground` and `batchsim` differ.

The `parsim` and `batchsim` commands use the Parallel Computing Toolbox license to run the simulations in parallel. `parsim` runs the simulations in serial if a parallel pool cannot be created or if the Parallel Computing Toolbox license is not available. `batchsim` commands cannot run without Parallel Computing Toolbox license.



parsim Workflow

Using `parsim` command with Parallel Computing Toolbox to run multiple simulations sets up a parallel pool automatically and runs simulations in parallel. The client is always tied to the MATLAB workers.



Basic Parsim Workflow

1. Create an array of `Simulink.SimulationInput` objects, `in`, to specify changes to the model.
2. Specify one-time setup required for running multiple simulations. You can use `SetupFcn` and `TransferBaseWorkspaceVariables` to perform setup on the parallel workers.
3. Run `parsim(in)` to execute these multiple simulations in parallel. If a parallel pool does not exist, `parsim` creates it. `parsim` uses default settings.
4. You can open Simulation Manager by setting the `'SimulationManager'` argument to `'on'` with `parsim`, `parsim(in,'SimulationManager','on')`. Simulation Manager enables you to monitor the simulations as they are running and gives you access to outputs of the simulations when they are ready. Simulation Manager gives provides you with numerous information about the simulations running on the workers. For more information, see **Simulation Manager**.
5. Once all the simulations are complete, you get an array of `Simulink.SimulationOutput` objects.

Limitations

- Closing the MATLAB session terminates simulations on the workers, disabling retrieval of partial results.

parsim with RunInBackground Workflow

1. Create an array of `Simulink.SimulationInput` objects, `in`, to specify changes to the model.
2. Specify one-time setup required for running multiple simulations. You can use `SetupFcn` and `TransferBaseWorkspaceVariables` to perform setup on the parallel workers.

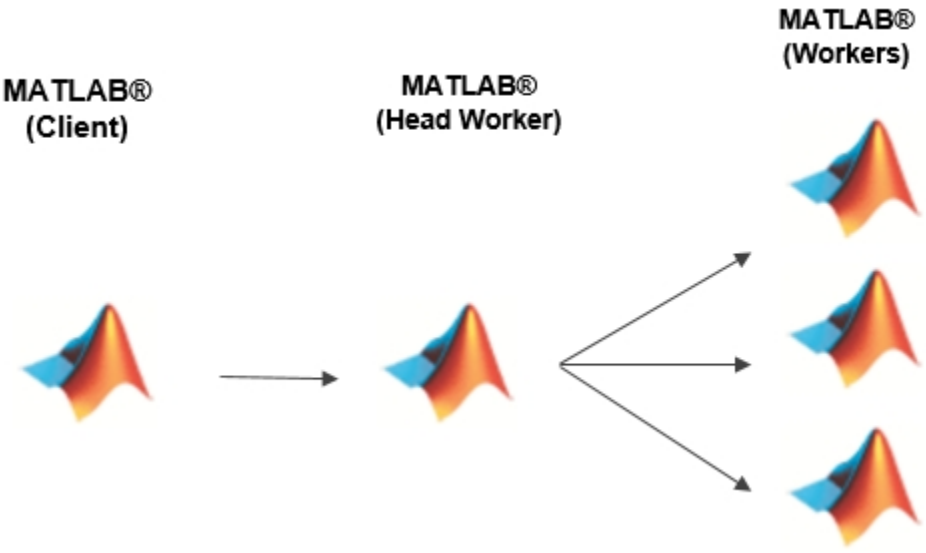
3. Run `parsim` with `RunInBackground` option set to `'on'`: `parsim(in,'RunInBackground','on')`. Setting the `'RunInBackground'` option to `'on'` runs the simulations asynchronously. This keeps the MATLAB command prompt available enabling you to work on other tasks.
4. With `'RunInBackground'` option set to `'on'`, `parsim` returns a `Simulink.Simulation.Future` object. You can poll this object to check the status of simulations, fetch the outputs of simulations when they are completed, or cancel simulations. For more information, see `Simulink.Simulation.Future`.

Limitations

- Closing the MATLAB session terminates simulations on the workers, disabling retrieval of partial results. If the `future` object is cleaned, you are subsequently unable to access the results of the simulations.
- Using a `fetchNext` method loop on `Future` objects, along with Simulation Manager causes them to compete for retrieval of `Future` objects. Use either a `fetchNext` next loop or Simulation Manager to get the outputs of completed simulations.

batchsim Workflow

A batch workflow typically means submitting jobs to run simulations on MATLAB workers and subsequently accessing the results of those simulations. When you run simulations in batches, you offload the execution of simulations onto a compute cluster. To learn more about batch processing, see “Simple Batch Processing” (Parallel Computing Toolbox).



1. Create an array of `Simulink.SimulationInput` objects, `in`, to specify changes to the model.
2. Specify one-time setup required for running multiple simulations. You can use `SetupFcn` and `TransferBaseWorkspaceVariables` to perform setup on the parallel workers.
3. To run on a cluster or desktop background, call `batchsim` to offload the execution of simulations. With `batchsim`, you can use most of the arguments that are compatible with `parsim` and `batch` commands. For more information, see `batchsim`.

Using `batchsim`, you offloaded simulations to a MATLAB worker. To run the batch job on multiple workers, specify a pool size, `N`, that is an integer specifying the number of workers to make into a parallel pool for the job: `batchsim(in, 'Pool', N)`. There must be at least `N+1` workers available on the cluster. If the pool size is not specified, `batchsim(in)` runs the simulations on a single worker in the cluster specified by the default cluster profile.

Note `batchsim` errors out when used with a pool size if Parallel Computing Toolbox license is not available.

4. `batchsim` offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.
5. On submitting a batch job, `batchsim` returns a job object containing a job ID and other information that you can use to access the batch job at a later time. Access this job again to check the progress of simulations.

Limitations

- Because the client session is not tied to the workers, you cannot access the outputs unless the job is finished.
- `batchsim` does not give you the ability to monitor simulations with Simulation Manager. For batch jobs, you can use a batch job monitor that tells you if the job is queues, in progress, or completed. For more information, see “Job Monitor” (Parallel Computing Toolbox)

See Also

Functions

`Simulation Manager` | `batch` | `batchsim` | `getSimulationJobs` | `parcluster` | `parsim`

Classes

`Simulink.Simulation.Future` | `Simulink.Simulation.Job` | `Simulink.SimulationInput`

See Also

More About

- “Run Multiple Simulations” on page 27-2
- “Run Parallel Simulations Using parsim” on page 27-5
- “Run Parallel Simulations” on page 26-7
- “Analyze Results Using Simulation Manager” on page 27-13
- “Batch Processing” (Parallel Computing Toolbox)

Analyze Results Using Simulation Manager

The Simulation Manager allows you to monitor multiple simulations, in serial or in parallel, and their progress. You can view the details of every run, such as parameters, elapsed time, and diagnostics. The Simulation Manager provides the option to analyze and compare your logged signal results in the **Simulation Data Inspector**. Through Simulation Manager, you can select a run and apply its values to the model. Simulation Manager opens when you run a `parsim` or a `sim` command with `ShowSimulationManager` argument set to `on`. For more information, see `Simulation Manager`.

The dimensions of the tank have an impact on the total cost of production of the product. For this example, we observe the behavior of `TotalCost` for different values of the width and the height. By analyzing the behavior of the parameters, we find the combination of `A` and `h` that results in lowest `TotalCost`. To solve this design problem, we run multiple simulations (in parallel or in serial) with different values of the parameters `A` and `h`.

This example shows how you can use the Simulation Manager to analyze the simulations and solve a design problem using a model of a continually stirred tank reactor. The reactors are specialized tanks that are used to mix various chemicals or compounds to create a product. The important variables used in this model are:

- Variable `A`, which represents the cross-sectional area of the tank (width).
- Variable `h`, which represents the height.
- Variable `TotalCost`, which represents the cost, in dollars, to produce a tankful of product.

Simulation Manager enables you to analyze the simulations as they are running. When the simulations are in progress, you can visualize the simulation data of the model by plotting the simulation outputs against the input parameters. Visualizing trend of the simulations as they are happening allows you to learn about the design space of the simulations and evaluate whether the simulations are running as required. You can stop the simulations if the progress is not as desired thus saving time.

Open Simulation Manager

In this example, use a set of sweep parameters provided to the model through `Simulink.SimulationInput` objects and then use the `parsim` command to run the simulations in parallel.

Create a `PostSimFcn` function as follows in a MATLAB script to call later in the parameter sweep. Name the file `calculateCost.m`. The `PostSimFcn` function calculates `TotalCost` from the variables `A` and `h`, and its calculation can vary depending on the application.

```
function costOutput = calculateCost(out)

    costOutput.yout = out.yout;

    coolantOutput = out.yout.get('Coolant').Values.Data;
    costCoolant = abs(mean(coolantOutput - 294))/30;
    costOutput.costFromCoolant = costCoolant;

    concentrationOutput = out.yout.get('Residual Concentration').Values.Data;
    costConcentration = 10*std(concentrationOutput - 2);
    costOutput.costFromConcentration = costConcentration;
```

```
    costOutput.TotalCost = costCoolant + costConcentration;

end
```

Open the model.

```
openExample('simulink/OpenTheModelToUseWithSimulationManagerExample');
open_system('simManagerCSTR');
```

Create a sample of values for parameter sweep.

```
rangeA = [0.1, 5];
rangeH = [0.1, 5];

rng default;

numSamples = 60;
allAValues = rangeA(1) + (rangeA(2) - rangeA(1)).*rand(numSamples, 1);
allhValues = rangeH(1) + (rangeH(2) - rangeH(1)).*rand(numSamples, 1);
```

Create an array of `Simulink.SimulationInput` objects. For this example, the `TotalCost` is calculated and returned using the `PostSimFcn`.

```
in(1:numSamples) = Simulink.SimulationInput('simManagerCSTR');
in = in.setPostSimFcn(@(out)calculateCost(out));
```

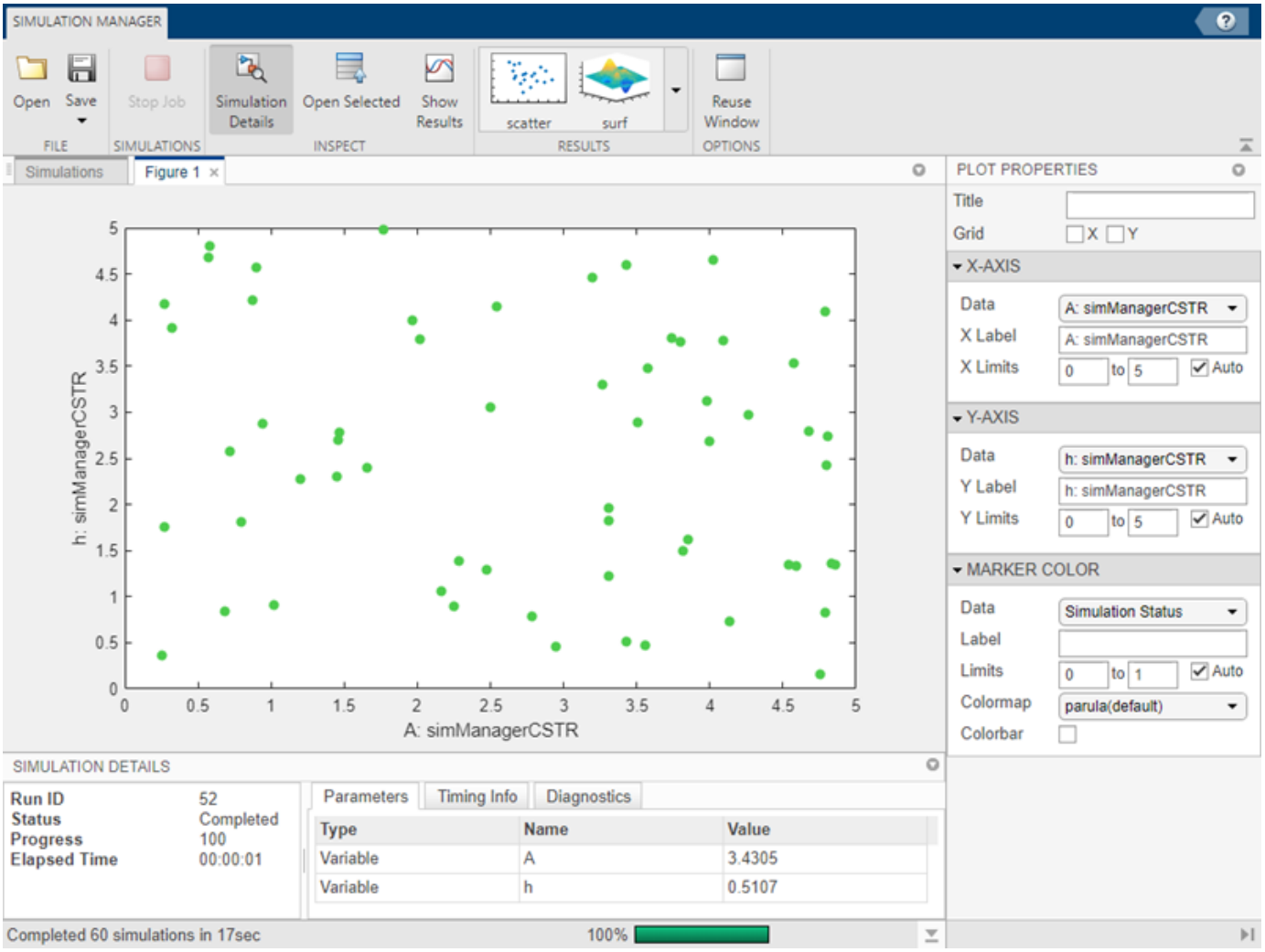
Run the simulations in parallel and open the Simulation Manager.

```
for k = 1:numSamples
    in(k) = in(k).setVariable('A', allAValues(k), 'Workspace', 'simManagerCSTR');
    in(k) = in(k).setVariable('h', allhValues(k), 'Workspace', 'simManagerCSTR');
end

out = parsim(in, 'ShowSimulationManager', 'on');
```

The default view in the Simulation Manager shows a scatter plot with two parameters on its X and Y axes. In this case, the variable `A` is on the X-axis and variable `h` is on the Y-axis. When the simulations are running you can see dots appear on the plot, color coded according to the simulation status. Completed simulations are marked in green, in-progress simulations are blue, and simulations with errors are red.

The **Plot Properties** panel on the right enables you to edit and configure the data that plot shows. By selecting grid for X and Y axes, the plot appears with grid lines.

If a particular parameter is a time-series, the Simulation Manager plots only the last value of the time series.



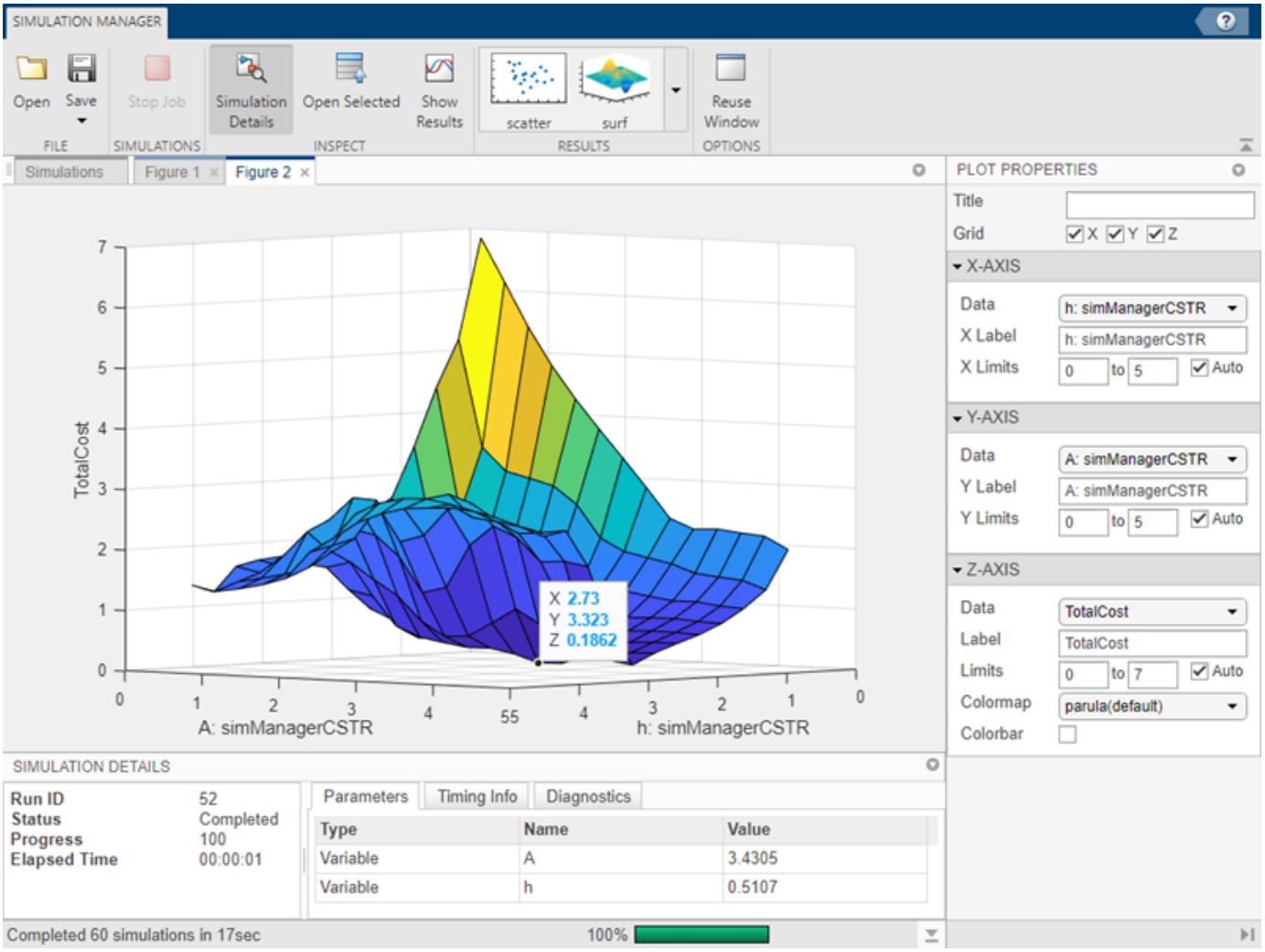
With the Simulation Manager, you can visualize the above data in a surf plot. Click the **surf** plot in the **Results** section of the toolstrip.

Add and Configure Plots

The **Results** section of the Simulation Manager allows you to add multiple plots and configure them. Click the **surface plot** in **Results** section of the Simulation Manager toolstrip. Using the plot properties, change the parameters to display on the plots. You can change properties such as the plot labels, axes labels and you can add a colormap to denote the third parameter. You can also change the value limits of the colormap.

With the second plot and the first plot together, you can determine the value of the variables `A` and `h`, that gives the best `TotalCost`.

For the Z-axis of the surf plot, change the **Data** to `TotalCost`. By observing the surf plot, you can find the lowest point of `TotalCost`. Clicking the lowest point shows the values for X-axis and Y-axis, which is `h` and `A`, respectively.



Save and Load Simulation Manager

Save the session information, which includes simulation data all the plot configurations. To save the session, click the **Save** button on the toolstrip. The session saves as a `.mldatx` file. To reopen the saved session, navigate to the location where the file is saved and double-click the `.mldatx` file.

The Simulation Manager allows you to reuse the plot configuration when you want to run similar simulations with different sets of values. To reuse the plot configuration, click the **Reuse Window** button on the toolstrip. Selecting the **Reuse Window** saves the plot configurations, such as labels, axis orientation, which parameters to display on which axis that you can reuse with the next simulation of the same model. When you use this functionality while running simulations for different models, the simulation errors out due to a limitation.

See Also

`Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

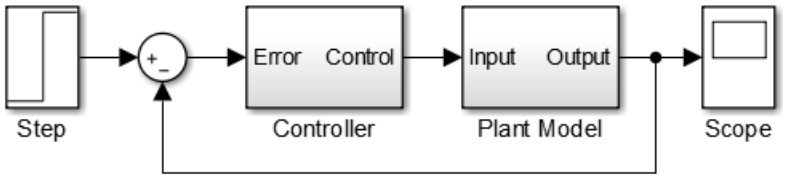
More About

- “Run Parallel Simulations” on page 26-7
- “Run Parallel Simulations Using parsim” on page 27-5
- “Multiple Simulation Workflows” on page 27-9
- “Work with Big Data for Simulations” on page 72-29

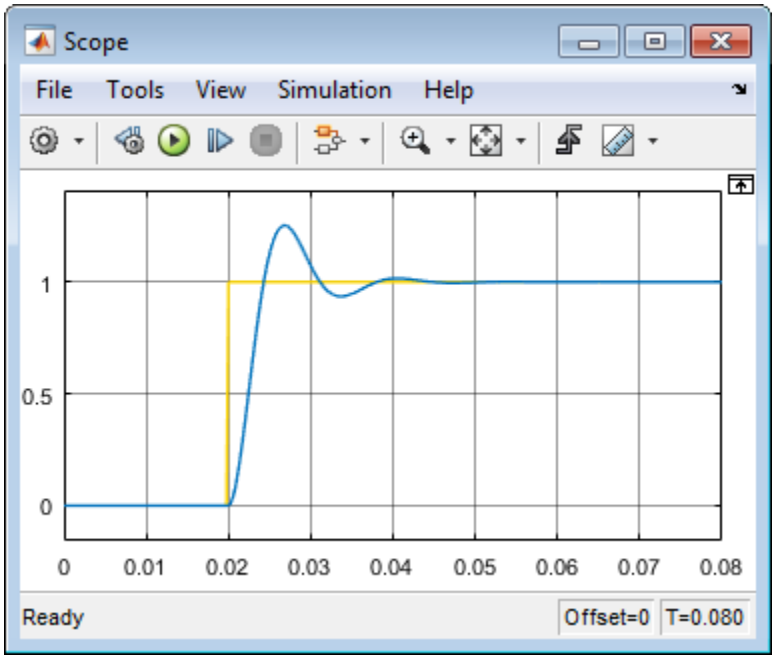
Visualizing and Comparing Simulation Results

Prototype and Debug Models with Scopes

Simulink scope blocks and viewers offer a quick and lightweight way to visualize your simulation data over time. If you are prototyping a model design, you can attach signals to a Scope block. After simulating the model, you can use the results to validate your design. See “Scope Blocks and Scope Viewer Overview” on page 28-6 and “Model and Validate a System”.

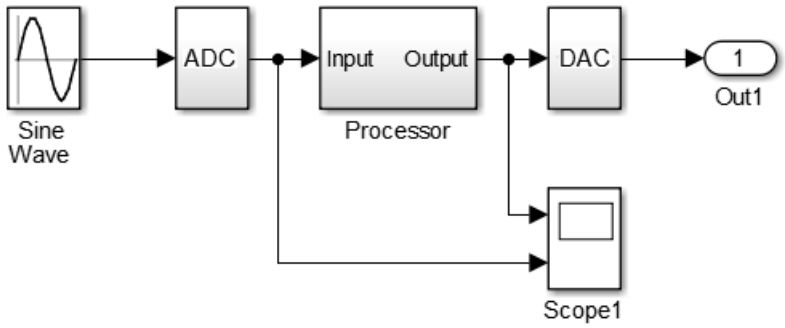


A Scope block or Scope viewer opens to a Scope window where you can display and evaluate simulation data.

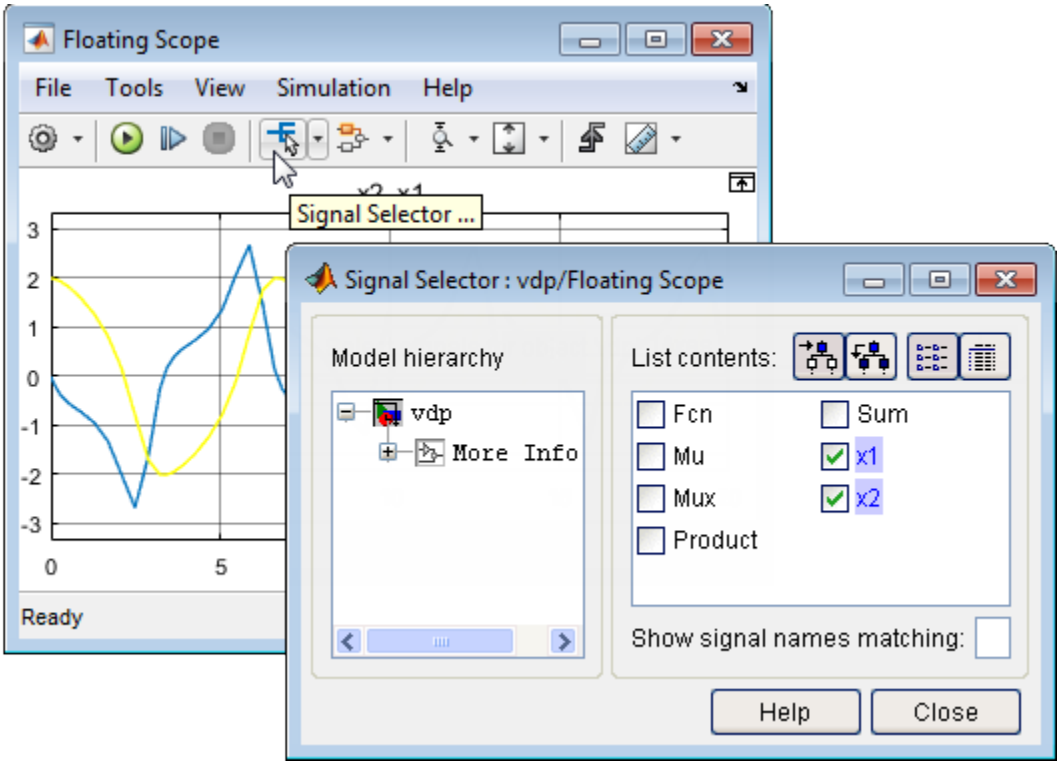


The toolbar contains controls for starting, stopping, and stepping forward through a simulation  . You can use these controls to debug a model by viewing signal data at each time interval. See “How Stepping Through a Simulation Works” on page 2-3.

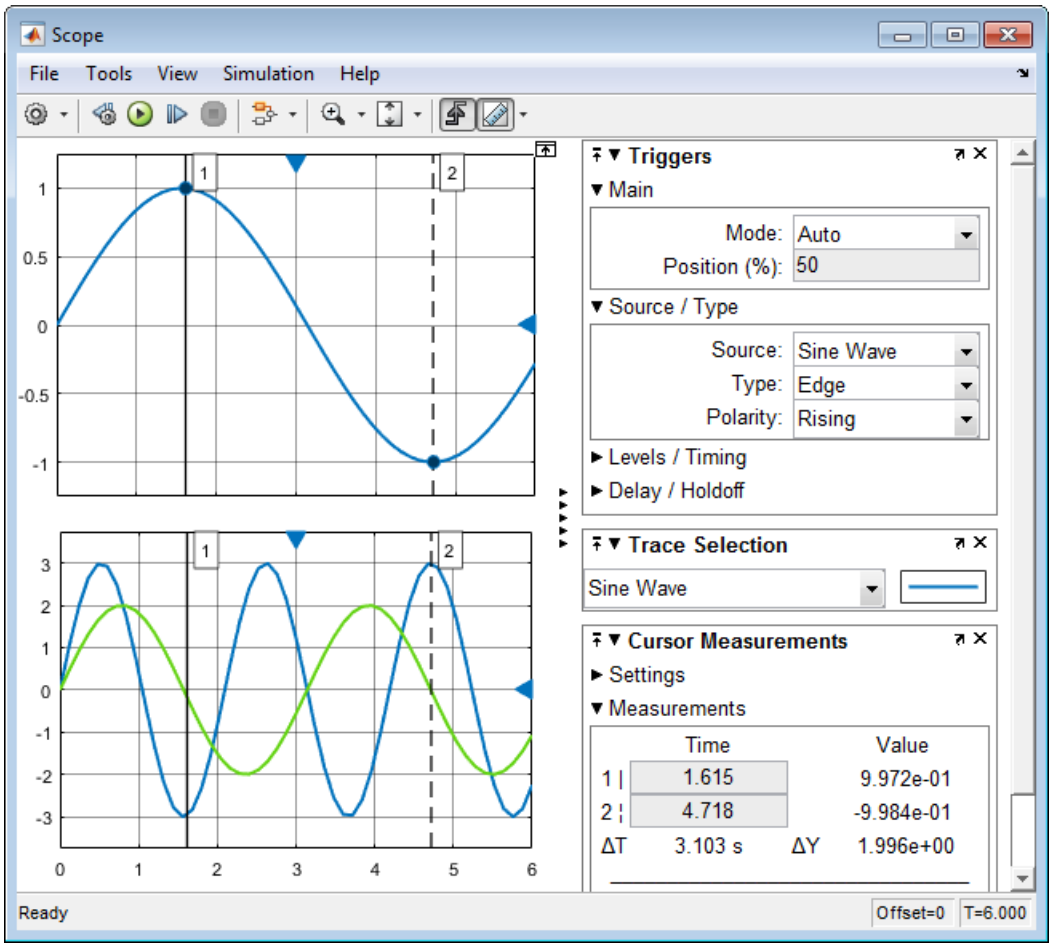
Connect signal lines to a Scope block using multiple input ports, see “Number of input ports”.



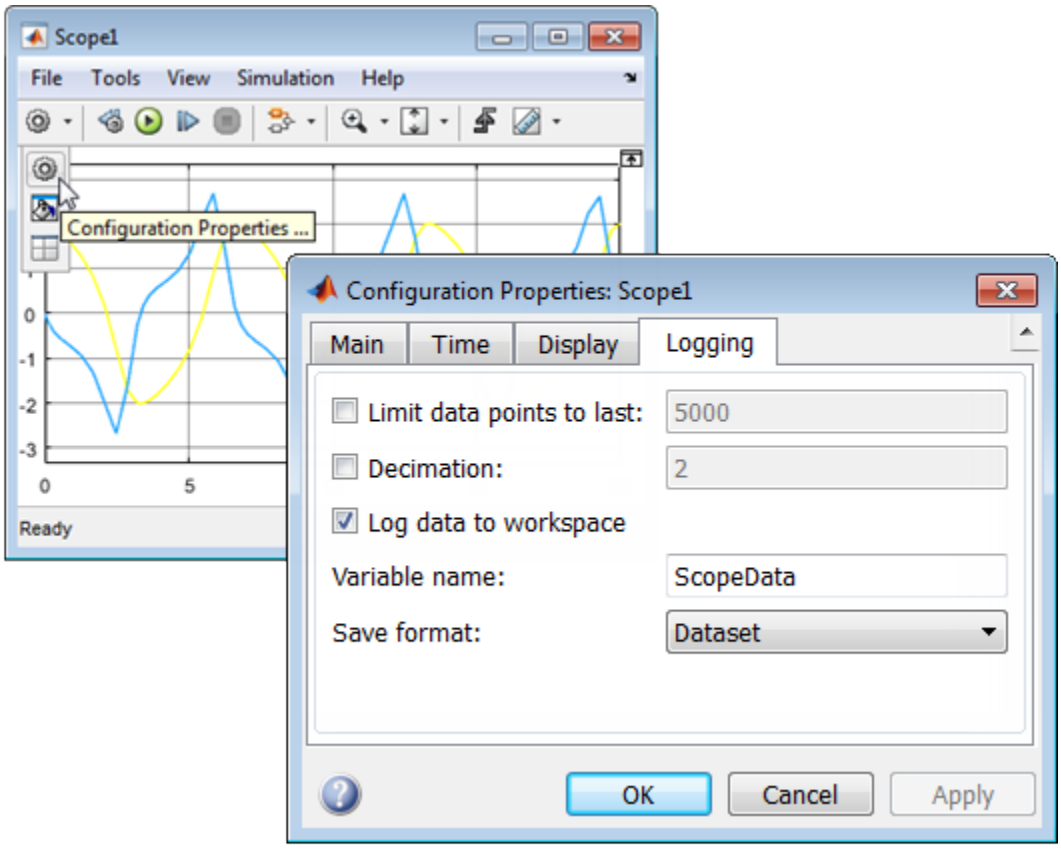
Attach signals to a Floating Scope block or signal viewer directly from the model. See “Add Signals to an Existing Floating Scope or Scope Viewer” on page 28-68.



Use the oscilloscope-like tools available with a scope to debug your model. Set triggers to capture events, use interactive cursors to measure signal values at various points, and review signal statistics such as maximum and mean values. See “Scope Triggers Panel” on page 28-12 and “Cursor Measurements Panel” on page 28-23.



Save or log signal data to the MATLAB workspace, and then plot data in a MATLAB figure widow. Use MATLAB functions or your own scripts to analyze the data. See “Save Simulation Data from Floating Scope” on page 28-69.



See Also

Floating Scope | Scope | Scope Viewer

Related Examples

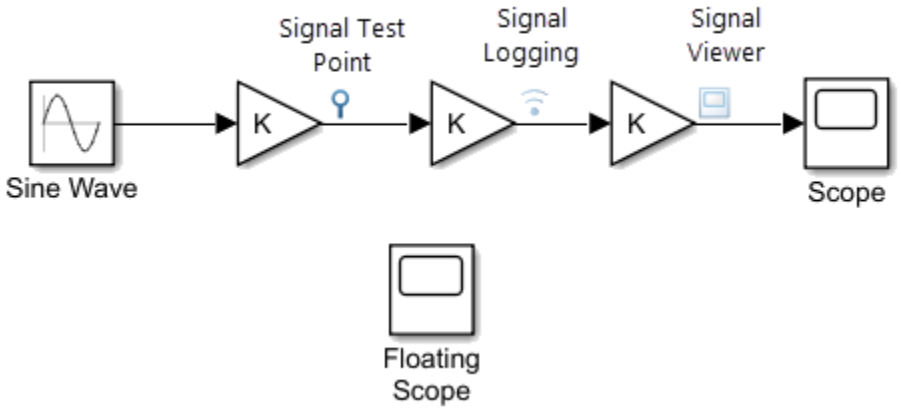
- “Common Scope Block Tasks” on page 28-51
- “Floating Scope and Scope Viewer Tasks” on page 28-67
- “Scope Triggers Panel” on page 28-12
- “Cursor Measurements Panel” on page 28-23
- “Control Scope Blocks Programmatically” on page 28-80
- “Scope Blocks and Scope Viewer Overview” on page 28-6
- “Viewers and Generators Manager” on page 28-77

Scope Blocks and Scope Viewer Overview

In this section...

Overview of Methods

Simulink scopes provide several methods for displaying simulation data and capturing the data for later analysis. Symbols on your block diagram represent the various data display and data capture methods.



For more information about these methods:

- Scope and Floating Scope blocks — Scope, Floating Scope, “Common Scope Block Tasks” on page 28-51, “Floating Scope and Scope Viewer Tasks” on page 28-67.
- Scope Viewer — “Viewers and Generators Manager” on page 28-77, “Floating Scope and Scope Viewer Tasks” on page 28-67.
- Signal Logging — “Save Simulation Data from Floating Scope” on page 28-69.
- Signal Test Point — “Configure Signals as Test Points” on page 75-43.

Simulink Scope Versus Floating Scope

Scope blocks and Floating Scope blocks both display simulation results, but they differ in how you attach signals and save data. Simulation behavior for a Floating Scope and a Scope Viewer is identical, but you manage them differently in your model.

Capability	Simulink Scope	Simulink Floating Scope	Simulink Scope Viewer
Attaching signals	Connect signal lines to a Scope block using input ports.	Attach signals interactively from the model before and during a simulation. See "Add Signals to an Existing Floating Scope or Scope Viewer" on page 28-68 and "Quickly Switch Visualization of Different Signals on a Floating Scope" on page 28-73.	Attach signals from the Viewers and Generators Manager, interactively from the toolstrip, or using the signal line context menu.
Access to signals	Because signals lines are connected to a Scope block, access signals at different levels of a model hierarchy using GoTo blocks.	Because signals are attached without signal lines, you do not have to route lines to a Floating Scope block. You can access most signals inside the model hierarchy, including referenced models and Stateflow charts. You cannot access optimized signals.	Scope viewers are attached to signal lines. You can access most signals inside the model hierarchy, including referenced models and Stateflow charts. You cannot access optimized signals.
Data logging	Save data to a MATLAB variable as an array, structure, or object.	Save data to a MATLAB variable as an object.	Save data to a MATLAB variable as an object.
Simulation control	Run, forward, and back toolbar buttons.	Run, forward, and back toolbar buttons.	Run, forward, and back toolbar buttons.
Scale axes after simulation	Toolbar buttons to scale X-axis and Y-axis limits **Axes scaling** set to `Auto` for the X-axis and Y-axis.	Toolbar buttons to scale X-axis and Y-axis limits. **Axes scaling** set to `Auto` for only the Y-axis.	Toolbar buttons to scale X-axis and Y-axis limits. **Axes scaling** set to `Auto` for the X-axis and Y-axis.
Add to model	Add block from Simulink sinks library.	Add block from Simulink sinks library.	Add using "Viewers and Generators Manager" on page 28-77.
Visual indication in model	Scope block connected to signals.	Floating Scope block not attached to any signal lines.	Viewer icons located above signal lines for all attached signals.

Capability	Simulink Scope	Simulink Floating Scope	Simulink Scope Viewer
Manage scopes centrally	No.	No.	Use the Viewers and Generators Manager to add or delete viewers, and attach or remove signals.
Manage scopes locally	Attach signal lines to Scope block in ports.	Attach signals from the Floating Scope window.	Add viewers and attach additional signals within a model hierarchy using the context menus or from the Scope viewer window.
Simulink Report Generator support	Yes.	Yes.	No.
Connecting Constant block with Sample time set to `inf` (constant sample time)	Plots the data value at the first time step and anytime you tune a parameter.	Plots all data values.	Plots the data value at the first time step and anytime you tune a parameter.

Simulink Scope Versus DSP System Toolbox Time Scope

If you have a Simulink and a DSP System Toolbox license, you can use either the Simulink Scope or DSP System Toolbox Time Scope. Choose the scope based on your application requirements, how the blocks work, and the default values of each block.

If you have a DSP System Toolbox license and you have been using Time Scopes, continue to do so in your applications. Using the Time Scope block requires a DSP System Toolbox license.

Feature	Scope	Time Scope
Location in block library	Simulink Sinks library	DSP System Toolbox Sinks library
Trigger and measurement panels	With Simulink only: • Trigger • Cursor Measurement With DSP System Toolbox or Simscape license: • Signal Statistics • Bilevel Measurements • Peak Finder	• Trigger • Cursor Measurements • Signal Statistics • Bilevel Measurements • Peak Finder

Feature	Scope	Time Scope
Simulation mode support for block-based sample times For block-based sample times, all the inputs of the block run at the same rate. For rapid-accelerator mode, see “Behavior of Scopes and Viewers with Rapid Accelerator Mode” on page 35-16.	• Normal • Accelerator • Rapid-Accelerator • External	• Rapid-Accelerator • External
Simulation mode support for port-based sample times For port-based sample times, the input ports can run at different rates.	No.	• Normal • Accelerator
Frame processing of signals	Included in Scope block with DSP System Toolbox license.	Included in Time Scope block.
Sample time propagation	If the different ports have different sample rates, the scope uses the greatest common divisor of the rates.	When using port-based sample times, the different ports of the Scope block inherit the different rates and plots the signals according to those rates.
Save model to previous Simulink release	If saving to a release before R2015a, the Scope block is converted to a scope with the features available in that release.	No change in features.

This table lists the differences in Configuration Property default values between the Scope and Time Scope blocks.

Property	Scope Default	Time Scope Default
Open at start of simulation	Cleared	Selected
Input processing	Elements as channels (sample based)	Columns as channels (frame based)
Maximize Axes	Off	Auto
Time Units	None	Metric (based on Time Span)
Time-axis labels	Bottom displays only	All
Show time-axis label	Cleared	Selected
Plot Type	Auto	Line
Title	%<Signal Label>	No title
Y label	No label	Amplitude

See Also

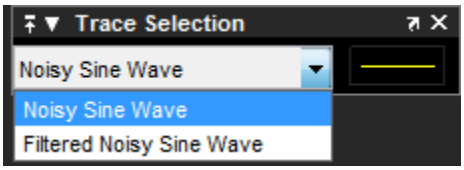
Floating Scope | Scope | Scope Viewer | Time Scope

Related Examples

- “Common Scope Block Tasks” on page 28-51
- “Display Time-Domain Data” (DSP System Toolbox)
- “Configure Time Scope Block” (DSP System Toolbox)
- “Floating Scope and Scope Viewer Tasks” on page 28-67

Scope Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears. Use this panel to select which signal to measure. To open the Trace Selection panel:

- From the menu, select **Tools > Measurements > Trace Selection**.
- Open a measurement panel.
- 

See Also

Floating Scope | Scope

Related Examples

- “Scope Triggers Panel” on page 28-12

Scope Triggers Panel

In this section...

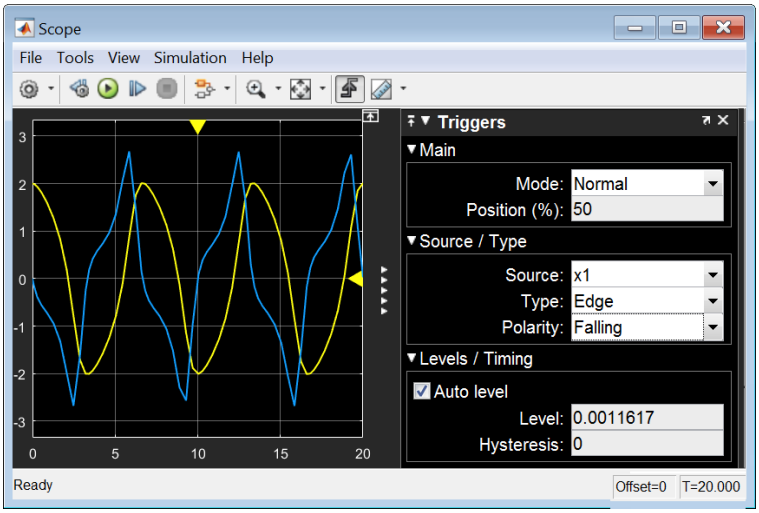
What Is the Trigger Panel

The Trigger panel defines a trigger event to synchronize simulation time with input signals. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.

To open the Trigger panel:

1. Open a Scope block window.
2. On the toolbar, click the Triggers button  .
3. Run a simulation.

   Triangle trigger pointers indicate the trigger time and trigger level of an event. The marker color corresponds to the color of the source signal.



Main Pane

Mode — Specify when the display updates.

- **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.

  Normal — Display data from the last trigger event. If no event occurs, the display remains blank.

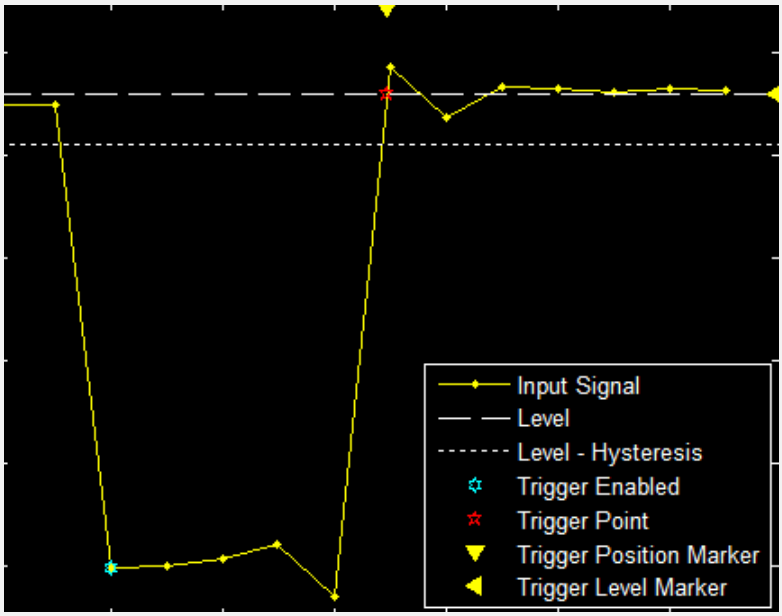
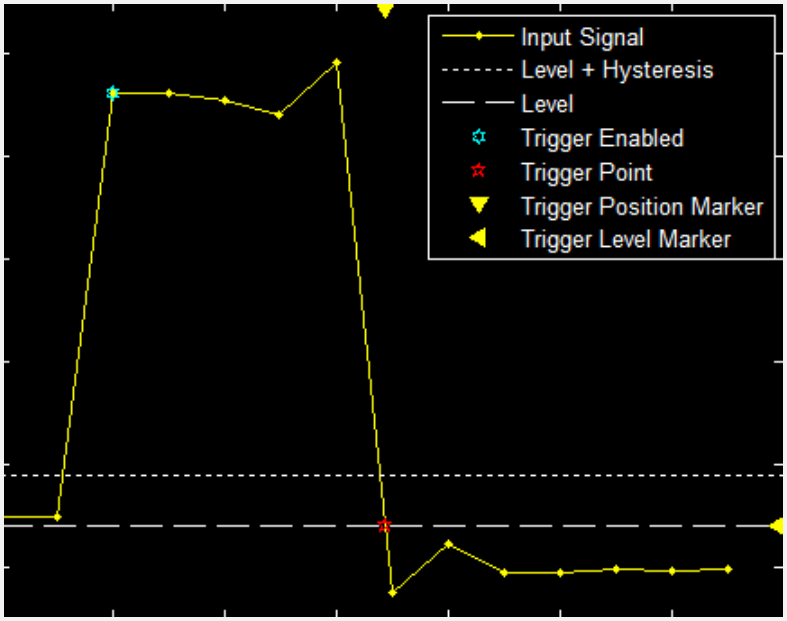
- **Once** — Display data from the last trigger event and freeze the display. If no event occurs, the display remains blank. Click the **Rearm** button to look for the next trigger event.
- **Off** — Disable triggering.

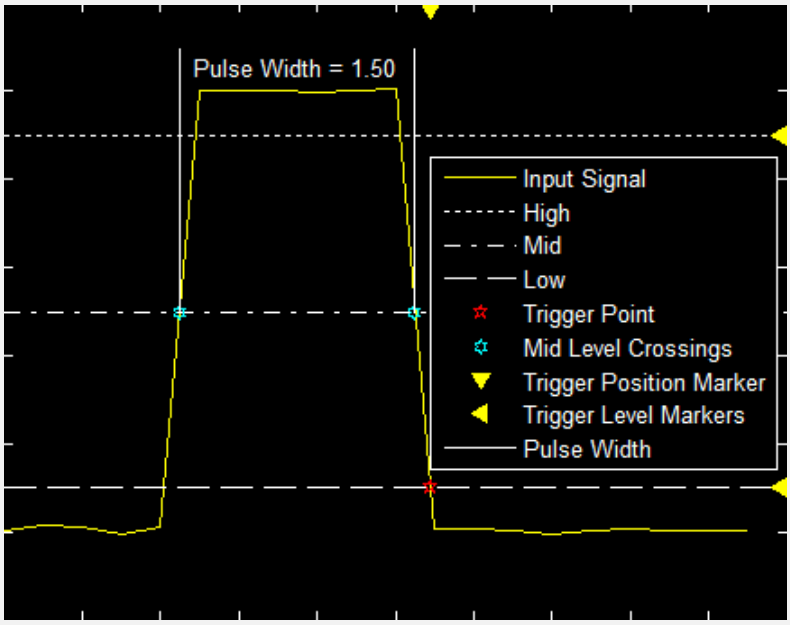
Position (%) — Specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

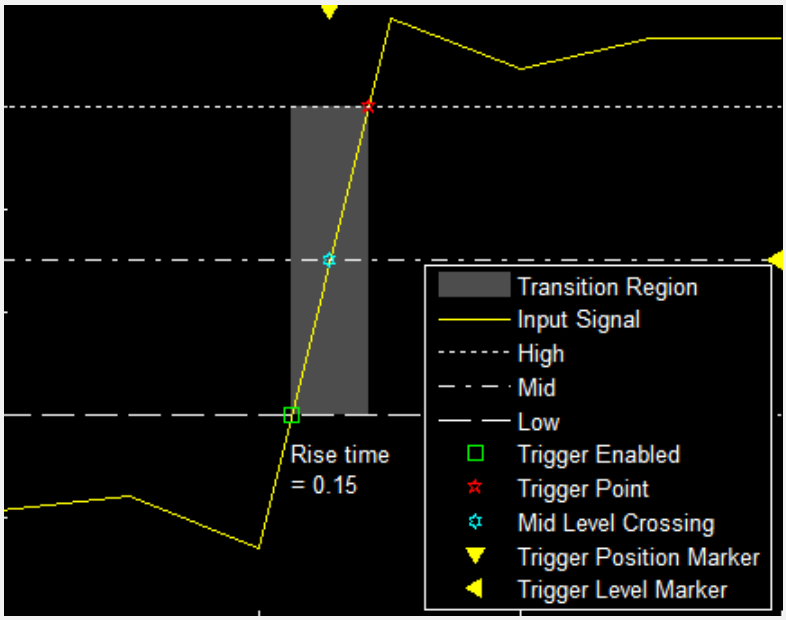
Source/Type and Levels/Timing Panes

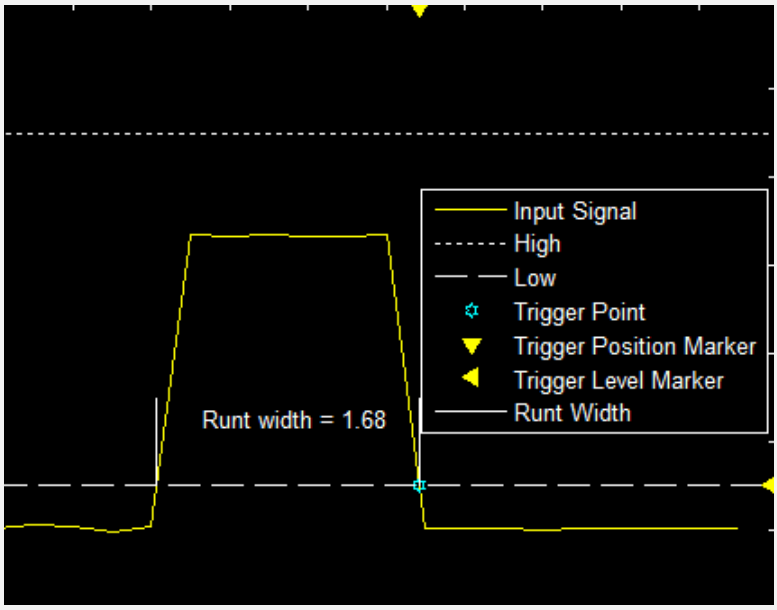
Source — Select a trigger signal. For magnitude and phase plots, select either the magnitude or the phase.

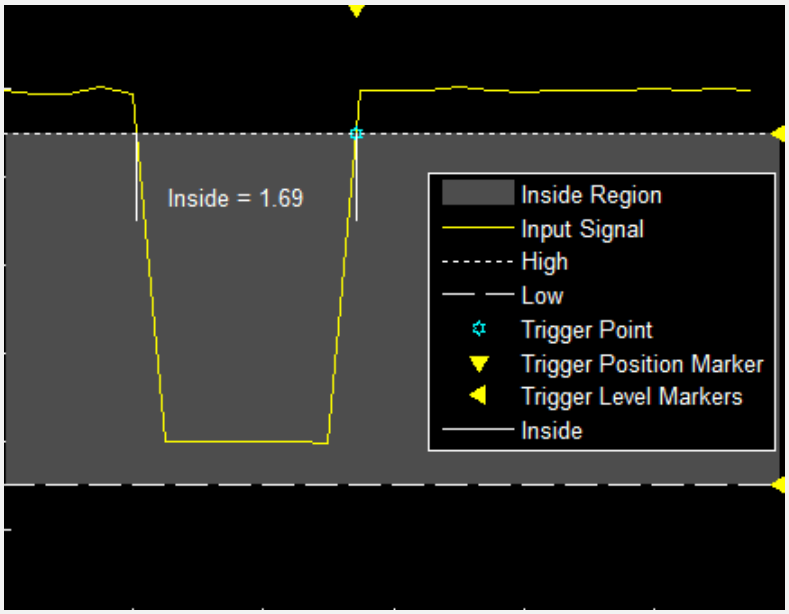
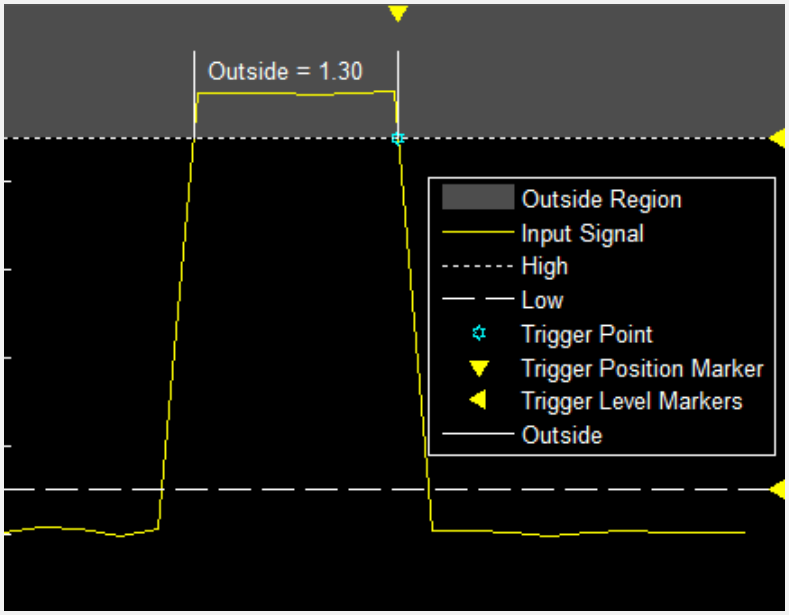
Type — Select the type of trigger.

Trigger Type	Trigger Parameters
`Edge` — Trigger when the signal crosses a threshold.	**Polarity** — Select the polarity for an edge-triggered signal. • `Rising` — Trigger when the signal is increasing.  • `Falling` — Trigger when the signal value is decreasing.  • `Either` — Trigger when the signal is increasing or decreasing. **Level** — Enter a threshold value for an edge triggered signal. **Auto level** is 50%

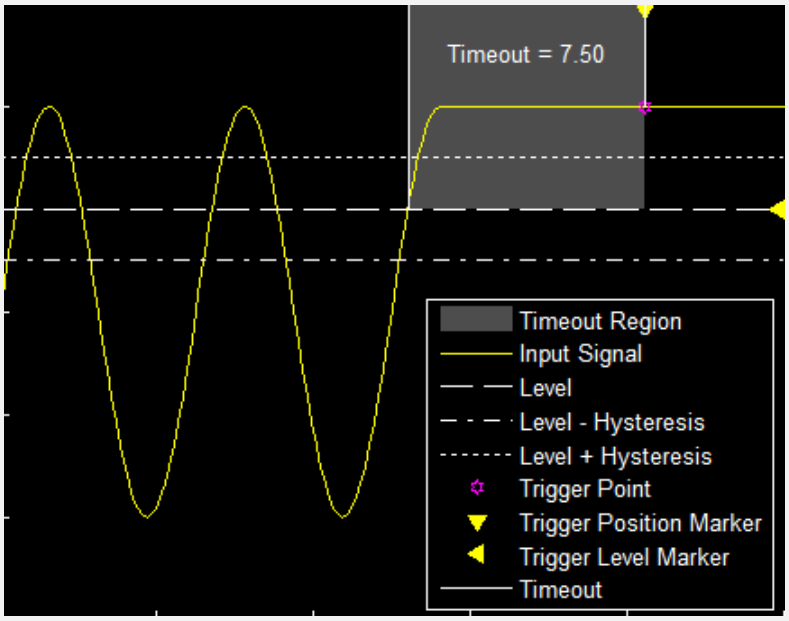
Trigger Type	Trigger Parameters
	Hysteresis — Enter a value for an edge-triggered signal. See “Hysteresis of Trigger Signals” on page 28-21
`Pulse Width` — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.	**Polarity** — Select the polarity for a pulse width-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the pulse crosses the low threshold for a second time.  • `Negative` — Trigger on a negative-polarity pulse when the pulse crosses the high threshold for a second time. • `Either` — Trigger on both positive-polarity and negative-polarity pulses. **Note** A glitch-trigger is a special type of a pulse width-trigger. A glitch-Trigger occurs for a pulse or spike whose duration is less than a specified amount. You can implement a glitch trigger by using a pulse width-trigger and setting the **Max Width** parameter to a small value. **High** — Enter a high value for a pulse width-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a pulse width-triggered signal. **Auto level** is 10%. **Min Width** — Enter the minimum pulse-width for a pulse width triggered signal. Pulse width is measured between the first and second crossings of the middle threshold. **Max Width** — Enter the maximum pulse width for a pulse width triggered signal.

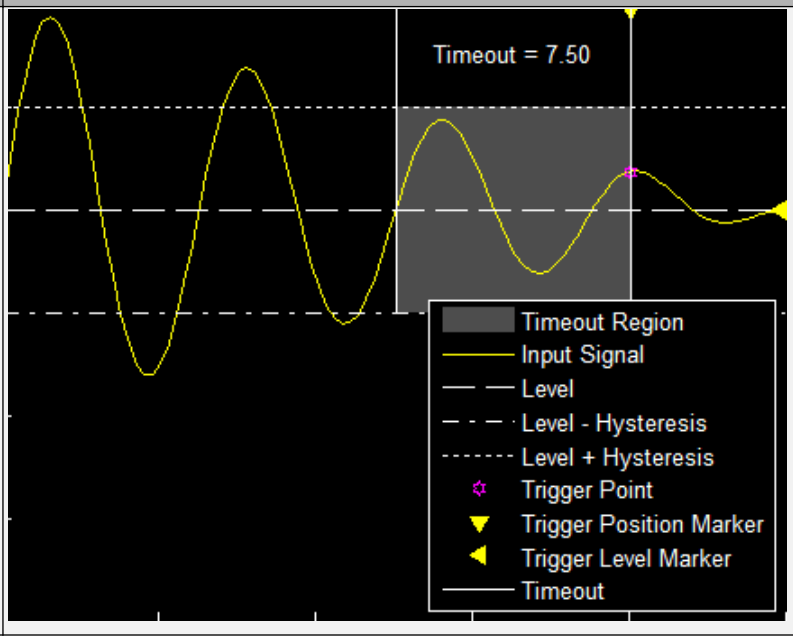
Trigger Type	Trigger Parameters
`Transition` — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.	**Polarity** — Select the polarity for a transition-triggered signal. • `Rise Time` — Trigger on an increasing signal when the signal crosses the high threshold.  • `Fall Time` — Trigger on a decreasing signal when the signal crosses the low threshold. • `Either` — Trigger on an increasing or decreasing signal. **High** — Enter a high value for a transition-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a transition-triggered signal. **Auto level** is 10%. **Min Time** — Enter a minimum time duration for a transition-triggered signal. **Max Time** — Enter a maximum time duration for a transition-triggered signal.

Trigger Type	Trigger Parameters
`Runt`— Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.	**Polarity** — Select the polarity for a runt-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the signal crosses the low threshold a second time, without crossing the high threshold.  • `Negative` — Trigger on a negative-polarity pulse. • `Either` — Trigger on both positive-polarity and negative-polarity pulses. **High** — Enter a high value for a runt-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a runt-triggered signal. **Auto level** is 10%. **Min Width** — Enter a minimum width for a runt-triggered signal. Pulse width is measured between the first and second crossing of a threshold. **Max Width** — Enter a maximum pulse width for a runt-triggered signal.

Trigger Type	Trigger Parameters
`Window` — Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.	**Polarity** — Select the region for a window-triggered signal. • `Inside` — Trigger when a signal leaves a region between the low and high levels.  • `Outside` — Trigger when a signal enters a region between the low and high levels.  • `Either` — Trigger when a signal leaves or enters a region between the low and high levels.

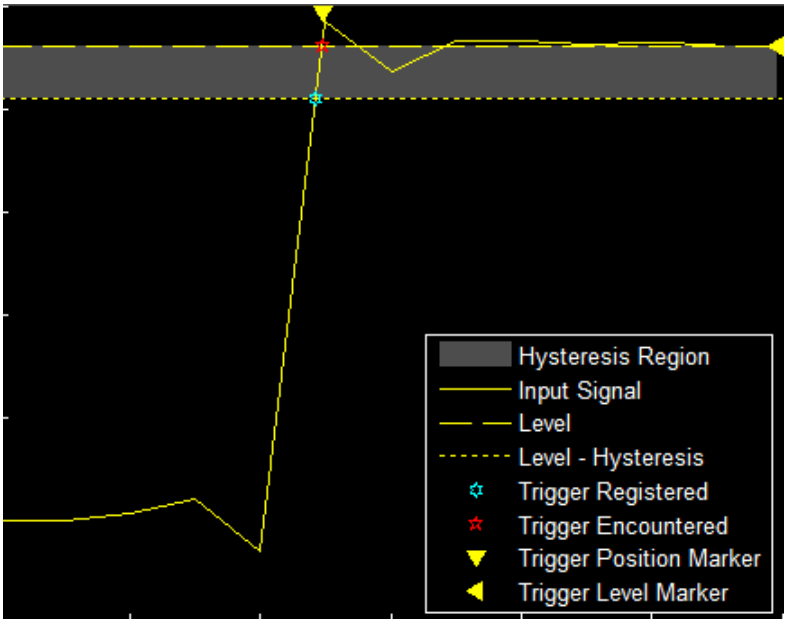
Trigger Type	Trigger Parameters
	High — Enter a high value for a window-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a window-trigger signal. **Auto level** is 10%. **Min Time** — Enter the minimum time duration for a window-triggered signal. **Max Time** — Enter the maximum time duration for a window-triggered signal.

Trigger Type	Trigger Parameters
`Timeout` — Trigger when a signal stays above or below a threshold longer than a specified time	**Polarity** — Select the polarity for a timeout-triggered signal. • `Rising` — Trigger when the signal does not cross the threshold from below. For example, if you set **Timeout** to 7.50 seconds, the scope triggers 7.50 seconds after the signal crosses the threshold.  • `Falling` — Trigger when the signal does not cross the threshold from above. • `Either` — Trigger when the signal does not cross the threshold from either direction **Level** — Enter a threshold value for a timeout-triggered signal. **Hysteresis** — Enter a value for a timeout-triggered signal. See “Hysteresis of Trigger Signals” on page 28-21. **Timeout** — Enter a time duration for a timeout-triggered signal. Alternatively, a trigger event can occur when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the threshold.

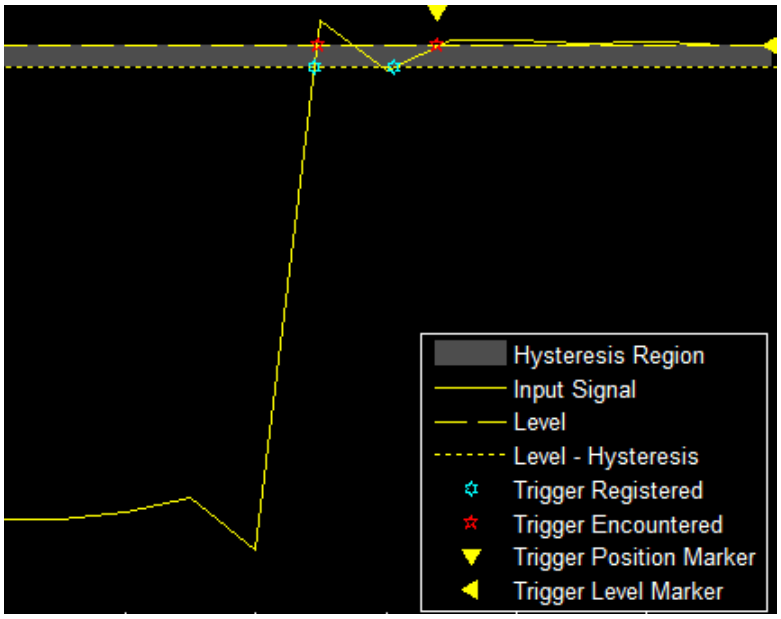
Trigger Type	Trigger Parameters
	

Hysteresis of Trigger Signals

Hysteresis (V) — Specify the hysteresis or noise reject value. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores the times that a signal crosses the trigger level within the hysteresis region.



You can reduce the hysteresis region size by decreasing the hysteresis value. In this example, if you set the hysteresis value to 0.07, the scope also considers the second rising edge to be a trigger event.



Delay/Holdoff Pane

Offset the trigger position by a fixed delay, or set the minimum possible time between trigger events.

- **Delay (s)** — Specify the fixed delay time by which to offset the trigger position. This parameter controls the amount of time the scope waits after a trigger event occurs before displaying a signal.
- **Holdoff (s)** — Specify the minimum possible time between trigger events. This amount of time is used to suppress data acquisition after a valid trigger event has occurred. A trigger holdoff prevents repeated occurrences of a trigger from occurring during the relevant portion of a burst.

See Also

Floating Scope | Scope

Related Examples

- “Cursor Measurements Panel” on page 28-23

Cursor Measurements Panel

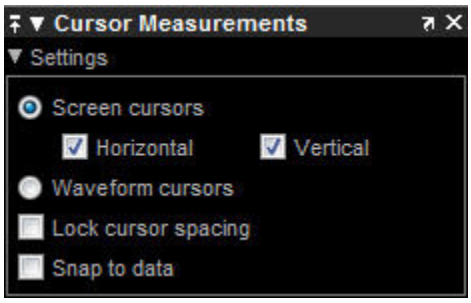
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

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.

Display screen cursors with signal times and values. To open the Cursor measurements panel:

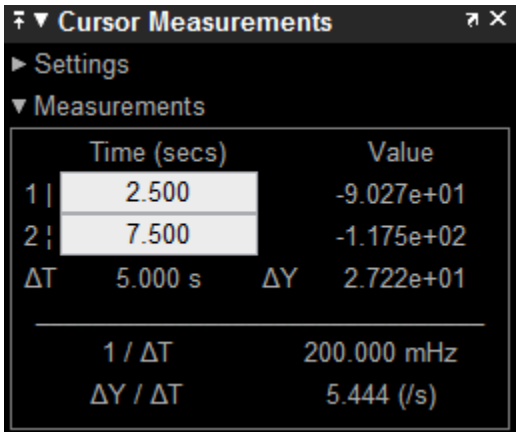
- From the menu, select **Tools > Measurements > Cursor Measurements**.
- On the toolbar, click the Cursor Measurements button.

In the **Settings** pane, you can 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 spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

The Measurements pane displays time and value measurements.



- **1** — View or modify the time or value at cursor number one (solid line cursor).
- **2** — View or modify the time or value at cursor number two (dashed line cursor).
- **ΔT** or **ΔX** — Shows the absolute value of the time (*x*-axis) difference between cursor number one and cursor number two.
- **ΔY** — Shows the absolute value of the signal amplitude difference between cursor number one and cursor number two.
- **1/ΔT** or **1/ΔX** — Shows the rate. The reciprocal of the absolute value of the difference in the times (*x*-axis) between cursor number one and cursor number two.
- **ΔY/ΔT** or **ΔY/ΔX** — Shows the slope. The ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times (*x*-axis) between cursors.

See Also

Floating Scope | Scope

Related Examples

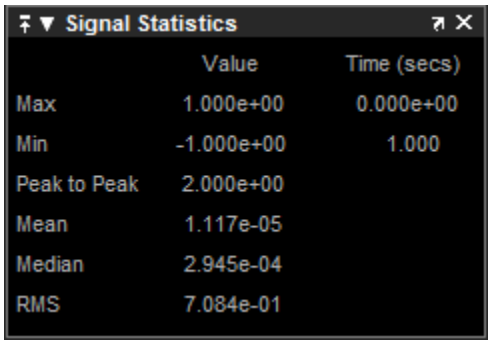
- “Scope Triggers Panel” on page 28-12

Scope Signal Statistics Panel

Note The Signal Statistics panel requires a DSP System Toolbox or Simscape license.

Display signal statistics for the signal selected in the **Trace Selection** panel. To open the Signal Statistics panel:

- From the menu, select **Tools** > **Measurements** > **Signal Statistics**.
- On the toolbar, click the Signal Statistics button.



The statistics shown are:

- **Max** — Maximum or largest value within the displayed portion of the input signal.
- **Min** — Minimum or smallest value within the displayed portion of the input signal.
- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean squared of the input signal.

When you use the zoom options in the scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

See Also

Floating Scope | Scope

Related Examples

- “Scope Triggers Panel” on page 28-12

Scope Bilevel Measurements Panel

In this section...

Bilevel Measurements

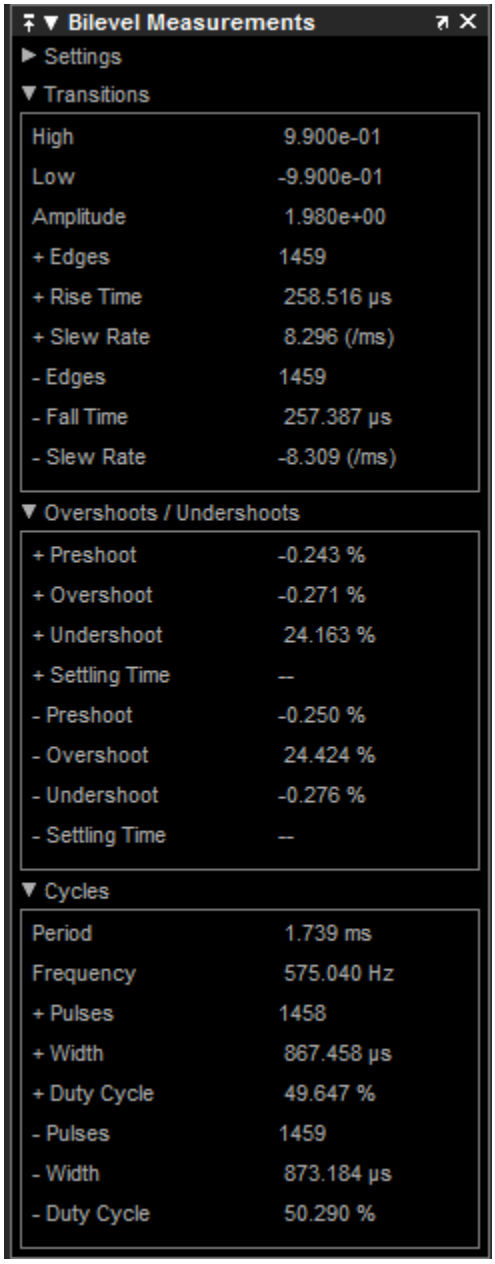
Note The Bilevel Measurements panel requires a DSP System Toolbox or Simscape license.

Display information about signal transitions, overshoots, undershoots, and cycles. To open the Bilevel Measurements panel:

- From the menu, select **Tools > Measurements > Bilevel Measurements**.
- On the toolbar, click the Bilevel Measurements  button.

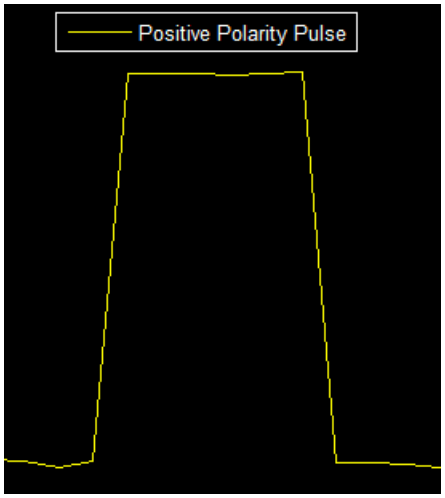
Settings

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level.

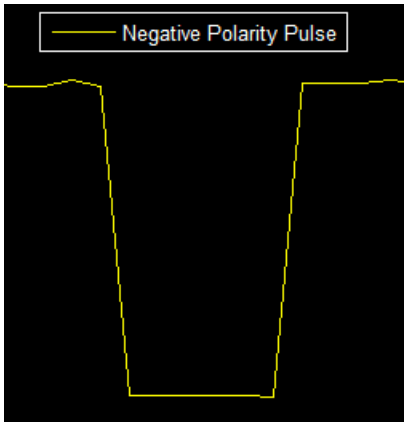


- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low- state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low- state levels manually.

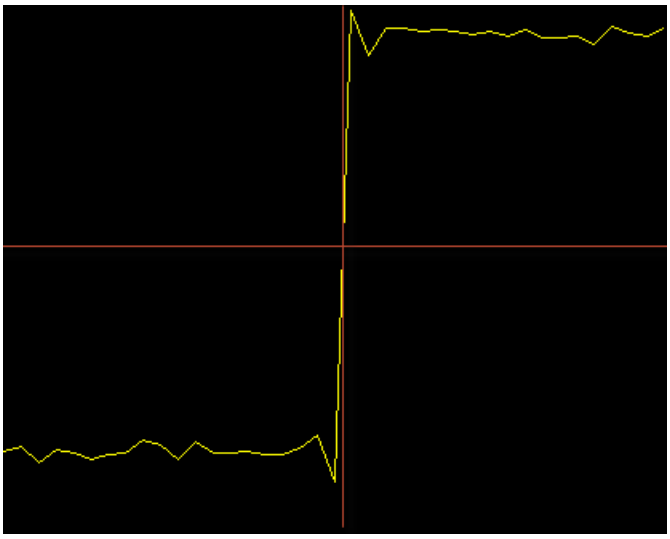
  - **High** — Used to specify manually the value that denotes a positive polarity, or high-state level.



- **Low** — Used to specify manually the value that denotes a negative polarity, or low-state level.



- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

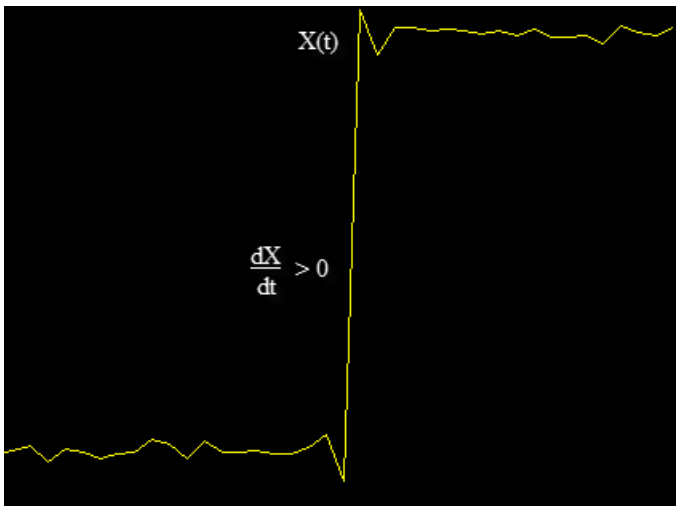


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

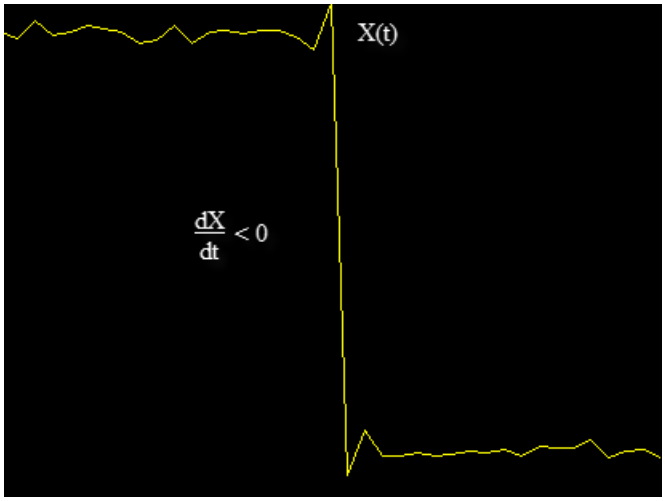
Display calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or rising edge, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.



When there is a plus sign (+) next to a text label, the measurement is a rising edge, a transition from a low-state level to a high-state level.

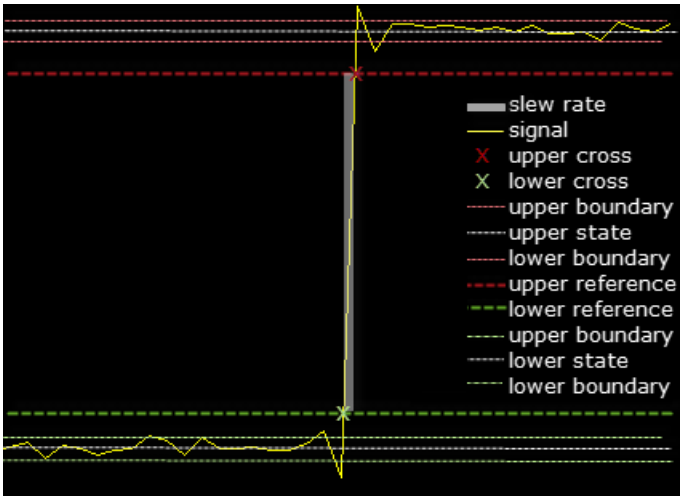
A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.



When there is a minus sign (–) next to a text label, the measurement is a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals—Time Domain Properties dialog box.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals—Time Domain Properties dialog box.
- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.

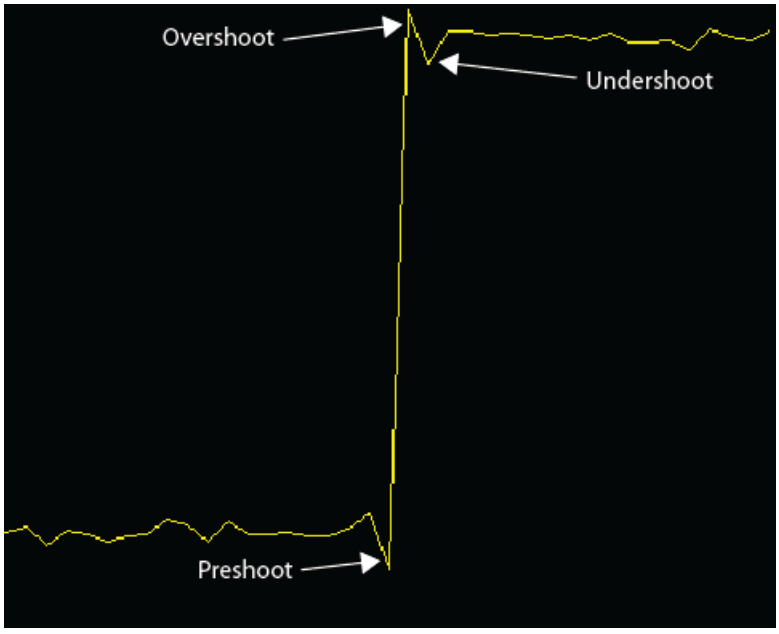


- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal.

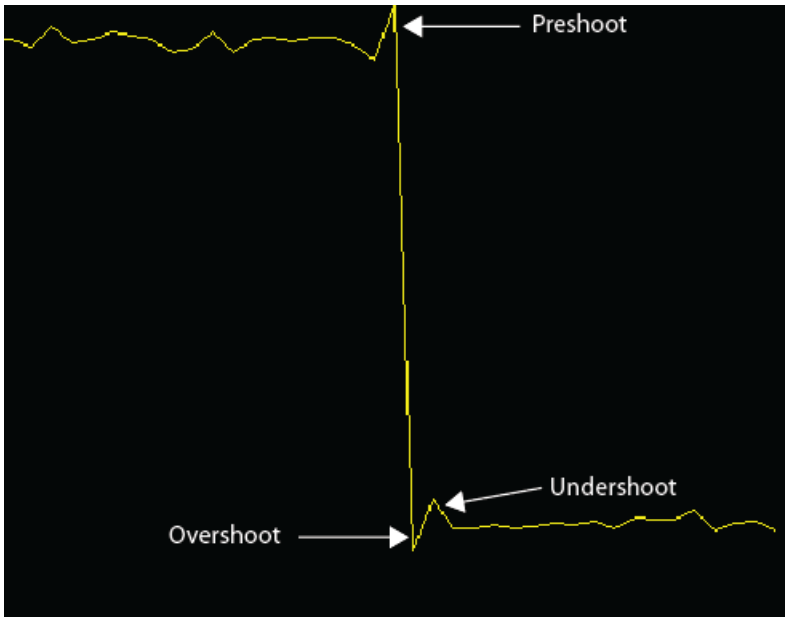
Overshoots / Undershoots Pane

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. Overshoot and undershoot refer to the amount that a signal respectively exceeds and falls below its final steady-state value. Preshoot refers to the amount before a transition that a signal varies from its initial steady-state value.

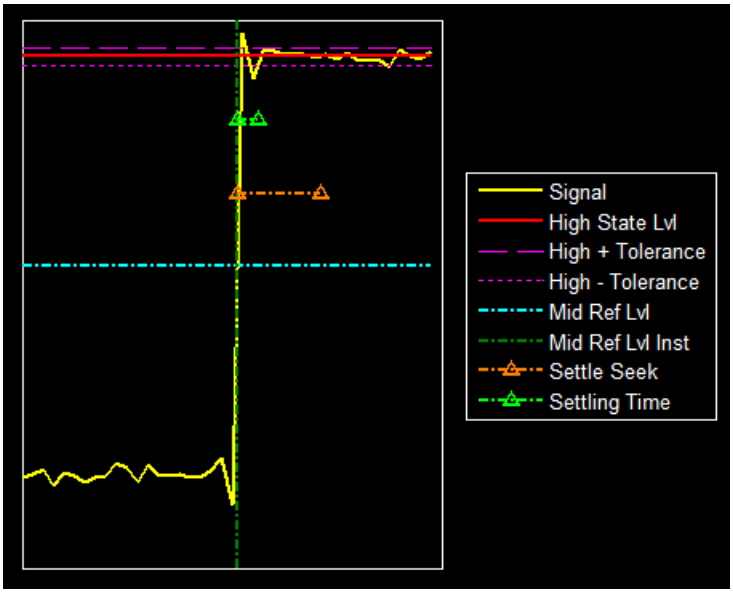
This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.



The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.
- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition.
- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition.
- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle-seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.



You can modify the settle-seek duration parameter in the **Settings** pane.

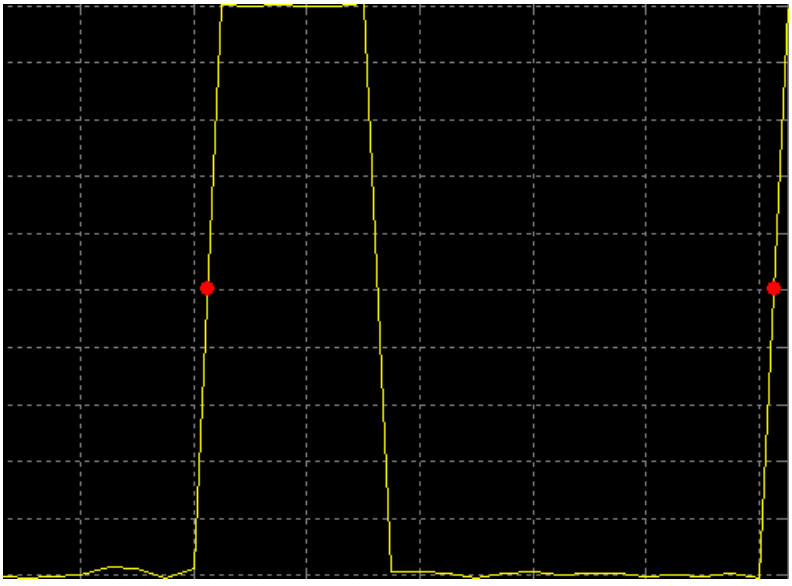
- **– Preshoot** — Average highest aberration in the region immediately preceding each falling transition.
- **– Overshoot** — Average highest aberration in the region immediately following each falling transition.
- **– Undershoot** — Average lowest aberration in the region immediately following each falling transition.
- **– Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle-seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can modify the settle-seek duration parameter in the **Settings** pane.

Cycles Pane

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

Properties to set:

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.



- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- **+ Pulses** — Number of positive-polarity pulses counted.
- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal.

- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal.
- **– Pulses** — Number of negative-polarity pulses counted.
- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal.
- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

See Also

Floating Scope | Scope

Related Examples

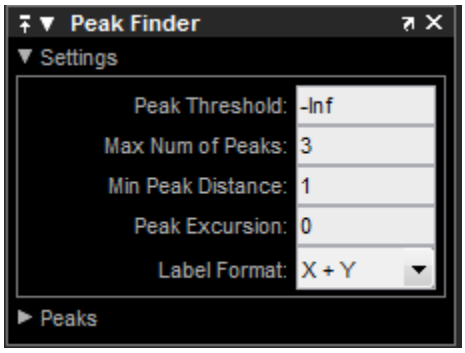
- “Scope Triggers Panel” on page 28-12

Peak Finder Measurements Panel

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 peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion.

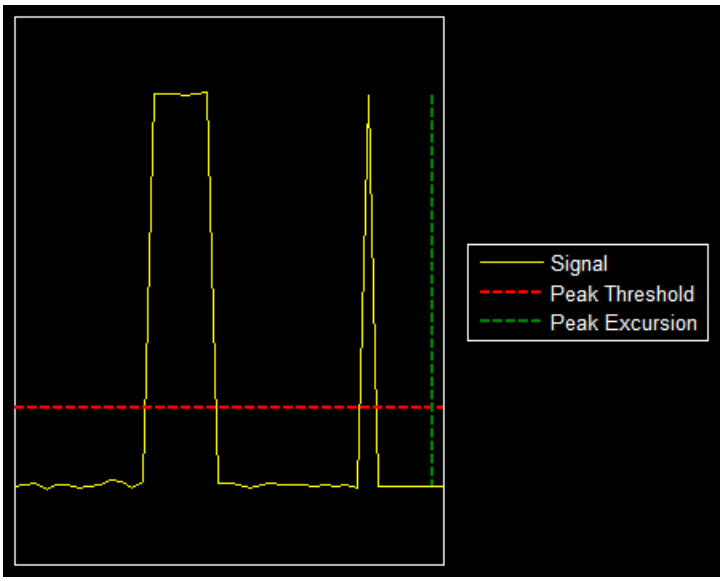
- From the menu, select **Tools > Measurements > Peak Finder**.
- On the toolbar, click the Peak Finder button.

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 `findpeaks` function reference.



Properties to set:

- **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. Peak excursion is illustrated alongside peak threshold in the following figure.



The peak threshold is a minimum value necessary for a sample value to be a peak. The peak excursion is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.

- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, you must first expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.

  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays 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 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 or hide all the peak values on the display, use the check box in the top-left corner of the **Peaks** pane.

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.

See Also

Floating Scope | Scope

Related Examples

- “Scope Triggers Panel” on page 28-12

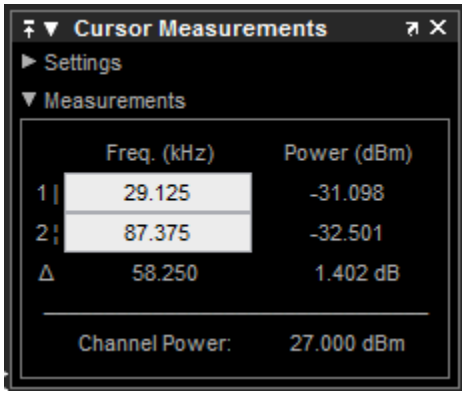
Spectrum Analyzer Cursor Measurements Panel

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.

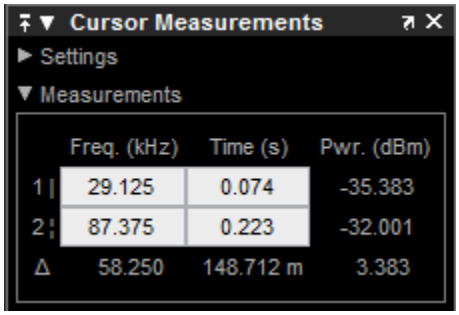
Note If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements button.

The **Cursor Measurements** panel for the spectrum and dual view:

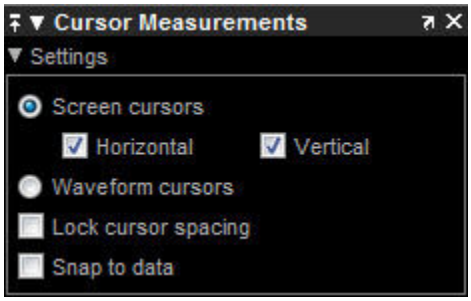


The **Cursor Measurements** panel for the spectrogram view. You must pause the spectrogram display before you can use cursors.



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.

In the **Settings** pane, you can 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 spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

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, time (for spectrograms only), or both, at cursor number one.
- **2** — Shows or enables you to modify the frequency, 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), or both, 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.

Spectrum Analyzer Channel Measurements Panel

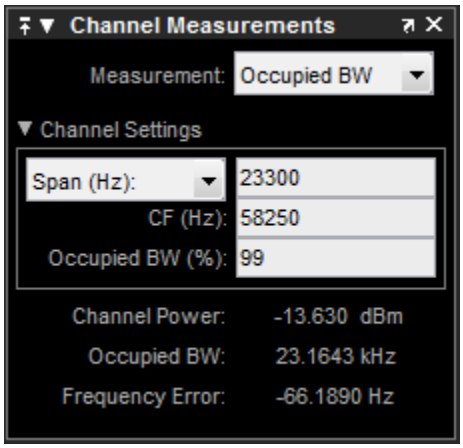
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements.

- From the menu, select **Tools** > **Measurements** > **Channel Measurements**.
- On the toolbar, click the Channel 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” (DSP System Toolbox) 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.

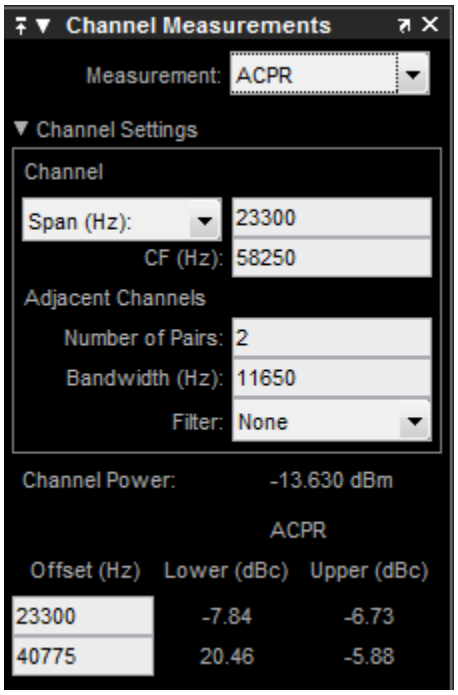


- **Channel Settings** — Modify the parameters for calculating the channel measurements.

  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.

Spectrum Analyzer Distortion Measurements Panel

The **Distortion Measurements** panel displays harmonic distortion and intermodulation distortion measurements.

- From the menu, select **Tools > Measurements > Distortion Measurements**.
- On the toolbar, click the Distortion Measurements  button.

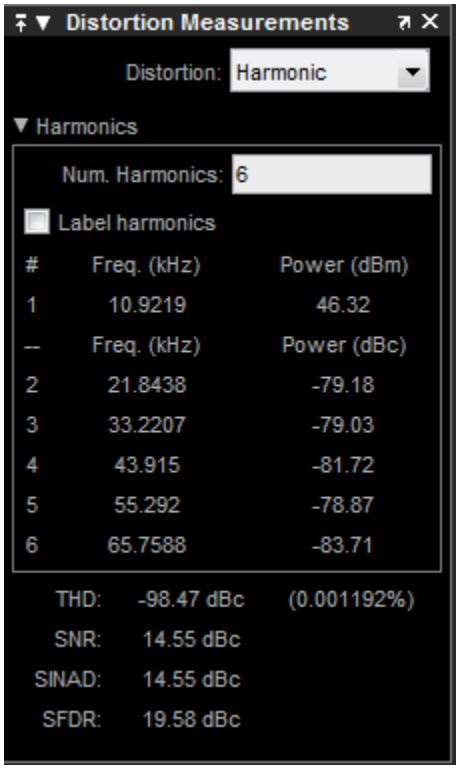
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

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 “Distortion Measurements” (DSP System Toolbox) for information on how distortion measurements are calculated.

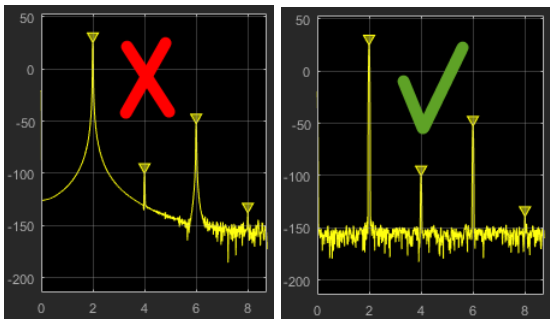
When you select `Harmonic` as the **Distortion**, the following fields appear.



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 view the best 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. Also, the noise floor should be visible.

For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



- **Num. Harmonics** — Number of harmonics to display, including the fundamental frequency. Valid values of **Num. Harmonics** are from 2 to 99. 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 1 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.

  $$THD = 10 \cdot \log_{10}(D/S)$$

- **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).

  $$SNR = 10 \cdot \log_{10}(S/N)$$

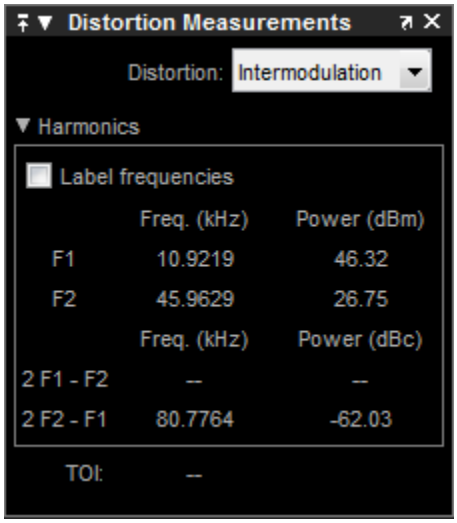
  If you see —— as the reported SNR, the total non-harmonic content of your signal is less than 30% of the total signal.
- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, S to all other content (including noise, N, and harmonic distortion, D), in decibels relative to the carrier (dBc).

  $$SINAD = 10 \cdot \log_{10}\left(\frac{S}{N+D}\right)$$

- **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.

  $$SNR = 10 \cdot \log_{10}(S/R)$$

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.

Spectral Masks

Add upper and lower masks to the Spectrum Analyzer to visualize spectrum limits and compare spectrum values to specification values.

To open the **Spectral Mask** pane, in the toolbar, select the spectral mask button,  .

Set Up Spectral Masks

In the Spectrum Analyzer window:

1. In the **Spectral Mask** pane, select a **Masks** option.
2. In the **Upper limits** or **Lower limits** box, enter the mask limits as a constant scalar, an array, or a workspace variable name.
3. (Optional) Select additional properties:
   - **Reference level** — Set a reference level for the mask. Enter a specific value or select `Spectrum peak`.
   - **Channel** — Select a channel to use for the mask reference.
   - **Frequency offset** — Set a frequency offset for mask.

From the command-line, to add a spectral mask to the `dsp.SpectrumAnalyzer` System object or the `SpectrumAnalyzerConfiguration` block configuration object:

1. Create a `SpectralMaskSpecfication` object.
2. Set properties, such as `EnabledMasks`, `LowerMask`, or `UpperMask`. For a full list of properties, see SpectralMask (block) and SpectralMask (System object™).
3. In the `dsp.SpectrumAnalyzer` or `SpectrumAnalyzerConfiguration` object, set the `SpectralMask` property equal to your `SpectralMaskSpecfication` object.

For example:

```
mask = SpectralMaskSpecification();
mask.EnabledMasks = 'Upper';
mask.UpperMask = 10;
scope = dsp.SpectrumAnalyzer();
scope.SpectralMask = mask;
scope.SpectralMask

ans =

  SpectralMaskSpecification with properties:

           EnabledMasks: 'Upper'
              UpperMask: 10
              LowerMask: -Inf
         ReferenceLevel: 'Custom'
    CustomReferenceLevel: 0
     MaskFrequencyOffset: 0

  Events for class SpectralMaskSpecification: MaskTestFailed
```

Check Spectral Masks

You can check the status of the spectral mask in several different ways:

- In the Spectrum Analyzer window, select the spectral mask button,  . In the **Spectral Mask** pane, the **Statistics** section shows statistics about how often the masks fail, which channels have caused a failure, and which masks are currently failing.
- To get the current status of the spectral masks, call `getSpectralMaskStatus`.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events”.

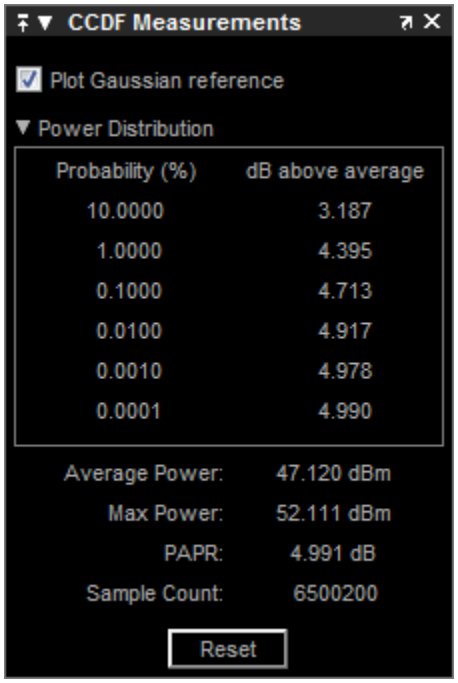
Spectrum Analyzer CCDF Measurements Panel

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.

To open this dialog box:

- From the menu, select **Tools > Measurements > CCDF Measurements**
- In the toolbar, click the CCDF Measurements button.



- **Plot Gaussian reference** — 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.

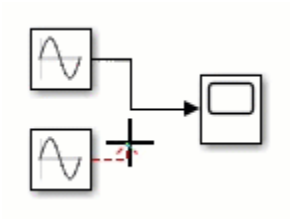
Common Scope Block Tasks

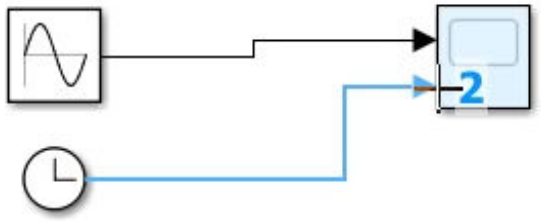
In this section...

To visualize your simulation results over time, use a Scope block or Time Scope block

Connect Multiple Signals to a Scope

To connect multiple signals to a scope, drag additional signals to the scope block. An additional port is created automatically.





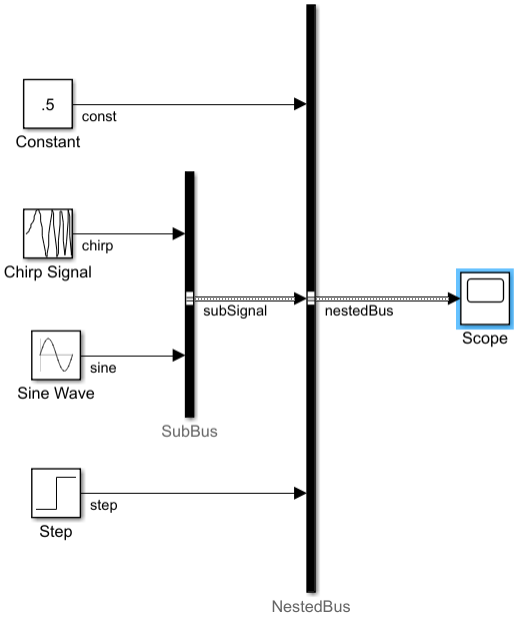
To specify the number of input ports:

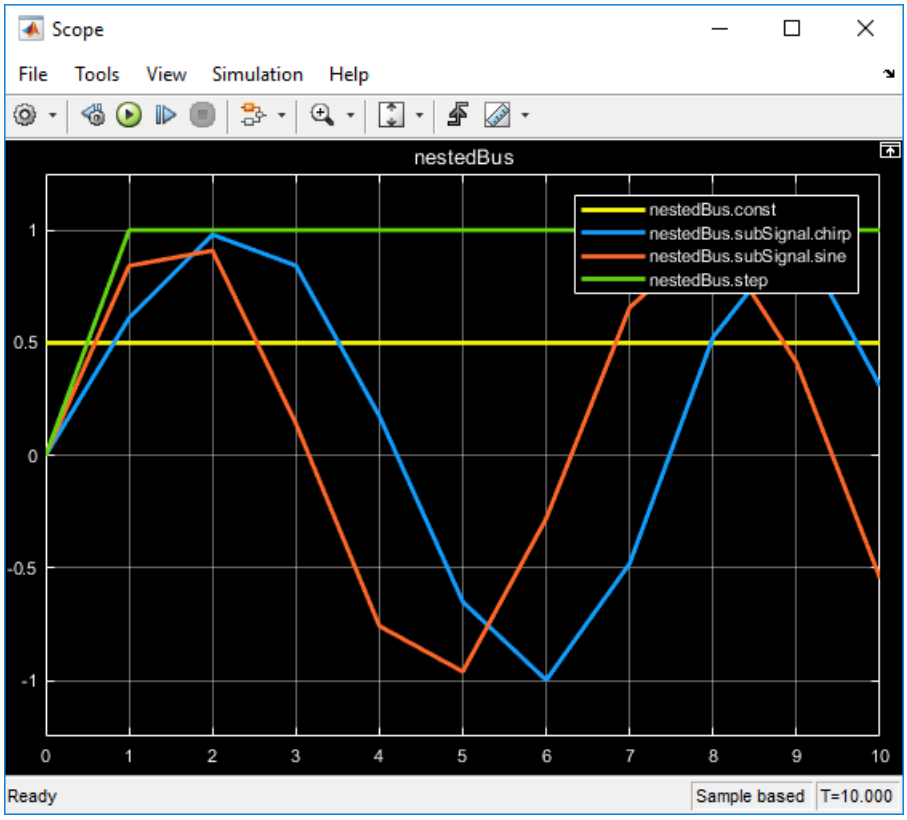
1. Open a scope window.
2. From the toolbar, select **File** > **Number of Input Ports** > **More**.
3. Enter the number of input ports, up to 96.

Signals from Nonvirtual Buses and Arrays of Buses

You can connect signals from nonvirtual buses and arrays of buses to a Scope block. To display the bus signals, use normal or accelerator simulation mode. The Scope block displays each bus element

signal, in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened. For example, in this model the `nestedBus` signal has the `const`, `subSignal`, and `step` signals as elements. The `subSignal` sub-bus has the `chirp` and `sine` signals as its bus elements. In the Scope block, the two elements of the `subSignal` bus display between the `const` and `step` signals.



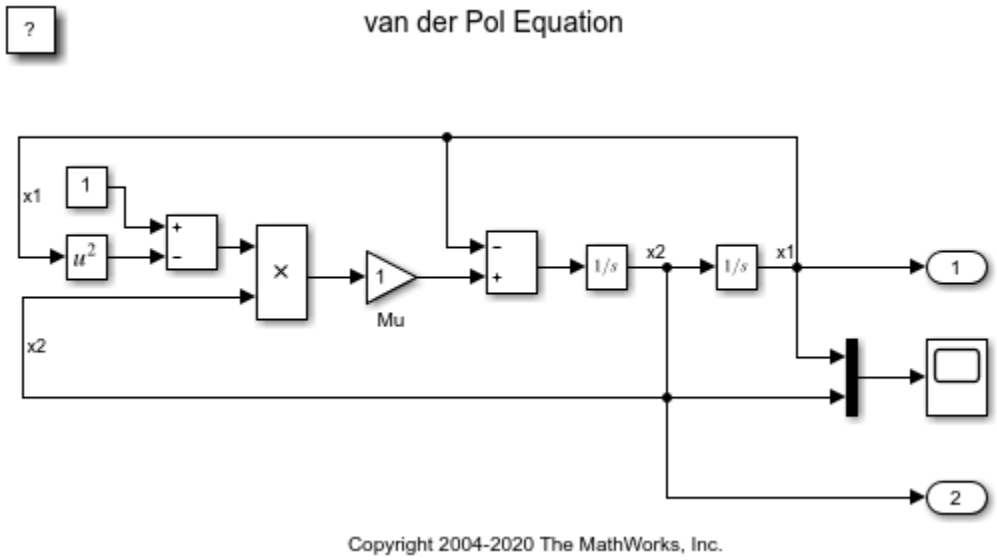


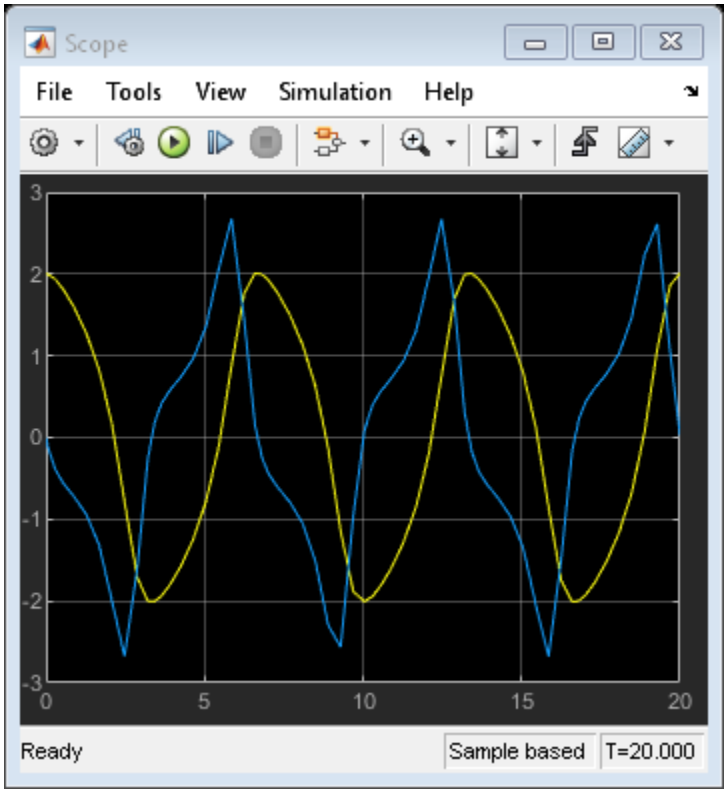
Save Simulation Data Using Scope Block

This example shows how to save signals to the MATLAB Workspace using the Scope block. You can us these steps for the Scope or Time Scope blocks. To save data from the Floating Scope or Scope viewer, see “Save Simulation Data from Floating Scope” on page 28-69.

Using the `vdp` model, turn on data logging to the workspace. You can follow the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Logging tab, turn on **Log data to workspace**.

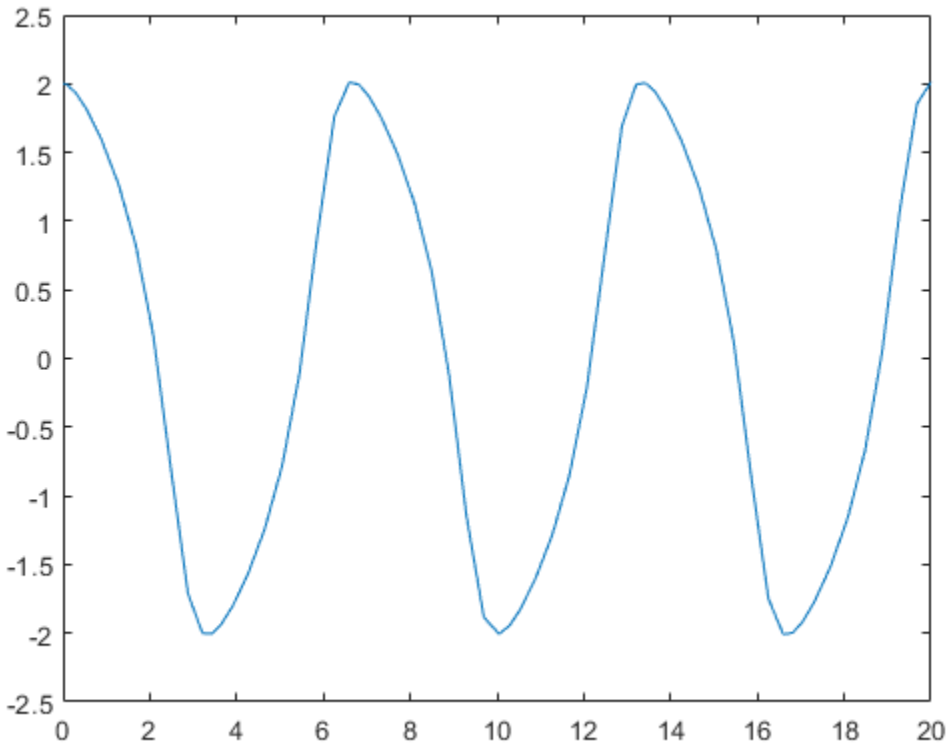
```
vdp
scopeConfig = get_param('vdp/Scope','ScopeConfiguration');
scopeConfig.DataLogging = true;
scopeConfig.DataLoggingSaveFormat = 'Dataset';
out = sim('vdp');
```





In the MATLAB Command window, view the logged data from the `out.ScopeData` structure.

```
x1_data = out.ScopeData{1}.Values.Data(:,1);
x1_time = out.ScopeData{1}.Values.Time;
plot(x1_time,x1_data)
```



Pause Display While Running

Use the Simulink Snapshot to pause the scope display while the simulation keeps running in the background.

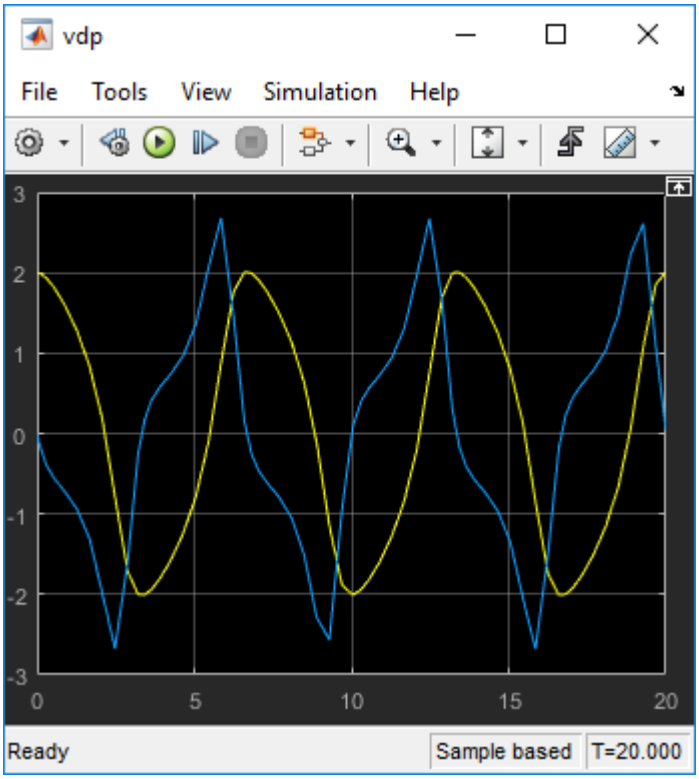
1. Open a scope window and start the simulation.
2. Select **Simulation** > **Simulink Snapshot**.

   The scope window status in the bottom left is **Frozen**, but the simulation continues to run in the background.
3. Interact with the paused display. For example, use measurements, copy the scope image, or zoom in or out.
4. To unfreeze the display, select **Simulation** > **Simulink Snapshot** again.

Copy Scope Image

This example uses the model vdp to demonstrate how to copy and paste a scope image.

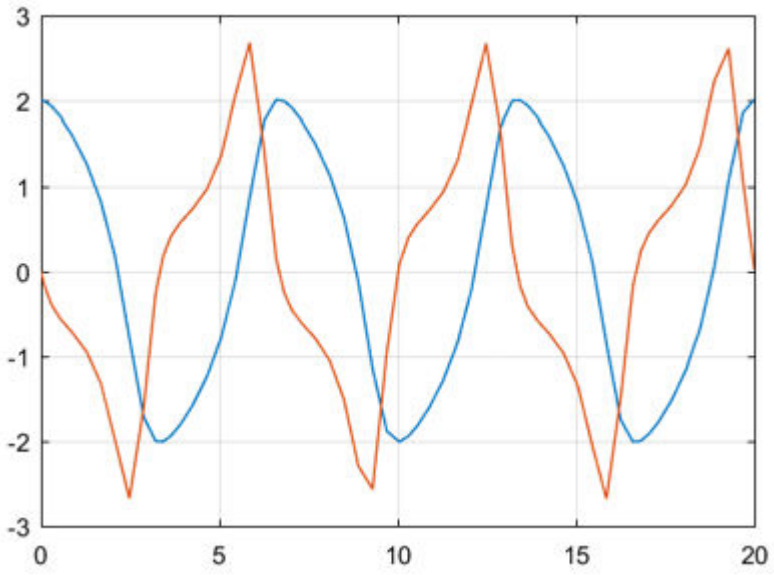
1. Add a scope block to your model.
2. Connect signals to scope ports. See “Connect Multiple Signals to a Scope” on page 28-51. For example, in the `vdp` model, connect the signals `x1` and `x2` to a scope.
3. Open the scope window and run the simulation.



4 Select **File > Copy to Clipboard**.

5 Paste the image into a document.

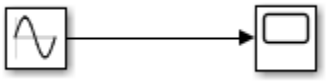
The van der Pol Equation results from my model:



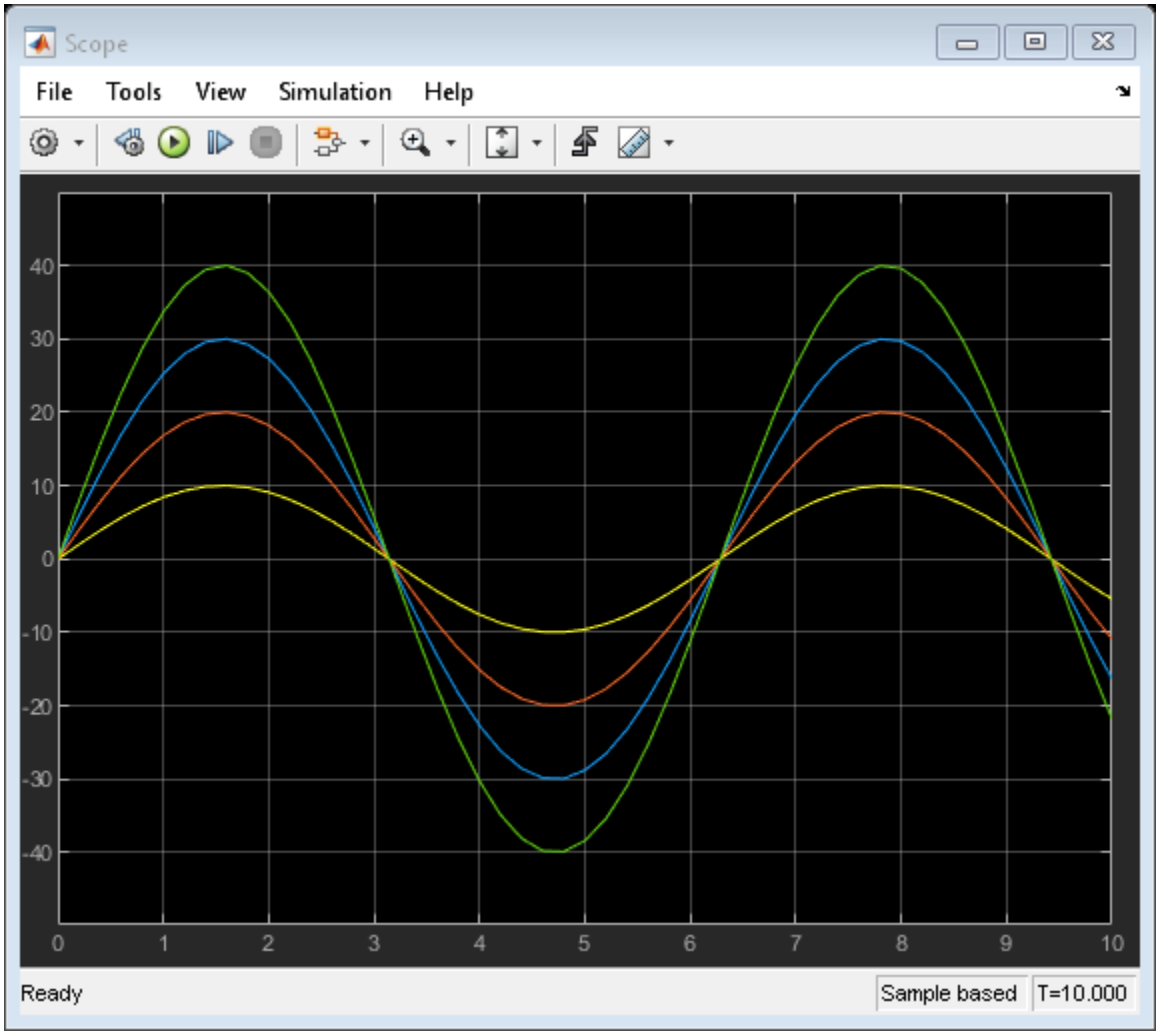
By default, **Copy to Clipboard** saves a printer-friendly version of the scope with a white background and visible lines. If you want to paste the exact scope plot displayed, select **View > Style**, then select the **Preserve colors for copy to clipboard** check box.

Plot an Array of Signals

This example shows how the scope plots an array of signals.



In this simple model, a Sine Wave block is connected to a scope block. The Sine Wave block outputs four signals with the amplitudes `[10, 20; 30 40]`. The scope displays each sine wave in the array separately in the matrix order (1,1), (2,1), (1,2), (2,2).



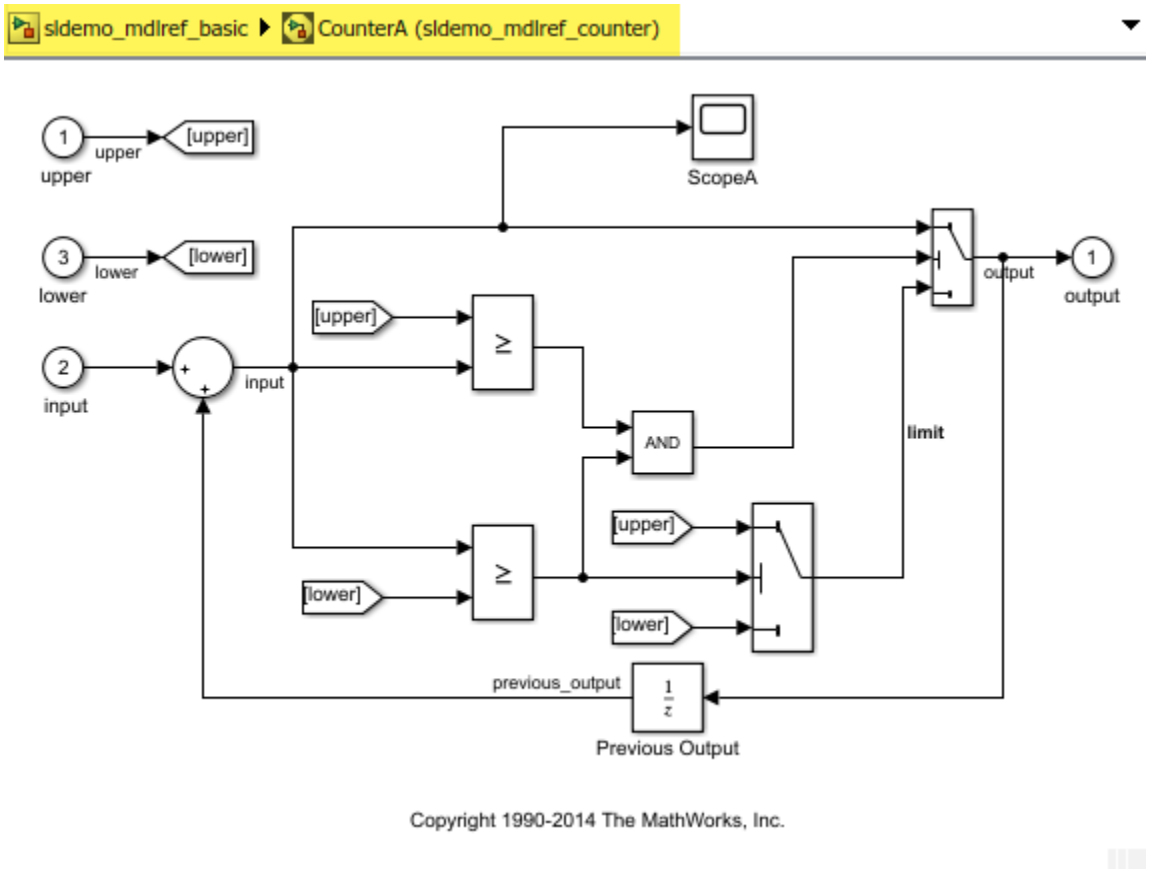
Scopes in Referenced Models

This example shows the behavior of scopes in referenced models. When you use a scope in a referenced model, you see different output in the scope depending on where you started the simulation: from the top model or the scope in the referenced model.

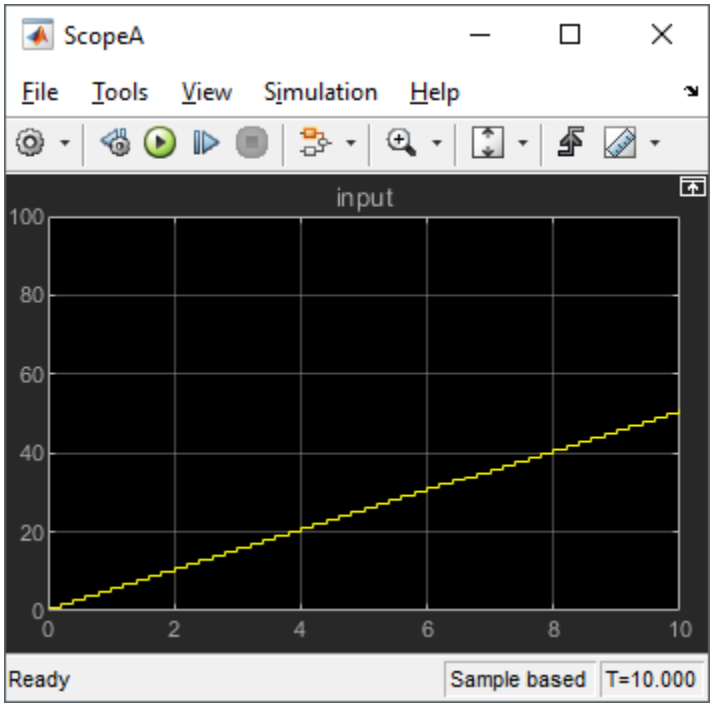
Note Scope windows display simulation results for the most recently opened top model. Playback controls in scope blocks and viewers simulate the model containing that block or viewer.

This example uses the `sldemo_mdlref_counter` model both as a top model and as a referenced model from the `sldemo_mdlref_basic` model.

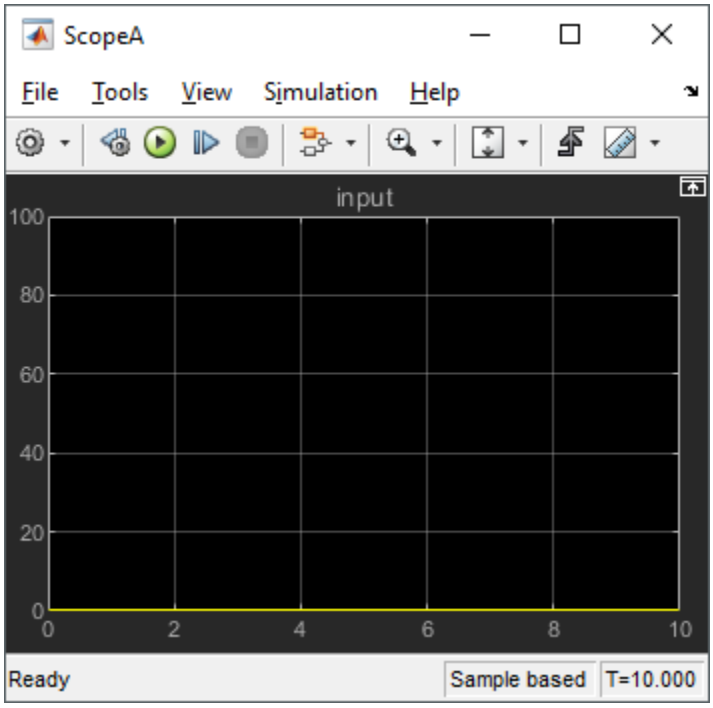
Open the `sldemo_mdlref_basic` model and double-click the CounterA block. The `sldemo_mdlref_counter` model opens as a referenced model, as evidenced by the breadcrumb above the canvas.



Run the simulation using the main run button, then open up the ScopeA scope. The scope visualizes the data from the entire model.



If you rerun the simulation using the run button in the scope, the scope only visualizes data as if the referenced model is opened in isolation. Playback controls in scope blocks and viewers simulate the model containing that block or viewer. In this case, the referenced model input, without the top model, is zero the entire time.

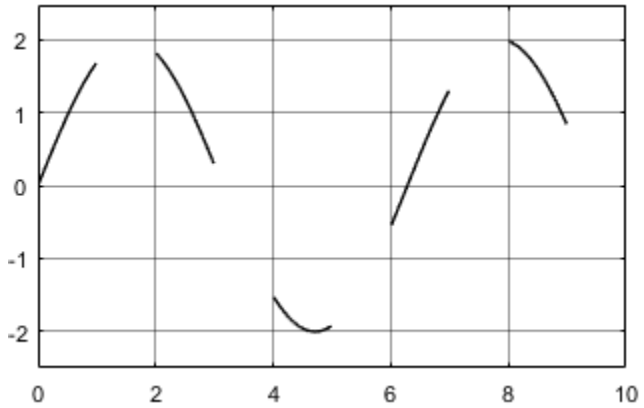


Note If you click run from the scope, the model does not show that the model is running in the background. For the simulation status, look at the status bar in the scope.

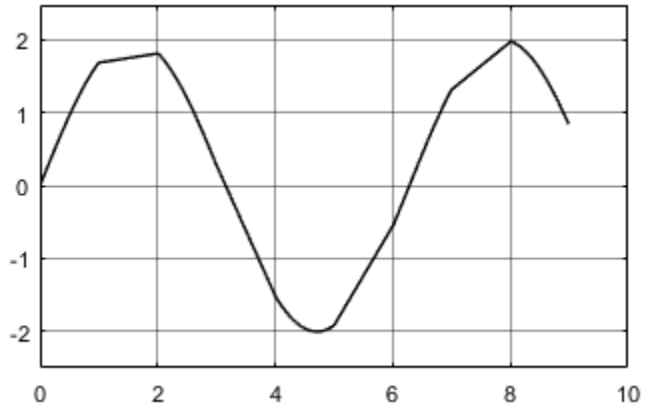
Scopes Within an Enabled Subsystem

When placed within an Enabled Subsystem block, scopes behave differently depending on the simulation mode:

- Normal mode — A scope plots data when the subsystem is enabled. The display plot shows gaps when the subsystem is disabled.



- External, Accelerator, and Rapid modes — A scope plots data when the subsystem is enabled. The display connects the gaps with straight lines.



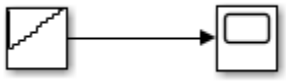
Modify x-axis of Scope

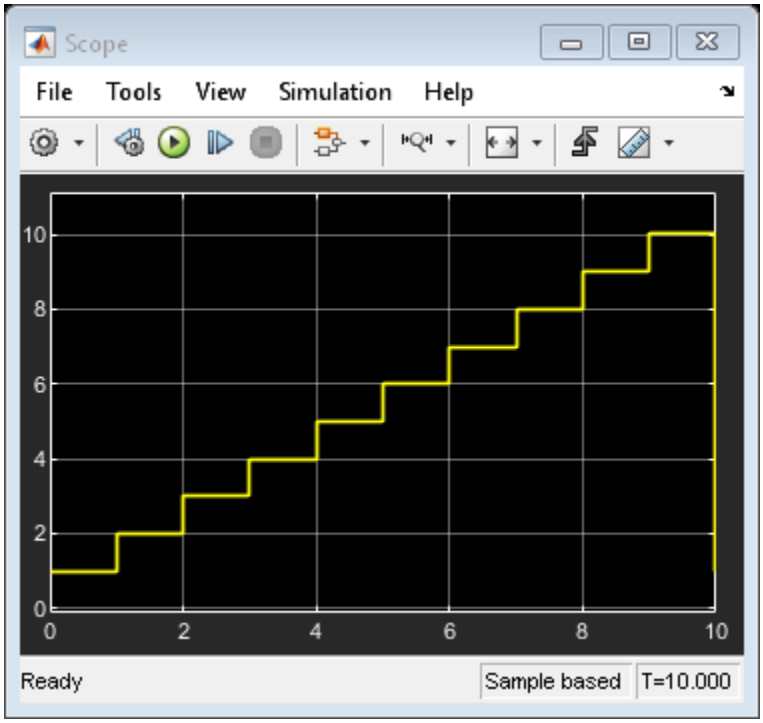
This example shows how to modify the *x*-axis values of the Scope block using the **Time span** and **Time display offset** parameters. The **Time span** parameter modifies how much of the simulation time is shown and offsets the *x*-axis labels. The **Time display offset** parameter modifies the labels used on the *x*-axis.

You can also use this procedure for the Time Scope block, Floating Scope block, or Scope viewer.

Open the model and run the simulation to see the original scope output. The simulation runs for 10 time steps stepping up by 1 at each time step.

```
model = 'ModifyScopeXAxis';
open_system(model);
sim(model);
open_system([model,'/Scope']);
```

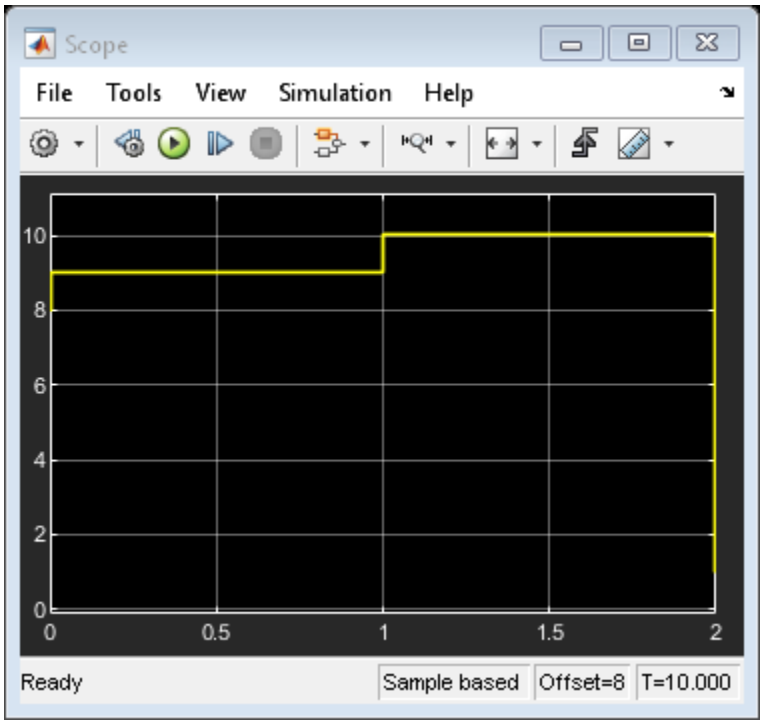




Modify Time Span Shown

Modify the **Time span** parameter to 2. You can follow the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Time tab.

```
scopeConfig = get_param([model,'/Scope'],'ScopeConfiguration');
scopeConfig.TimeSpan = '2';
sim(model);
open_system([model,'/Scope']);
```



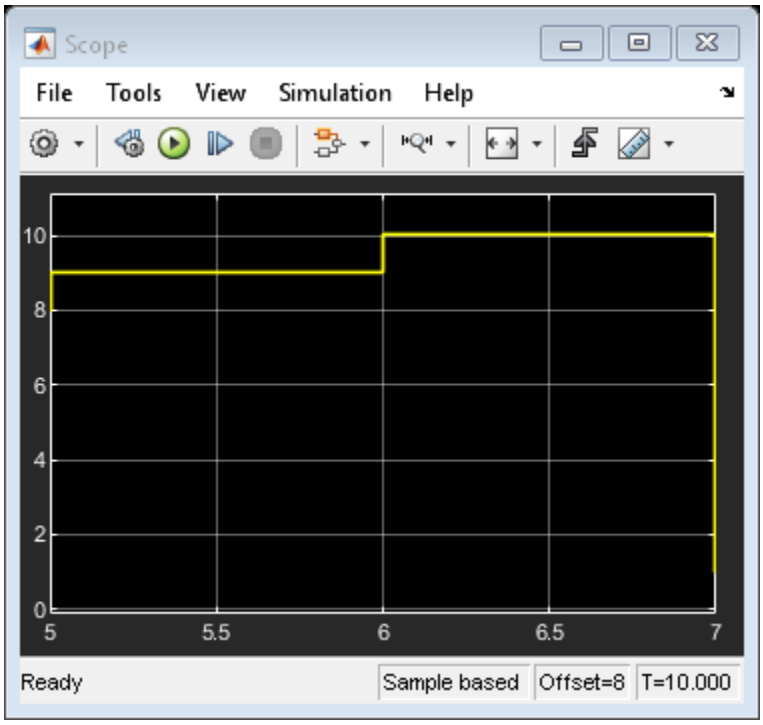
The *x*-axis of the scope now shows only the last 2 time steps and offsets the *x*-axis labels to show 0-2. The bottom toolbar shows that the *x*-axis is offset by 8. This offset is different from the **Time display offset** value.

The **Time span** parameter is useful if you do not want to visualize signal initialization or other start-up tasks at the beginning of a simulation. You can still see the full simulation time span if you click the **Span x-axis** button.

Offset *x*-axis Labels

Modify the **Time display offset** parameter to 5. Again, use the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Time tab.

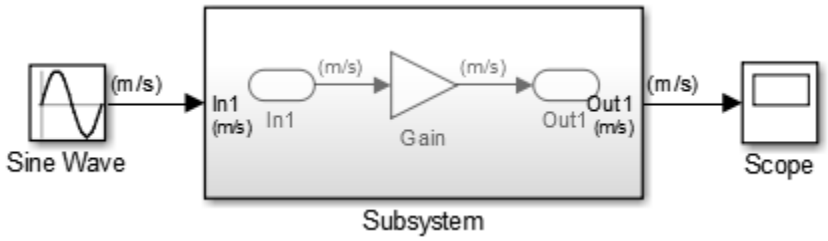
```
scopeConfig.TimeDisplayOffset = '5';
sim(model);
open_system([model,'/Scope']);
```



Now, the same time span of 2 is show in the scope, but the *x*-axis labels are offset by 5, starting at 5 and ending at 7. If you click the **Span x-axis** button, the *x*-axis labels still start at 5.

Show Signal Units on a Scope Display

You can specify signal units at a model component boundary (Subsystem and Model blocks) using Inport and Outport blocks. See “Unit Specification in Simulink Models” on page 9-2 . You can then connect a Scope block to an Outport block or a signal originating from an Outport block. In this example, the **Unit** property for the Out1 block was set to `m/s`.

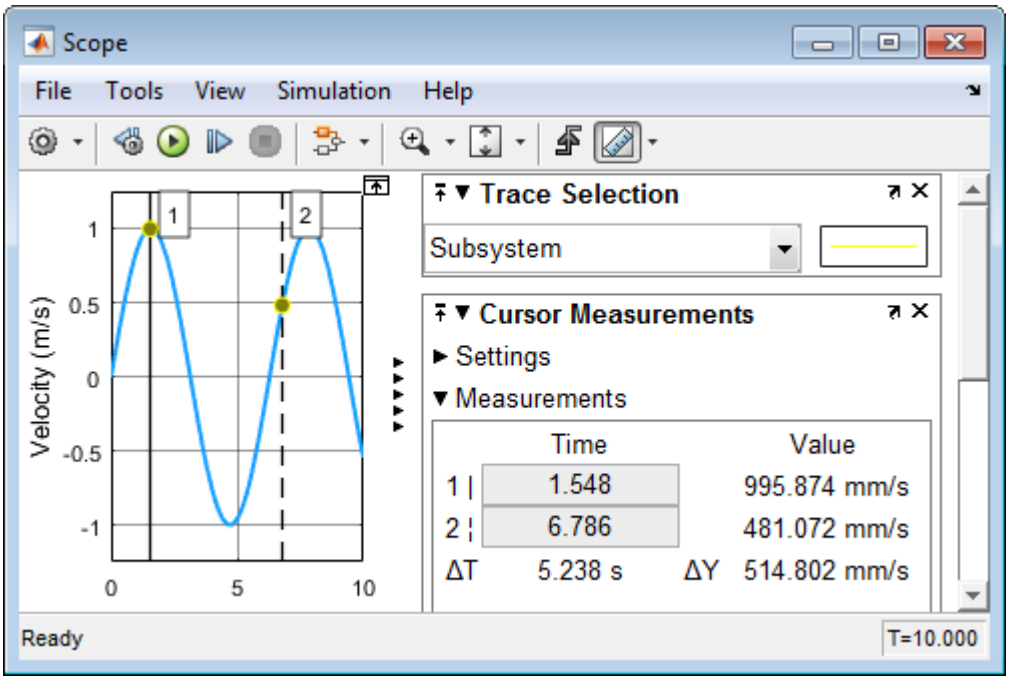


Show Units on a Scope Display

1. From the Scope window toolbar, select the Configuration Properties button .
2. In the Configuration Properties: Scope dialog box, select the **Display** tab.
3. In the **Y-label** box, enter a title for the y-axis followed by `(%<SignalUnits>)`. For example, enter

   ```
   Velocity (%<SignalUnits>)
   ```
4. Click **OK** or **Apply**.

Signal units display in the *y*-axis label as meters per second (`m/s`) and in the Cursor Measurements panel as millimeters per second (`mm/s`).



From the Simulink toolstrip, you can also select **Debug > Information Overlays > Units**. You do not have to enter `(%<SignalUnits>)` in the **Y-Label** property.

Show Units on a Scope Display Programmatically

1. Get the scope properties. In the Command Window, enter

   ```
   load_system('my_model')
   s = get_param('my_model/Scope','ScopeConfiguration');
   ```

2. Add a *y*-axis label to the first display.

   ```
   s.ActiveDisplay = 1
   s.YLabel = 'Velocity (%<SignalUnits>)';
   ```

You can also set the model parameter `ShowPortUnits` to `'on'`. All scopes in your model, with and without `(%<SignalUnits>)` in the **Y-Label** property, show units on the displays.

```
load_system('my_model')
get_param('my_model','ShowPortUnits')

ans =
off

set_param('my_model', 'ShowPortUnits','on')

ans =
on
```

Determine Units from a Logged Data Object

When saving simulation data from a scope with the `Dataset` format, you can find unit information in the `DataInfo` field of the timeseries object.

Note Scope support for signal units is only for the `Dataset` logging format and not for the legacy logging formats `Array`, `Structure`, and `Structure With Time`.

1. From the Scope window toolbar, select the Configuration Properties button .
2. In the Configuration Properties window, select the **Logging** tab.
3. Select the **Log data to workspace** check box. In the text box, enter a variable name for saving simulation data. For example, enter `ScopeData`.
4. From the Scope window toolbar, select the run button .
5. In the Command Window, enter

```
ScopeData.getElement(1).Values.DataInfo
```

```
Package: tsdata
Common Properties:
             Units: m/s (Simulink.SimulationData.Unit)
     Interpolation: linear (tsdata.interpolation)
```

Connect Signals with Different Units to a Scope

When there are multiple ports on a scope, Simulink ensures that each port receives data with only one unit. If you try to combine signals with different units (for example by using a Bus Creator block), Simulink returns an error.

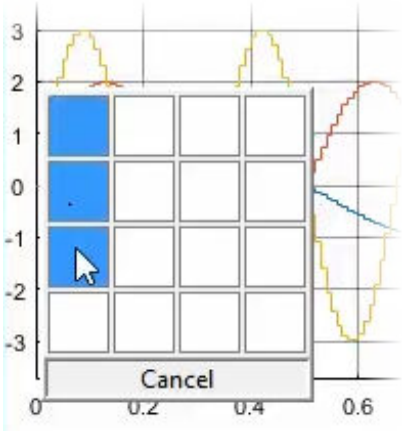
Scopes show units depending on the number of ports and displays:

- **Number of ports equal to the number of displays** — One port is assigned to one display with units for the port signal shown on the *y*-axis label.
- **Greater than the number of displays** — One port is assigned to one display, with the last display assigned the remaining signals. Different units are shown on the last *y*-axis label as a comma-separated list.

Select Number of Displays and Layout

1. From a Scope window, select the Configuration Properties button .
2. In the Configuration Properties dialog box, select the **Main** tab, and then select the **Layout** button.
3. Select the number of displays and the layout you want.

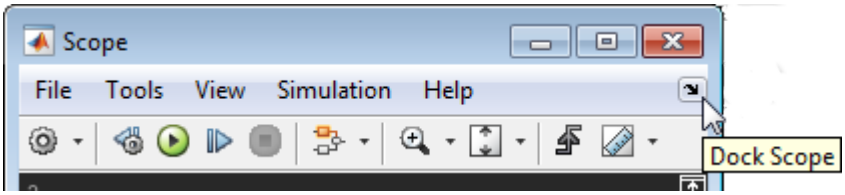
   You can select more than four displays in a row or column. Click within the layout, and then drag your mouse pointer to expand the layout to a maximum of 16 rows by 16 columns.



4 Click to apply the selected layout to the Scope window.

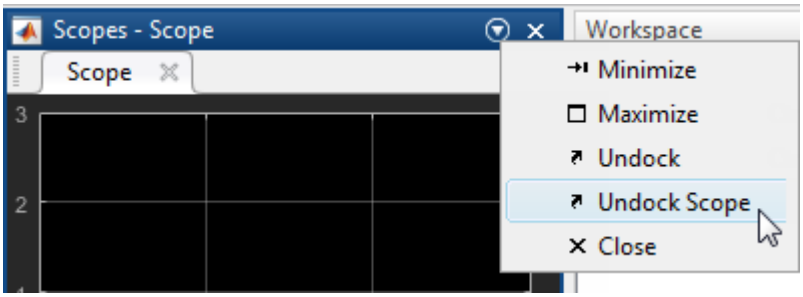
Dock and Undock Scope Window to MATLAB Desktop

1 In the right corner of a Scope window, click the Dock Scope button.



The Scope window is placed above the Command Window in the MATLAB desktop.

2 Click the Show Scope Actions button, and then click **Undock Scope**.



See Also

Floating Scope | Scope | Scope Viewer

Related Examples

- “Scope Blocks and Scope Viewer Overview” on page 28-6
- “Floating Scope and Scope Viewer Tasks” on page 28-67

Floating Scope and Scope Viewer Tasks

In this section...

These tasks walk through frequently used Floating Scope and Scope Viewer procedures.

Add Floating Scope Block to Model and Connect Signals

To add a Floating Scope block from the Simulink block library:

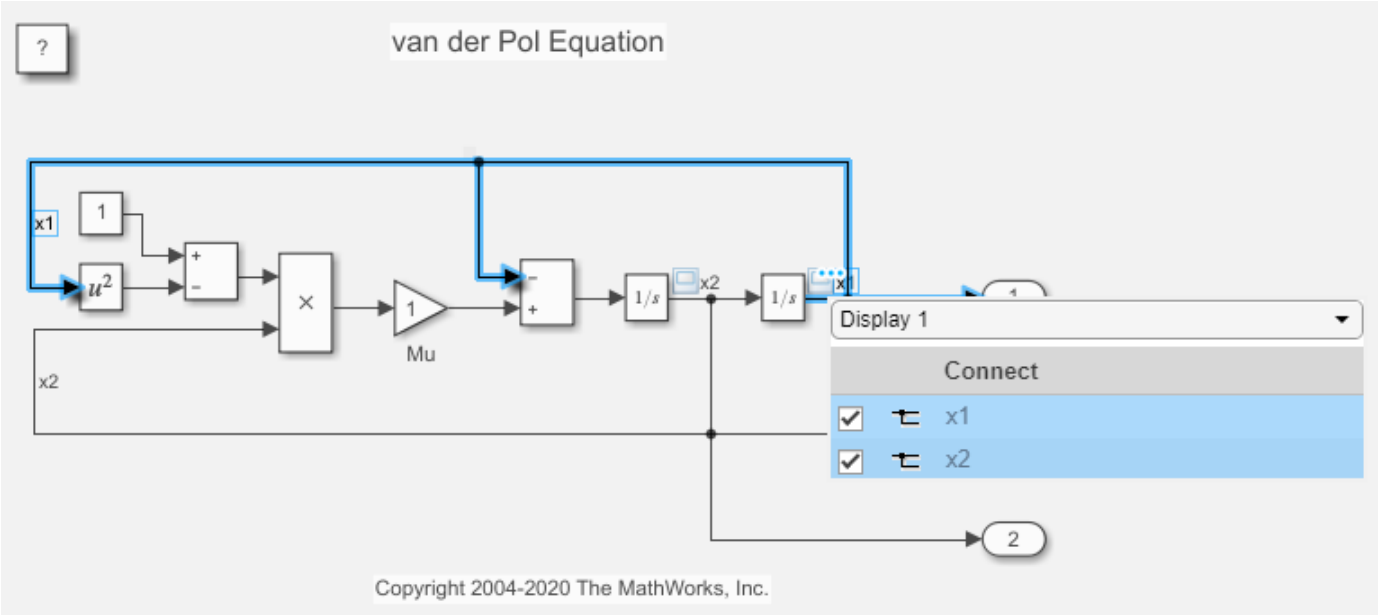
1. From the **Simulation** tab, click **Library Browser** .
2. From Simulink / Sinks, drag a copy of the Floating Scope block into your model.

To connect signals to your floating scope:

1. Double-click the Floating Scope block.
2. In the scope window toolbar, click the signal selection button . You may need to select the dropdown next to the lock button to see signal selection.

   The Simulink Editor canvas grays, indicating you can interactively select signals to connect to the scope.
3. Select the signals you want to connect to the scope. To select multiple signals, click and drag.
4. In the **Connect** pop-up, select the check box next to the signals you want to connect.



5 For scopes with multiple displays (subplots), select the desired display from the drop-down at the top of the Connect pop-up and connect the signals.

6 Click the **X** in the upper-right of the Simulink canvas.

Your signals are now connected to the floating scope. Run a simulation to see the plotted signals.

Add Scope Viewer to a Signal

1 Select a signal to view.

2 In the **Simulation** tab, in the **Prepare** gallery, select **Add Viewer**.

3 From the Add Viewer window, select a viewer, for example **Scope**.

Add Signals to an Existing Floating Scope or Scope Viewer

Connect signals to an existing Floating Scope or Scope viewer.

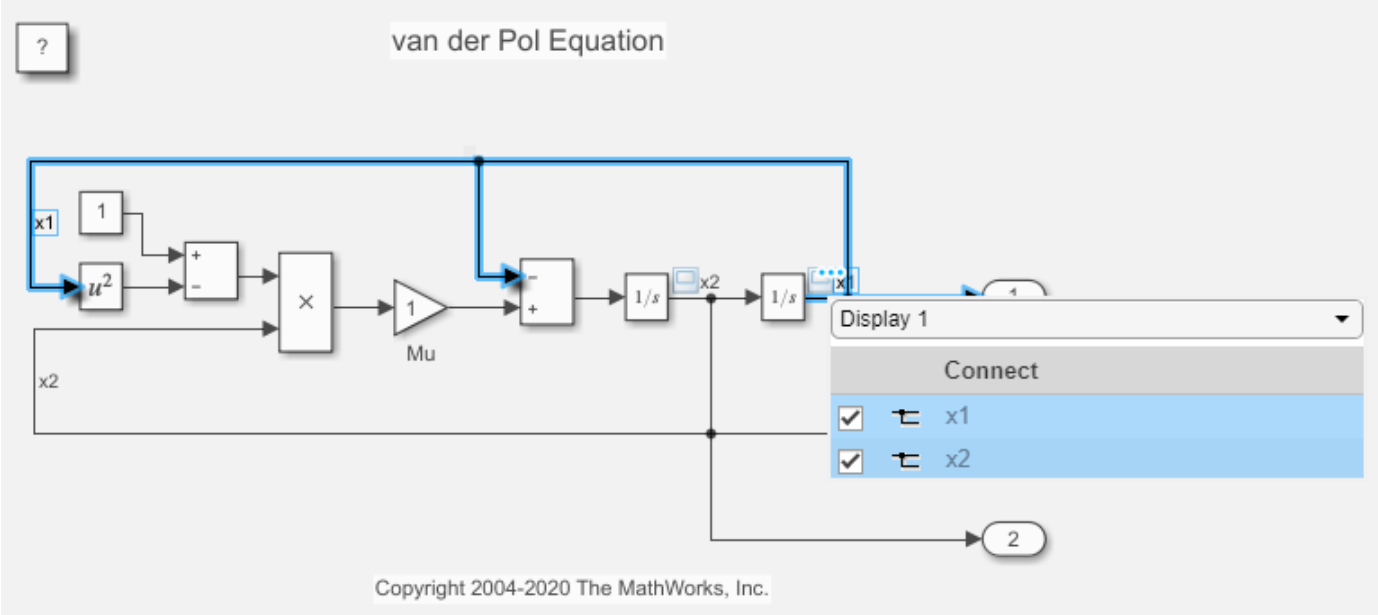
1 Open the scope by double-clicking a Floating Scope block or Scope viewer icon .

2 In the scope window toolbar, click the signal selection button . You may need to select the dropdown next to the lock button to see signal selection.

The Simulink Editor canvas grays, indicating you can interactively select signals to connect to the scope.

3 Select the signals you want to connect to the scope. Top select multiple signals, click and drag.

4 In the **Connect** pop-up, select the check box next to the signals you want to connect.



5 For scopes with multiple displays (subplots), select the desired display from the drop-down at the top of the Connect pop-up and connect the signals.

6 Click the **X** in the upper-right of the Simulink canvas.

Your signals are now connected to the scope. Run a simulation to see the plotted signals.

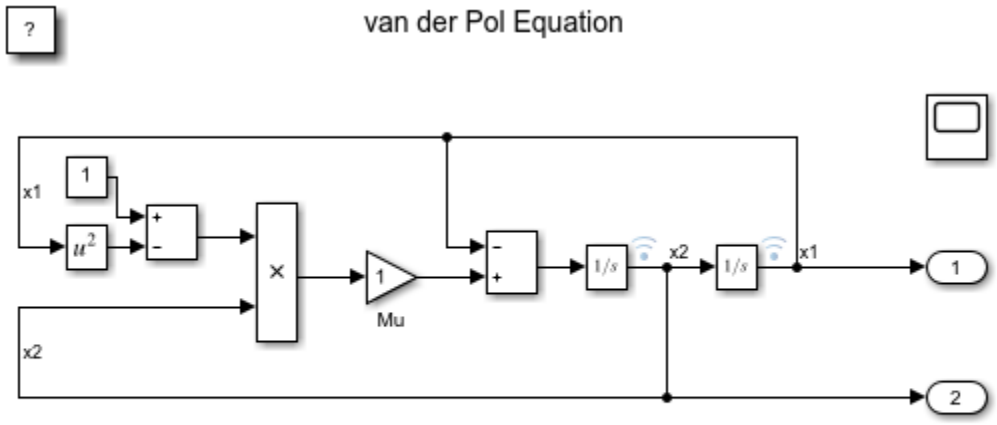
Save Simulation Data from Floating Scope

This example shows how to save signals to the MATLAB® Workspace using a Floating Scope block. You can follow the same procedure to save data from a Scope Viewer. To save data from a Scope or Time Scope block, see “Save Simulation Data Using Scope Block” on page 28-53.

This example uses a modified version of the `vdp` model. In this example, a floating scope is connected to the `x1` and `x2` signals.

Set Up Signal Logging from the Floating Scope

```
model = 'vdpFloatingScope';
open_system(model);
```

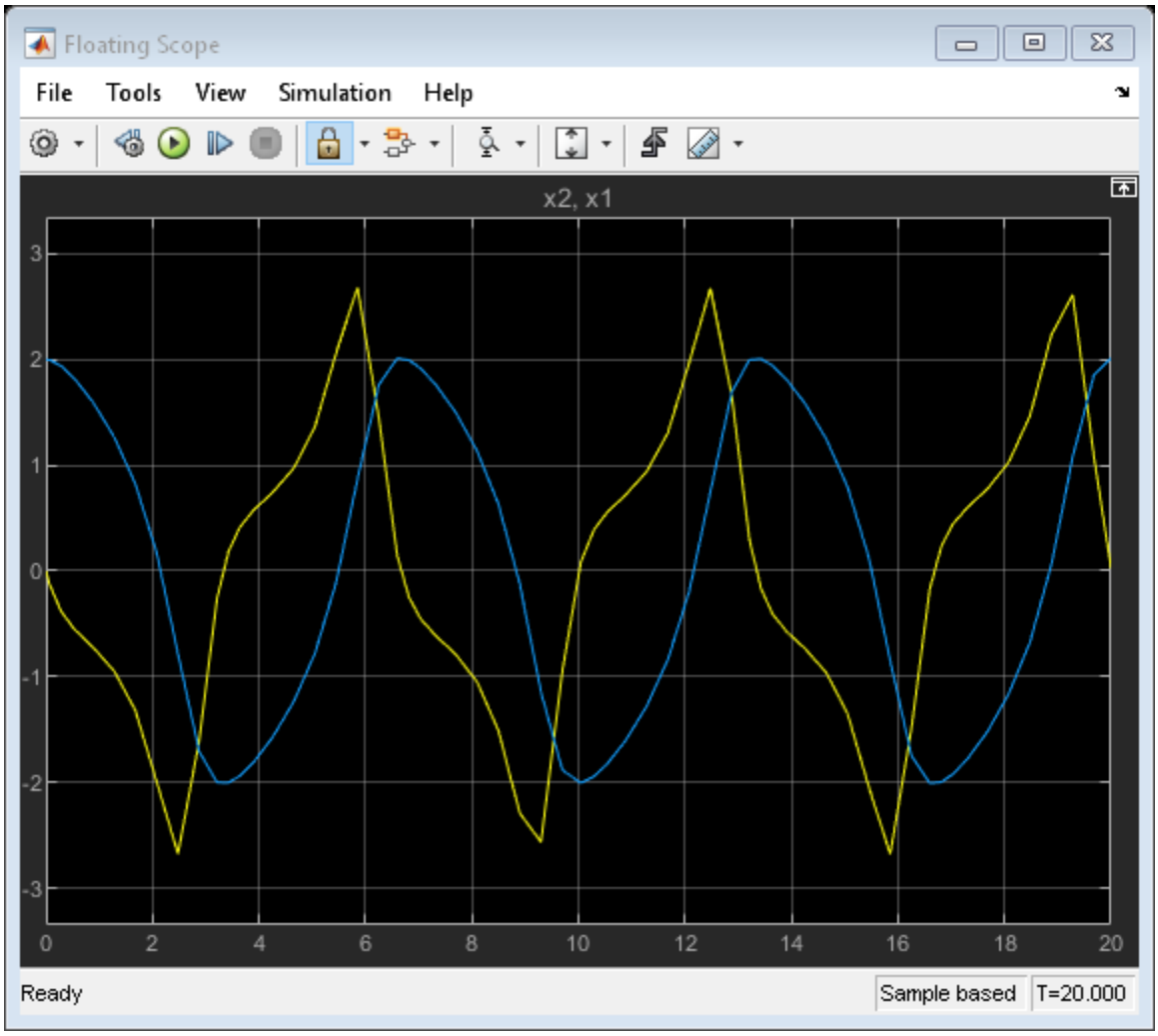


1. From the Floating Scope window, select **View > Configuration Properties**.
2. On the **Logging** tab, select **Log/Unlog Viewer Signals to Workspace**. When you click this button, Simulink places logging symbols on the signals connected to the Floating Scope.
3. From the Simulink Editor, on the **Modeling** tab, click **Model Settings**.
4. In the Configuration Properties window, select **Data Import/Export**.
5. Select the **Signal logging** check box. You can also set the parameter name from this window. The default parameter name is `logsout`.

Use Saved Simulation Data

Run a simulation. Simulink saves data to the MATLAB Workspace in a variable `out`.

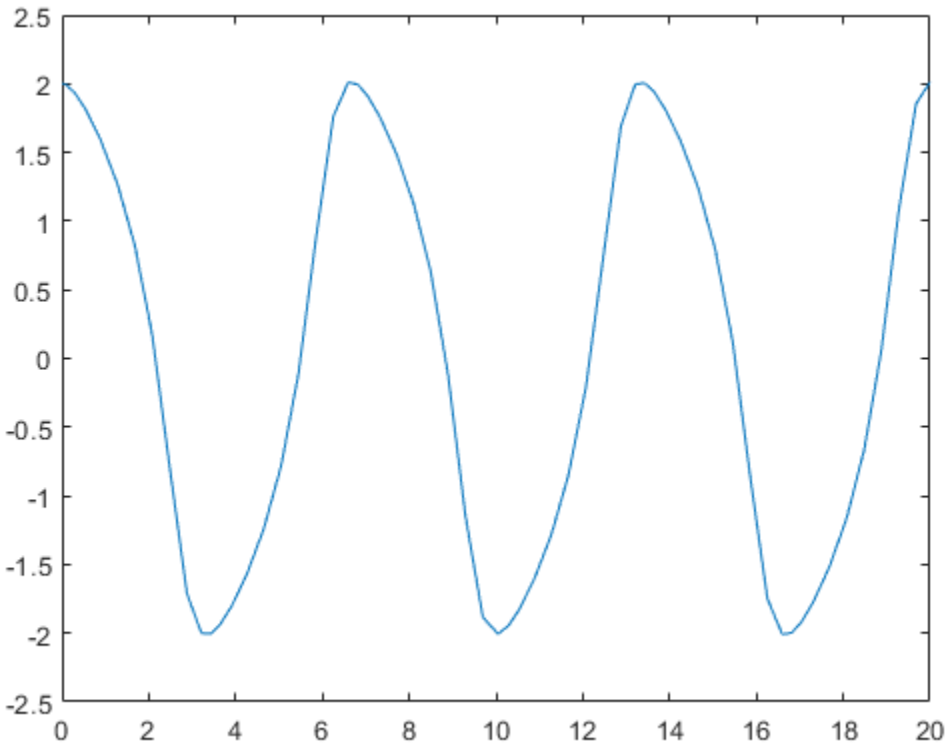
```
out = sim(model);
```



Simulink saves all logged or saved simulation data as one structure. In `out`, the scope data object `logsout` has one element for each logged signal.

In the MATLAB Command Window, plot the log data for `x1`.

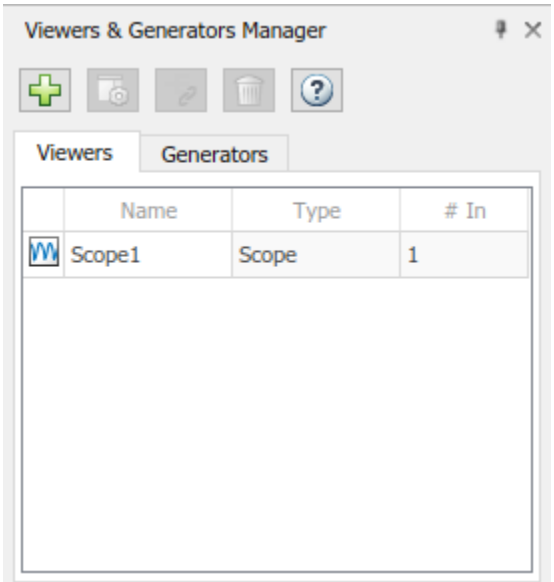
```
x1_data = out.logsout.get('x1').Values.Data;
x1_time = out.logsout.get('x2').Values.Time;
plot(x1_time,x1_data);
```



Add and Manage Viewers

Open the Viewers and Generators Manager. From the Simulink toolstrip **Simulation** tab, expand the **Prepare** gallery and select **Viewers Manager**.

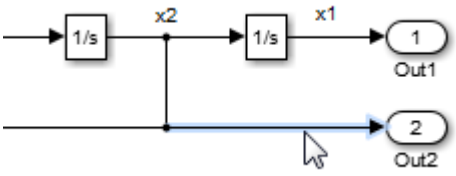
You can see any viewers or generators you added in the Viewers and Generators Manager.



- To add viewers or generators, click and choose a viewer. New viewers and generators are not connected to any signals.
- To connect signals, click on a viewer or generator, and click .
- To delete a viewer or generator, click .

Quickly Switch Visualization of Different Signals on a Floating Scope

1. Open a Floating Scope window.
2. On the toolbar, click the Lock button so that the icon is unlocked .
3. In the model, click a signal line to select and highlight the signal line. To select multiple signals, hold down the **Shift** key while selecting signals.



After clicking on the canvas, the selected display removes connections to any previously selected signals.

4. Run a simulation. As long as the unlock icon is showing, you can click between signals to switch which signal appears in the Floating Scope.

See Also

Floating Scope | Scope | Scope Viewer

Related Examples

- “Scope Blocks and Scope Viewer Overview” on page 28-6
- “Common Scope Block Tasks” on page 28-51
- “Step Through a Simulation” on page 2-12

Generate Signals Without Source Blocks

Signal generators create signals without adding a block to your model. Generators are added and managed through the Viewers and Generators Manager.

Attach Signal Generator

Context Menu

1. In the Simulink Editor, right-click the input to a block.
2. From the context menu, select **Create And Connect Generator** > ***`product`*** > ***`generator`***.

   The name of the generator you choose appears in a box connected to the block input.

Viewers and Generators Manager

1. From the **Simulation** tab, open the **Prepare** gallery and select **Viewers Manager**. The Viewers and Generators Manager panel opens to the side of the Simulink editor.
2. In the Viewers and Generators panel, select the **Generators** tab.
3. Click the  button and select a generator.

   The generator is added to the list of generators.
4. Click the generator you just added from the list and select the  button.
5. The canvas grays, indicating you are now in connect mode. Click the block you want to connect the generator to and in the pop-up, select the check box for the input port you want to attach the generator to.
6. Close connect mode by clicking the **X** in the upper right corner of the canvas.

Modify Signal Generator Parameters

Context Menu

1. Right-click the generator name and select **Generator Parameters**. In the Generator Parameters dialog box, enter parameters for this generator.
2. To rename a signal generator, double-click the generator label and type your new name for the generator.

Viewers and Generators Manager

1. From the Viewers and Generators Manager panel, select the generator from the **Generator** tab.
2. Click  and enter parameter values for this generator.
3. To rename a signal generator, double-click the generator name in the Generators list and type your new name for the generator.

Remove Signal Generator

Context Menu

1. Right-click a generator.
2. From the context menu, select **Disconnect Generator**.

   To also delete the generator, select **Disconnect and Delete Generator**

Viewers and Generators Manager

1. From the Viewers and Generators Manager panel, select the generator from the **Generator** tab.
2. Enter connect mode by selecting the  button
3. On the grayed canvas, select the generator and, in the pop-up, clear the check box next to the port you want to disconnect.
4. To delete the generator completely, in the Viewers and Generators Manager panel, also click  .

See Also

Scope Viewer

Related Examples

- “Common Scope Block Tasks” on page 28-51
- “Floating Scope and Scope Viewer Tasks” on page 28-67
- “Viewers and Generators Manager” on page 28-77

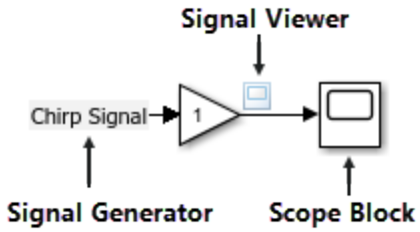
Viewers and Generators Manager

Using the Viewers and Generators Manager, you can manage viewers and generators from a central point.

Note The Viewers and Generators Manager requires that you have Java enabled when you start MATLAB.

Viewers and generators are not blocks. Blocks are dragged from the Library Browser and managed with block dialog boxes. Viewers and generators are mechanisms that create or visualize signals, but are added and managed through the Viewers and Generators Manager.

Symbols identify a viewer attached to a signal line, and signal names identify generators.



Open the Viewers and Generators Manager

From the **Simulation** tab, open the **Prepare** gallery and select **Viewers Manager**. The Viewers and Generators Manager panel opens to the side of the Simulink editor.

You can also access the Viewers and Generators Manager by right-clicking a signal or block input and selecting **Viewers and Generators Manager**.

Change Parameters

1. Open the Viewers and Generators Manager.
2. In the list of **Generators** or **Viewers**, select the viewer or generator you want to modify.
3. Select the  button.
   - For a generator, the parameters dialog box opens for that generator type.
   - For a viewer, either a parameter dialog opens or the viewer itself opens. If the viewer opens, you can access parameters from the  button.
4. Review and change parameters.

Connect Viewers and Generators

Connect signals to a new viewer or generator using the Viewers and Generators Manager.

1. Open the Viewers and Generators Manager panel.

2. Select either the **Viewers** or **Generators** tab.
3. Click  .
4. From the list of viewers or generators, select the viewer or generator you just added.
5. Select the  button. The canvas grays, indicating you are now in connect mode.
6. For generators, click the block you want to connect the generator to and in the pop-up, select the check box for the input port you want to attach to.

   For viewers, click the signal or signals you want to visualize and in the pop-up, select the check box next to the signals you want to connect. If you have multiple displays or specific inputs, select the display you want to connect to from the drop down before selecting the check box.
7. Close connect mode by clicking the **X** in the upper right corner of the canvas.

View Test Point Data

Use a Scope viewer available from the Viewers and Generators Manager to view any signal that is defined as a test point in a referenced model. A test point is a signal that you can always see when using a Scope viewer in a model.

Note With some viewers (for example, XY Graph, To Video Display, Matrix Viewer), you cannot use the Signal Selector to select signals with test points in referenced models.

For more information, see “Configure Signals as Test Points” on page 75-43.

Customize Viewers and Generators Manager

You can add custom signal viewers or generators so that they appear in the Viewers and Generators Manager. This procedure adds a custom viewer named `newviewer` to the Viewers and Generators Manager:

1. Create a new Simulink library by selecting **Simulation** > **New** > **Library**.
2. Save the library as `newlib`.
3. In the MATLAB Command Window, set the library type to a viewer library:

   ```
   set_param('newlib','LibraryType','SSMgrViewerLibrary')
   ```

   To set the library type for generators, use the type `'SSMgrGenLibrary'`. For example:

   ```
   set_param('newlib','LibraryType','SSMgrGenLibrary')
   ```
4. Set the display name of the library:

   ```
   set_param('newlib','SSMgrDisplayString','My Custom Library')
   ```
5. Add your custom viewer or generator to the library by dragging and dropping into the Simulink canvas.

   Note If the viewer is a compound viewer, such as a subsystem with multiple blocks, make the top-level subsystem an atomic one.

6. Set the `iotype` of the viewer. For example:

   ```
   set_param('newlib/newviewer','iotype','viewer')
   ```

7. Save the library `newlib`.

8. Using the MATLAB editor, create a file named `sl_customization.m`. In this file, enter a directive to incorporate the new library as a viewer library.

   For example, to save `newlib` as a viewer library, add these lines:

   ```
   function sl_customization(cm)
   cm.addSigScopeMgrViewerLibrary('newlib')
   %end function
   ```

   To add a library as a generator library, use this syntax instead:

   ```
   cm.addSigScopeMgrGeneratorLibrary('newlib')
   ```

9. Add a corresponding `cm.addSigScope` line for each viewer or generator library you want to add.

10. Save the `sl_customization.m` file on your MATLAB path. Edit this file to add new viewer or generator libraries.

11. To see the new custom libraries, restart MATLAB and start the Viewers and Generators Manager.

See Also

Floating Scope | Scope | Scope Viewer

Related Examples

- “Common Scope Block Tasks” on page 28-51
- “Floating Scope and Scope Viewer Tasks” on page 28-67
- “Scope Blocks and Scope Viewer Overview” on page 28-6

Control Scope Blocks Programmatically

This example shows how to control scopes with programming scripts.

Use Simulink Configuration Object

Use a Scope Configuration object for programmatic access to scope parameters.

- Modify the title, axis labels, and axis limits
- Turn on or off the legend or grid
- Control the number of inputs
- Change the number of displays and which display is active

Create a model and add a Scope and Time Scope block. Then, use `get_param` to create the Scope Configuration object `myConfiguration`.

```
mdl = 'myModel';
new_system(mdl);
add_block('simulink/Sinks/Scope', [mdl '/myScope']);
add_block('dspsnks4/Time Scope', [mdl '/myTimeScope']);
myConfiguration = get_param([mdl '/myScope'],'ScopeConfiguration')
```

```
myConfiguration =
Scope configuration with properties:

                               Name: 'myScope'
                           Position: [360 302 560 420]
                            Visible: 0
                OpenAtSimulationStart: 0
                      DisplayFullPath: 0
    PreserveColorsForCopyToClipboard: 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: '%<SignalLabel>'
                           ShowLegend: 0
                             ShowGrid: 1
                  PlotAsMagnitudePhase: 0
                               YLimits: [-10 10]
                                YLabel: ''
                          DataLogging: 0
             DataLoggingVariableName: 'ScopeData'
          DataLoggingLimitDataPoints: 0
                DataLoggingMaxPoints: '5000'
```

```
              DataLoggingDecimateData: 0
                DataLoggingDecimation: '2'
                DataLoggingSaveFormat: 'Dataset'
```

Set a property.

```
myConfiguration.DataLoggingMaxPoints ='10000';
```

Find the Scope and Time Scope blocks.

```
find_system(mdl,'LookUnderMasks','on','IncludeCommented','on', ...
'AllBlocks','on','BlockType','Scope')
```

```
ans = 2x1 cell
    {'myModel/myScope'    }
    {'myModel/myTimeScope'}
```

Find only Simulink Scope blocks.

```
find_system(mdl,'LookUnderMasks','on','IncludeCommented','on',...
'AllBlocks','on','BlockType','Scope','DefaultConfigurationName',...
'Simulink.scopes.TimeScopeBlockCfg')
```

```
ans = 1x1 cell array
    {'myModel/myScope'}
```

Find only the DSP Time Scope blocks.

```
find_system(mdl,'LookUnderMasks','on','IncludeCommented','on',...
'AllBlocks','on','BlockType','Scope','DefaultConfigurationName',...
'spbscopes.TimeScopeBlockCfg')
```

```
ans = 1x1 cell array
    {'myModel/myTimeScope'}
```

Scope Configuration Properties

For details about the Scope Configuration object properties, see `TimeScopeConfiguration`.

See Also

Floating Scope | Scope | Scope Viewer

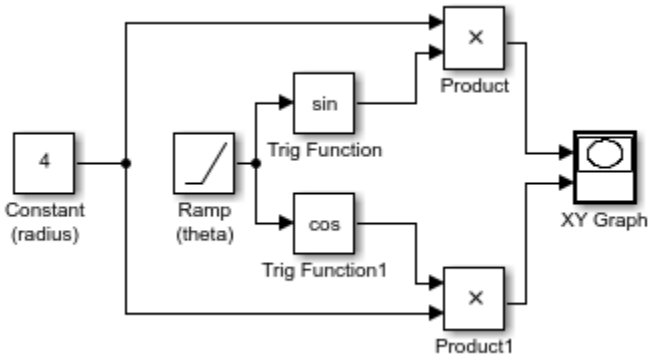
Related Examples

- “Common Scope Block Tasks” on page 28-51
- “Floating Scope and Scope Viewer Tasks” on page 28-67
- “Scope Blocks and Scope Viewer Overview” on page 28-6

Plot Circle with XY Graph

This example plots a circle in Simulink using an XY Graph block.

The `ex_xygraph_block_circle` model computes a circle of radius 4, centered at the origin of x-y plane.



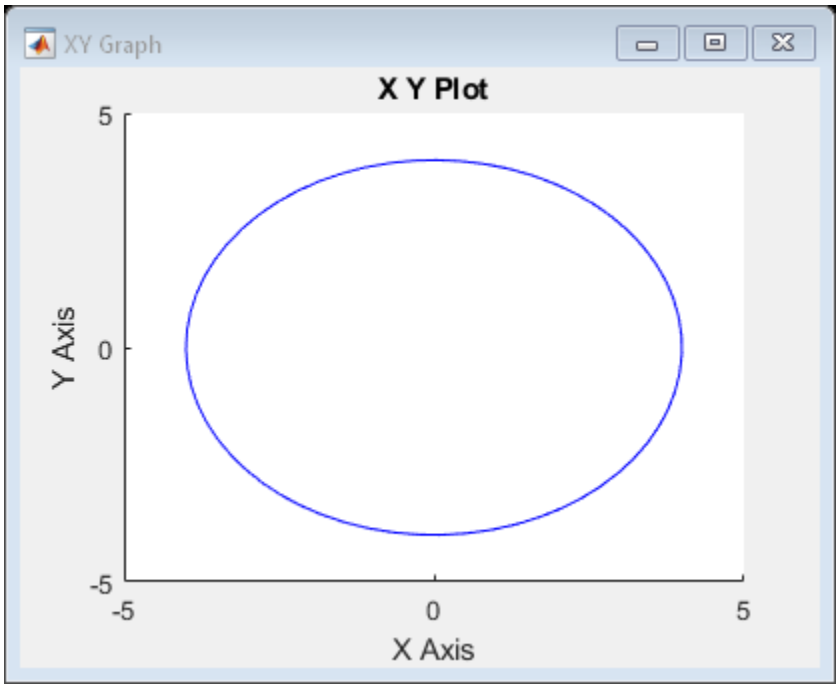
The plotted functions are:

$$x = 4\cos(\theta)$$

$$y = 4\sin(\theta)$$

When you simulate the model, a figure window appears showing the plotted circle.

```
ans =

  Simulink.SimulationOutput:
                  tout: [101x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```



See Also

Trigonometric Function | XY Graph

Inspecting and Comparing Simulation Data

View Data in the Simulation Data Inspector

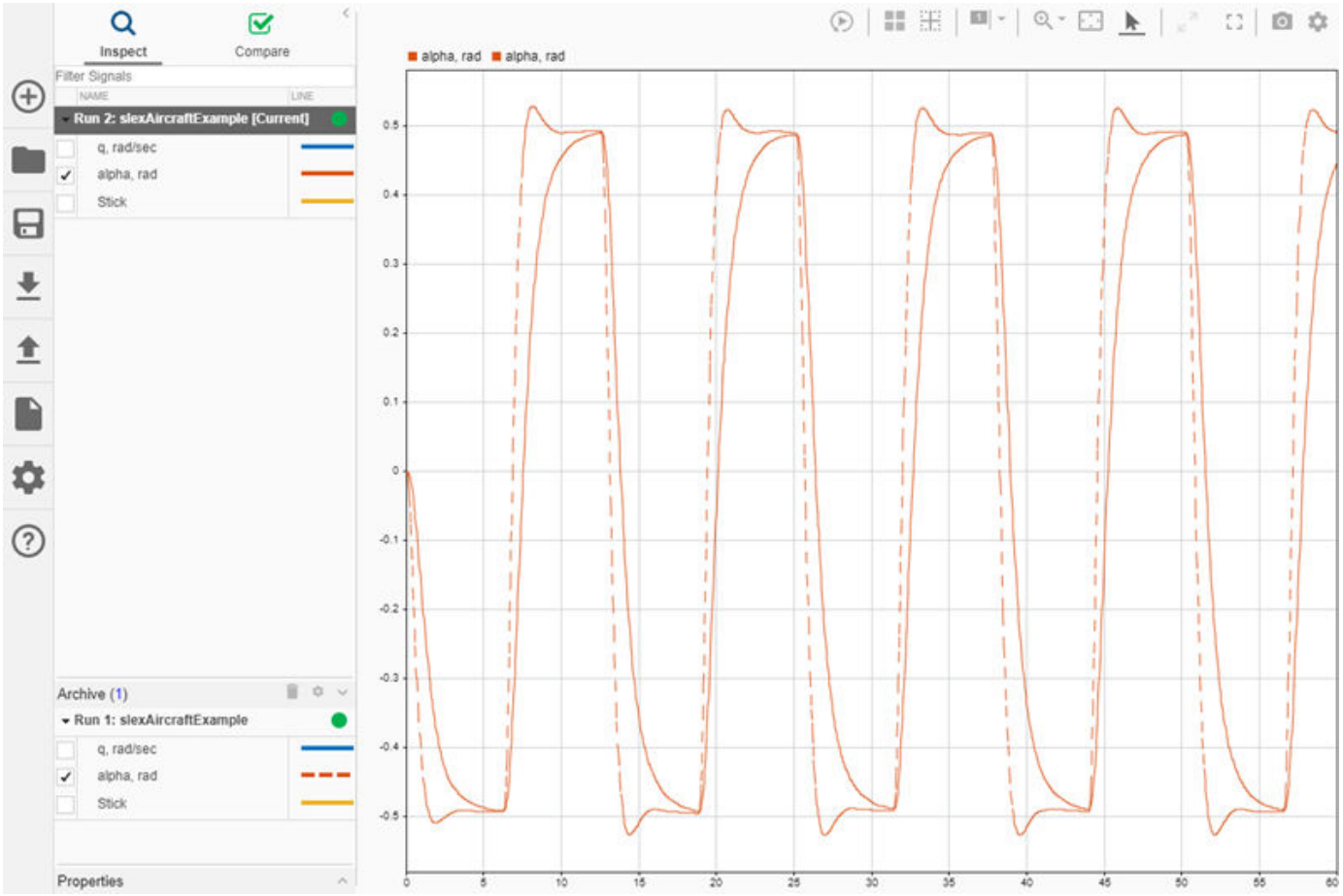
You can use the Simulation Data Inspector to visualize the data you generate throughout the design process. Simulation data that you log in a Simulink model logs to the Simulation Data inspector. You can also import test data and other recorded data into the Simulation Data Inspector to inspect and analyze it alongside the logged simulation data. The Simulation Data Inspector offers several types of plots, which allow you to easily create complex visualizations of your data.

View Logged Data

Logged signals as well as outputs and states logged using the `Dataset` format automatically log to the Simulation Data Inspector when you simulate a model. You can also record other kinds of simulation data so the data appears in the Simulation Data Inspector at the end of the simulation. To see states and output data logged using a format other than `Dataset` in the Simulation Data Inspector, in the **Model Configuration Parameters Data Import/Export** pane, select the **Record logged workspace data in Simulation Data Inspector** option.

Note When you log states and outputs using the `Structure` or `Array` format, you must also log time for the data to record to the Simulation Data Inspector.

The Simulation Data Inspector displays available data in the table in the **Inspect** pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. For more information, see “Create Plots Using the Simulation Data Inspector” on page 29-94.



The Simulation Data Inspector manages incoming simulation data using the archive. By default, the previous run moves to the archive when you start a new simulation. You can plot signals from the archive, or you can drag runs of interest back into the work area.

Import Data from the Workspace or a File

You can import data from the base workspace or from a file to view on its own or alongside simulation data. The Simulation Data Inspector supports all built-in data types and many data formats for importing data from the workspace. In general, whatever the format, sample values must be paired with sample times. File import supports MAT, CSV, and Microsoft Excel files as well as MDF files with `.mdf`, `.mf4`, `.mf3`, `.data`, and `.dat` extensions. The Simulation Data Inspector allows up to 8000 channels per signal in a run created from imported workspace data.

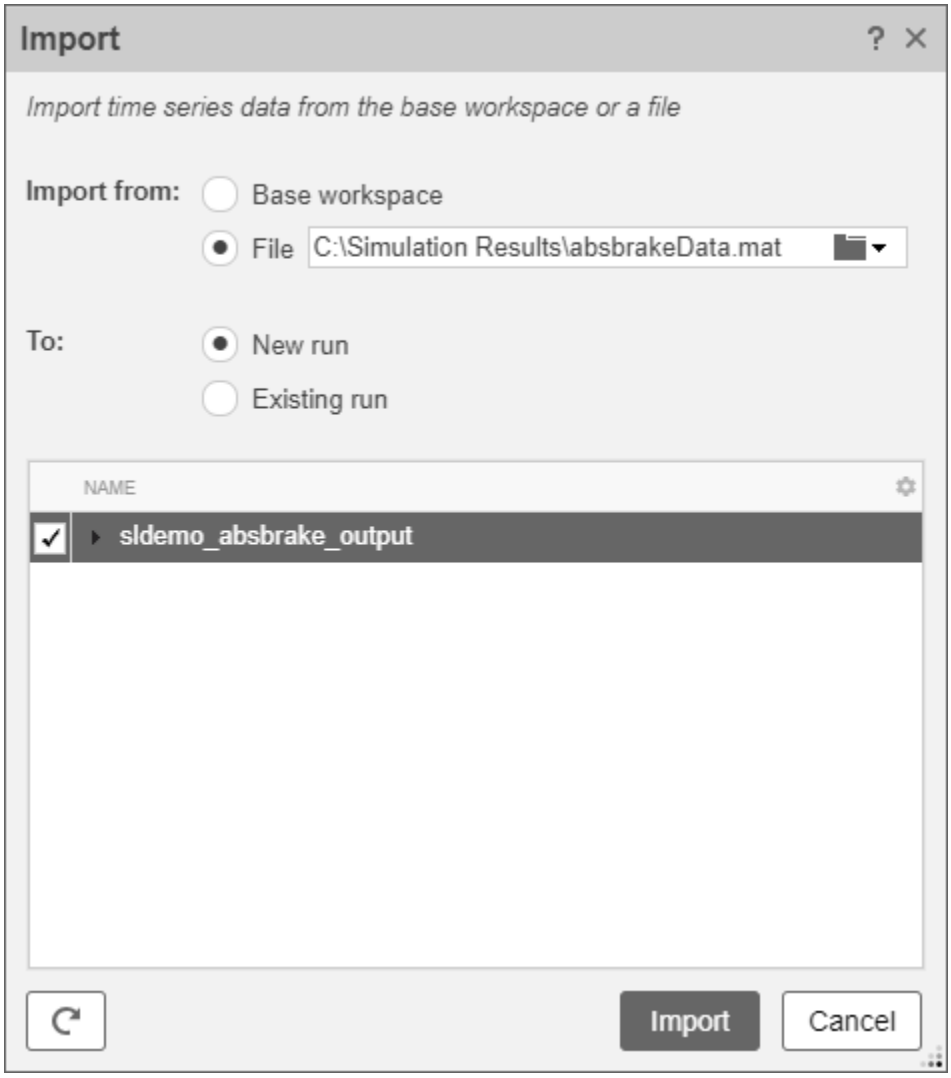
Note The Simulation Data Inspector can import data from CSV and Microsoft Excel files when the data in the file is formatted according to “Import Data from a CSV File into the Simulation Data Inspector” on page 29-51 or “Microsoft Excel Import and Export Format” on page 29-43.

To import data from the workspace or from a file that is saved in a data or file format that the Simulation Data Inspector does not support, you can write your own workspace data or file reader to import the data using the `io.reader` class. You can also write a custom reader to use instead of the built-in reader for supported file types. For examples, see:

- “Import Data Using a Custom File Reader” on page 29-17
- “Import Workspace Variables Using a Custom Data Reader” on page 29-11

To import data, select the **Import** button in the Simulation Data Inspector. 

In the Import dialog, you can choose to import data from the workspace or from a file. The table below the options shows data available for import. If you do not see your workspace variable or file contents in the table, that means the Simulation Data Inspector does not have a built-in or registered reader that supports that data. You can select which data to import using the check boxes, and you can choose whether to import that data into an existing run or a new run.

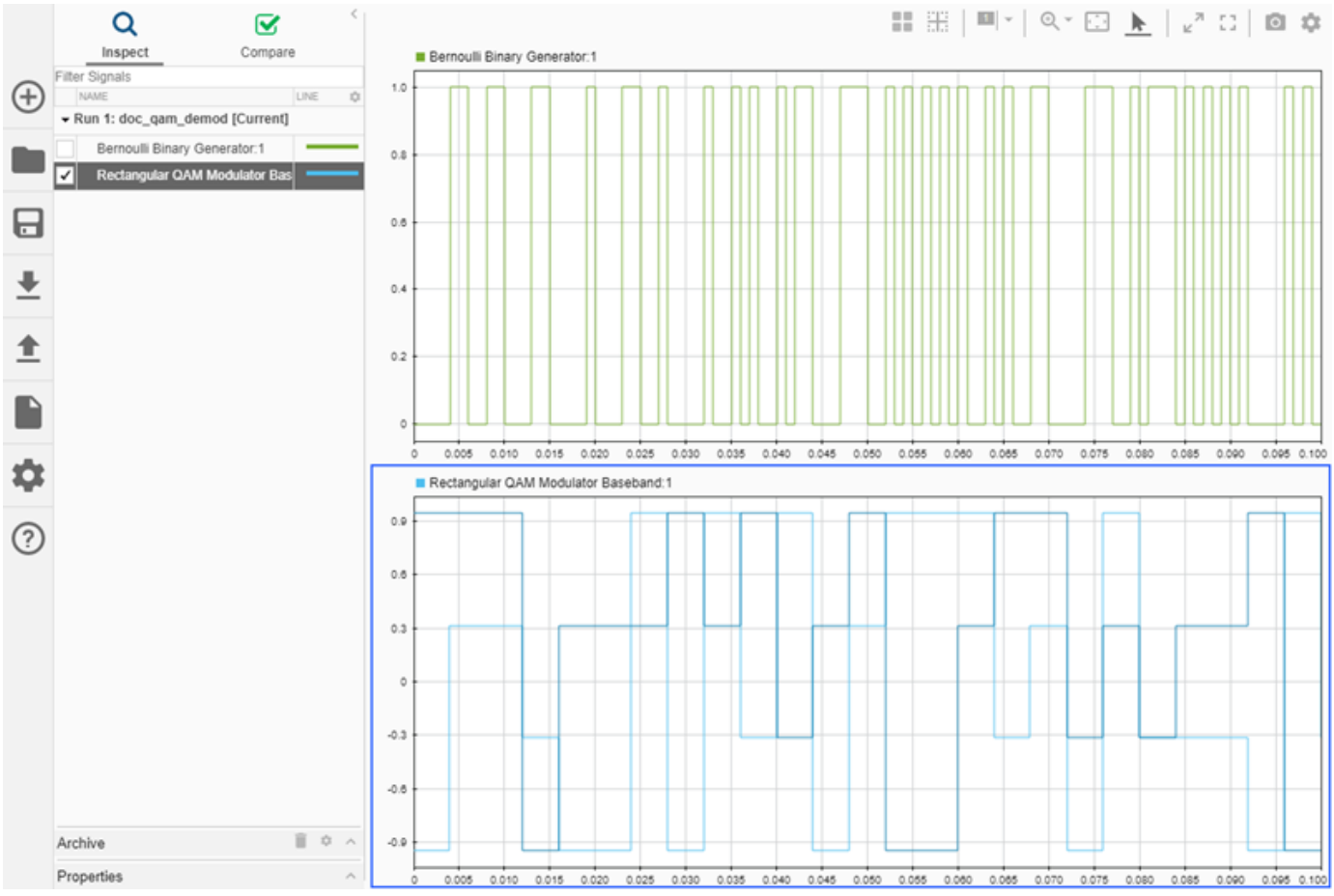


When you import data into a new run, the run always appears in the work area. You can manually move imported runs to the archive.

View Complex Data

To view complex data in the Simulation Data Inspector, import the data or log the signals to the Simulation Data Inspector. You can control how to visualize the complex signal using the **Properties** pane in the Simulation Data Inspector and in the **Instrumentation Properties** for the signal in the model. To access the **Instrumentation Properties** for a signal, right-click the logging badge for the signal and select **Properties**.

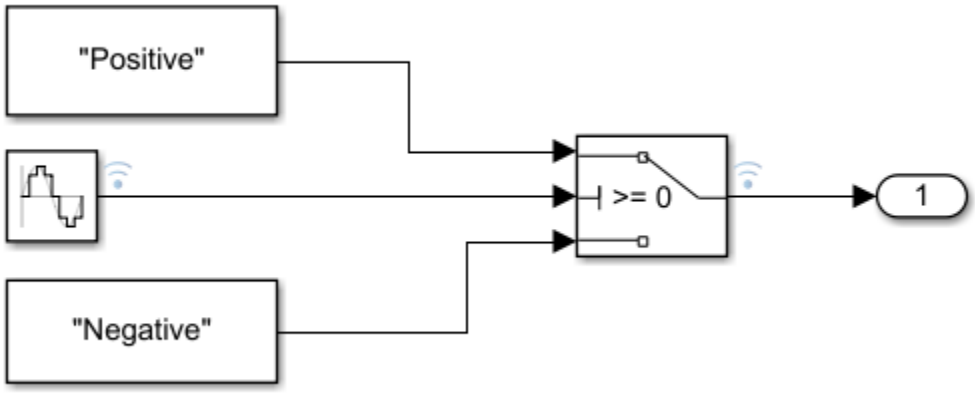
You can specify the **Complex Format** as Magnitude, Magnitude-Phase, Phase, or Real-Imaginary. If you select Magnitude-Phase or Real-Imaginary for the **Complex Format**, the Simulation Data Inspector plots both components of the signal when you select the check box for the signal. For signals in Real-Imaginary format, the **Line Color** specifies the color of the real component of the signal, and the imaginary component is a different shade of the **Line Color**. For example, the `Rectangular QAM Modular Baseband` signal on the lower graph displays the real component of the signal in light blue, matching the **Line Color** parameter, and the imaginary component is shown in a darker shade of blue.



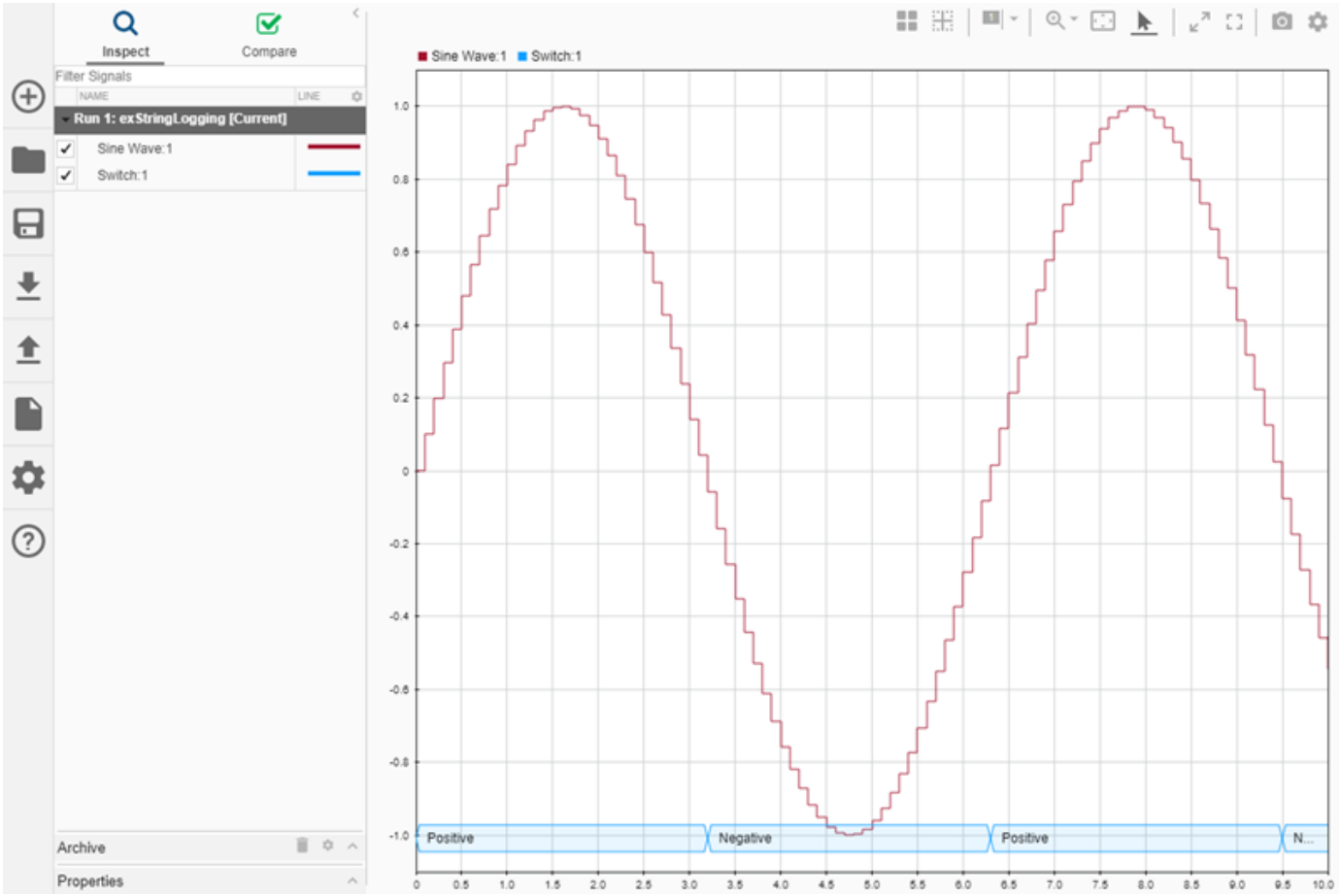
For signals in Magnitude-Phase format, the **Line Color** specifies the color of the magnitude component, and the phase is displayed in a different shade of the **Line Color**.

View String Data

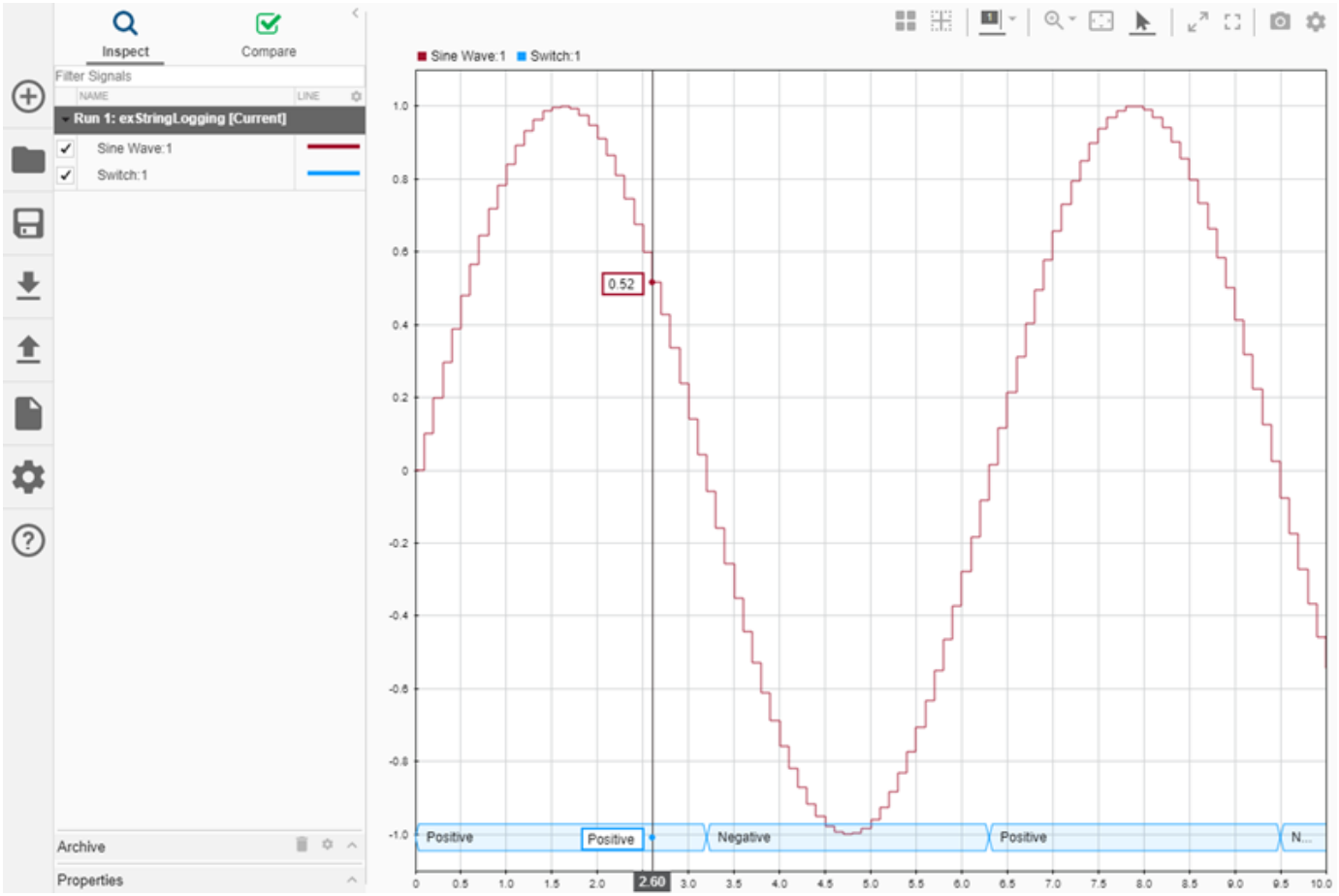
You can log and view string data with your signal data in the Simulation Data Inspector. For example, consider this simple model. The value of the sine wave block controls whether the switch sends a string reading `Positive` or `Negative` to the output.



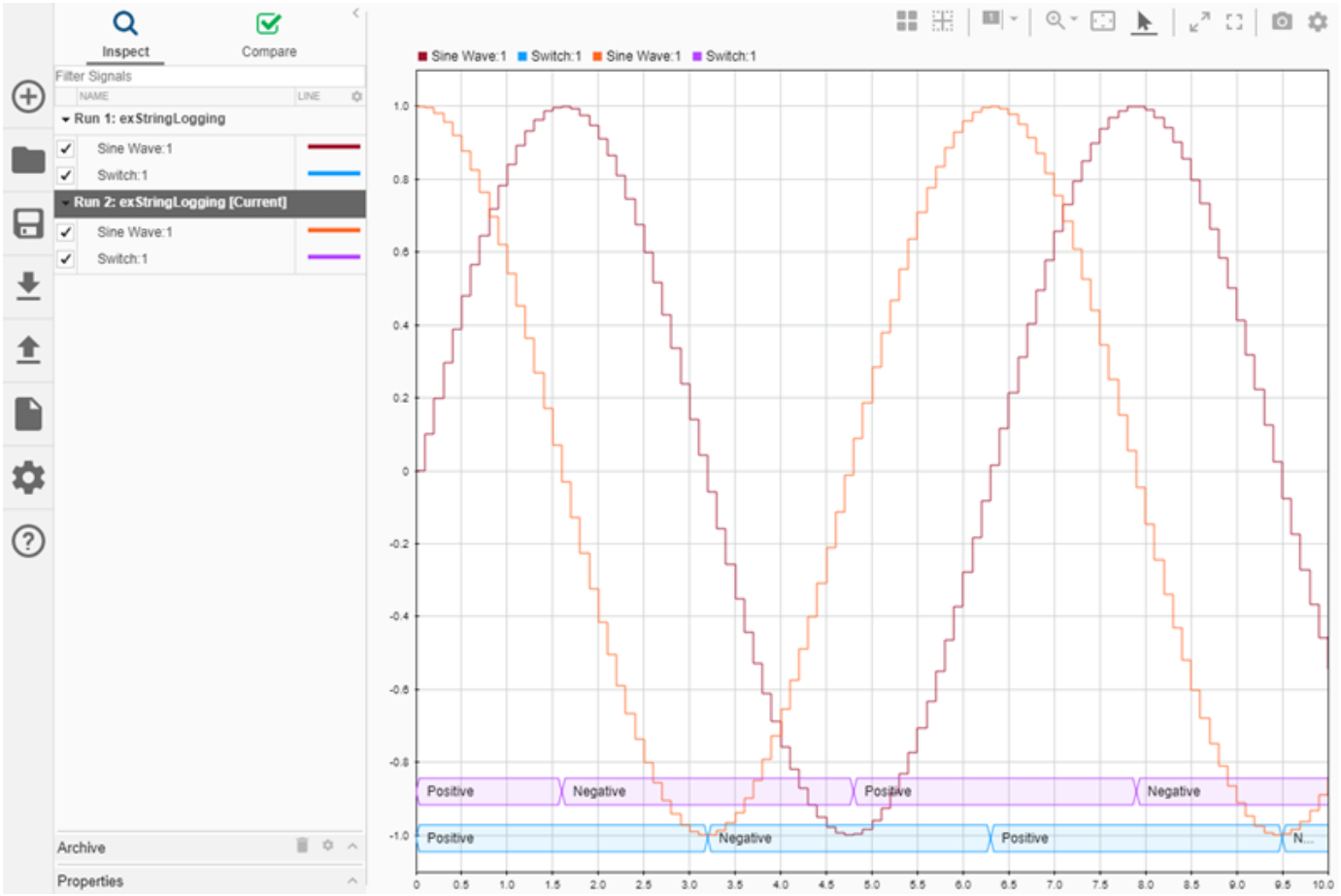
The plot shows the results of simulating the model. The string signal is shown at the bottom of the graphical viewing area. The value of the signal is displayed inside a band, and transitions in the string signal's value are marked with criss-crossed lines.



You can use cursors to inspect how the string signal values correspond with the sine signal's values.



When you plot multiple string signals on a plot, the signals stack in the order they were simulated or imported, with the most recent signal positioned at the top. For example, you might consider the effect of changing the phase of the sine wave controlling the switch.

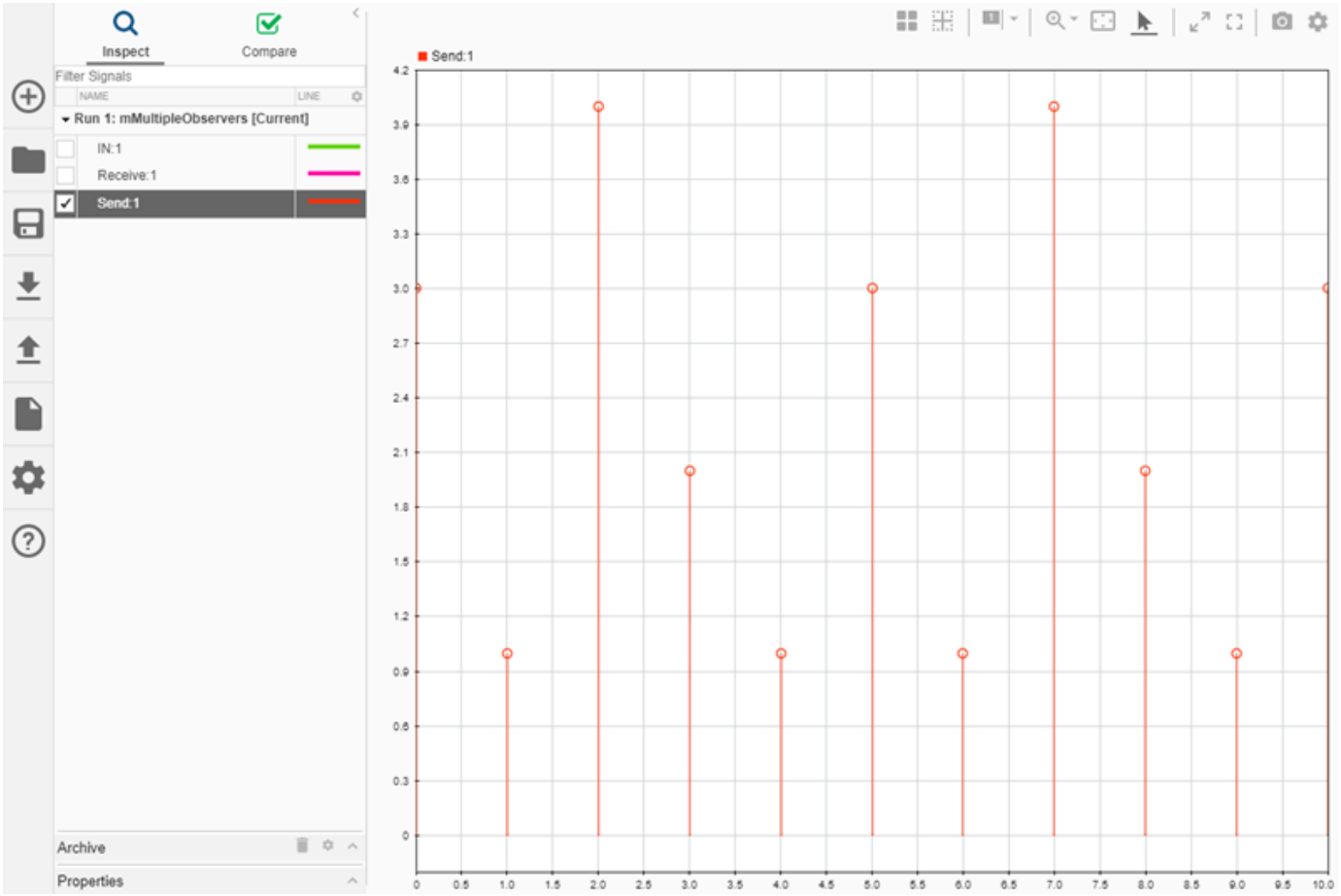


View Frame-Based Data

Processing data in frames rather than point by point provides a performance boost needed in some applications. To view frame-based data in the Simulation Data Inspector, you have to specify that the signal is frame-based in the **Instrumentation Properties** for the signal. To access the **Instrumentation Properties** dialog for a signal, right-click the signal's logging badge and select **Properties**. To specify a signal as frame-based, select **Columns as channels (frame based)** for **Input processing**.

View Event-Based Data

You can log or import event data to the Simulation Data Inspector. To view the logged event-based data, select the check box next to `Send: 1`. The Simulation Data Inspector displays the data as a stem plot, with each stem representing the number of events that occurred for a given sample time.



See Also

More About

- Inspect Simulation Data on page 29-107
- Compare Simulation Data on page 29-130
- Share Simulation Data Inspector Data and Views on page 29-83
- Decide How to Visualize Data on page 30-2
- Dataset Conversion for Logged Data on page 72-12

Import Workspace Variables Using a Custom Data Reader

When your workspace data is in a format that built-in readers do not support, you can write a custom data reader to import the data into the Simulation Data Inspector. This example explains the parts of the class definition for a custom workspace reader and shows how to register the reader with the Simulation Data Inspector. Open the `SimpleStructReader.m` file to view the complete class definition.

Create Workspace Data

First, create workspace data to import into the Simulation Data Inspector using the custom reader. Suppose you store each signal as a structure with fields for the data (`d`), the time values (`t`), and the signal name (`n`).

```
time = 0:0.1:100;
time = time';
lineData = 1/4*time;
sineWave = sin((2*pi)/50*time);
squareWave = square((2*pi)/30*time);

mySineVar.d = sineWave;
mySineVar.t = time;
mySineVar.n = "Sine Wave";

myLineVar.d = lineData;
myLineVar.t = time;
myLineVar.n = "Line";

mySquareVar.d = squareWave;
mySquareVar.t = time;
mySquareVar.n = "Square Wave";
```

Write the Class Definition for a Custom Reader

Write a class definition that specifies how the custom reader extracts relevant data and metadata from the workspace variables. Save the class definition file in a location on the MATLAB™ path.

The class definition starts by inheriting from the `io.reader` class, followed by property and method definitions. The custom reader in this example defines the property `ChannelIndex`. You can use the reader to import individual structures or an array of structures from the workspace. The `ChannelIndex` property is used when importing an array of structures.

```
classdef SimpleStructReader < io.reader

    properties
        ChannelIndex
    end
```

Every custom reader must define the `getName`, `getTimeValues`, and `getDataValues` methods. When you write a custom reader to import data from the workspace, you must also define the `supportsVariable` method. The reader in this example also defines the `getChildren` method to support importing an array of structures.

The `supportsVariable` method checks which variables in the workspace are supported by the reader. In this example, the `supportsVariable` returns `true` when:

1. The structure contains the appropriate fields.
2. The `n` field of the structure contains a string or character array that represents the signal name.
3. The `t` field of the structure is a column vector of double data.
4. The `d` field contains numeric data.
5. The `d` field is the same size as the `t` field, meaning there is a sample value for each time step.

```
function supported = supportsVariable(~, val)
    % Support structure with fields t (time), d (data), and n (name)
    supported = ...
        isstruct(val) && ...
        isfield(val,'t') && ...
        isfield(val,'d') && ...
        isfield(val,'n');
    if supported
        for idx = 1:numel(val)
            varName = val(idx).n;
            time = val(idx).t;
            varData = val(idx).d;

            % Name must be string or character array
            if ~ischar(varName) && ~isstring(varName)
                supported = false;

            % Time must be double column vector
            elseif ~isa(time,'double') || ~iscolumn(time)
                supported = false;

            % Data size must match time size
            else
                timeSz = size(time);
                dataSz = size(varData);

                if ~isnumeric(varData) || ~isequal(dataSz, timeSz)
                    supported = false;
                end
            end
        end
    end
end
```

The `getChildren` method creates a `SimpleStructReader` object for each structure in an array of structures. When the variable to import is not scalar, the `getChildren` method assigns a value to the `ChannelIndex` property added to the class for the custom reader. The `VariableValue` property for each `SimpleStructReader` object returned by the `getChildren` method is the array of structures. Other methods use the `ChannelIndex` property to extract the appropriate signal name, signal data, and time values from each object.

```
function childObj = getChildren(obj)
    childObj = {};
    if ~isscalar(obj.VariableValue) && isempty(obj.ChannelIndex)
        numChannels = numel(obj.VariableValue);
        childObj = cell(numChannels,1);
        for idx = 1:numChannels
            childObj{idx} = SimpleStructReader;
            childObj{idx}.VariableName = sprintf('%s(%d)',obj.VariableName,idx);
            childObj{idx}.VariableValue = obj.VariableValue;
```

```
            childObj{idx}.ChannelIndex = idx;
        end
    end
end
```

The `getName` method assigns the name stored in the `n` field of the structure to each imported signal. When the imported variable is scalar, the method gets the name from the `VariableValue` property of the `SimpleStructReader` object. When the imported data is an array of structures, the appropriate structure is extracted from the `VariableValue` property using the `ChannelIndex` property. The top-level node of the array is named `Signal Array`.

```
function retName = getName(obj)
    if isscalar(obj.VariableValue)
        retName = char(obj.VariableValue.n);
    elseif ~isempty(obj.ChannelIndex)
        varVal = obj.VariableValue(obj.ChannelIndex);
        retName = char(varVal.n);
    else
        retName = 'Signal Array';
    end
end
```

The `getTimeVals` and `getDataVals` methods handle scalar and nonscalar structures similar to how the `getName` method does. For a scalar structure, both methods extract the appropriate field from the `VariableValue` property of the `SimpleStructReader` object. For a nonscalar structure, both methods access the appropriate structure in the `VariableValue` property using the `ChannelIndex` property. Finally, for the top-level node of the array, time and data are both returned as empty.

```
function timeVals = getTimeValues(obj)
    if isscalar(obj.VariableValue)
        timeVals = obj.VariableValue.t;
    elseif ~isempty(obj.ChannelIndex)
        varVal = obj.VariableValue(obj.ChannelIndex);
        timeVals = varVal.t;
    else
        timeVals = [];
    end
end

function dataVals = getDataValues(obj)
    if isscalar(obj.VariableValue)
        dataVals = obj.VariableValue.d;
    elseif ~isempty(obj.ChannelIndex)
        varVal = obj.VariableValue(obj.ChannelIndex);
        dataVals = varVal.d;
    else
        dataVals = [];
    end
end
```

Register a Custom Reader

After you write the class definition for the custom reader, you must register the reader before you can use it to import data into the Simulation Data Inspector. The Simulation Data Inspector does not store registered readers between MATLAB sessions, so you need to register a custom reader at the start of each new MATLAB session. To register the workspace data reader in this example, use the `registerWorkspaceReader` method.

```
registerWorkspaceReader(SimpleStructReader);
```

To confirm that the reader is registered, use the `io.reader.getRegisteredWorkspaceReaders` method.

```
io.reader.getRegisteredWorkspaceReaders
```

```
ans =
"SimpleStructReader"
```

Import Workspace Data in a Custom Format

Once you register the custom workspace data reader, you can import workspace variables stored using the custom format into the Simulation Data Inspector using the UI or the `Simulink.sdi.createRun` function.

To import data using the UI, open the Simulation Data Inspector. You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector from the MATLAB Command Window. Then, select **Import**.

The Import dialog shows the data in the base workspace that the Simulation Data Inspector is able to import using built-in and registered custom readers. Because the custom reader is registered, the `Line`, `Sine Wave`, and `Square Wave` signals are available for import, while the `lineData`, `sineWave`, and `squareWave` variables are not. Select the data you want to import and select **Import**. The data imports into a run called `Imported_Data`.

To import data from the workspace programmatically, use the `Simulink.sdi.createRun` function.

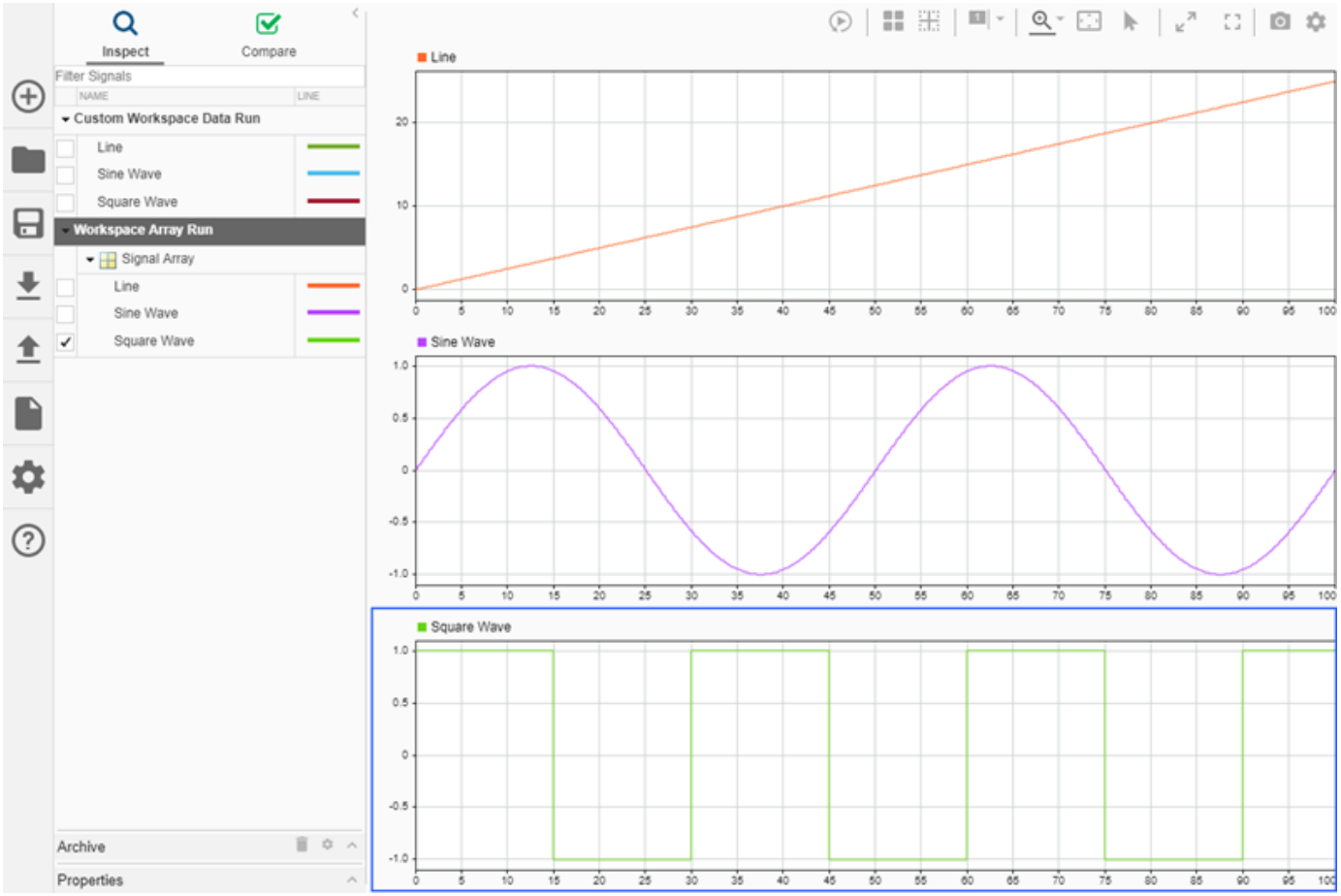
```
Simulink.sdi.createRun('Custom Workspace Data Run','vars',myLineVar,mySineVar,mySquareVar);
```

The custom reader in this example can also import an array of structures. Importing an array of workspace variables rather than importing them individually groups the variables together when you import the data to an existing run. Create an array that contains the `myLineVar`, `mySineVar`, and `mySquareVar` structures, and import the array using the `Simulink.sdi.createRun` function.

```
myVarArray = [myLineVar; mySineVar; mySquareVar];
Simulink.sdi.createRun('Workspace Array Run','vars',myVarArray);
```

Inspect and Analyze Imported Data

After importing data, you can use the Simulation Data Inspector to inspect and analyze the imported data on its own or alongside related simulation data.



See Also

Classes

`io.reader`

Functions

`Simulink.sdi.createRun`

More About

- “View Data in the Simulation Data Inspector” on page 29-2
- “Import Data Using a Custom File Reader” on page 29-17

Import Data Using a Custom File Reader

When you want to visualize data stored in a data or file format the Simulation Data Inspector does not support, you can use the `io.reader` class to write your own custom file reader for the Simulation Data Inspector. This example explains the parts of a class definition for a custom file reader and demonstrates how to register the reader with the Simulation Data Inspector. Open the `ExcelFirstColumnTimeReader.m` file to view the complete class definition.

Write the Class Definition for a Custom Reader

Write a class definition that specifies how your custom reader extracts relevant data and metadata from files and variables that use custom formats. Save the class definition file in a location on the MATLAB™ path.

The custom reader in this example uses the `readtable` function to load data from a Microsoft Excel™ file and uses the first column in the file as time data.

The class definition starts by inheriting from the `io.reader` class, followed by method definitions that return required and relevant data and metadata.

```
classdef ExcelFirstColumnTimeReader < io.reader
```

Every custom reader must define the `getName`, `getTimeValues`, and `getDataValues` methods. Additional methods are available to access certain metadata that might exist in the custom file. The class definition for this example defines the abstract methods as well as the `supportsFile` and `getChildren` methods.

The `supportsFile` method checks the file contents to make sure the file contains signal data.

```
function supported = supportsFile(~,filename)
    try
        t = readtable(filename);
        supported = height(t) > 0 && numel(t.Properties.VariableNames) > 1;
    catch
        supported = false;
    end
end
```

To import multiple signals from a file, treat the data as hierarchical with the file being the top node. The reader uses the `getChildren` method to create an `ExcelFirstColumnTimeReader` object for each signal in the file.

```
function childObj = getChildren(obj)
    childObj = {};
    if isempty(obj.VariableName)
        t = readtable(obj.FileName);
        vars = t.Properties.VariableNames;
        vars(1) = [];
        childObj = cell(size(vars));
        for idx = 1:numel(vars)
            childObj{idx} = ExcelFirstColumnTimeReader;
            childObj{idx}.FileName = obj.FileName;
            childObj{idx}.VariableName = vars{idx};
        end
    end
end
```

The `getTimeValues` method reads the data in the file using the `readtable` function and returns the data in the first column for the Simulation Data Inspector to use as time data.

```
function timeVals = getTimeValues(obj)
    timeVals = [];
    if ~isempty(obj.VariableName)
        t = readtable(obj.FileName);
        timeName = t.Properties.VariableNames{1};
        timeVals = t.(timeName);
    end
end
```

The `getName` method uses the file name as the name for the top-level node of the imported data. Signals are named using the `VariableName` property for the corresponding `ExcelFirstColumnTimeReader` object returned by the `getChildren` method.

```
function retName = getName(obj)
    if isempty(obj.VariableName)
        fullName = obj.FileName;
        [filepath,name,ext] = fileparts(fullName);
        retName = strcat(name,ext);
    else
        retName = obj.VariableName;
    end
end
```

The `getDataValues` method returns the data in each column that has data besides the first as signal data. Data for each signal is accessed using the `VariableName` property for the corresponding object returned by the `getChildren` method.

```
function dataVals = getDataValues(obj)
    dataVals = [];
    if ~isempty(obj.VariableName)
        t = readtable(obj.FileName);
        dataVals = t.(obj.VariableName);
    end
end
```

Register a Custom Reader for the Simulation Data Inspector

After you write the class definition file for the custom data reader, you need to register the reader with the Simulation Data Inspector before you can use it to import data. The Simulation Data Inspector does not store registered readers between MATLAB sessions, so you need to register a custom reader at the start of each new MATLAB session. You can register a custom reader to read data from the workspace or to read data from a file. To register the file reader in this example, use the `registerFileReader` method.

```
registerFileReader(ExcelFirstColumnTimeReader,[".xlsx" "xls"]);
```

To confirm that the file reader is registered, use the `io.reader.getRegisteredFileReaders` method.

```
io.reader.getRegisteredFileReaders
```

```
ans = 1x2 string
    "ExcelFirstColumnTimeReader"    "ExcelFirstColumnTimeReader"
```

Import Data from a File in a Custom Format

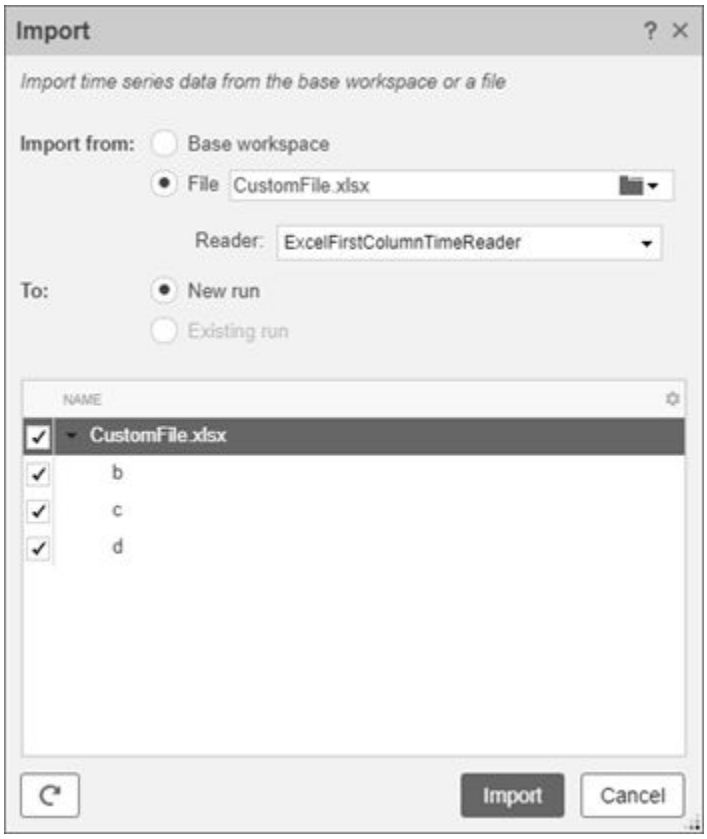
Once you register the custom file reader, you can import data from a file in a custom format using the Simulation Data Inspector UI or using the `Simulink.sdi.createRun` function. This example imports a simple data set from a file that contains four columns of data. The custom file reader in this example always loads the first column, in this case `a`, as time data.

	A	B	C	D
1	a	b	c	d
2	1	6	6	1
3	2	5	4	7
4	3	4	2	6
5	4	3	6	1
6	5	2	7	3
7	6	1	2	4

To import the data using the UI, open the Simulation Data Inspector. You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector from the MATLAB Command Window. Then, click the **Import** button.

In the Import dialog, select the **File** option and import the data from the file into a new run. Click the folder to browse the file system and select the file you want to import. The file for this example is called `CustomFile.xlsx`.

After you select a file with an extension that corresponds to one or more registered custom readers, an option to select the reader you want to use appears in the dialog. In this example, you can choose between the built-in Microsoft Excel file reader or the custom reader written for this example. By default, a custom reader is selected when one is available for the extension of the selected file.



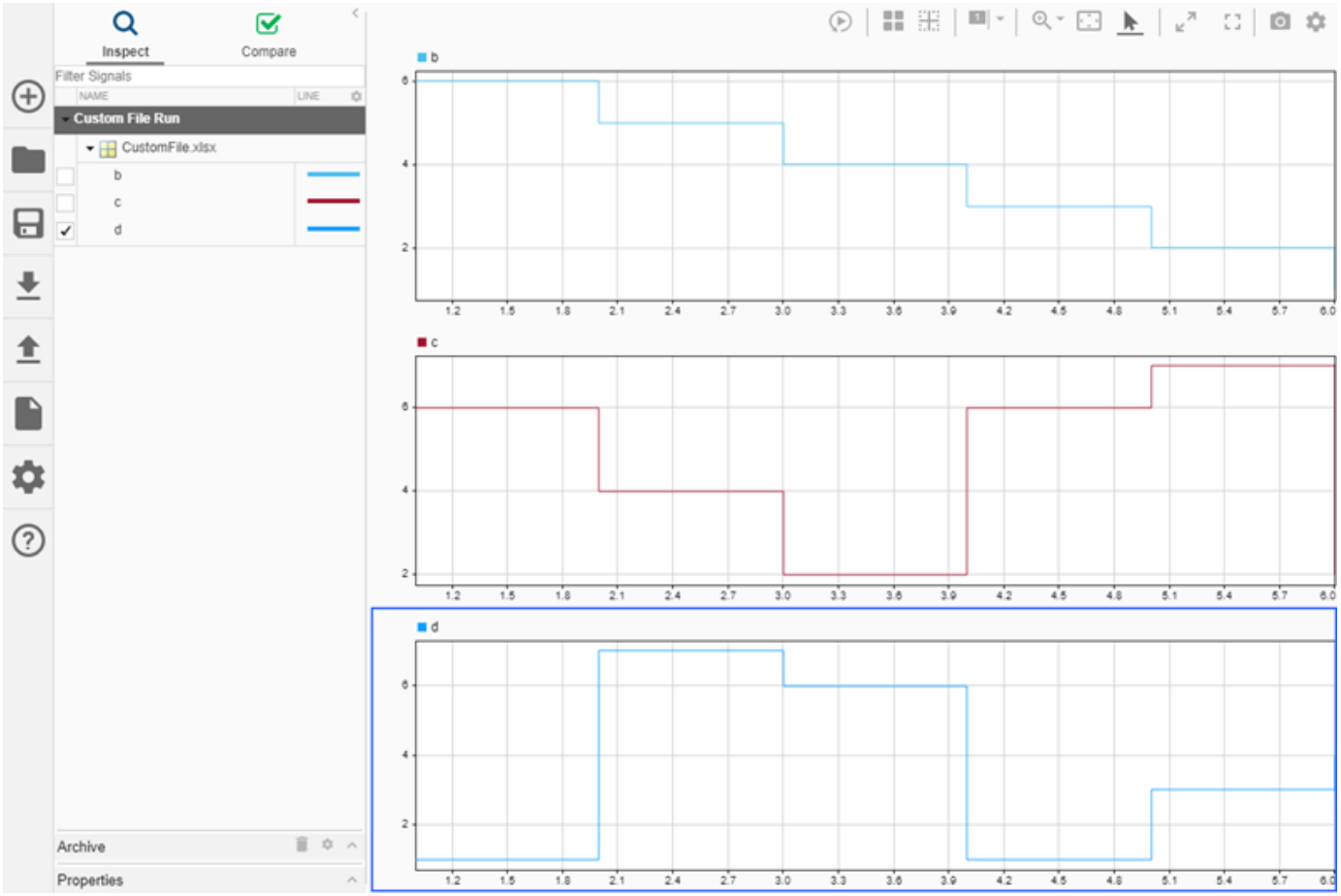
You can choose which signals you want to import from the file. After making your selection, click **Import**. The data is imported to a new run called `Imported_Data`.

To import data programmatically, you can use the `Simulink.sdi.createRun` function. The Simulation Data Inspector has a built-in reader for Microsoft Excel files. You can specify which reader to use to import the data in the call to the `Simulink.sdi.createRun` function. When you do not specify the reader you want to use to import the data, the Simulation Data Inspector uses the first custom reader that supports the file extension.

```
Simulink.sdi.createRun('Custom File Run','file','CustomFile.xlsx','ExcelFirstColumnTimeReader');
```

Inspect and Analyze Imported Data

After importing your data, you can use the Simulation Data Inspector to inspect and analyze the imported data on its own or alongside related simulation data.



See Also

Classes
`io.reader`

Functions
`Simulink.sdi.createRun`

More About

- “View Data in the Simulation Data Inspector” on page 29-2
- “Import Workspace Variables Using a Custom Data Reader” on page 29-11

View and Replay Map Data

You can add a map to your layout in the Simulation Data Inspector to view routes of longitude and latitude data. The map data is synchronized with signal data in the Simulation Data Inspector, allowing you to analyze signal activity throughout the route. To analyze the relationship between the route and signal values, you can replay the data. When you replay data in the Simulation Data Inspector, synchronized cursors and markers move across the plots in the view.

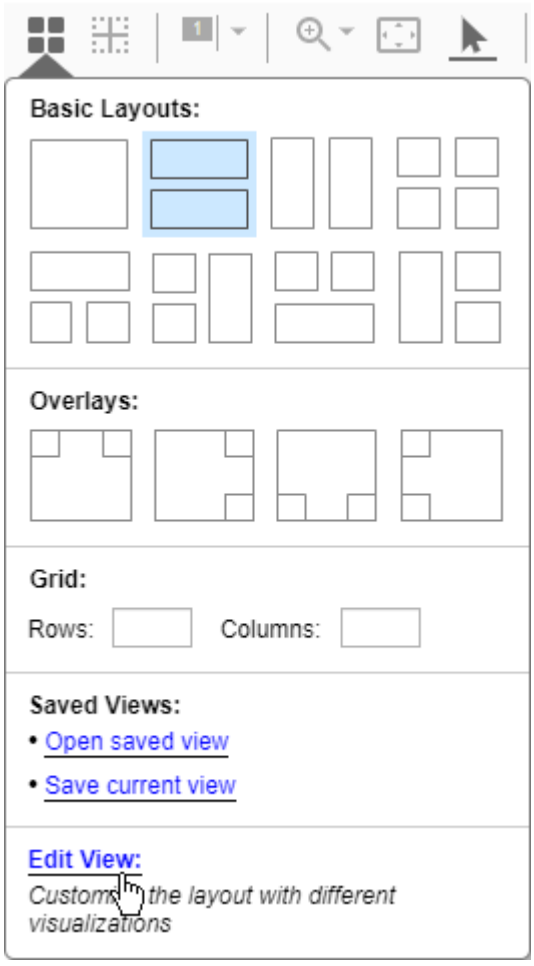
To view map data in the Simulation Data Inspector, you must have an active internet connection.

If your model includes latitude and longitude signals, you can log data to the Simulation Data Inspector and visualize it during simulation. You can also import the data from the workspace or a file using the instructions in “Import Data from the Workspace or a File” on page 29-3 or the `Simulink.sdi.createRun` function. For this example, import a file of recorded map and speed data using `Simulink.sdi.createRun`. Then, open the Simulation Data Inspector.

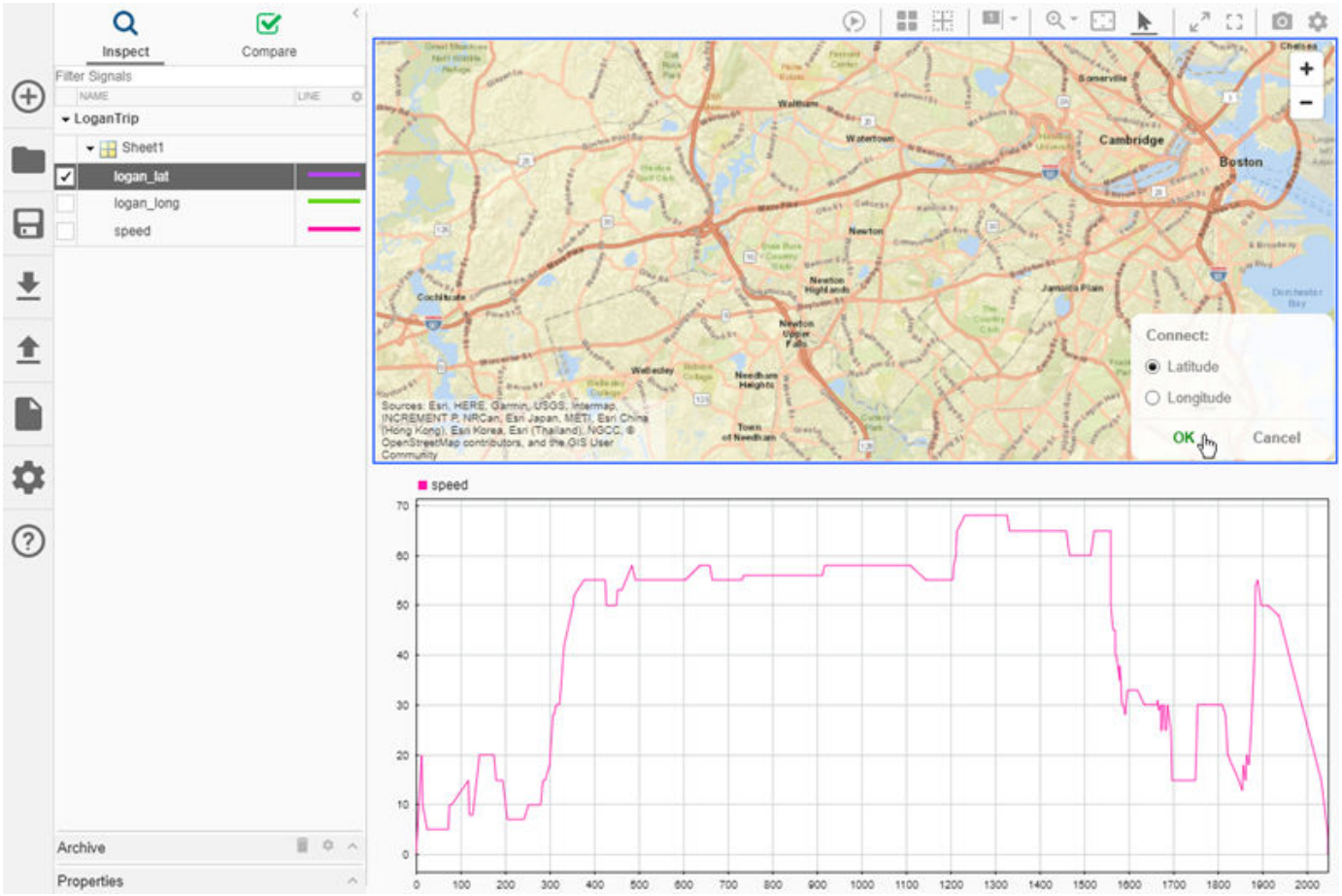
```
mapRun = Simulink.sdi.createRun('LoganTrip','file','logan_ah_gps_data.xlsx');
Simulink.sdi.view
```

View Map Data

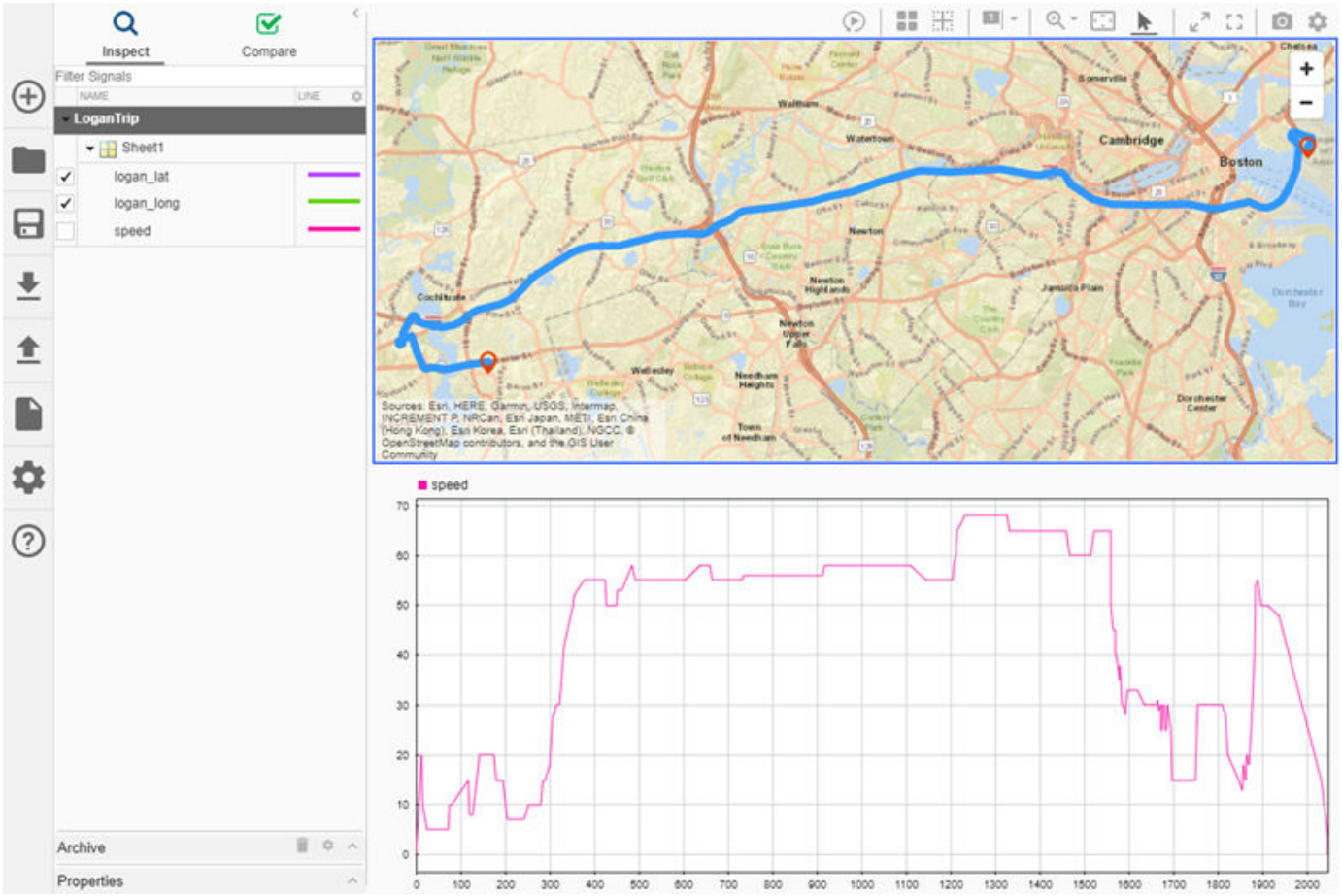
Configure the plot area in a `2x1` layout so you can view the speed data and map data together. To plot the speed data on the lower plot, select the plot and then select the check box next to the `speed` signal. To add a map to the layout, open the **Visualization Gallery** by clicking **Edit View** on the **Layout** menu.



From the visualization gallery, drag a map onto the top subplot in the layout. Check the `logan_lat` signal and click **OK** on the dialog in the bottom-right of the map to use the signal to specify the latitude data on the map.

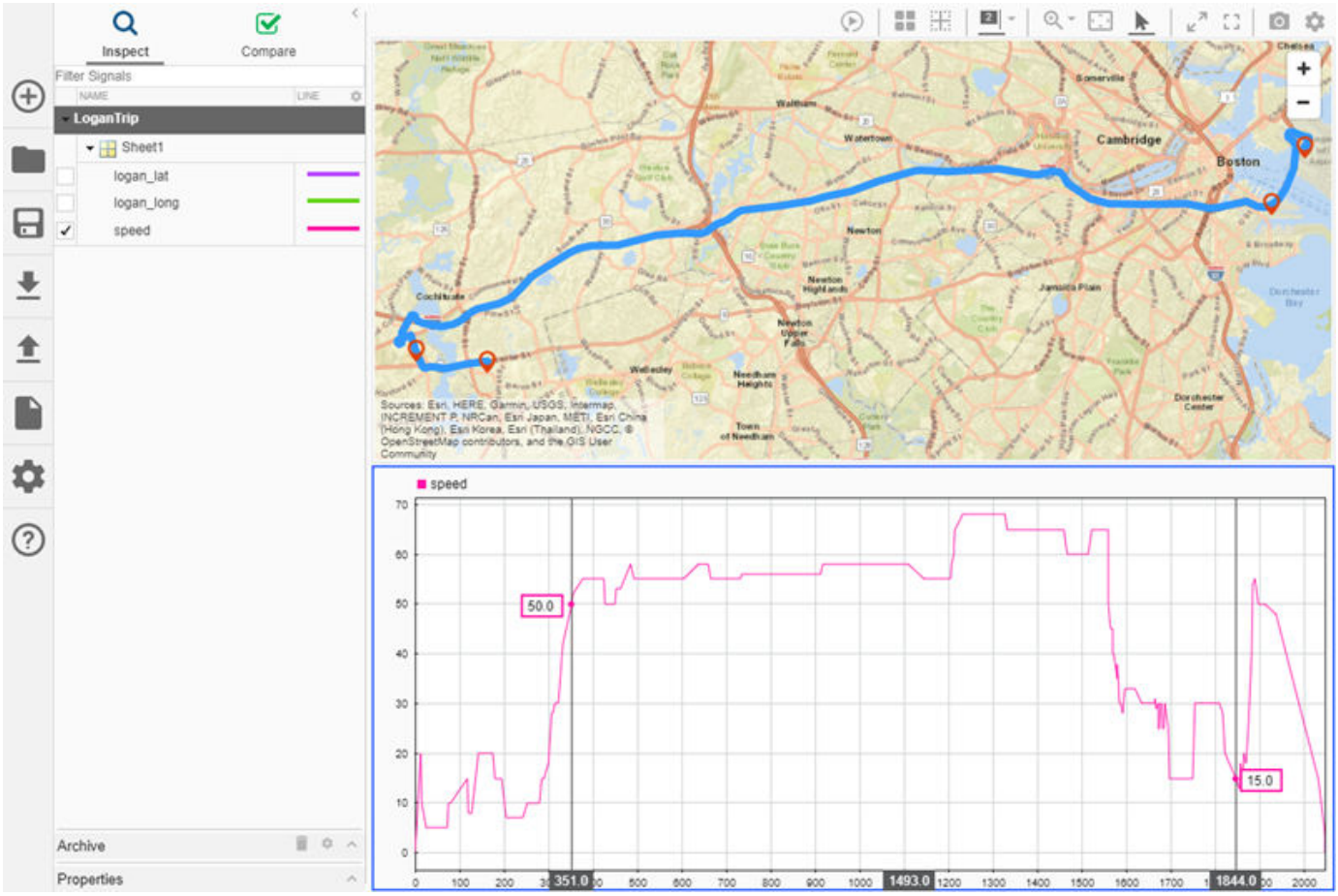


Then, check the `logan_long` signal and click **OK** to use the signal as longitude data on the map.



You can also plot map data by selecting multiple signals while holding **Ctrl** and then dragging those signals onto the map. The menu in the lower-right shows drop-downs you can use to specify which signal to use for the latitude data and which to use for the longitude data.

You can add cursors to the plots to analyze the relationship between the `speed` signal and the route shown on the map. To add two cursors to the layout, click the drop-down arrow on the cursors button in the toolbar and select **Two Cursors**. Move the cursor on the left. The marker on the right of the map moves as you move the left cursor because the route starts in Boston and moves East to West with time.



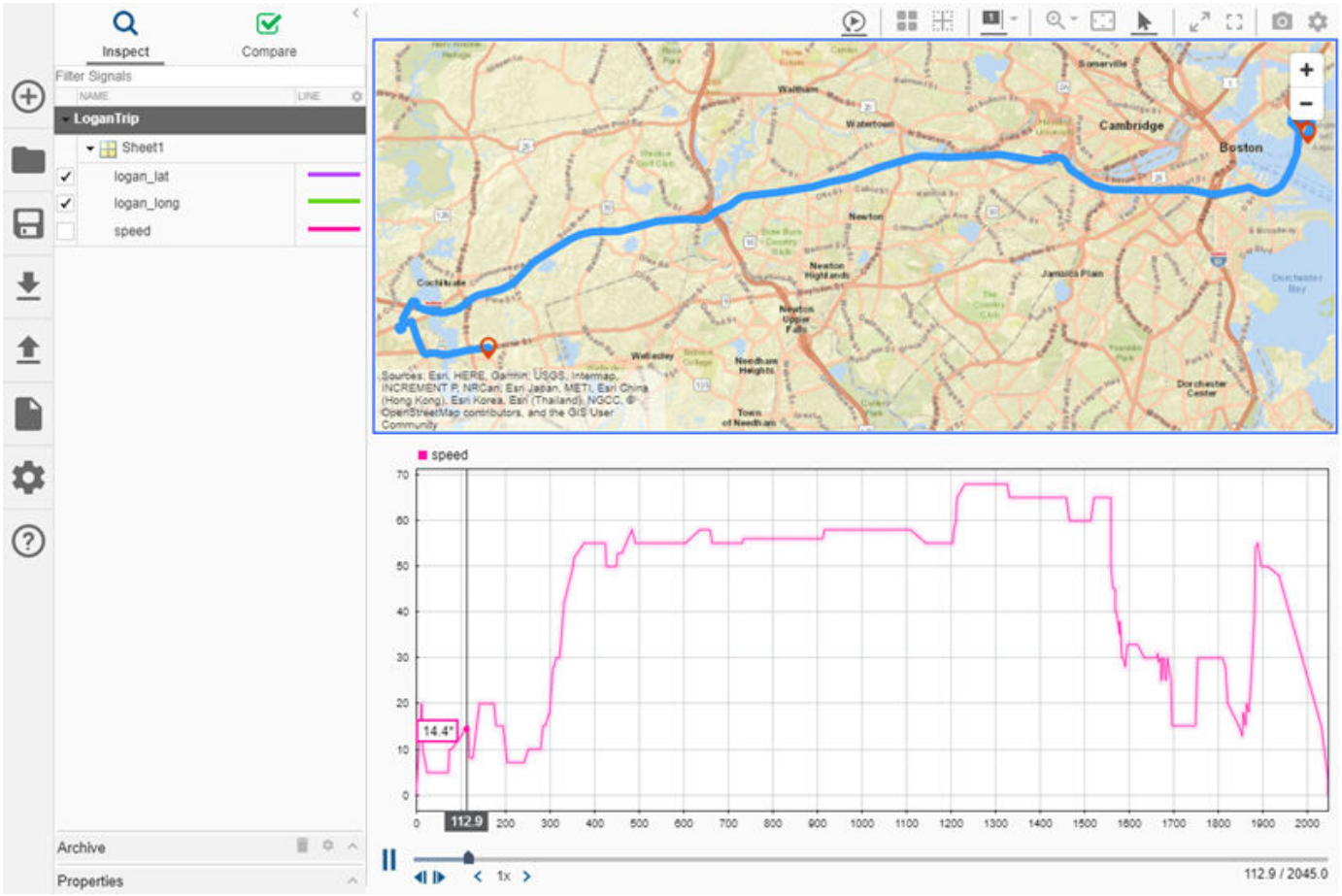
To remove the cursors from the plot, click the **Show/hide data cursors** button.

Replay Map Data

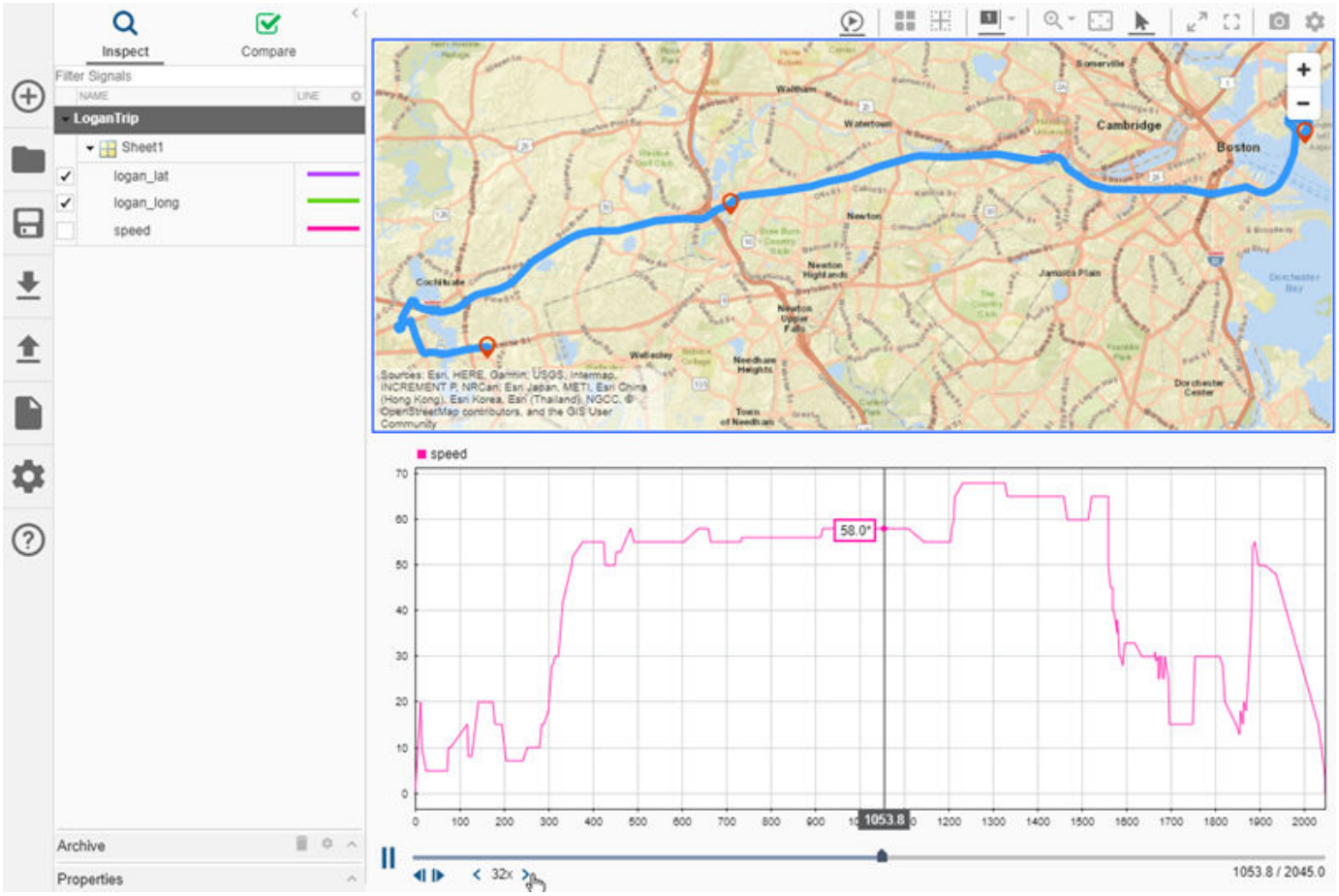
You can replay data in the Simulation Data Inspector to automate moving cursors across the visualizations in your view. To add the replay controls to the view, click the **Show/hide replay controls** button. 

Then, press the **Replay** button. 

A data marker on the map and a cursor on the time plot sweep synchronously across the screen. The synchronous replay facilitates an intuitive understanding of the relationship between speed and position throughout the trip.



By default, the Simulation Data Inspector replays data at one second per second, meaning that the cursor moves through one second of data in one second of clock time. The data for this trip spans approximately 34 minutes. You can increase the replay speed using the arrow to the right of the label or by clicking the label and typing the desired speed. Increase the replay speed to 32x using the arrow.



For another example, see “Replay Data in the Simulation Data Inspector” on page 29-125.

Related Topics

“View Data in the Simulation Data Inspector” on page 29-2

“Create Plots Using the Simulation Data Inspector” on page 29-94

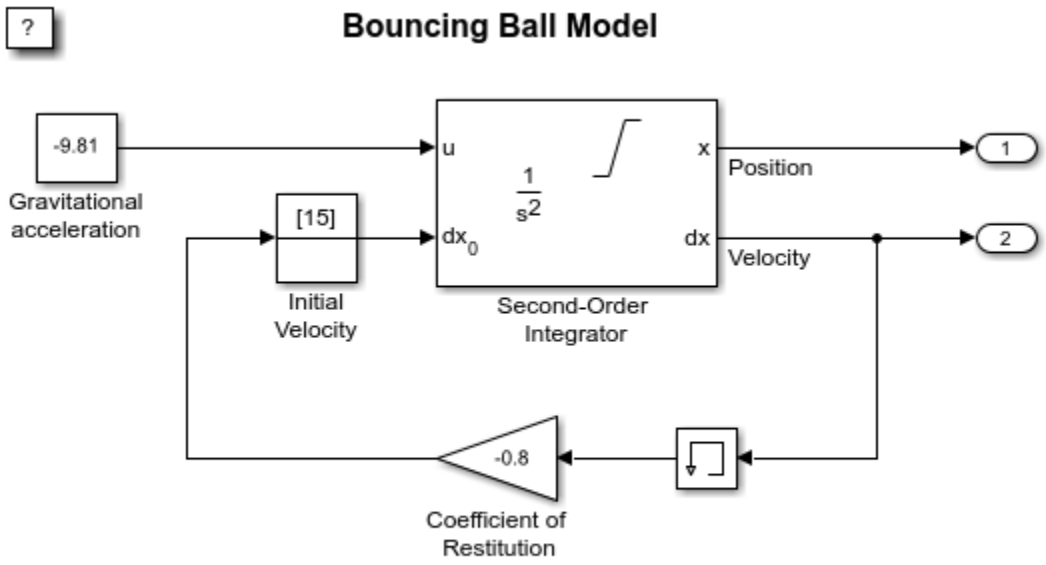
Visualize Simulation Data on an XY Plot

When you log simulation data in a model, you can view the simulation results on an XY visualization in the Simulation Data Inspector. Then, you can replay the simulation to animate the relationships between the signals in your model. This example simulates a model and plots data logged in the simulation using time plot and XY visualizations in the Simulation Data Inspector. The example also shows how to inspect the plotted data using replay controls and cursors.

Simulate the Model and Open the Simulation Data Inspector

The model in this example simulates the dynamics of a bouncing ball, logging the velocity and position of the ball as outputs.

```
open_system('ex_sldemo_bounce')
```



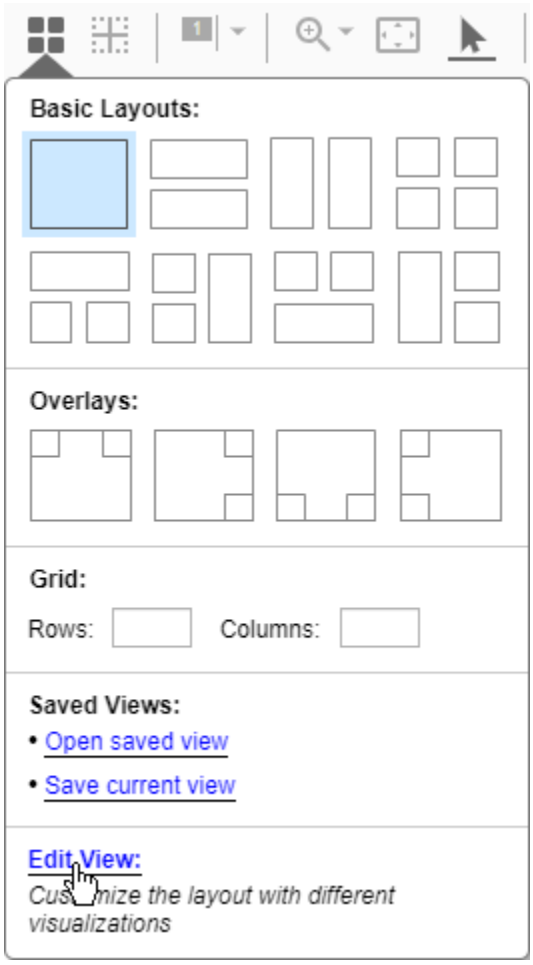
Because the model is configured to log output data, the signals connected to the Outport blocks log to the workspace and the Simulation Data Inspector. Simulate the model and open the Simulation Data Inspector.

```
sim('ex_sldemo_bounce');
Simulink.sdi.view
```

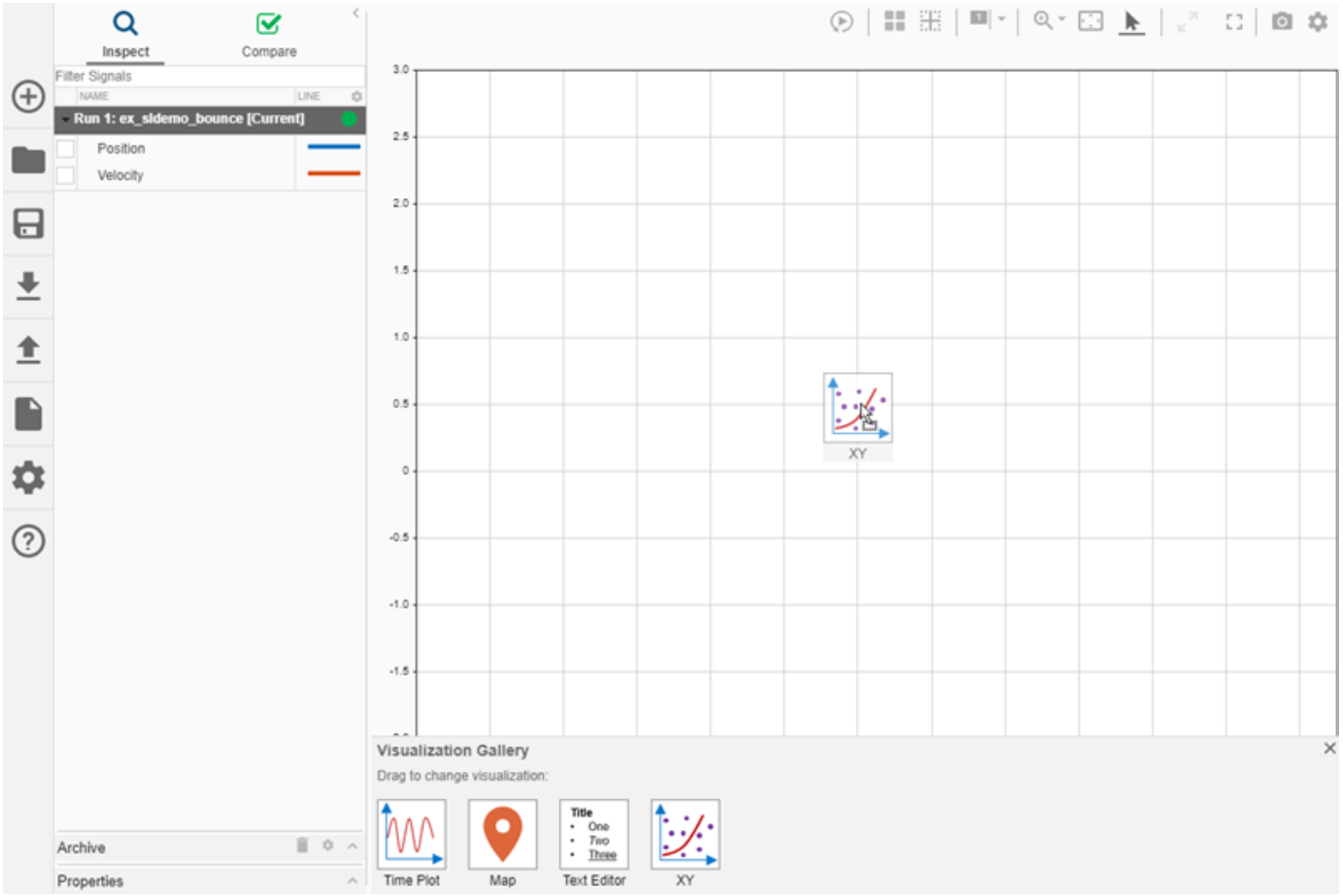
Plot Data on an XY Visualization

To plot the data on an XY visualization, you need to add the visualization to the layout. By default, the Simulation Data Inspector uses time plot visualizations for each subplot in the layout.

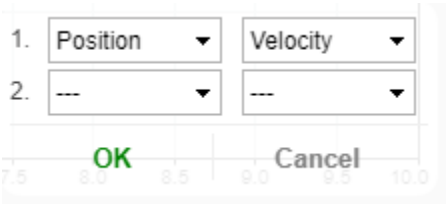
To add an XY visualization to your layout, open the **Layout** menu and click **Edit View** to open the **Visualization Gallery**.



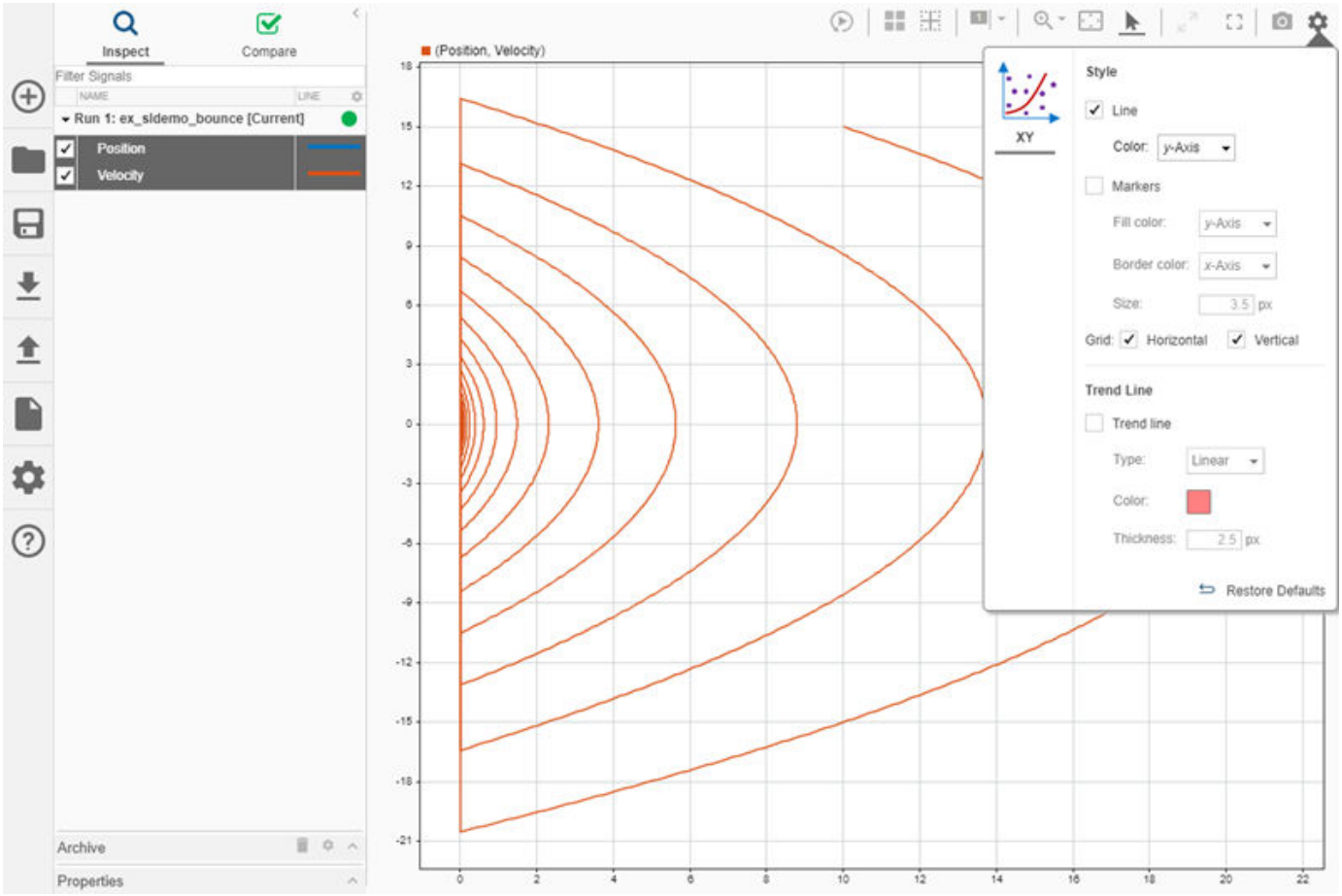
From the **Visualization Gallery**, drag and drop the **XY** icon onto the plot.



To plot the signals on the XY visualization, select both signals then drag and drop them onto the plot. You can specify which signal to use as the *x* data and which to use as the *y* data. For this example, use `Position` as the *x* data and `Velocity` for the *y* data.



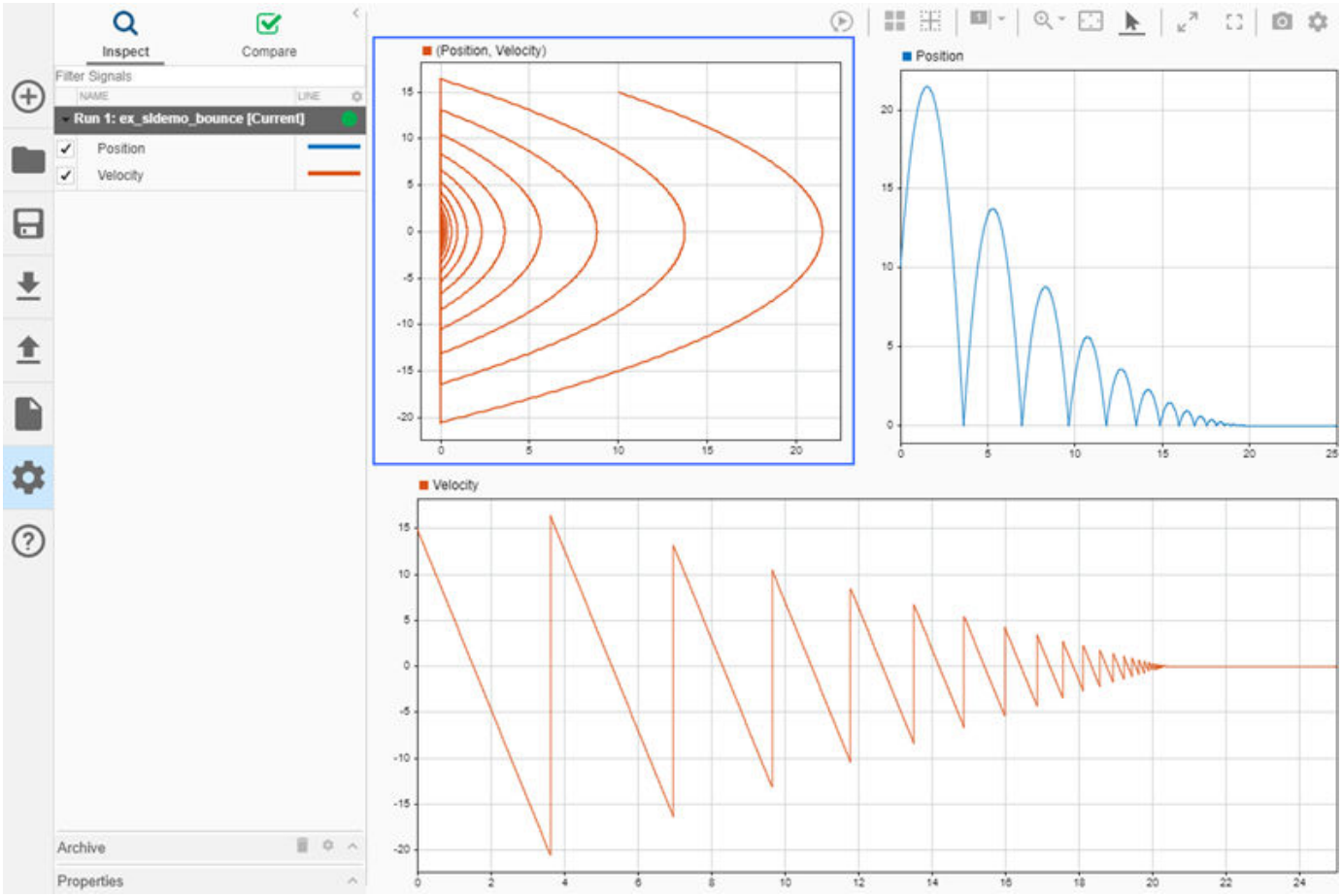
You can customize the appearance of the XY visualization by opening the **Visualization Settings**. The line and marker colors match the color of the signal that provides the *x* data or the signal that provides the *y* data. By default, the line uses the *y*-axis signal color.



Add Time Plots and Inspect the Data

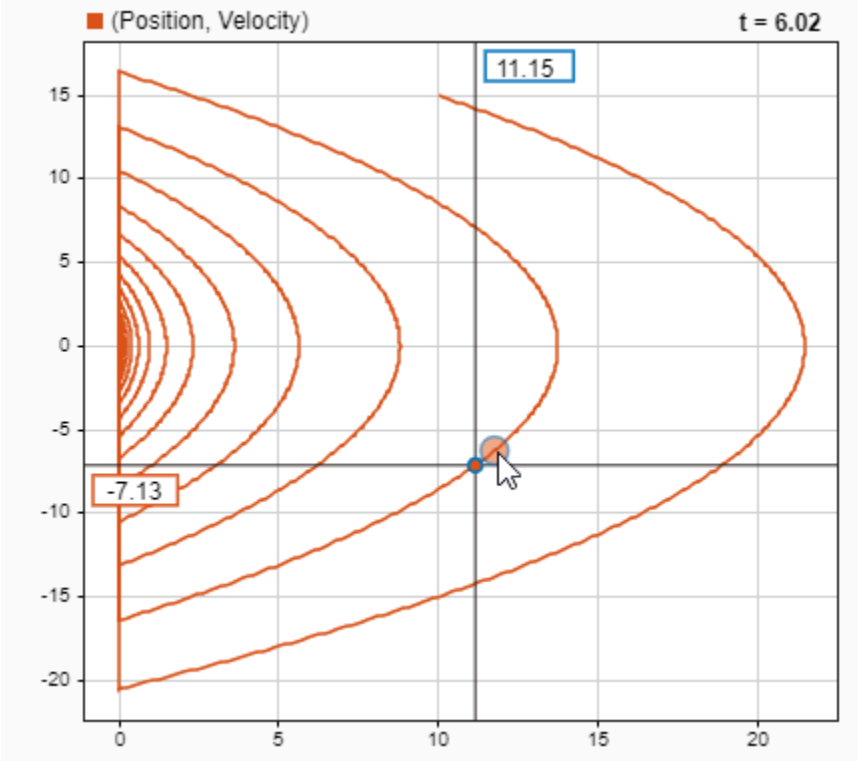
You can include multiple visualizations in a layout in the Simulation Data Inspector. For example, you can choose to use a layout with three subplots so you can see each signal on a time plot alongside the XY visualization.

In the **Layout** menu, choose the layout with two subplots on top of a third from the **Basic Layouts** section. Then, plot the `Position` signal in the upper right time plot, and plot the `Velocity` signal on the bottom time plot.



To inspect the data, add a cursor. In the XY visualization, the vertical line of the cursor shows the *x*-axis value, and the horizontal line shows the *y*-axis value. The time corresponding to the point is displayed in the upper-right of the plot.

Move the cursor in the XY visualization along the plotted line. As you move the cursor, the next data point for the cursor to snap to is highlighted. You can also move the cursor in the XY plot using the arrow keys on your keyboard or by pausing on a point on the line and clicking the highlighted point.

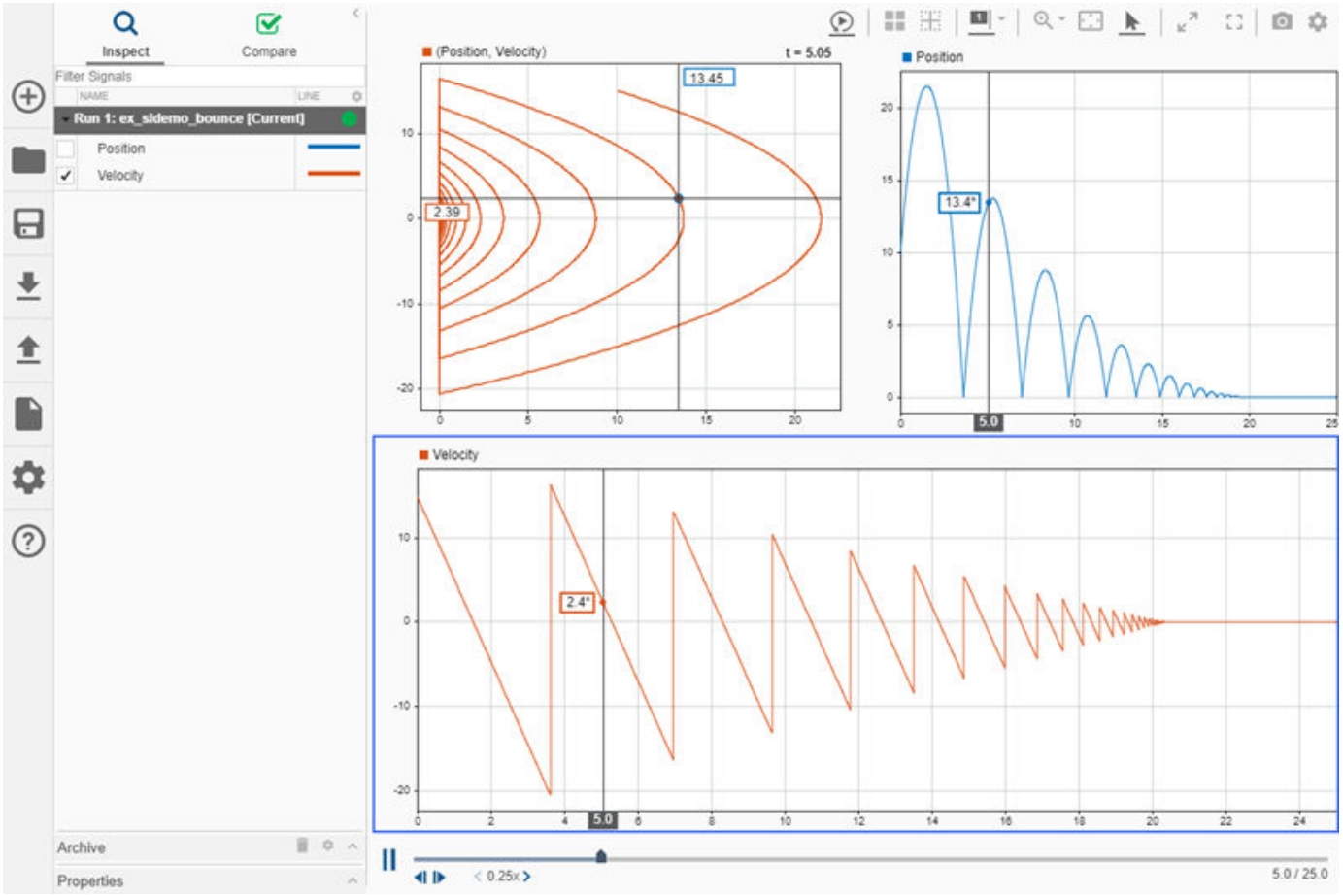


When you drag a cursor in a time plot, the cursor in the XY visualization moves synchronously through the plotted data. The XY visualization can only have one cursor. When you add two cursors to the layout, the XY cursor moves with the left cursor in the time plot.

Replay the Data

Now that you have a comprehensive visualization of the simulation data, replaying the data can help you understand the relationship between the signals. When you replay data in the Simulation Data Inspector, animated cursors sweep through the logged simulation data from the start time to the end time. Add the replay controls to the view by clicking the **Show/hide replay controls** button.

You can control the speed of the replay and pause at any time. By default, the Simulation Data Inspector replays data at one second per second, meaning the cursor moves through one second of data in one second of clock time. The data in this example spans 25 seconds. Slow the replay speed by clicking the arrow to the left of the label.



For more information about using replay controls, see “Replay Data in the Simulation Data Inspector” on page 29-125.

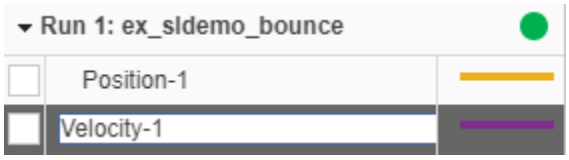
Analyze Data from Multiple Simulations

To analyze how changes in simulation parameters affect the data, you can plot multiple series on an XY visualization. In the model or MATLAB™ Command Window, change the **Initial value** parameter of the Initial Velocity block to 25 and simulate the model.

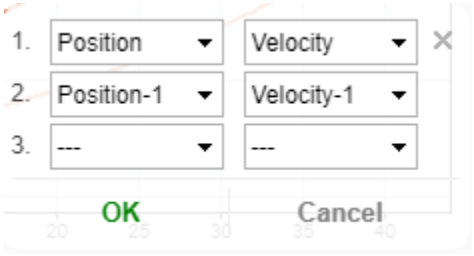
```
set_param('ex_sldemo_bounce/Initial Velocity','Value','25')
sim('ex_sldemo_bounce');
```

The Simulation Data Inspector moves the first run to the archive and transfers the view to the new run. Drag the first run from the archive into the work area.

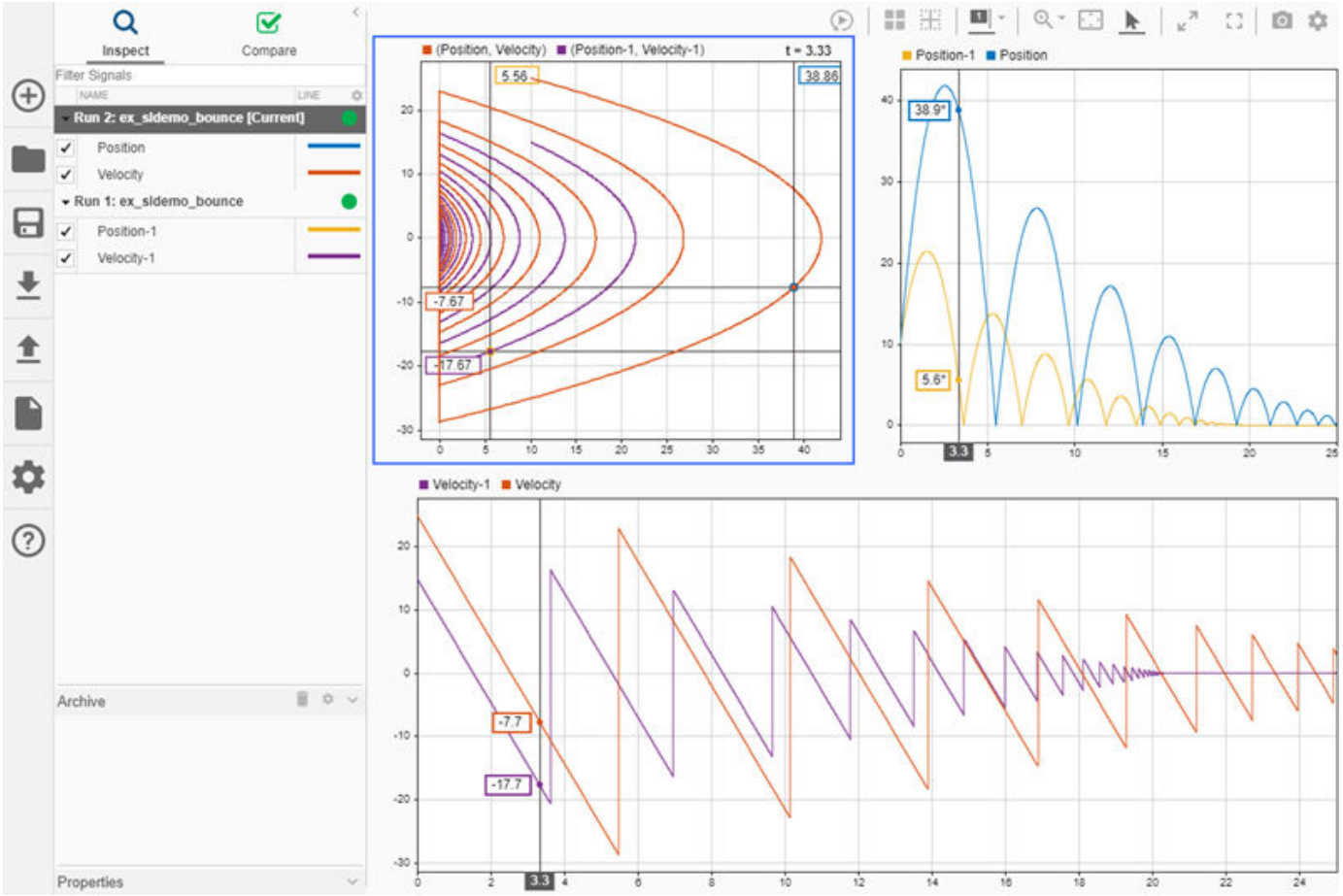
The signals in both runs have the same names and the same signal colors. Before you add data from the first run to the plots, change the signal names and colors. For example, you can rename the signals from the first run `Position-1` and `Velocity-1`. To modify a signal color, click the line representation in the table, select a new color, and click **Set**. To modify the signal names, double-click the name in the table and enter the new name.



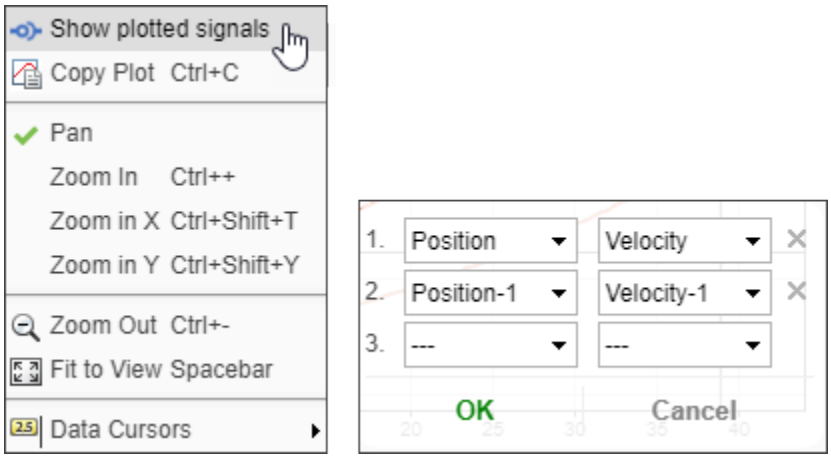
Now, add the data from the first run to the plots in the layout. Plot the signals from the first run on the time plots. To add another series to the XY visualization, select both signals and drag them onto the plot. The connection dialog shows the first series that is already plotted. Select the `Position-1` signal as the *x*-data for series 2 and `Velocity-1` as the series 2 *y*-data. Then click **OK**.



When you add multiple series to an XY visualization, each series gets a cursor. All cursors on the XY visualization move synchronously, so all signal values displayed on the cursors correspond to the same time.



You can manage the signals plotted on an XY visualization using the context menu. Right-click the XY plot area and select `Show plotted signals` to bring up the connection dialog. From the connection dialog, you can remove series from the plot or modify the signals that provide the *x*-data and *y*-data for each series.



Analyze Data Using the XY Visualization

In the Simulation Data Inspector, you can plot time series data on an **XY** visualization to analyze relationships between signals. This example demonstrates how to view and analyze data on an **XY** visualization using trend lines and example data. The example data file attached includes more data than the example covers. You can apply the steps covered in the following sections to each set of signals to use each trend line option.

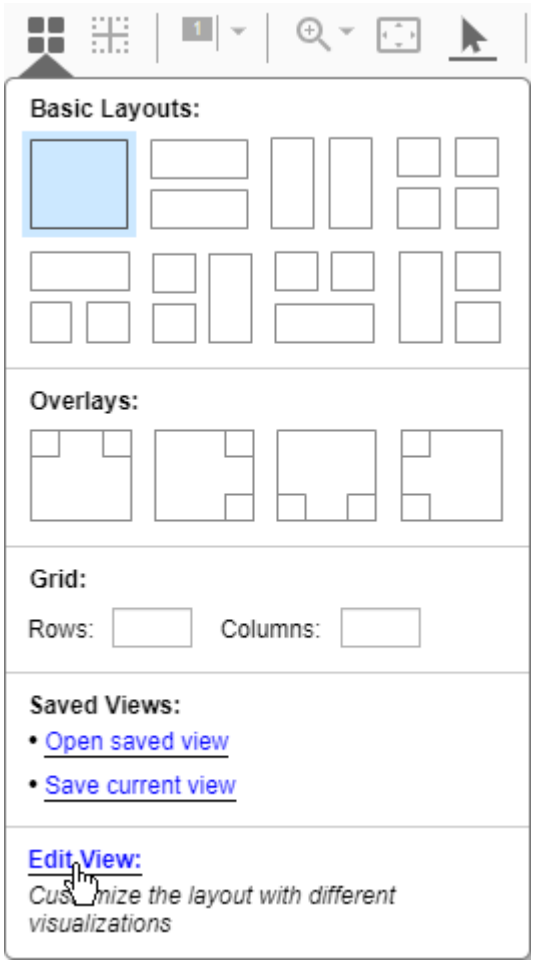
Import and Plot Data on the XY Visualization

The data for this example is stored in a MAT-file. You can import the data through the UI, or you can use the `Simulink.sdi.createRun` function. Then, open the Simulation Data Inspector to view the data.

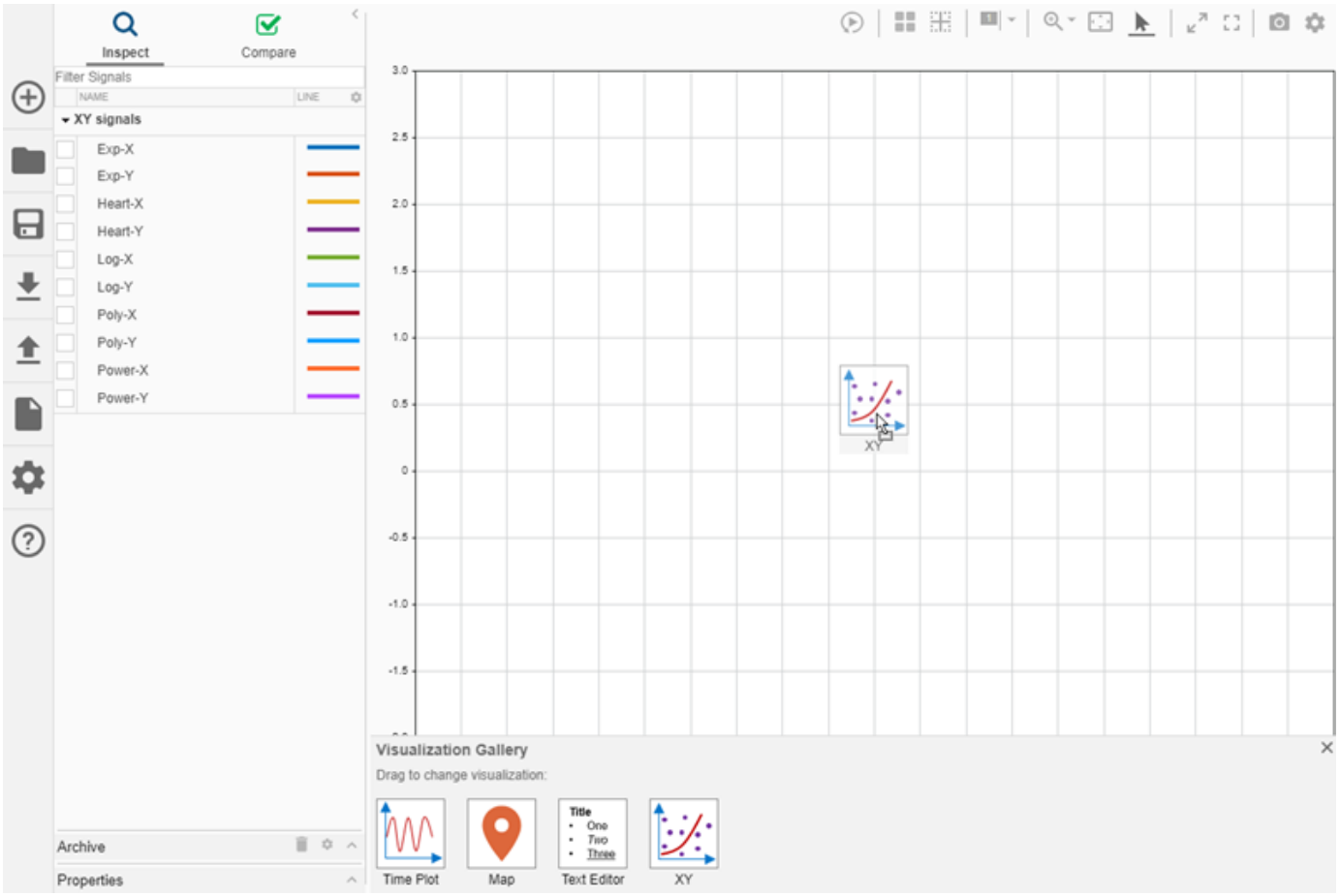
```
xyRunID = Simulink.sdi.createRun('XY signals','file','ex_xy_data.mat');
Simulink.sdi.view
```

To plot the data on an **XY** visualization, you need to add the visualization to the layout. By default, the Simulation Data Inspector uses **Time Plot** visualizations for each subplot in the layout.

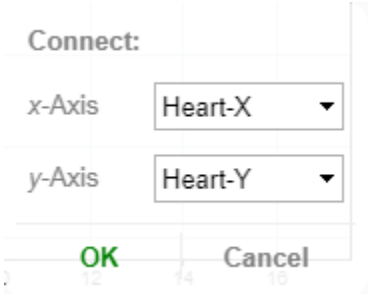
To add an **XY** visualization to your layout, open the **Visualization Gallery** by selecting **Edit View** from the **Layouts** menu.



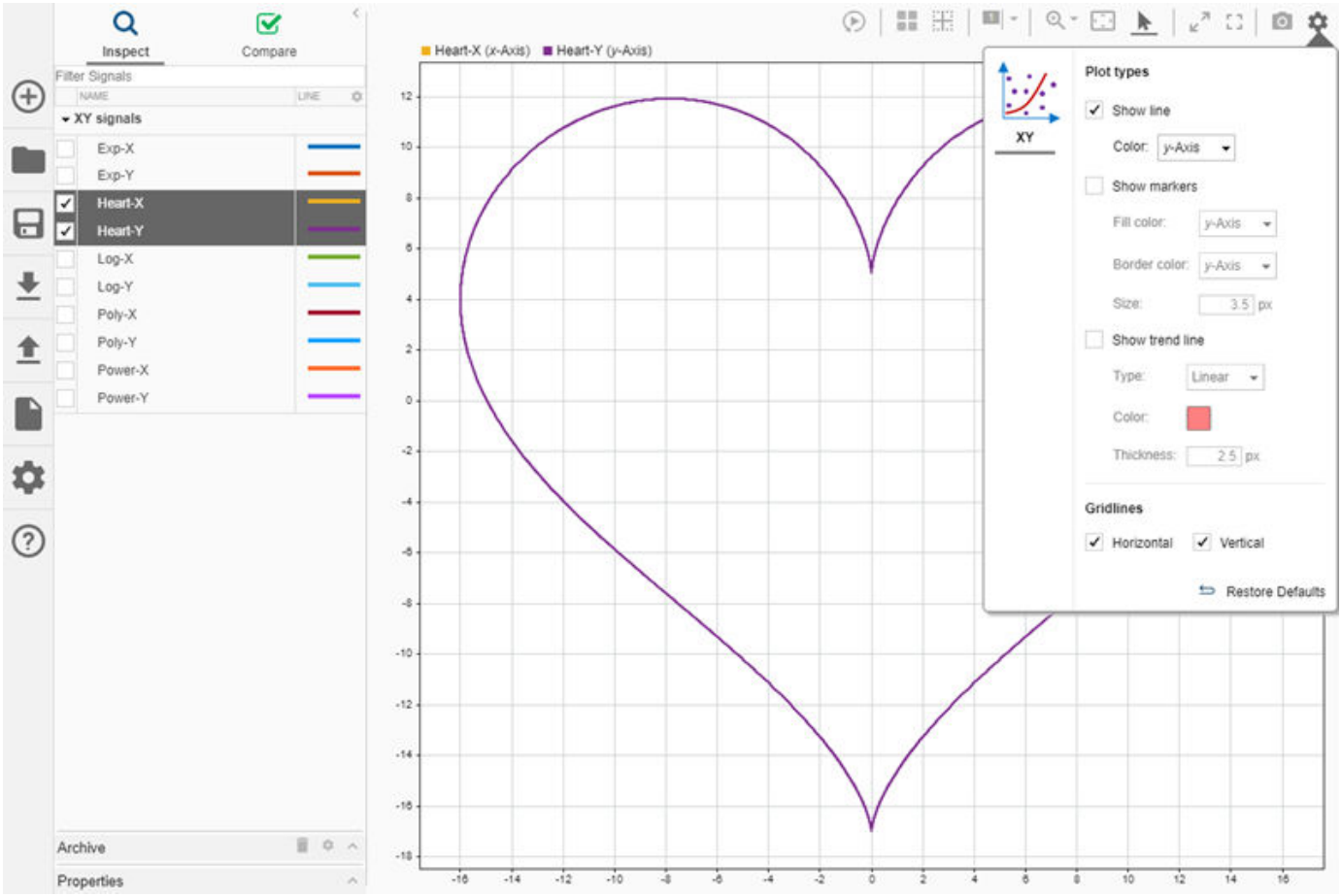
From the **Visualization Gallery**, drag and drop the **XY** icon onto the plot.



To plot signals on the **XY** visualization, select two signals and drag them onto the plot. For example, drag and drop the `Heart-X` and `Heart-Y` signals onto the plot. Specify which signal to use as *x*-data and which to use as *y*-data in the dialog that appears in the lower-right of the plot.



You can customize the appearance of the **XY** visualization using the **XY Settings**. The line and marker colors can match the color for the signal providing the *x* data or for the signal that provides the *y* data. By default, the line and markers use the **y-Axis** signal color. Specified settings apply for all **XY** visualizations in your layout.



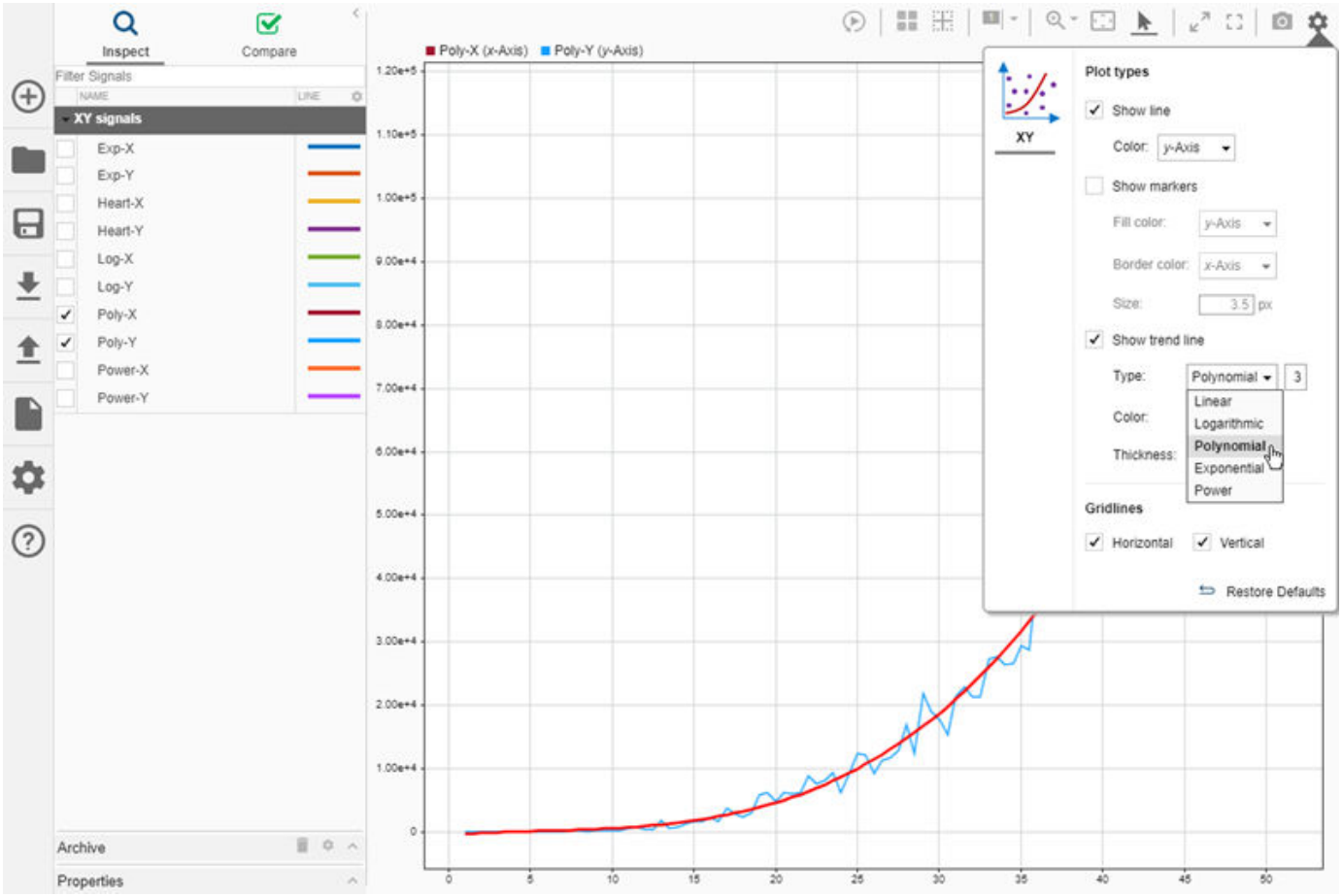
Analyze XY Data Using a Trend Line

You can add a trend line to an **XY** visualization to analyze the relationship between the *x*- and *y*-data. The trend line is most useful when the relationship can be expressed as a function. The `Heart-X` and `Heart-Y` data plotted in the prior section is not well-suited for trend line analysis because the *y*-data is not well correlated with the *x*-data. Trend line analysis works well when the relationships between the *x* and *y* data can be expressed as a function.

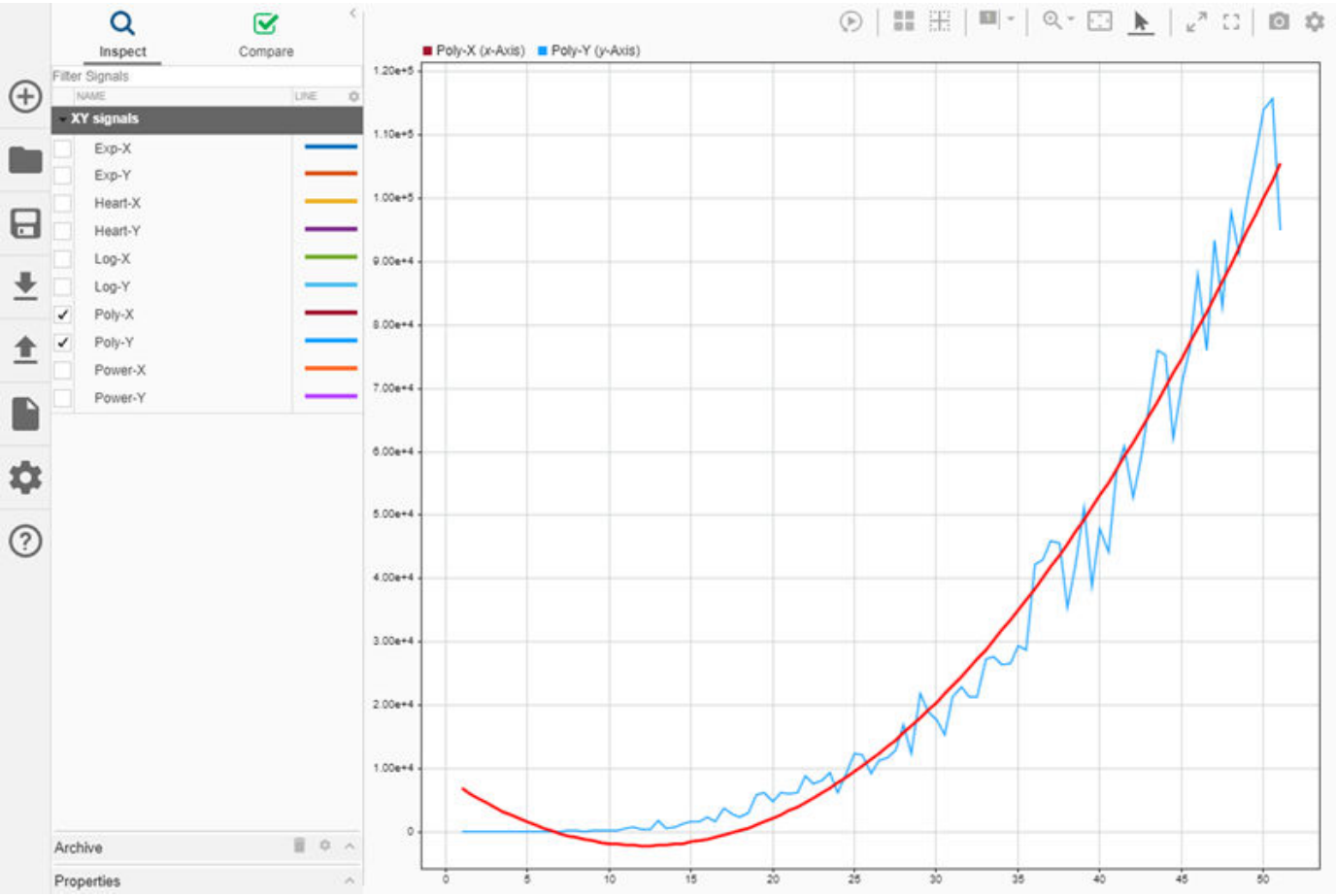
The example data includes *x*- and *y*-data well-suited for each available type of trend line. You can try plotting each pair of signals to see how each trend line helps you analyze the data. For example, plot the `Poly-X` and `Poly-Y` signals on the **XY** visualization:

1. Select both signals.
2. Drag and drop the signals onto the plot.
3. Select `Poly-X` for **x-Axis** and `Poly-Y` for **y-Axis** in the drop-down menus on the dialog. You can only plot one pair of signals on the **XY** visualization.
4. Click **OK**.

To add a trend line, open the **XY Settings** and select **Show trend line**. The default trend line type is **Linear**. Select `Polynomial` from the **Type** drop-down.



By default, the Simulation Data Inspector computes the trend line in the form of a third-order polynomial. You can use the text box next to the selected **Type** to specify the order of the polynomial, between 2 and 6. Modify the order of the trend line polynomial to see how it affects the fit of the trend line. For example, when you change to use a second-order polynomial, the trend line does not fit the start of the data as well.



Try out other trend line options to see how they fit the `Poly-X` and `Poly-Y` data. You can add these types of trend line to the **XY** visualization:

- **Linear**: The trend line equation is of the form $y = mx + b$.
- **Logarithmic**: The trend line equation is of the form $y = a\ln(x) + b$. The x-data must not contain `0` or negative values.
- **Polynomial**: The trend line equation is of the form $y = b_6x^6 \ldots + b_2x^2 + b_1x + a$, where the number of terms is determined by the specified order.
- **Exponential**: The trend line equation is of the form $y = ae^{bx}$. The y-data must not contain `0` or negative values.
- **Power**: The trend line equation is of the form $y = ax^b$. The x- and y-data must not contain `0` or negative values.

Try plotting other pairs of signals in the **XY** visualization and explore how each trend line option fits each data set.

Microsoft Excel Import and Export Format

Using the Simulation Data Inspector or Simulink Test, you can import data from a Microsoft Excel file or export data to a Microsoft Excel file. The tools use the same import file format, so you can use the same Microsoft Excel file with both.

Tip When the format of the data in your Microsoft Excel file does not match the specification in this topic, you can write your own file reader to import the data using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the Microsoft Excel file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

	A	B	C	D
1	time	signal1	signal2	signal3
2	0	1	1	4
3	1	2	4	8
4	2	3	9	15
5	3	3	9	16
6	3	4	16	23
7	4	5	25	42

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values imported from the Microsoft Excel file render as missing data in the Simulation Data Inspector. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2	0	1	1	0	4
3	1	2	4	2	8
4	2	3	9	3	15
5	3	3	9	5	16
6	3	4	16		
7	4	5	25		

Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

The file can include metadata for signals such as data type, units, and interpolation method. Metadata for each signal is listed in rows between the signal names and the signal data. You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Label each piece of metadata according to this table. The table also indicates which tools and operations support each piece of metadata.

Property Descriptions

Signal Property	Label	Values	Simulation Data Inspector Import	Simulation Data Inspector Export	Simulink Test Import and Export
Data type	`Type:`	Built-in data type.	**Supported**	**Supported**	**Supported**
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.	**Supported**	**Supported**	**Supported**
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.	**Supported**	**Supported**	**Supported**
Synchronization method	`Sync:`	`union` or `intersection`.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Relative tolerance	`RelTol:`	Percentage, represented as a decimal. For example, `RelTol: 0.1` specifies a 10% relative tolerance.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Absolute tolerance	`AbsTol:`	Numeric value.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Time tolerance	`TimeTol:`	Numeric value, in seconds.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**

Signal Property	Label	Values	Simulation Data Inspector Import	Simulation Data Inspector Export	Simulink Test Import and Export
Leading tolerance	`LeadingTol:`	Numeric value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Lagging tolerance	`LaggingTol:`	Numeric Value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Block Path	`BlockPath:`	Path to the block that generated the signal.	**Supported**	**Supported**	**Supported**
Port Index	`PortIndex:`	Integer.	**Supported**	**Supported**	**Supported**

When an imported file does not specify signal metadata, `double` data type, `linear` interpolation, and `union` synchronization are used.

User-Defined Data Types

In addition to built-in data types, you can use other labels in place of the `DataType:` label to specify fixed-point, enumerated, alias, and bus data types.

Property Descriptions

Data Type	Label	Values	Simulation Data Inspector Import	Simulation Data Inspector Export	Simulink Test Import and Export
Enumeration	`Enum:`	Name of the enumeration class.	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*
Alias	`Alias:`	Name of a `Simulink.AliasType` object in the MATLAB workspace.	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*	**Not Supported**	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*
Fixed-point	`Fixdt:`	• `fixdt` constructor. • Name of a `Simulink.NumericType` object in the MATLAB workspace. • Name of a fixed-point data type as described in "Fixed-Point Numbers in Simulink" (Fixed-Point Designer).	**Supported**	**Not Supported**	**Supported**
Bus	`Bus:`	Name of a `Simulink.Bus` object in the MATLAB workspace.	**Supported**	**Not Supported**	**Supported**

When you specify the type using the name of a `Simulink.Bus` object and the object is not in the MATLAB workspace, the data still imports from the file. However, individual signals in the bus use data types described in the file rather than data types defined in the `Simulink.Bus` object.

Complex, Multidimensional, and Bus Signals

You can import and export complex, multidimensional, and bus signals using a Microsoft Excel file. The signal name for a column of data indicates whether that data is part of a complex, multidimensional, or bus signal. Microsoft Excel file import and export do not support array of bus signals.

Multidimensional signal names include index information in parentheses. For example, the signal name for a column might be `signal1(2,3)`. When you import data from a file that includes multidimensional signal data, elements in the data not included in the file take zero sample values with the same data type and complexity as the other elements.

Complex signal data is always in real-imaginary format. Signal names for columns containing complex signal data include `(real)` and `(imag)` to indicate which data each column contains. When you import data from a file that includes imaginary signal data without specifying values for the real component of that signal, the signal values for the real component default to zero.

Multidimensional signals can contain complex data. The signal name includes the indication for the index within the multidimensional signal and the real or imaginary tag. For example, `signal1(1,3)(real)`.

Dots in signal names specify the hierarchy for bus signals. For example:

- `bus.y.a`
- `bus.y.b`
- `bus.x`

	A	B	C	D	E
1	time	bus.y.a	bus.y.b	time	bus.x
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Function-Call Signals

Signal data specified in columns before the first time column is imported as one or more function-call signals. The data in the column specifies the times at which the function-call signal was enabled. The imported signals have a value of `1` for the times specified in the column. The time values for function-call signals must be double, scalar, and real, and must increase monotonically.

When you export data from the Simulation Data Inspector, function-call signals are formatted the same as other signals, with a time column and a column for signal values.

Simulation Parameters

You can import data for parameter values used in simulation. In the Simulation Data Inspector, the parameter values are shown as signals. Simulink Test uses imported parameter values to specify values for those parameters in the tests it runs based on imported data.

Parameter data is specified using two or three columns. The first column specifies the parameter names, with the cell in the header row for that column labeled `Parameter:`. The second column specifies the value used for each parameter, with the cell in the header row labeled `Value:`. Parameter data may also include a third column that contains the block path associated with each parameter, with the cell in the header row labeled `BlockPath:`. Specify names, values, and block paths for parameters starting in the first row that contains signal data, below rows used to specify signal metadata. For example, this file specifies values for two parameters, X and Y.

	A	B	C	D	E	F	G
1	time	signal1	signal2	time	signal3	Parameter:	Value:
2		Interp: zoh			Interp: zoh		
3		Type: int8	Type: int32				
4		Unit: m			Unit: m/s		
5	0	1	1	0	4	X	2
6	1	2	4	2	8	Y	1.2
7	2	3	9	3	15		
8	3	3	9	5	16		
9	3	4	16				
10	4	5	25				

Multiple Runs

You can include data for multiple runs in a single file. Within a sheet, you can divide data into runs by labeling data with a simulation number and a source type, such as `Input` or `Output`. Specify the simulation number and source type as additional signal metadata, using the label `Simulation:` for the simulation number and the label `Source:` for the source type. The Simulation Data Inspector uses the simulation number and source type only to determine which signals belong in each run. Simulink Test uses the information to define inputs, parameters, and acceptance criteria for tests to run based on imported data.

You do not need to specify the simulation number and output type for every signal. Signals to the right of a signal with a simulation number and source use the same simulation number and source until the next signal with a different source or simulation number. For example, this file defines data for two simulations and imports into four runs in the Simulation Data Inspector:

- **Run 1** contains `signal1` and `signal2`.
- **Run 2** contains `signal3`, X, and Y.
- **Run 3** contains `signal4`.
- **Run 4** contains `signal5`.

	A	B	C	D	E	F	G	H	I	J
1	time	signal1	signal2	time	signal3	Parameter:	Values:	time	signal4	signal5
2		Interp: zoh			Interp: zoh					
3		Type: int8	Type: int32							
4		Unit: m			Unit: m/s					
5		Simulation: 1							Simulation: 2	
6		Source: Input			Source: Output				Source: Input	Source: Output
7	0	1	1	0	4	X	2	1	2	1
8	1	2	4	2	8	Y	1.2	2	6	3
9	2	3	9	3	15			3	4	5
10	3	3	9	5	16			4	8	7
11	3	4	16					5	10	2
12	4	5	25							

You can also use sheets within the Microsoft Excel file to divide the data into runs and tests. When you do not specify simulation number and source information, the data on each sheet is imported into a separate run in the Simulation Data Inspector. When you export multiple runs from the Simulation Data Inspector, the data for each run is saved on a separate sheet. When you import a Microsoft Excel file that contains data on multiple sheets into Simulink Test, you are prompted to specify how to import the data.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.exportRun`

More About

- “View Data in the Simulation Data Inspector” on page 29-2
- “Import Data from a CSV File into the Simulation Data Inspector” on page 29-51
- “Import Data Using a Custom File Reader” on page 29-17

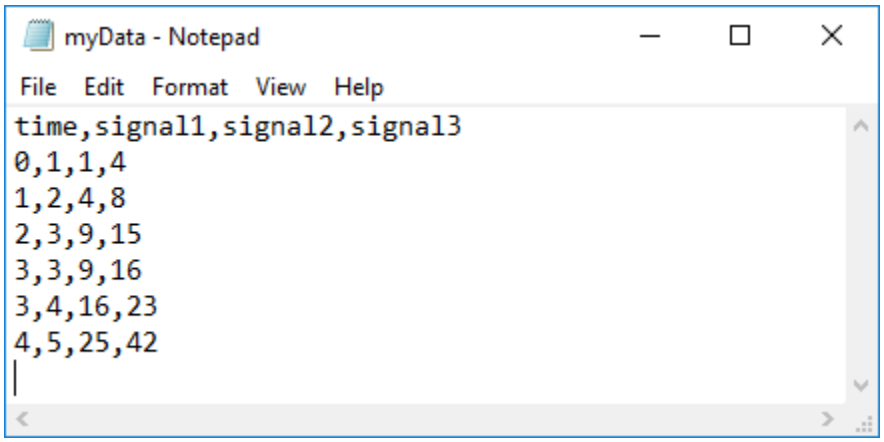
Import Data from a CSV File into the Simulation Data Inspector

To import data into the Simulation Data Inspector from a CSV file, format the data in the CSV file. Then, you can import the data using the Simulation Data Inspector UI or the `Simulink.sdi.createRun` function.

Tip When you want to import data from a CSV file where the data is formatted differently from the specification in this topic, you can write your own file reader for the Simulation Data Inspector using the `io.reader` class.

Basic File Format

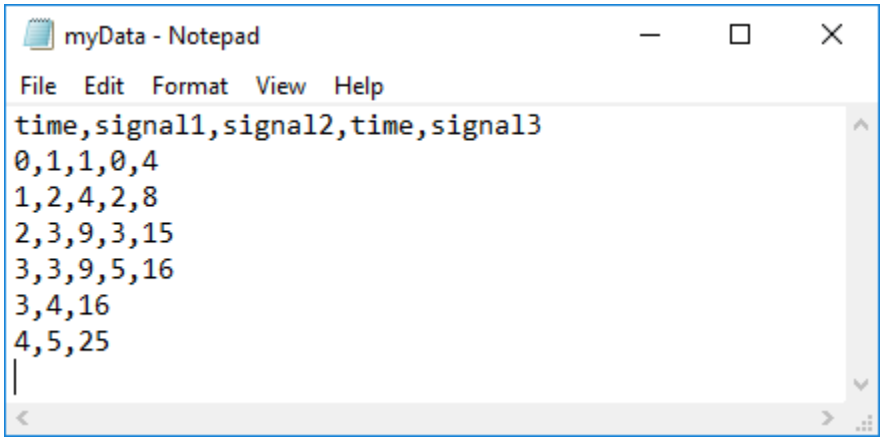
In the simplest format, the first row in the CSV file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.



The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values render as missing data. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.



Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

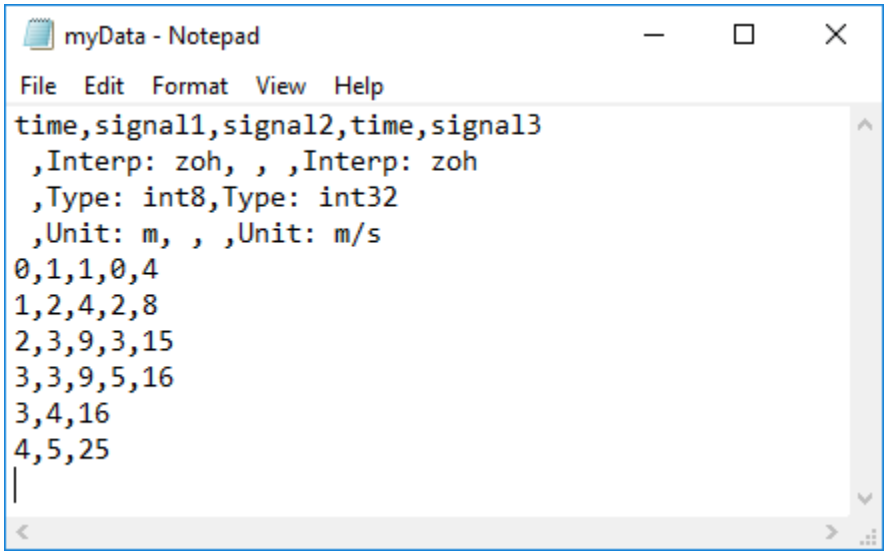
You can specify signal metadata in the CSV file to indicate the signal data type, units, interpolation method, block path, and port index. List metadata for each signal in rows between the signal name and the signal data. Label metadata according to this table.

Signal Property	Label	Value
Data type	`Type:`	Built-in data type.
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.
Block Path	`BlockPath:`	Path to the block that generated the signal.
Port Index	`PortIndex:`	Integer.

You can also import a signal with a data type defined by an enumeration class. Instead of using the `Type:` label, use the `Enum:` label and specify the value as the name of the enumeration class. The definition for the enumeration class must be saved on the MATLAB path.

When an imported file does not specify signal metadata, the Simulation Data Inspector assumes double data type and linear interpolation. You can specify the interpolation method as linear, zoh (zero-order hold), or none. If you do not specify units for the signals in your file, you can assign units to the signals in the Simulation Data Inspector after you import the file.

You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

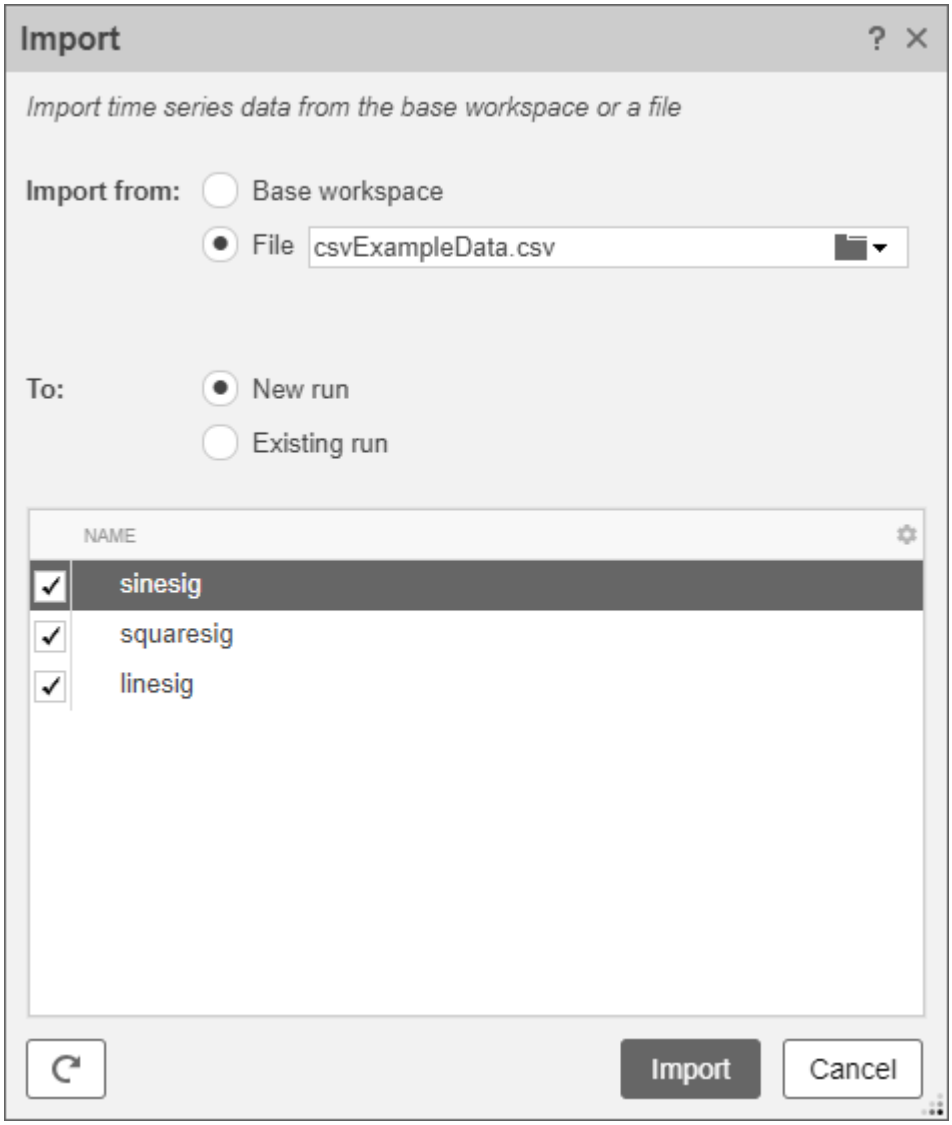


Import Data from a CSV File

You can import data from a CSV file using the Simulation Data Inspector UI or using the `Simulink.sdi.createRun` function.

To import data using the UI, open the Simulation Data Inspector using the `Simulink.sdi.view` function or the **Data Inspector** button in the Simulink™ toolstrip. Then, click the **Import** button.

In the Import dialog, select the option to import data from a file and navigate in the file system to select the file. After you select the file, data available for import shows in the table. You can choose which signals to import and whether to import them to a new or existing run. This example imports all available signals to a new run. After selecting the options, click the **Import** button.



When you import data into a new run using the UI, the new run name includes the run number followed by `Imported_Data`.

When you import data programmatically, you can specify the name of the imported run.

```
csvRunID = Simulink.sdi.createRun('CSV File Run','file','csvExampleData.csv');
```

See Also

Functions
`Simulink.sdi.createRun`

More About

- “View Data in the Simulation Data Inspector” on page 29-2

- “Microsoft Excel Import and Export Format” on page 29-43
- “Import Data Using a Custom File Reader” on page 29-17

Configure the Simulation Data Inspector

The Simulation Data Inspector supports a wide range of use cases for analyzing and visualizing data. You can modify preferences in the Simulation Data Inspector to match your visualization and analysis requirements. The preferences that you specify persist between MATLAB sessions.

By specifying preferences in the Simulation Data Inspector, you can configure options such as:

- How signals and metadata are displayed.
- Which data automatically imports from parallel simulations.
- Where prior run data is retained and how much prior data to store.
- How much memory is used during save operations.
- The system of units used to display signals.

Open the Simulation Data Inspector preferences by selecting the **Preferences** button. 

Note You can restore all preferences in the Simulation Data Inspector to default values by clicking **Restore Defaults** in the dialog or by using the `Simulink.sdi.clearPreferences` function.

Incoming Run Names and Location

You can configure how the Simulation Data Inspector handles incoming runs from import or simulation. You can choose whether new runs are added at the top of the work area or the bottom and specify a naming rule to use for runs created from simulation.

By default, the Simulation Data Inspector adds new runs below prior runs in the work area. The **Archive** settings also affect the location of runs. By default, prior runs are moved to the archive when a new simulation run is created.

The run naming rule is used to name runs created from simulation. You can create the run naming rule using a mix of literal text that is used in the run name as-is and one or more tokens that represent metadata about the run. By default, the Simulation Data Inspector names runs using the run index and model name: `Run <run_index>: <model_name>`.

Tip To rename an existing run, double-click the name in the work area and enter the new name, or modify the run name in the **Properties** pane.

Programmatic Use

You can programmatically check and modify the naming rule using the `Simulink.sdi.getRunNamingRule` and `Simulink.sdi.setRunNamingRule` functions. Restore the naming rule to its default programmatically using the `Simulink.sdi.resetRunNamingRule` function.

Signal Metadata to Display

You can control which signal metadata is displayed in the work area of the **Inspect** pane and in the results section on the **Compare** pane in the Simulation Data Inspector. You specify the metadata to display separately for each pane using the **Table Columns** preferences in the **Inspect** and **Compare** sections of the Preferences dialog, respectively.

Inspect Pane

By default, the signal name and the line style and color used to plot the signal are displayed on the **Inspect** pane. To display different or additional metadata in the work area on the **Inspect** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Inspect** section. You can always view complete metadata for the selected signal in the **Inspect** pane using the **Properties** pane.

Note Metadata displayed in the work area on **Inspect** pane is included when you generate a report of plotted signals. You can also specify metadata to include in the report regardless of what is displayed in the work area when you create the report programmatically using the `Simulink.sdi.report` function.

Compare Pane

By default, the **Compare** pane shows the signal name, the absolute and relative tolerances used in the signal comparison, and the maximum difference from the comparison result. To display different or additional metadata in the results on the **Compare** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Compare** section. You can always view complete metadata for the signals compared for a selected signal result using the **Properties** pane, where metadata that differs between the compared signals is highlighted. Signal metadata displayed on the **Compare** pane does not affect the contents of comparison reports.

Signal Selection on the Inspect Pane

You can configure how you select signals to plot on the selected subplot in the Simulation Data Inspector. By default, you use check boxes next to each signal to plot. You can also choose to plot signals based on selection in the work area. Use **Check Mode** when creating views and visualizations that represent findings and analysis of a data set. Use **Browse Mode** to quickly view and analyze data sets with a large number of signals.

For more information about creating visualizations using **Check Mode**, see “Create Plots Using the Simulation Data Inspector” on page 29-94.

For more information about using **Browse Mode**, see “Visualize Many Logged Signals” on page 29-76.

Note To use **Browse Mode**, your layout must include only **Time Plot** visualizations.

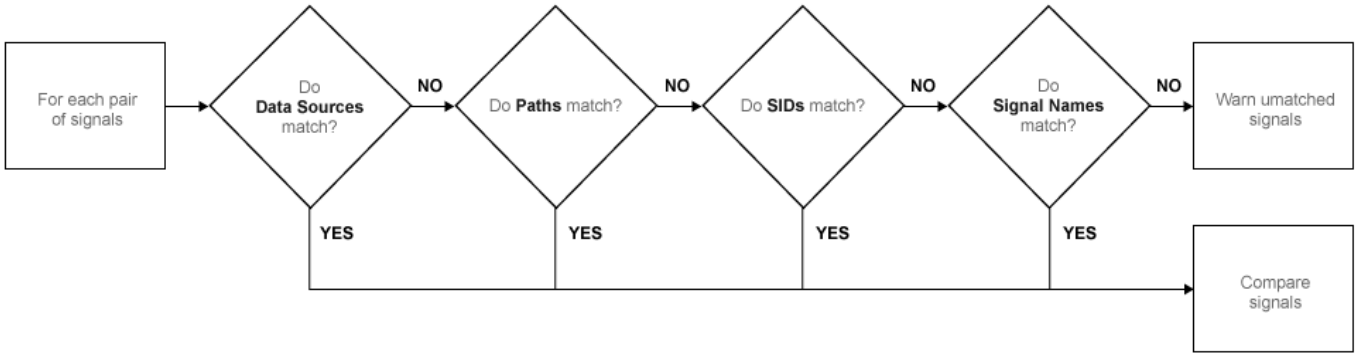
How Signals Are Aligned for Comparison

When you compare runs using the Simulation Data Inspector, the comparison algorithm pairs signals for signal comparison through a process called alignment. You can align signals between the compared runs using one or more of the signal properties shown in the table.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Simulink identifier For more information about SIDs, see “Simulink Identifiers” on page 1-7
Signal Name	Name of the signal

You can specify the priority for each piece of metadata used for alignment. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of **Then By** fields blank.

By default, the Simulation Data Inspector aligns signals between runs according to this flow chart.



For more information about configuring comparisons in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data” on page 29-139.

Colors Used to Display Comparison Results

You can configure the colors used to display comparison results using the Simulation Data Inspector preferences. You can specify whether to use the signal color from the **Inspect** pane or a fixed color for the baseline and compared signals. You can also choose colors for the tolerance and the difference signal.

By default, the Simulation Data Inspector displays comparison results using fixed colors for the baseline and compared signals. Using a fixed color allows you to avoid the baseline signal color and compared signal color being either the same or too similar to distinguish.

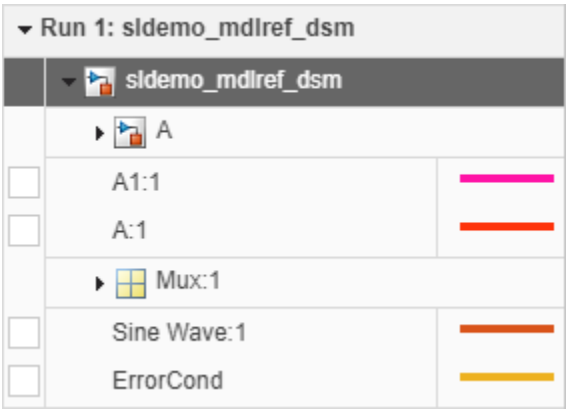
Signal Grouping

To visualize hierarchy within a data set in the Simulation Data Inspector, you can specify how to group signals within a run in the **Inspect** and **Compare** panes. The preference applies to both panes such that signals are always grouped the same way on the **Inspect** and **Compare** panes.

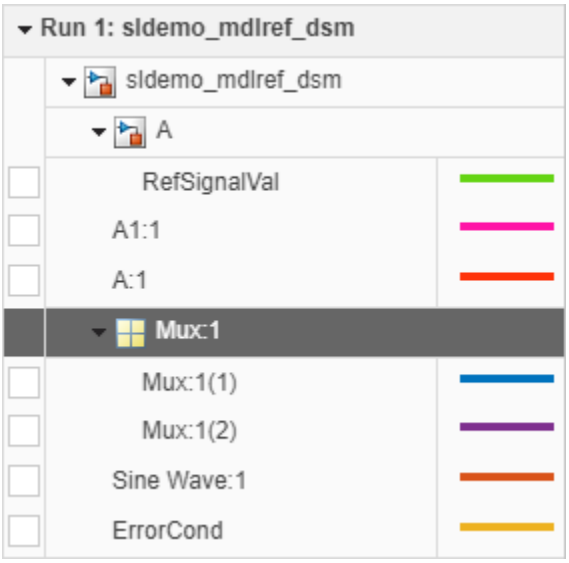
You can group signals in a run based on model hierarchy or data hierarchy. When you have Simscape, you can also group signals based on physical system hierarchy. By default, the Simulation Data Inspector groups signals by physical system hierarchy if you have a Simscape license, and by data hierarchy.

Grouping signals adds rows for the hierarchical nodes that you can expand to show the signals within that node. For example, you can group signals within a run by model hierarchy and then by data hierarchy.

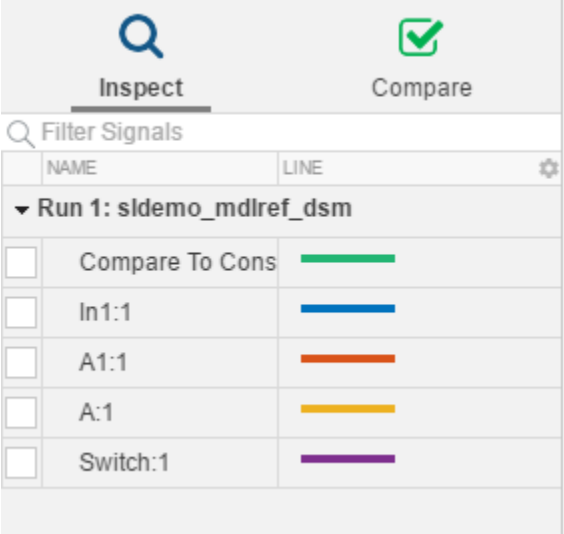
By default, all hierarchical nodes within the run are collapsed. You can expand the model node to see the logged signals.



Signals inside subsystem `A` are collapsed, and the signals in the Mux block output are grouped under `Mux: 1`. You can expand these groups to see the rest of the signals in the run. The check boxes for signals remain on the left, and the signal names indent to indicate the level of hierarchy.



To remove the hierarchy and display a flat list of signals in each run, select `None` for all grouping options.



Programmatic Use

To specify how to group signals programmatically, use the `Simulink.sdi.setTableGrouping` function.

Data to Stream from Parallel Simulations

When you run parallel simulations using the `parsim` function, you can stream logged simulation data to the Simulation Data Inspector. A dot next to the run name in the **Inspect** pane indicates the status of the simulation that corresponds to the run, so you can monitor simulation progress while visualizing the streamed data. You can control whether data streams from a parallel simulation based on the type of worker the data comes from.

By default, the Simulation Data Inspector is configured for manual import of data from parallel workers. You can use the Simulation Data Inspector programmatic interface to inspect the data on the worker and decide whether to send it to the client Simulation Data Inspector for further analysis. To manually move data from a parallel worker to the Simulation Data Inspector, use the `Simulink.sdi.sendWorkerRunToClient` function.

You may want to automatically stream data from parallel simulations that run on local workers or on local and remote workers. Streaming data from both local and remote workers may affect simulation performance, depending on how many simulations you run and how much data you log. When you choose to stream data from local workers or all parallel workers, all logged simulation data automatically shows in the Simulation Data Inspector.

Programmatic Use

You can configure Simulation Data Inspector support for parallel worker data programmatically using the `Simulink.sdi.enablePCTSupport` function.

Options for Saving and Loading Session Files

You can specify a maximum amount of memory to use while loading or saving a session file. By default, the Simulation Data Inspector uses a maximum of 100 MB of memory when you load or save a session file. You can specify a memory use limit as low as 50 MB.

To reduce the size of the saved session file, you can specify a compression option.

- `None` — Do not compress saved data.
- `Normal` — Compress the saved file as much as possible.
- `Fastest` — Compress the saved file less than `Normal` compression for faster save time.

Archive Behavior and Run Limit

The Simulation Data Inspector archive stores runs in a collapsible pane, allowing you to manage the contents of the work area without deleting run data. You can configure whether the Simulation Data Inspector automatically moves prior simulation runs to the archive. You can also limit the number of runs stored in the archive.

Manage Runs in the Archive

By default, the Simulation Data Inspector automatically archives simulation runs. When you simulate a model, the prior simulation run moves to the archive, and the Simulation Data Inspector updates the view to show the data for aligned signals in the current run.

The archive does not impose functional limitations on the runs and signals it contains. You can plot signals from the archive, and you can use runs and signals in the archive in comparisons. You can drag runs of interest from the archive to the work area and vice versa whether the **Automatically Archive** setting is enabled or disabled. To prevent the Simulation Data Inspector from automatically moving prior simulations runs to the archive, clear the **Automatically archive** setting.

When you import runs into the Simulation Data Inspector, the imported runs appear in the work area, and the **Current** tag remains on the most recent simulation run. You can import signals to existing runs in the work area and in the archive.

Tip You can delete the contents of the archive manually using the trash icon.

Limit Data Retention

To reduce the amount of data the Simulation Data Inspector retains, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the size limit, the Simulation Data Inspector starts to delete runs on a first-in, first-out basis.

The size limit applies only to runs in the archive. For the Simulation Data Inspector to automatically limit the data it retains by deleting old runs, select **Automatically archive** and specify a size limit.

By default, the Simulation Data Inspector retains the last `20` runs moved to the archive. To remove the limit, select **No limit**. To specify the maximum number of runs to store in the archive, select **Last n runs** and enter the desired limit. The Simulation Data Inspector warns you when you specify a limit that would delete runs already in the archive.

Programmatic Use

Configure the **Automatically archive** setting programmatically using the `Simulink.sdi.setAutoArchiveMode` function.

Specify the number of runs to retain in the archive using the `Simulink.sdi.setArchiveRunLimit` function.

Signal Display Units

Signals in the Simulation Data Inspector have two units properties: stored units and display units. The stored units represent the units of the data saved to disk. The display units specify how the Simulation Data Inspector displays the data. You can configure the Simulation Data Inspector to use a system of units to define the display units for all signals. You can choose either the **SI** or **US Customary** system of units, or you can display data using its stored units.

When you use a system of units to define display units for signals in the Simulation Data Inspector, the display units update for any signal with display units that are not valid for that unit system. For example, if you select **SI** units, the display units for a signal may update from `ft` to `m`.

Note The system of units you choose to use in the Simulation Data Inspector does not affect the stored units for any signal. You can convert the stored units for a signal using the `convertUnits` function. Conversion may result in loss of precision.

In addition to selecting a system of units, you can specify override units so that all signals of a given measurement type are displayed using consistent units. For example, if you want to visualize all signals that represent weight using units of `kg`, specify `kg` as an override unit.

Tip For a list of units supported by Simulink, enter `showunitslist` in the MATLAB Command Window.

You can also modify the display units for a specific signal using the **Properties** pane. For more information, see “Modify Signal Properties in the Simulation Data Inspector” on page 29-120.

Programmatic Use

Configure the unit system and override units using the `Simulink.sdi.setUnitSystem` function. You can check the current units preferences using the `Simulink.sdi.getUnitSystem` function.

See Also

Functions

`Simulink.sdi.clearPreferences` | `Simulink.sdi.enablePCTSupport` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode` | `Simulink.sdi.setRunNamingRule` | `Simulink.sdi.setTableGrouping`

More About

- “Iterate Model Design Using the Simulation Data Inspector” on page 29-71

- “How the Simulation Data Inspector Compares Data” on page 29-139
- “Compare Simulation Data” on page 29-130
- “Create Plots Using the Simulation Data Inspector” on page 29-94
- “Modify Signal Properties in the Simulation Data Inspector” on page 29-120

Control Display of Streaming Data Using Triggers

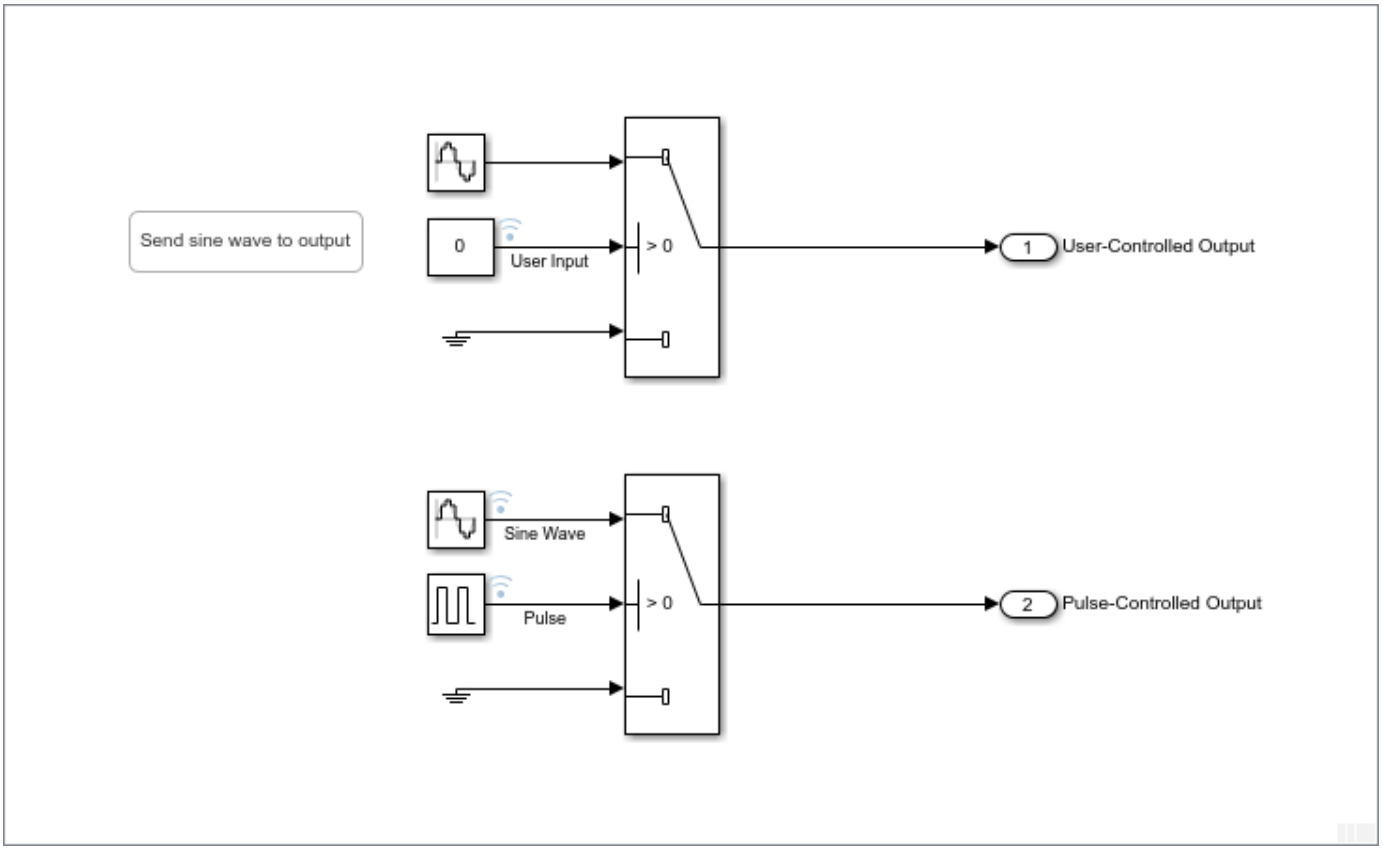
A *trigger* is an event that determines when to draw the plot of a streaming signal. Trigger settings specify criteria for the trigger event and where the trigger event is displayed within the updated data. Triggers facilitate viewing and analyzing streaming data by allowing you to capture transient changes in signals and stabilize the display of steady-state periodic signals. You can use triggers to inspect streaming data using several tools, such as the Scope block and the Simulation Data Inspector. This example uses the Simulation Data Inspector add, configure, and modify triggers on signals streaming from a model simulation.

Note The following examples add a trigger to signals that are plotted in the Simulation Data Inspector. However, you can add a trigger to a signal from the active simulation that is not plotted.

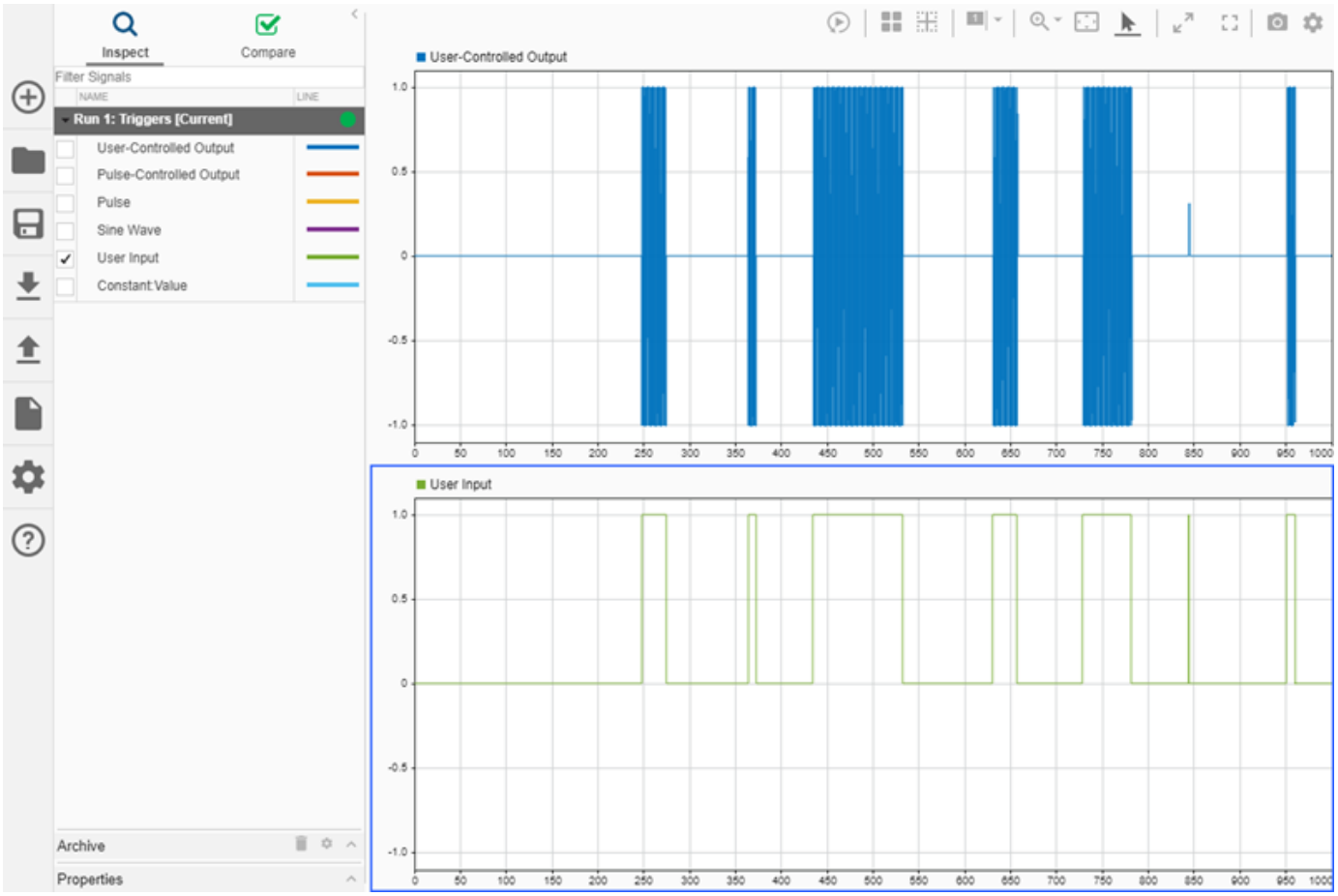
For complete descriptions of all trigger types and options, see “Scope Triggers Panel” on page 28-12.

Examine the Model

The model in this example generates two logged output signals. Both signals are created by passing the output of a Sine Wave block through a switch. Both switches are configured to allow the sine signal through when the control signal is greater than zero. The switch for the first output is controlled by user input supplied using the Push Button block. The switch for the second output is controlled by the output of a Pulse Generator block.



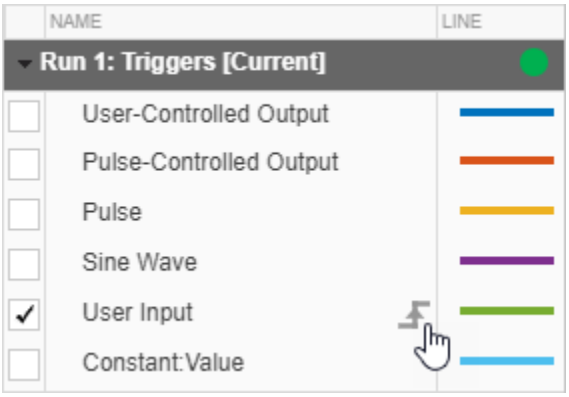
The model uses simulation pacing so the model behavior and signal display are responsive to user interaction. Click **Run** to start a simulation. Then, click **Data Inspector** to open the Simulation Data Inspector. In the Simulation Data Inspector, change the layout to show two subplots. Plot the `User-Controlled Output` signal on the top subplot and the `User Input` signal on the bottom subplot. Press and hold the Push Button block periodically throughout the simulation to pass the Sine Wave block output to the Outport block.



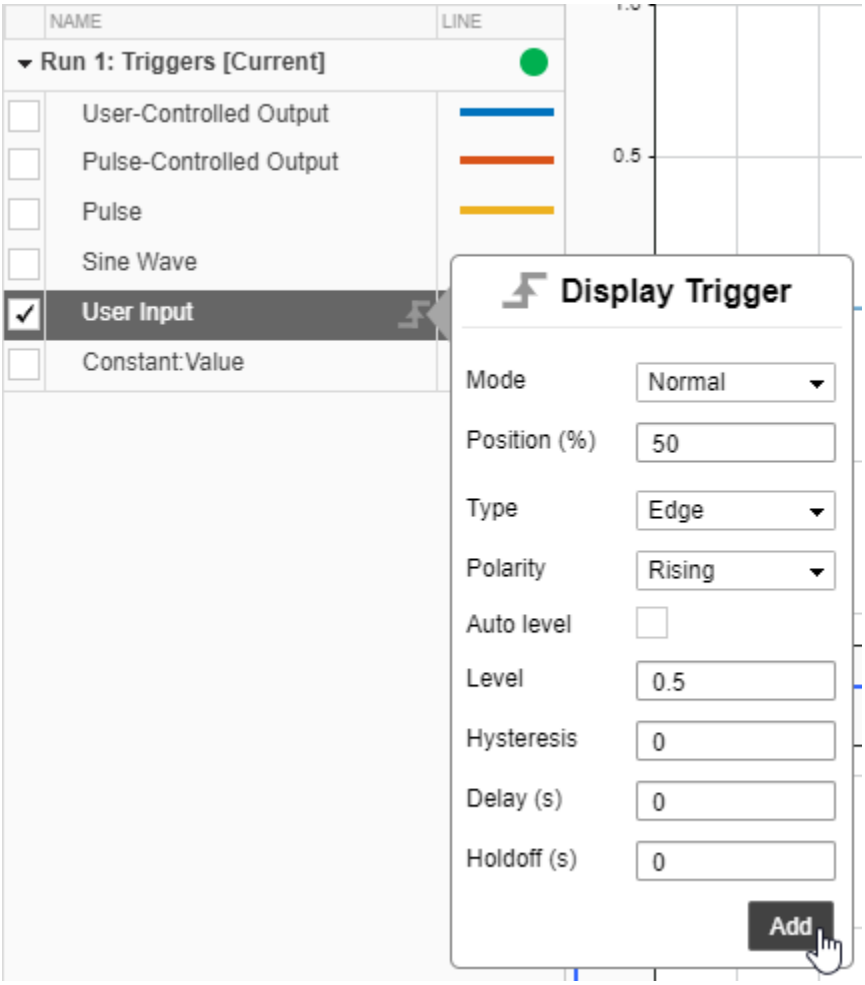
Interactively Generate Trigger Events

The sine wave characteristics are difficult to see when you view the signals over the entire simulation time. When you use a trigger to control when the plot updates, you can view the streaming data over a smaller time range, which allows you to see the sine wave more clearly. Using the trigger ensures that you see the signal behavior of interest in the smaller time range. In this example, you want to see the change in the output signal when you press the Push Button block.

To trigger plot updates based on button presses, add a trigger to the `User Input` signal. To add a trigger, pause on the row for the `User Input` signal in the Simulation Data Inspector and click the trigger icon that appears.



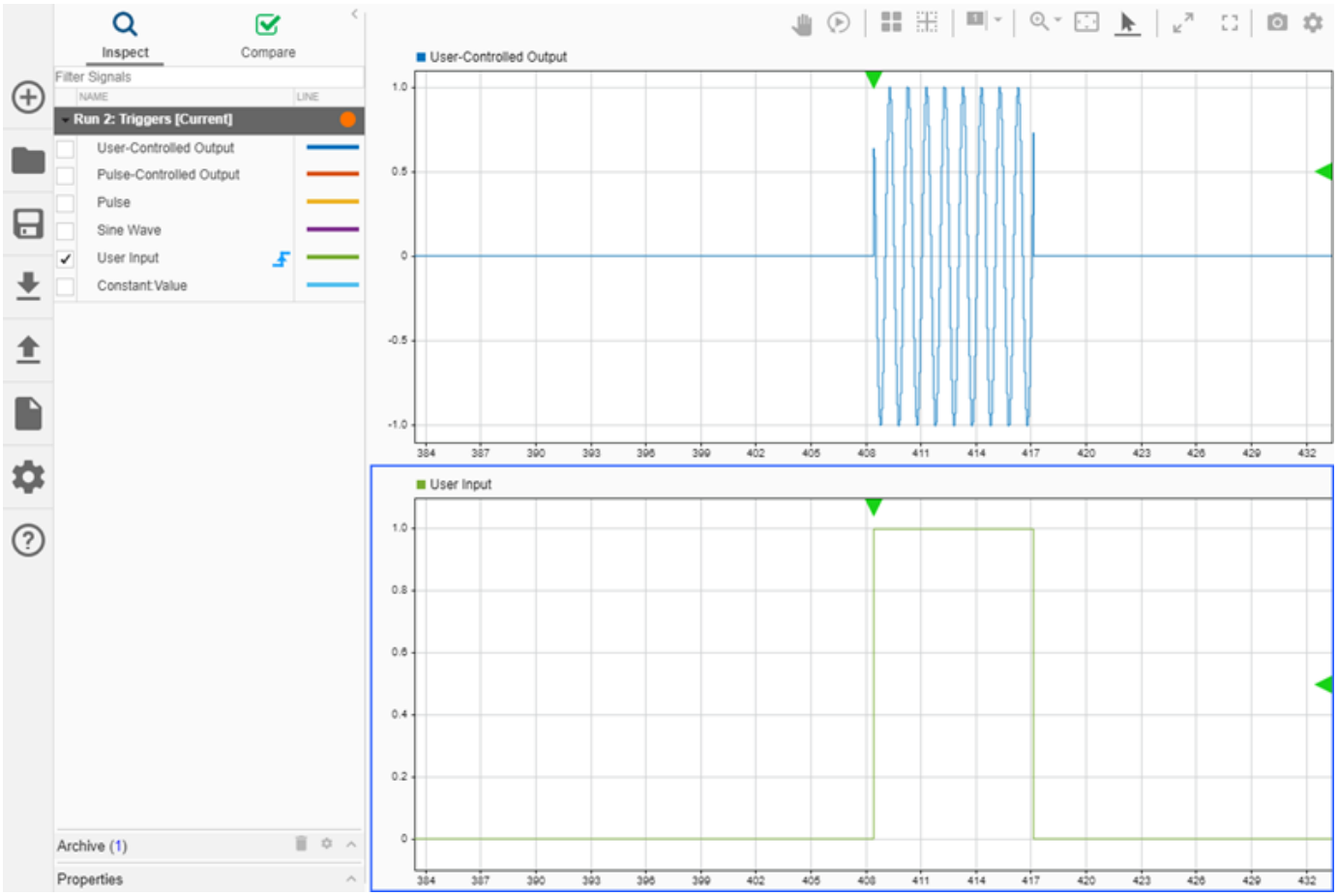
The default settings in the Display Trigger dialog configure an auto-leveled rising-edge trigger, with the trigger event positioned in the middle of the displayed data. Change **Mode** to **Normal**, then clear the **Auto level** check box and set the **Level** to `0.5`. Then, click **Add**.



Using normal mode for the trigger means that the plot only updates when a trigger occurs. In auto mode, the display also updates when a trigger is not detected for the duration of the time span. The trigger level of `0.5` puts the level in the middle of the signal range.

Before simulating the model, adjust the time range displayed in the plots. In the **Visualization Settings**, set **Time span** to `50`.

Simulate the model and view the streaming signals in the Simulation Data Inspector. When you start a new simulation, the trigger configured in the Simulation Data Inspector automatically transfers to the corresponding signal in the current simulation. The display updates in response to the trigger events you cause by pressing the Push Button block in the model. On the smaller time span, you can see the shape of the sine wave in the output signal. The trigger position and level are marked with green triangles. You can click and drag the trigger markers to adjust the level and position. A new value is applied for the level or position when you let go of the marker after dragging.



Experiment with this configuration to test out different kinds of triggers and different trigger settings. For example, try using a falling-edge trigger. You can also adjust the simulation pacing in the model to explore how timing can affect trigger behavior. When the trigger occurs, the Simulation Data Inspector collects all the data to display before updating the plot. If you slow the simulation pacing closer to a rate where one simulation second takes one second of clock time, you can see the delay between causing a trigger by pressing the Push Button block and the plot update in the Simulation Data Inspector.

Capture Signal Transient Response

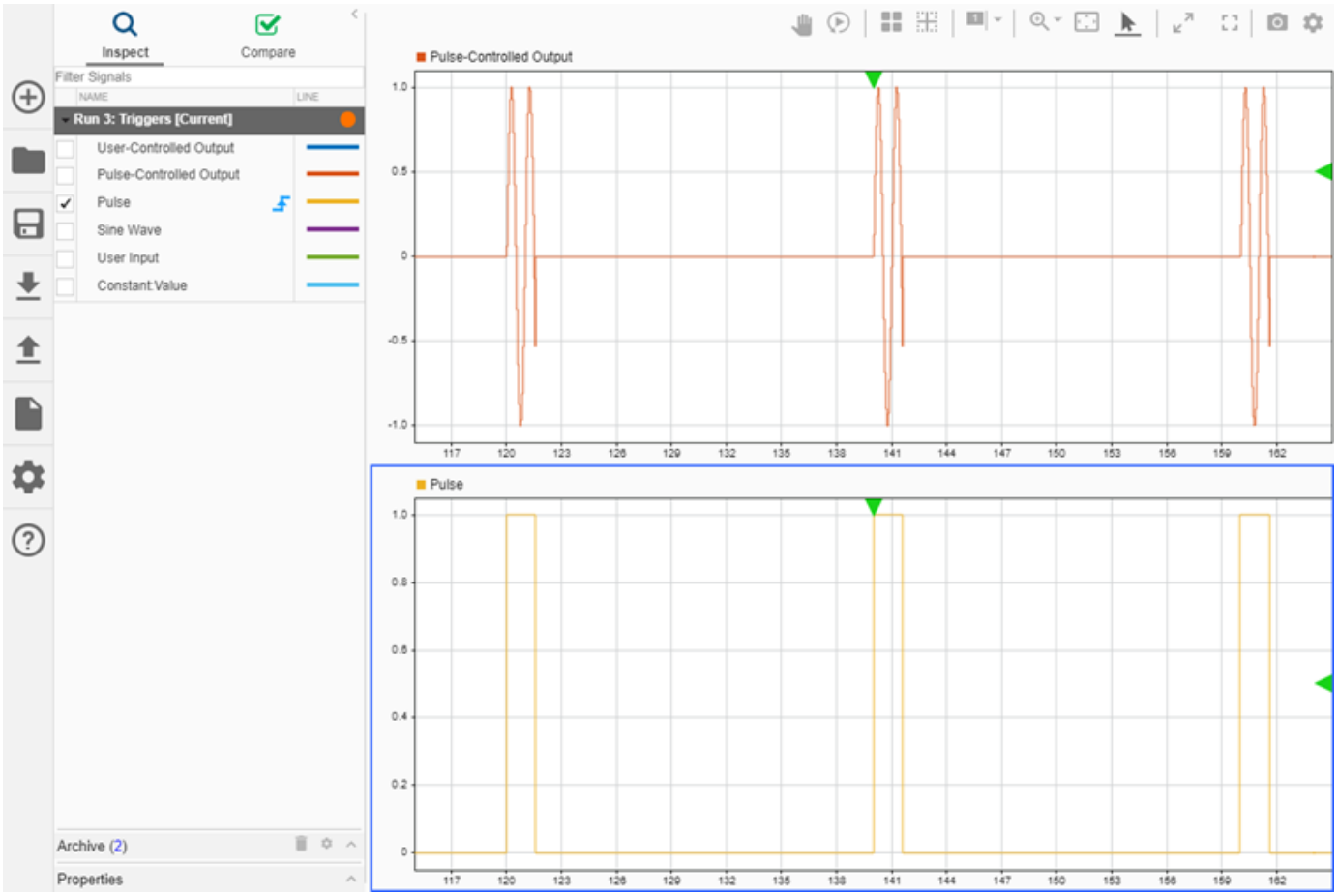
A trigger allows you to capture transient signal responses to changes in the system under test. For example, you can use the input signal to an inverter as a trigger when you want to measure the rise

or fall time for the inverter output signal. The display updates only when the trigger occurs, so the data is displayed long enough for you to make a measurement. When you interactively generate trigger events, you capture the output signal transient response to the user input. Systems often do not have user input and respond to changes in upstream signals, such as the `Pulse` signal that controls the switch that generates the `Pulse-Controlled Output` signal.

Add a trigger to the `Pulse` signal in the Simulation Data Inspector. Use a rising-edge trigger in normal mode with a level of `0.5` again. Click **Add Trigger** in the warning dialog that appears.

Before simulating the model, update the plots so that the `Pulse-Controlled Output` is shown on the top plot and the `Pulse` signal is shown on the bottom plot.

Simulate the model. The Simulation Data Inspector display updates are not triggered by the user input in this simulation, so you do not need to press the Push Button block. Observe how the `Pulse-Controlled Output` signal is different from the `User-Controlled Output` signal from the previous simulation and how the plot updates occur in response to the periodic `Pulse` signal.



You can experiment in additional simulations by adjusting the pulse width or period of the `Pulse` signal. You can also try out different types of triggers, such as a pulse width trigger.

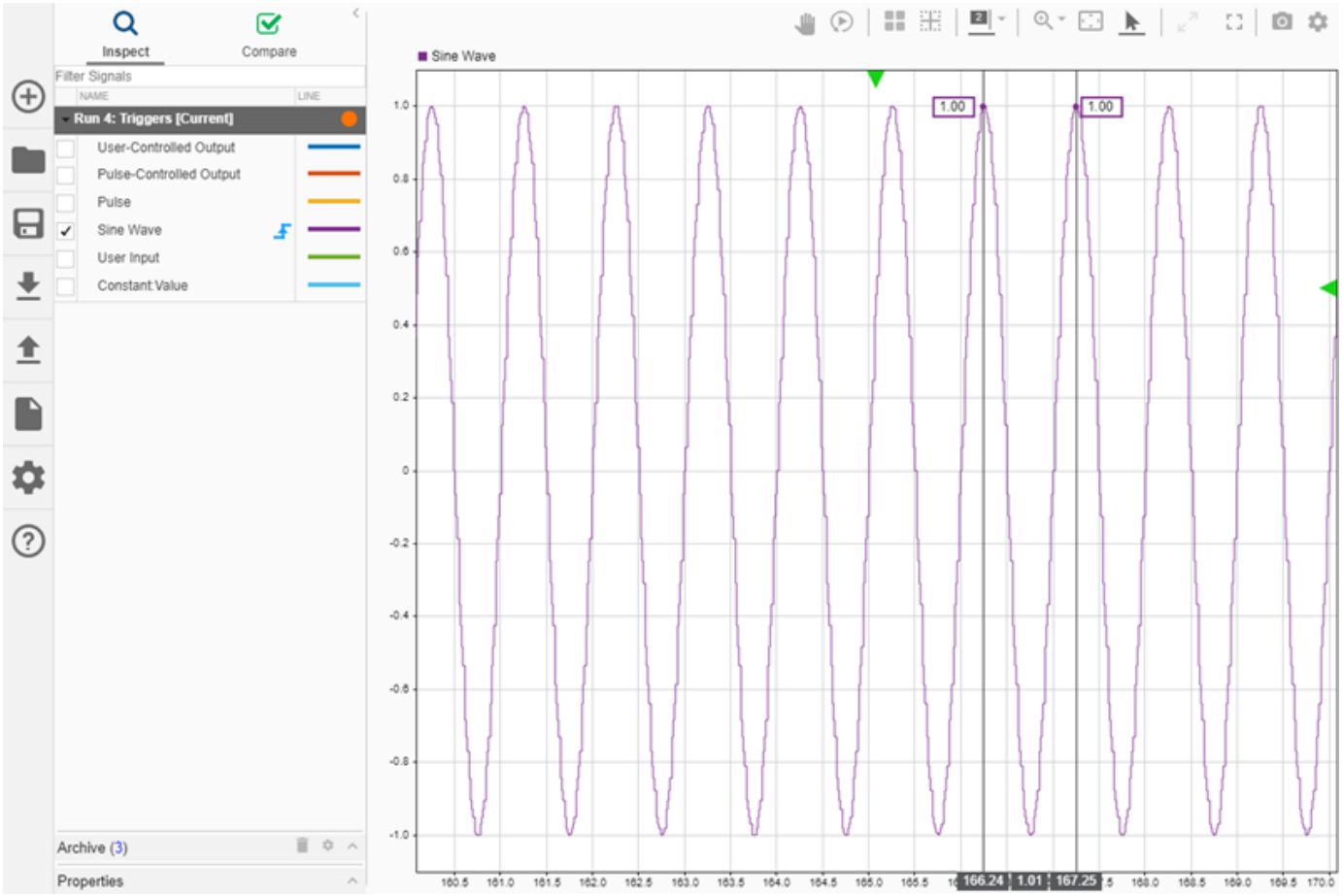
Stabilize a Steady-State Periodic Signal

Using a trigger can stabilize the display of a periodic steady-state signal that you want to view and analyze. Because the trigger event is always in the stable position in the plot, a streaming periodic signal can appear still in the plot, rather than shifting or scrolling. When the display is stable, you can inspect and analyze signal characteristics, such as frequency and amplitude.

To see how a trigger can stabilize the display of a periodic signal, move the trigger to the `Sine Wave` signal. Use the same settings to configure the rising-edge trigger in normal mode with a level of `0.5`.

Before simulation, update the plot layout to show a single plot. Plot the `Sine Wave` signal, and set the **Time span** to `10`.

Simulate the model. During simulation, you can add cursors in the Simulation Data Inspector and measure the amplitude and period of the signal.



To contrast, remove the trigger and simulate the model again. Notice the differences in how the display updates. Try simulating with the **Update mode** set to `Wrap` and then set to `Scroll`. Specify **Update mode** in **Visualization Settings**.

See Also

More About

- “Scope Triggers Panel” on page 28-12
- “View Data in the Simulation Data Inspector” on page 29-2
- “Iterate Model Design Using the Simulation Data Inspector” on page 29-71

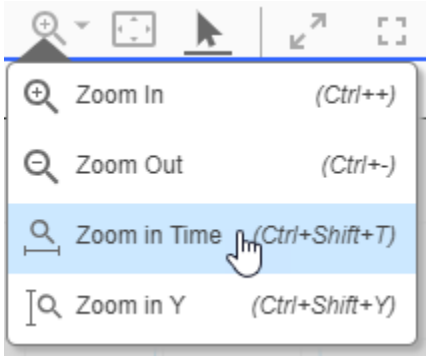
Iterate Model Design Using the Simulation Data Inspector

The Simulation Data Inspector facilitates iterative design, debugging, and optimization by providing easy access to visualizations of logged data during simulation. Using the Simulation Data Inspector archive, you can maintain a focused work area and control data retention for multiple simulations. When you automatically archive simulation runs, you can create a view that is automatically applied to subsequent simulation runs. With **Browse Mode** enabled, you can inspect large amounts of logged signals on **Time Plot** visualizations more quickly.

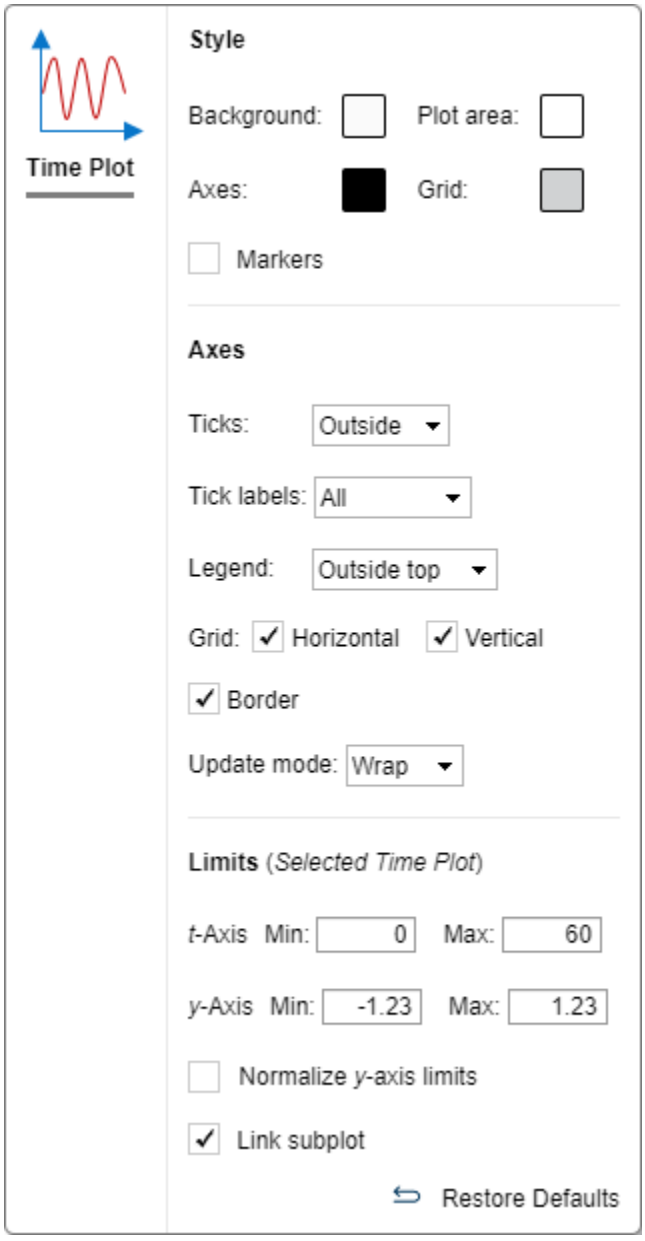
View and Inspect Signals During Simulation

You can plot logged signals in the Simulation Data Inspector during simulation. Viewing signals during a simulation can help build understanding of the model and allow you to make decisions about model parameters without waiting for a simulation to complete.

When you plot data that is streaming from an active simulation, the Simulation Data Inspector automatically updates the time span of the plot to match the stop time for the simulation. To view the streaming data on a smaller time scale, change the ***t*-Axis Max** value or zoom in time. To update the ***t*-Axis Max**, open the **Visualization Settings** . To zoom in time, select the **Zoom in time** option from the **Zoom** menu and click and drag in the plot area to specify the time span.

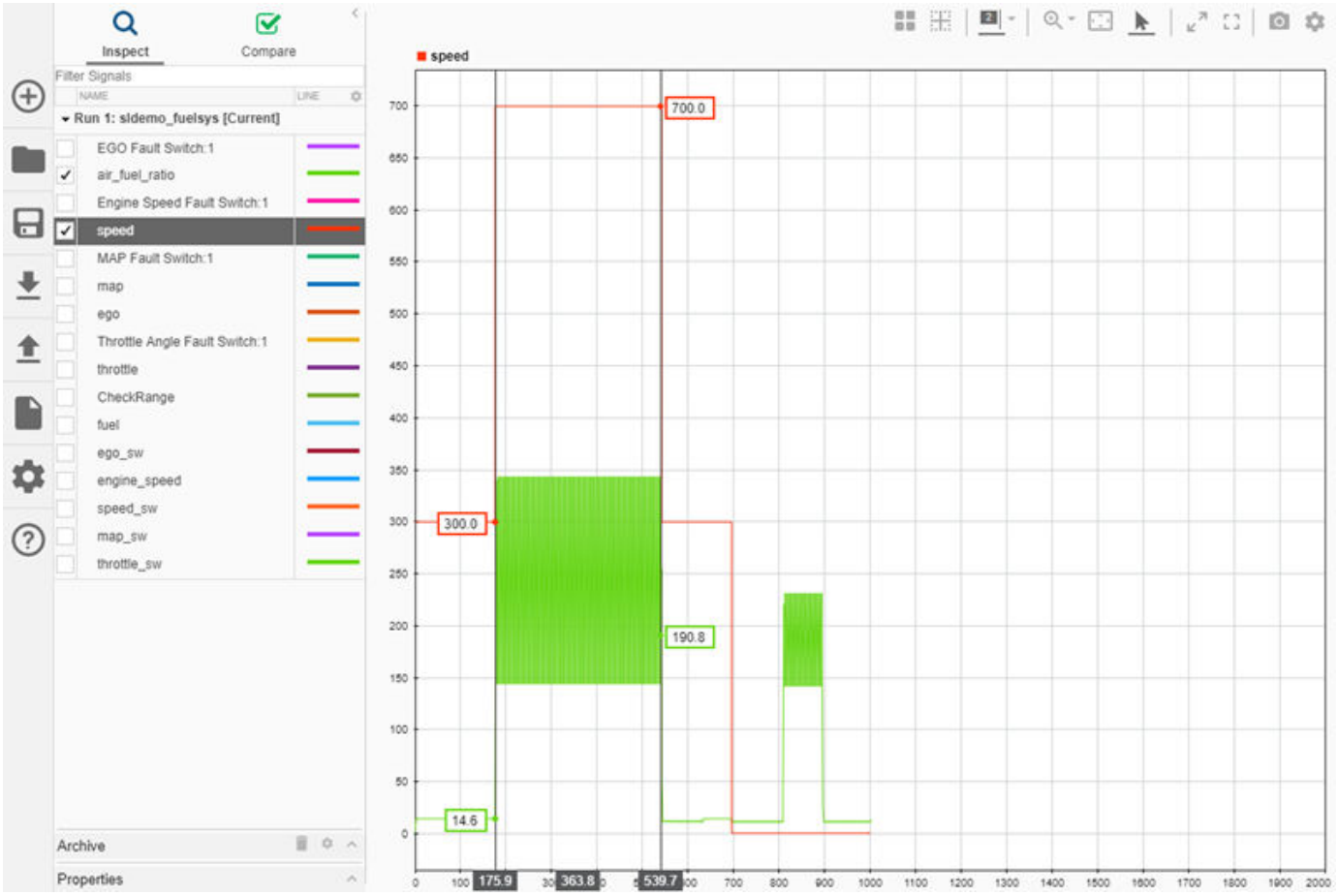


Use the **Update mode** setting to specify whether the Simulation Data Inspector wraps or scrolls the plot after the streaming data reaches the maximum time you specified.



To closely examine signals throughout design iteration, add data cursors to your plots. You can add a data cursor to the plot during simulation, while a simulation is paused, or after the simulation has finished. To add a data cursor, click the cursor button  . Drag the data cursor across the plot to inspect signal values. When you display multiple signals on a plot, the data cursor indicates the value for each signal in a color-coded box.

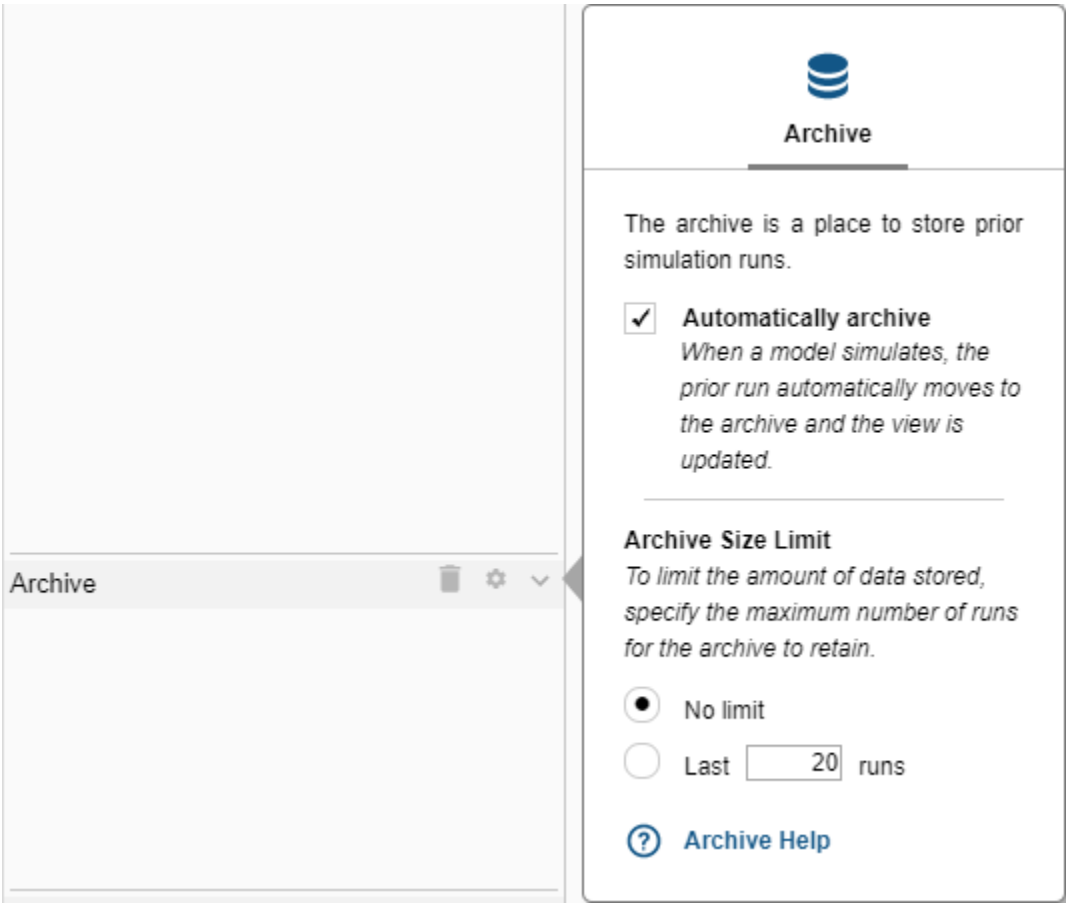
Click the drop-down on the cursors button and select **Two Cursors** to add a second data cursor and explore temporal characteristics of your data. You can move each cursor independently, or you can move them together. The number between the two cursors displays the time difference between the cursors.



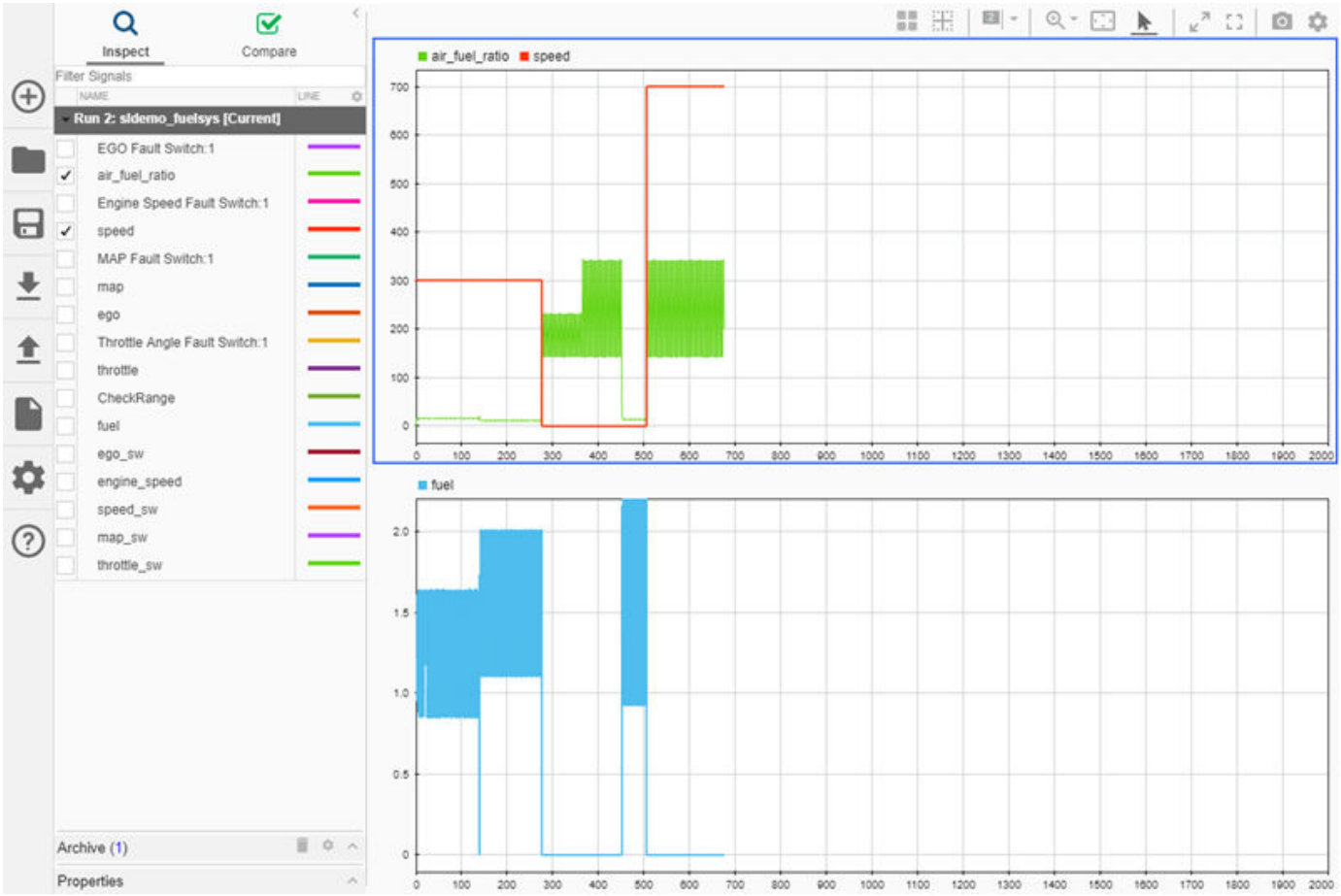
You can type directly into the box to specify a desired cursor separation. To move the cursors together, click and hold the middle number indicator, and drag it across the graphical viewing area to your desired position. For more information about using data cursors, see “Inspect Simulation Data” on page 29-107.

Automatically Transfer View to Current Simulation

By default, the Simulation Data Inspector is configured to automatically archive simulation runs and transfer a view from the previous run to the current simulation. When you run a new simulation, the Simulation Data Inspector moves the prior simulation run to the archive and updates the view to show aligned signals from the current run. You can control the **Automatically archive** setting on the **Archive** settings pane.



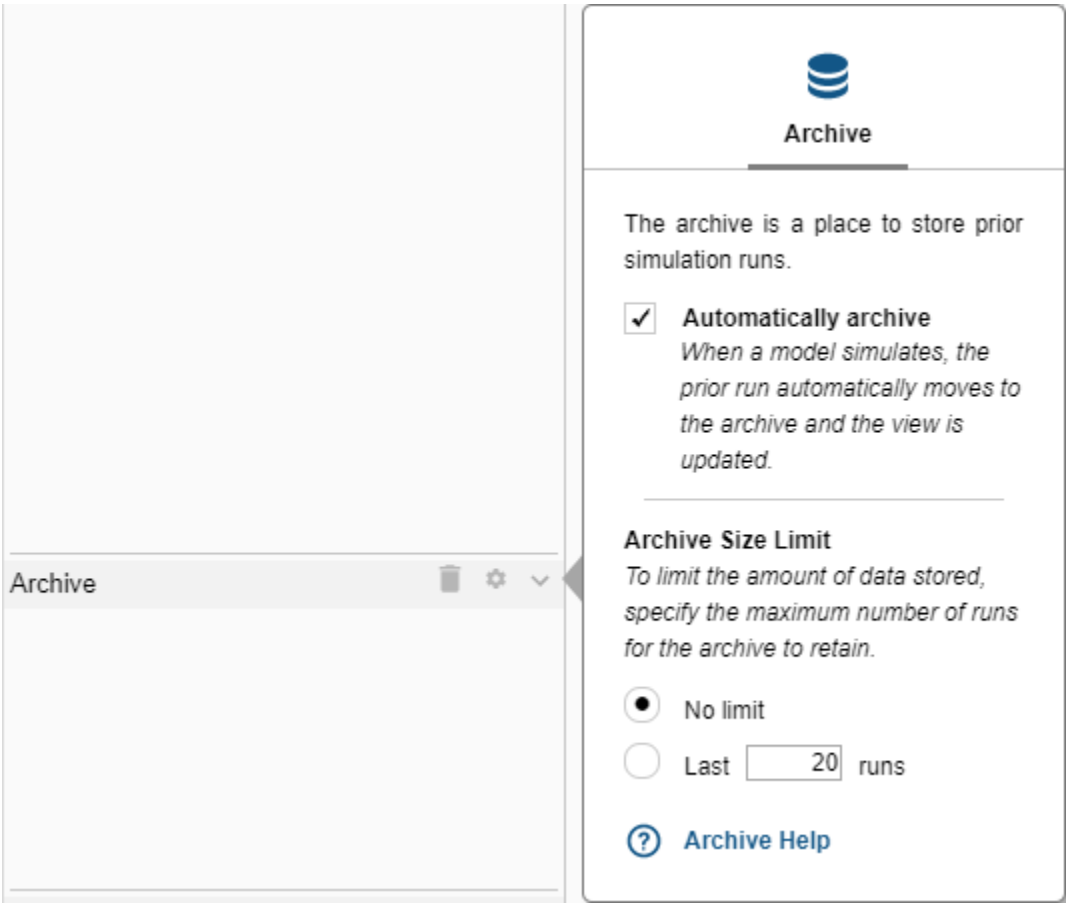
For example, create a view of signals from the `sldemo_fuelsys` model. Then, simulate the `sldemo_fuelsys` model again. The signals from the previous run clear from the plots, and the Simulation Data Inspector plots data from the current simulation as the simulation runs. The previous run also moves into the archive, reducing clutter in the work area.



You can access and plot data from previous runs from within the archive, and you can drag runs of interest into the work area from the archive.

Control Data Retention

Running many simulations with logged signals can create a lot of data, especially during iterative design and debugging workflows. To limit the amount of disk space occupied by logged data, you can limit the amount of data the Simulation Data Inspector stores in the archive on the **Archive** settings pane.

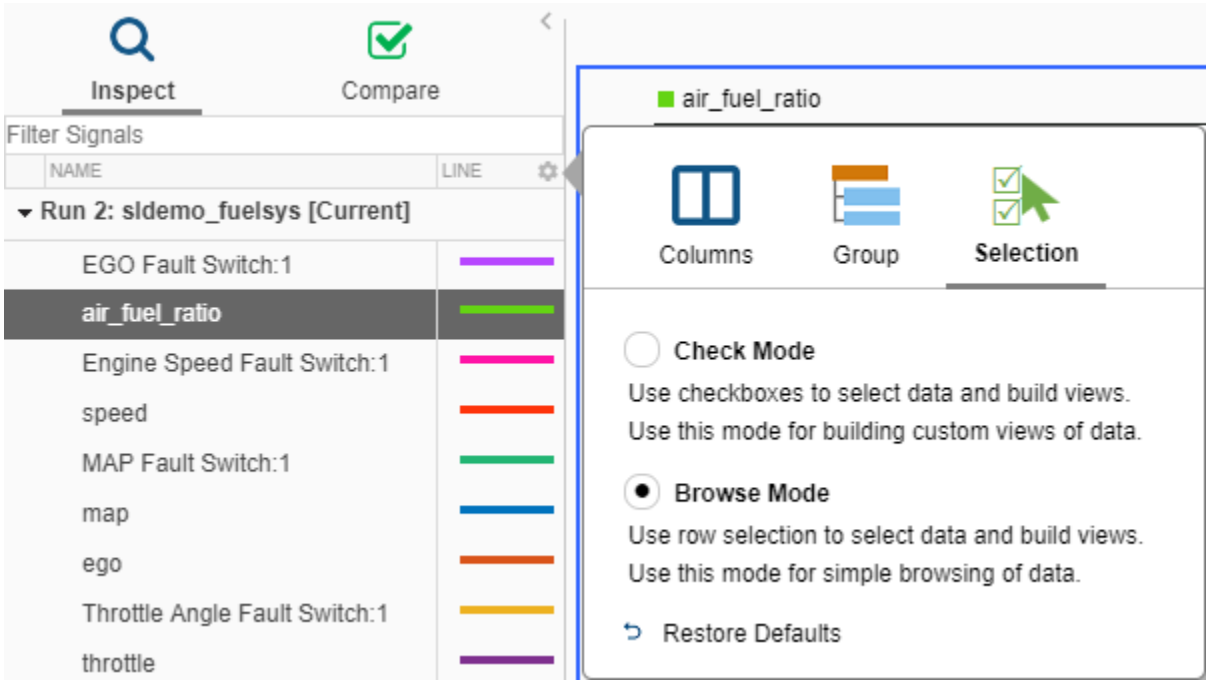


By default, the Simulation Data Inspector does not limit the number of runs stored in the archive. To impose a limit, select the **Last runs** radio button, and specify the desired limit. When the number of runs in the archive reaches the limit, the Simulation Data Inspector deletes the run that has been in the archive the longest. If the archive contains more runs than the limit you specify, the Simulation Data Inspector provides a warning before deleting your run data.

Tip When you do not want the Simulation Data Inspector to store run data, specify an archive limit of `0`. With an archive limit of `0` and **Automatically archive** enabled, the Simulation Data Inspector automatically transfers your configured view to the current simulation and only retains the current simulation data.

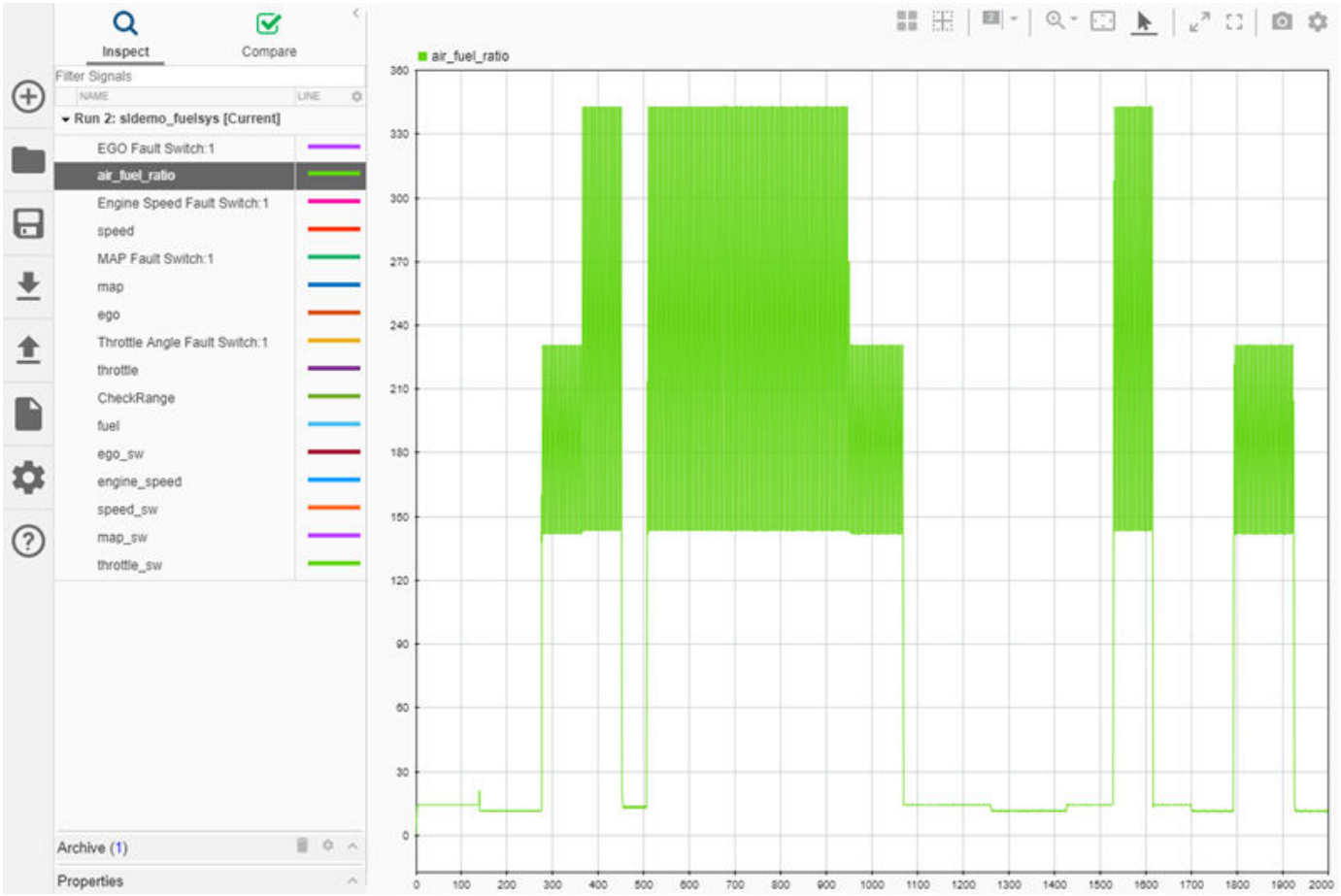
Visualize Many Logged Signals

You can use **Browse Mode** to inspect many signals quickly on **Time Plot** visualizations. To enable **Browse Mode**, click the gear icon in the navigation pane, and navigate to the **Selection** pane. Click the radio button next to **Browse Mode**.

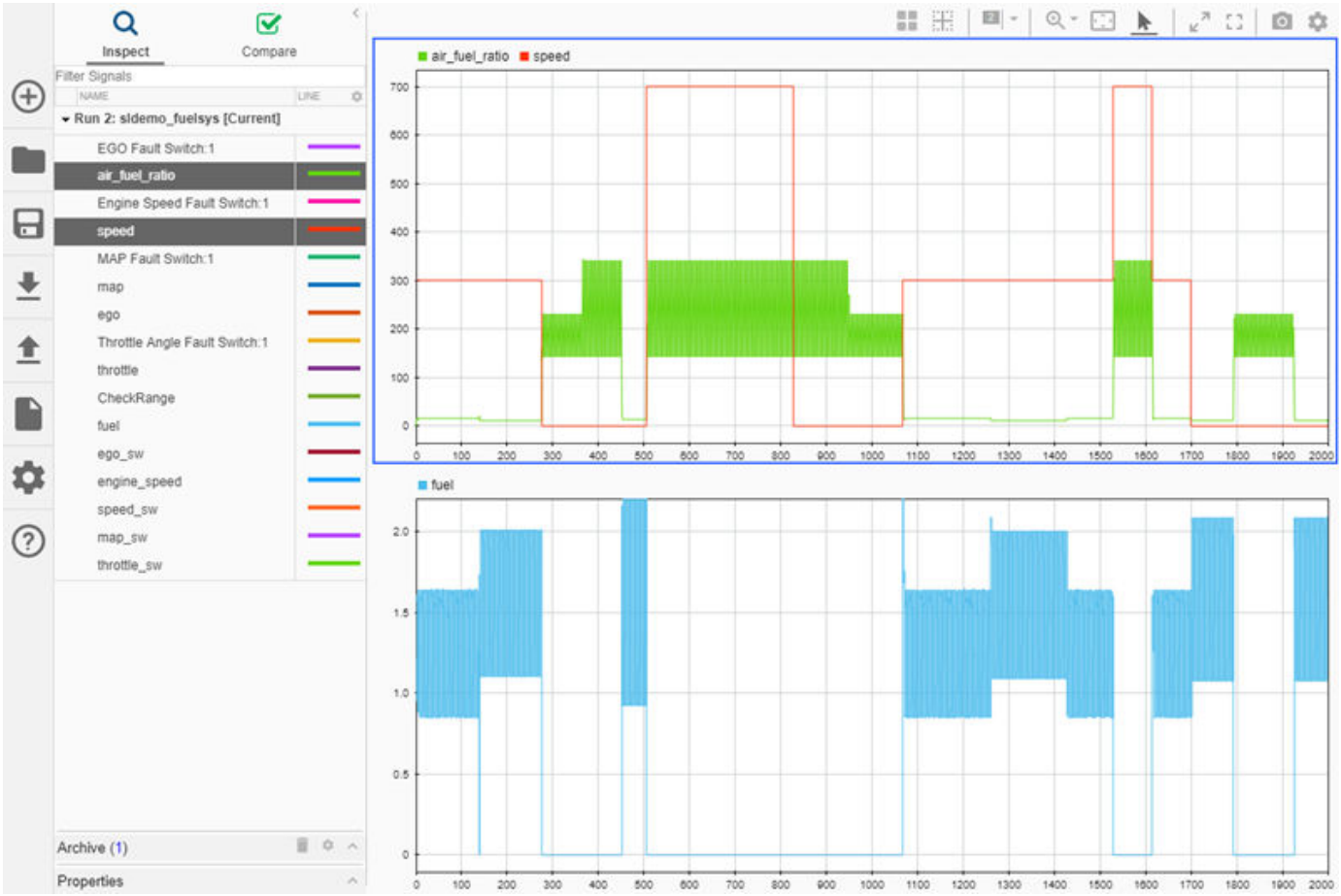


Note To use **Browse Mode**, your layout must include only **Time Plot** visualizations.

In **Browse Mode**, signals selected in the navigation pane are plotted on the active **Time Plot**. You can use the arrow keys on your keyboard to change the selected signals, and you can select multiple signals by holding the **Shift** or **Ctrl** keys. When you select a row for a run or hierarchical organization, no signals are displayed in the graphical viewing area.



You can also use plot layouts in **Browse Mode** to make more complex visualizations. When the graphical viewing area contains multiple subplots, selecting new signals using the mouse or keyboard changes only the active plot.



When you finish browsing your simulation results, enable **Check Mode** on the **Selection** pane of the work area settings to build plots that showcase your data. For more information about creating plots with the Simulation Data Inspector, see “Create Plots Using the Simulation Data Inspector” on page 29-94.

See Also

Related Examples

- “Access Data in a MATLAB Function During Simulation” on page 29-80
- “Inspect Simulation Data” on page 29-107
- “Compare Simulation Data” on page 29-130
- “View Data in the Simulation Data Inspector” on page 29-2
- “Inspect and Compare Data Programmatically” on page 29-150

Access Data in a MATLAB Function During Simulation

You can stream signal data to a MATLAB callback function during simulation using the **Data Access** functionality in the **Instrumentation Properties** for a signal. The function you provide receives data in packets asynchronously throughout the simulation. The callback function executes each time it receives new data. You can write a callback function to process signal data during simulation or to create a custom visualization of a signal. The callback function does not affect signal values in your model. This example illustrates the steps required to access signal data during simulation using a simple callback function that plots the signal.

Note Data access does not support Rapid Accelerator mode, referenced models, fixed-point data, or bus signals and only supports 1-D and 2-D matrix signals.

Write a Callback Function for Data Access

The data access callback function always receives signal data as the first argument. You can choose to send the simulation time and a parameter. When you include all three arguments, simulation time is the second argument. Your callback function can be specific to a single signal, or you can use the same callback to process and visualize multiple signals. The callback function only has access to data for a single signal at a time. This example creates a callback function that receives signal data, time, and a parameter used to identify which signal the function is processing for a given call.

Author your callback function in an m-file with the same name as your function. For more information about writing MATLAB functions, see “Create Functions in Files”. The example callback function uses the optional parameter to assign a numerical identifier to each signal. The parameter is used to create a unique figure for each signal and assign each signal a different line color. To accumulate signal data on the plot, the callback includes `hold on`. For each call to the callback, the function receives a portion of the signal data. You can use the callback to accumulate the packets if desired.

```
function plotSignals(y,time,sigNum)

    figure(sigNum)

    if isequal(sigNum,1)
        marker = 'ro-';
    elseif isequal(sigNum,2)
        marker = 'go-';
    else
        marker = 'bo-';
    end

    hold on;
    plot(time,y,marker);

end
```

The callback function provides no return values. If you specify the function with returns, the return data is discarded.

Tip Use `assignin` to access data from the callback function in the base workspace.

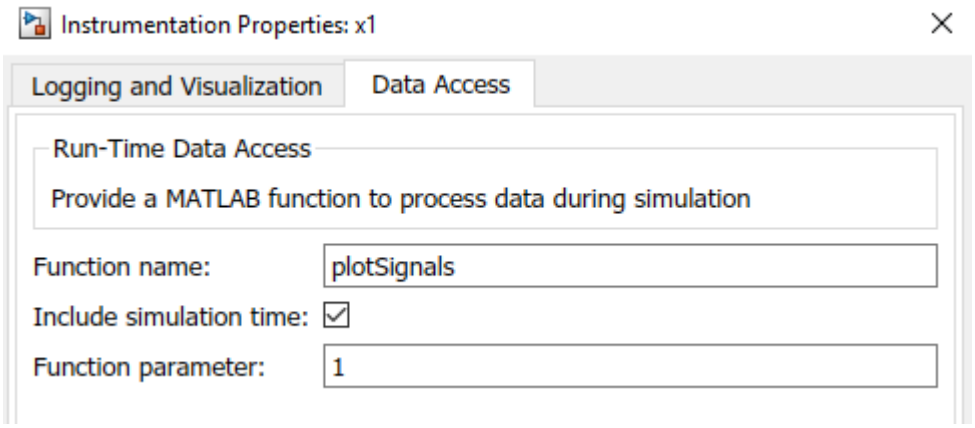
Save the callback function in a location on the MATLAB path. If the location where you save the callback is not on the path, you can add it to the path. Right-click the directory containing the callback in the **Current Folder** section in MATLAB and select **Add to Path**.

Configure Signals for Data Access

To access data for a signal with the data access callback, you need to log the signal. To mark a signal for logging, right-click the signal in your model and select **Log Selected Signals**. To configure data access for the signal, right-click the logging badge and select **Properties**. On the **Data Access** tab of the **Instrumentation Properties**, you can specify the name of your callback function, whether the callback function takes time as an argument, and the optional parameter value.

For example, open the `vdp` model and configure the `x1` and `x2` signals to use the callback from the previous section.

1. Enter `vdp` in the MATLAB **Command Window**.
2. Select the `x1` and `x2` signals. Then, right-click the selected signals and choose **Log Selected Signals** from the context menu.
3. To open the **Instrumentation Properties**, right-click the logging badge for `x1` and select **Properties**.
4. On the **Data Access** pane, enter the name of the callback, check **Include simulation time**, and enter `1` as the **Function parameter**.



5. Open the **Instrumentation Properties** for `x2` and configure the **Data Access** pane to use the `plotSignals` callback with a **Function parameter** value of `2`.

Simulate the `vdp` model. The callback generates Figures 1 and 2 to display the `x1` and `x2` signals during simulation.

You can modify the callback function to create a custom visualization or to create a plot of processed signal data. Errors related to the data access callback function do not interrupt simulation. The errors surface in the **Diagnostic Viewer** as a warning.

See Also

More About

- “View Data in the Simulation Data Inspector” on page 29-2

Save and Share Simulation Data Inspector Data and Views

After you inspect, analyze, or compare your data in the Simulation Data Inspector, you can share your results with others. The Simulation Data Inspector provides several options for sharing and saving your data and results, depending on your needs. With the Simulation Data Inspector, you can:

- Save your data and layout modifications in a Simulation Data Inspector session.
- Share your layout modifications in a Simulation Data Inspector view.
- Share images and figures of plots you create in the Simulation Data Inspector.
- Create a Simulation Data Inspector report.
- Export your data from the Simulation Data Inspector.

Save and Load Simulation Data Inspector Sessions

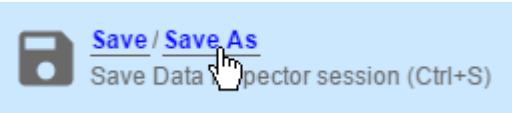
If you want to save or share data along with a configured view in the Simulation Data Inspector, save your data and settings in a Simulation Data Inspector session. You can save sessions as MAT- or MLDATX-files. The default format is MLDATX. When you save a Simulation Data Inspector session, the session file contains:

- All runs, data, and properties from the **Inspect** pane, including which run is the current run and which runs are in the archive.
- Plot display selection for signals in the **Inspect** pane.
- Subplot layout and line style and color selections.

Note Comparison results and global tolerances are not saved in Simulation Data Inspector sessions.

To save a Simulation Data Inspector session:

1. Hover over the save icon on the left side bar. Then, click **Save As**.



2. Name the file.
3. Browse to the location where you want to save the session, and click **Save**.

   For large datasets, a status overlay in the bottom right of the graphical viewing area displays information about the progress of the save operation and allows you to cancel the save operation.

The **Save** tab of the Simulation Data Inspector preferences menu on the left side bar allows you to configure options related to save operations for MLDATX-files. You can set a limit as low as 50MB on the amount of memory used for the save operation. You can also select one of three **Compression** options:

- `None`, the default, applies no compression during the save operation.
- `Normal` creates the smallest file size.
- `Fastest` creates a smaller file size than you would get by selecting `None`, but provides a faster save time than `Normal`.

To load a Simulation Data Inspector session, click the open icon  on the left side bar. Then, browse to select the MLDATX-file you want to open, and click **Open**.

Alternatively, you can double-click the MLDATX-file. MATLAB and the Simulation Data Inspector open if they are not already open.

When the Simulation Data Inspector already contains runs and you open a session, all of the runs in the session move to the archive. The view updates to reflect show plotted signals from the session file. You can drag runs between the work area and archive as desired.

When the Simulation Data Inspector does not contain runs and you open a session, the Simulation Data Inspector puts runs in the work area and archive as specified in the file.

Share Simulation Data Inspector Views

When you have different sets of data that you want to visualize the same way, you can save a view. A view saves the layout and appearance characteristics of the Simulation Data Inspector without saving the data. Specifically, a view saves:

- Plot layout, axis ranges, linking characteristics, and normalized axes.
- Location of signals in the plots, including plotted signals in the archive.
- Signal grouping and columns on display in the **Inspect** pane.
- Signal color and line styling.

To save a view:

1. Click the layout button  .
2. Click **Save current view**.
3. In the dialog box, specify a name for the view and browse to the location where you want to save the MLDATX-file.
4. Click **Save**.

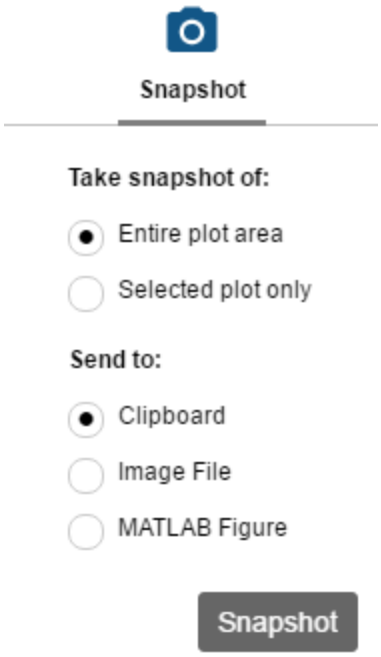
To load a view:

1. Click the layout button  .
2. Click **Open saved view**.
3. Browse to the view you would like to load, and click **Open**.

Share Simulation Data Inspector Plots

Use the snapshot feature to share the plots you generate in the Simulation Data Inspector. You can export your plots to the clipboard to paste into a document, as an image file, or to a MATLAB figure. You can choose to capture the entire plot area, including all subplots in the plot area, or to capture only the selected subplot.

Click the camera icon on the toolbar to access the snapshot menu. Use the radio buttons to select the area you want to share and how you want to share the plot. After you make your selections, click **Snapshot** to export the plot.



If you create an image, select where you would like to save the image in the file browser.

You can create snapshots of your plots in the Simulation Data Inspector programmatically using `Simulink.sdi.snapshot`.

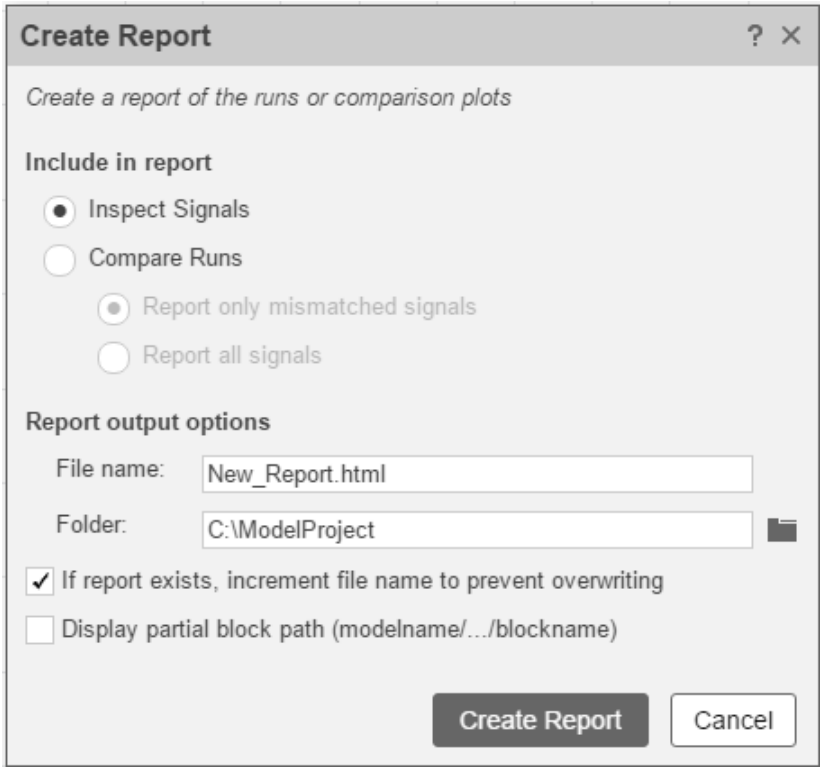
Create a Simulation Data Inspector Report

To generate documentation of your results quickly, create a Simulation Data Inspector report. You can create a report of your data in either the **Inspect** or the **Compare** pane. The report is an HTML file that includes information about all the signals and plots in the active pane. The report includes all signal information displayed in the signal table in the navigation pane. For more information about configuring the table, see “Inspect Metadata” on page 29-115.

To generate a Simulation Data Inspector Report:

1. Click the create report icon on the left side bar.
2. Under **Include in report**, specify the type of report you want to create.
   - Select **Inspect Signals** to include the plots and signals from the **Inspect** pane.
   - Select **Compare Runs** to include the data and plots from the **Compare** pane. When you generate a **Compare Runs** report, you can choose to **Report only mismatched signals** or

to **Report all signals**. If you select **Report only mismatched signals**, the report shows only signal comparisons that are not within the specified tolerances.



3 Specify a **File name** for the report, and navigate to the **Folder** where you want to save the report.

4 Click **Create Report**.

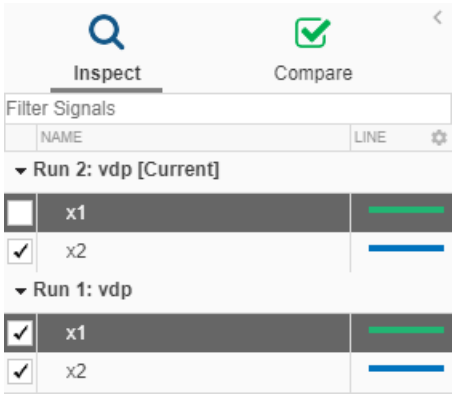
The generated report automatically opens in your default browser.

Export Data from the Simulation Data Inspector

You can use the Simulation Data Inspector to export data to the base workspace, a MAT-file, or a Microsoft Excel file. You can export a selection of runs and signals, runs in the work area, or all runs in the **Inspect** pane, including the **Archive**.

When you export a selection of runs and signals, make the selection of data to export before clicking the export button. 

Only the selected runs and signals are exported. In this example, only the `x1` signals from Run 1 and Run 2 are exported. The check box selections for the plotting area do not affect whether a signal is exported.



When you export a single signal, the signal is stored in `timeseries` format in the workspace variable or MAT-file. Exported data for a run or multiple signals is stored in `Simulink.SimulationData.Dataset` format.

Note When you export a run that contains logged parameter data, the exported `Simulink.SimulationData.Dataset` contains a `Simulink.SimulationData.Parameter` element for each logged parameter.

To export data to a file, select the **File** option in the **Export** dialog. You can specify a file name and browse to the location where you want to save the exported file. When you export data to a MAT-file, a single exported signal is stored in `timeseries` format, and runs or multiple signals are stored in `Simulink.SimulationData.Dataset` format. When you export data to a Microsoft Excel file, the data is stored in the format described in “Microsoft Excel Import and Export Format” on page 29-43.

To export to a Microsoft Excel file, select the XLSX extension from the drop-down. When you export data to a Microsoft Excel file, you can specify additional options for the format of the data in the exported file. If the file name you provided already exists, you can choose to overwrite the entire file or to only overwrite sheets containing data that corresponds to the exported data. You can also choose which metadata to include and whether signals with identical time data share a time column in the exported file.

See Also

Related Examples

- “Organize Your Simulation Data Inspector Workspace” on page 29-144
- “View Data in the Simulation Data Inspector” on page 29-2
- “Inspect Simulation Data” on page 29-107
- “Compare Simulation Data” on page 29-130

Create an Interactive Comparison Report

When you compare data using the Simulation Data Inspector, you can create an interactive web-based comparison report to share or archive the results. The report is a self-contained HTML file, and viewing the report only requires a web browser. You can use the report to analyze the comparison results similarly to how you would in the Simulation Data Inspector. You can change the layout of the report between the interactive view and a printable view.

This example shows how to create an interactive comparison report using data loaded from a session file. For more information about importing data into the Simulation Data Inspector or logging data from a simulation, see “View Data in the Simulation Data Inspector” on page 29-2.

Load and Compare Data

This example uses data created by simulating the `slexAircraftExample` model that was saved in the session file `dataToCompare.mldatx`. Both simulations used the square wave input from the Pilot block in the model. The first simulation uses the input filter design saved in the model with a time constant of `0.1`. The time constant was changed to `1` for the second simulation.

Open the Simulation Data Inspector and the session file to load the data into the Simulation Data Inspector for comparison.

```
Simulink.sdi.view
Simulink.sdi.load('dataToCompare.mldatx');
```

In the Simulation Data Inspector, navigate to the **Compare** tab and compare the two runs.

1. Expand the **Baseline** drop-down and select `Run 1: slexAircraftExample`.
2. Expand the **Compare to** drop-down and select `Run 2: slexAircraftExample`.
3. Click **Compare**.

Alternatively, you can use the `Simulink.sdi.compareRuns` function to perform the comparison.

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

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

Generate the Interactive Comparison Report

To save the comparison results, you can generate an interactive web report. To create the report, click the **Report** button.

In the Create Report dialog, you can specify the type of report you want to create and the data you want to include in the report. Select `Compare` as the **Type**. In this example, all signals in the run comparison did not match, so a report containing only mismatched signals contains all the signals.

You can specify a title and author to include in the header for the report. By default, the title is created from the names of the compared runs. This example provides a different title that describes the variable values evaluated by the simulations and comparison.

Enter a descriptive file name for the report file. You can also specify where to save the report. By default, the Simulation Data Inspector saves the report in a folder called `sdireports` in the working directory.

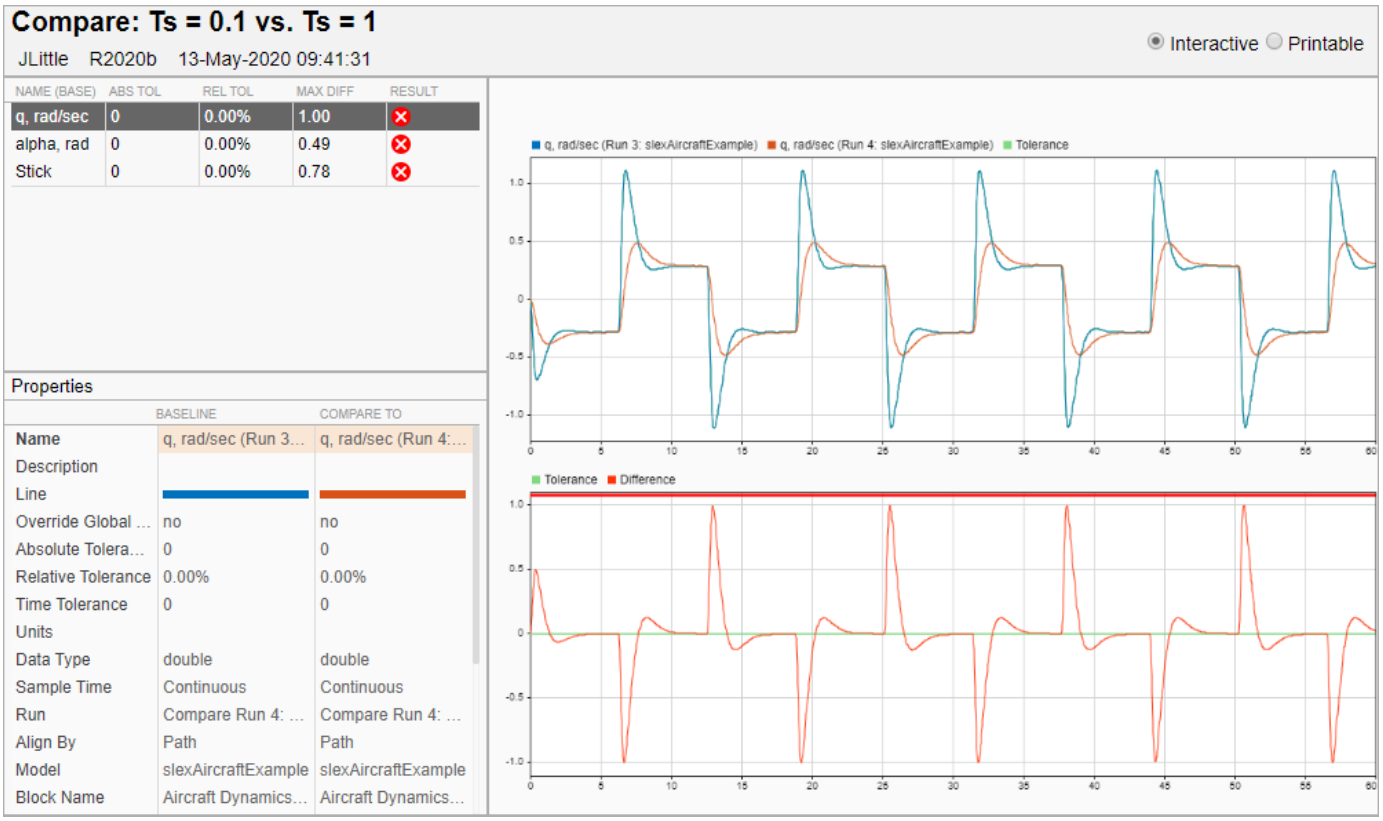
You can also enable the option to add an incrementing number to the end of the specified file name to avoid overwriting existing report files. For models that log signals throughout the model hierarchy, consider enabling the **Display partial block path** option to prevent long block paths from appearing in the report. In this example, the signals are all logged in the top-level of the model hierarchy.

After specifying the options for the report you want to create, click **Create Report**. The generated report automatically opens in the system browser.

Inspect Comparison Results in the Interactive Report

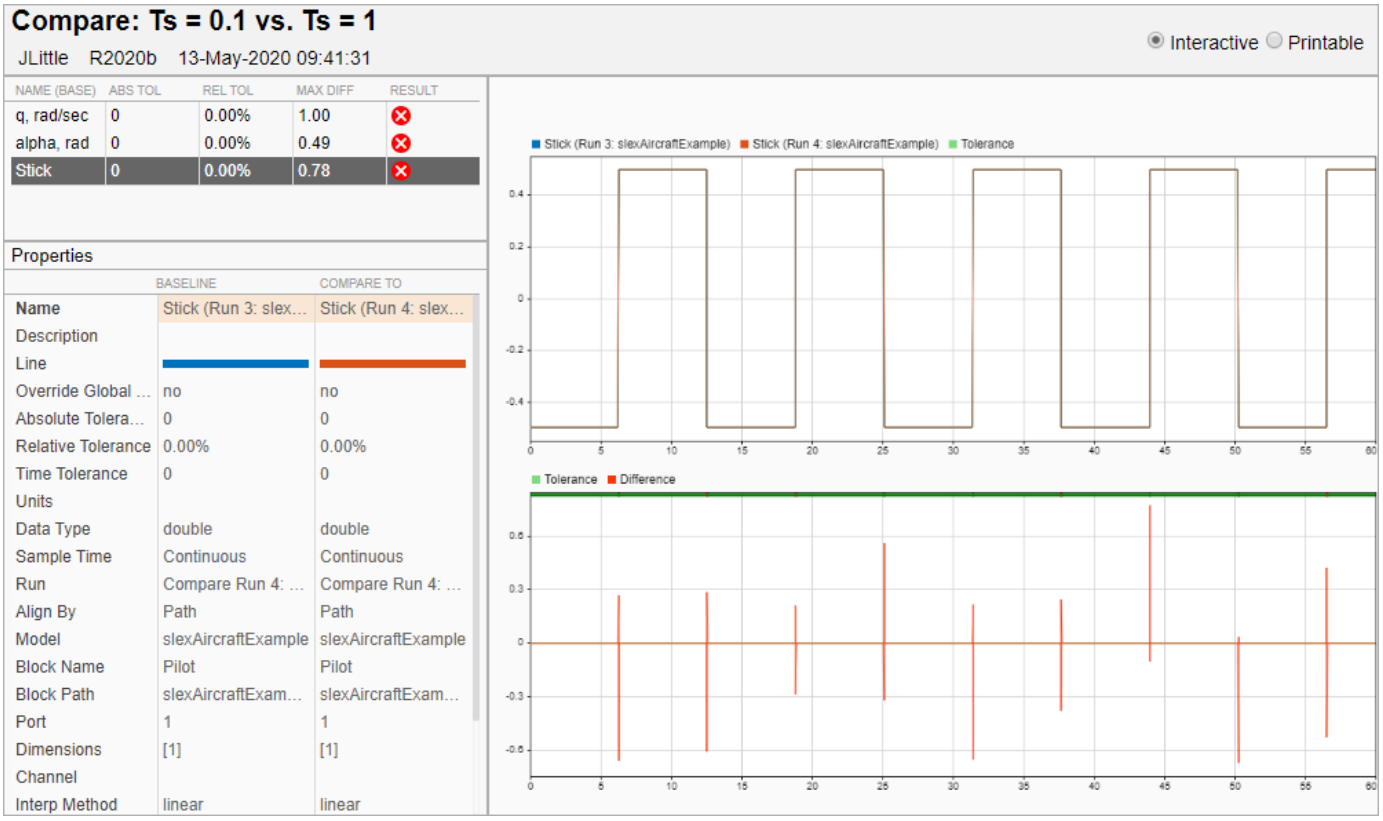
The interactive report information and layout is similar to what you see on the **Compare** pane of the Simulation Data Inspector. In the **Comparisons** table on the left, you can select the signal result you want to view in the plots. The **Properties** table shows all the metadata for the compared signals and highlights metadata that differs. You can scroll in the **Properties** table to see all the metadata. When

the comparison contains more signals than fit in the results table, you can scroll in the results table as well.



You can adjust the size of the panes in the report by dragging the borders. For example, you could make the **Properties** pane larger, since the table of compared signals does not use all the space in the signals pane.

The information in the table indicates that the comparison results for the `Stick` signal are out of tolerance with a maximum difference of `0.78`. In the model, the `Stick` signal is the output from a Signal Generator block, and the filter time constant should not affect the block output. Select the `Stick` signal to investigate the comparison results.



Inspecting the plots in the report allows you to see that the differences occur at the rising and falling edges of the square wave. The `slexAircraftExample` model uses a variable-step solver. The differences occur because the new time constant value changes the system dynamics, and the solver calculates slightly different time steps for the second simulation. The transition of the square wave is sharp, so small timing differences can result in large magnitude differences.

Because the differences in the `Stick` signal are due to modeling rather than a real change in system dynamics, you could go back to the Simulation Data Inspector and specify a small time tolerance (For example, `0.01`) for the `Stick` signal so the comparison results are within tolerance. To share the updated results, you can generate a new comparison report.

Print Comparison Report

After finalizing the analysis and presentation of the comparison results, you can print a comprehensive view of the results using the interactive web report. To switch to the printable view, select the **Printable** radio button, or press Ctrl+P to see a print preview in the browser. In the printable view, the metadata and plots for all signals are stacked, with the summary table at the top.

See Also

`Simulink.sdi.report`

Related Examples

- “View Data in the Simulation Data Inspector” on page 29-2
- “Compare Simulation Data” on page 29-130

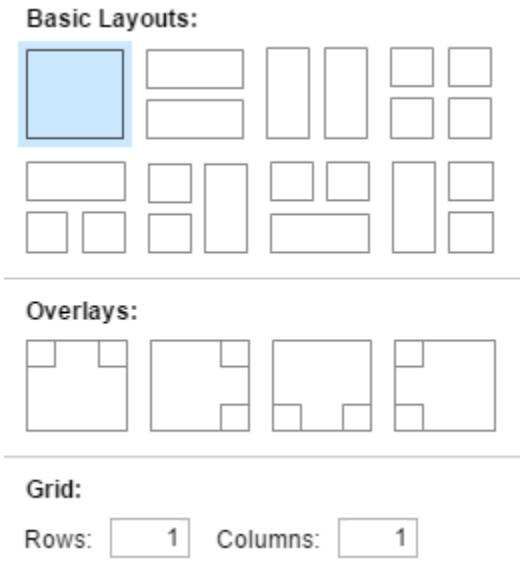
- “Save and Share Simulation Data Inspector Data and Views” on page 29-83

Create Plots Using the Simulation Data Inspector

Plots can help showcase important features or trends in your data and allow you to share your findings with others. The Simulation Data Inspector allows you to select from a variety of visualization types and layouts and customize plot and signal appearances to present your data most effectively. This topic uses simulation data generated from a simulation of the `slexAircraftExample` model that logs the output of the Actuator Model block and the `Stick`, `alpha, rad`, and `q, rad/sec` signals.

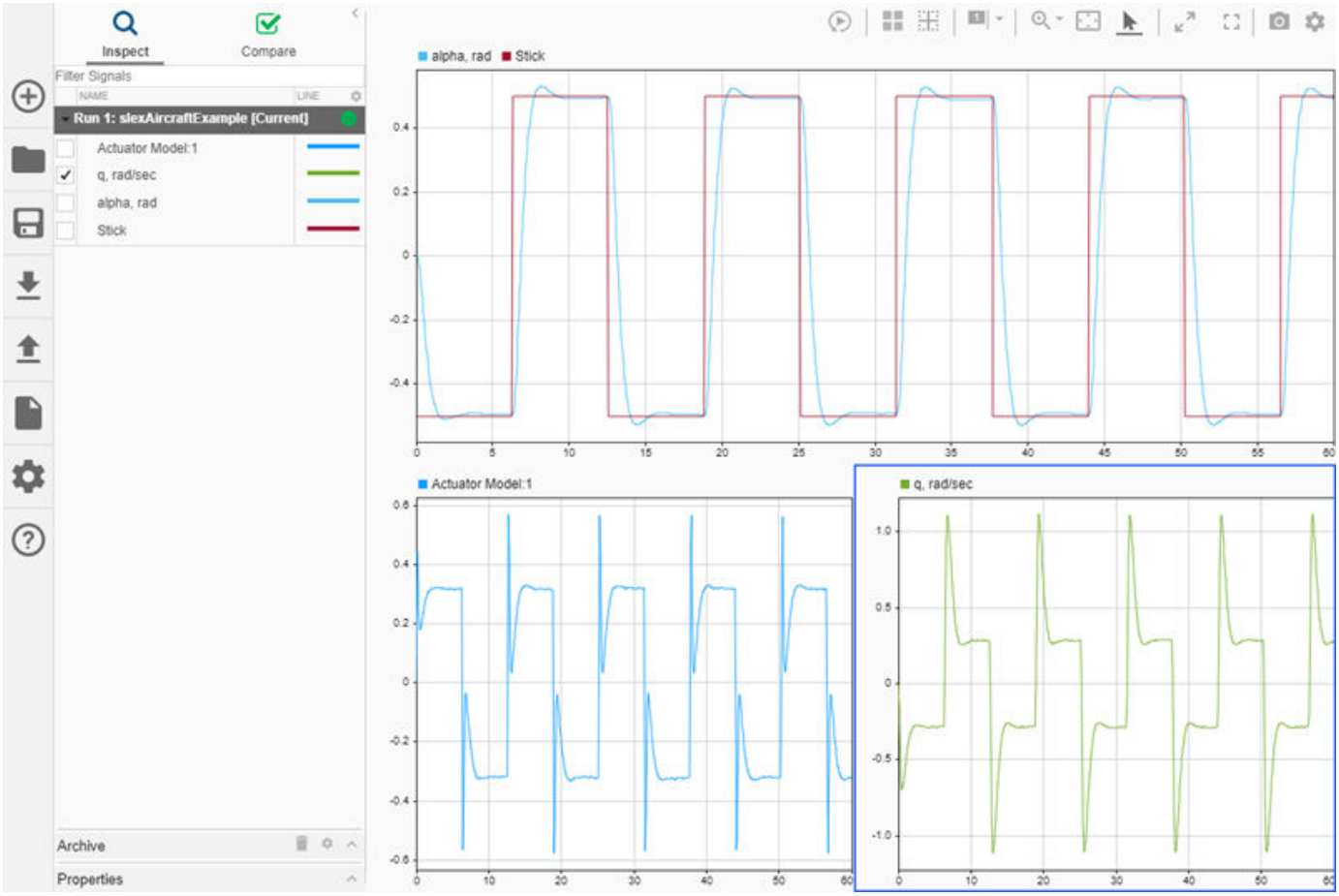
Select a Plot Layout

You can select from three types of layouts under the layout menu in the Simulation Data Inspector  .



- **Basic Layouts** offer templates for layouts including up to four subplots.
- **Overlays** have overlay subplots in two corners of a main plot.
- **Grid** layouts create a grid of subplots according to dimensions you specify from `1×1` to `8×8`.

Select the plot layout that best highlights the characteristics of your data. For example, with a basic three-plot layout, you can use the large plot to show a main result and show intermediate signals on the smaller plots.



Moving Between Subplot Layouts

When you plot a signal on a subplot, the Simulation Data Inspector links the signal to the subplot, identifying the subplot with a number. As you move between subplot layouts, the association between the signal and subplot remains, even as the shape or visibility of the subplot may change.

Subplots of **Grid** layouts follow columnwise numbering, from `1` at the top left subplot to `8` in the bottom left, to `64` at the bottom right. The number system is fixed, regardless of which subplot configuration you select. For example, if you display four subplots in a `2×2` configuration, the bottom right subplot of the configuration is numbered `10`, not `4`. The number is `10` because in the largest possible `8×8` matrix, that subplot is numbered `10` when you count the subplots in a columnwise manner.

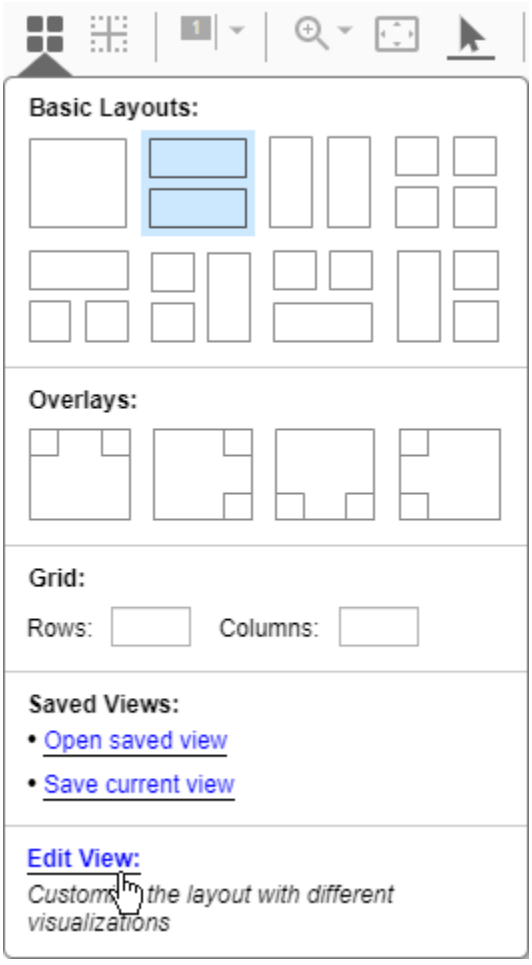
Basic Layouts subplots also use the fixed `8×8` columnwise numbering as a base. For example, the top plot in this three-subplot layout is subplot `1`, and the two subplots below it are `2` and `10`.

In this three-subplot layout, the index for the right subplot is `9`, and the indices for the two subplots on the left are `1` and `2`.

For the **Overlays** layouts, the index for the main plot is always 1, and the indices for the overlaid plots are 2 and 9.

Add Visualizations

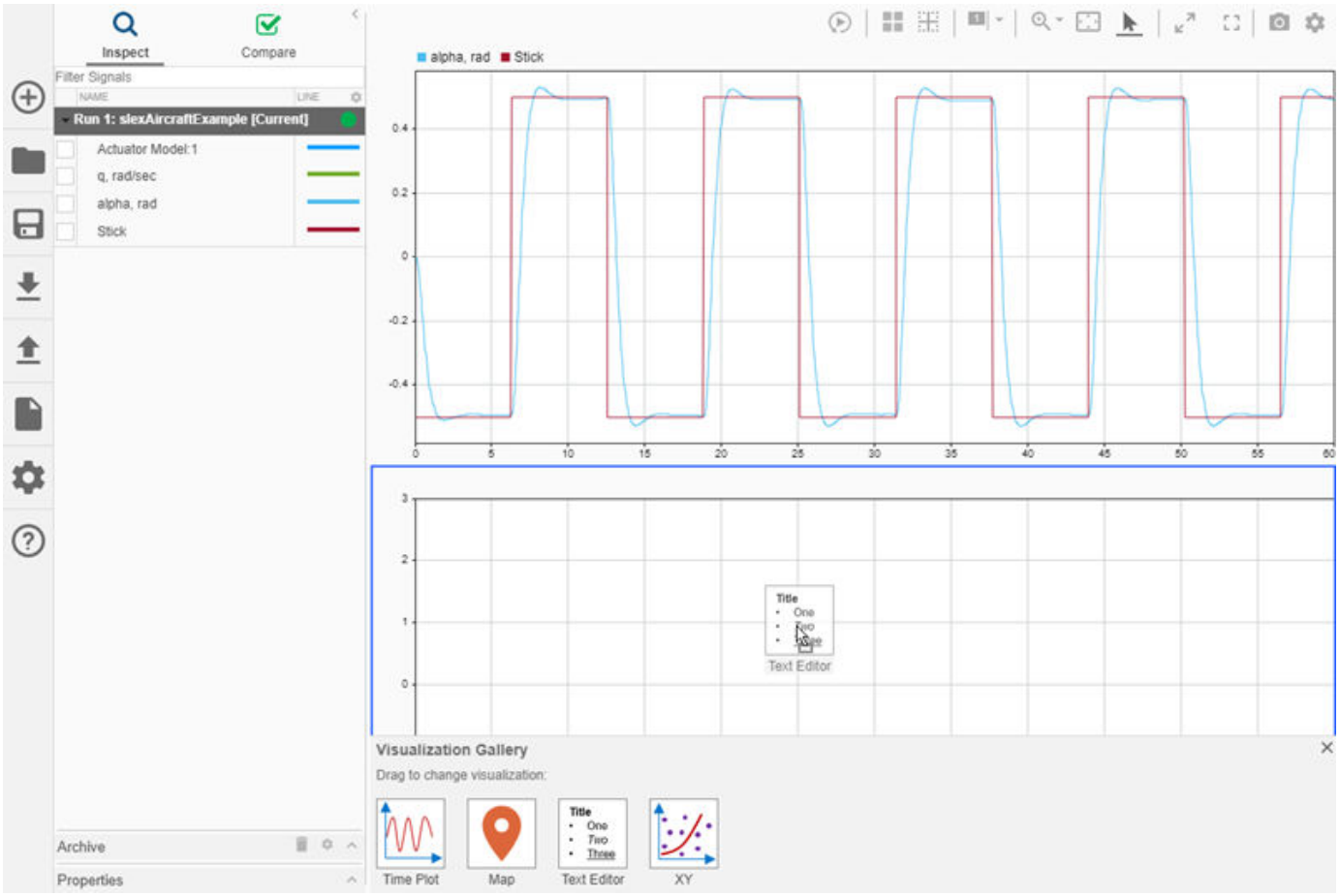
You can choose from several visualizations to use for your data in the Simulation Data Inspector. To view available visualizations, open the **Visualization Gallery** by clicking **Edit View** on the layout menu.



You can drag the visualization you want to add to the plot where you want to add it. You can choose a **Time Plot**, **Map**, **Text Editor**, or **XY** visualization for each subplot in your layout. For more information about using **Map** and **XY** visualizations, see

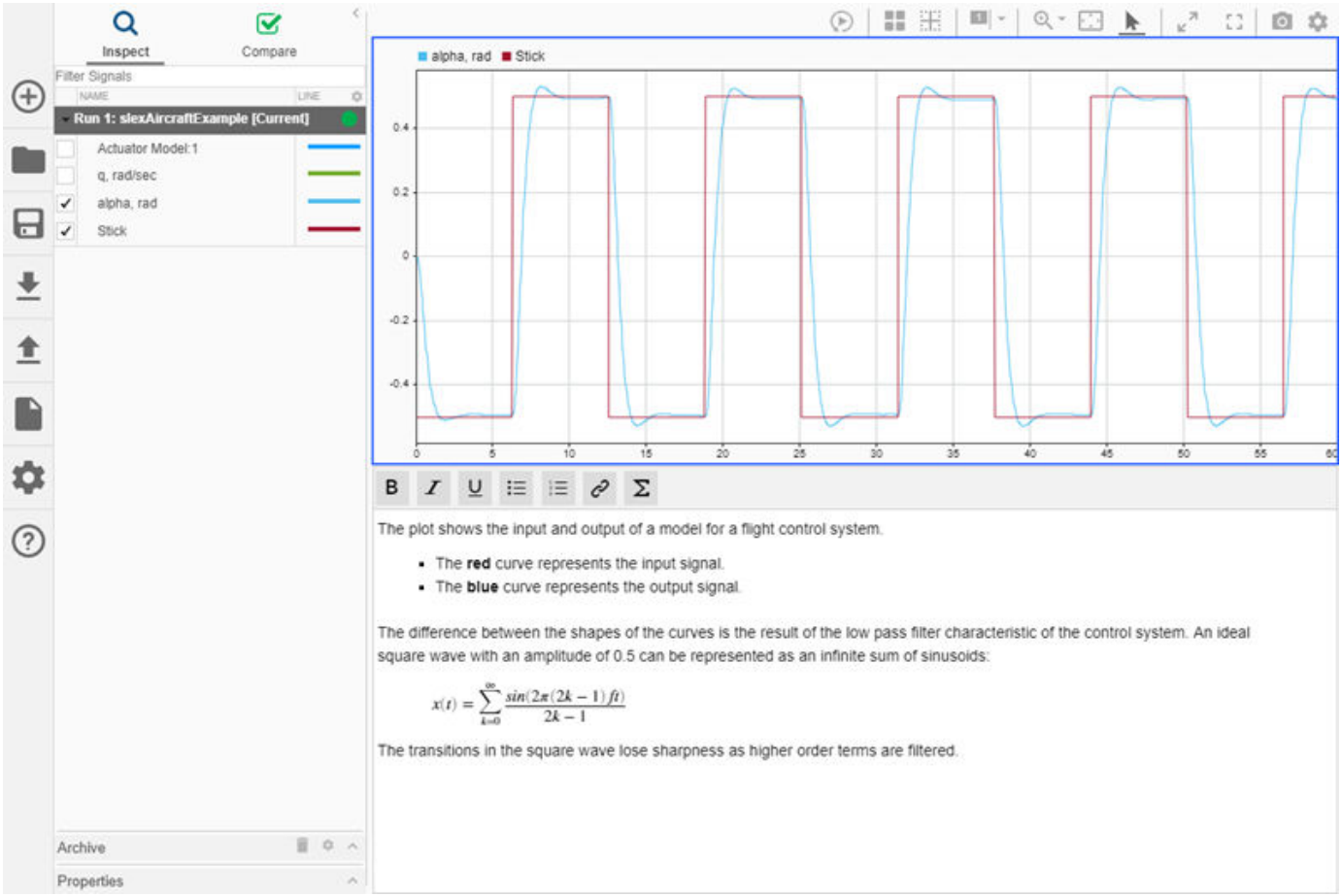
- “View and Replay Map Data” on page 29-22
- “Visualize Simulation Data on an XY Plot” on page 29-29

- “Analyze Data Using the XY Visualization” on page 29-38



Add a text editor to your plot layout when you want to present conclusions, descriptions, or observations alongside your data. The text editor includes a toolbar for rich text formatting, including:

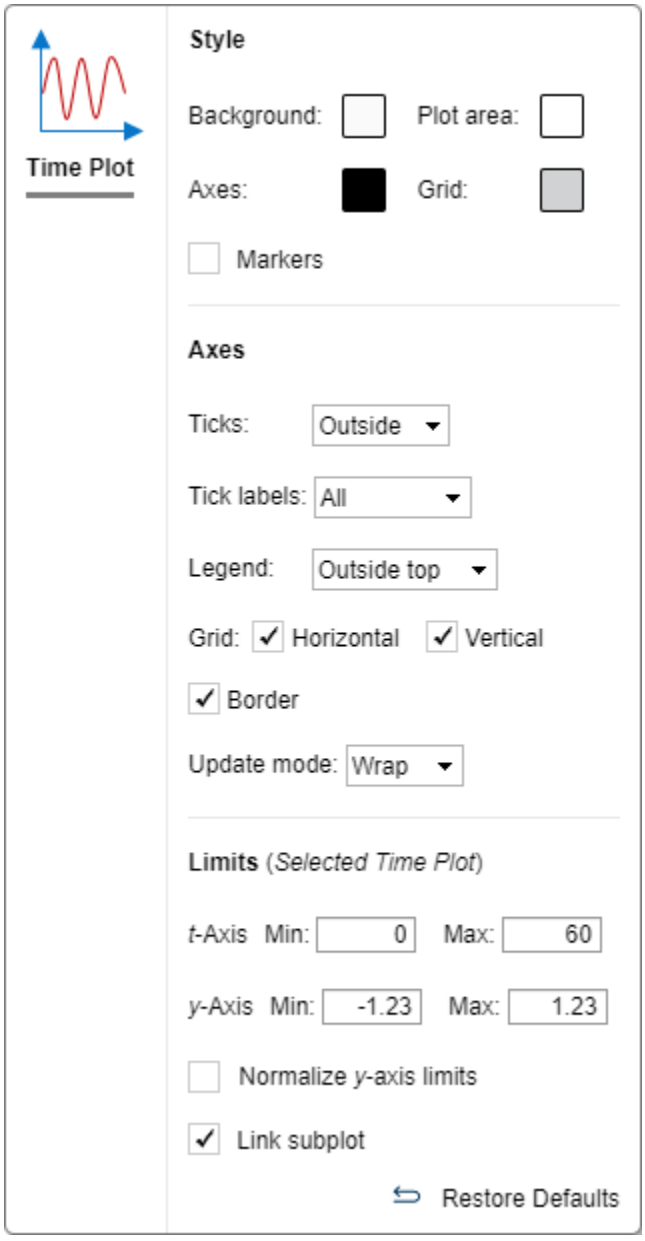
- Bullet points
- Numbered lists
- Hyperlinks
- LaTeX equations



Customize Time Plot Appearance

After you choose a plot layout, you can customize the appearance of each visualization using the visualization settings menu . The number of menu tabs depends on the visualizations present in your layout. When your layout does not include a **Map** or **XY** visualization, the corresponding settings tabs are not shown.

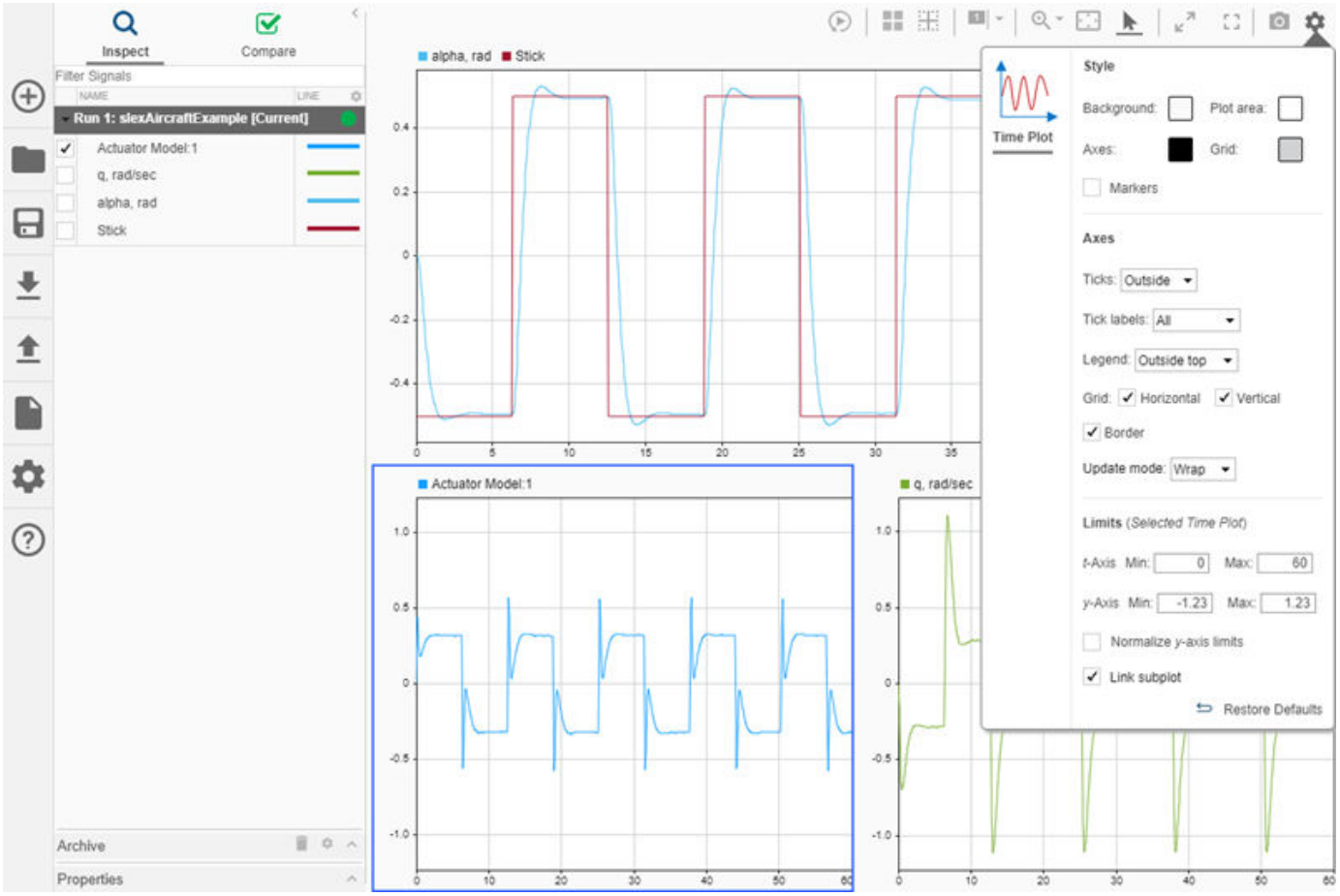
You can customize the appearance of **Time Plot** visualizations in the Simulation Data Inspector, including choosing custom colors, choosing to show or hide attributes such as the grid or legend, and specifying axes limits. To maximize the area in the visualization available for the plot, you can move the tick marks, tick mark labels, and legend inside the plot or hide them.



The **Limits** section gives you control over the limits of the *t*- and *y*-axes for individual subplots. You may also choose to normalize the *y*-axis. Except for the *t*-axis limits, the settings you configure in the **Limits** section apply to the active subplot. To configure *t*-axis limits individually for a subplot, unlink the subplot. For more information, see “Linked Subplots” on page 29-111.

To improve the basic layout plot created in the previous section, change the *y*-axis limits of the left graph to match those on the right.

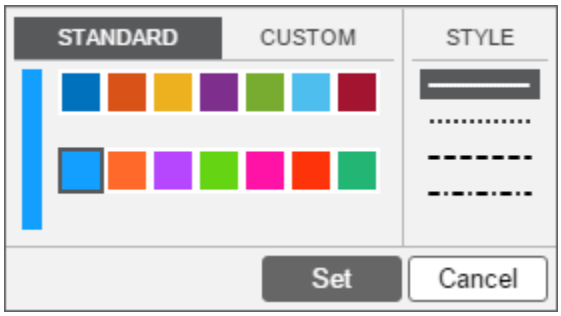
1. Select the `q, rad/sec` plot to view its *y*-axis limits in the **Limits** tab.
2. Select the `Actuator Model: 1` plot as the active plot.
3. In the **Limits** section, change the *y*-axis limits of the `Actuator Model: 1` plot to match the limits of the `q, rad/sec` plot.



You can clear the active subplot or all subplots using the clear subplots menu on the toolbar.

Customize Signal Appearance

The Simulation Data Inspector allows you to modify the color and style of each signal individually. For each signal, you can select from a palette of standard colors or specify a custom color with RGB values.



You can access the line style menu for a signal in the Simulation Data Inspector or in the Simulink Editor:

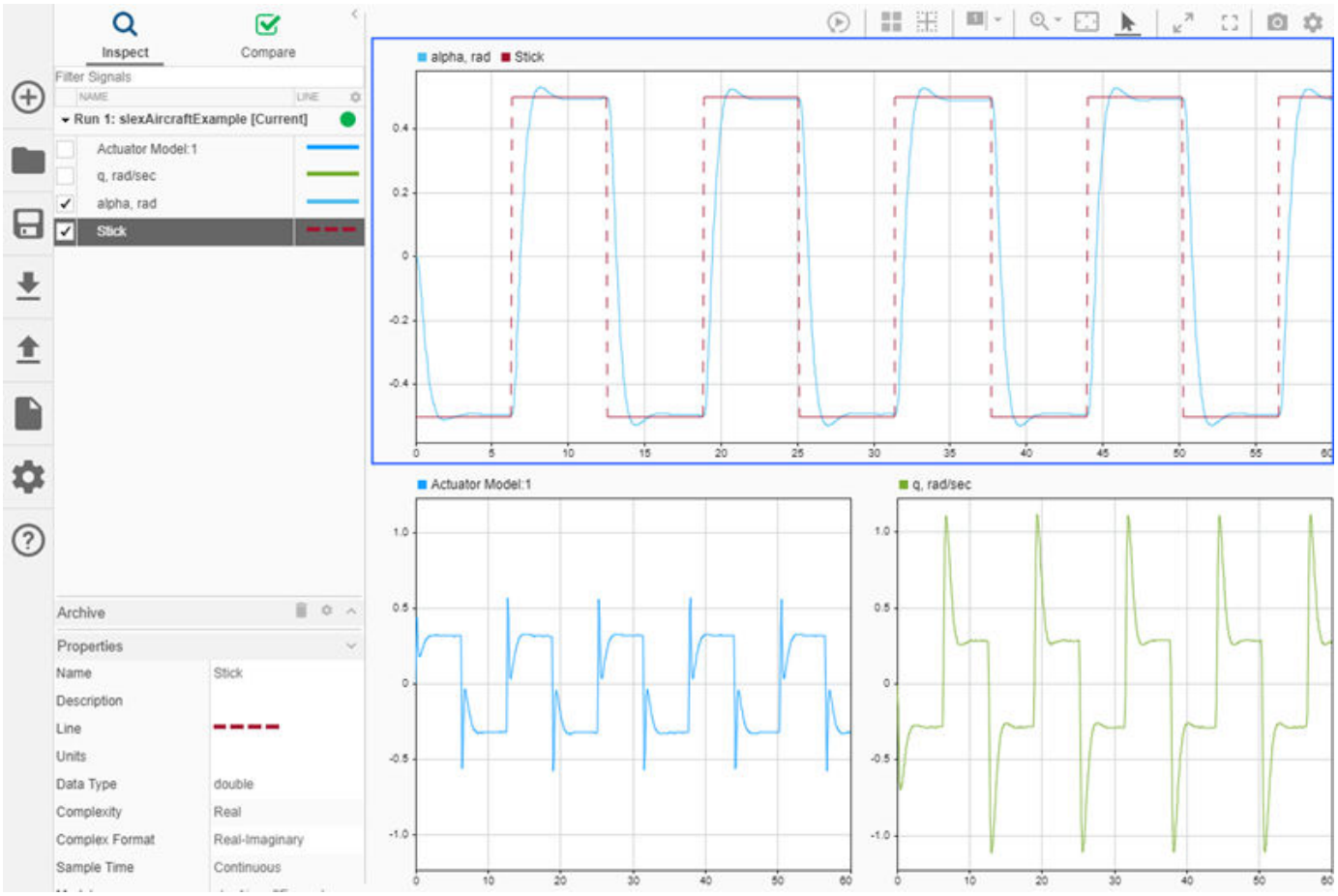
- In the Simulation Data Inspector, click the graphic representation in the **Line** column of the work area or the **Line** row in the **Properties** pane.
- From the Simulink Editor, right-click the logging badge for a signal, and select **Properties** to open the **Instrumentation Properties** for the signal. Then, click the graphic representation of the **Line**.

  From the **Instrumentation Properties**, you can also select subplots where you want to plot the signal. Changes made through the **Instrumentation Properties** take effect for subsequent simulations.

If a signal is connected to a Dashboard Scope block, you can also modify the signal line style and color through the Dashboard Scope block dialog box. Changes to line style and color made in the Simulink Editor remain consistent for a signal. When you change the line style and color in the Simulation Data Inspector, the signal line style and color do not change in the Simulink Editor.

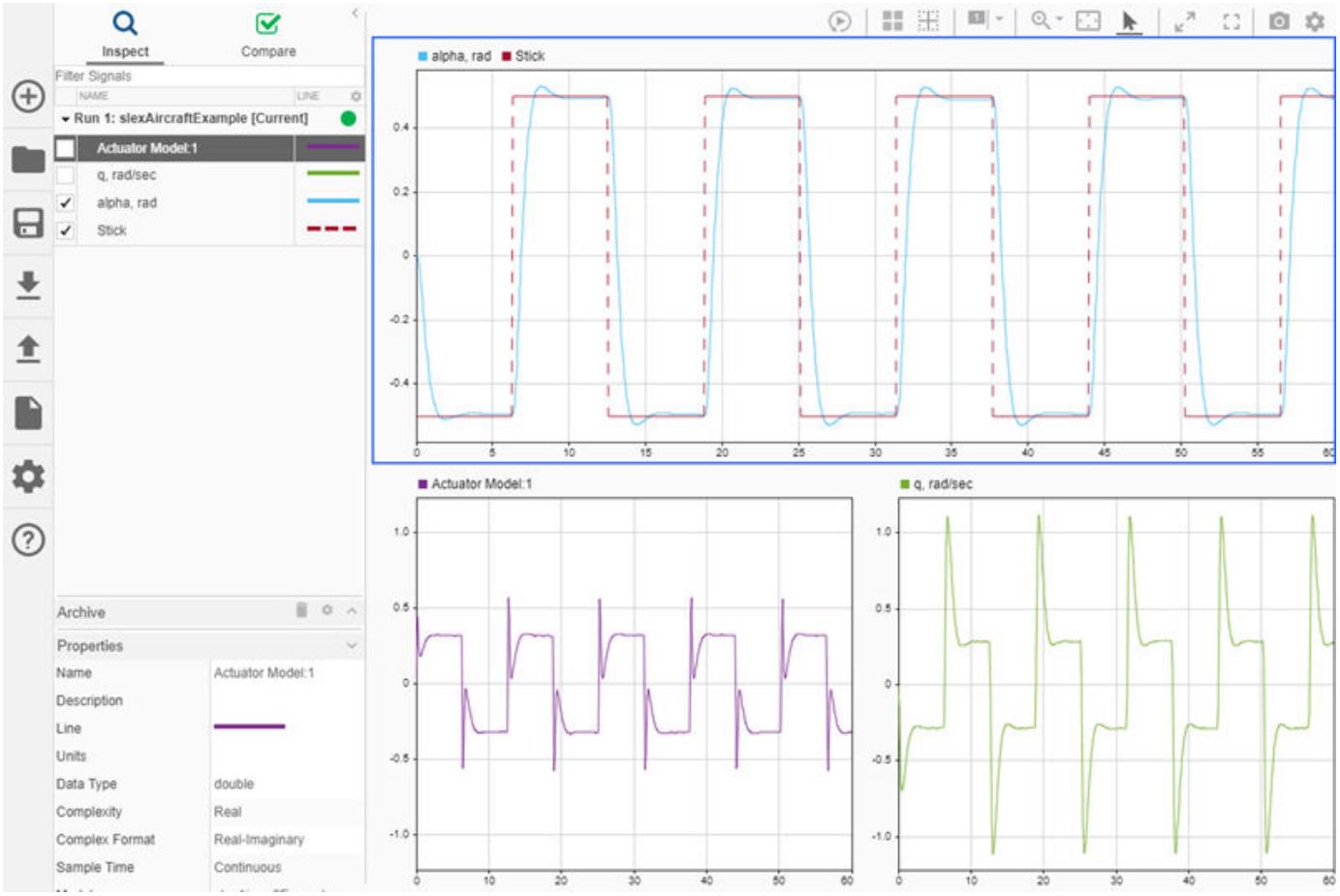
Line style customization can help emphasize differences between signals in your plot. For example, change the line style of the `Stick` signal to dashed to visually indicate that it is an ideal, rather than simulated, signal.

1. Click the **Line** column for the `Stick` signal.
2. Select the dashed option (third in the list).
3. Click **Set**.



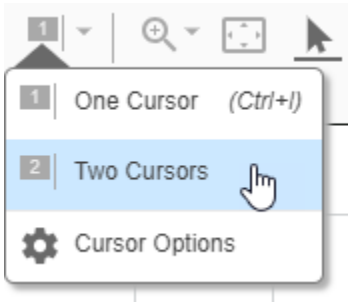
The `alpha, rad` and `Actuator Model: 1` signals are very close in color. You can select bold, easily distinguishable colors for all of the signals in your plot.

1. Select the `Actuator Model: 1` signal in the work area.
2. Click the **Line** row entry in the **Properties** pane.
3. Specify the desired color. This example uses the dark purple (fourth option) in the standard palette.
4. Click **Set**.

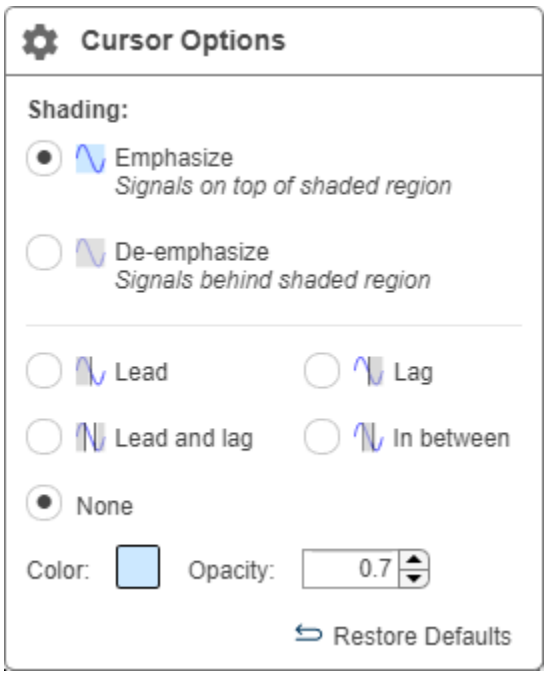


Shade Signal Regions

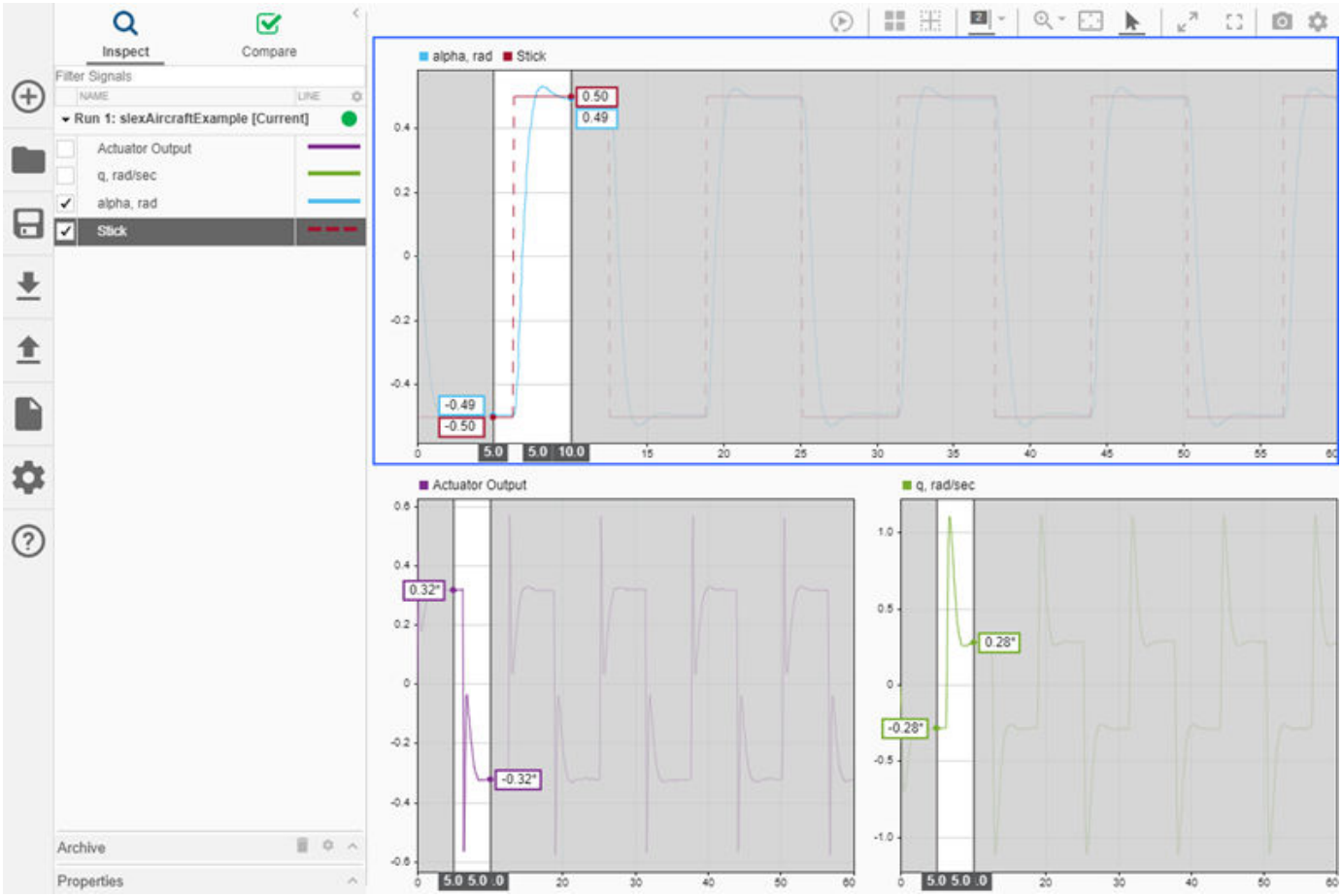
You can shade an area of a time plot to draw attention to a region of interest in the plotted data. For example, you can highlight the area around a signal transition. First, add two cursors to the plot area by clicking the arrow next to the **Show/hide cursors** button and selecting **Two cursors**.



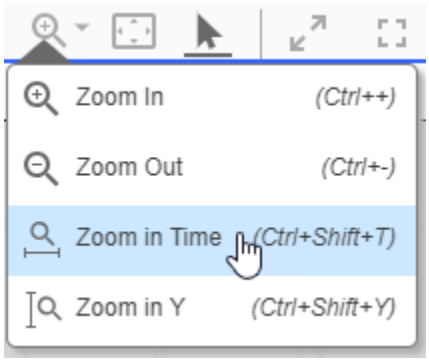
Next, click the arrow again and configure the **Cursor Options** to match your requirements. You can specify whether to emphasize or de-emphasize the shaded area, the area to shade relative to the cursors, and the shading color and opacity. You can specify separate color and opacity settings for the **Emphasize** and **De-emphasize** options.



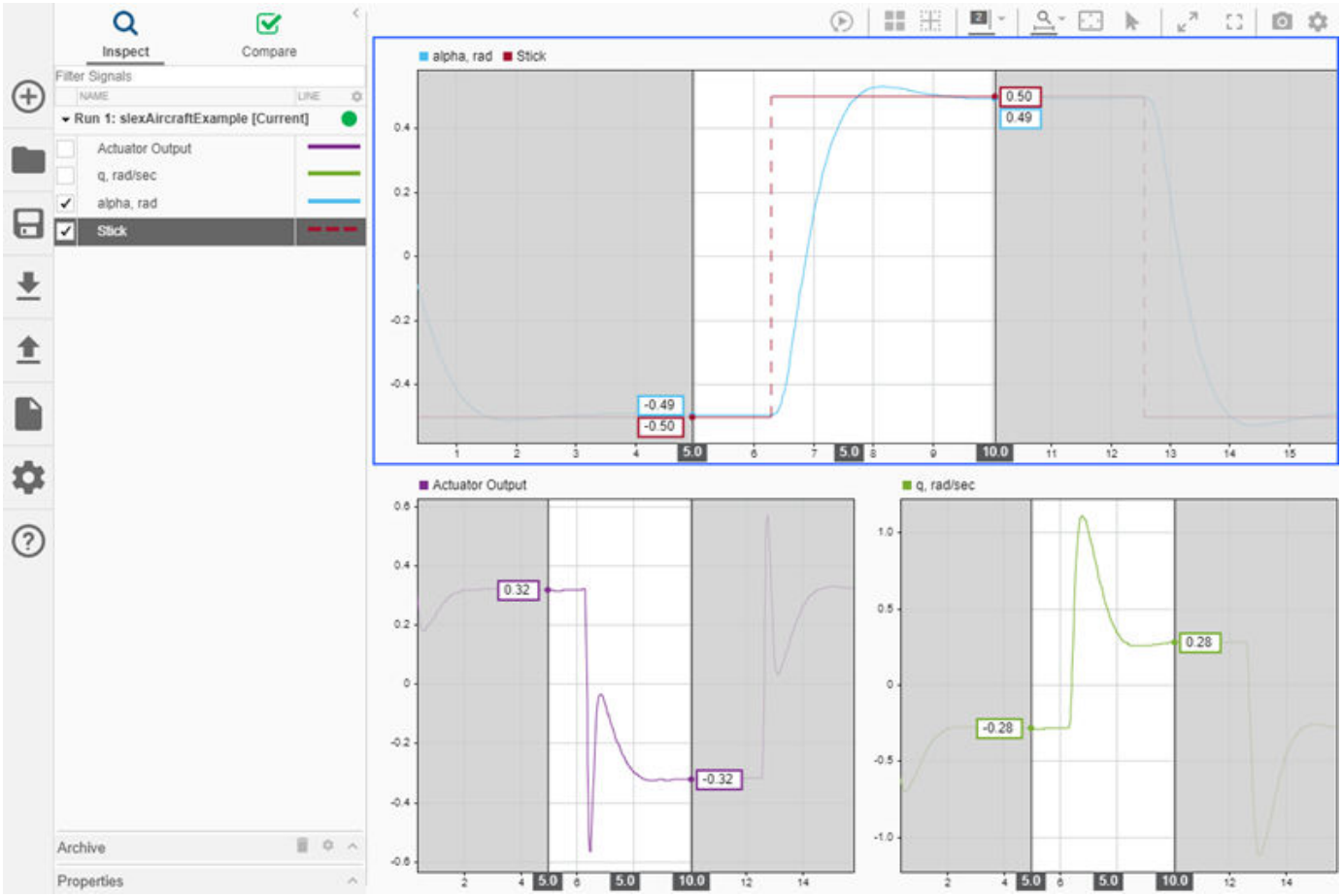
For this example, select the **De-emphasize** and **Lead and lag** options. Click in the plot area to close the **Cursor Options** dialog. The gray shading obscures the area outside of the cursors and highlights the signal region between the cursors. Move the cursors to highlight an area of interest, like the first rising edge of the waveform.



You can use the **Zoom in Time** option to zoom in synchronously on the region of interest. Select **Zoom in Time** from the **Zoom In** menu.



Then click and drag to select a time span.

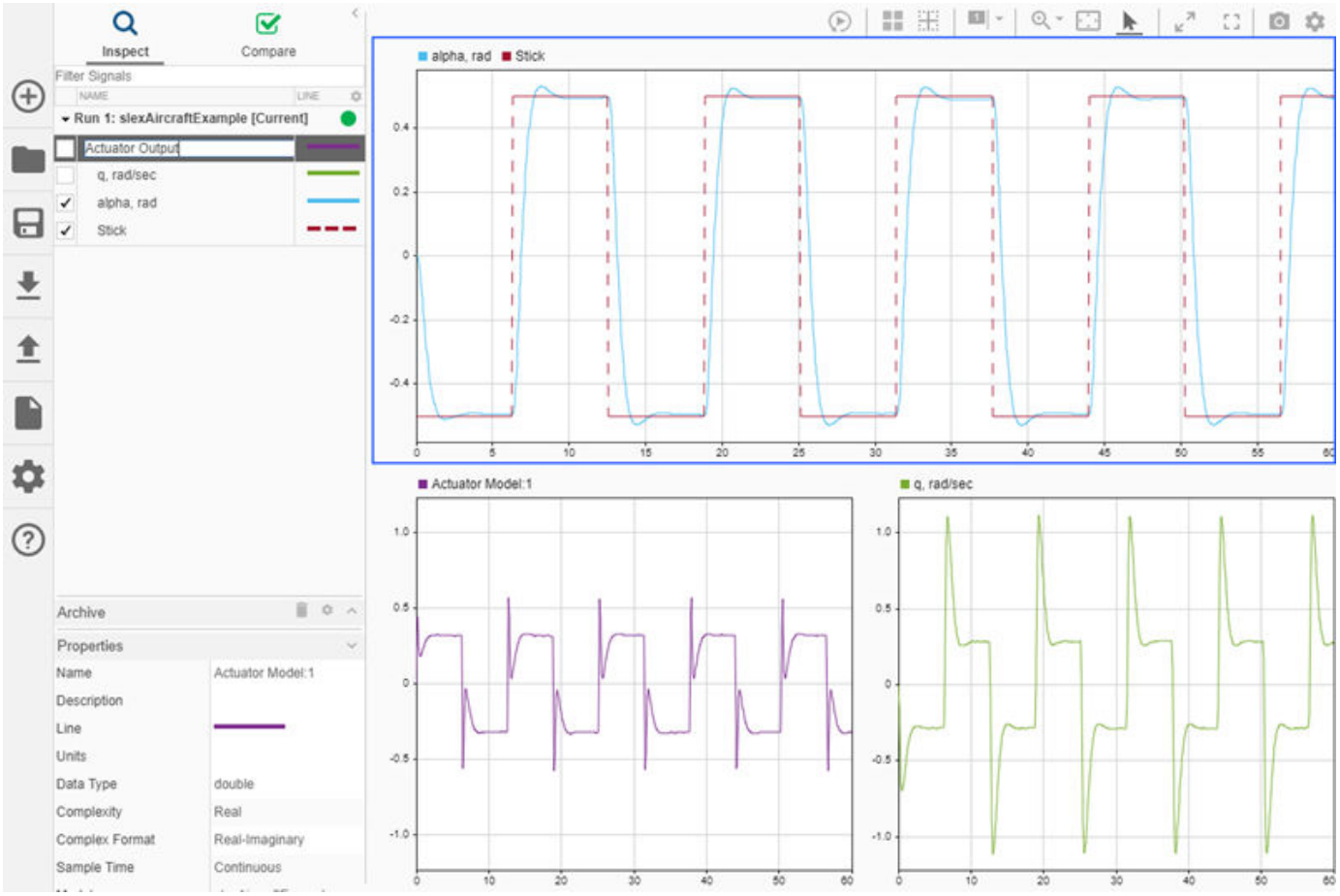


You can use the **Snapshot** menu to save snapshots of highlighted signal regions. For more information, see “Save and Share Simulation Data Inspector Data and Views” on page 29-83.

Rename Signals

You can rename signals in the Simulation Data Inspector when a more descriptive name can help convey information. Changing a signal name in the Simulation Data Inspector does not affect signal names used in models. You cannot rename bus signals in the Simulation Data Inspector.

To change a signal name, double-click the signal name in the work area or archive, or edit the **Name** field in the **Properties** pane. Change the name of the `Actuator Model: 1` signal to `Actuator Output`.



See Also

`Simulink.sdi.Signal` | `Simulink.sdi.setSubPlotLayout` | `plotOnSubPlot`

More About

- “Save and Share Simulation Data Inspector Data and Views” on page 29-83
- “Inspect Simulation Data” on page 29-107
- “Compare Simulation Data” on page 29-130

Inspect Simulation Data

You can use the Simulation Data Inspector to view and inspect signals from simulations or imported data. The Simulation Data Inspector provides a comprehensive view of your data by allowing you to group data from multiple simulations and sources on multiple subplots. You can zoom and pan within plots and use data cursors for close examination of signal values, and you can replay data to analyze signal relationships.

This example shows you how to view and inspect signal data from the `slexAircraftExample` model using the Simulation Data Inspector.

Configure Signals for Logging

This example uses signal logging to send data to the Simulation Data Inspector. You can also import signal data from the base workspace or a CSV, MDF, Microsoft Excel, or MAT-file. For more information, see “View Data in the Simulation Data Inspector” on page 29-2.

Open the `slexAircraftExample` model, mark several signals for logging, and run a simulation.

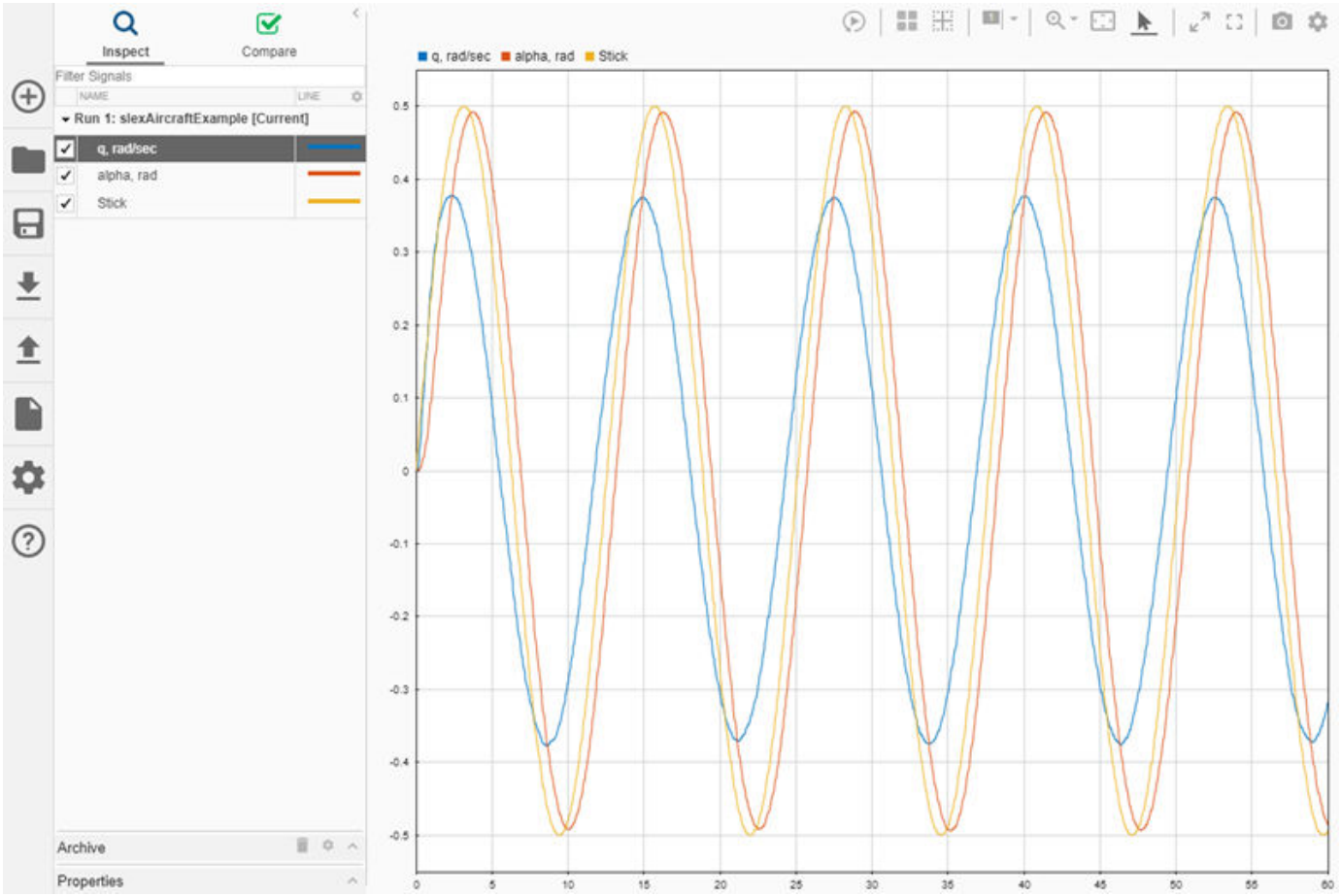
1. To open the model, enter `slexAircraftExample` in the MATLAB Command Window.
2. To log the `q, rad/sec`, the `Stick`, and the `alpha, rad` signals, select the signals in the model. Then, click **Log Signals**.

   The logging badge appears above each signal marked for logging.
3. Double-click the Pilot block. Set **Wave form** to `sine`, and click **OK**.
4. In the Simulink Editor, click the **Data Inspector** button to open the Simulation Data Inspector.
5. Simulate the model. The simulation run appears in the Simulation Data Inspector.

   By default, the **Inspect** pane lists all logged signals in rows, organized by simulation run. You can expand or collapse any of the runs to view the signals in a run. For more information on signal grouping, see “Organize Your Simulation Data Inspector Workspace” on page 29-144.

View Signals

To select signals you want to plot, use the check boxes next to the signals in the navigation pane. Select the check boxes next to the `q, rad/sec`, `Stick`, and `alpha, rad` signals. When the signal you want to plot is easier to find in your model, you can click the logging badge for the signal to plot it in the Simulation Data Inspector.



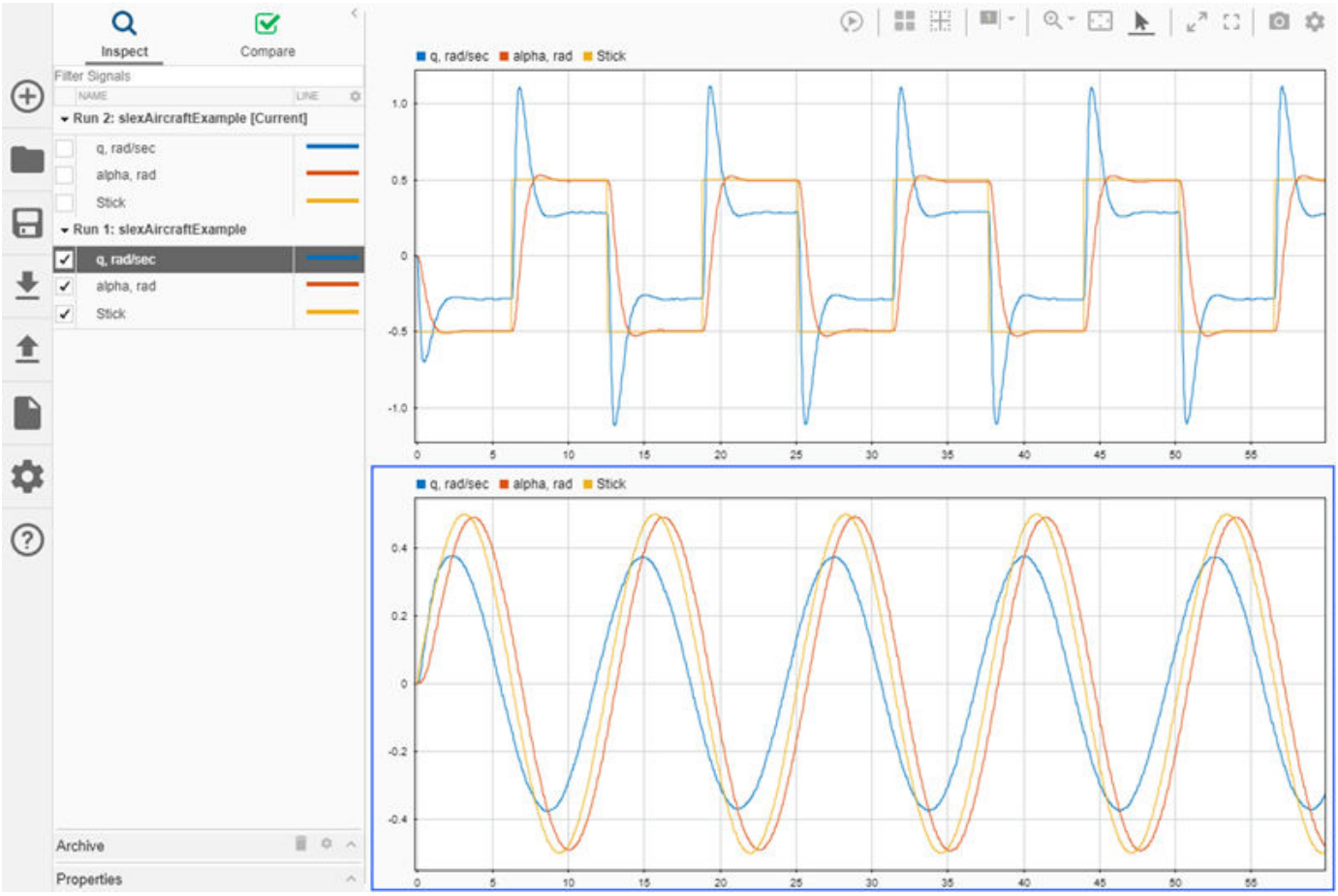
You can also use signal browsing mode to quickly view all the signals in your run on **Time Plot** visualizations. For more information about how to enable and use **Browse Mode**, see “Visualize Many Logged Signals” on page 29-76.

View Signals on Multiple Plots

You can use subplot layouts to view groups of signals on different subplots. For example, you can group the same signal from different simulation runs or group signals that have a similar range of values.

1. In the model, double-click the Pilot block. Set **Wave form** to `square`, and click **OK**.
2. Simulate the model.
3. By default, the Simulation Data Inspector automatically moves prior runs into the **Archive** and transfers the view to the current run. Drag `Run 1` from the **Archive** into the work area.
4. Click the **Layout** button and select the 2×1 plot layout.
5. Click the lower subplot in the viewing area. In the **Inspect** pane, select the check boxes for the `q, rad/sec`, `Stick`, and `alpha, rad` signals under `Run 1`.

   The check boxes in the **Inspect** pane indicate the signals plotted in the selected subplot, outlined in blue.



You can also move signals graphically, rather than using the check boxes. Click and drag the signal from the **Inspect** pane or another subplot to the subplot where you want to plot it.

For more information on working with plots, see “Create Plots Using the Simulation Data Inspector” on page 29-94.

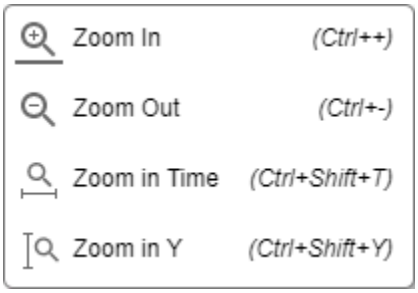
Zoom, Pan, and Resize Plots

You can closely inspect signals in a larger viewing area using the **Maximize** and **Full Screen** options on the toolbar above the viewing area.

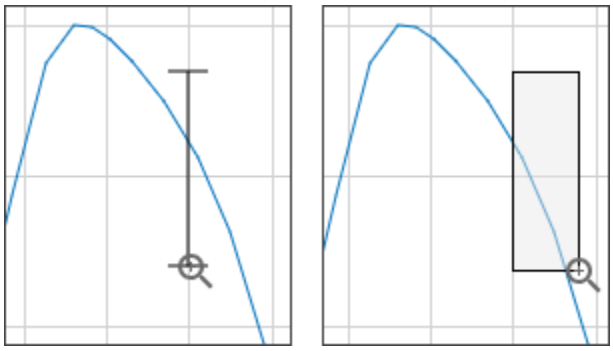
- — Select **Maximize** to expand the active subplot to occupy the entire graphical viewing area.
- — Select **Full Screen** to view your entire layout using your whole screen. The Simulation Data Inspector automatically collapses the navigation pane so the view of the layout is as large as possible.

You can zoom and pan within subplots that use a time plot visualization to inspect signal values and the relationships between signals. The zoom and pan controls in the Simulation Data Inspector are on the toolstrip above the graphical viewing area. Each icon allows you to control aspects of the plot using your mouse.

- You can select the zoom action for the zoom button from the drop-down.



  -  When you want to zoom in on a selected area in the plot, select **Zoom In**. While you have **Zoom In** selected, the Simulation Data Inspector adapts the zoom action it performs based on the area of the plot you select. When you select a very narrow area, the Simulation Data Inspector zooms on the *y*-axis only. When you select a very short area, the Simulation Data Inspector zooms on the *x*-axis only. The highlight around the selected region indicates the zoom behavior for that selection.



    You can click the graphical viewing area to zoom in a fixed amount on both axes. You can also click and drag to select an area to define the *y*- and *t*-axes. Scrolling with the mouse wheel zooms in and out on both axes.

  -  Select **Zoom Out** to zoom out a fixed amount when you click inside the plot area.
  -  **Zoom in Time** makes all the mouse actions zoom on the *t*-axis. You can click the graphical viewing area to zoom in a fixed amount. You can click and drag the graphical viewing area to select a portion of the plot as the limits for the *t*-axis. Scrolling with the mouse wheel zooms in and out on the *t*-axis.
  -  When you select **Zoom in Y**, all the mouse actions zoom on the *y*-axis. You can click in the graphical viewing area to zoom in a fixed amount. You can also click and drag to select a portion of the plot as the limits for the *y*-axis. Scrolling with the mouse wheel zooms in and out on the *y*-axis.

- With the mouse pointer selected, you can select signals by clicking them and pan by clicking anywhere on the plot and dragging the mouse.
- Click the fit-to-view option to scale the axes to accommodate your plotted data.

Linked Subplots

Subplots are linked by default. Linked plots have a synchronized response when you:

- Click a plot and drag to pan.
- Perform any zoom operation.
- Fit to view.
- Adjust **T-Axis** limits.

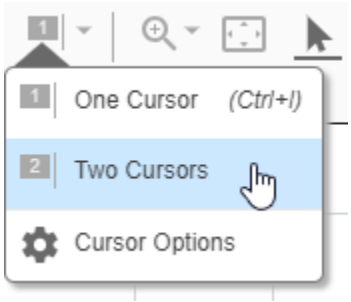
To inspect data independently in a subplot, you can unlink the subplot.

1. Select the subplot you want to unlink.
2. Click the **Visualization Settings** button in the upper right of the viewing area.
3. In the **Limits** section, clear the **Link Subplot** option.

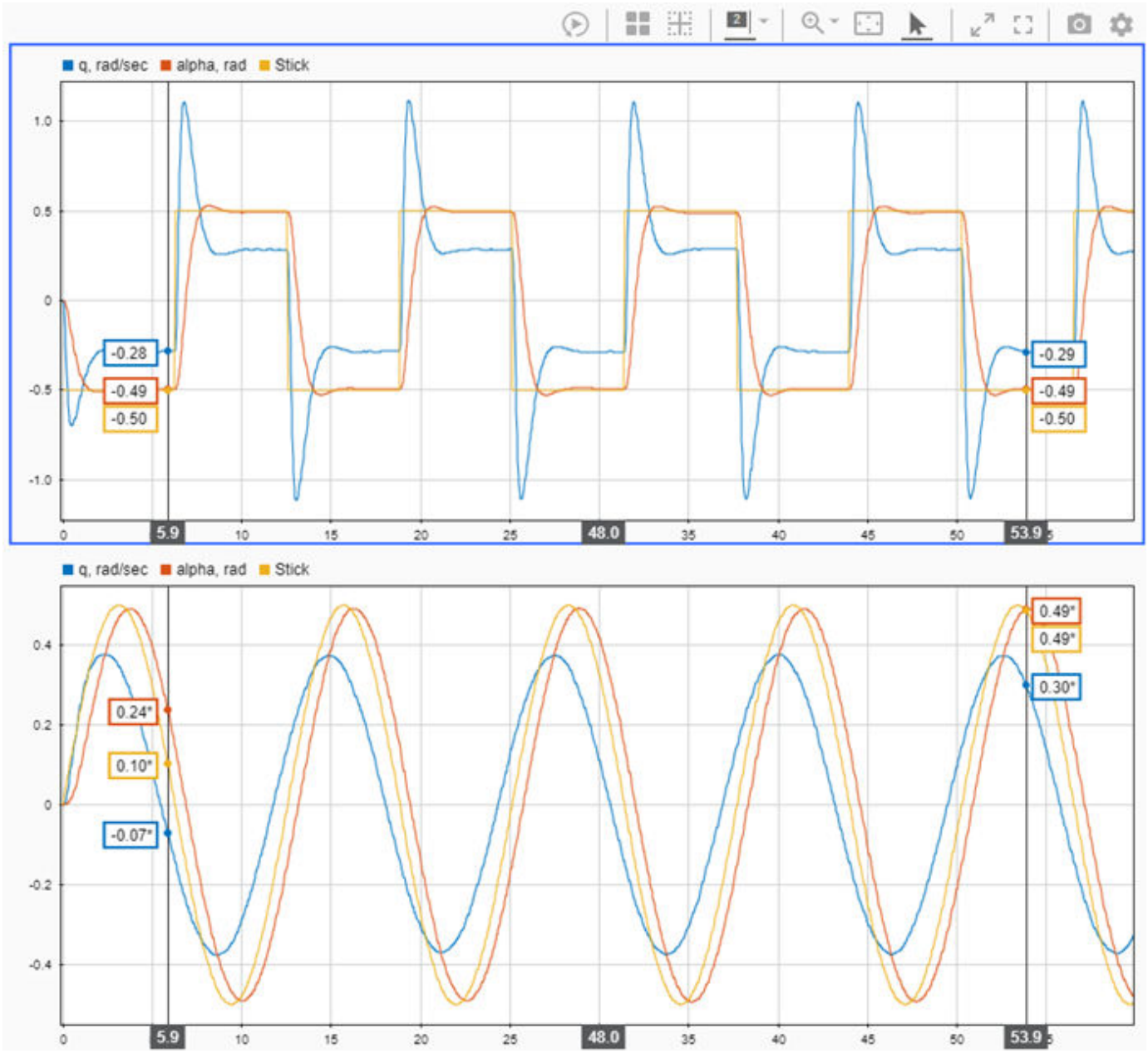
   The broken link symbol appears on the unlinked subplot.

Inspect Simulation Data Using Cursors

In the Simulation Data Inspector, you can inspect signals using data cursors. You can use one or two cursors to inspect your data. To add two cursors, select **Two Cursors** from the **Show/hide data cursors** drop-down.

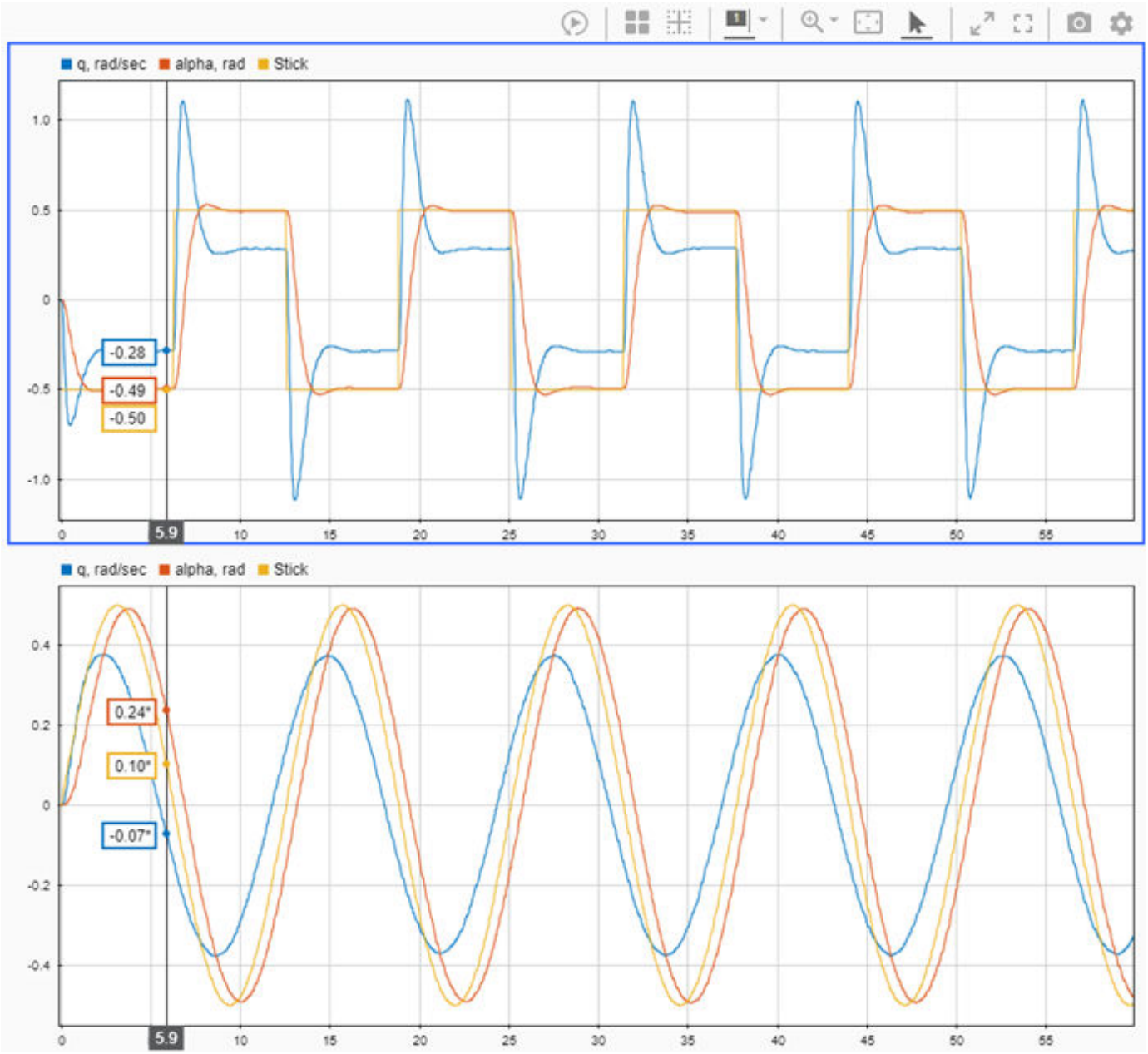


With two cursors, three time values are displayed: the time corresponding to each cursor position and the time spanned by the cursors. You can move the two cursors together by dragging the span label between the two cursors. You can also set the span by typing the desired value into the label field.



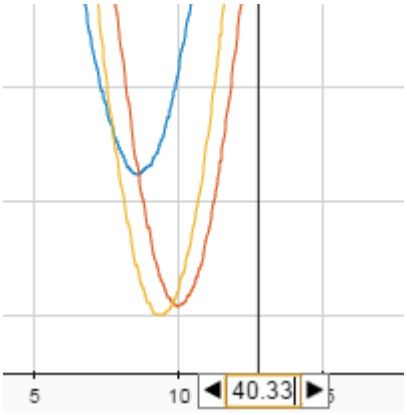
Practice inspecting data with cursors using one cursor.

1 Select **One Cursor** from the **Show/hide data cursors** drop-down.

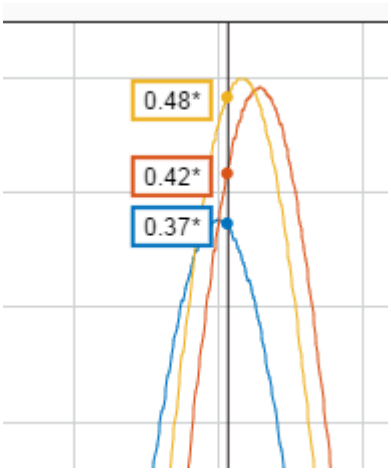


2 Drag the data cursor left or right to a point of interest. You can also use the arrow keys to move the data cursor from sample to sample.

To inspect the data at a specific point in time, click the cursor time field and enter the desired time value, for example `40.33`.



If the signal does not have a sample at the point of interest, the Simulation Data Inspector interpolates the value for the indicated time. An asterisk in the data cursor label indicates that the displayed value is interpolated. For information regarding interpolation methods, see “Interpolation” on page 29-141.

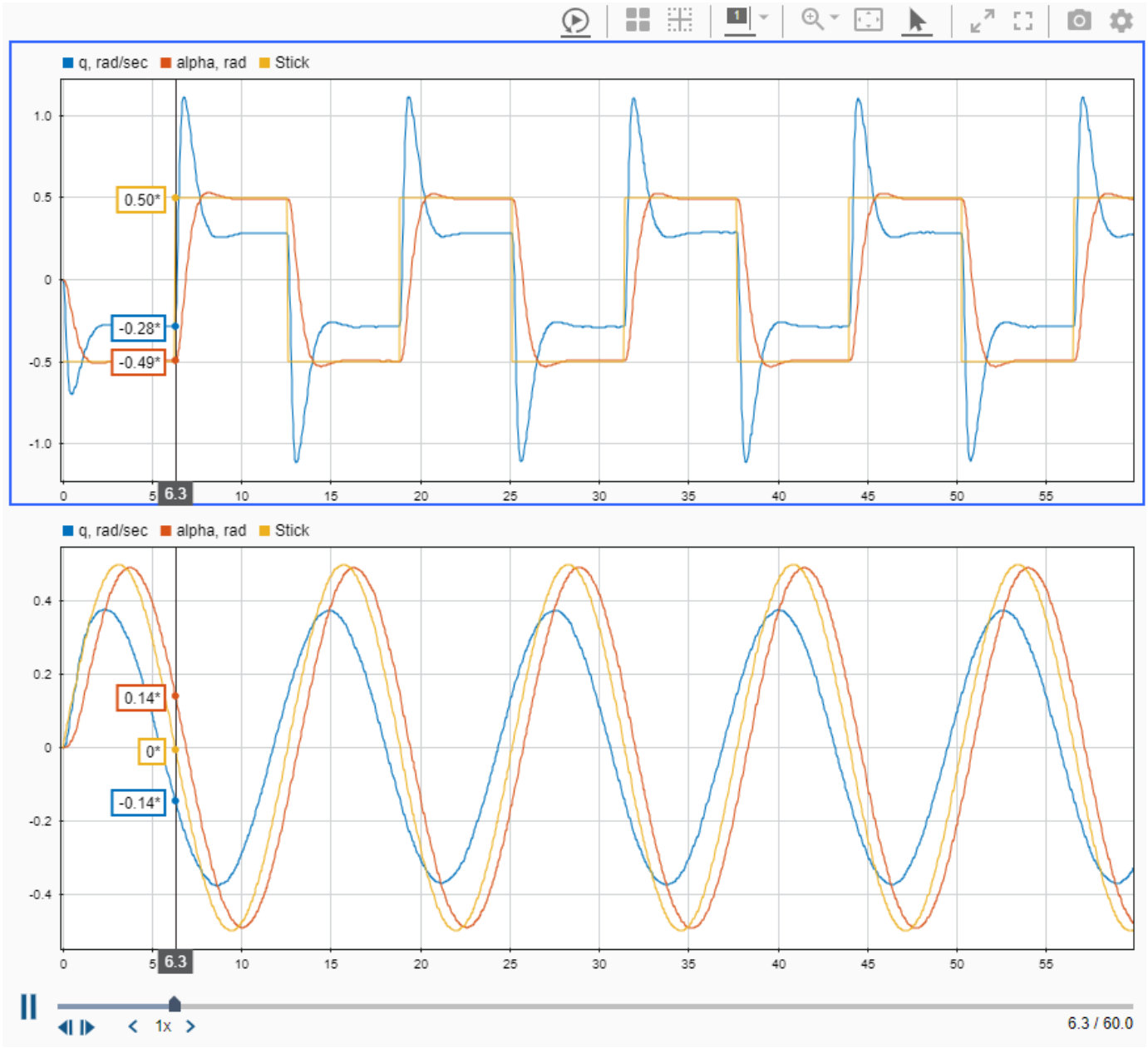


3 When you have finished inspecting the data, click the cursor button to remove the cursor from the viewing area.

Replay Data

You can replay data in the Simulation Data Inspector to inspect signal value changes and relationships between signals. Replaying data in the Simulation Data Inspector has no effect on any models or simulations. To replay data, first show the replay controls by clicking the **Show/hide replay controls** button. Then, press the **Replay** button.

The Simulation Data Inspector synchronously sweeps a cursor displaying the signal value across all subplots in the view. By default, data replays at a rate of one second per second, which means that replaying one second of data takes one second of clock time. You can adjust the replay speed using the arrows on either side of the label or by clicking the label to type the desired speed. You can also pause the replay and use the **Step forward** and **Step backward** buttons to inspect signal values, sample by sample. For a detailed example, see “Replay Data in the Simulation Data Inspector” on page 29-125.

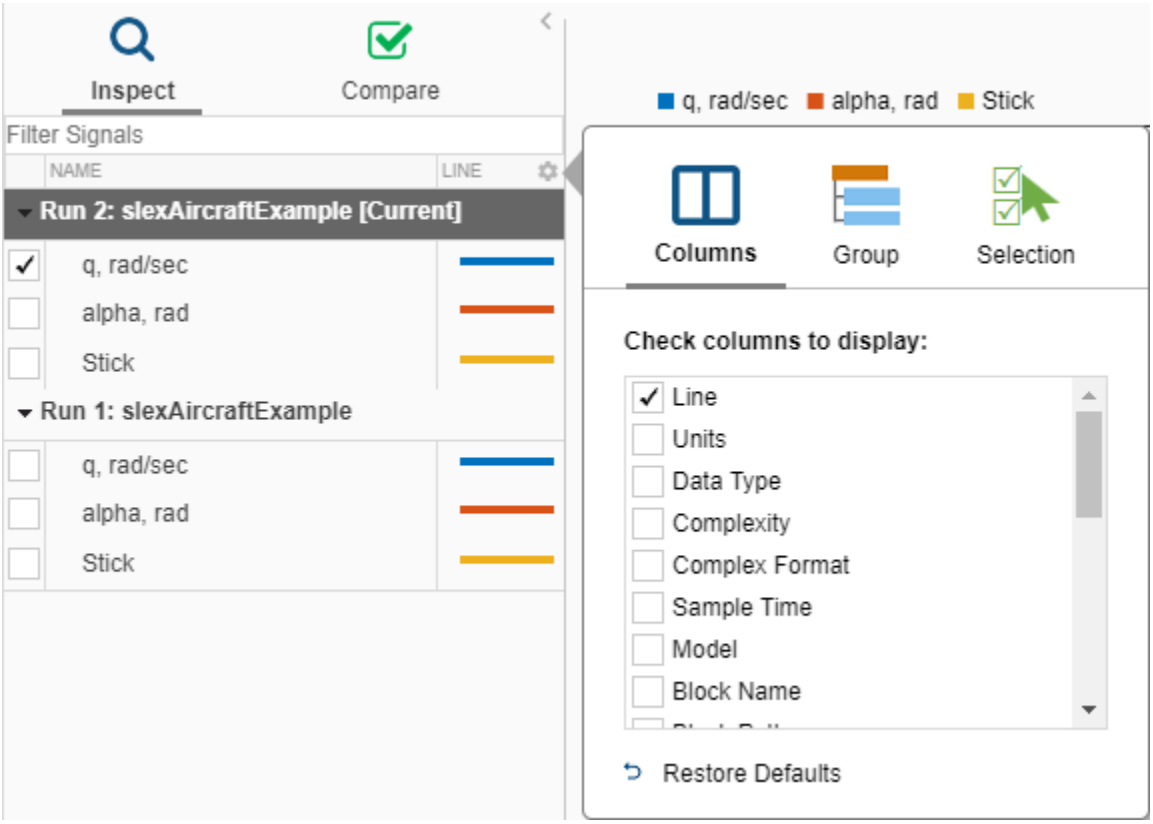


Inspect Metadata

The Simulation Data Inspector allows you to view run and signal metadata. You can view signal metadata in the **Properties** pane or in the table of signals under each run. You can view run data only in the **Properties** pane.

The **Properties** pane displays the metadata for the selected run or signal. You can edit properties using the white box in the **Values** column. When you view a comparison, the Simulation Data Inspector highlights metadata differences in red.

Columns in the navigation pane allow you to display signal properties in the table of signals under each run. To add or remove columns in the table, select the columns you want to display from the list on the **Columns** tab of the navigation pane's **Preferences** menu. Columns appear in the table in the order in which you select them.


Inspect
Compare
Filter Signals
NAME
LINE
Run 2: slexAircraftExample [Current]
q, rad/sec
alpha, rad
Stick
Run 1: slexAircraftExample
q, rad/sec
alpha, rad
Stick
q, rad/sec
alpha, rad
Stick
Columns
Group
Selection
Check columns to display:
Line
Units
Data Type
Complexity
Complex Format
Sample Time
Model
Block Name
Restore Defaults

Property Descriptions

Property Name	Value
Line	Signal line style and color
Units	Signal measurement units
Data Type	Signal data type
Complexity	Signal type — real or complex
Complex Format	Format for visualizing complex data
Sample Time	Type of sampling
Model	Name of the model that generated the signal
Block Name	Name of the signal's source block
Block Path	Path to the signal's source block
Port	Index of the signal on the block's output port
Dimensions	Dimensions of the matrix containing the signal
Channel	Index of signal within matrix
Run	Name of the simulation run containing the signal
Absolute Tolerance	User-specified, positive-valued absolute tolerance for the signal
Relative Tolerance	User-specified, positive-valued relative tolerance for the signal
Override Global Tolerance	User-specified property that determines whether signal tolerances take priority over global tolerances
Time Tolerance	User-specified, positive-valued time tolerance for the signal
Interp Method	User-specified interpolation method used to plot the signal
Sync Method	User-specified synchronization method used to coordinate signals for comparison
Time Series Root	Name of the variable associated with signals imported from the MATLAB workspace
Time Source	Name of the array containing the time data for signals imported from the MATLAB workspace
Data Source	Name of the array containing the data for signals imported from the MATLAB workspace

On the **Compare** pane, many parameters have a Baseline column and a Compare To column that you can display independently. If the **Baseline** and **Compare to** signals both have a property, but you can only display one property column, the column shows the **Baseline** property. In addition to the parameters listed for the **Inspect** pane, the **Compare** pane has columns specific to comparisons.

- **Max Difference** — The maximum difference between the **Baseline** and **Compare to** signals
- **Align By** — Primary signal alignment criterion specified in the **Alignment** tab of the Simulation Data Inspector **Preferences** menu

By default, the table displays the baseline name column and a column indicating whether the comparisons passed or failed.

See Also

Related Examples

- “Compare Simulation Data” on page 29-130
- “Create Plots Using the Simulation Data Inspector” on page 29-94
- “Save and Share Simulation Data Inspector Data and Views” on page 29-83

Modify Signal Properties in the Simulation Data Inspector

You can modify signal display units, data type, and names in the Simulation Data Inspector. When you modify the data type for a signal, the Simulation Data Inspector converts the signal values stored on disk. Changes you make to signal properties in the Simulation Data Inspector do not affect any models, files, or workspace variables.

You may want to save the original signal data or export modified signal data to use as simulation input. You can export data from the Simulation Data Inspector to the workspace or a file. For more information, see “Export Data from the Simulation Data Inspector” on page 29-86.

This example uses data from a simulation of the `slexAircraftExample` model. To generate the data for this example:

1. Enter `slexAircraftExample` in the MATLAB Command Window.
2. To log the `q, rad/sec`, the `Stick`, and the `alpha, rad` signals to the Simulation Data Inspector, select the signals in the model. Then, right-click the selection, and select **Log Selected Signals** from the context menu.
3. Double-click the Pilot block. Set **Wave form** to `sine`, and click **OK**.
4. Simulate the model.

Modify Signal Units

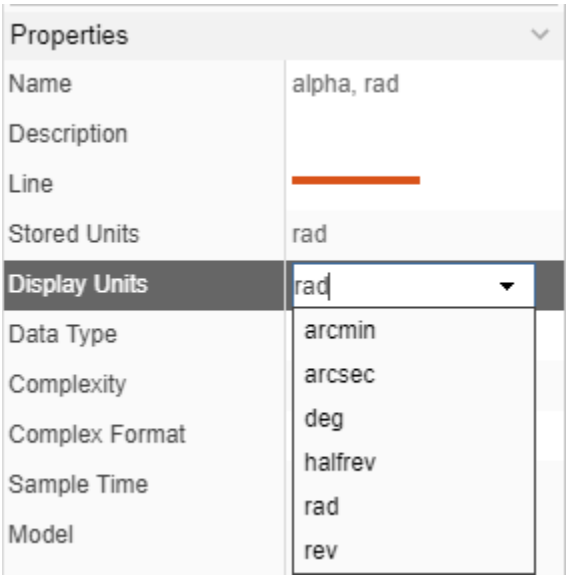
Signals in the Simulation Data Inspector have two unit properties: display units and stored units. To analyze sets of data using consistent units, modify the display units for a signal. The Simulation Data Inspector converts data when the stored units and display units for a signal differ such that signals are always plotted using values that correspond to the display units. The units you specify are properties of the signals in the Simulation Data Inspector only and do not affect the signal properties in models. To learn how to specify units in a model, see “Unit Specification in Simulink Models” on page 9-2.

The `slexAircraftExample` model does not specify units for its signals. However, some signal names indicate the intended units.

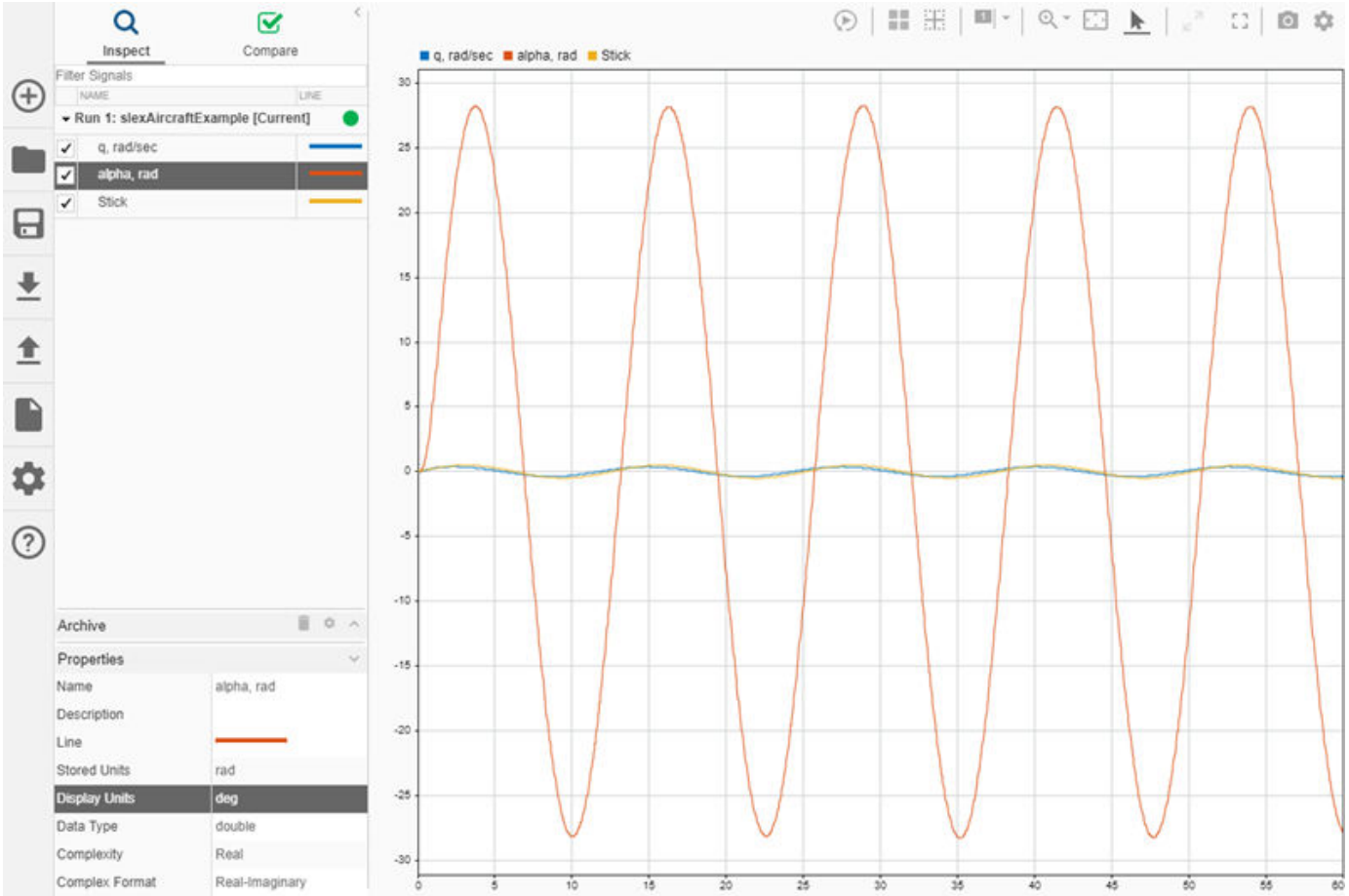
1. In the **Inspect** pane, select the `alpha, rad` signal.
2. To view the properties for the selected signal, expand the **Properties** pane.
3. To specify units of radians for the signal, in the **Properties** pane, type `rad` into the white text box next to the **Display Units** property.

For a list of units supported by Simulink, enter `showunitslist` into the MATLAB Command Window.

Now, the Simulation Data Inspector interprets the data for the `alpha, rad` signal as having units of `rad`. When you specify display units for a signal without units, the specified units also set the value for the stored units. Once a signal has units Simulink recognizes, you can convert the display units for the signal to supported compatible units using the Simulation Data Inspector. When you click the **Display Units** text box to modify the units, the Simulation Data Inspector provides a drop-down list of compatible units.



Change the `alpha, rad` signal display units to `deg` using the drop-down or by typing the new units into the **Display Units** field. When you change the units, the Simulation Data Inspector performs the conversion, and the plot updates to show the converted signal values.



You can also configure unit preferences in the Simulation Data Inspector to use units from a system of measurement or consistent units for a measurement type, such as length, for all logged and imported data. For more information, see “Signal Display Units” on page 29-62.

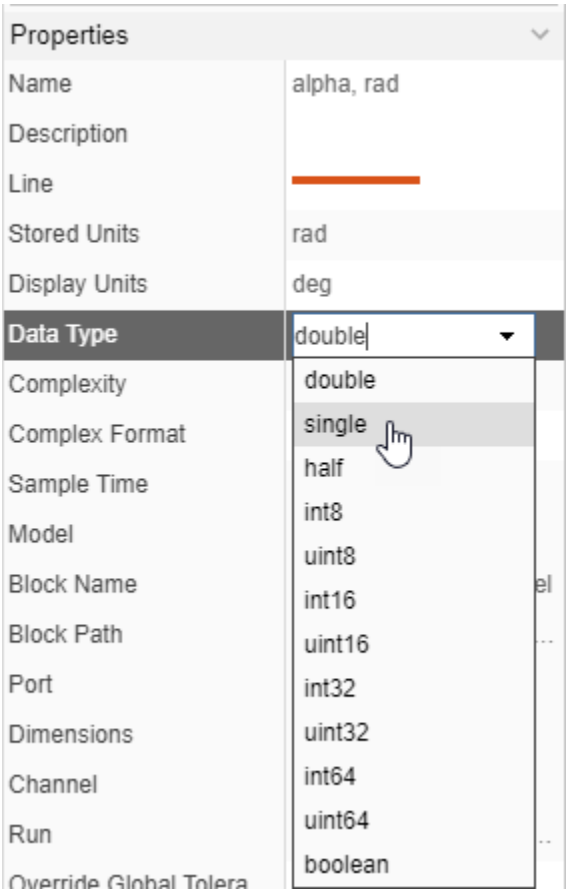
Note You can convert the stored units for a signal using the `convertUnits` function. Unit conversion does not support undo and may result in loss of precision.

Modify Signal Data Type

You can modify the data type for a signal to analyze the effect on signal values or to create a signal to use as simulation input. Converting the data type in the Simulation Data Inspector does not affect any signal properties in the model. You can convert signal data types to all built-in data types. If you have a license for Fixed-Point Designer, you can also convert to fixed-point data types. For a list of built-in data types, see “Data Types Supported by Simulink” on page 67-4.

Note When you convert to a lower precision data type, you lose precision in the data that cannot be recovered. You can save a copy of the data in the Simulation Data Inspector before changing signal data types. For more information, see “Save and Share Simulation Data Inspector Data and Views” on page 29-83.

The data type for the `alpha, rad` signal is `double`. Suppose you want to convert the data type to `single`. First, select the `alpha, rad` signal. Then, expand the **Properties** pane and click the white text field next to the **Data Type** property. You can type `single` or select it from the drop-down of data types. The Simulation Data Inspector converts the data when you click outside the drop-down.



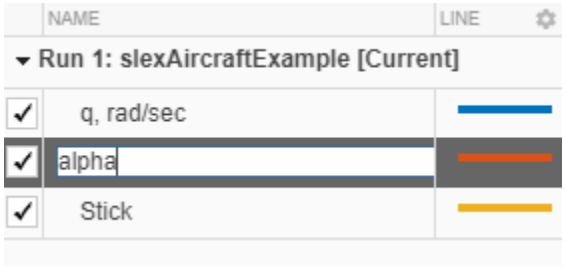
Because the `single` data type has less precision than the `double` data type, the Simulation Data Inspector returns a warning about irreversible precision loss. Click **Continue**. The plot shows no visible difference in the signal. If you saved the data for the original signal, you can compare the converted signal to analyze the effect of changing the data type. For more information on comparing data in the Simulation Data Inspector, see “Compare Simulation Data” on page 29-130.

Fixed-Point Conversions

If you have a license for Fixed-Point Designer, you can convert signal data types to fixed-point data types in the Simulation Data Inspector. Specify the data type in the **Data Type** property text box using the `fixdt` function.

Modify Signal Names

You can modify the names of signals in the Simulation Data Inspector. Changing the name of a signal in the Simulation Data Inspector does not affect signal names specified in the model. You can specify a new name from the work area, the **Archive**, or the **Properties** pane. To modify the signal name, click the signal name and type the new name. For example, change the name of the `alpha, rad` signal to `alpha` because the **Units** property now has the units information.



When you export data from the Simulation Data Inspector after modifying the name, the exported data uses the new name.

See Also

More About

- “View Data in the Simulation Data Inspector” on page 29-2
- “Inspect Simulation Data” on page 29-107
- “Compare Simulation Data” on page 29-130
- “Create Plots Using the Simulation Data Inspector” on page 29-94

Replay Data in the Simulation Data Inspector

This example shows how to replay data in the Simulation Data Inspector to inspect and build intuitive understanding of your data. When you replay data, a cursor sweeps synchronously across all the subplots in your view, displaying signal values. You can control the replay speed, pause to move the cursor sample-by-sample, and zoom to inspect your data. You can replay imported data and data logged from a simulation.

This example uses logged simulation data saved in a session. The session contains two runs of data logged from the `ex_vdp_mu` model. Data from `Run 1: ex_vdp_mu` is plotted in the viewing area.

Load the session and open the Simulation Data Inspector to view the data.

```
Simulink.sdi.load('ex_vdp_mu.mldatx');
Simulink.sdi.view
```

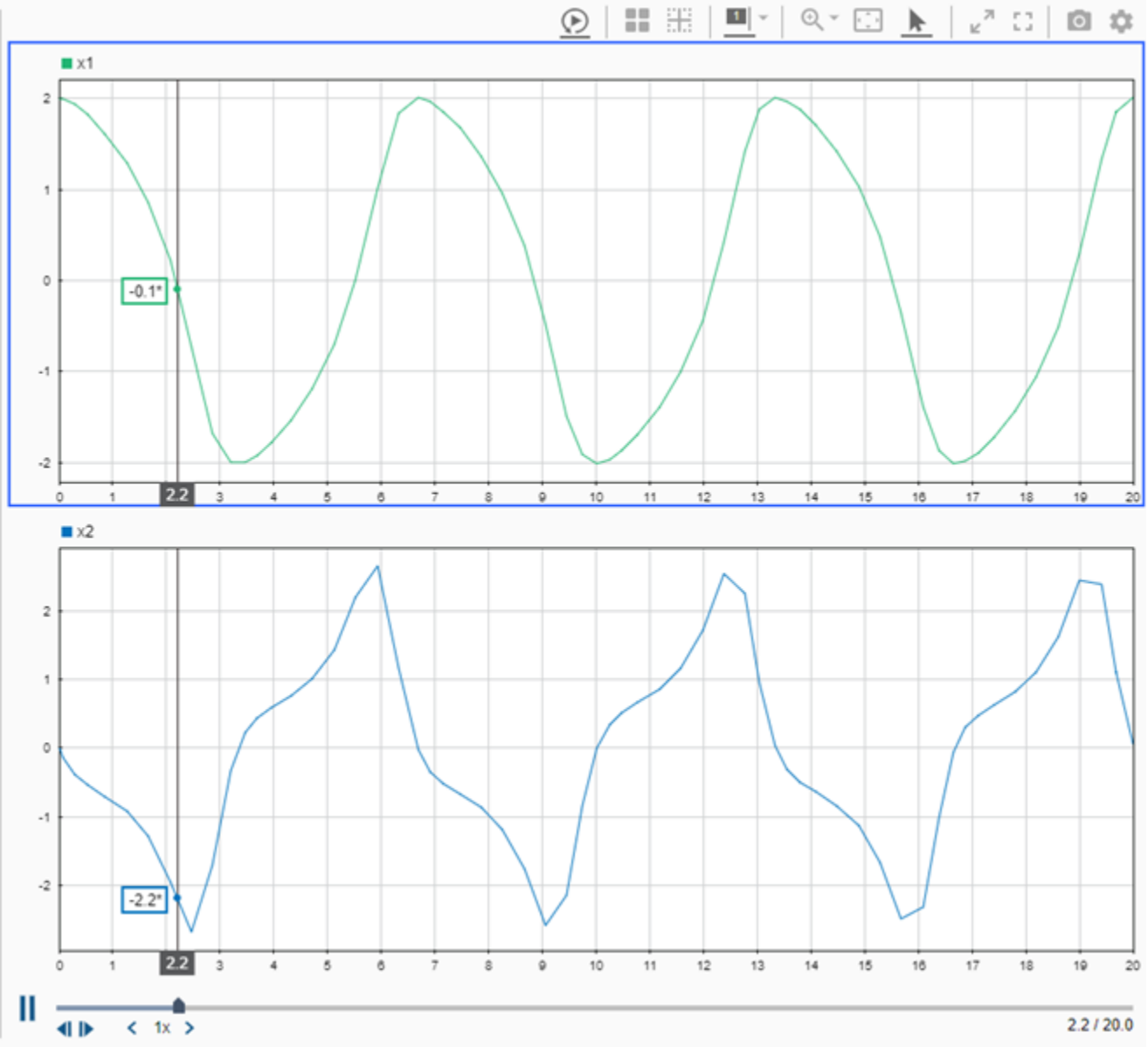
Replay Data

When you replay data, the Simulation Data Inspector animates synchronized cursors to move through a time period defined by the maximum time among the plotted signals. Replaying and pausing in the Simulation Data Inspector has no effect on any models or simulations.

To show the replay controls in the Simulation Data Inspector, click the **Show/hide replay controls** button. 

Click the **Replay** button. 

Synchronized cursors sweep across the subplots, indicating the signal value as the cursor moves in time. By default, the Simulation Data Inspector replays data at a rate of one second per second, meaning that the cursor moves through one second of data in the span of one second of clock time. You can adjust the replay speed to suit the data you want to inspect. To change the speed, click one of the arrows on either side of the label or click the label and enter your desired replay speed.

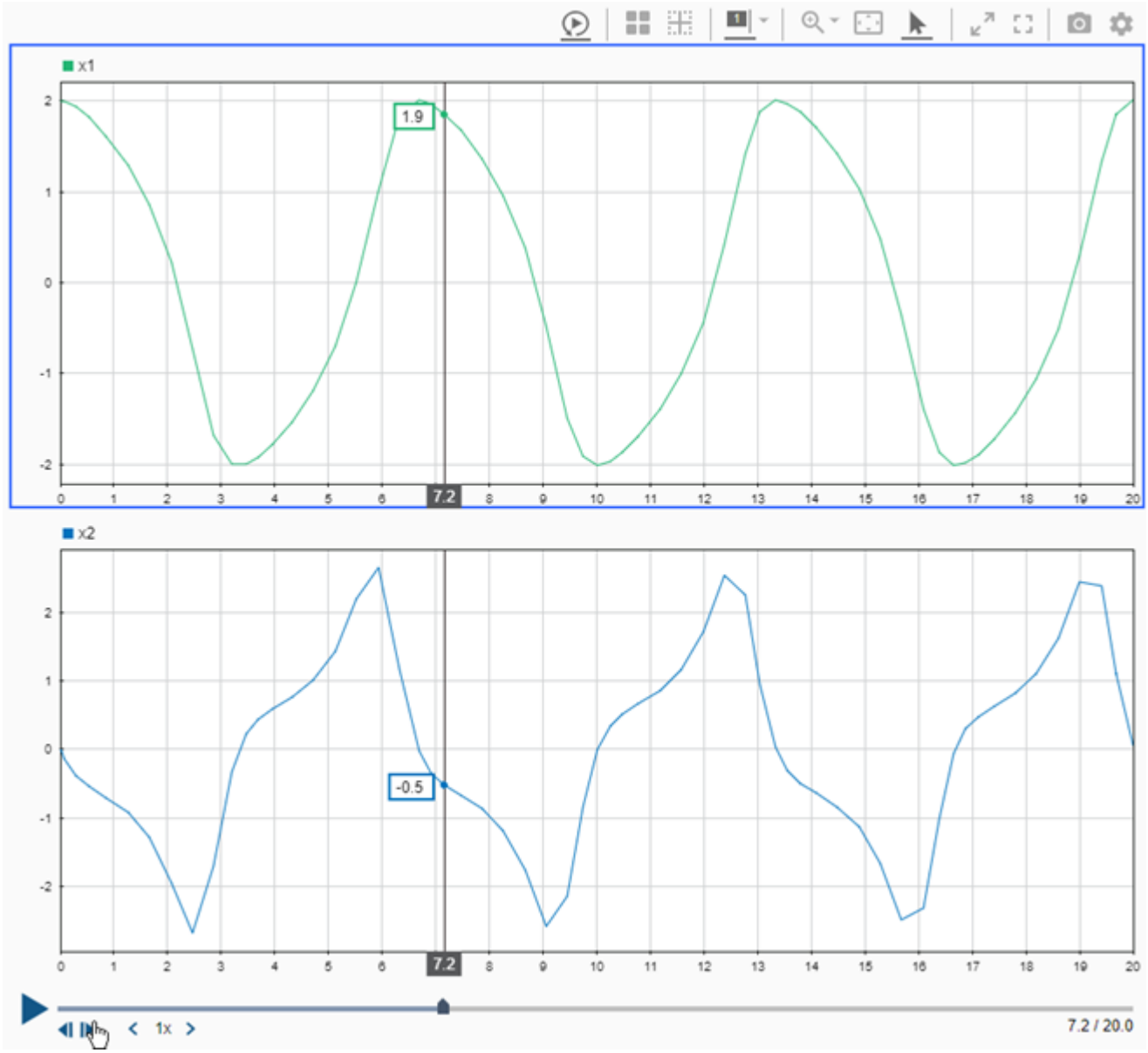


The Simulation Data Inspector moves the cursor based on the maximum time in the plotted signals and the replay speed. The position of the cursor at a given time during replay does not depend on the time values for samples in the plotted signals. When the cursor position does not correspond with a sample in a plotted signal, the Simulation Data Inspector interpolates a value to display, using the method defined in the **Interpolation** property of the signal. An asterisk at the end of the cursor value indicates that the value is interpolated.

Inspect Data Values During Replay

To inspect sample values in your data, you can pause the replay. When the replay is paused, you can use the **Step forward** and **Step backward** buttons to move the cursor sample-by-sample. The cursors move to the next time point among all plotted signals. When a plotted signal does not have a sample, the Simulation Data Inspector interpolates the value.

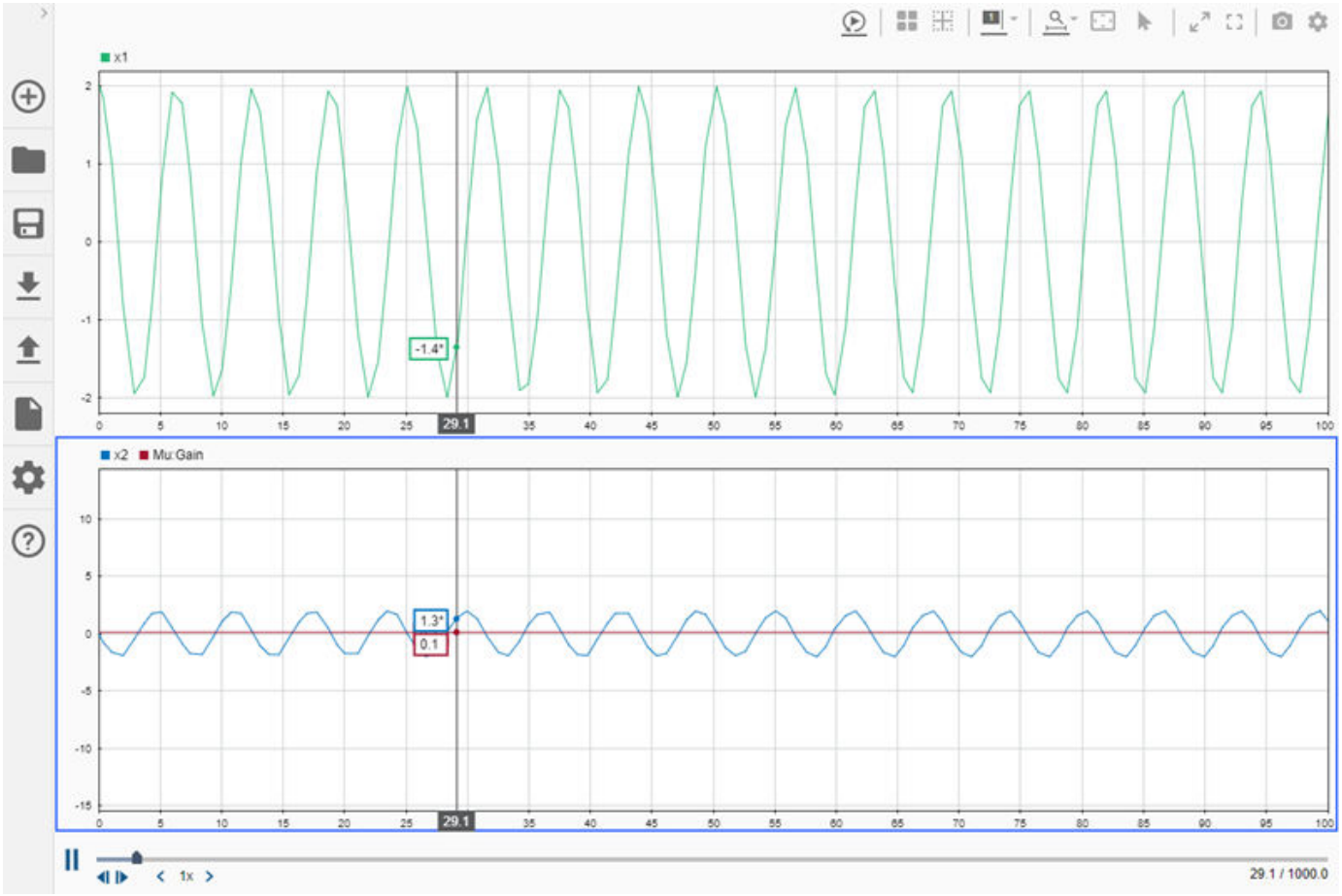
Pause the replay and use the **Step forward** and **Step backward** buttons to view the signal values.



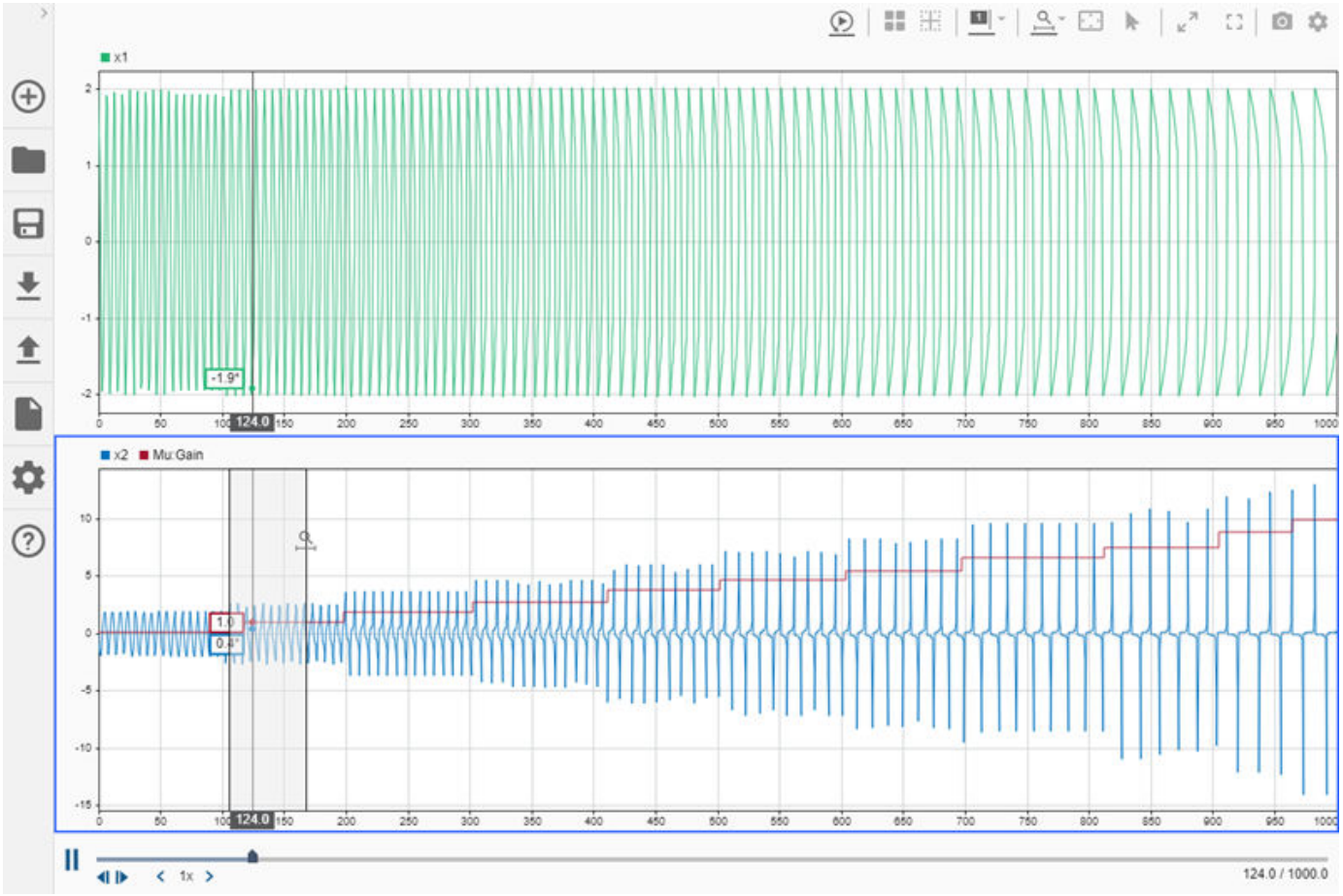
Zoom During Replay

You can zoom in on a region of your data during replay to inspect signal values and relationships more closely. Viewing a portion of your signal can be particularly useful when your data spans a long duration with a high sample rate. For example, plot the signals from `Run 2: ex_vdp_mu`.

The data in run 2 corresponds to a much longer simulation of the `ex_vdp_mu` model, where the value of `Mu` changed over the course of the simulation. Viewing the entire signal at once makes analyzing the changing characteristics of the curve difficult. Before replaying the data, change the **Time span** in the **Visualization settings** to 100 to show the first 100 seconds. You can also collapse the navigation pane to allow the viewing area to occupy more space. Then, replay the data.



When the cursor nears the end of the zoomed portion of the data, click the **Fit to view** button to show the complete signal. Then, you can use the **Zoom in time** option to select another region of the signal to inspect more closely during the replay.



When replaying a long simulation, you can also adjust the replay speed or drag the time marker to replay data in an area of interest.

See Also

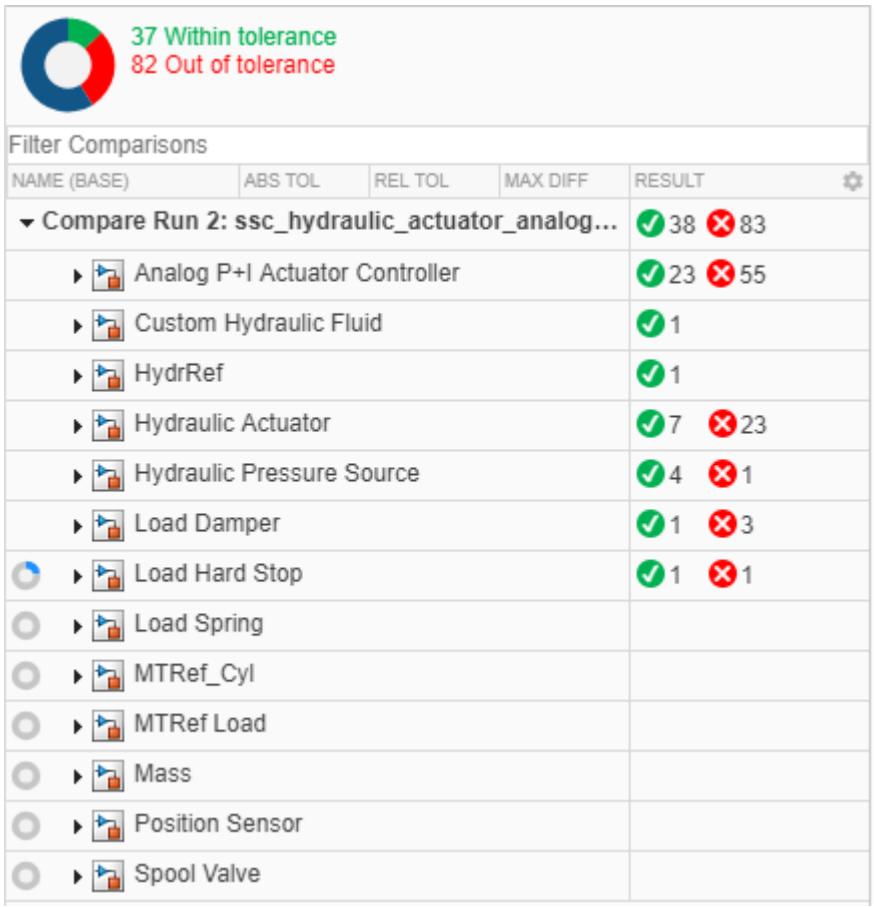
More About

- “View Data in the Simulation Data Inspector” on page 29-2
- “Inspect Simulation Data” on page 29-107
- “Create Plots Using the Simulation Data Inspector” on page 29-94

Compare Simulation Data

The Simulation Data Inspector can compare the data and metadata for runs and individual signals you import from the workspace or a file or log in a simulation. You can analyze comparison results using tolerances, and you can control aspects of the comparison through comparison settings. This example illustrates the basic steps to perform a comparison using the Simulation Data Inspector UI. For more information about tolerance calculations and the comparison settings, see “How the Simulation Data Inspector Compares Data” on page 29-139. For information about programmatic comparisons, see `Simulink.sdi.compareRuns`, `Simulink.sdi.compareSignals` and “Inspect and Compare Data Programmatically” on page 29-150.

The runs and signals used in this example are relatively small and simple. When you compare long signals or runs containing many signals, the Simulation Data Inspector displays incremental progress indicators during the comparison, and the **Compare** button becomes a **Cancel** button you can click to cancel the comparison at any point. For hierarchical data, the **Results** column in the comparison view summarizes the results on each hierarchical node.



This example uses the data generated in “Inspect Simulation Data” on page 29-107.

Setup

This example continues from “Inspect Simulation Data” on page 29-107. You can also use this script to generate the data required for the example.

```
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 3, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot', 'WaveForm', 'sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot', 'WaveForm', 'square')

% Simulate Model
sim('slexAircraftExample')
```

Compare Signals

You can compare signals to analyze the relationship between the inputs and outputs of a model. For example, compare the `Stick` input signal to the output signal, `alpha, rad`. Then specify tolerance values to analyze the result.

To compare the `alpha, rad` signal to the `Stick` signal:

1. Navigate to the **Compare** pane.
2. To view a list of signals available for comparison, expand the **Baseline** drop-down and select **Signals**.

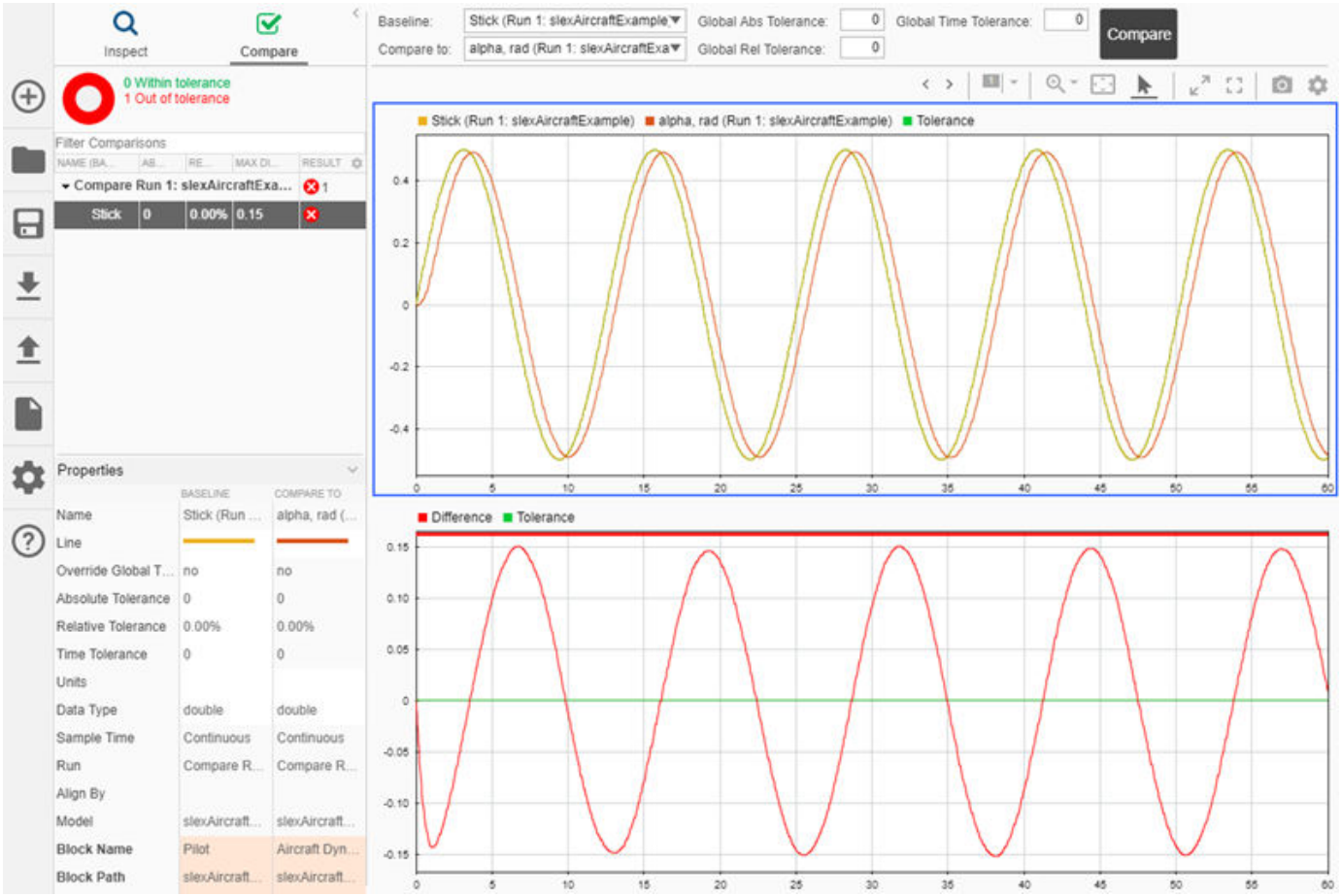
3. Select `Stick (Run 1: slexAircraftExample)`.

4 Select `alpha, rad (Run 1: slexAircraftExample)` from the **Compare to** drop-down.

5 Click **Compare**.

Alternatively, you can select the **Baseline** and **Compare to** signals through the context menu by right-clicking the signal in the **Inspect** pane.

The signals are not identical, so they do not match within the absolute, relative, and time tolerances, all set to 0 by default.



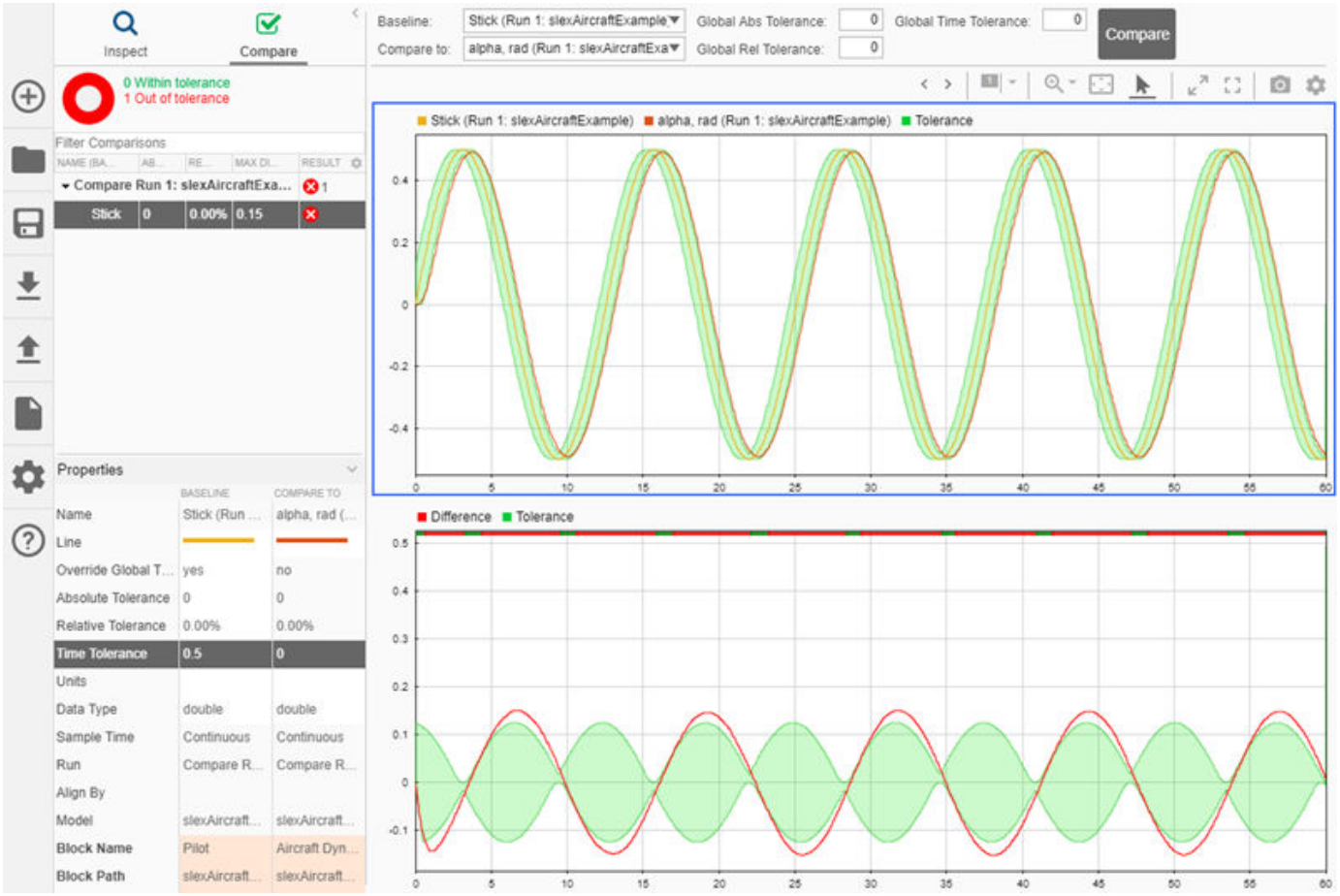
Signal Time Tolerance

Looking at the top plot in the comparison view, you can see the `alpha, rad` signal lags the `Stick` signal. For signal comparisons, the Simulation Data Inspector uses tolerance values specified for the **Baseline** signal. Add a time tolerance to the `Stick` signal to account for the lag.

Select the `Stick` signal and enter `0.5` in the **Time Tolerance** field of the **Properties** pane. When you specify a tolerance for a **Baseline** signal, its **Override Global Tolerance** field automatically changes to `yes`. When you click away from the **Time Tolerance** field, the comparison runs again, using the signal-level tolerance you specified. If you want to use global tolerance values for the signal, click the **Override Global Tolerance** field and select `no` from the drop-down.

The Simulation Data Inspector draws the tolerance band around the plotted **Baseline** signal and around the signed difference signal displayed in the bottom subplot. The bar along the top of the

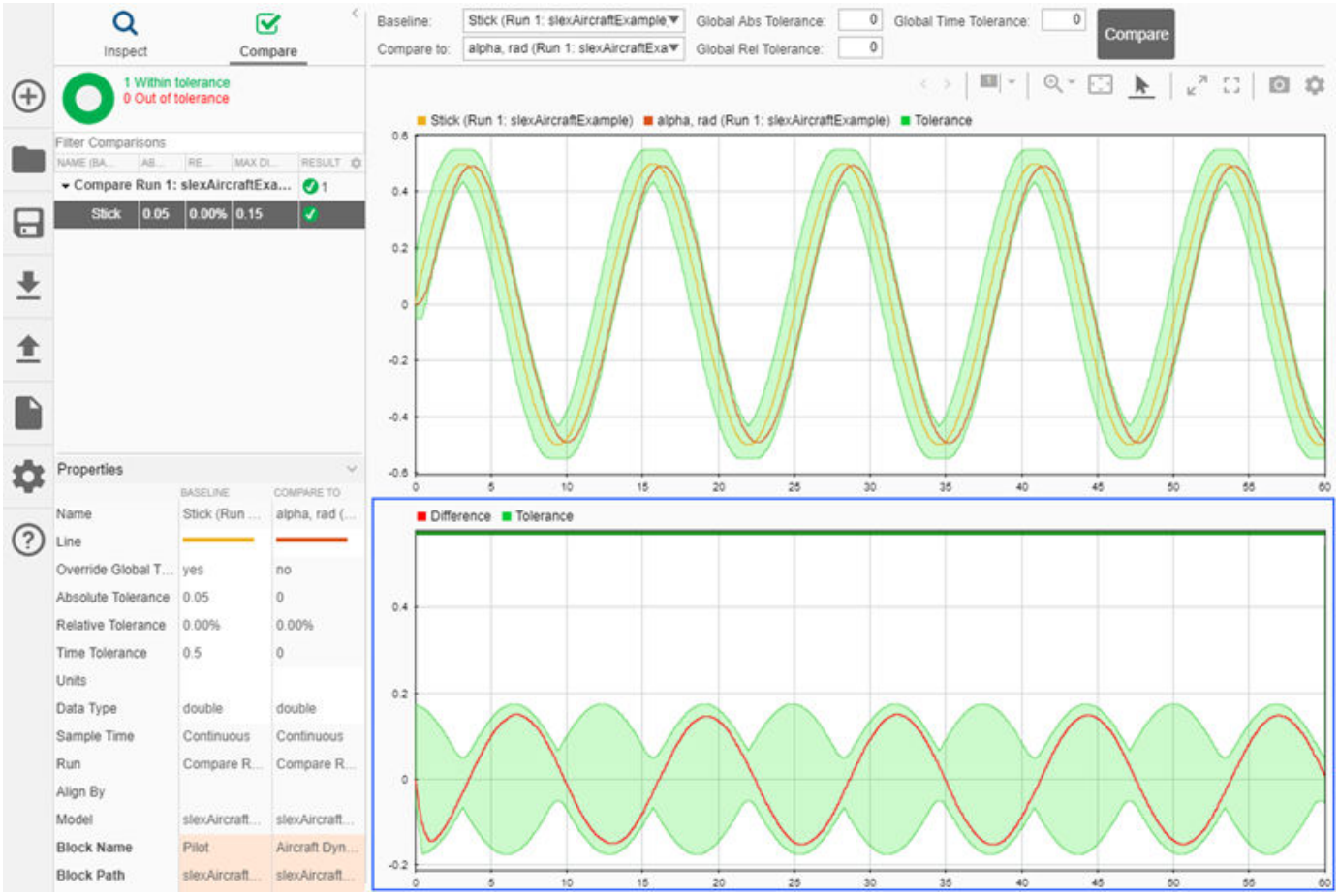
difference plot shows regions within tolerance and out of tolerance for the comparison in green and red.



Signal Magnitude Tolerance

The time tolerance covers the phase difference between the two signals, but the comparison still includes regions out of tolerance due to the amplitude difference. You can add a magnitude tolerance as either an **Absolute Tolerance** or **Relative Tolerance**.

To add an absolute tolerance to the `Stick` signal, enter `0.05` into the **Absolute Tolerance** field in the properties pane. With the combination of the absolute and time tolerances, the signal comparison passes.



Note The Simulation Data Inspector draws the tolerance region with the most lenient interpretation of the specified tolerances for each point. For more information on how the Simulation Data Inspector calculates the tolerance band, see “Tolerance Specification” on page 29-142.

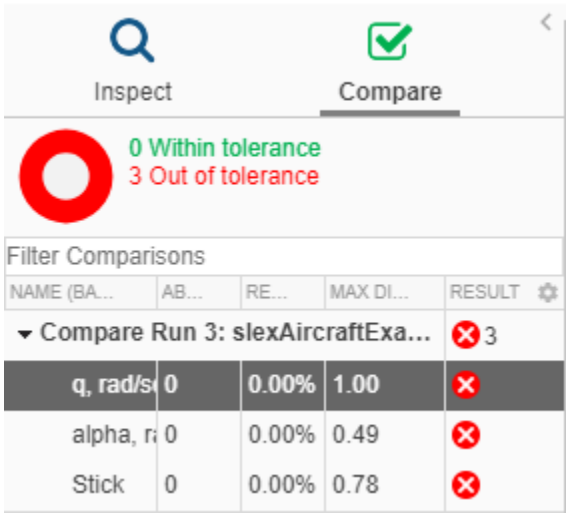
Compare Runs

You can also use the Simulation Data Inspector to compare all the signals in a run to all the signals in another run. Run comparisons can provide useful information about the effects of changing model parameters. For example, change the frequency cutoff of the filter for the control stick input signal. Then, evaluate the effect on the logged signals using the Simulation Data Inspector.

1. Click the **Model Explorer** button to access the **Model Workspace** variables.
2. Change the value of `Ts` in the **Model Workspace** from `0.1` to `1` and close the **Model Explorer**.
3. Simulate the model with the new filter.
4. In the **Compare** pane in the Simulation Data Inspector, expand the **Baseline** drop-down, and select **Runs**.
5. From the list of runs, select `Run 2: slexAircraftExample`.
6. Expand the **Compare to** drop-down and select `Run 3: slexAircraftExample`.

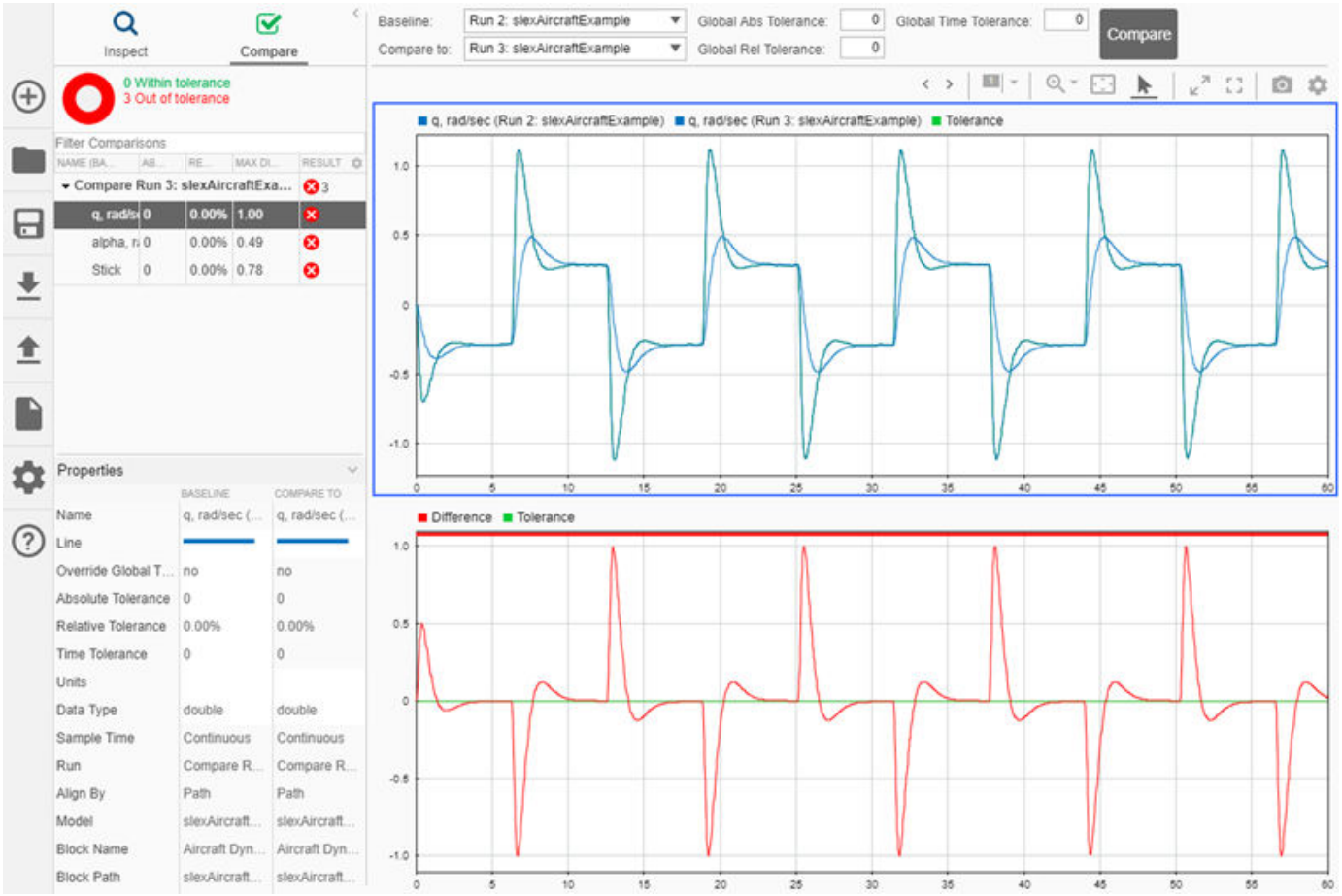
7 Click **Compare**.

The **Compare** pane lists all signals from the compared runs and summarizes the results of the comparison in the **Results** column. In this example, all three signals aligned, and none matched within the specified tolerance values, all of which are set to zero.



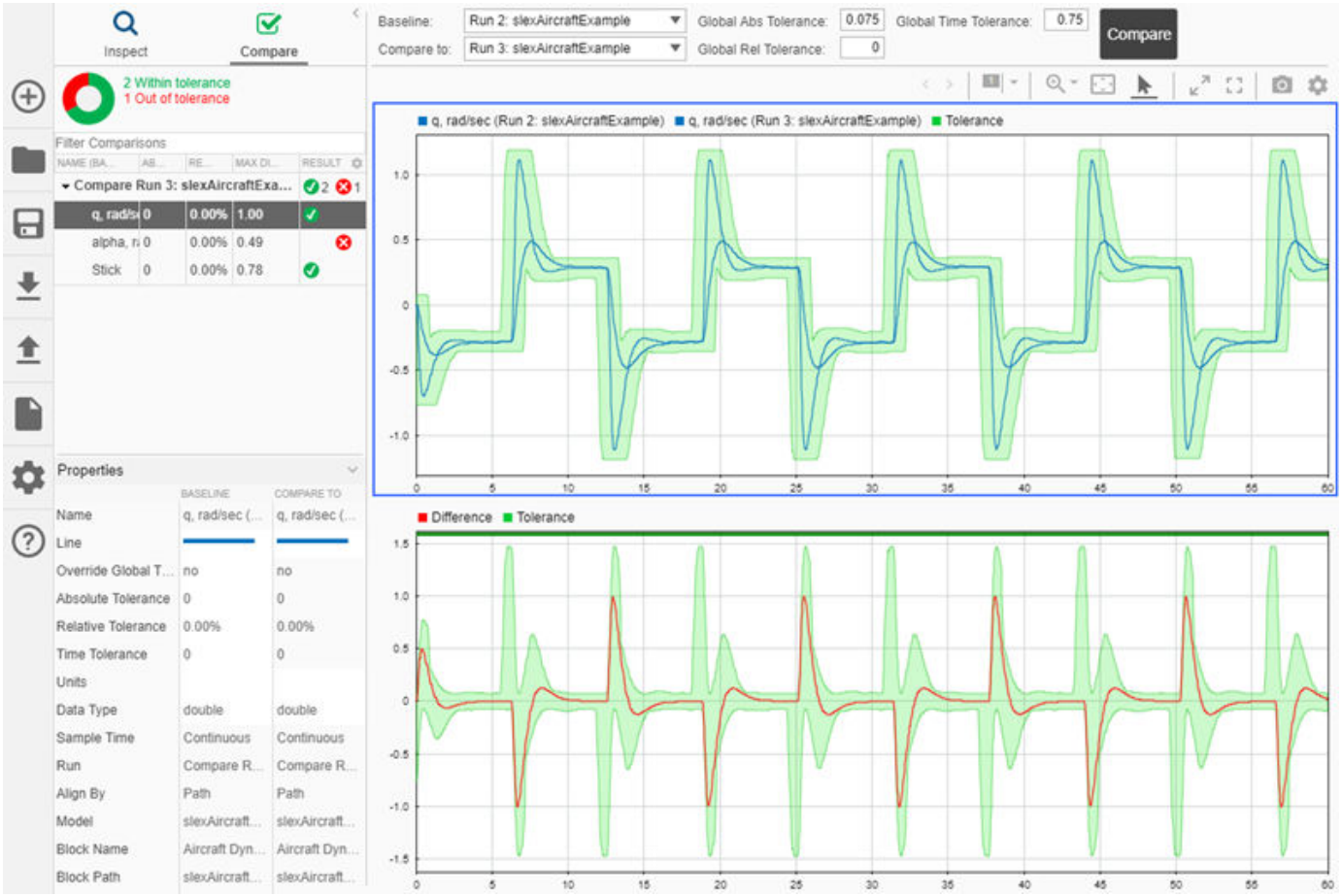
Note The Simulation Data Inspector only compares signals from the **Baseline** run that align with a signal from the **Compare To** run. If a signal from the **Baseline** run does not align with a signal from the **Compare To** run, the signal is listed in the **Compare** pane with a warning. . For more information on signal alignment, see “Signal Alignment” on page 29-139.

To plot comparison data, select the signal you want to view in the **Compare** pane. Here, the top plot shows the `q, rad/sec` signals from the **Baseline** and **Compare To** runs. The bottom plot shows the difference between the signals and a graphical representation of the tolerance.

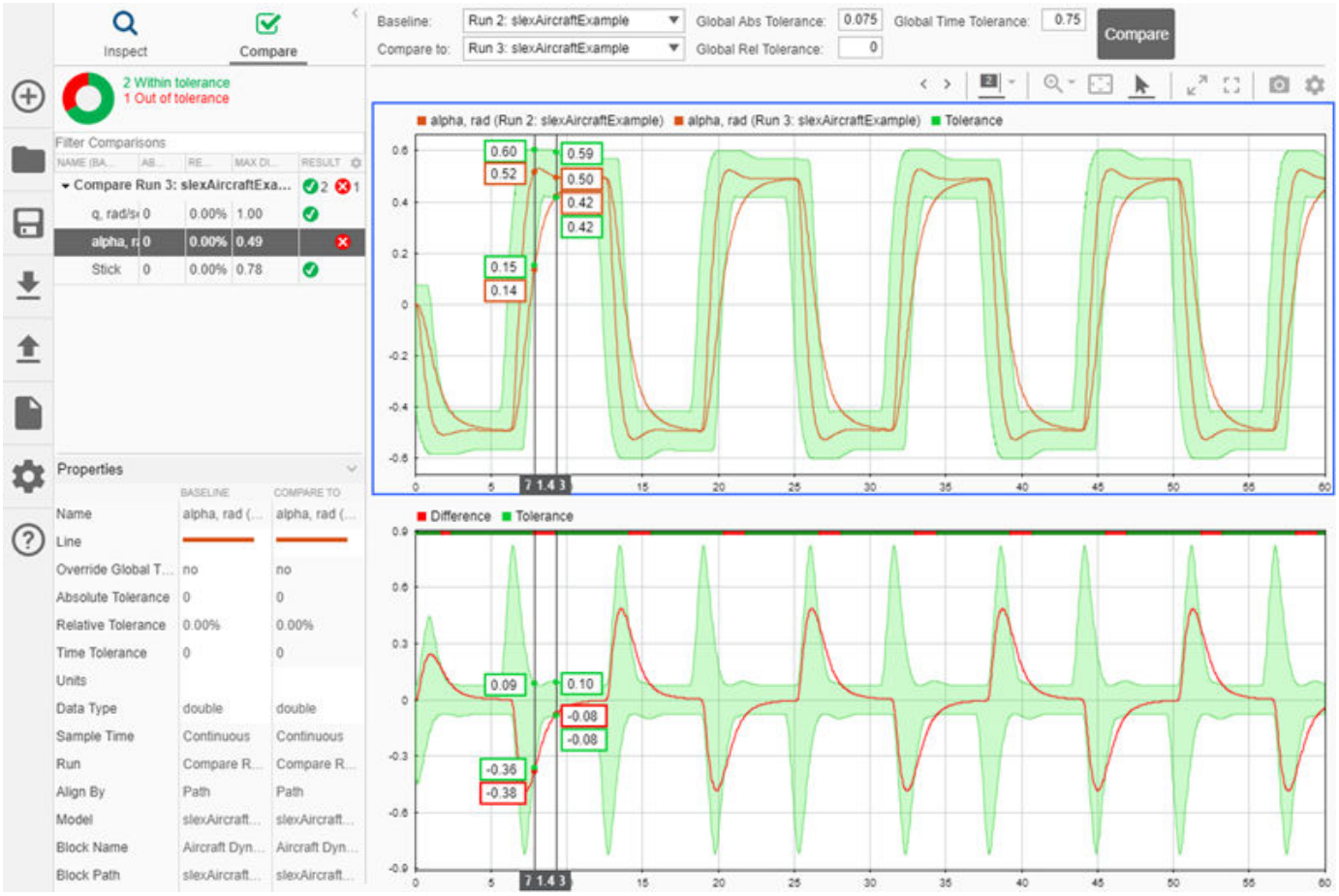


Global Tolerances

To qualify signals in the run comparison, you can add global tolerances to the comparison. Change the **Global Time Tolerance** to `0.75` and the **Global Abs Tolerance** to `0.075`, and click the **Compare** button to run the comparison. The Simulation Data Inspector draws the tolerance band around the **Baseline** signal and on the signed difference plot on the lower half of the graphical viewing area. With the new tolerance values, the `Stick` and `q, rad/sec` signals pass the comparison.



View the `alpha, rad` signal to analyze the comparison's out of tolerance regions. Click the arrow buttons in the tool strip to navigate through the out of tolerance regions. Two cursors on the plot show the beginning and end of the first out of tolerance region. You can use your keyboard arrows to explore the signal and tolerance values throughout each out of tolerance region. To view the next out of tolerance region, click the right arrow button in the tool strip.



To resolve the out of tolerance regions, you can choose to modify the global tolerance values or to add a signal specific tolerance to the `alpha, rad` signal using the signal properties.

See Also

Related Examples

- “How the Simulation Data Inspector Compares Data” on page 29-139
- “Inspect Simulation Data” on page 29-107
- “Share Simulation Data Inspector Views” on page 29-84
- “Create Plots Using the Simulation Data Inspector” on page 29-94

How the Simulation Data Inspector Compares Data

You can tailor the Simulation Data Inspector comparison process to fit your requirements in multiple ways. When comparing runs, the Simulation Data Inspector:

1. Aligns signal pairs in the **Baseline** and **Compare To** runs based on the **Alignment** settings.

   The Simulation Data Inspector does not compare signals that it cannot align.
2. Synchronizes aligned signal pairs according to the specified **Sync Method**.

   Values for time points added in synchronization are interpolated according to the specified **Interpolation Method**.
3. Computes the difference of the signal pairs.
4. Compares the difference result against specified tolerances.

When the comparison run completes, the results of the comparison are displayed in the navigation pane.

Status	Comparison Result
	Difference falls within the specified tolerance.
	Difference violates specified tolerance.
	The signal does not align with a signal from the **Compare To** run.

When you compare signals with differing time intervals, the Simulation Data Inspector compares the signals on their overlapping interval.

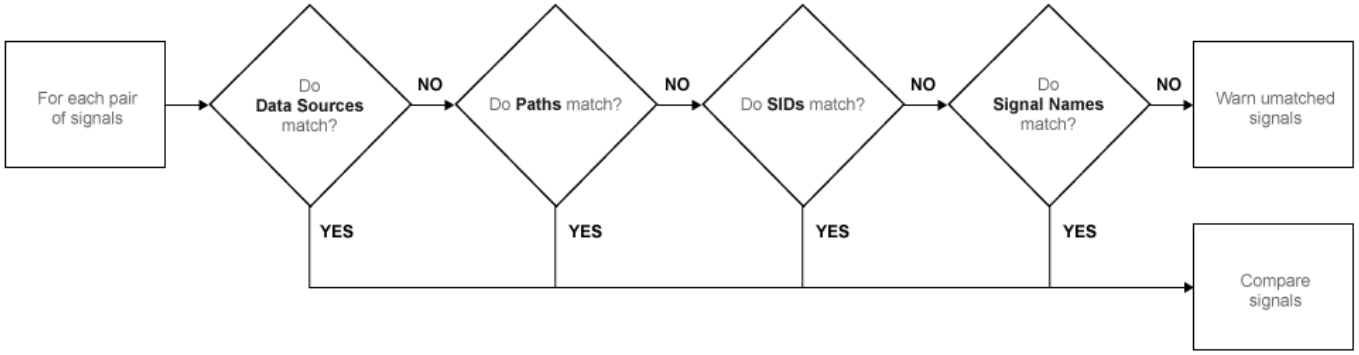
Signal Alignment

In the alignment step, the Simulation Data Inspector decides which signal from the **Compare To** run pairs with a given signal in the **Baseline** run. When you compare signals with the Simulation Data Inspector, you complete the alignment step by selecting the **Baseline** and **Compare To** signals.

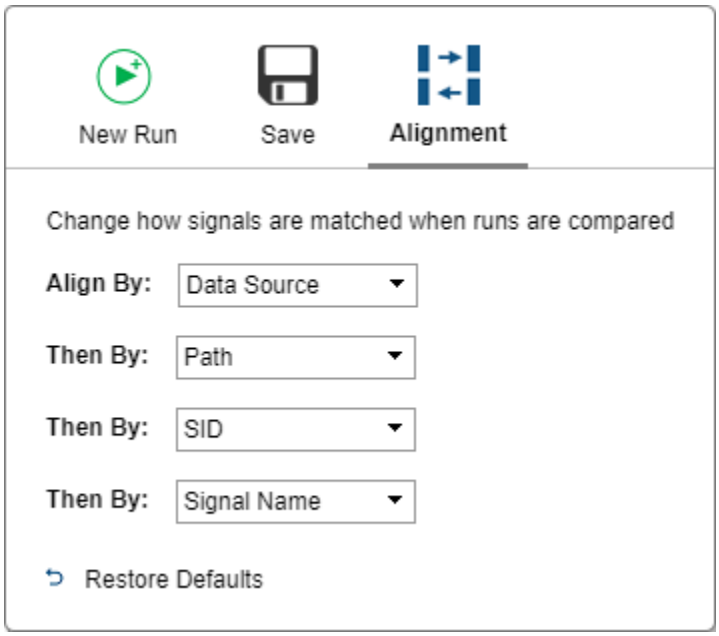
The Simulation Data Inspector aligns signals using a combination of their Data Source, Path, SID, and Signal Name properties.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Simulink identifier For more information about SIDs, see “Simulink Identifiers” on page 1-7
Signal Name	Name of the signal in the model

With the default alignment settings, the Simulation Data Inspector aligns signals between runs according to this flow chart.



You can specify the priority for each of the signal properties used for alignment in the Simulation Data Inspector **Preferences**. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of the **Then By** fields blank.

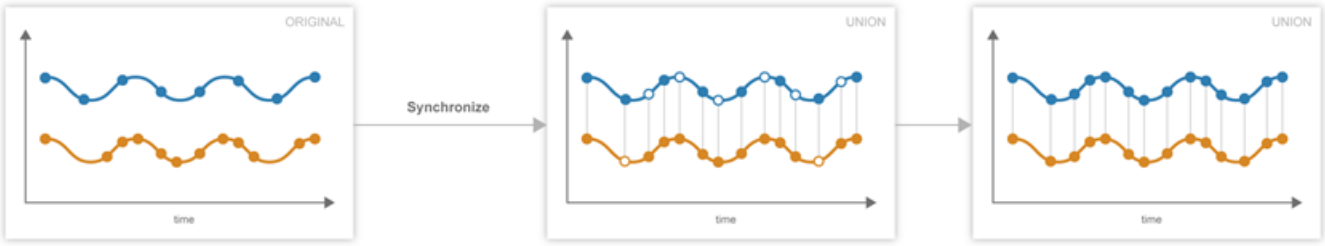


Synchronization

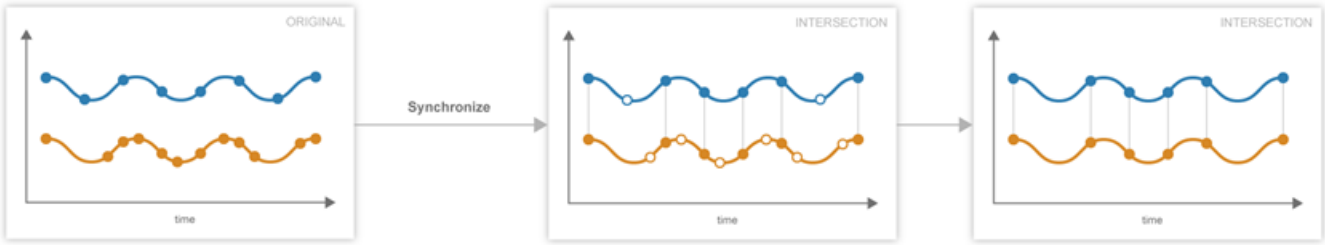
Often, signals that you want to compare don't contain the exact same set of time points. The synchronization step in Simulation Data Inspector comparisons resolves discrepancies in signals' time vectors. You can choose `union` or `intersection` as the synchronization method.

When you specify `union` synchronization, the Simulation Data Inspector builds a time vector that includes every sample time between the two signals. For each sample time not originally present in either signal, the Simulation Data Inspector interpolates the value. The second graph in the illustration shows the union synchronization process, where the Simulation Data Inspector identifies samples to add in each signal, represented by the unfilled circles. The final plot shows the signals after the Simulation Data Inspector has interpolated values for the added time points. The Simulation

Data Inspector computes the difference using the signals in the final graph, so that the computed difference signal contains all the data points between the signals.



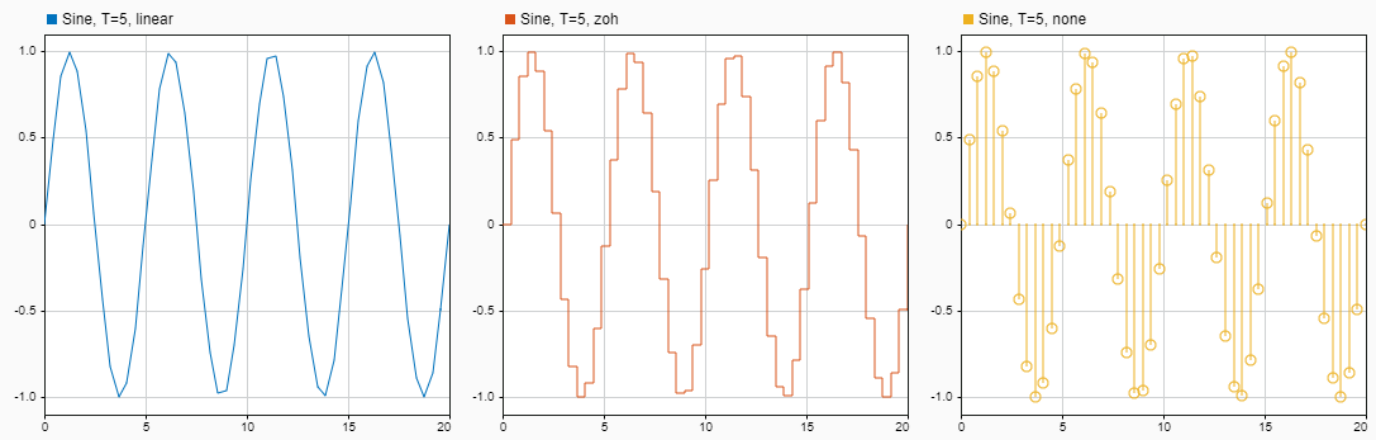
When you specify `intersection` synchronization, the Simulation Data Inspector uses only the sample times present in both signals in the comparison. In the second graph, the Simulation Data Inspector identifies samples that do not have a corresponding sample for comparison, shown as unfilled circles. The final graph shows the signals used for the comparison, without the samples identified in the second graph.



The choice between the synchronization options involves a trade off between speed and accuracy. The interpolation required by `union` synchronization takes time, but provides a more precise result. When you use `intersection` synchronization, the comparison finishes quickly because the Simulation Data Inspector computes the difference for fewer data points and does not interpolate. However, some data is discarded and precision lost with `intersection` synchronization.

Interpolation

The interpolation property of a signal determines how the Simulation Data Inspector displays the signal and how additional data values are computed in synchronization. You can choose to interpolate your data with a zero-order hold (zoh) or a linear approximation. You can also specify no interpolation.



When you specify `zoh` or `none` for the **Interpolation Method**, the Simulation Data Inspector replicates the data of the previous sample for interpolated sample times. When you specify `linear` interpolation, the Simulation Data Inspector uses samples on either side of the interpolated point to linearly approximate the interpolated value. Typically, discrete signals use `zoh` interpolation and continuous signals use `linear` interpolation. You can specify the **Interpolation Method** for your signals in the signal properties.

Tolerance Specification

The Simulation Data Inspector allows you to specify the scope and value of the tolerance for your signal. You can define a tolerance band using any combination of absolute, relative, and time tolerance values, and you can specify whether the specified tolerance applies to an individual signal or to all the signals in a run.

Tolerance Scope

In the Simulation Data Inspector, you can specify the tolerance for your data globally or for an individual signal. Global tolerance values apply to all signals in a run that do not have **Override Global Tol** set to `yes`. You can specify global tolerance values for your data at the top of the graphical viewing area in the **Compare** view. To specify signal specific tolerance values, edit the signal properties and ensure the **Override Global Tol** property is set to `yes`.

Tolerance Computation

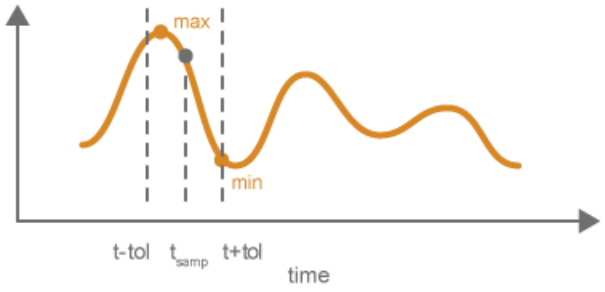
In the Simulation Data Inspector, you can specify a tolerance band for your run or signal using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your run or signal using multiple types of tolerances, each tolerance can yield a different answer for the tolerance at each point. The Simulation Data Inspector computes the overall tolerance band by selecting the most lenient tolerance result for each data point.

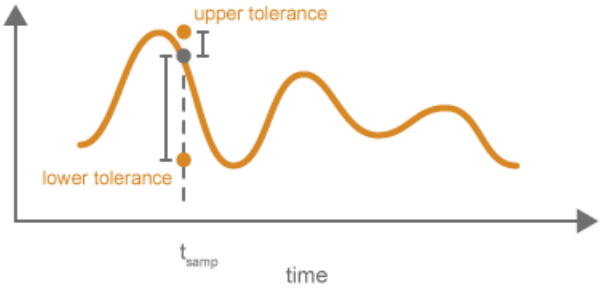
When you define your tolerance using only the absolute and relative tolerance properties, the Simulation Data Inspector computes the tolerance for each point as a simple maximum.

```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

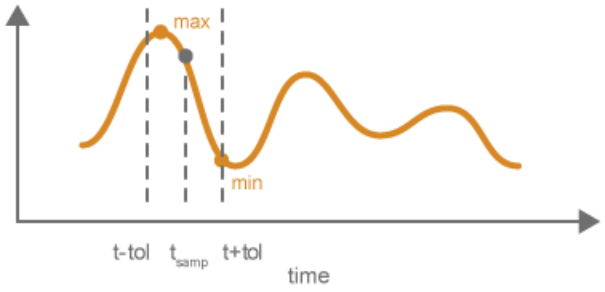
The upper boundary of the tolerance band is formed by adding `tolerance` to the **Baseline** signal. Similarly, the Simulation Data Inspector computes the lower boundary of the tolerance band by subtracting `tolerance` from the **Baseline** signal.

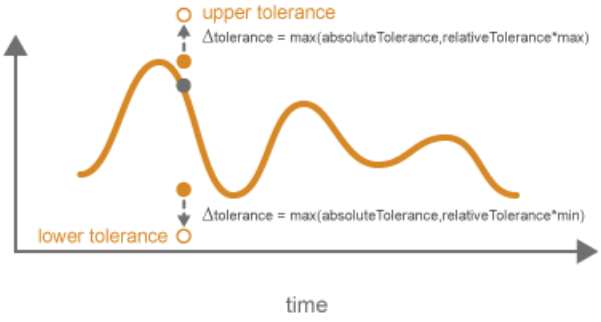
When you specify a time tolerance, the Simulation Data Inspector evaluates the time tolerance first, over a time interval defined as `[(`t_{samp}`-tol), (`t_{samp}`+tol)]` for each sample. The Simulation Data Inspector builds the lower tolerance band by selecting the minimum point on the interval for each sample. Similarly, the maximum point on the interval defines the upper tolerance for each sample.





If you specify a tolerance band using an absolute or relative tolerance in addition to a time tolerance, the Simulation Data Inspector applies the time tolerance first, and then applies the absolute and relative tolerances to the maximum and minimum points selected with the time tolerance.






```
upperTolerance = max + max(absoluteTolerance,relativeTolerance*max)

lowerTolerance = min - max(absoluteTolerance,relativeTolerance*min)
```

See Also

Related Examples

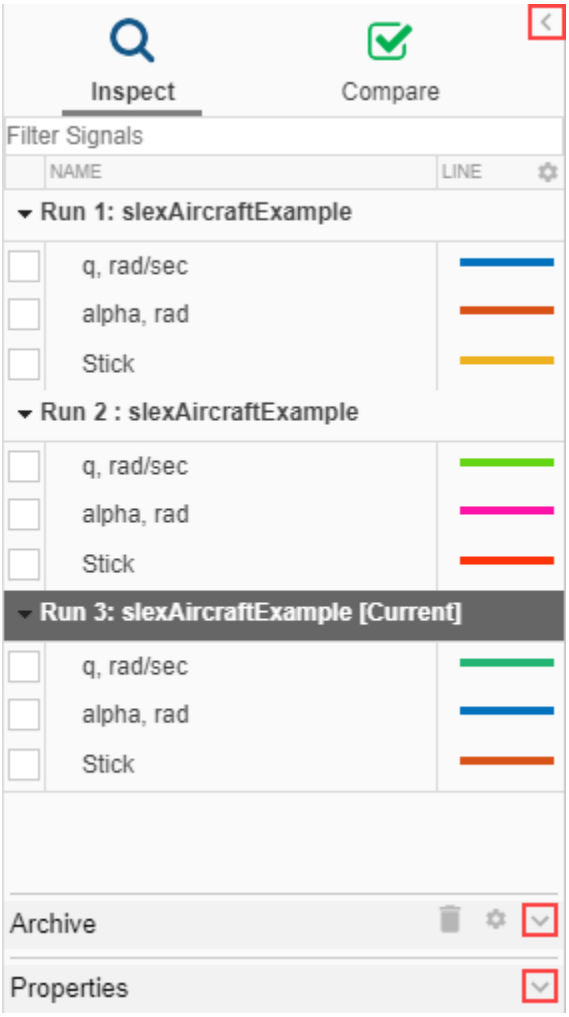
- “Compare Simulation Data” on page 29-130

Organize Your Simulation Data Inspector Workspace

You can modify the layout and content of the panels in the Simulation Data Inspector to help you organize your data. You can set new run naming rules, change how signals are grouped within runs in the navigation pane, and use filters to find the signal you want to inspect.

Modify the Layout

You can collapse and expand the navigation, archive, and Properties panes in the Simulation Data Inspector using the chevrons in the upper-right corner of each pane, highlighted in the image.

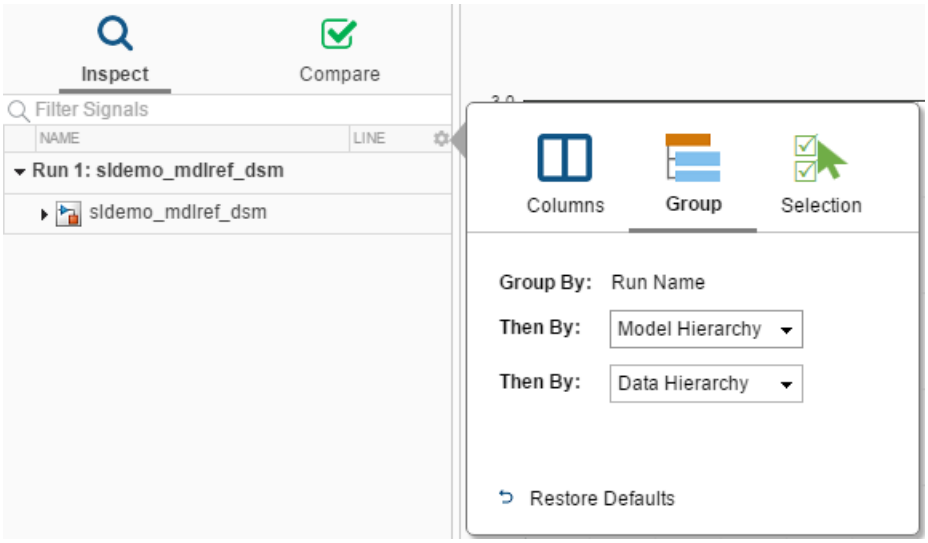


Modify Signal Grouping

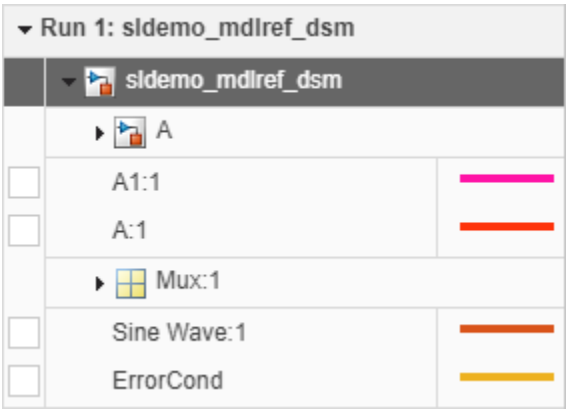
You can customize the hierarchy for how your data is grouped in the **Inspect** and **Compare** panes. The data is grouped first by run name, which cannot be modified. You can then group your data by data or model hierarchy. If you have a Simscape license, you can also group your data by physical system hierarchy. Changes to signal grouping apply to both the **Inspect** and **Compare** panes, regardless of the pane from which you edit the grouping.

As an example, change the Simulation Data Inspector settings to group signals by run name, then by model hierarchy, and then by data hierarchy.

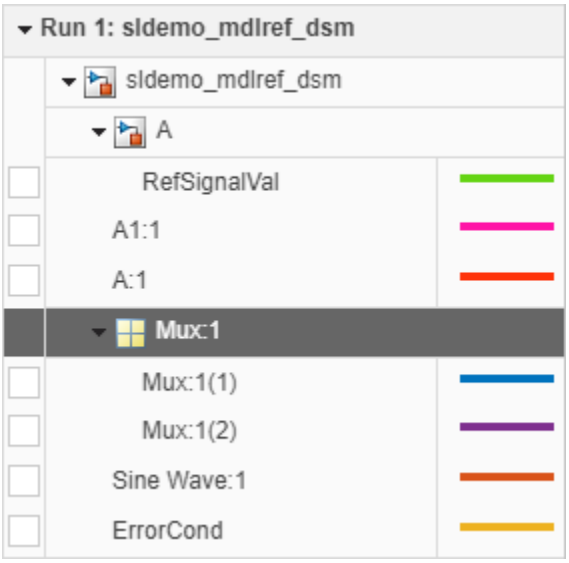
1. Click the **Preferences** button in the upper right of the navigation pane.
2. On the **Group** pane, select `Model Hierarchy` in the first **Then By** list.
3. In the second **Then By** list, select `Data Hierarchy`.



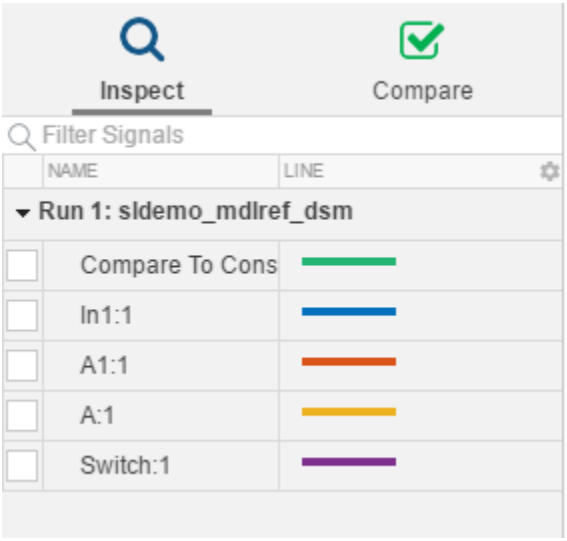
The Simulation Data Inspector groups the signals by run name, then by model hierarchy, and then by data hierarchy. By default, all hierarchies within the run are collapsed. You can expand the model group to see the logged signals.



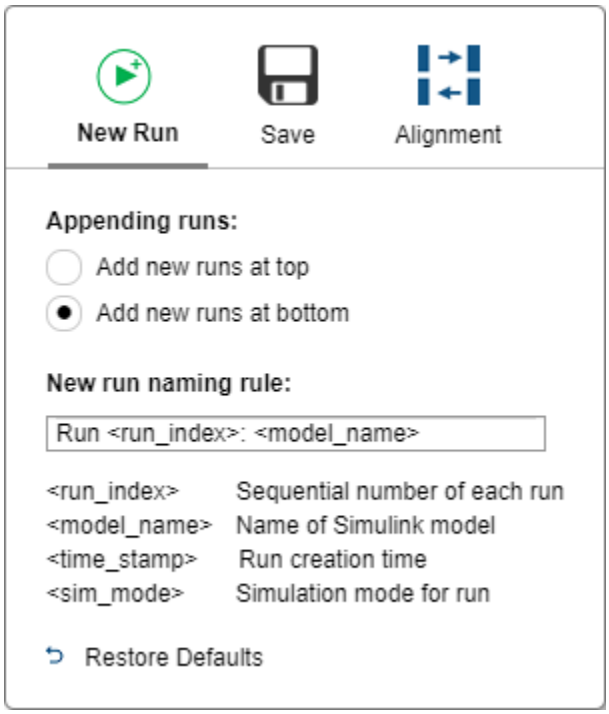
Signals inside subsystem A are still collapsed under A, and the signals in the Mux block output are grouped under Mux: 1. You can expand these groups to see the rest of the signals in the run. The check boxes for signals remain on the left side of the navigation pane, and the signal names indent to indicate the level of hierarchy.



To remove the hierarchy and display a flat list of signals in each run, select `None` from both **Then By** lists on the **Group** pane.



You can also specify whether to add new runs to the top or bottom of the runs list in the **Inspect** pane. The **New Run** tab in the **Preferences** window allows you to configure how new runs are added to the **Inspect** pane. The default configuration adds new runs to the bottom of the runs list.



Specify How the Simulation Data Inspector Names Runs

You can specify how existing and future runs are named in the Simulation Data Inspector.

To rename an existing run double-click the run row, type the new run name, and press **Enter**. Alternatively, you can select the run you want to rename and type the new name into the **Name** row of the properties pane.

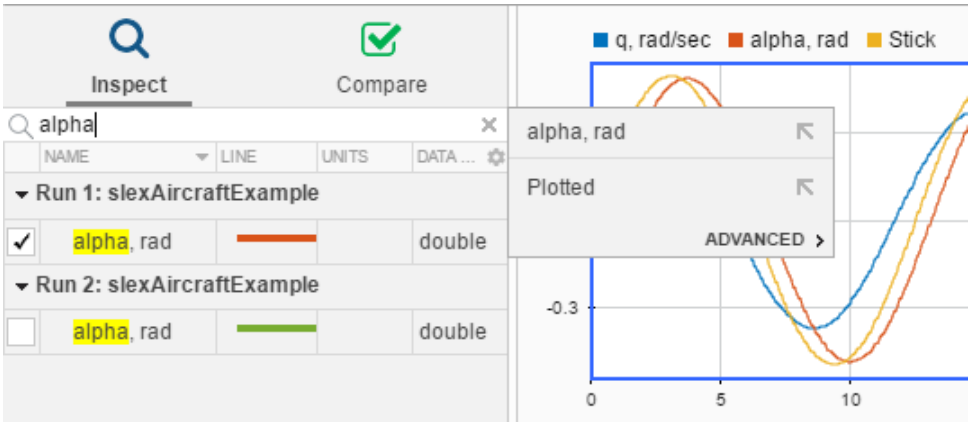
To specify how you would like the Simulation Data Inspector to name future runs, open the **New Run** tab on the **Preferences** menu. The default value for the **New run naming rule** is `Run <run_index>: <model_name>`.

To change the run naming rule, enter your desired options from the list of available parameters along with any other regular characters. For example, to include the simulation mode in subsequent run names, enter `Run <run_index>: <model_name>: <sim_mode>` in the **New run naming rule** box. With this rule, simulating model `slexAircraftExample` in `normal` mode, the name of the first run appears as `Run 1: slexAircraftExample: normal`.

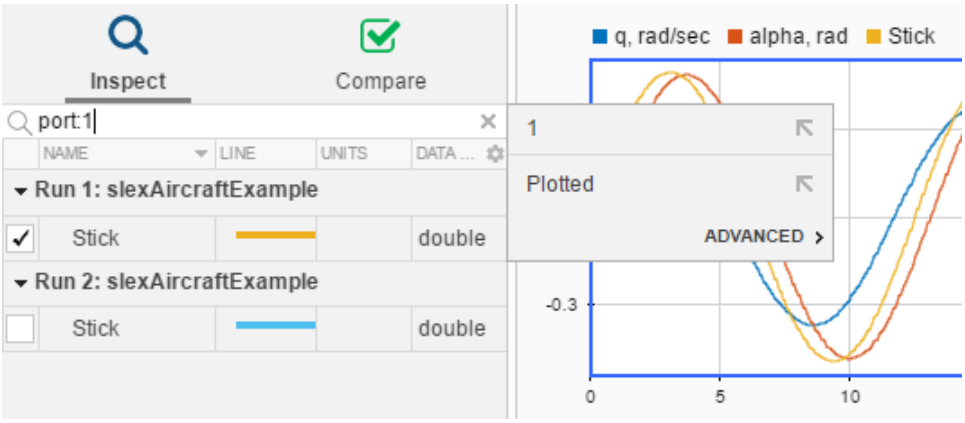
Filter Runs and Signals in the Work Area

You can filter runs and signals displayed in the work area of the **Inspect** pane and in the **Compare** pane to help search through large amounts of data in the Simulation Data Inspector. You can filter the data by text contained in the run or signal names and properties.

To show only signals named `alpha, rad`, type `alpha` into the filter signals text box. Matches for the search criteria are highlighted in the filter results displayed in the pane. The filter dialog box suggests completions for the text typed into the search query.

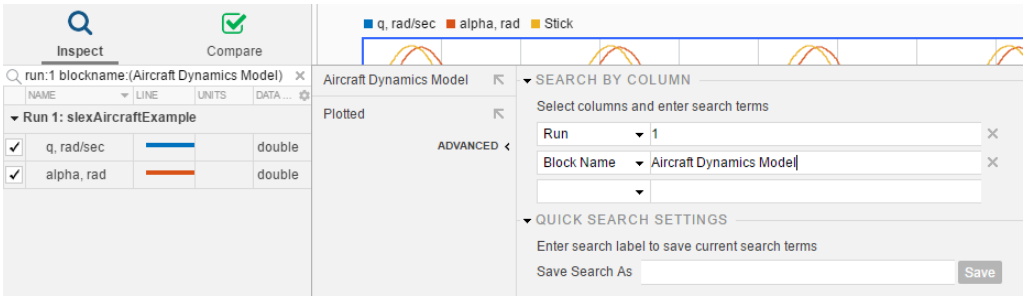


To filter for a signal or run property, use colons to separate the property name and filter value. For example, `port:1` filters for signals that use port 1 in the model. Because the property column is not visible in the **Inspect** pane, the result is not highlighted.



You can also construct more complicated filter queries that include multiple properties using the **Advanced** section of the filter dialog box.

1. Open the **Advanced** section of the filter dialog box.
2. Select a column to add to the filter, and enter the value.



Note Filters work by matching text. For example, an absolute tolerance filter for a value of `00.1` does not return signals with an absolute tolerance of `0.1`.

For convenience, you can save filter configurations. To save the filter, enter a name in the **Save Search As** box and click **Save** on the filter dialog box. Saved filters show as options in the filter list.

See Also

Related Examples

- “View Data in the Simulation Data Inspector” on page 29-2
- “Save and Share Simulation Data Inspector Data and Views” on page 29-83

Inspect and Compare Data Programmatically

You can harness the capabilities of the Simulation Data Inspector from the MATLAB command line using the Simulation Data Inspector API.

The Simulation Data Inspector organizes data in runs and signals, assigning a unique numeric identification to each run and signal. Some Simulation Data Inspector API functions use the run and signal IDs to reference data, rather than accepting the run or signal itself as an input. To access the run IDs in the workspace, you can use `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`. You can access signal IDs through a `Simulink.sdi.Run` object using the `getSignalIDByIndex` method.

The `Simulink.sdi.Run` and `Simulink.sdi.Signal` classes provide access to your data and allow you to view and modify run and signal metadata. You can modify the Simulation Data Inspector preferences using functions like `Simulink.sdi.setSubPlotLayout`, `Simulink.sdi.setRunNamingRule`, and `Simulink.sdi.setMarkersOn`. To restore the Simulation Data Inspector's default settings, use `Simulink.sdi.clearPreferences`.

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

Create `timeseries` objects to contain data for a sine signal and a cosine signal. Give each `timeseries` object a descriptive name.

```
time = linspace(0,20,100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Run and Add Data

Use the `Simulink.sdi.view` function to open the Simulation Data Inspector.

```
Simulink.sdi.view
```

To import data into the Simulation Data Inspector from the workspace, create a `Simulink.sdi.Run` object using the `Simulink.sdi.Run.create` function. Add information about the run to its metadata using the `Name` and `Description` properties of the `Run` object.

```
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';
```

Use the `add` function to add the data you created in the workspace to the empty run.

```
add(sinusoidsRun,'vars',sine_ts,cos_ts);
```

Plot the Data in the Simulation Data Inspector

Use the `getSignalByIndex` function to access `Simulink.sdi.Signal` objects that contain the signal data. You can use the `Simulink.sdi.Signal` object properties to specify the line style and color for the signal and plot it in the Simulation Data Inspector. Specify the `LineColor` and `LineDashed` properties for each signal.

```
sine_sig = getSignalByIndex(sinusoidsRun,1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';

cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.LineDashed = '--';
```

Use the `Simulink.sdi.setSubPlotLayout` function to configure a 2-by-1 subplot layout in the Simulation Data Inspector plotting area. Then use the `plotOnSubplot` function to plot the sine signal on the top subplot and the cosine signal on the lower subplot.

```
Simulink.sdi.setSubPlotLayout(2,1);

plotOnSubPlot(sine_sig,1,1,true);
plotOnSubPlot(cos_sig,2,1,true);
```

Close the Simulation Data Inspector and Save Your Data

When you have finished inspecting the plotted signal data, you can close the Simulation Data Inspector and save the session to an MLDATX file.

```
Simulink.sdi.close('sinusoids.mldatx')
```

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate how to compare the input and output signals of the control system.

Configure and Simulate the Model

The `slexAircraftExample` model does not log data. Load the model and mark the input and output signals for logging.

```
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')
```

Simulate the model. The data for the logged signals logs to the Simulation Data Inspector and to the workspace.

```
out = sim('slexAircraftExample');
```

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the data. The `Simulink.sdi.Run.getLatest` function returns the most recently created run in the Simulation Data Inspector repository. Use the `getSignalIDByIndex` function to access the signal IDs for the logged signals.

```
aircraftRun = Simulink.sdi.Run.getLatest;

signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);
```

Specify Tolerance Values

You can specify tolerance values to use in the comparison as a property in the logged `Simulink.sdi.Signal` object. Use the `Simulink.sdi.getSignal` function to access the `Signal` object using the signal ID.

```
signal1 = Simulink.sdi.getSignal(signalID1);
signal1.AbsTol = 0.1;
```

Compare Signals

Use the `Simulink.sdi.compareSignals` function to compare the input and output signals. This example uses the `isValidSignalID` function to verify that both signal IDs are still valid before calling the `Simulink.sdi.compareSignals` function. A signal ID becomes invalid when the signal is deleted from the Simulation Data Inspector. After the comparison, check the status in the `Simulink.sdi.DiffSignalResult` object.

```
if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    match = sigDiff.Status
end
```

```
match =
OutOfTolerance
```

The comparison result is out of tolerance. You can use the `Simulink.sdi.view` function to inspect and analyze the comparison results.

Compare and Analyze Simulation Data Programmatically

This example shows how to compare runs of simulation data and then analyze and save the results using the Simulation Data Inspector programmatic interface.

Create Simulation Data

First, create simulation data by simulating a model that logs data. This example uses the `ex_sldemo_absbrake` model and analyzes the effect of changing the `Desired relative slip` value.

Load the model. Use the `set_param` function to specify an initial value for the relative slip and simulate the model.

```
load_system('ex_sldemo_absbrake')

set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.24')
out_1 = sim('ex_sldemo_absbrake');
```

Use the `set_param` function to specify a different value for the relative slip and simulate the model again.

```
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.25')
out_2 = sim('ex_sldemo_absbrake');
```

Compare Runs Using Global Tolerance Values

First, use the `Simulink.sdi.getAllRunIDs` function to get the run IDs that correspond to the last two simulation runs.

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

Now, use the `Simulink.sdi.compareRuns` function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

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

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 2
      WithinTolerance: 2
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

Two signal comparisons within the run were within tolerance, and two were out of tolerance.

Plot Comparison Results

You can use plots to analyze the comparison results. Access the signal result for the `Ww` signal from the `DiffRunResult` object that contains the comparison results using the `getResultByIndex` function. Check the `Status` property of the `Simulink.sdi.DiffSignalResult` object.

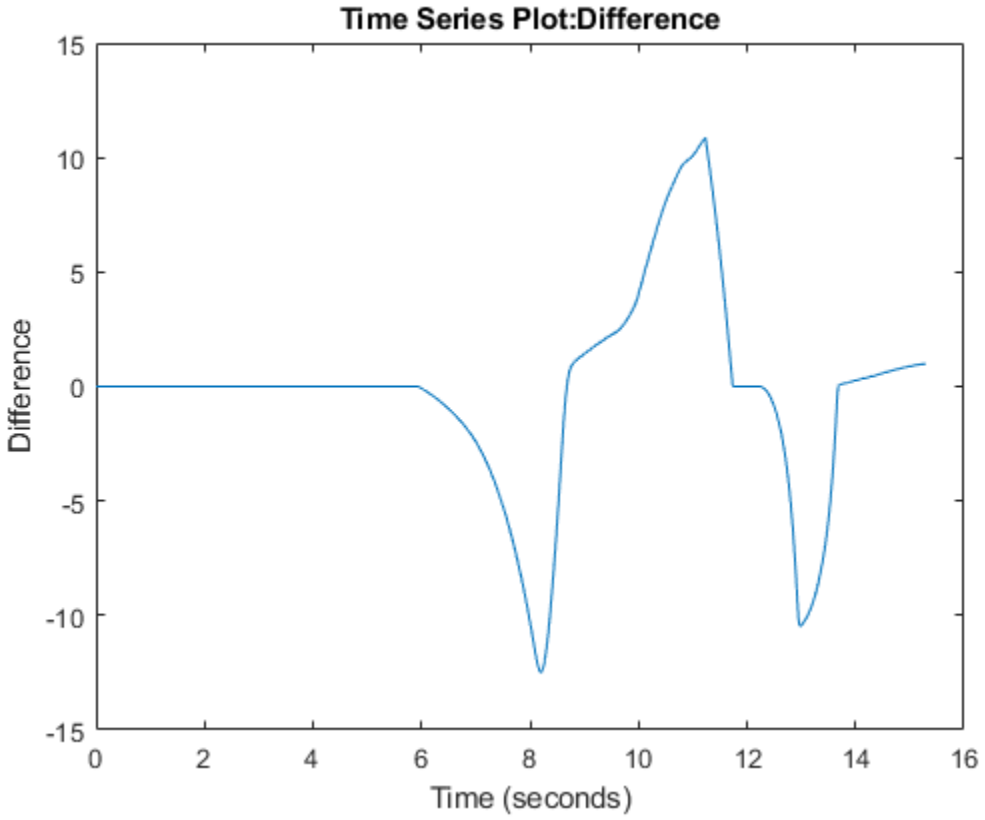
```
signalResult_Ww = getResultByIndex(runResult,1)
```

```
signalResult_Ww =
  DiffSignalResult with properties:

            Name: 'yout.Ww'
          Status: OutOfTolerance
         AlignBy: 'Path'
       SignalID1: 179239
       SignalID2: 179285
   MaxDifference: 12.4878
           Sync1: [1x1 timeseries]
           Sync2: [1x1 timeseries]
            Diff: [1x1 timeseries]
```

```
signalResult_Ww.Status

ans =
OutOfTolerance
```

The `Ww` signal comparison results are out of tolerance. Plot the difference signal to analyze the result.

```
figure(1)
plot(signalResult_Ww.Diff)
```



Save Comparison Results

You can save the comparison results to an MLDATX file to analyze later or to share with a colleague. Use the `saveResult` function to save the run data and comparison results.

```
saveResult(runResult,'desiredSlipResults')
```

The MLDATX file `desiredSlipResults` is created in the working directory. Use the `Simulink.sdi.load` function or the `open` function to view the results in the MLDATX file.

Analyze Simulation Data Using Signal Tolerances

Using the Simulation Data Inspector programmatic interface, you can specify signal tolerance values to use in comparisons. This example uses the `slexAircraftExample` model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

Configure the Model

Load the model and mark signals of interest for logging. This example logs data for the `q` and `alpha` signals.

```
load_system('slexAircraftExample')

Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')
```

Run Simulations

Run simulations with different low-pass filter time constants to generate results to compare. The `slexAircraftExample` model stores variables associated with the model in the model workspace. To modify the time constant value, access the model workspace and use the `assignin` function.

```
out1 = sim('slexAircraftExample');

modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

out2 = sim('slexAircraftExample');
```

Access and Compare Simulation Results

Access the simulation results using the Simulation Data Inspector programmatic interface. Each simulation creates a run in the Simulation Data Inspector with a unique run ID. You use the run IDs to compare the simulation results.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-1);
runIDTs2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then inspect the `Status` property of the signal result to see whether the signals fell within the default tolerance of 0.

```
diffRun1 = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);

sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

sig1Result1.Status

ans =
OutOfTolerance

sig2Result1.Status

ans =
OutOfTolerance
```

Compare Runs with Signal Tolerances

By default, signals use `0` for all tolerance values, so the comparison returns out-of-tolerance results when the signals are not identical. To further analyze the effect of the time constant change, specify tolerance values for the signals. You can specify tolerances for a programmatic comparison using the properties of the `Simulink.sdi.Signal` objects in the runs you compare. The comparison uses the

tolerances specified for the baseline `Signal` object. This example specifies a combination of time and absolute tolerances.

To specify tolerances, first access the `Simulink.sdi.Signal` objects that correspond to each signal in the runs you want to compare.

```
run1 = Simulink.sdi.getRun(runIDTs1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

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

Check the `Name` property to identify each `Signal` object.

```
sig1.Name

ans =
'q, rad/sec'


sig2.Name

ans =
'alpha, rad'
```

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

```
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal using the `AbsTol` and `TimeTol` properties of the `alpha` signal object in the baseline run.

```
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;
```

Compare the runs again and access the results.

```
diffRun2 = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);
```

Check the `Status` property of each signal to determine whether the comparison results fell within the specified tolerances.

```
sig1Result2.Status

ans =
WithinTolerance

sig2Result2.Status
```

```
ans =
WithinTolerance
```

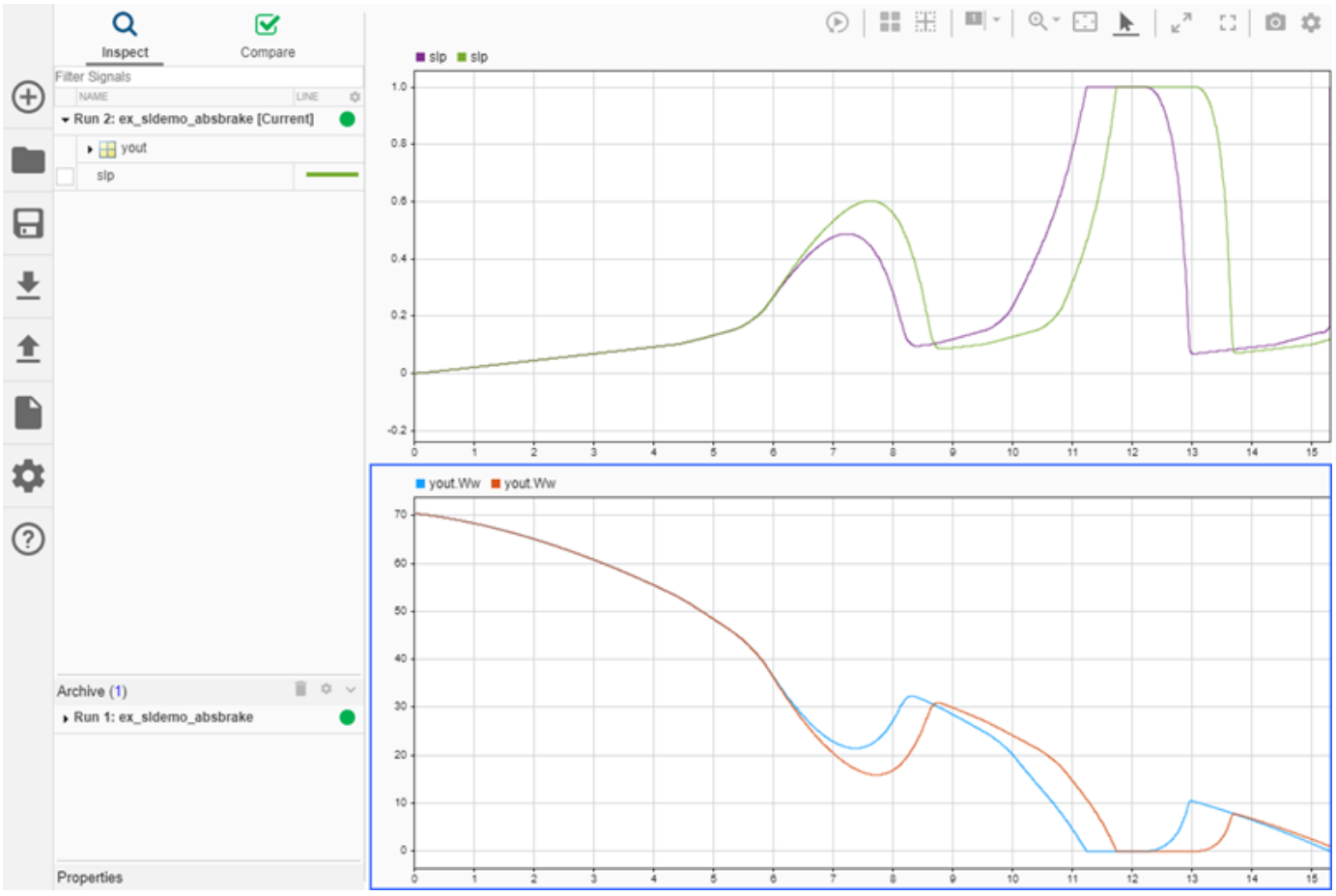
Create a Report for Plotted Signals

Create a report that contains information about and plots of the signals plotted in the **Inspect** pane of the Simulation Data Inspector. By default, the report contains the metadata displayed for signals in the table on the **Inspect** pane. This example shows how to specify which metadata to include in the report.

Load the Session File

This example populates the Simulation Data Inspector with data and plotted signals by loading a saved session file. A session file contains the signal data as well as information about plotted signals and plot layout. Load the session file.

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



Create a Report for Plotted Signals

The report includes plots and metadata for the plotted signals. By default, the report includes the metadata that corresponds to the columns displayed in the signals table on the **Inspect** pane. You can include more data in the report by displaying more columns in the **Inspect** pane. You can also specify

the information you want in the report programmatically using the `'ColumnsToReport'` name-value pair and the enumeration class `Simulink.sdi.SignalMetaData`.

```
signalMetadata = [Simulink.sdi.SignalMetaData.Run, ...
    Simulink.sdi.SignalMetaData.Line, ...
    Simulink.sdi.SignalMetaData.BlockName, ...
    Simulink.sdi.SignalMetaData.SignalName];

Simulink.sdi.report('ReportType','Inspect', 'ReportOutputFile', ...
    'absbrake_slp_report.html', 'ColumnsToReport', signalMetadata);
```

The report shows tables of the metadata for plotted signals, organized by run, above a snapshot of the plot.

Save and Restore a Set of Logged Signals

This example shows how to use the `Simulink.HMI.InstrumentedSignals` object to save a set of logged signals to restore after running a simulation with a different signal logging configuration.

Save the Initial Signal Logging Configuration

This example uses the `sldemo_fuelsys` model, which is configured to log 10 signals. Open the model and use the `get_param` function to get a `Simulink.HMI.InstrumentedSignals` object representing the signal logging configuration.

```
load_system sldemo_fuelsys

initSigs = get_param('sldemo_fuelsys','InstrumentedSignals');
```

You can save the initial signal logging configuration in a MAT-file for later use.

```
save initial_instSigs.mat initSigs
```

Remove All Logging Badges

To return to a baseline of no logged signals, you can use the `set_param` function to remove all logging badges from signals in your model. Then, you can easily select a different configuration of signals to log in the Simulink™ Editor or using the `Simulink.sdi.markSignalForStreaming` function.

```
set_param('sldemo_fuelsys','InstrumentedSignals',[])
```

Restore Saved Logging Configuration

After working with a different set of logged signals, you can restore a saved configuration using the `Simulink.HMI.InstrumentedSignals` object. For example, if you saved the logging configuration to a MAT-file, you can load the MAT-file contents into the workspace and use the `set_param` function to restore the previously saved logging configuration.

```
load initial_instSigs.mat

set_param('sldemo_fuelsys','InstrumentedSignals',initSigs)
```

See Also

Simulation Data Inspector

Related Examples

- “Compare Simulation Data” on page 29-130
- “How the Simulation Data Inspector Compares Data” on page 29-139
- “Create Plots Using the Simulation Data Inspector” on page 29-94
- “Organize Your Simulation Data Inspector Workspace” on page 29-144

Keyboard Shortcuts for the Simulation Data Inspector

You can use several keyboard shortcuts to facilitate working with the Simulation Data Inspector. In the table, where the shortcut looks like **Ctrl+N**, to use the shortcut, you hold down the **Ctrl** key and then press the **N** key.

Note On Macintosh platforms, use the **command** key instead of **Ctrl**.

General Actions

Task	Shortcut
Start a new session	**Ctrl+N**
Open a session	**Ctrl+O**
Save a session	**Ctrl+S**
Compare	**Ctrl+E**
Link/Unlink a subplot	**Ctrl+U**
Delete a run or signal	**Delete**

Plot Zooming

Task	Shortcut
Zoom in T (time)	**Ctrl+Shift+T**
Zoom in Y	**Ctrl+Shift+Y**
Zoom in T and Y	**Ctrl++** (Numeric keypad only)
Zoom out	**Ctrl+-** (Numeric keypad only)
Fit to view	**Spacebar**
Cancel zoom operation or signal dragging	**Esc**

Data Cursors

Task	Shortcut
Show a data cursor	**Ctrl+I**
Hide all data cursors	**Shift+Del**
Move a selected data cursor to next data point	**Right arrow**
Move a selected data cursor to previous data point	**Left arrow**
Activate first (left) cursor	**Ctrl+1**
Activate second (right) cursor	**Ctrl+2**

Import Dialog Box

These actions pertain to the import table.

Task	Shortcut
Expand all nodes	**Ctrl+=**
Collapse all nodes	**Shift+Ctrl+=**
Select a node	**Space**
Expand a single node	**Right arrow**
Collapse a single node	**Left arrow**

See Also

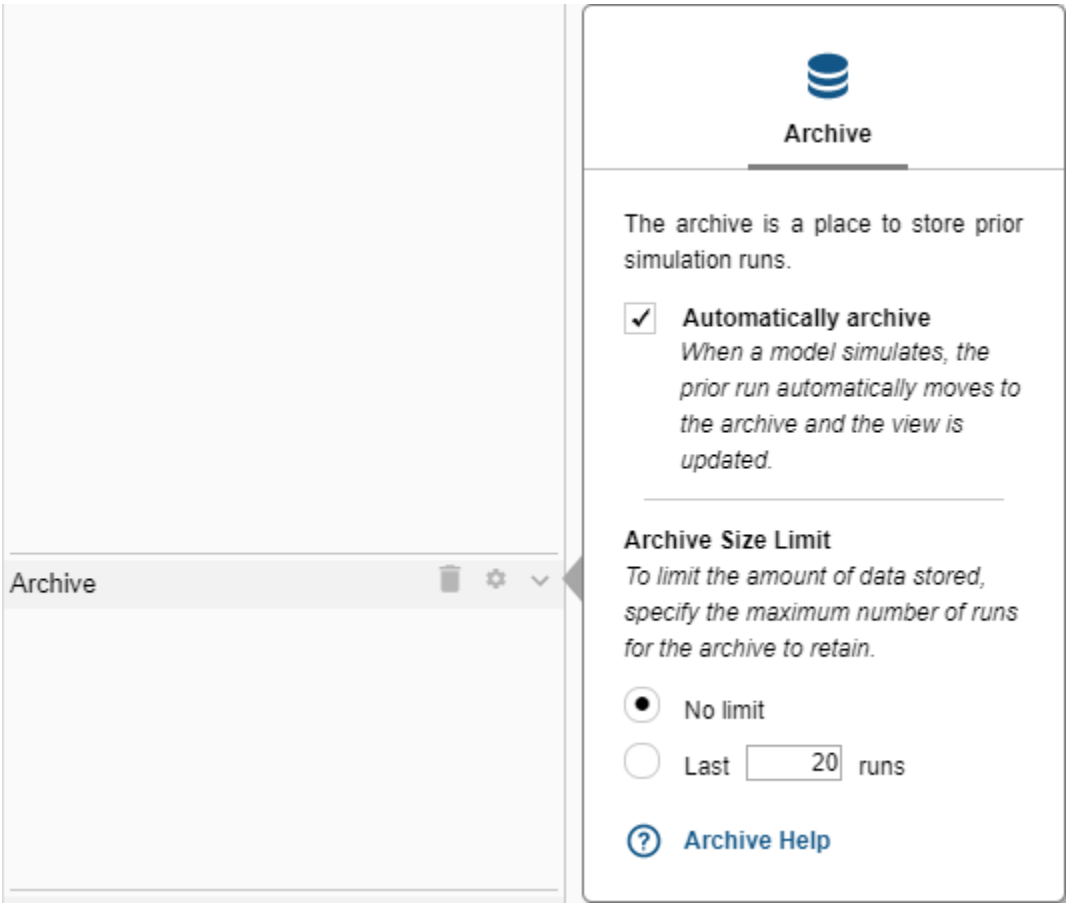
Simulation Data Inspector

Related Examples

- “Organize Your Simulation Data Inspector Workspace” on page 29-144

The Simulation Data Inspector Archive

The archive in the Simulation Data Inspector stores runs in a collapsible pane, allowing you to manage the contents of your work area without deleting run data. You can configure the Simulation Data Inspector to automatically move prior simulation runs to the archive or not, and you can limit the number of runs stored in the archive.



Manage Runs in the Archive

By default, the Simulation Data Inspector automatically archives simulation runs. When you simulate a model, the Simulation Data Inspector automatically moves the prior simulation run to the archive and updates the view to show aligned signals from the current run.

The archive does not impose any functional limitations on the runs and signals it contains. You can plot signals from the archive, and you can use runs and signals in the archive in comparisons. You can drag runs of interest from the archive to the work area and vice versa. You can manually move runs between the work area and archive whether the **Automatically Archive** check box is enabled or disabled. If you want to manage the data in the archive entirely manually, clear the **Automatically Archive** check box on the **Archive** settings pane.

When you import runs into the Simulation Data Inspector, the imported runs appear in the work area, and the **Current** tag remains on the most recent simulation run. You can import signals to existing runs in the work area and in the archive.

To delete all the runs in the archive, click the trash icon.

Limit Data Retention

To reduce the amount of data retained by the Simulation Data Inspector, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the **Archive Size Limit**, the Simulation Data Inspector starts to delete runs on a first in first out basis. By default, the Simulation Data Inspector has no limit on the number of runs stored in the archive. To specify the maximum number of runs stored in the archive, select the **Last n runs** radio button, and enter your desired limit. If you enter a limit that would delete runs already in the archive, the Simulation Data Inspector gives a warning. You can also delete the contents of the archive manually, using the trash icon.

See Also

`Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.getAutoArchiveMode` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

More About

- “Iterate Model Design Using the Simulation Data Inspector” on page 29-71
- “Organize Your Simulation Data Inspector Workspace” on page 29-144

Tune and Visualize Your Model with Dashboard Blocks

In this section...

The blocks in the Dashboard library help you control and visualize your model during simulation and while the simulation is paused. This example uses the Fault-Tolerant Fuel Control System model to showcase the control and visualization capabilities of Dashboard blocks.

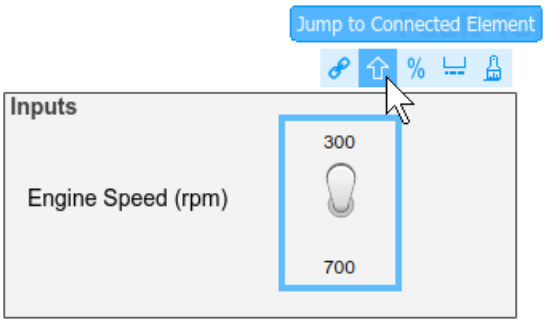
To open the model, enter `sldemo_fuelsys` into the MATLAB command window. To open the Dashboard subsystem, double-click it or click the Open the Dashboard link.

Note Dashboard blocks cannot connect to signals inside referenced models or subsystems.

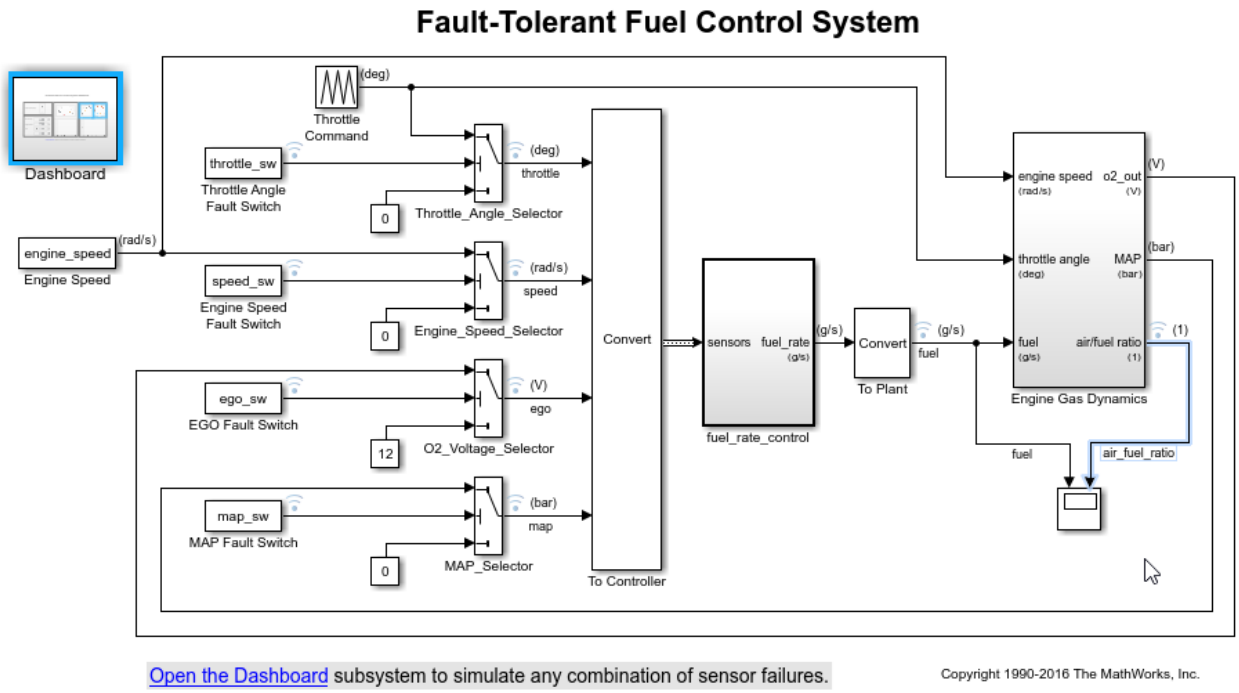
Explore Connections Within the Model

The Dashboard subsystem contains blocks for controlling and visualizing signals in the Fault-Tolerant Fuel Control System model. Explore the connections between the signals and Dashboard blocks. Click either a signal or a Dashboard block to highlight the connections.

From the Dashboard subsystem, click the Toggle Switch in the Fuel panel. Hover the mouse over the ellipsis above the block and then click the arrow above it to jump to the connected block or signal.



From the top level of the model, click the `air_fuel_ratio` signal and see the Dashboard subsystem, Quarter Gauge, and Half Gauge highlighted.



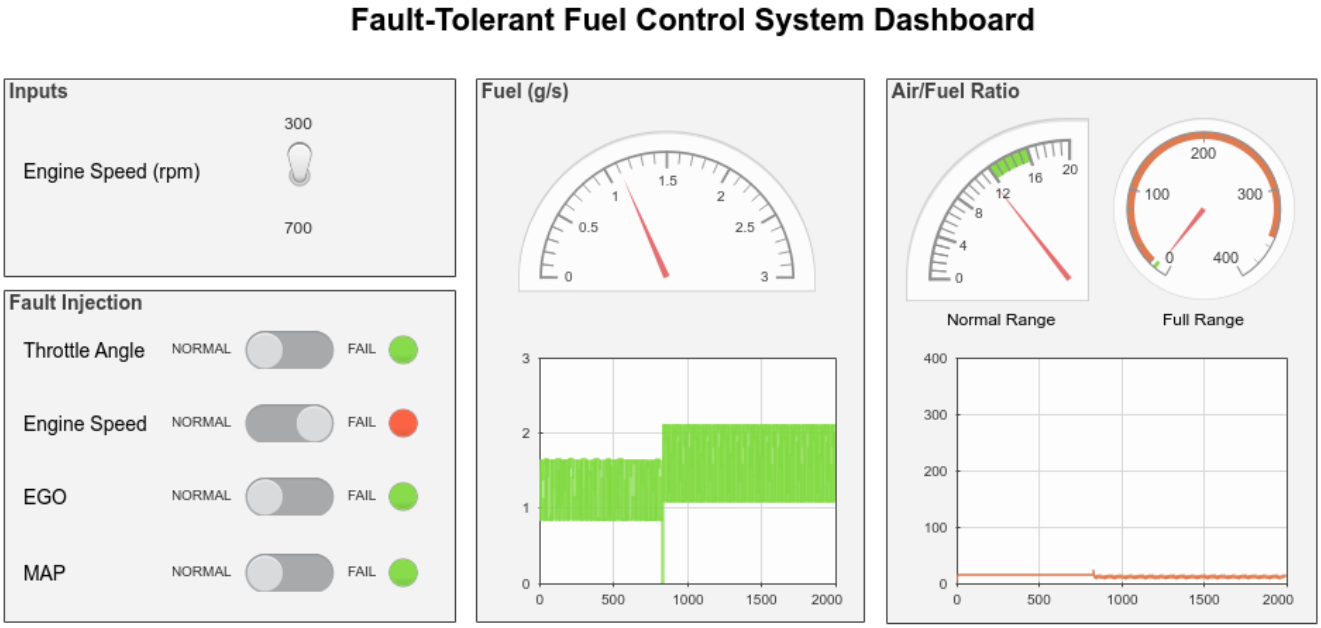
Simulate Changing Model States

In the Dashboard subsystem, switches provide control over the state of the throttle angle, engine speed, EGO, and MAP signals. For each sensor signal, the switch toggles between `normal` and `fail`, allowing you to simulate the system response to each single-point failure. Clicking any one of these switches before simulation, during simulation, or while a simulation is paused changes the state in the model.

Run the simulation and observe the control system response to each single-point failure.

1. Start the simulation.
2. As the simulation runs, click one of the switches to simulate a component failure.

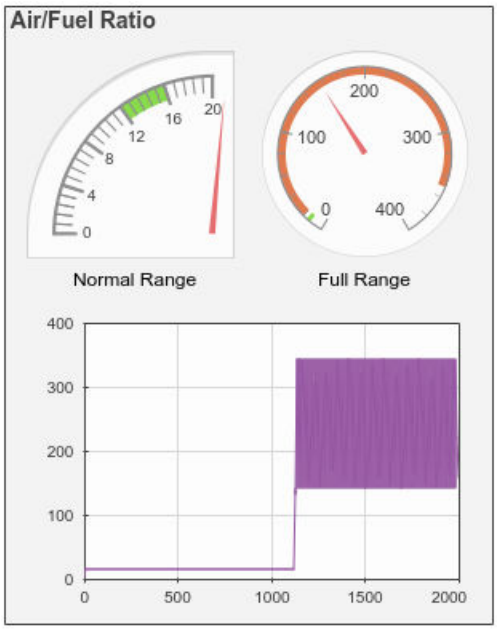
   Observe the changes in the `fuel` and `air_fuel_ratio` signals in the Dashboard Scope and Gauge blocks when you flip each switch.
3. Stop the simulation when you are finished.



View Signal Data

Dashboard blocks allow you to view signal data using gauges, lamps, and dashboard scopes. In this example, the Dashboard blocks provide visual feedback about the fuel input and air-to-fuel ratio during simulation, after simulation, and while a simulation is paused.

To capture different types of information and more effectively visualize a signal, connect multiple Dashboard blocks to a single signal. For example, you can visualize the `air_fuel_ratio` signal using the Gauge, Quarter Gauge, and Dashboard Scope blocks.

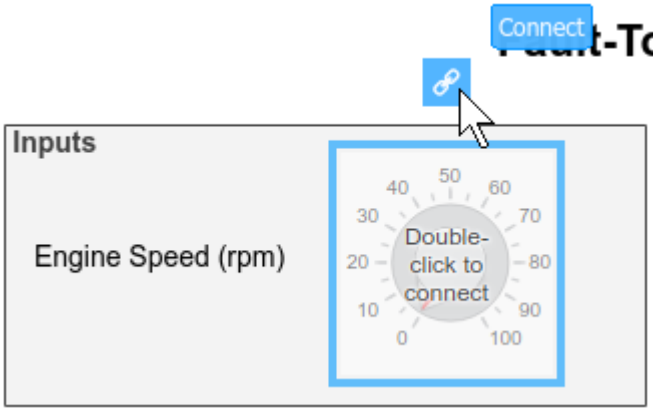


Use the Quarter Gauge block, labeled Normal Range in the example, to see small fluctuations in the instantaneous `air_fuel_ratio` signal while its value lies within the normal operational range. The Gauge block, labeled Full Range, allows you to see the behavior of the instantaneous `air_fuel_ratio` signal outside of the normal range. The Dashboard Scope block shows the variations of the `air_fuel_ratio` signal over time.

Tune Parameters During Simulation

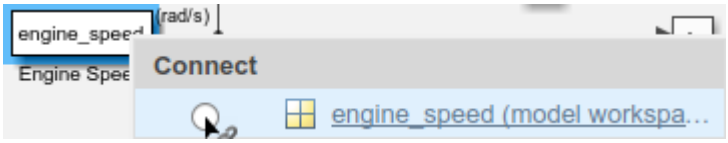
Dashboard blocks allow you to tune parameters in your model during a simulation. To explore the tuning capability within the fuel system model, replace the engine speed Toggle Switch block with a Knob:

1. Delete the engine speed Toggle Switch.
2. Add a Knob block from the Dashboard library.
3. Click the **Connect** button that appears when you pause on the Knob block. When you click the **Connect** button, the Simulink Editor enters connect mode, which facilitates connecting Dashboard blocks to signals and parameters in your model.



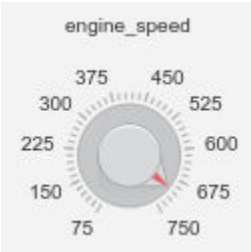
4 Navigate to the top level of the model and select the Engine Speed block. The Engine Speed block is a Constant block whose **Constant value** parameter you can tune with the Knob block.

When you select the Engine Speed block, the names of tunable parameters associated with the block appear in a table.



5 Select the option button next to `engine_speed` in the table. Then, to exit connect mode, click the **Exit** button in the upper-right of the model canvas.

Now that the Knob block is connected to the `engine_speed` parameter, you can set the tick interval and range to values that make sense for the simulation. Double-click the Knob block to access the block parameters. In this example, set **Minimum** to `75`, **Maximum** to `750`, and **Tick Interval** to `75`. Then, click **OK**.



Simulate the model and tune the `engine_speed` parameter using the Knob.

1 Start the simulation.

2 As the simulation runs, drag the pointer on the Knob to adjust the value of `engine_speed`.

Notice as you use the Knob to adjust the value of `engine_speed`, the `air_fuel_ratio` value displayed on the Gauge blocks and in the Dashboard Scope block changes.

3 Stop the simulation when you have finished tuning the parameter.

See Also

Related Examples

- “Decide How to Visualize Simulation Data” on page 30-2

Interactively Design and Debug Models Using Panels

You can use a panel containing one or more Dashboard blocks to help you design and debug a model. A panel floats above the model canvas and follows you throughout a model hierarchy. With panels, you can control and monitor components of your model in place, without adding collections of Dashboard blocks throughout the model.

When you use panels, the blocks in the panel are not associated with the model in the same way as blocks in the canvas. For example, you cannot programmatically interact with blocks in panels using `get_param`, `set_param`, `gcb`, or `gcbh`. Otherwise, blocks promoted to panels retain their interactive behavior. You can connect Dashboard blocks in a panel using connect mode, inspect block properties using the Property Inspector, and modify connections during simulation.

This example uses the `sldemo_fuelsys` model to illustrate the steps required to create panels and how you can use panels to debug your models. To open the model, type `sldemo_fuelsys` into the MATLAB Command Window.

Create a New Panel

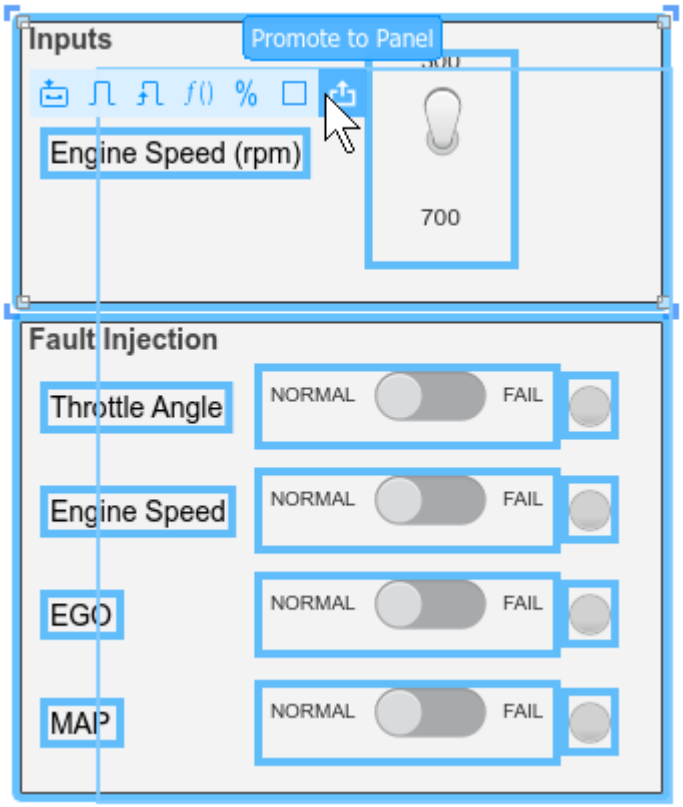
The `sldemo_fuelsys` model has a `Dashboard` subsystem that contains controls and indicators for interactively simulating the model. You can use panels in addition to or instead of using a `Dashboard` subsystem or including Dashboard blocks throughout your model. For example, you can create three panels, each containing a subset of the blocks in the `Dashboard` subsystem:

- A control panel, including the blocks in the **Inputs** and **Fault Injection** areas.
- A visualization panel for the `fuel` signal, including the blocks in the **Fuel (g/s)** area.
- A visualization panel for the `air_fuel_ratio` signal, including the blocks in the **Air/Fuel Ratio** area.

You do not need to separate controls and visualizations into different panels. A single panel can contain both kinds of blocks. For this example, the modular division allows you to have access to the controls while choosing which signal to monitor, depending on the subsystem you need to understand or debug.

Navigate inside the `Dashboard` subsystem, and create the controls panel:

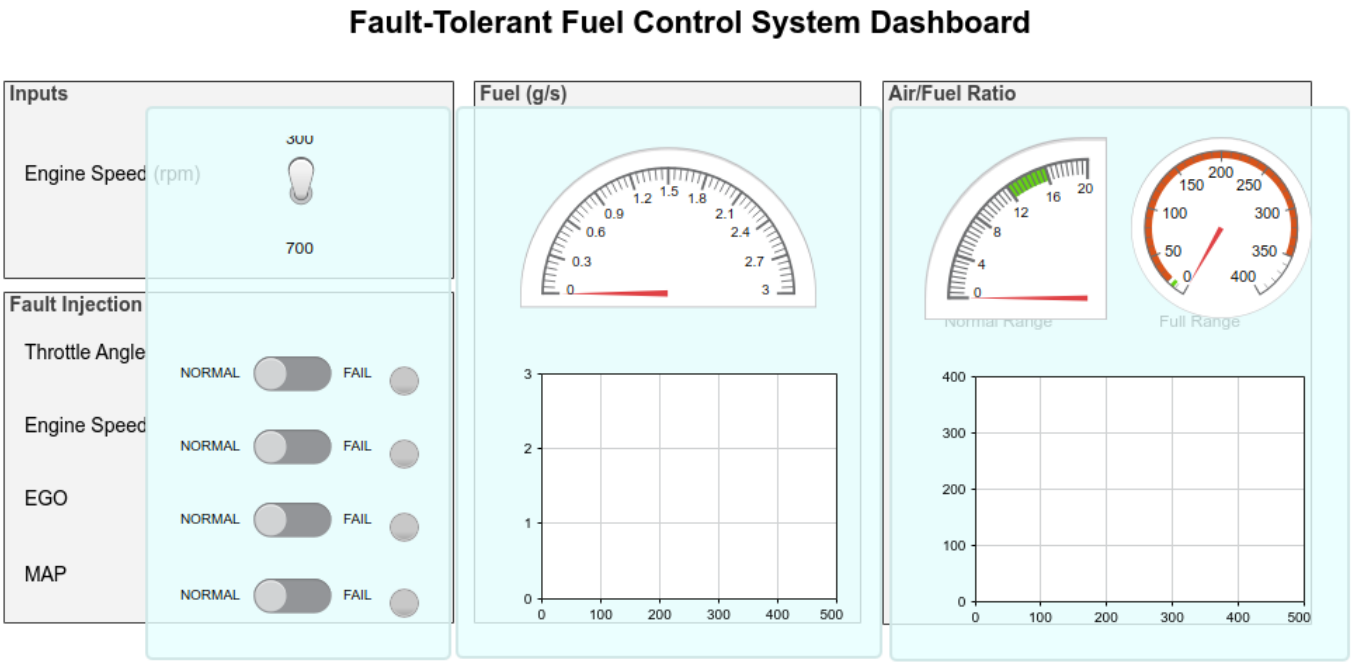
1. Select the blocks inside the areas labeled **Inputs** and **Fault Injection**.
2. To promote the selection of blocks to a panel, pause on the ellipsis that appears at the end of the selection and select **Promote to Panel** from the menu that appears.



Note You can only promote Dashboard blocks and blocks in the Aerospace Blockset™ Flight Instruments library to a panel. When you try to promote a selection that contains model elements other than Dashboard blocks, only the Dashboard blocks promote to the panel.

When you promote blocks to a panel, the blocks move from the canvas into the panel. If you want to move a block back into the canvas from a panel, enter edit mode and drag the block out of the panel and into the canvas. To enter edit mode for a panel, select the panel, pause on the ellipsis that appears above the selected panel, and select **Edit Panel**.

Follow similar steps to create two more panels: one containing the blocks in the **Fuel (g/s)** area, and another containing the blocks in the **Air/Fuel Ratio** area.



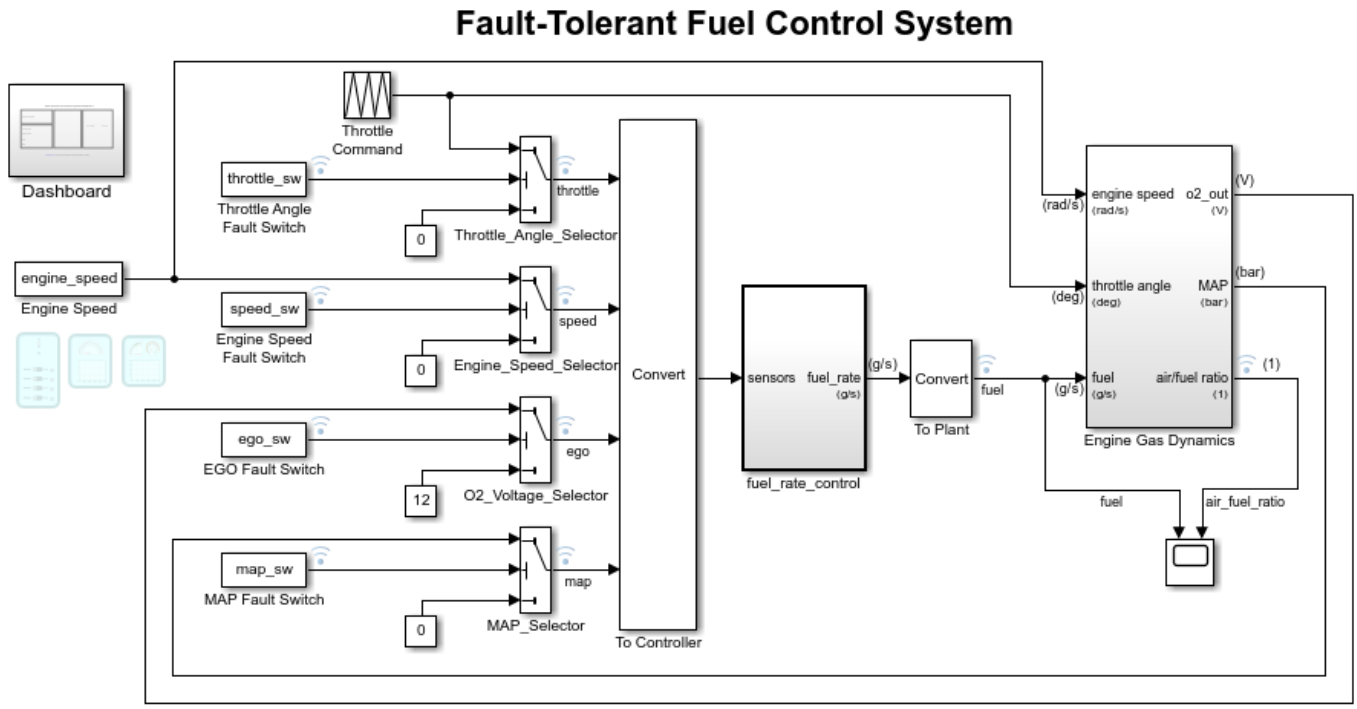
Note To use panels saved in a referenced model, open the referenced model as a top model.

Manage Panels in Your Model

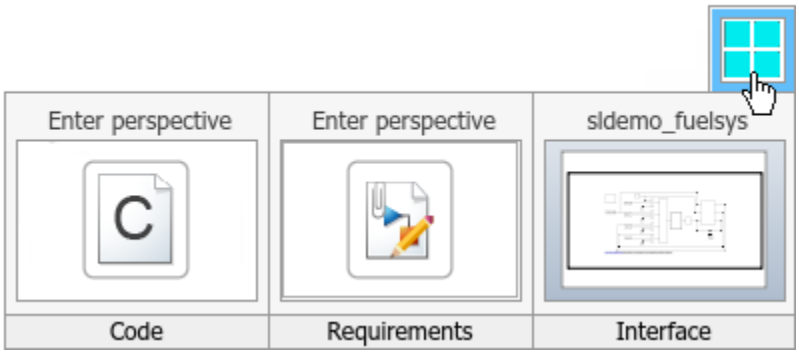
When you use panels in your model, the panel remains accessible, floating over the canvas, no matter where you are in a model hierarchy. For example, navigate to the top level of the `sldemo_fuelsys` model. The panels remain visible and in the same position, similar to a heads-up display.

You can adjust the size of a panel from the corners. When you resize a panel outside of edit mode, the contents of the panel scale proportionally. To resize a panel without scaling the contents, enter edit mode as described in “Edit and Annotate a Panel” on page 29-173. You can reposition a panel by selecting an empty area in the panel and dragging.

If you need to inspect or edit your model, you can minimize one or more panels by double-clicking each panel. Minimized panels remain visible in the model, represented by icons. You can move the icon to a convenient place in your model while you edit or inspect the model contents. When you need to use the panel, you can restore the panel by double-clicking the icon.



You can also use the **Manage Panels** dialog to control panel visibility in the model. To access all panels available in a model or change the visibility of a panel, select **Manage Panels** from the perspectives controls. You can access the perspectives controls by clicking the lower right corner of the Simulink Editor.



The **Manage Panels** dialog is only available when your model includes panels. If there are no panels in your model, the **Manage Panels** dialog does not show in the perspectives controls.

The **Manage Panels** dialog hovers over the canvas and displays an icon for each panel in the model. Because this model contains three panels, you see three icons. You can use the **Manage Panels** dialog to hide or show individual panels in your model, depending on the relevance of each panel to your present workflow. Click an icon to toggle the visibility for the corresponding panel. When panels are hidden, the icon becomes more transparent in the dialog, and the panel is not visible in the model. You can use the menu above the **Manage Panels** dialog to show or hide all panels in the model.

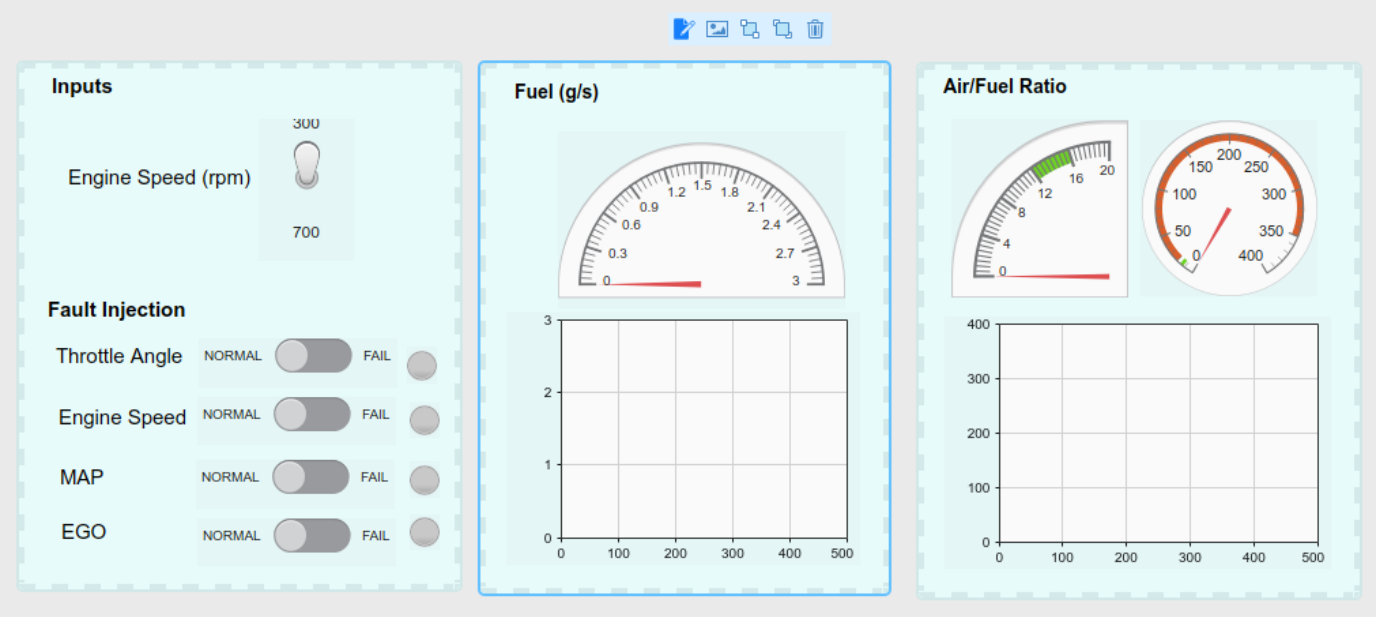
Because panels are positioned relative to the canvas and do not scale with zoom actions, you may lose track of one or more panels in the model as you design and debug. Use the **Fit Visible Panels to View** button to gather all visible panels in your model into the visible canvas.

Edit and Annotate a Panel

You can edit and annotate panels using edit mode. To enter edit mode for a panel, select the panel, pause on the ellipsis that appears above the selected panel, and select **Edit Panel**.

In edit mode, you can resize panels without resizing the panel contents, delete a panel, and change the appearance of a panel. For example, you could specify a background image to use as the panel background, or you can add annotations to label the contents of the panel. You can also add and remove blocks in edit mode by dragging blocks into or out of the panel. When your model has multiple panels, you can move blocks and labels between panels in edit mode.

Add annotations to each panel to label each interactive element. To add an annotation, double-click an empty part of the panel and start typing. Use the rich text menu to modify the annotations. For example, make section labels bold. To edit another panel while in edit mode, select the next panel. Clicking outside of a panel exits edit mode.



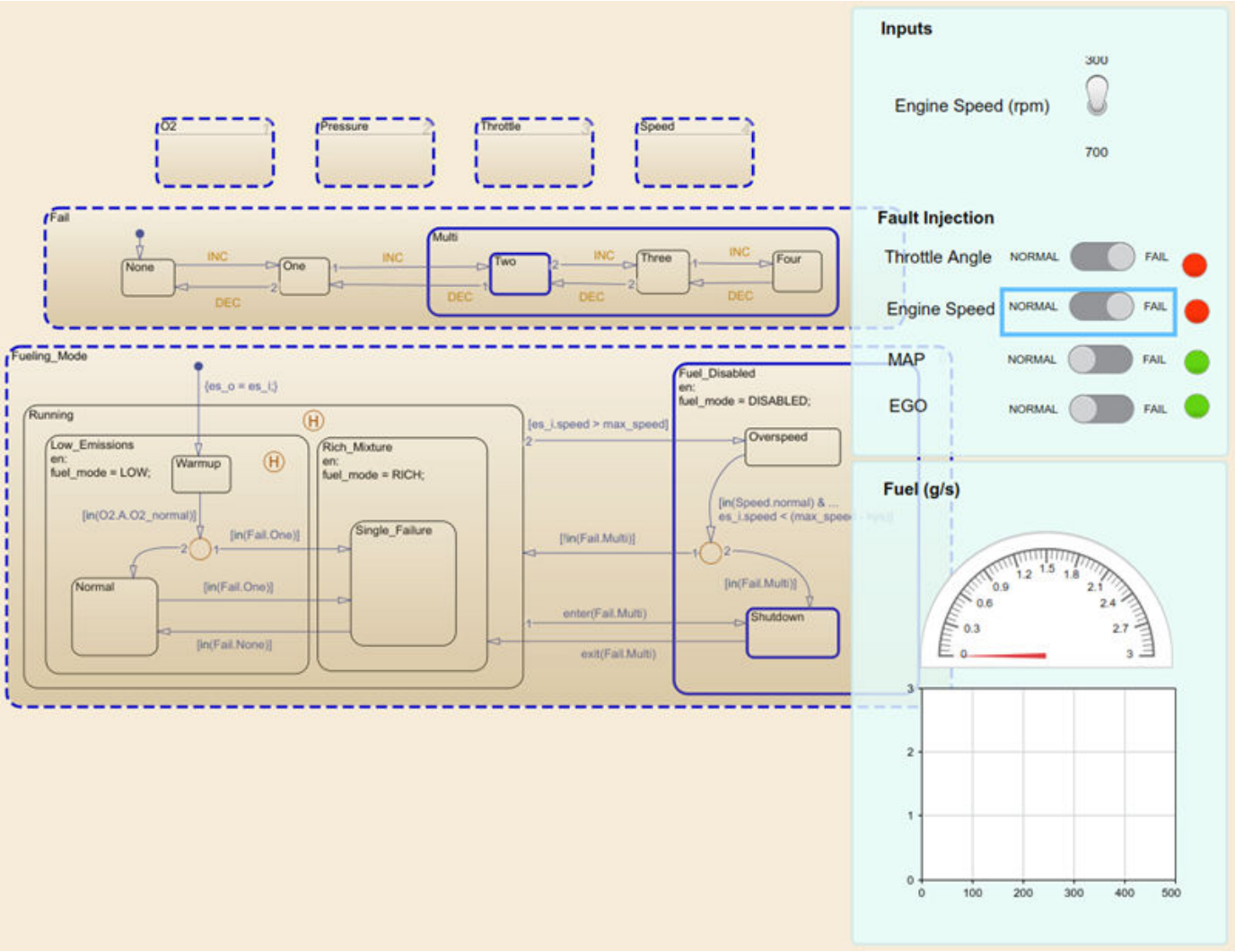
When you finish editing and annotating the panel, exit edit mode by selecting the **Done Editing** option in the menu above the panel or clicking outside of the panel.

Interactively Simulate a Model Using Panels

Now you have modular panels to use while interactively simulating the `sldemo_fuelsys` model. Suppose you need to understand and debug the `fuel_rate_control` subsystem. In this subsystem, the controls panel and the panel that visualizes the `fuel` signal are most useful. Before starting a simulation, navigate inside the `fuel_rate_control` subsystem and then inside the `control_logic` Stateflow chart. Minimize or hide the panel that visualizes the `air_fuel_ratio` signal, and arrange the controls panel and fuel panel so you can see the contents of the chart.

Start a simulation. Once the simulation starts, modify the position for one or more **Fault Injection** Slider Switch blocks to induce a failure in the system. To modify the value a Dashboard block passes to the model, select the block and then click to select the new value. Observe the changes in the Stateflow chart as the system state changes in response to one or more failures.

Tip If the simulation progresses too quickly, you can use simulation pacing to slow the progression of simulation time relative to clock time. For more information, see “Simulation Pacing” on page 2-17.



See Also

More About

- “Tune and Visualize Your Model with Dashboard Blocks” on page 29-164
- “Simulation Pacing” on page 2-17

Analyzing Simulation Results

Decide How to Visualize Simulation Data

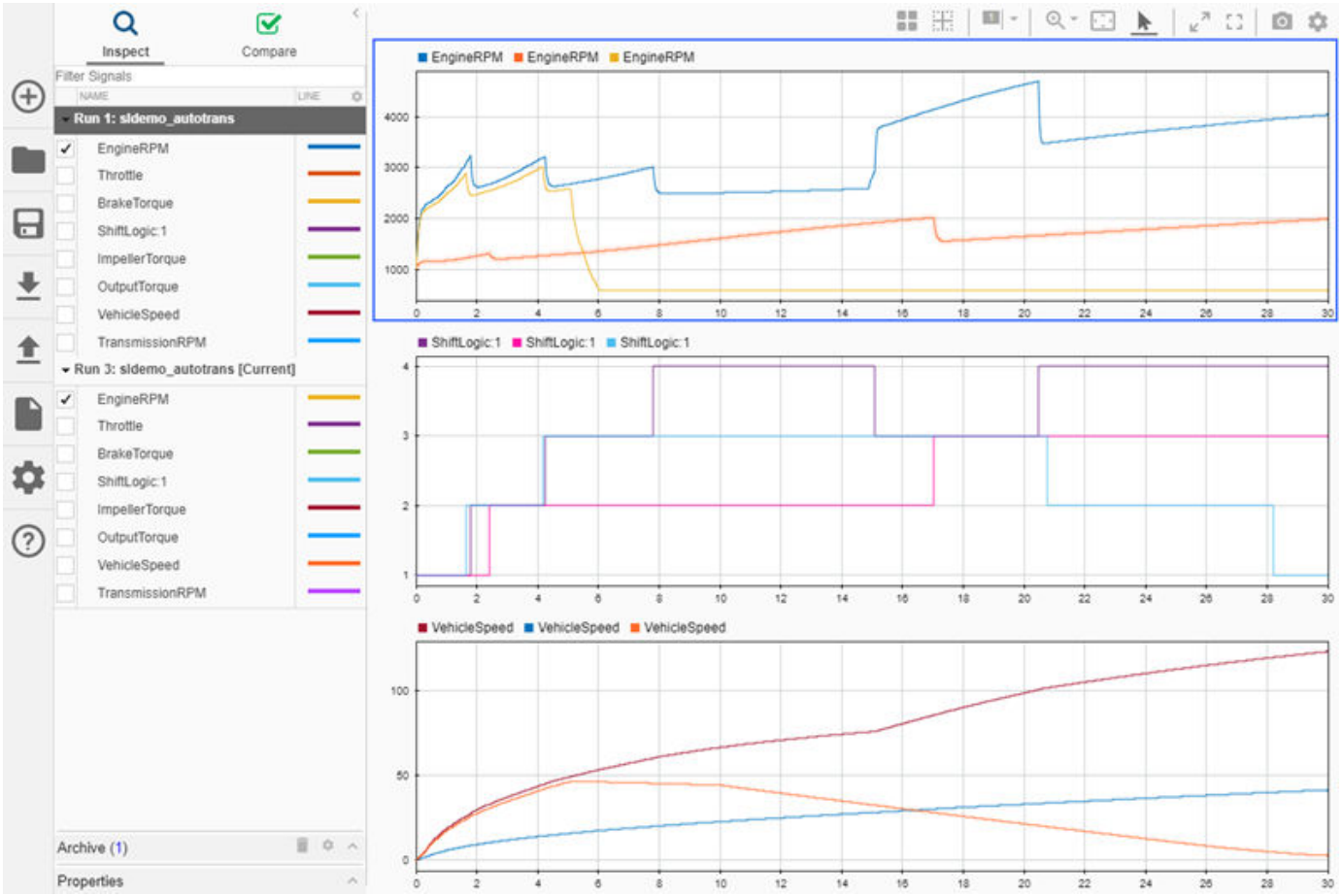
Visualizing simulation data throughout the modeling process helps you understand and tune model behavior. Simulink offers several complementary visualization tools you can use throughout the modeling process. Some visualization tools also offer the ability to save simulation data. Learn about each technique so you can choose the right tools to visualize your model data.

- Simulation Data Inspector — View, analyze, and compare logged data throughout the modeling process.
- Scope block, Floating Scope block, and Scope Viewer — Debug and analyze data using an environment and capabilities similar to a bench-top oscilloscope.
- Dashboard blocks — Build an interactive interface to tune and monitor a model.
- Port value displays — View instantaneous signal values while debugging.
- Custom MATLAB visualization — Write MATLAB code to visualize data.

Simulation Data Inspector

You can use the **Simulation Data Inspector** for most visualization tasks. The Simulation Data Inspector integrates with data logging in your model and works well for visualizing many signals throughout a model. Use comparisons in the Simulation Data Inspector to prototype, debug, tune, calibrate, and test your models. The Simulation Data Inspector supports:

- Viewing signals during simulation
- Logging, importing, and exporting data
- Configurable subplot layouts and visualization settings
- Viewing data using multiple visualization options, including maps and XY plots
- Post-processing and data analysis using comparisons with tolerance values
- Saving plots and data to share or archive results



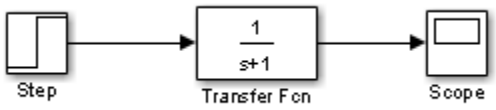
To visualize simulation data using the Simulation Data Inspector, log data in the model. When you log signals and outputs, the logged data is automatically available in the Simulation Data Inspector during and after simulation. In the model Configuration Parameters, select **Data Import/Export > Record logged workspace data in Simulation Data Inspector** to make logged states and Simscape data available in the Simulation Data Inspector after simulation. When you want to view signals without logging the data, consider using a Scope block, the Floating Scope block, or a Scope Viewer.

Open the Simulation Data Inspector from **Simulation > Data Inspector** or by clicking a signal logging badge. When you open the Simulation Data Inspector by clicking the logging badge on a signal, the signal is automatically plotted.

For more information, see **Simulation Data Inspector**.

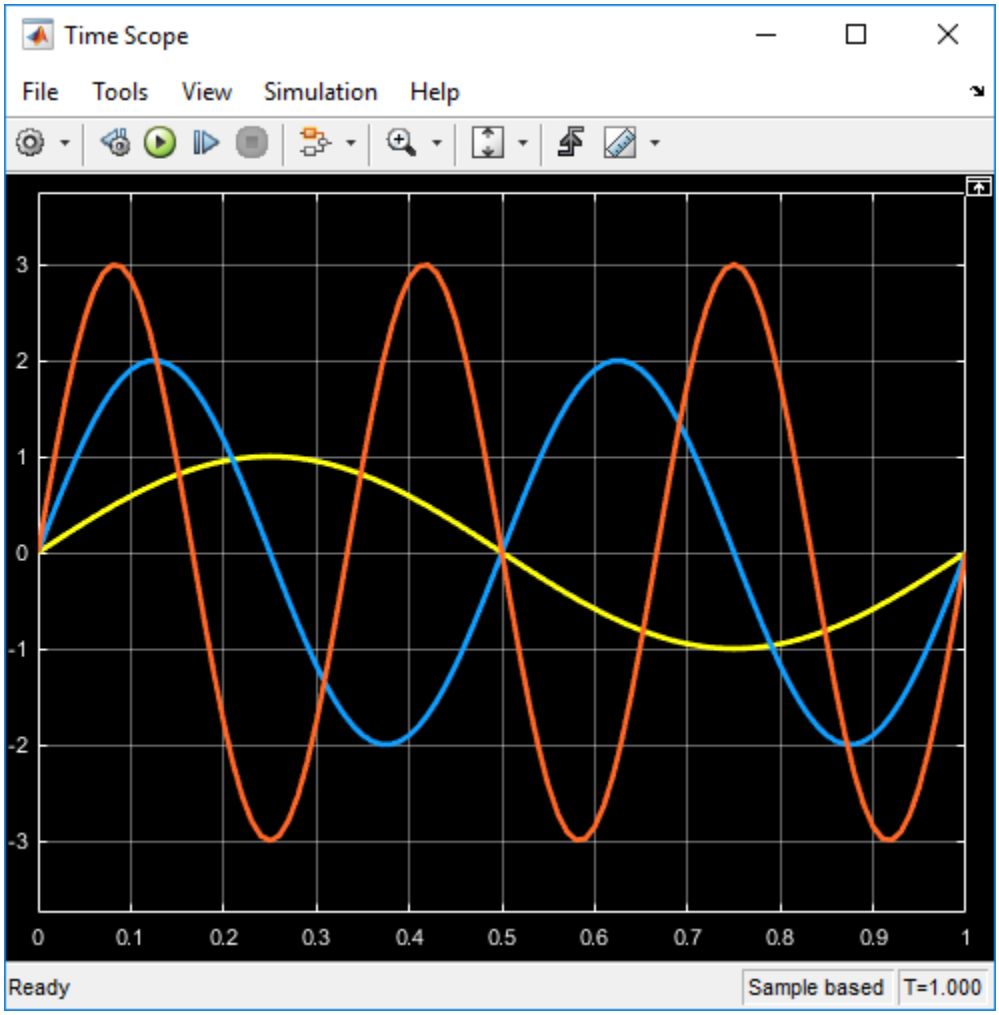
Scope Blocks and the Scope Viewer

Scope blocks, Floating Scope blocks, and the Scope Viewer visualize connected signals in a similar way as a bench-top oscilloscope. The scope blocks and Scope Viewer use the same interface to visualize and analyze connected signals. The interface is similar to other domain-specific tools, such as the Spectrum Analyzer and Array Plot blocks. Scope blocks are easy to add and connect in your model, and you do not need to log data in order to view signal data.



Scope blocks, Floating Scope blocks, and the Scope Viewer support:

- Viewing signals during simulation, including rapid accelerator simulations.
- Simple connection and accessibility for lightweight debugging.
- Signal visualization without logging.
- Optional signal logging.
- Starting simulations from the visualization interface using playback controls.
- Oscilloscope measurements, including cursors and triggers. Additional measurements are available when you have a Simscape or DSP System Toolbox license.
- Configurable plots and display.
- Configurable triggers to capture events.



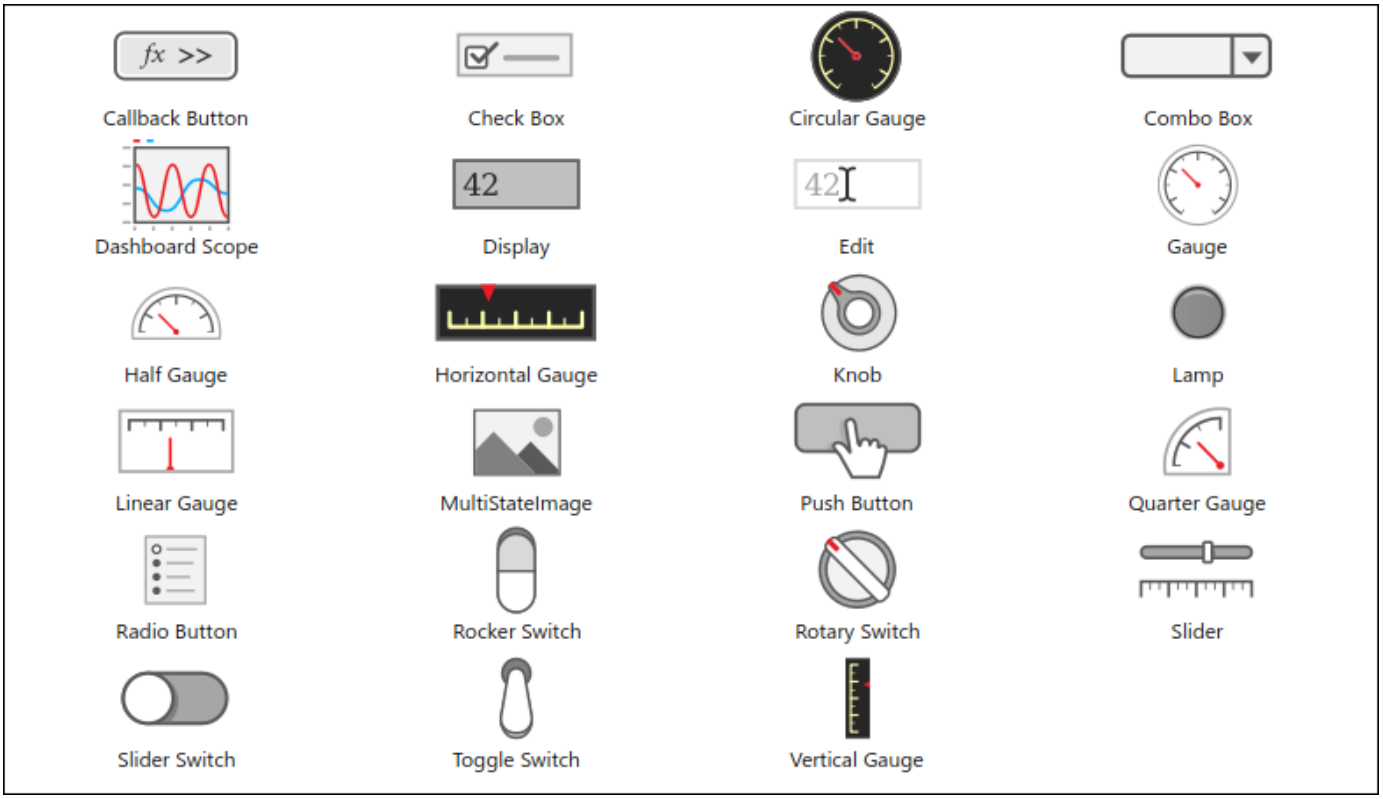
Use the Scope block, the Floating Scope block, and the Scope Viewer for prototyping and lightweight debugging. For example, in the early stages of model development, you can connect a Scope block to a signal to quickly verify component behavior. By adding triggers, you can tune a specific peak or other artifact in a signal. A Scope block may or may not be permanent in your model.

When you need to view data for signals throughout a model hierarchy, consider using the Simulation Data Inspector, especially if you already log the signals you want to view.

For more information, see “Scope Blocks and Scope Viewer Overview” on page 28-6.

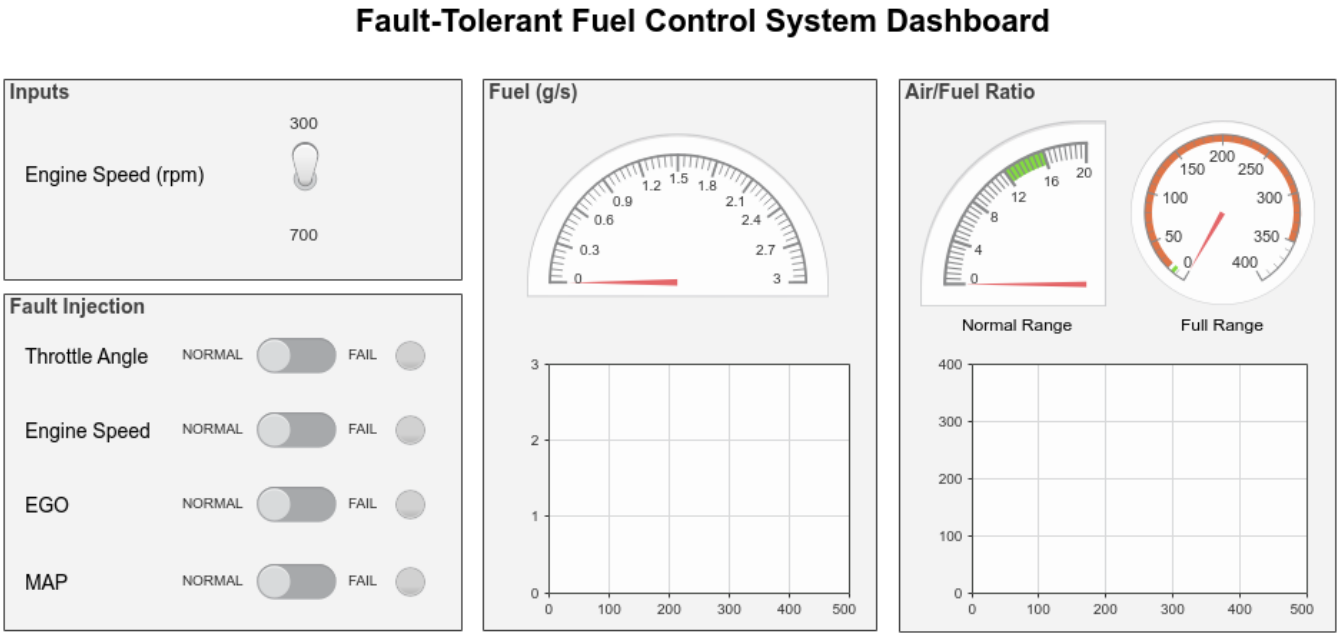
Dashboard Blocks

The Dashboard library includes control blocks you can use to tune variables and parameters in your model and visualization blocks you can use to view signal data. Use Dashboard blocks to view instantaneous signal data and build an interactive interface for your model.



You can use Dashboard block controls to modify the values of variables and tunable parameters in your model during simulation. Visualization Dashboard blocks are updated continuously throughout simulation. When you need to debug a model that uses Dashboard blocks, consider using simulation pacing or simulation stepping to slow the simulation so you can view the instantaneous signal values. You can also promote Dashboard blocks to one or more panels so the interactive dashboard can follow you throughout a model hierarchy as you design and debug your model. For more information, see:

- “Simulation Pacing” on page 2-17
- “Step Through a Simulation” on page 2-12
- “Interactively Design and Debug Models Using Panels” on page 29-169



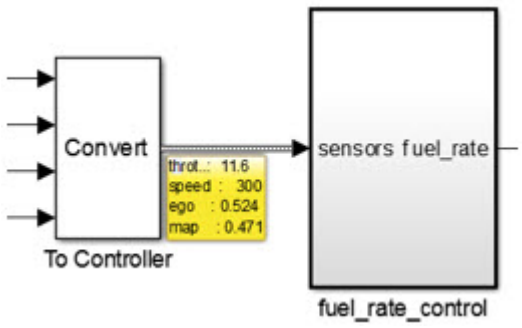
The Dashboard Scope block provides a simple view of a signal over time. You can zoom, pan, and add cursors to a Dashboard Scope block through the context menu. For more debugging and analysis capabilities, use a Scope block or log data to the Simulation Data Inspector.

For more information about using Dashboard blocks, see “Tune and Visualize Your Model with Dashboard Blocks” on page 29-164.

Port Value Displays

You can enable port value displays to view a signal value at a given time point. Port value displays can supplement existing visualizations while prototyping and debugging models. For example, you can view signal values on port value displays as you step through a simulation while visualizing the temporal behavior using a Scope block. You can also use port value displays on their own to support debugging without adding any blocks to your model or logging data.

To display a signal value, right-click the signal and select **Show Value Label of Selected Port**.



When inspecting signal data by displaying the value on the port, consider using simulation pacing or stepping through the simulation. For more information, see “Simulation Pacing” on page 2-17 and “Step Through a Simulation” on page 2-12.

For more information, see “Display Port Values for Debugging” on page 36-16.

Custom MATLAB Visualizations

When you log simulation data to the workspace or a file, you can write MATLAB code and scripts to create custom visualizations. You can log signals, outputs, and states in a model, and you can also enable data logging for signals connected to Scope Viewers, Floating Scope blocks and Scope blocks. For more information about logging data in your model, see “Export Simulation Data” on page 72-2.

Note When you post-process data using MATLAB code and scripts, you do not need to write custom MATLAB code to visualize the data. You can import the processed data into the Simulation Data Inspector for visualization alongside the logged data.

The visualization code you write can depend on the format of the logged data. By default, logging uses the `Dataset` format and produces a single simulation output. Logged data is returned to the workspace as a `Simulink.SimulationOutput` object that contains a `Simulink.SimulationData.Dataset` object for each type of logging used in the simulation. Each `Dataset` object contains `Simulink.SimulationData.Signal` objects that store the data for an individual signal as a `timeseries` object. For more information, see “Data Format for Logged Simulation Data” on page 72-7.

When you log a signal, you can visualize the signal during simulation using custom MATLAB code. For more information, see “Access Data in a MATLAB Function During Simulation” on page 29-80.

See Also

Floating Scope | Scope | Scope Viewer | **Simulation Data Inspector**

Related Examples

- “Export Simulation Data” on page 72-2
- “View Data in the Simulation Data Inspector” on page 29-2
- “Scope Blocks and Scope Viewer Overview” on page 28-6
- “Tune and Visualize Your Model with Dashboard Blocks” on page 29-164

Linearizing Models

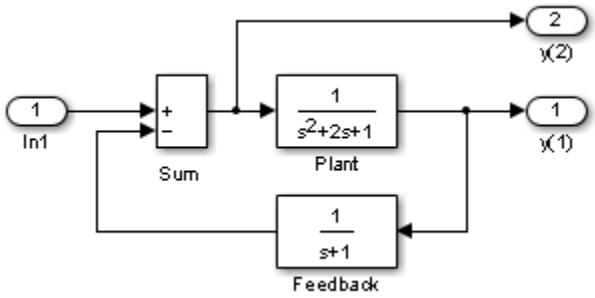
In this section...

About Linearizing Models

The Simulink product provides the `linmod`, `linmod2`, and `dlinmod` functions to extract linear models in the form of the state-space matrices A, B, C, and D. State-space matrices describe the linear input-output relationship as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du,$$

where x, u, and y are state, input, and output vectors, respectively. For example, the following model is called `lmod`.



To extract the linear model of this system, enter this command.

```
[A,B,C,D] = linmod('lmod')

A =
    -2    -1    -1
     1     0     0
     0     1    -1
B =
     1
     0
     0
C =
     0     1     0
     0     0    -1
D =
     0
     1
```

Inputs and outputs must be defined using Inport and Outport blocks from the Ports & Subsystems library. Source and sink blocks do not act as inputs and outputs. Inport blocks can be used in conjunction with source blocks, using a Sum block. Once the data is in the state-space form or

converted to an LTI object, you can apply functions in the Control System Toolbox product for further analysis:

- Conversion to an LTI object

  ```
  sys = ss(A,B,C,D);
  ```

- Bode phase and magnitude frequency plot

  ```
  bode(A,B,C,D) or bode(sys)
  ```

- Linearized time response

  ```
  step(A,B,C,D) or step(sys)
  impulse(A,B,C,D) or impulse(sys)
  lsim(A,B,C,D,u,t) or lsim(sys,u,t)
  ```

You can use other functions in the Control System Toolbox and the Robust Control Toolbox™ products for linear control system design.

When the model is nonlinear, an operating point can be chosen at which to extract the linearized model. Extra arguments to `linmod` specify the operating point.

```
[A,B,C,D] = linmod('sys', x, u)
```

For discrete systems or mixed continuous and discrete systems, use the function `dlinmod` for linearization. This function has the same calling syntax as `linmod` except that the second right-hand argument must contain a sample time at which to perform the linearization.

Linearization with Referenced Models

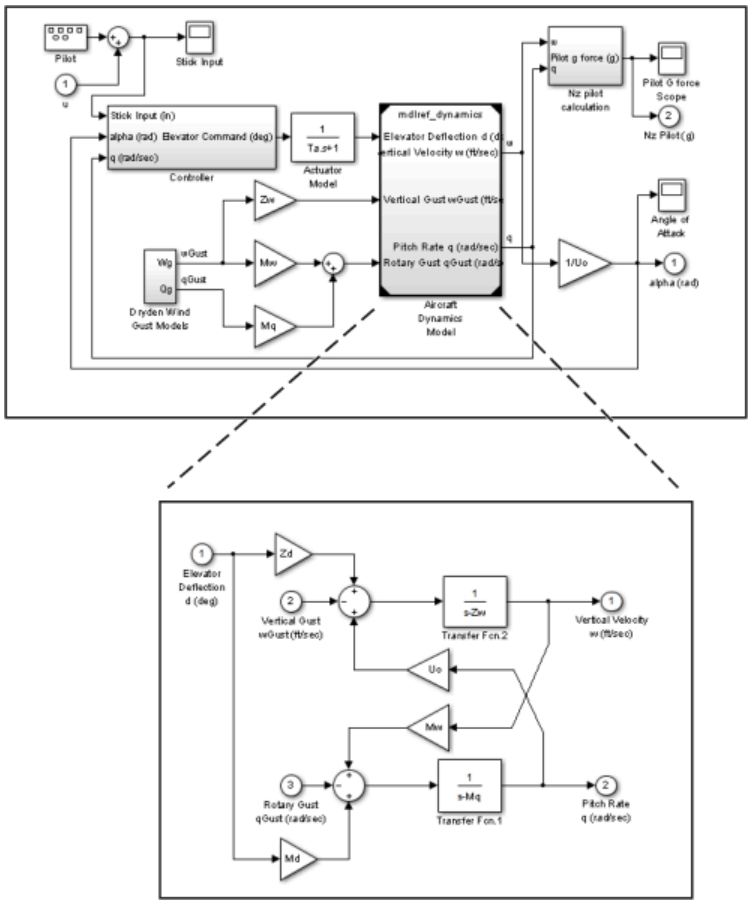
You can use `linmod` to extract a linear model from a Simulink environment that contains Model blocks.

Note In Normal mode, the `linmod` command applies the block-by-block linearization algorithm on blocks inside the referenced model. If the Model block is in Accelerator mode, the `linmod` command uses numerical perturbation to linearize the referenced model. Due to limitations on linearizing multirate Model blocks in Accelerator mode, you should use Normal mode simulation for all models referenced by Model blocks when linearizing with referenced models. For an explanation of the block-by-block linearization algorithm, see the Simulink Control Design documentation.

For example, open the referenced model `mdlref_dynamics` and top model `mdlref_f14`.

```
open_system([docroot '/toolbox/simulink/ug/examples/analysis/mdlref_dynamics']);
open_system([docroot '/toolbox/simulink/ug/examples/analysis/mdlref_f14'])
```

The Aircraft Dynamics Model block refers to the model `mdlref_dynamics`.



To linearize the `mdlref_f14` model, call the `linmod` command on the top `mdlref_f14` model as follows.

```
[A,B,C,D] = linmod('mdlref_f14')
```

The resulting state-space model corresponds to the complete `f14` model, including the referenced model.

You can call `linmod` with a state and input operating point for models that contain Model blocks. When using operating points, the state vector `x` refers to the total state vector for the top model and any referenced models. You must enter the state vector using the structure format. To get the complete state vector, call

```
x = Simulink.BlockDiagram.getInitialState(topModelName)
```

Linearization Using the 'v5' Algorithm

Calling the `linmod` command with the `'v5'` argument invokes the perturbation algorithm created prior to MATLAB software version 5.3. This algorithm also allows you to specify the perturbation values used to perform the perturbation of all the states and inputs of the model.

```
[A,B,C,D]=linmod('sys',x,u,para,xpert,upert,'v5')
```

Using `linmod` with the `'v5'` option to linearize a model that contains Derivative or Transport Delay blocks can be troublesome. Before linearizing, replace these blocks with specially designed blocks

that avoid the problems. These blocks are in the Simulink Extras library in the Linearization sublibrary.

You access the Extras library by opening the Blocksets & Toolboxes icon:

- For the Derivative block, use the Switched derivative for linearization.

When using a Derivative block, 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, it is better implemented (although this is not always possible) with a single Transfer Fcn block of the form

$$\frac{s}{s+a}.$$

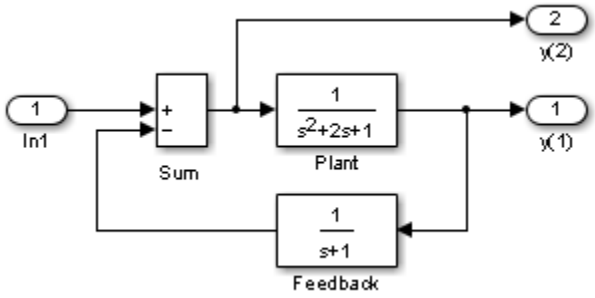
In this example, the blocks on the left of this figure can be replaced by the block on the right.

See Also

`dlinmod` | `linmod`

Finding Steady-State Points

The Simulink `trim` function uses a model to determine steady-state points of a dynamic system that satisfy input, output, and state conditions that you specify. Consider, for example, this model, called `ex_lmod`.



You can use the `trim` function to find the values of the input and the states that set both outputs to 1. First, make initial guesses for the state variables (`x`) and input values (`u`), then set the desired value for the output (`y`).

```
x = [0; 0; 0];
u = 0;
y = [1; 1];
```

Use index variables to indicate which variables are fixed and which can vary.

```
ix = [];      % Don't fix any of the states
iu = [];      % Don't fix the input
iy = [1;2];   % Fix both output 1 and output 2
```

Invoking `trim` returns the solution. Your results might differ because of roundoff error.

```
[x,u,y,dx] = trim('lmod',x,u,y,ix,iu,iy)

x =
   0.0000
   1.0000
   1.0000
u =
   2
y =
   1.0000
   1.0000
dx =
   1.0e-015 *
    -0.2220
    -0.0227
     0.3331
```

Note that there might be no solution to equilibrium point problems. If that is the case, `trim` returns a solution that minimizes the maximum deviation from the desired result after first trying to set the derivatives to zero. For a description of the `trim` syntax, see `trim`.

See Also

`trim`

Improving Simulation Performance and Accuracy

How Optimization Techniques Improve Performance and Accuracy

The design of a model and choice of configuration parameters can affect simulation performance and accuracy. Solvers handle most model simulations accurately and efficiently with default parameter values. However, some models yield better results when you adjust solver parameters. Information about the behavior of a model can help you improve simulation performance, particularly when you provide this information to the solver. Use optimization techniques to better understand the behavior of your model and modify the model settings to improve performance and accuracy.

To optimize your model and achieve faster simulation automatically using Performance Advisor, see “Automated Performance Optimization”.

To learn more about accelerator modes for faster simulation, see “Acceleration”.

See Also

Related Examples

- “Speed Up Simulation” on page 31-3
- “Check and Improve Simulation Accuracy” on page 31-11
- “How Profiler Captures Performance Data” on page 31-5

More About

- “Modeling Techniques That Improve Performance” on page 31-13

Speed Up Simulation

Several factors can slow simulation. Check your model for some of these conditions.

- Your model includes an Interpreted MATLAB Function block. When a model includes an Interpreted MATLAB Function block, the MATLAB execution engine is called at each time step, drastically slowing down the simulation. Use the Math Function block whenever possible.
- Your model includes a MATLAB file S-function. MATLAB file S-functions also call the MATLAB execution engine at each time step. Consider converting the S-function either to a subsystem or to a C-MEX file S-function.
- Your model includes a Memory block. Using a Memory block causes the variable-order solvers (`ode15s` and `ode113`) to reset back to order 1 at each time step.
- The maximum step size is too small. If you changed the maximum step size, try running the simulation again with the default value (`auto`).
- Your accuracy requirements are too high. The default relative tolerance (0.1% accuracy) is usually sufficient. For models with states that go to zero, if the absolute tolerance parameter is too small, the simulation can take too many steps around the near-zero state values. See the discussion of this error in “Maximum order”“Maximum order” in the online documentation.
- The time scale is too long. Reduce the time interval.
- The problem is stiff, but you are using a nonstiff solver. Try using `ode15s`. For more information, see “Stiffness of System” on page 31-15.
- The model uses sample times that are not multiples of each other. Mixing sample times that are not multiples of each other causes the solver to take small enough steps to ensure sample time hits for all sample times.
- The model contains an algebraic loop. The solutions to algebraic loops are iteratively computed at every time step. Therefore, they severely degrade performance. For more information, see “Algebraic Loop Concepts” on page 3-27.
- Your model feeds a Random Number block into an Integrator block. For continuous systems, use the Band-Limited White Noise block in the Sources library.
- Your model contains a scope viewer that displays too many data points. Try adjusting the viewer property settings that can affect performance. For more information, see Scope Viewer.
- You need to simulate your model iteratively. You change tunable parameters between iterations but do not make structural changes to the model. Every iteration requires the model to compile again, thus increasing overall simulation time. Use fast restart to perform iterative simulations. In this workflow, the model compiles only once and iterative simulations are tied to a single compile phase. See “How Fast Restart Improves Iterative Simulations” on page 81-2 for more information.

See Also

Related Examples

- “How Profiler Captures Performance Data” on page 31-5
- “Check and Improve Simulation Accuracy” on page 31-11

More About

- “How Optimization Techniques Improve Performance and Accuracy” on page 31-2
- “Modeling Techniques That Improve Performance” on page 31-13
- “How Fast Restart Improves Iterative Simulations” on page 81-2

How Profiler Captures Performance Data

In this section...

How Profiler Works

Profiler captures performance data while your model simulates. It identifies the parts of your model that require the most time to simulate. Use the profiling information to decide where to focus your model optimization efforts.

Note You cannot use Profiler in Rapid Accelerator mode.

Simulink stores performance data in the *simulation profile report*. The data shows the time spent executing each function in your model.

The basis for Profiler is an execution model that this pseudocode summarizes.

```
Sim()
    ModelInitialize().
    ModelExecute()
        for t = tStart to tEnd
        Output()
        Update()
        Integrate()
            Compute states from derivs by repeatedly calling:
                MinorOutput()
                MinorDeriv()
            Locate any zero crossings by repeatedly calling:
                MinorOutput()
                MinorZeroCrossings()
        EndIntegrate
        Set time t = tNew.
    EndModelExecute
    ModelTerminate
EndSim
```

According to this conceptual model, Simulink runs a model by invoking the following functions zero, one, or many times, depending on the function and the model.

Function	Purpose	Level
`sim`	Simulate the model. This top-level function invokes the other functions required to simulate the model. The time spent in this function is the total time required to simulate the model.	System
`ModelInitialize`	Set up the model for simulation.	System

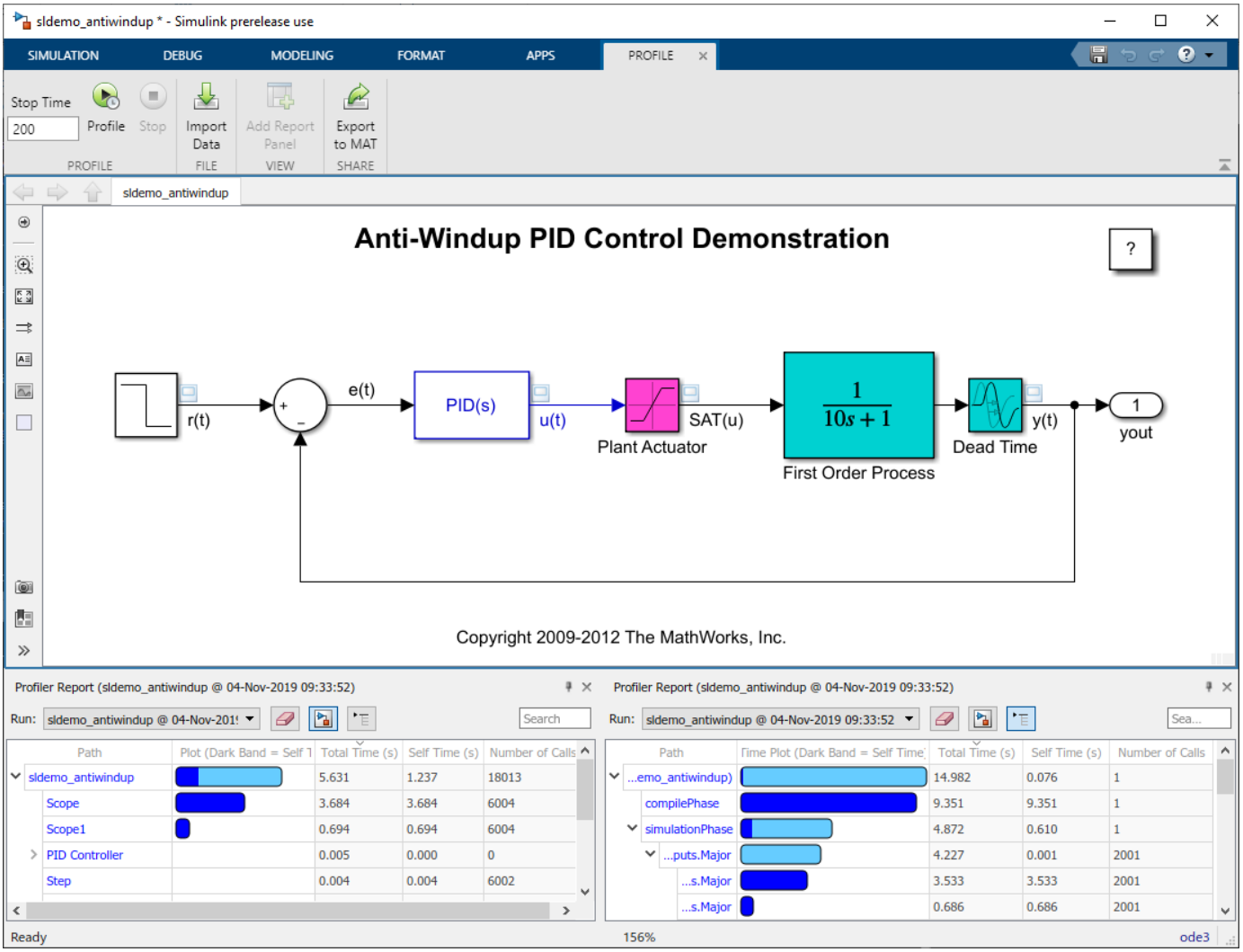
Function	Purpose	Level
`ModelExecute`	Execute the model by invoking the output, update, integrate, etc., functions for each block at each time step from the start to the end of simulation.	System
`Output`	Compute the outputs of a block at the current time step.	Block
`Update`	Update the state of a block at the current time step.	Block
`Integrate`	Compute the continuous states of a block by integrating the state derivatives at the current time step.	Block
`MinorOutput`	Compute block output at a minor time step.	Block
`MinorDeriv`	Compute the state derivatives of a block at a minor time step.	Block
`MinorZeroCrossings`	Compute zero-crossing values of a block at a minor time step.	Block
`ModelTerminate`	Free memory and perform any other end-of-simulation cleanup.	System
`Nonvirtual Subsystem`	Compute the output of a nonvirtual subsystem at the current time step by invoking the output, update, integrate, etc., functions for each block that it contains. The time spent in this function is the time required to execute the nonvirtual subsystem.	Block

Profiler measures the time required to execute each invocation of these functions. After the model simulates, Profiler generates a report that describes the amount of simulation time spent on each function.

Start Profiler

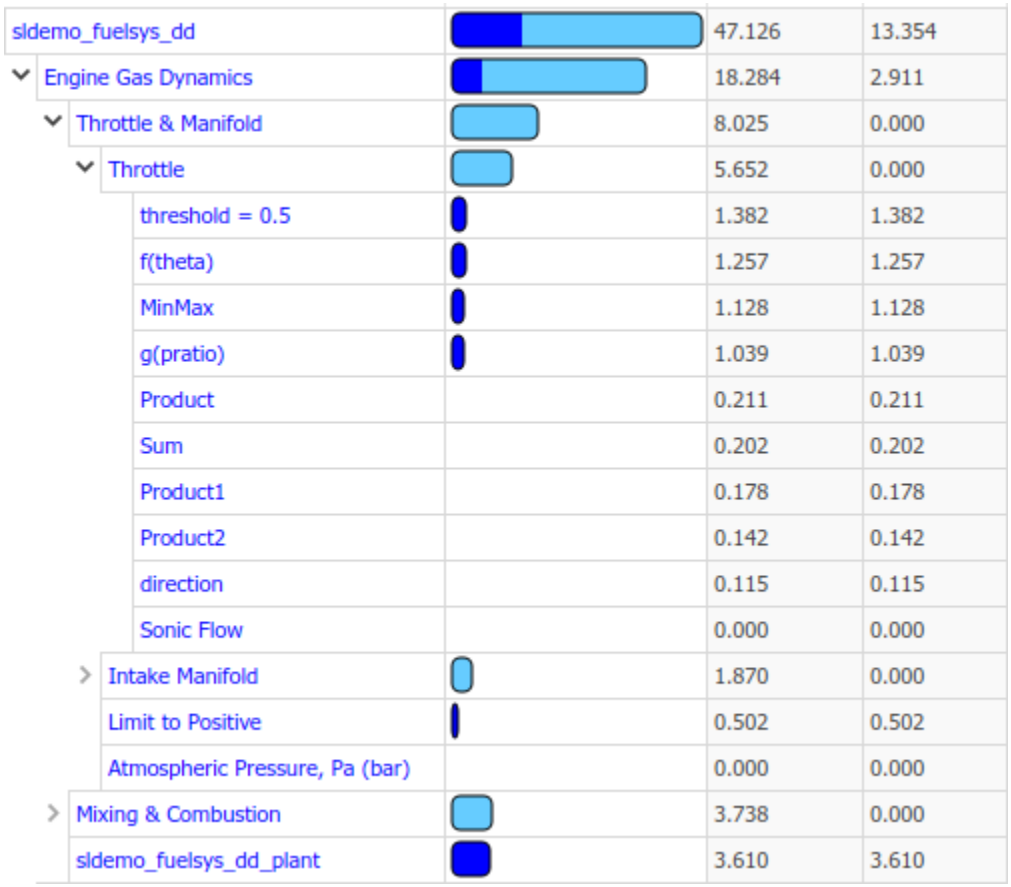
1. Open the model.
2. On the **Debug** tab, select **Performance Advisor > Simulink Profiler**.
3. Simulate the model.

When simulation is complete, Simulink generates and displays the simulation profile for the model in a panel in the Simulink editor.



Block Hierarchy View

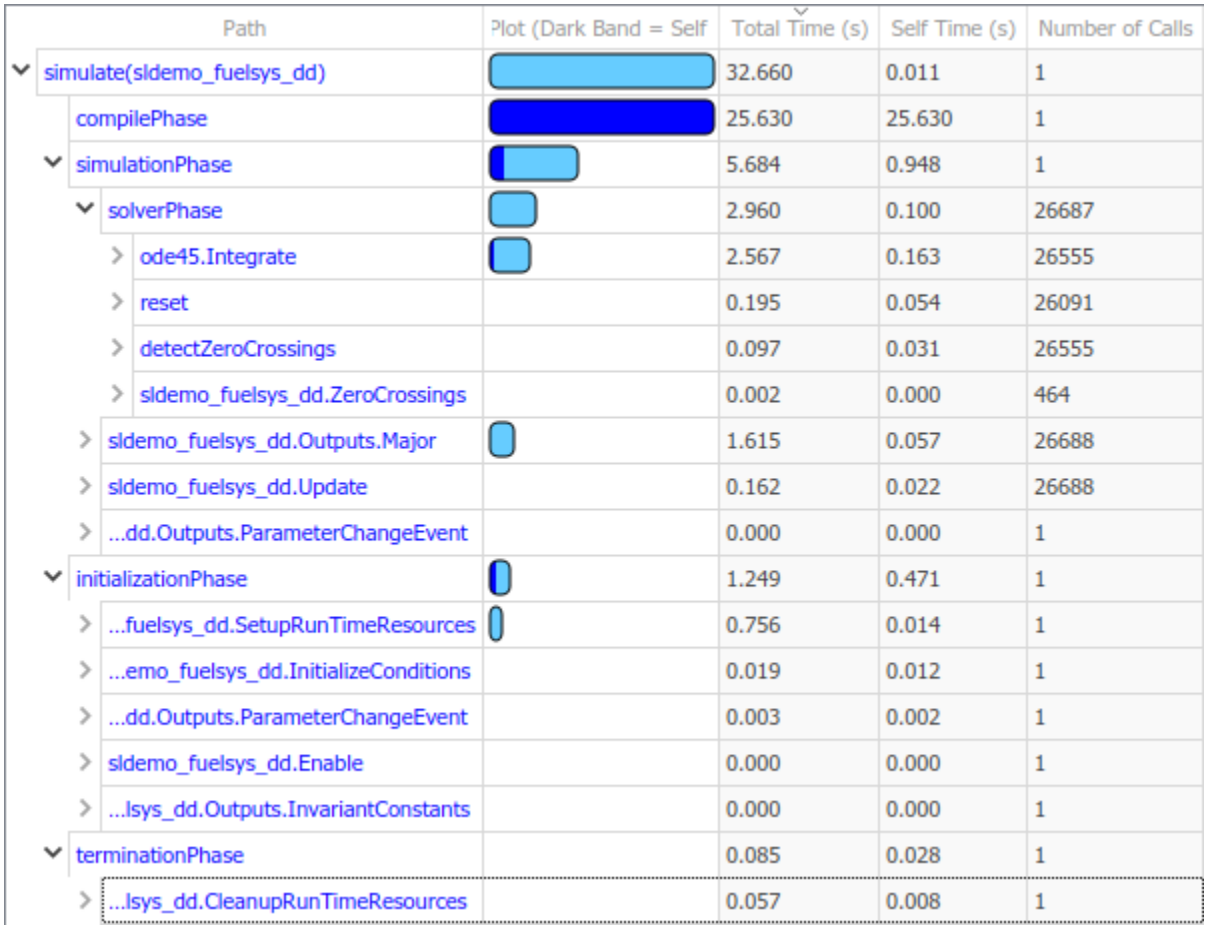
The block hierarchy view of the report presents the profiling information of the model in a nested tree form. The first row of the table—which is the top of the tree—corresponds to the entire model. Subsequent rows at the first level of nesting correspond to blocks at the root level of the model.



sldemo_fuelsys_dd	47.126	13.354
Engine Gas Dynamics	18.284	2.911
Throttle & Manifold	8.025	0.000
Throttle	5.652	0.000
threshold = 0.5	1.382	1.382
f(theta)	1.257	1.257
MinMax	1.128	1.128
g(pratio)	1.039	1.039
Product	0.211	0.211
Sum	0.202	0.202
Product1	0.178	0.178
Product2	0.142	0.142
direction	0.115	0.115
Sonic Flow	0.000	0.000
Intake Manifold	1.870	0.000
Limit to Positive	0.502	0.502
Atmospheric Pressure, Pa (bar)	0.000	0.000
Mixing & Combustion	3.738	0.000
sldemo_fuelsys_dd_plant	3.610	3.610

Execution Stack view

The execution stack view breaks down the profiling report by simulation phase of the model. The main simulation phases of a model are compilation, initialization, simulation, and termination. For a more detailed explanation of simulation phases, see “Simulation Phases in Dynamic Systems” on page 3-2.



Path	Plot (Dark Band = Self	Total Time (s)	Self Time (s)	Number of Calls
simulate(sldemo_fuelsys_dd)		32.660	0.011	1
compilePhase		25.630	25.630	1
simulationPhase		5.684	0.948	1
solverPhase		2.960	0.100	26687
ode45.Integrate		2.567	0.163	26555
reset		0.195	0.054	26091
detectZeroCrossings		0.097	0.031	26555
sldemo_fuelsys_dd.ZeroCrossings		0.002	0.000	464
sldemo_fuelsys_dd.Outputs.Major		1.615	0.057	26688
sldemo_fuelsys_dd.Update		0.162	0.022	26688
...dd.Outputs.ParameterChangeEvent		0.000	0.000	1
initializationPhase		1.249	0.471	1
...fuelsys_dd.SetupRunTimeResources		0.756	0.014	1
...emo_fuelsys_dd.InitializeConditions		0.019	0.012	1
...dd.Outputs.ParameterChangeEvent		0.003	0.002	1
sldemo_fuelsys_dd.Enable		0.000	0.000	1
...lsys_dd.Outputs.InvariantConstants		0.000	0.000	1
terminationPhase		0.085	0.028	1
...lsys_dd.CleanupRunTimeResources		0.057	0.008	1

Save Profiler Results

You can save the Profiler report to a `mat` file. At a later time, you can import and review the report by either loading the saved `mat` file to the current workspace or import it into an existing profiler session.

To save a profiling report, select **Export to MAT** in the **Share** section of the **Profile** tab and optionally specify a name for the `mat` file.

To view a saved report in an open profiling session in the Simulink editor, select **Import from File** in the **Profile** tab.

To view a saved report in the command line, load the `mat` file containing the report. The profiling information is loaded into the current workspace as a `Simulink.profiler.Data` object. For more information, see `Simulink.profiler.Data`.

See Also

Related Examples

- “Speed Up Simulation” on page 31-3

- “Check and Improve Simulation Accuracy” on page 31-11

More About

- “How Optimization Techniques Improve Performance and Accuracy” on page 31-2
- “Modeling Techniques That Improve Performance” on page 31-13

Check and Improve Simulation Accuracy

Check Simulation Accuracy

1. Simulate the model over a reasonable time span.
2. Reduce either the relative tolerance to 1e-4 (the default is 1e-3) or the absolute tolerance.
3. Simulate the model again.
4. Compare the results from both simulations.

If the results are not significantly different, the solution has converged.

If the simulation misses significant behavior at the start, reduce the initial step size to ensure that the simulation does not step over that behavior.

Unstable Simulation Results

When simulation results become unstable over time,

- The system can be unstable.
- If you are using the `ode15s` solver, try restricting the maximum order to 2 (the maximum order for which the solver is A-stable). You can also try using the `ode23s` solver.

Inaccurate Simulation Results

If simulation results are not accurate:

- For a model that has states whose values approach zero, if the absolute tolerance parameter is too large, the simulation takes too few steps around areas of near-zero state values. Reduce this parameter value in the **Solver pane** of model configuration parameters or adjust it for individual states in the function block parameters of the Integrator block.
- If reducing the absolute tolerances does not improve simulation accuracy enough, reduce the size of the relative tolerance parameter. This change reduces the acceptable error and forces smaller step sizes and more steps.

Certain modeling constructs can also produce unexpected or inaccurate simulation results.

- A Source block that inherits sample time can produce different simulation results if, for example, the sample times of the downstream blocks are modified (see “How Propagation Affects Inherited Sample Times” on page 7-30).
- A Derivative block found in an algebraic loop can result in a loss in solver accuracy.

See Also

Related Examples

- “Speed Up Simulation” on page 31-3
- “How Profiler Captures Performance Data” on page 31-5

More About

- “How Optimization Techniques Improve Performance and Accuracy” on page 31-2
- “Modeling Techniques That Improve Performance” on page 31-13

Modeling Techniques That Improve Performance

In this section...

Accelerate the Initialization Phase

Speed up a simulation by accelerating the initialization phase, using these techniques.

Simplify Graphics Using Mask Editor

Complex graphics and large images take a long time to load and render. Masked blocks that contain such images can make your model less responsive. Where possible, remove complex drawings and images from masked blocks.

If you want to keep the image, replace it with a smaller, low-resolution version. Use mask editor and edit the icon drawing commands to keep the image that is loaded by the call to `image()`.

For more information on mask editor, see “Mask Editor Overview”.

Consolidate Function Calls

When you open or update a model, Simulink runs the mask initialization code. If your model contains complicated mask initialization commands that contain many calls to `set_param`, consolidate consecutive calls into a single call with multiple argument pairs. Consolidating the calls can reduce the overhead associated with these function calls.

To learn more, see “Mask Callback Code” on page 39-14.

Load Data Using MAT-file

If you use MATLAB scripts to load and initialize data, you can improve performance by loading MAT-files instead. The data in a MAT-file is in binary and can be more difficult to work with than a script. However, the load operation typically initializes data more quickly than the equivalent MATLAB script.

For more information, see “MAT-Files for Signal Data” on page 70-4.

Reduce Model Interactivity

In general, the more interactive a model is, the longer it takes to simulate. Use these techniques to reduce the interactivity of your model.

Disable Debugging Diagnostics

Some enabled diagnostic features can slow simulations considerably. Consider disabling them in the model configuration parameters **Diagnostics** pane.

Note Running **Array bounds exceeded** and **Solver data inconsistency** can slow down model runtime performance. For more information, see “Array bounds exceeded” and “Solver data inconsistency”.

Disable MATLAB Debugging

After verifying that your MATLAB code works correctly, disable these checks in the model configuration parameters **Simulation Target** pane.

- **Enable debugging/animation**
- **Detect wrap on overflow (with debugging)**
- **Echo expressions without semicolons**

For more information, see “Model Configuration Parameters: Simulation Target”.

Use BLAS Library Support

If your simulation involves low-level MATLAB matrix operations, use the Basic Linear Algebra Subprograms (BLAS) libraries to make use of highly optimized external linear algebra routines.

Disable Stateflow Animations

By default, Stateflow charts highlight the current active states in a model and animate the state transitions that take place as the model simulates. This feature is useful for debugging, but it slows the simulation.

To accelerate simulations, either close all Stateflow charts or disable the animation. Similarly, consider disabling animation or reducing scene fidelity when you use:

- Simulink 3D Animation
- Simscape Multibody visualization
- FlightGear
- Any other 3D animation package

To learn more, see “Speed Up Simulation” (Stateflow).

Adjust Scope Viewer Properties

If your model contains a scope viewer that displays a high rate of logging and you cannot remove the scope, adjust the viewer properties to trade off fidelity for rendering speed.

However, when you use decimation to reduce the number of plotted data points, you can miss short transients and other phenomena that you can see with more data points. To have more precise control over enabling visualizations, place viewers in enabled subsystems.

For more information, see Scope Viewer.

Reduce Model Complexity

Use these techniques to improve simulation performance by simplifying a model without sacrificing fidelity.

Replace Subsystems with Lower-Fidelity Alternatives

Replace a complex subsystem with one of these alternatives:

- A linear or nonlinear dynamic model that was created from measured input-output data using the System Identification Toolbox™.
- A high-fidelity, nonlinear statistical model that was created using the Model-Based Calibration Toolbox™.
- A linear model that was created using Simulink Control Design.
- A lookup table. For more information, see A lookup table.

You can maintain both representations of the subsystem in a library and use variant subsystems to manage them. Depending on the model, you can make this replacement without affecting the overall result. For more information, see “Optimize Generated Code for Lookup Table Blocks” on page 38-34.

Reduce Number of Blocks

When you reduce the number of blocks in your model, fewer blocks require updates during simulations and simulation is faster.

- Vectorization is one way to reduce your block count. For example, if you have several parallel signals that undergo a similar set of computations, try to combine them into a vector using a Mux block and perform a single computation.
- You can also enable the **Block Reduction** parameter in the **Configuration Parameters** dialog.

Use Frame-Based Processing

In frame-based processing, Simulink processes samples in batches instead of one at a time. If a model includes an analog-to-digital converter, for example, you can collect output samples in a buffer. Process the buffer in a single operation, such as a fast Fourier transform. Processing data in chunks this way reduces the number of times that the simulation needs to invoke blocks in your model.

In general, the scheduling overhead decreases as frame size increases. However, larger frames consume more memory, and memory limitations can adversely affect the performance of complex models. Experiment with different frame sizes to find one that maximizes the performance benefit of frame-based processing without causing memory issues.

Choose and Configure a Solver

Simulink provides a comprehensive library of solvers, including fixed-step and variable-step solvers, to handle stiff and nonstiff systems. Each solver determines the time of the next simulation step. A solver applies a numerical method to solve ordinary differential equations that represent the model.

The solver you choose and the solver options you select can affect simulation speed. Select and configure a solver that helps boost the performance of your model using these criteria. For more information, see “Choose a Solver” on page 25-5.

Stiffness of System

A stiff system has continuous dynamics that vary slowly and quickly. Implicit solvers are particularly useful for stiff problems. Explicit solvers are better suited for nonstiff systems. Using an explicit

solver to solve a stiff system can lead to incorrect results. If a nonstiff solver uses a very small step size to solve a model, this is a sign that your system is stiff.

Model Step Size and Dynamics

When you are deciding between using a variable-step or fixed-step solver, keep in mind the step size and dynamics of your model. Select a solver that uses time steps to capture only the dynamics that are important to you. Choose a solver that performs only the calculations needed to work out the next time step.

You use fixed-step solvers when the step size is less than or equal to the fundamental sample time of the model. With a variable-step solver, the step size can vary because variable-step solvers dynamically adjust the step size. As a result, the step size for some time steps is larger than the fundamental sample time, reducing the number of steps required to complete the simulation. In general, simulations with variable-step solvers run faster than those that run with fixed-step solvers.

Choose a fixed-step solver when the fundamental sample time of your model is equal to one of the sample rates. Choose a variable-step solver when the fundamental sample time of your model is less than the fastest sample rate. You can also use variable-step solvers to capture continuous dynamics.

Decrease Solver Order

When you decrease the solver order, you reduce the number of calculations that Simulink performs to determine state outputs, which improves simulation speed. However, the results become less accurate as the solver order decreases. Choose the lowest solver order that produces results with acceptable accuracy.

Increase Solver Step Size or Error Tolerance

Increasing the solver step size or error tolerance usually increases simulation speed at the expense of accuracy. Make these changes with care because they can cause Simulink to miss potentially important dynamics during simulations.

Disable Zero-Crossing Detection

Variable-step solvers dynamically adjust the step size, increasing it when a variable changes slowly and decreasing it when a variable changes rapidly. This behavior causes the solver to take many small steps near a discontinuity because this is when a variable changes rapidly. Accuracy improves, but often at the expense of long simulation times.

To avoid the small time steps and long simulations associated with these situations, Simulink uses zero-crossing detection to locate such discontinuities accurately. For systems that exhibit frequent fluctuations between modes of operation—a phenomenon known as chattering—this zero-crossing detection can have the opposite effect and thus slow down simulations. In these situations, you can disable zero-crossing detection to improve performance.

You can enable or disable zero-crossing detection for specific blocks in a model. To improve performance, consider disabling zero-crossing detection for blocks that do not affect the accuracy of the simulation.

For more information, see “Zero-Crossing Detection” on page 3-10.

Save the Simulation State

In the classic workflow, a Simulink model simulates repeatedly for different inputs, boundary conditions, and operating conditions. In many situations, these simulations share a common startup phase in which the model transitions from the initial state to another state. For example, you can bring an electric motor up to speed before you test various control sequences.

Using `SimState`, you can save the simulation state at the end of the startup phase and then restore it for use as the initial state for future simulations. This technique does not improve simulation speed, but it can reduce total simulation time for consecutive runs because the startup phase needs to be simulated only once.

See “Save and Restore Simulation Operating Point” on page 25-41 for more information.

See Also

Related Examples

- “Speed Up Simulation” on page 31-3
- “How Profiler Captures Performance Data” on page 31-5

More About

- “How Optimization Techniques Improve Performance and Accuracy” on page 31-2
- “Check and Improve Simulation Accuracy” on page 31-11

Use Performance Advisor to Improve Simulation Efficiency

Use Performance Advisor to check for conditions and configuration settings that can cause inefficient simulation performance. Performance Advisor analyzes a model and produces a report with suboptimal conditions or settings that it finds. It suggests better model configuration settings where appropriate, and provides mechanisms for fixing issues automatically or manually.

See Also

Related Examples

- “Performance Advisor Workflow” on page 32-2
- “Improve Simulation Performance Using Performance Advisor” on page 32-2

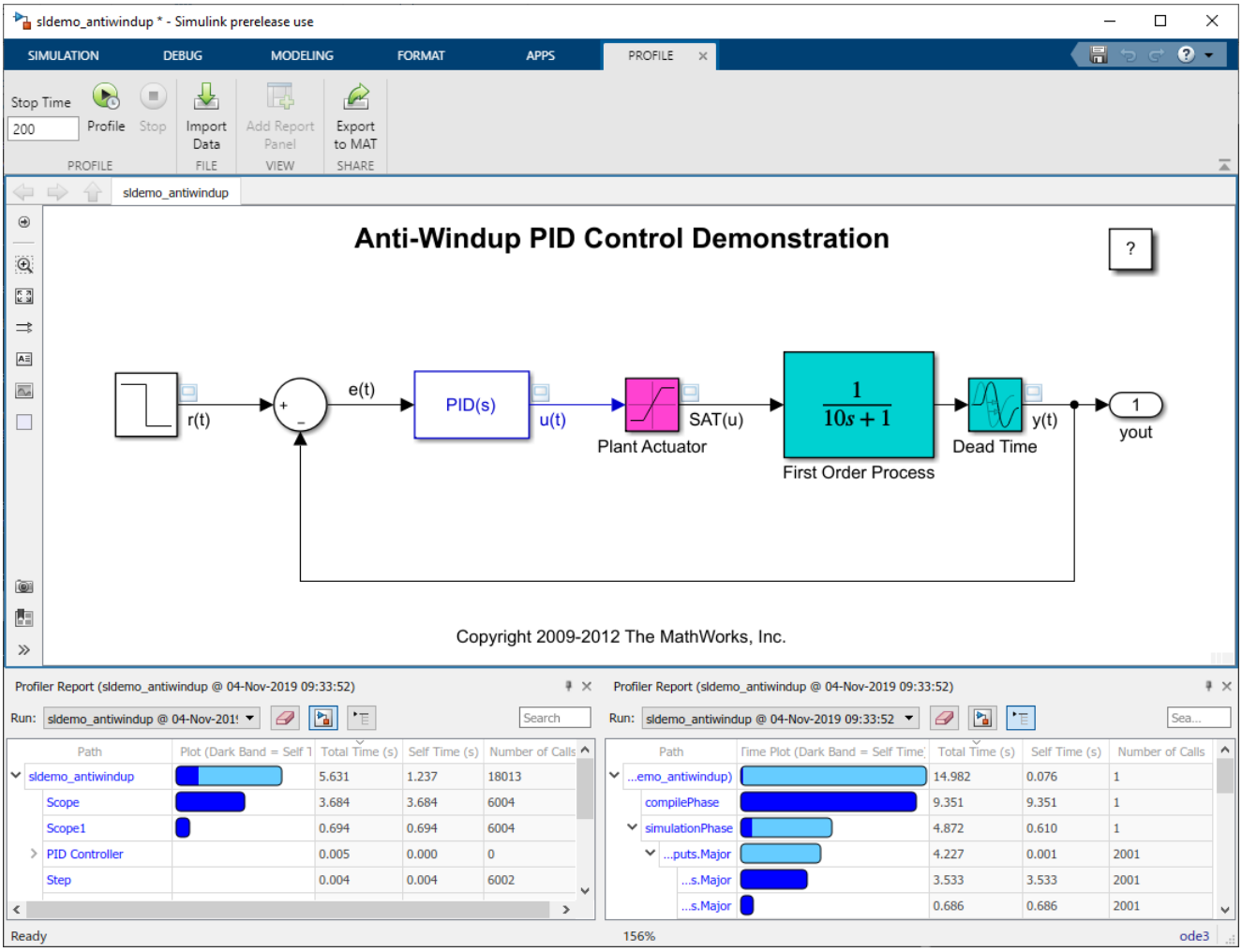
More About

- “Improve Simulation Performance Using Performance Advisor” on page 32-2
- “How Optimization Techniques Improve Performance and Accuracy” on page 31-2

Understanding Total Time and Self Time in Profiler Reports

The Simulink Profiler displays the performance of the components of your model and their simulation phases. Each row of the Profiler Report pane presents the following information:

- **Path**
- **Time Plot (Dark Band = Self Time)**
- **Total Time (s)**
- **Self Time (s)**
- **Number of Calls**



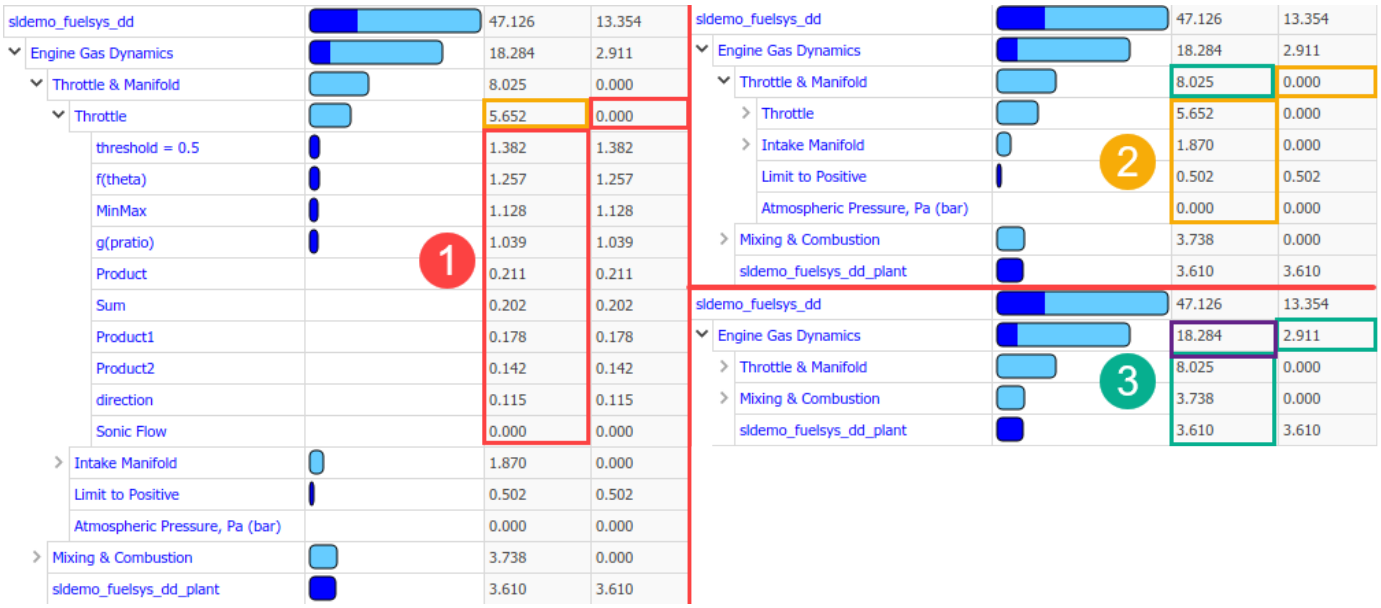
An indicator of the model's performance is the number of times a block is called in conjunction with the time taken by each call. For a model component with any level of hierarchy, it is useful to know how much of its total execution time is taken up by its constituent blocks vis-a-vis the time taken by its own execution, or self-time.

Use the **Total Time(s)** information to identify subsystems or referenced models that are expensive to run within the current model. As subsystems and model references typically correspond to high-level

conceptual entities such as physical models and algorithms, you can identify which component of your model is slowing down overall performance.

The graphic below shows part of the Profiler Report pane for the `sldemo_fuelsys_dd` model.

1. To begin, recursively expand the hierarchy of the Engine Gas Dynamics subsystem. For the Throttle subsystem, tally the total time of its constituents (from threshold=0.5 to Sonic Flow), highlighted in red. Observe that the sum of their execution times, along with the self time for Throttle block, is the total time of the Throttle block, shown in yellow.
2. Collapse the Throttle row of the hierarchy. Repeat the previous step for the contents of the Throttle & Manifold subsystem–which is highlighted in yellow–and then the Engine Gas Dynamics subsystem.



See Also

“How Profiler Captures Performance Data” on page 31-5

Performance Advisor

Improve Simulation Performance Using Performance Advisor

In this section...

Whatever the level of complexity of your model, you can make systematic changes that improve simulation performance. Performance Advisor checks for configuration settings that slow down your model simulations. It produces a report that lists the suboptimal conditions or settings it finds and suggests better configuration settings where appropriate.

You can use the Performance Advisor to fix some of these suboptimal conditions automatically or you can fix them manually.

Note Use Performance Advisor on top models. Performance Advisor does not traverse referenced models or library links.

To learn about faster simulation using acceleration modes, see “Acceleration”.

Performance Advisor Workflow

When the performance of a model is slower than expected, use Performance Advisor to help identify and resolve bottlenecks.

1. Prepare your model.
2. Create a baseline to compare measurements against.
3. Select the checks you want to run.
4. Run Performance Advisor with the selected checks and see recommended changes.
5. Make changes to the model. You can either:
   - Automatically apply changes.
   - Generate advice, and review and apply changes manually.
6. After applying changes, Performance Advisor performs a final validation of the model to see how performance has improved.
   - If the performance improves, the selected checks were successful. The performance check is complete.
   - If the performance is worse than the baseline, Performance Advisor reinstates the previous settings of the model.
7. Save your model.

Caution Performance Advisor does not automatically save your model after it makes changes. When you are satisfied with the changes to the model from Performance Advisor, save the model.

Prepare Your Model

Before running checks using Performance Advisor, complete the following steps:

Start Performance Advisor

To get started with Performance Advisor:

1. Make a backup of the model.
2. Verify that the model can simulate without error.
3. Close all applications, including web browsers. Leave only the MATLAB Command Window, the model you want to analyze, and Performance Advisor running.

   Running other applications can hinder the performance of model simulation and the ability of Performance Advisor to measure accurately.
4. Open Performance Advisor. In the Simulink Editor, on the **Debug** tab, click **Performance Advisor**.

Enable Data Logging for the Model

Make sure the model configuration parameters are set to enable data logging.

1. In the model, on the **Modeling** tab, click **Model Settings**.
2. In the Configuration Parameters dialog box, in the left pane, select **Data Import/Export**.
3. Set up signal logging. The model must log at least one signal for Performance Advisor to work. For example, select the **States** or **Output** check box.
4. Click **Configure Signals to Log** and select the signals to log.

   Note Select only the signals you are most interested in. Minimizing the number of signals to log can help performance. Selecting too many signals can cause Performance Advisor to run for a longer time.
5. Click **OK** in the Configuration Parameters dialog box.
6. Run the model once to make sure that the simulation is successful.

Select How Performance Advisor Applies Advice

Choose from these options to apply advice to the model:

- **Use check parameters**. Select the checks for which you want Performance Advisor to automatically apply advice. You can review the remaining checks and apply advice manually.

- **Automatically for all checks**. Performance Advisor automatically applies advice to all selected checks.
- **Generate advice only**. Review advice for each check and apply changes manually.

Select Validation Actions for the Advice

For the checks you want to run, validate an improvement in simulation time and accuracy by comparing against a baseline measurement. Each validation action requires the model to simulate. Use these validation options as global settings for the checks you select:

- **Use check parameters**. From the checks you want to run, select the ones for which you want to validate an improvement in performance. Specify validation action for fixes using individual settings for these checks.
- **For all checks**. Performance Advisor automatically validates an improvement in performance for the checks you select.
- **Do not validate**. Performance Advisor does not validate an improvement in performance. Instead, you can validate manually. When you select this option and also specify for Performance Advisor to apply advice automatically, a warning appears before Performance Advisor applies changes without validation.

These global settings for validation apply to all checks in the left pane except the Final Validation check. The Final Validation check validates the overall performance improvement in a model after you have applied changes. In case you do not want to validate changes resulting from other check results, you can run the Final Validation check to validate model changes for simulation time and accuracy.

Specify Runtime for Performance Advisor

You can specify a **Time Out** value in minutes if you want to limit the runtime duration of Performance Advisor. Use this option when running Performance Advisor on models with long simulation times.

If Performance Advisor times out before completing the checks you specify, in the left pane you can see the checks that failed.

Create a Performance Advisor Baseline Measurement

A baseline measurement is a set of simulation measurements that Performance Advisor measures check results against.

Note Before creating a baseline measurement, set the model configuration parameters to enable data logging. For more information, see “Enable Data Logging for the Model” on page 32-12.

1. In the model, select **Performance Tools > Performance Advisor** to start Performance Advisor.
2. In the left pane, in the Baseline folder, select **Create Baseline**.
3. In the right pane, under **Input Parameters**, enter a value in the **Stop Time** field for the baseline.

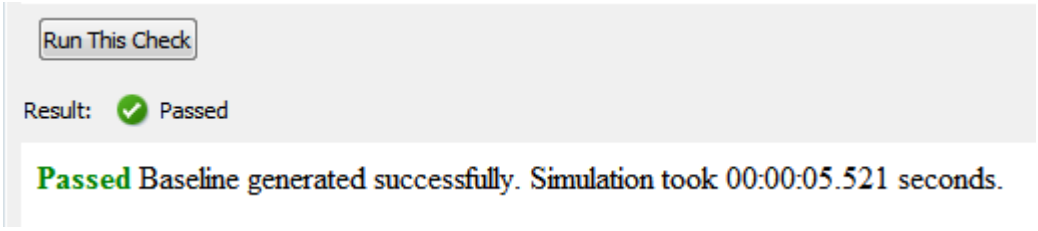
   When you enter a Stop Time value in Performance Advisor, this overrides the value set in the model. A large stop time can create a simulation that runs longer.

If you do not enter a value, Performance Advisor uses values from the model. Performance Advisor uses values from the model that are less than 10. Performance Advisor rounds values from the model larger than 10 to 10.

4 Select the **Check to view baseline signals and set their tolerances** check box to start the Signal Data Inspector after Performance Advisor runs a check. Using the Signal Data Inspector, you can compare signals and adjust tolerance levels.

5 Click **Run This Check**.

When a baseline has been created, a message like the following appears under **Analysis**:



After the baseline has been created, you can run Performance Advisor checks.

Run Performance Advisor Checks

1 After you have created a baseline measurement, select checks to run.

- In the left pane of Performance Advisor, expand a folder, such as **Simulation** or **Simulation Targets**, to display checks related to specific tasks.
- In the folder, select the checks you want to run using the check boxes.

Tip If you are unsure of which checks apply, you can select and run all checks. After you see the results, clear the checks you are not interested in.

2 Specify input parameters for selected checks. Use one of these methods:

- Apply global settings to all checks to take action, validate simulation time and validate simulation accuracy.
- Alternatively, for each check, in the right pane, specify input parameters.

Input Parameter	Description
Take action based on advice	`automatically` — Allow Performance Advisor to automatically make the change for you. `manually` — Review the change first. Then manually make the change or accept Performance Advisor recommendations.
Validate and revert changes if time of simulation increases	Select this check box to have Performance Advisor rerun the simulation and verify that the change made based on the advice improves simulation time. If the change does not improve simulation time, Performance Advisor reverts the changes.

Input Parameter	Description
Validate and revert changes if degree of accuracy is greater than tolerance	Select this check box to have Performance Advisor rerun the simulation and verify that, after the change, the model results are still within tolerance. If the result is outside tolerance, Performance Advisor reverts the changes.
Quick estimation of model build time	Select this check box to have Performance Advisor use the number of blocks of a referenced model to estimate model build time.

3. To run a single check, click **Run This Check** from the settings for the check. Performance Advisor displays the results in the right pane.

   You can also select multiple checks from the left pane and click **Run Selected Checks** from the right pane. Select **Show report after run** to display the results of the checks after they run.

4. To limit the run time of Performance Advisor, specify a **Time Out** value in minutes. Use this option for models with long simulation times. The default setting for this option is 60 minutes.

   Note The **Time Out** setting does not apply to a Quick Scan diagnosis.

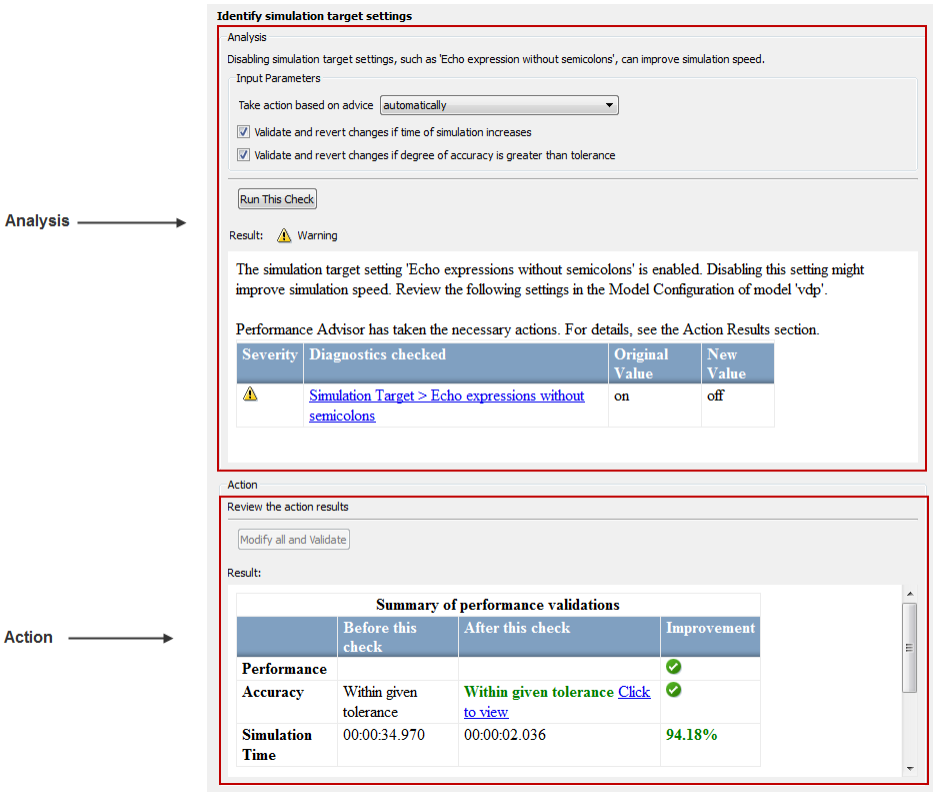
Performance Advisor also generates an HTML report of the current check results and actions in a file with a name in the form `model_name\report_#.html`

To view this report in a separate window, click the **Report** link in the right pane.

Note If you rename a system, you must restart Performance Advisor to check that system.

View and Respond to Results

After you run checks with Performance Advisor, the right pane shows the results:



To view the results of a check, in the left pane, select the check you ran. The right pane updates with the results of the check. This pane has two sections.

The **Analysis** section contains:

- Information about the check
- Option to run the simulation
- Settings to take action based on advice from Performance Advisor
- Result of the check (Passed, Failed or Warning)

The **Action** section contains:

- A setting to manually accept all recommendations for the check
- Summary of actions taken based on the recommendations for the check

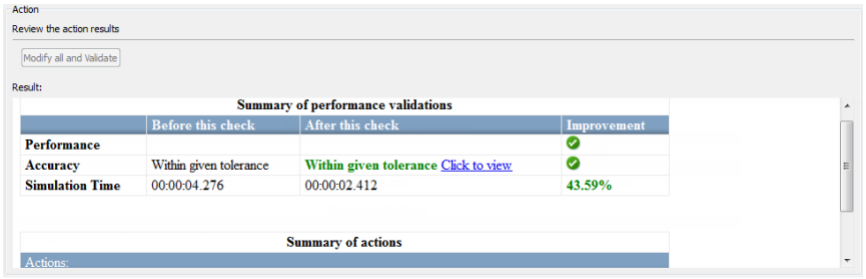
Respond to Results

Use the **Take action based on advice** parameter in the **Analysis** section to select how to respond to changes that Performance Advisor suggests.

Value	Response
`automatically`	• Performance Advisor makes the change for you. • You can evaluate the changes using the links in the summary table. • The **Modify All** button in the **Action** section is grayed out since Performance Advisor has already made all recommended changes for you.
`manually`	• Performance Advisor does not make the change for you. • The links in the summary table show recommendations. • Use the **Modify All** button in the **Action** section to implement all recommendations after reviewing them. Depending on how you set your validation input parameters before you ran the check, the button label can change to **Modify All and Validate**.

Review Actions

The **Action** section contains a summary of the actions that Performance Advisor took based on the **Input Parameters** setting. If the tool also performed validation actions, this section lists the results in a summary table. If performance has not improved, Performance Advisor reports that it reinstated the model to the settings it had before the check ran.



Severity	Description
	The actions succeeded. The table lists the percentage of improvement.
	The actions failed. For example, if Performance Advisor cannot make a recommended change, it flags it as failed. It also flags a check as failed if performance did not improve and reinstates the model to the settings it had before the check ran.

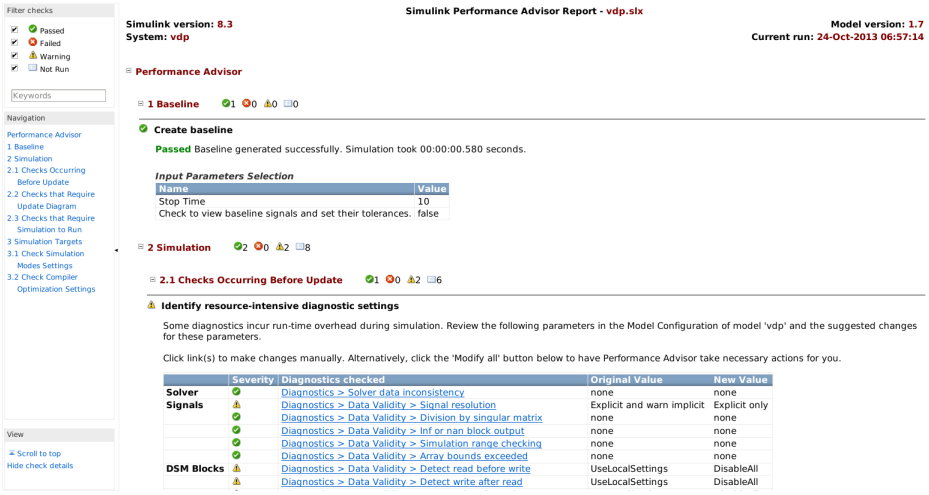
Caution Performance Advisor does not automatically save your model after it makes changes. When you are satisfied with the changes to the model from Performance Advisor, save the model.

View and Save Performance Advisor Reports

When Performance Advisor runs checks, it generates HTML reports of the results. To view a report, select a folder in the left pane and click the link in the **Report** box in the right pane.

As you run checks, Performance Advisor updates the reports with the latest information for each check in the folder. Time stamps indicate when checks ran.

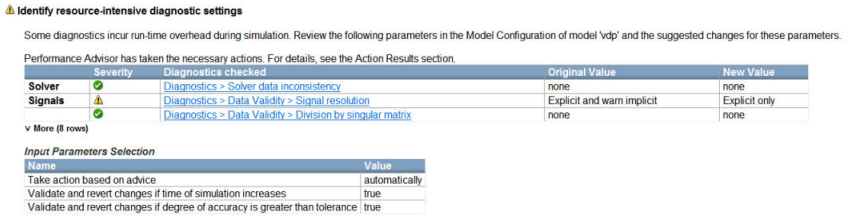
In the pane for global settings, when you select **Show report after run**, Performance Advisor displays a consolidated set of check results in the report.



You can perform these actions using the Performance Advisor report:

- Use the check boxes under **Filter checks** to view only the checks with the status that you are interested in viewing. For example, to see only the checks that failed or gave warnings, clear the **Passed** and **Not Run** check boxes.
- Perform a keyword search using the search box under **Filter checks**.
- Use the tree of checks under **Navigation** to jump to the category of checks or a specific check result that interests you.
- Expand and collapse content in the right pane of the report to view or hide check results.

Some checks have input parameters that you specify in the right pane of Performance Advisor. For example, **Identify resource intensive diagnostic settings** has several input parameters. When you run checks that have input parameters, Performance Advisor displays the values of the input parameters in the report.



Save Performance Advisor Reports

You can archive a Performance Advisor report by saving it to a new location. Performance Advisor does not update the saved version of a report when you run checks again. Archived reports serve as good points of comparison when you run checks again.

1. In the left pane of the Performance Advisor window, select the folder of checks for the report you want to save.
2. In the **Report** box, click **Save As**.

3 In the **Save As** dialog box, navigate to where you want to save the report, and click **Save**. Performance Advisor saves the report to the new location.

See Also

Related Examples

- “Improve vdp Model Performance” on page 32-12
- “Perform a Quick Scan Diagnosis” on page 32-11

More About

- “Simulink Performance Advisor Checks”
- “Acceleration”

External Websites

- Improving Simulation Performance in Simulink

Perform a Quick Scan Diagnosis

Quick Scan is a fast method to diagnose settings in a model and deliver an approximate analysis of performance. A model can compile and simulate several times during a normal run in Performance Advisor. Quick Scan enables you to review performance issues without compiling or changing the model or validating any fixes. In models with long compile times, use Quick Scan to get a rapid analysis of possible improvements.

When you perform a Quick Scan diagnosis, Performance Advisor

- Does not perform a baseline measurement.
- Does not automatically apply advice to the model.
- Does not validate any changes you make to the model.
- Does not time out if the Quick Scan diagnosis takes longer than the **Time Out** duration you specify.

Run Quick Scan on a Model

1. Select checks to run.

   Tip If you are unsure of which checks apply, you can select and run all checks. After you see the results, clear the checks you are not interested in.

   - In the left pane of Performance Advisor, expand a folder, such as **Simulation** or **Simulation Targets**, to display checks related to specific tasks.
   - In the folder, select the checks you want to run using the check boxes.
2. Select the **Show report after run** check box to display the results of the checks after they run.
3. Click **Quick Scan** on the right pane.

Checks in Quick Scan Mode

- “Identify resource-intensive diagnostic settings”
- “Check optimization settings”
- “Identify inefficient lookup table blocks”
- “Check MATLAB System block simulation mode”
- “Identify Interpreted MATLAB Function blocks”
- “Identify simulation target settings”
- “Check model reference rebuild setting”

See Also

Related Examples

- “Improve Simulation Performance Using Performance Advisor” on page 32-2
- “Improve vdp Model Performance” on page 32-12

Improve vdp Model Performance

In this section...

This example shows you how to run Performance Advisor on the `vdp` model, review advice, and make changes to improve performance.

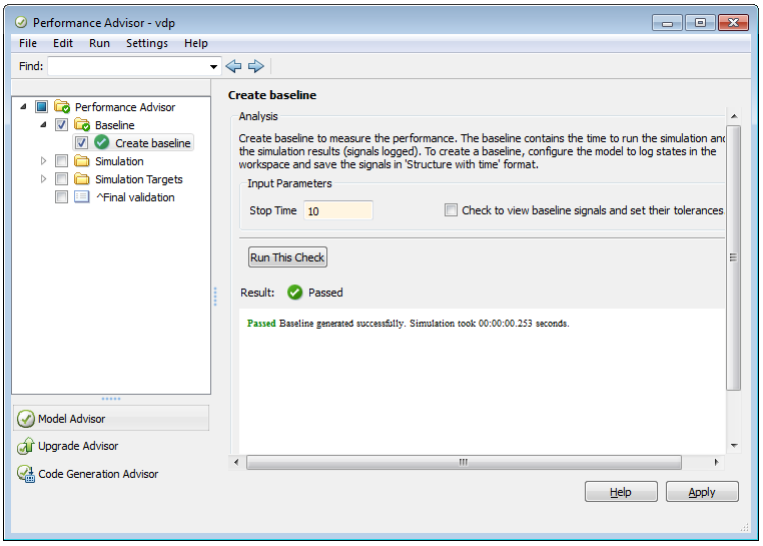
Enable Data Logging for the Model

1. In the `vdp` model, on the **Modeling** tab, click **Model Settings**.
2. In the Configuration Parameters dialog box, click **Data Import/Export** in the left pane.
3. Set up signal logging. The model must log at least one signal for Performance Advisor to work. For example, select the **States** or **Output** check box.
4. Click **Configure Signals to Log** .
5. To select signals to log, select a signal in `vdp`. Right click and select **Properties**.
6. In the Signal Properties dialog box, check the **Log signal data** option and click **OK**.
7. Click **OK** in the Configuration Parameters dialog box.
8. Run the model once to make sure that the simulation is successful.

Create Baseline

1. Open Performance Advisor. In the `vdp` model, on the **Debug** tab, click **Performance Advisor**.
2. In the right pane, under **Set Up**, select a global setting for **Take Action**. To automatically apply advice to the model, select `automatically for all checks`.
3. Select global settings to validate any improvements in simulation time and accuracy after applying advice. To select the default setting for validation, for **Validate simulation time** and **Validate simulation accuracy**, select `use check parameters`.

   Note To validate any improvements automatically, change the global settings to `For all checks`. However, this can increase simulation time as validating all checks requires more simulation runs.
4. Select **Show report after run**. This opens an HTML report of check results.
5. In the left pane, select the **Create baseline** check. Clear the other checks.
6. In the **Create baseline** pane, set **Stop Time** to `10`. Click **Apply**.
7. Click **Run This Check**. The right pane updates to show that the baseline was created successfully.



Select Checks and Run

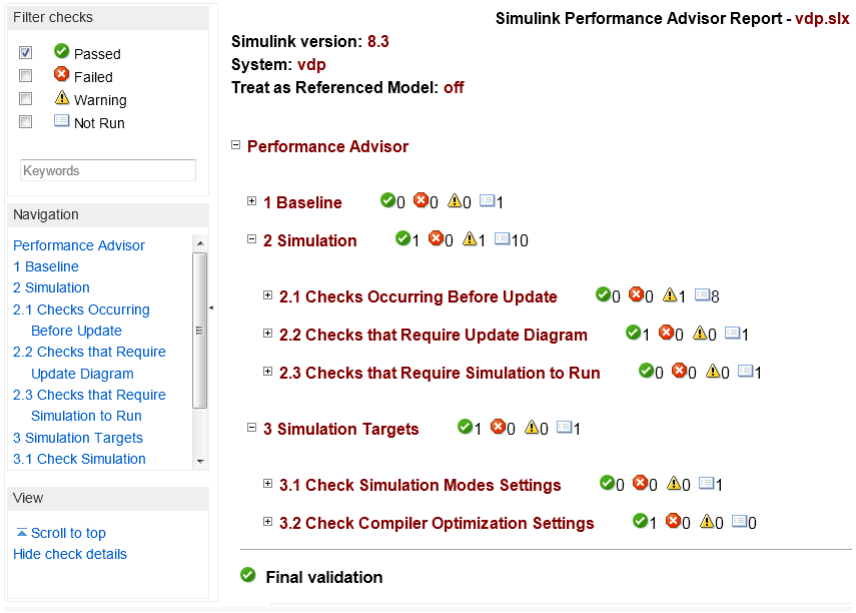
Note The global input parameters to take action and validate improvement apply to all the checks you select.

1. In the left pane, clear the baseline check. Select these checks:
   - In **Simulation > Checks Occurring Before Update**, select **Identify resource-intensive diagnostic settings**.
   - In **Simulation > Checks that Require Update Diagram**, select **Check model reference parallel build**.
   - In **Simulation Targets > Check Compiler Optimization Settings**, select **Select compiler optimizations on or off**.
   - Select **Final validation**.
2. For every check that you have selected in the left pane, select options in the right pane to validate any improvements in simulation time and accuracy. Note that **Take Action based on advice** is set to `automatically`, a result of **Take Action** being set to `automatically for all checks`.
3. Select a value for **Time Out** if you want to limit the runtime duration of Performance Advisor.
4. Click **Run Selected Checks**.

Performance Advisor runs the checks you selected and opens an HTML report with check results.

Review Results

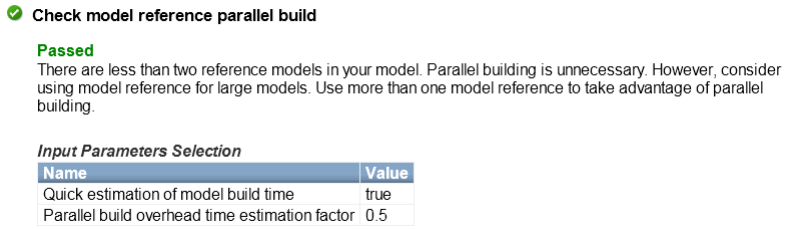
1. In the HTML report, filter the results to see only the checks that passed.



All of the selected checks passed successfully.

2 Navigate to the results for a particular check, for example **Check model reference parallel build**. Use the navigation tree in the left pane or scroll to the results for this check in the right pane.

3 Performance Advisor gives you information about this check, advice for performance improvement, as well as a list of related model configuration parameters.

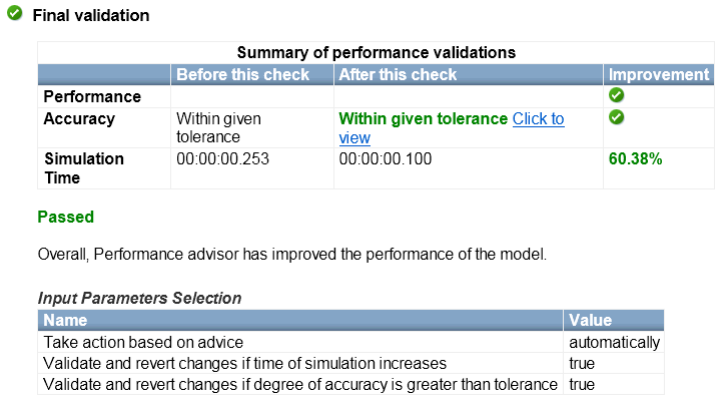


4 Filter the results to display warnings. See results for the **Identify resource-intensive diagnostic settings** check.

Performance Advisor identified diagnostic settings that incur runtime overhead during simulation. It modified values for some of these diagnostics. A table in the report shows the diagnostics checked and whether Performance Advisor suggested a change to the value.

If the performance of the model improved, the HTML report gives you information about this improvement. If the performance has deteriorated, Performance Advisor discards all changes and reinstates the original settings in the model.

5 See details for the **Final Validation** check.



Final validation

Summary of performance validations

	Before this check	After this check	Improvement
Performance			✓
Accuracy	Within given tolerance	Within given tolerance Click to view	✓
Simulation Time	00:00:00.253	00:00:00.100	60.38%

Passed

Overall, Performance advisor has improved the performance of the model.

Input Parameters Selection

Name	Value
Take action based on advice	automatically
Validate and revert changes if time of simulation increases	true
Validate and revert changes if degree of accuracy is greater than tolerance	true

This check validates the overall performance improvement in the model. The check results show changes in simulation time and accuracy, depending on whether performance improved or degraded.

Apply Advice and Validate Manually

Generate advice for a check, apply it, and validate any improvements manually.

1. In the left pane, click **Performance Advisor**. Select these settings and click **Apply**:
   - Set **Take Action** to `generate advice only`.
   - Set **Validate simulation time** to `use check parameters`.
   - Set **Validate simulation accuracy** to `use check parameters`.
2. For every check that you have selected in the left pane, select options in the right pane to validate any improvements in simulation time and accuracy. Note that **Take Action based on advice** is set to `manually`, a result of **Take Action** being set to `generate advice only`.
3. Select **Performance Advisor** in the left pane. Click **Run Selected Checks** in the Performance Advisor pane.

   If the performance of the model has improved, the **Final Validation** check results show the overall performance improvement.
4. In the results for **Identify resource-intensive diagnostic settings**, Performance Advisor suggests new values for the diagnostics it checked. Review these results to accept or reject the values it suggests.

   Alternatively, click **Modify all and Validate** to accept all changes and validate any improvement in performance.

See Also

Related Examples

- “Improve Simulation Performance Using Performance Advisor” on page 32-2
- “Perform a Quick Scan Diagnosis” on page 32-11

Solver Profiler

Examine Model Dynamics Using Solver Profiler

When model simulation slows down or stops responding, a close examination of the dynamics of the model can help you identify the factors affecting the simulation.

Understanding solver behavior enables you to interpret how the model simulates and what causes the solver to take small steps.

The Solver Profiler analyzes a model for patterns that affect its simulation. The Solver Profiler presents graphical and statistical information about the simulation, solver settings, events, and errors. You can use this data to identify locations in the model that caused simulation bottlenecks.

In addition, there are multiple factors that can limit the simulation speed. The Solver Profiler logs and reports all the major events that occur when simulating a model:

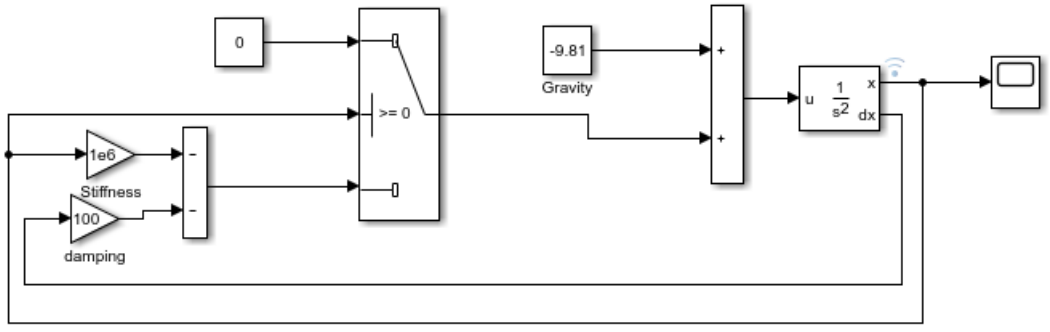
- Zero-crossing events
- Solver exception events
- Solver reset events
- Jacobian computation events

Note In order to accurately profile the solver performance, the **Solver Profiler** may temporarily modify some logging settings of your model. Your settings will be restored after profiling has been completed.

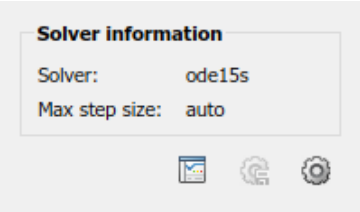
These events are common and necessary for an accurate simulation. However, they do incur computational cost and frequent occurrences can slow down or even stall the simulation.

To examine model dynamics and identify causes that affect the simulation:

1. Open the model that is simulating slowly or unsuccessfully.



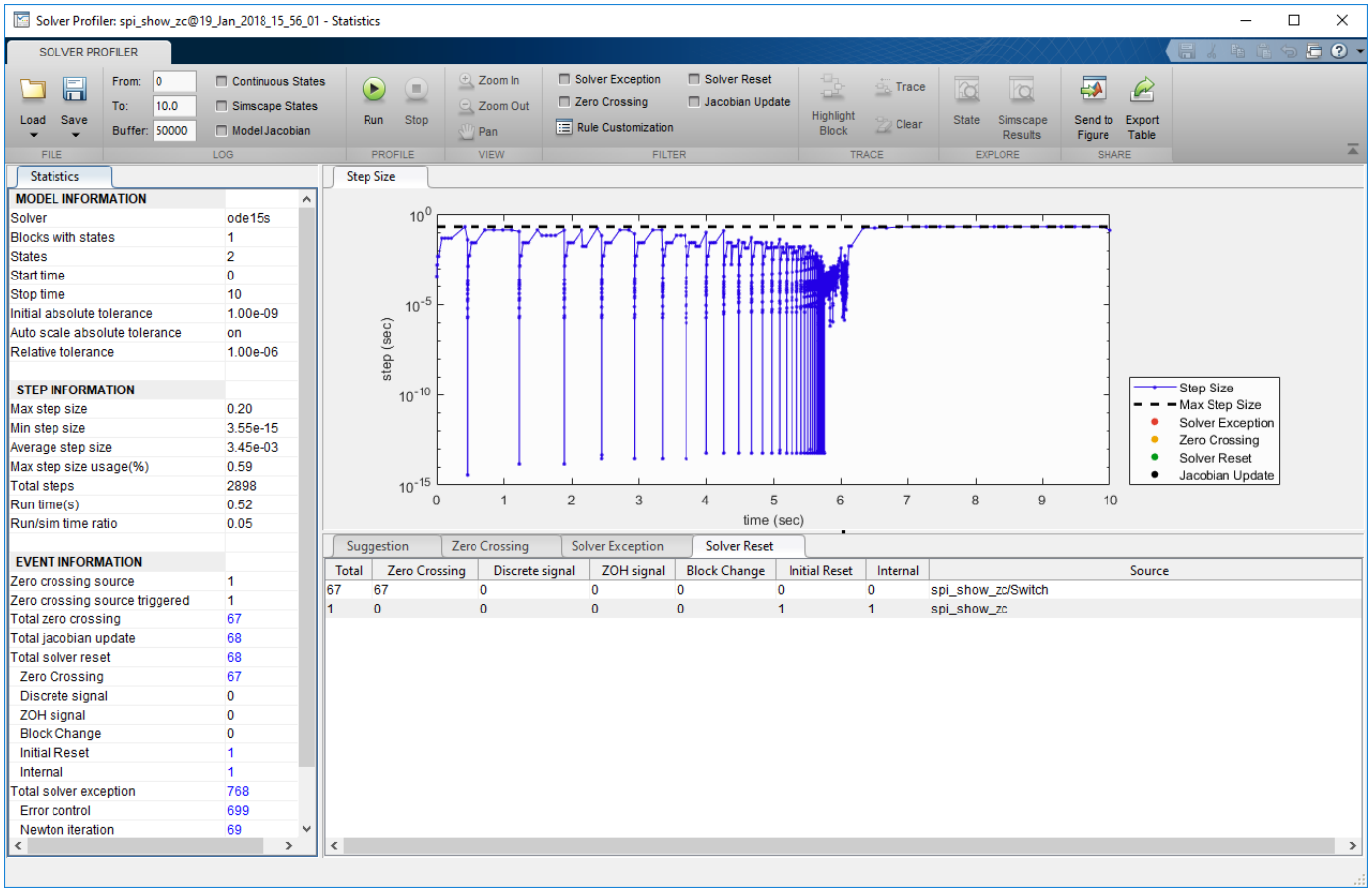
2. Open the Solver Profiler by clicking the hyperlink in the lower-right corner of the Simulink Editor.



3 The Solver Profiler provides smart logging and diagnostics of continuous model states and Simscape states. To enable this, select the **States & Zero Crossing** or **Simscape States** option before a run. Disable these options only if you are running out of memory. After the run, access the States Explorer or Simscape Explorer to examine those states.

4 Click **Run**. The profiler simulates the model and starts capturing solver performance data.

When the simulation ends, the profiler displays the statistics and exceptions it captured over the duration of the simulation.



Tip You can pause or stop the simulation at any time to view the information captured until that point.

5 Use the profiler plot to highlight the parts of the model that caused generate the most events.

6 Click **Save** to capture your profiling session, or exit without saving.

See Also

Solver Profiler | Zero Crossing Explorer | States Explorer | `solverprofiler.profileModel`

Understand Profiling Results

If you have a large model, it can be challenging to identify which parts of your model cause the solver to take small steps. The Solver Profiler logs and reports events when the solver attempts to take large steps. This report of events helps you identify which parts of your model to focus on to improve solver performance.

This topic presents simple examples that illustrate various events that the profiler reports.

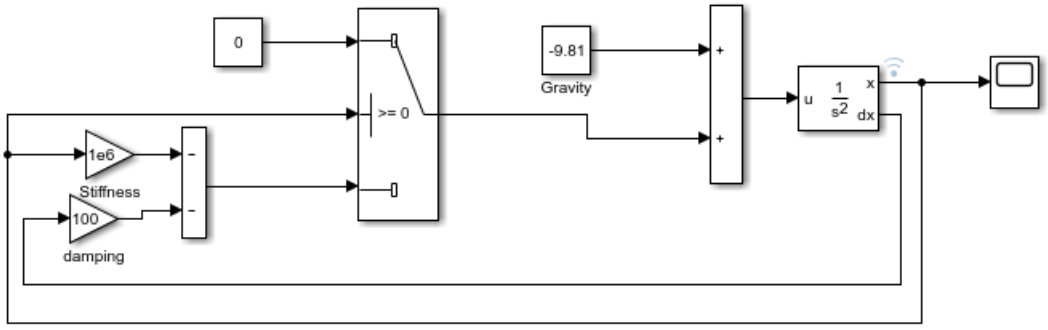
See Also

Related Examples

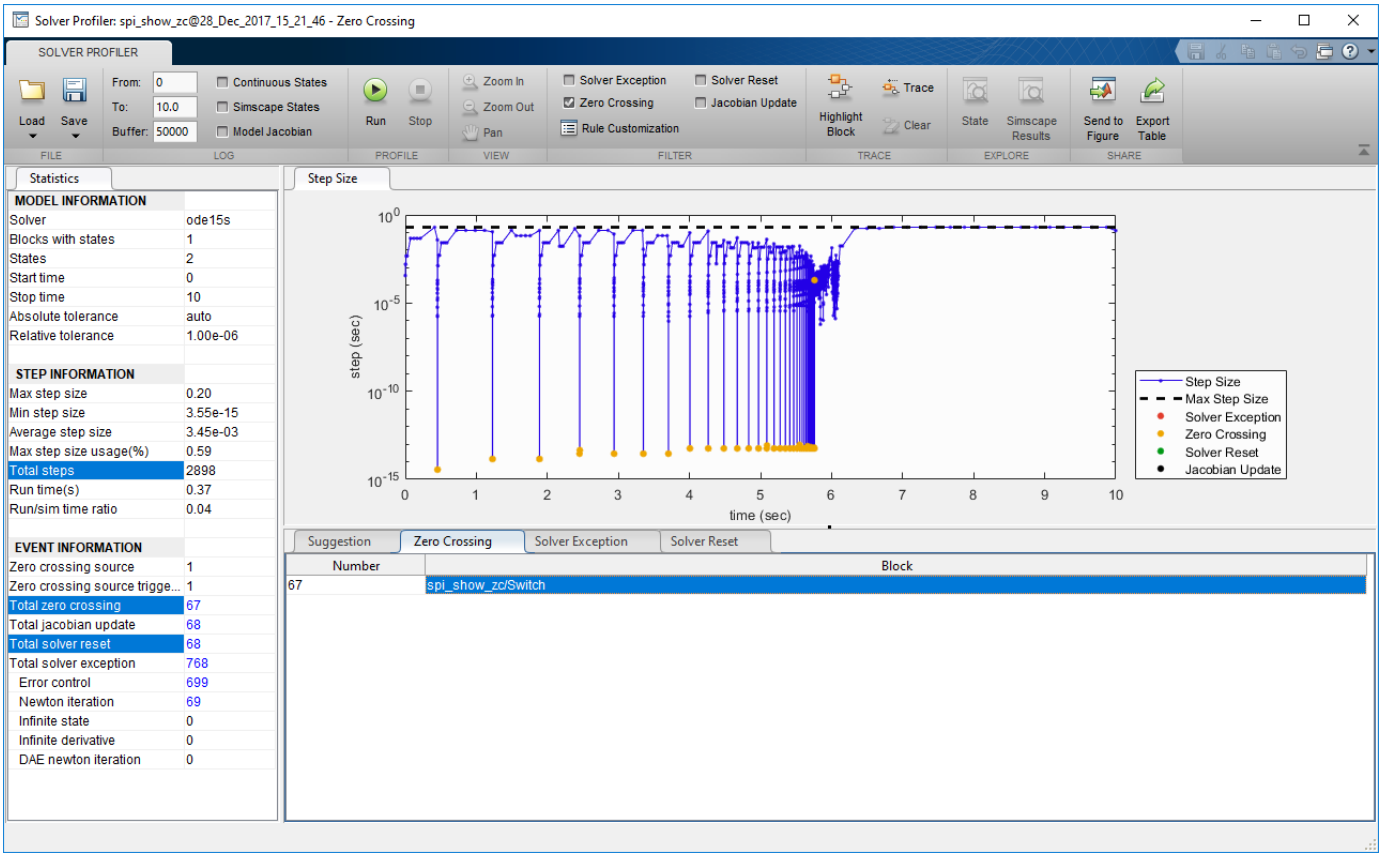
- “Examine Model Dynamics Using Solver Profiler” on page 33-2

Zero-Crossing Events

This example simulates a ball bouncing on a hard surface.

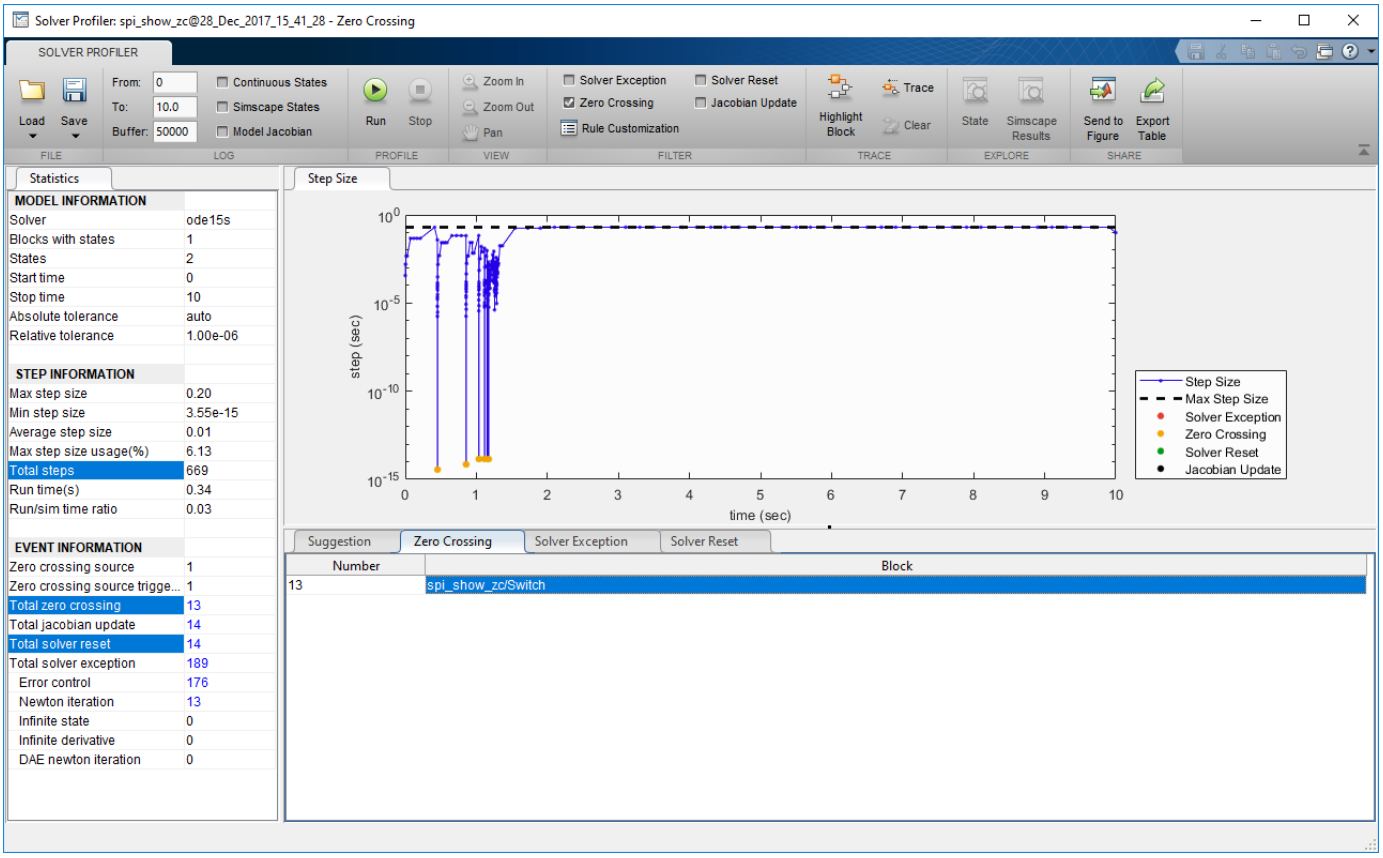


When you run the Solver Profiler on this model, the model simulates in 2898 steps and that it triggers 67 zero crossing events. To highlight the zero crossing events on the step size plot, click the **Zero Crossing** tab and select the block that is causing the event.



The result indicates that when the ball drops on the hard surface, it bounces 67 times before coming to a stop. The solver resets after each bounce, increasing the computational load. Having many resets improves accuracy at the cost of computation load. Therefore, it is important to know this tradeoff when modeling.

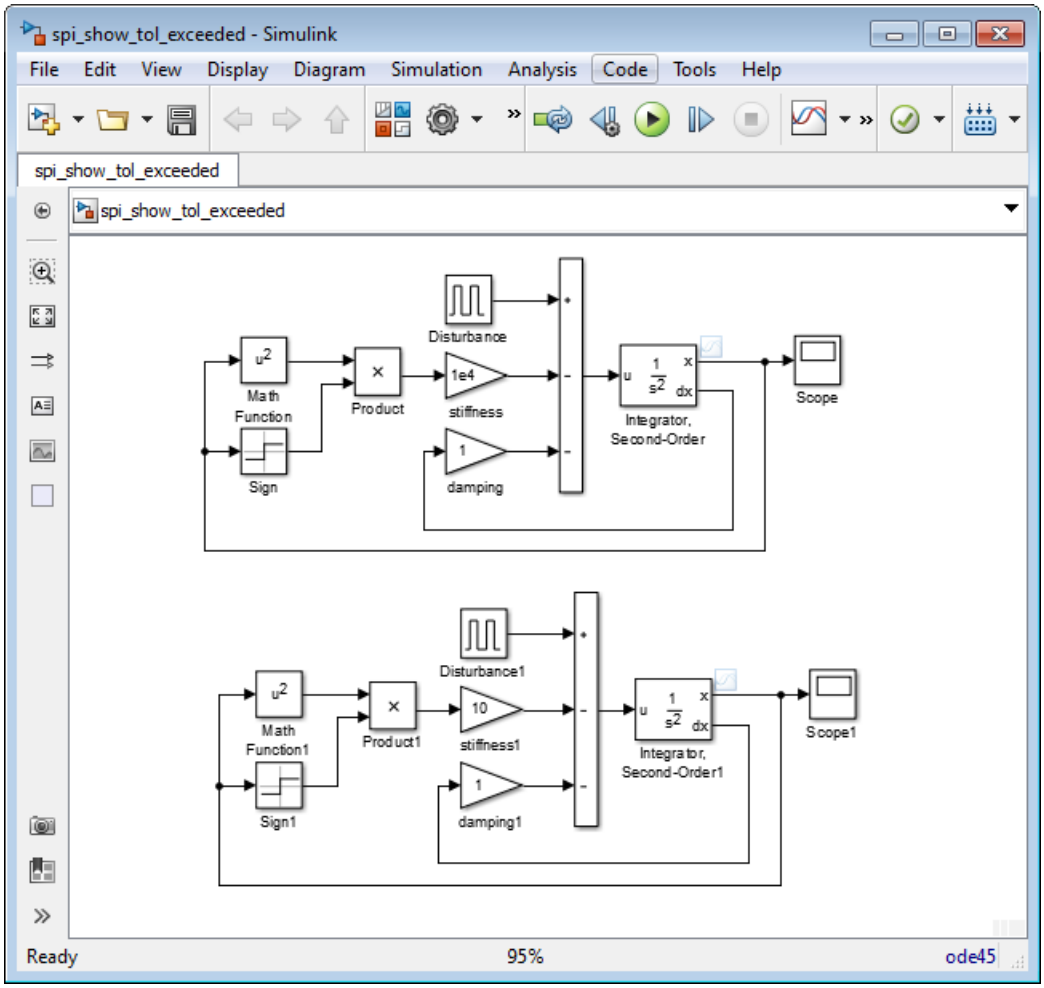
If this modeling construct belonged to a larger model, the Solver Profiler would help you locate it. You could then modify the model to improve solver performance. For example, you can decide to reduce the accuracy of the contact dynamic by increasing the damping factor, which would reduce the number of bounce events. Increasing the damping from 100 to 500 makes the ball bounce only 13 times, allowing the simulation to complete in only 669 steps.



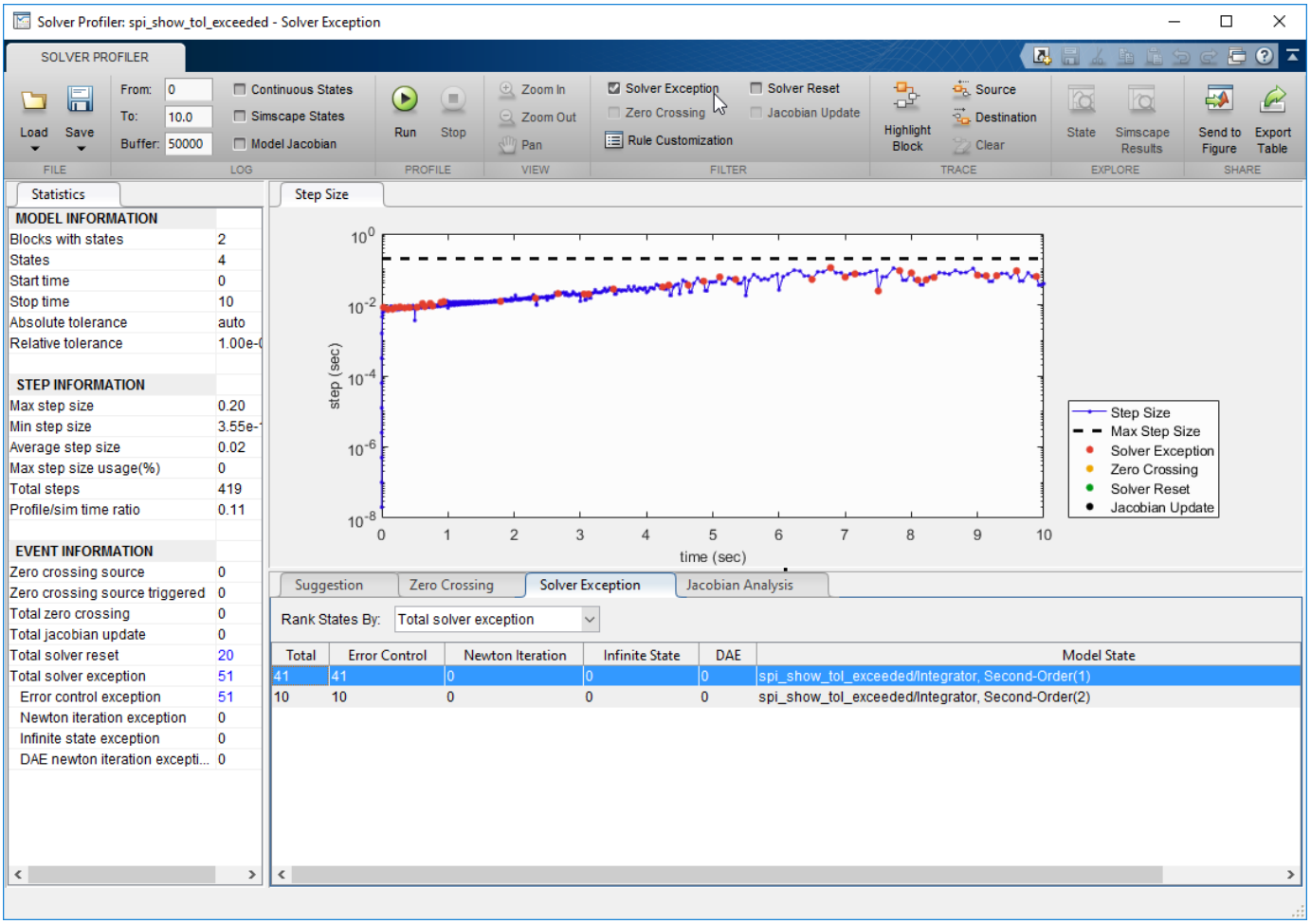
Solver Exception Events

Tolerance-Exceeding Events

This example simulates two identical nonlinear spring-damping systems. The two systems have different spring stiffness.



When you run the Solver Profiler on this model, you can see the tolerance-exceeding events in the **Solver Exception** tab.



The result indicates that the stiffer spring causes the solver tolerance to exceed the limit. Typically, model states that change the fastest tend to be closest to the solver tolerance limit.

The solver attempts to take the largest possible steps while optimally trading off between speed and accuracy. Occasionally, this tradeoff causes the solver to take steps that exceed the tolerance limit and forces it to reduce the step size. Exceeding the tolerance limit is not a poor modeling practice in itself. This profiler statistic is not meant to help you reduce tolerance exceeding events to zero.

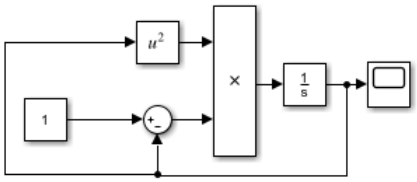
This statistic can help you identify parts of your model that are close to exceeding the tolerance limit. You can identify model components that change the fastest or are the stiffest. You can decide to retain this model dynamic in the simulation or simplify it to speed up simulation.

The tolerance-exceeding statistic can also help you identify modeling errors. If you do not expect the highlighted states to change as fast, you can examine your model for errors. In this example, the modeling error could be that the stiffness of the stiffer spring is specified in `N/m` instead of `N/mm`. This error makes the spring 1000 times stiffer than expected.

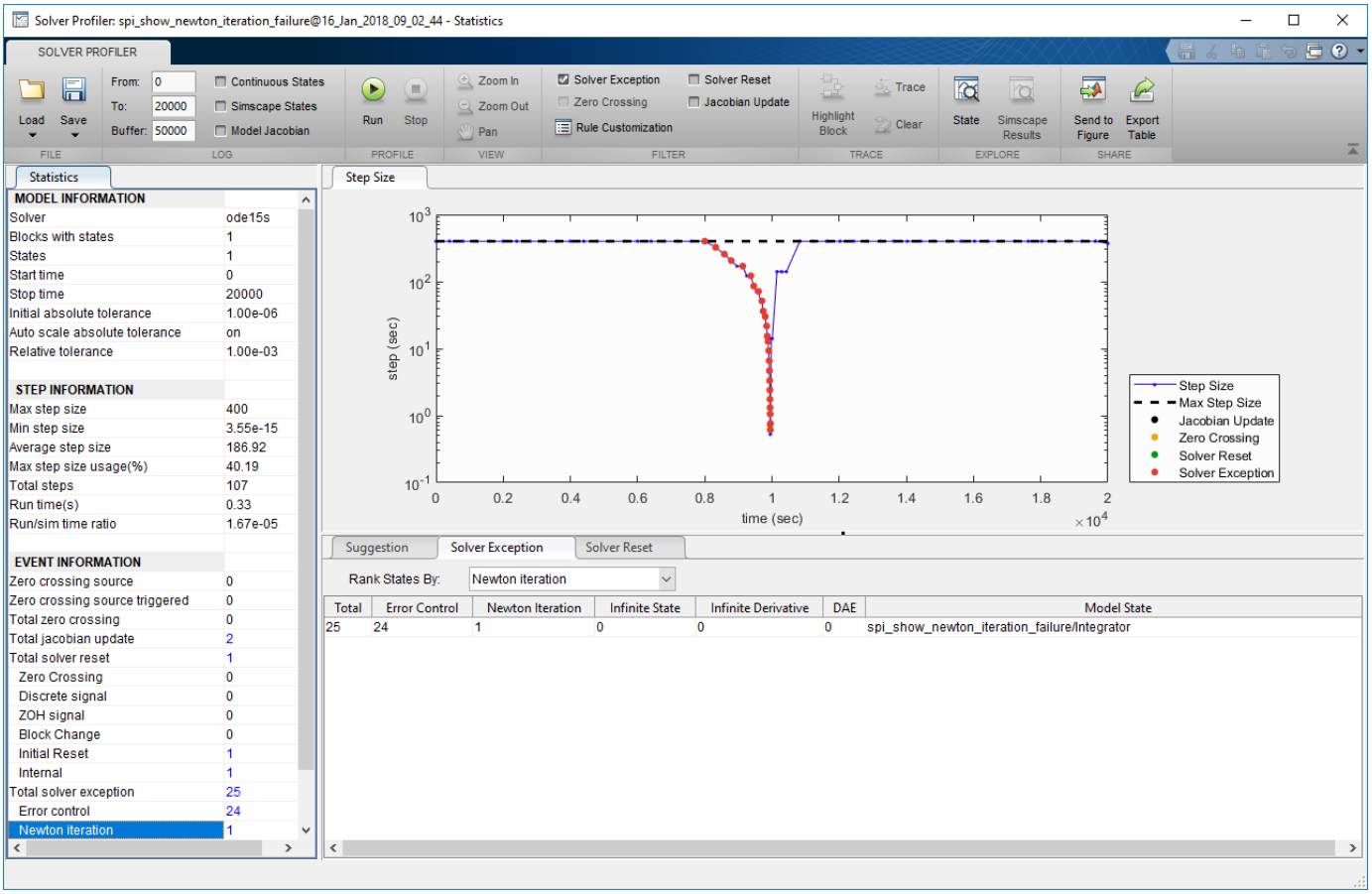
Newton Iteration Failures

Newton iteration failures are specific to implicit solvers like `ode15s` and `ode23t`, and they result from Newton iterations not converging after a few trials. Similar to tolerance-exceeding events, these failures tend to occur when a system changes quickly.

This example simulates how the radius of a ball of flame changes when you strike a match. The ball of flame grows rapidly until it reaches a critical size, when the amount of oxygen consumed balances the growth in ball surface.



When you run the Solver Profiler on this model, you can see the Newton iteration failures in the **Solver Exception** tab.



The result indicates that when the combustion begins, the solver tolerance is exceeded multiple times. When equilibrium is attained, the system appears different, and a Newton iteration failure occurs. The Jacobian of the system is recomputed, and the solver continues to move forward.

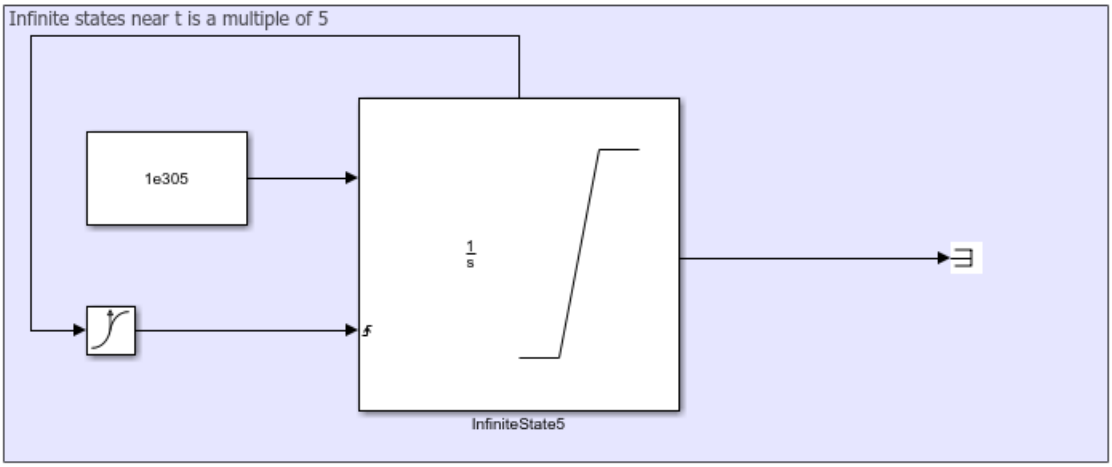
Newton failures are not indicative of poor modeling practices. This profiler statistic is not meant to help you reduce these failures to zero. In this example, you can reduce the solver tolerance to prevent this failure. But the solver then takes small steps unnecessarily, which is counterproductive. Therefore, in this example, this failure is acceptable.

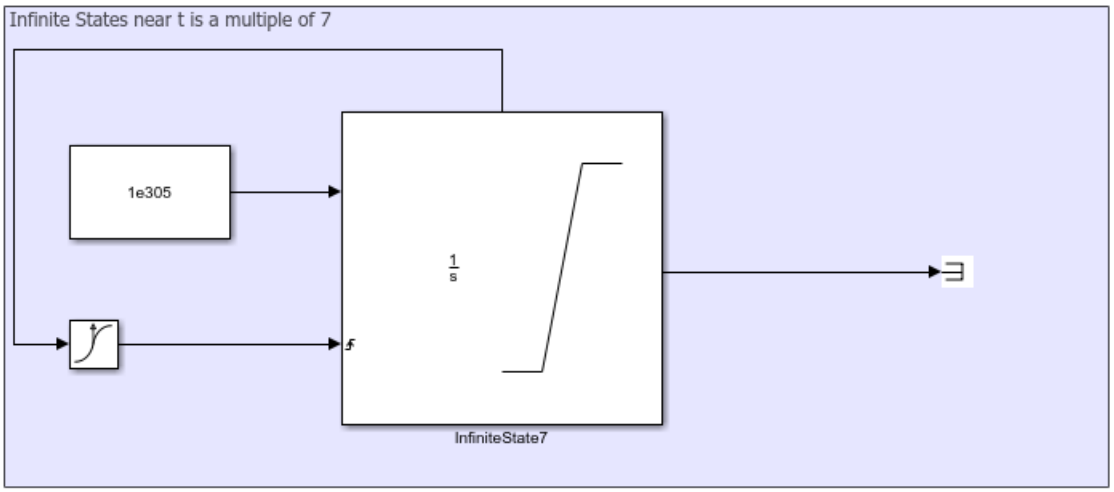
This type of failure becomes problematic when it occurs in large numbers over a short period, especially in Simscape models. Dense failures indicate that your model is not robust numerically. One way to improve numerical robustness is to tighten the solver tolerance. Another way is to modify the model to avoid rapid changes.

Infinite State and Infinite Derivative Exceptions

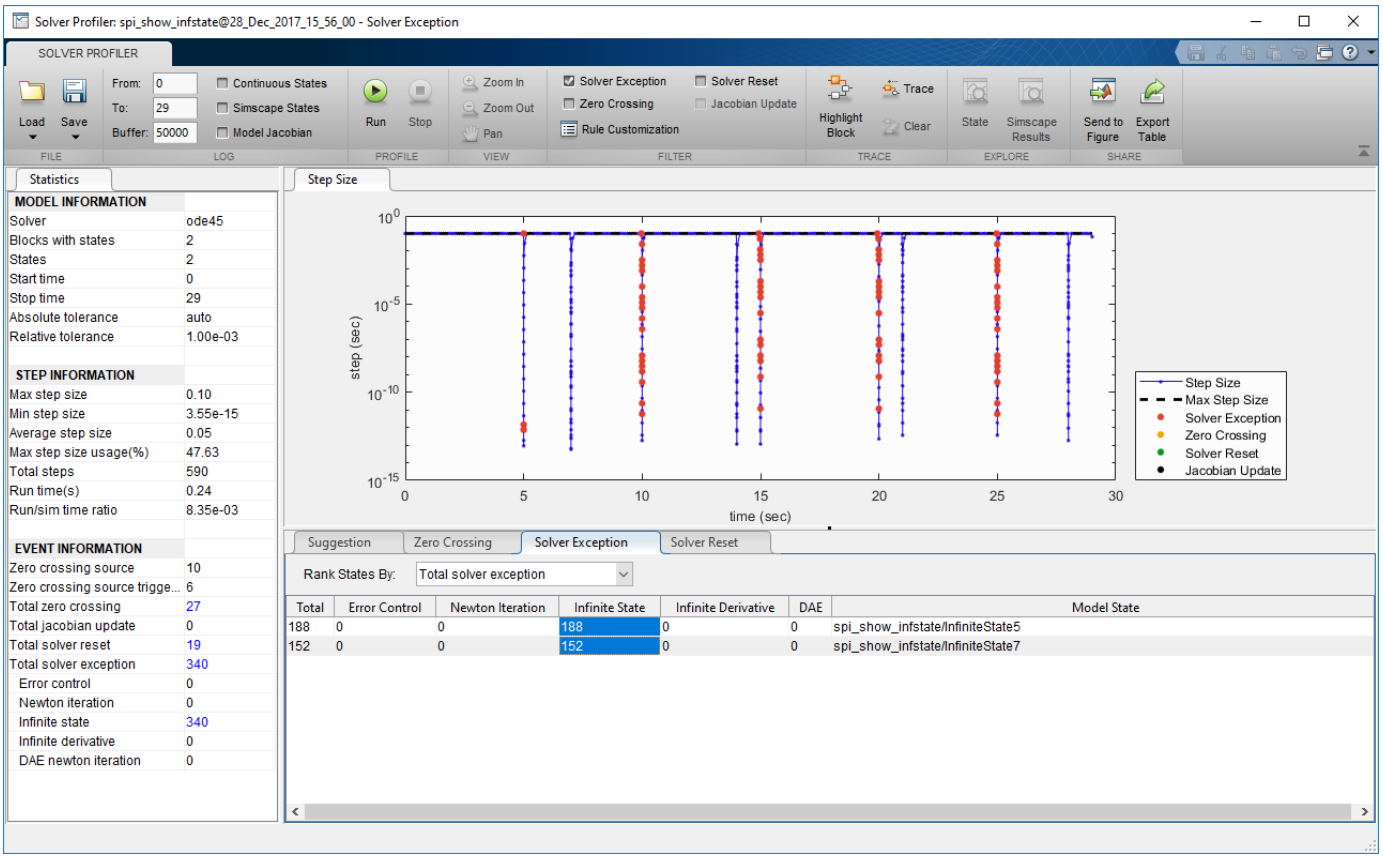
An infinite state exception occurs when the magnitude of a state approaches infinity. Similarly, when the magnitude of the derivative of a state approaches infinity, an infinite derivative exception occurs. The solver reduces the step size and makes another attempt at integration.

This example shows a set of two Integrator Limited blocks that have initial conditions slightly less than `realmax`. As they integrate the Constant input of `1e305`, the solver quickly reaches the infinite state exception.





When you run the Solver Profiler for this model, you can see that the InfiniteState5 and InfiniteState7 blocks have 188 and 152 infinite state exceptions in the **Solver Exceptions** tab.

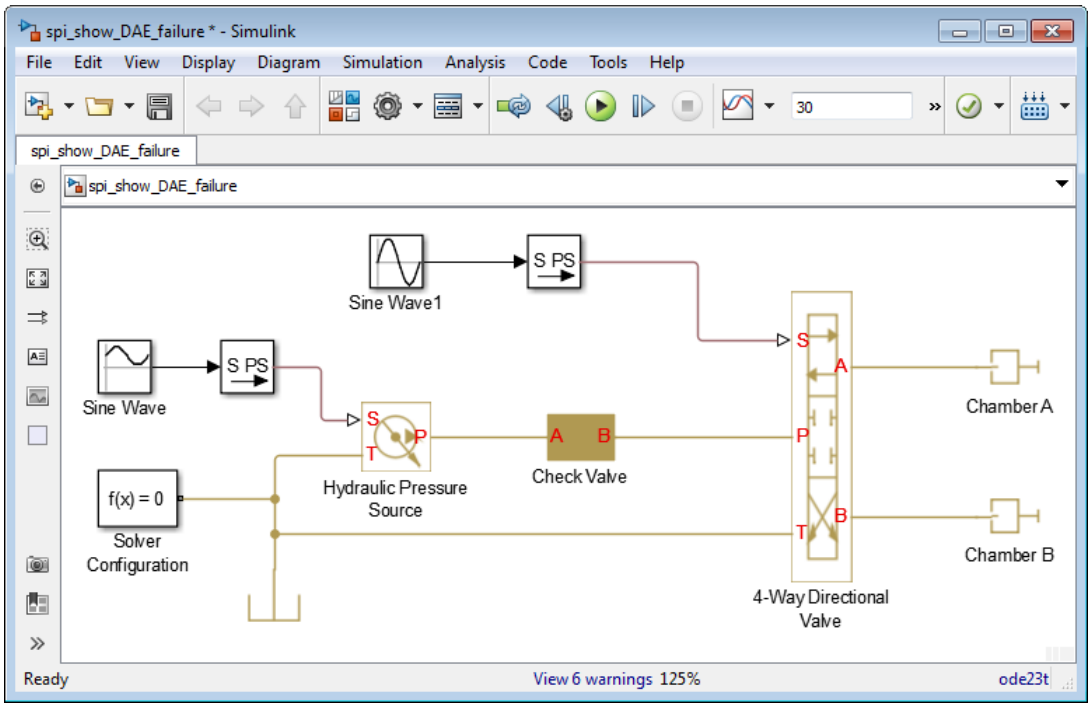


Differential Algebraic Equation Failures

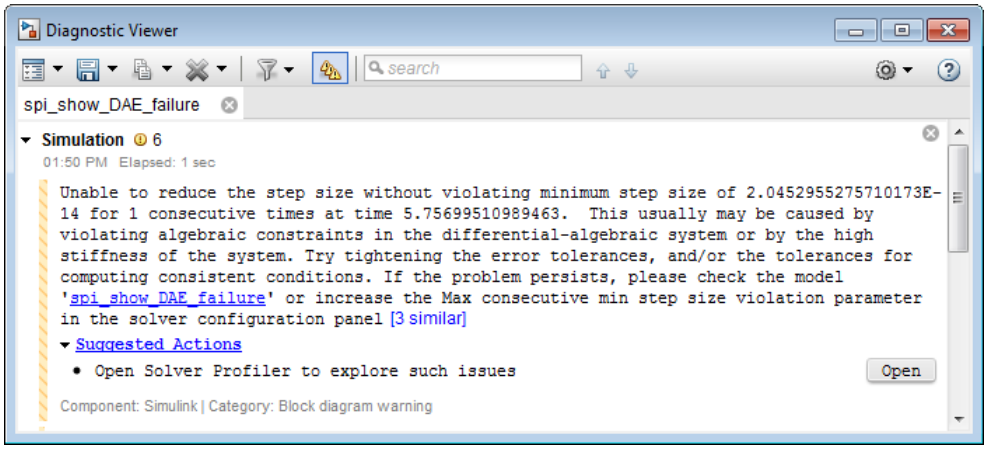
Most Simscape models use differential algebraic equations (DAEs), in contrast to Simulink models, which use ordinary differential equations.

The use of DAEs adds complexity to Simscape models. Solvers like `ode15s` and `ode23t` can handle many types of DAEs. However, when the algebraic constraints between Simscape components are complex and changing fast, the Newton iteration process fails to resolve to those constraints.

This example simulates a pressure source that can be directed toward one of two hydraulic chambers.

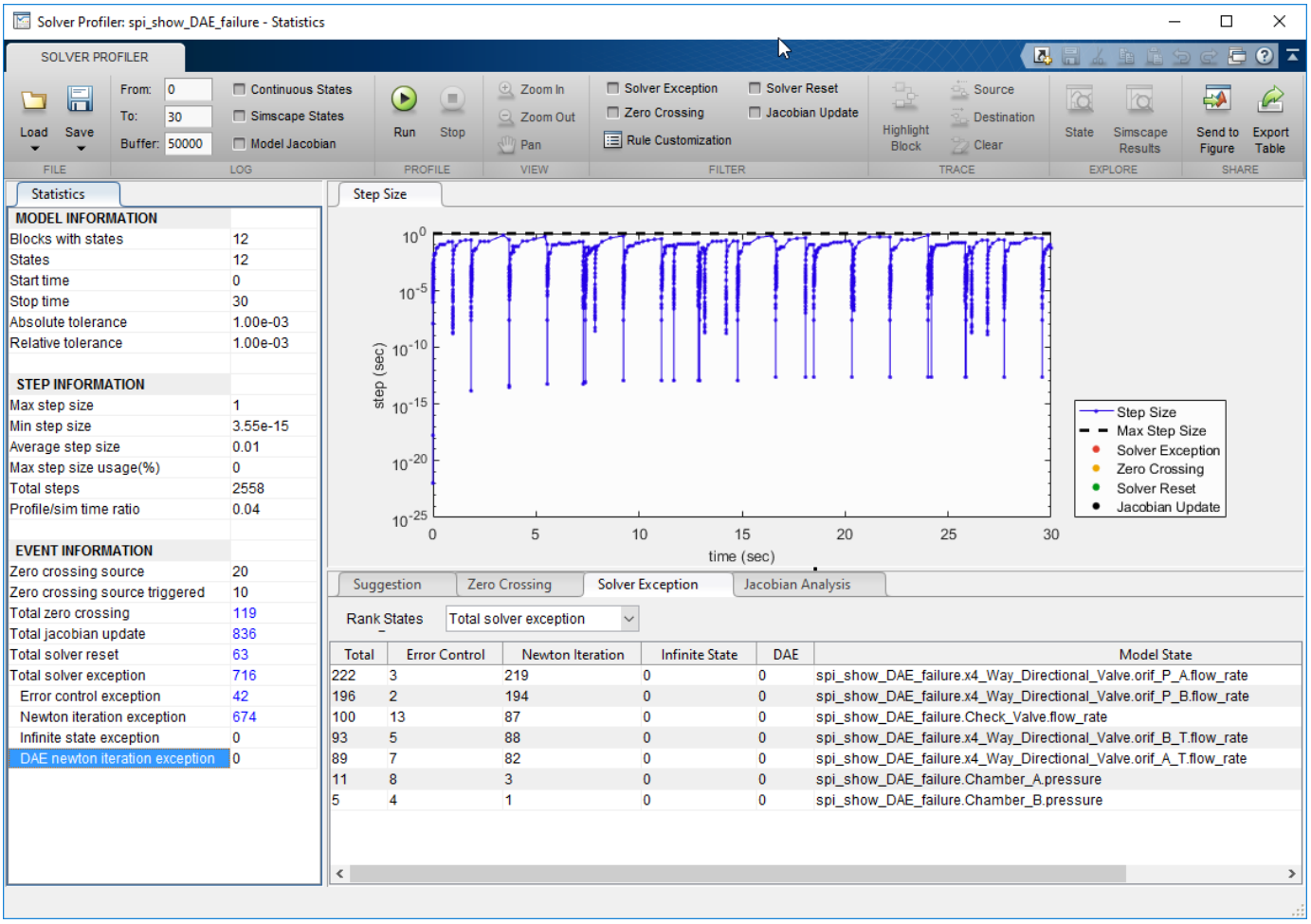


When you simulate this model, Simulink displays several warnings.



Typically, small models can handle such warnings and simulate to completion. However, this warning indicates that the model is not robust numerically. Minor changes to the model, or integration into a larger model, can result in errors.

When you run the Solver Profiler on this model, you can see the DAE failures in the **Solver Exception** tab.



In this case, the exception results from large solver tolerance. Tightening the solver tolerance forces the solver to take smaller steps and better capture the changes in algebraic constraints.

Alternatively, this exception can be avoided by removing the algebraic constraint. In this example, the **Check Valve** and the **4-Way Directional Valve** are directly connected. When their pressure–flow relationship changes rapidly, the solver is unable to capture the changes. Inserting a **Hydraulic Chamber** between those two components makes them compliant. To learn more about dry nodes, see Simscape documentation.

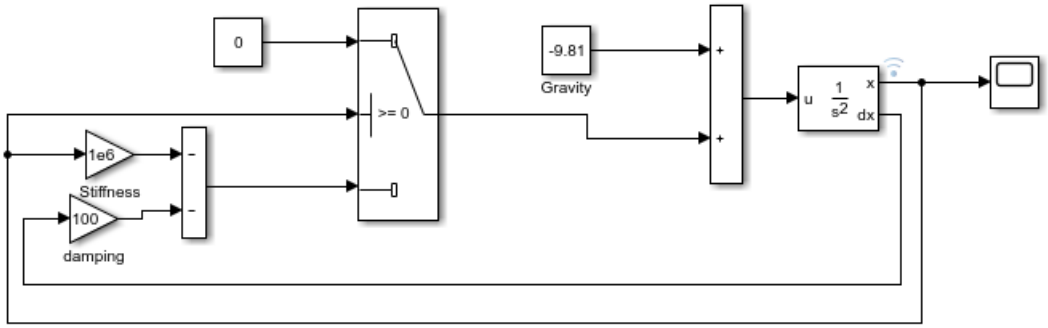
Solver Resets

The Solver Profiler logs events that cause the solver to reset its parameters because solver resets do incur computational cost. In addition to the total number of resets, you can also see a breakdown by type of reset event.

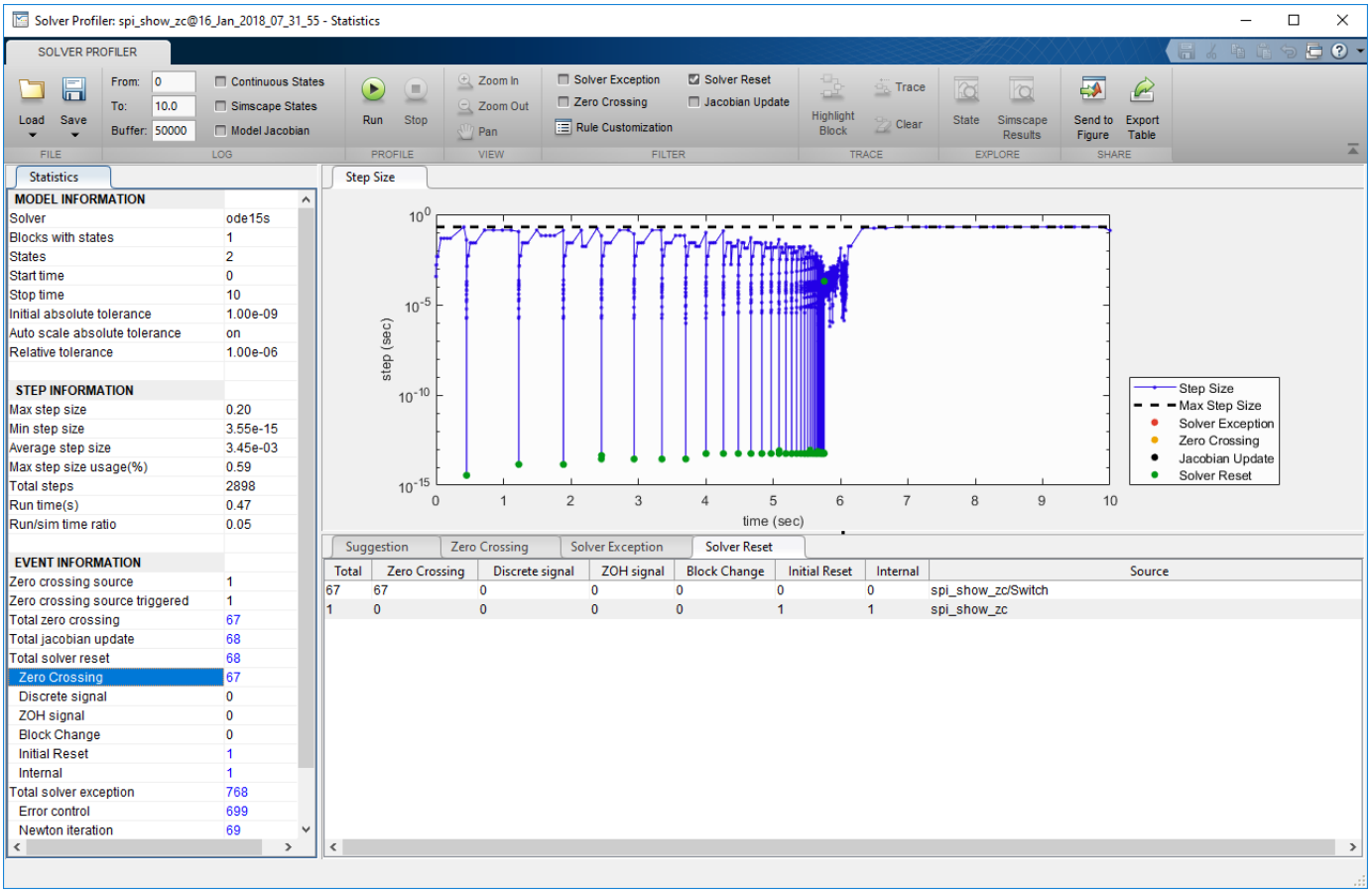
Note A solver reset event can have multiple causes. As a result, the number of **Total solver reset** events in the **Statistics** pane may be less than the sum of the solver resets of each type.

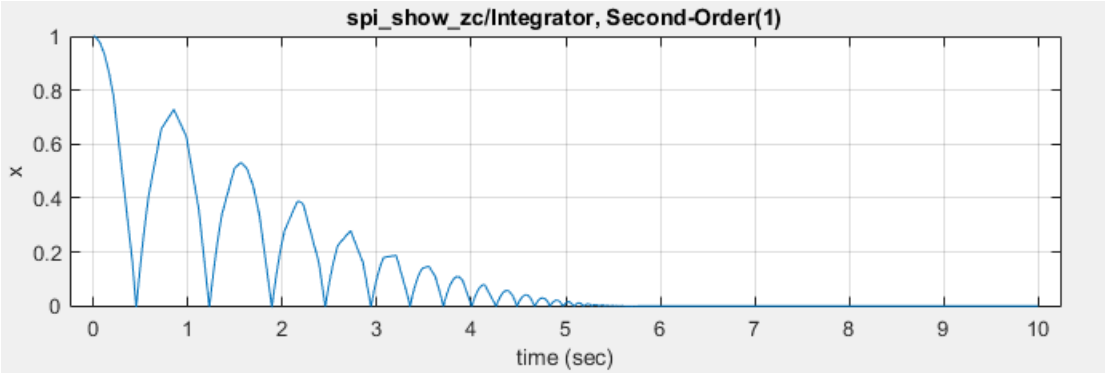
Zero-Crossing

Some zero crossing events occur at discontinuities in the signal. Consider the example model of a bouncing ball from the “Zero-Crossing Events” on page 33-6 section.



The Solver Profiler records 67 solver resets caused by zero crossings from the Switch block in the model. Compare the step-size and the **Solver Reset** highlighting with the output *x* of the Second-Order Integrator block, which is the height of the ball from the ground. Observe that the solver resets occur when the ball bounces off the ground.



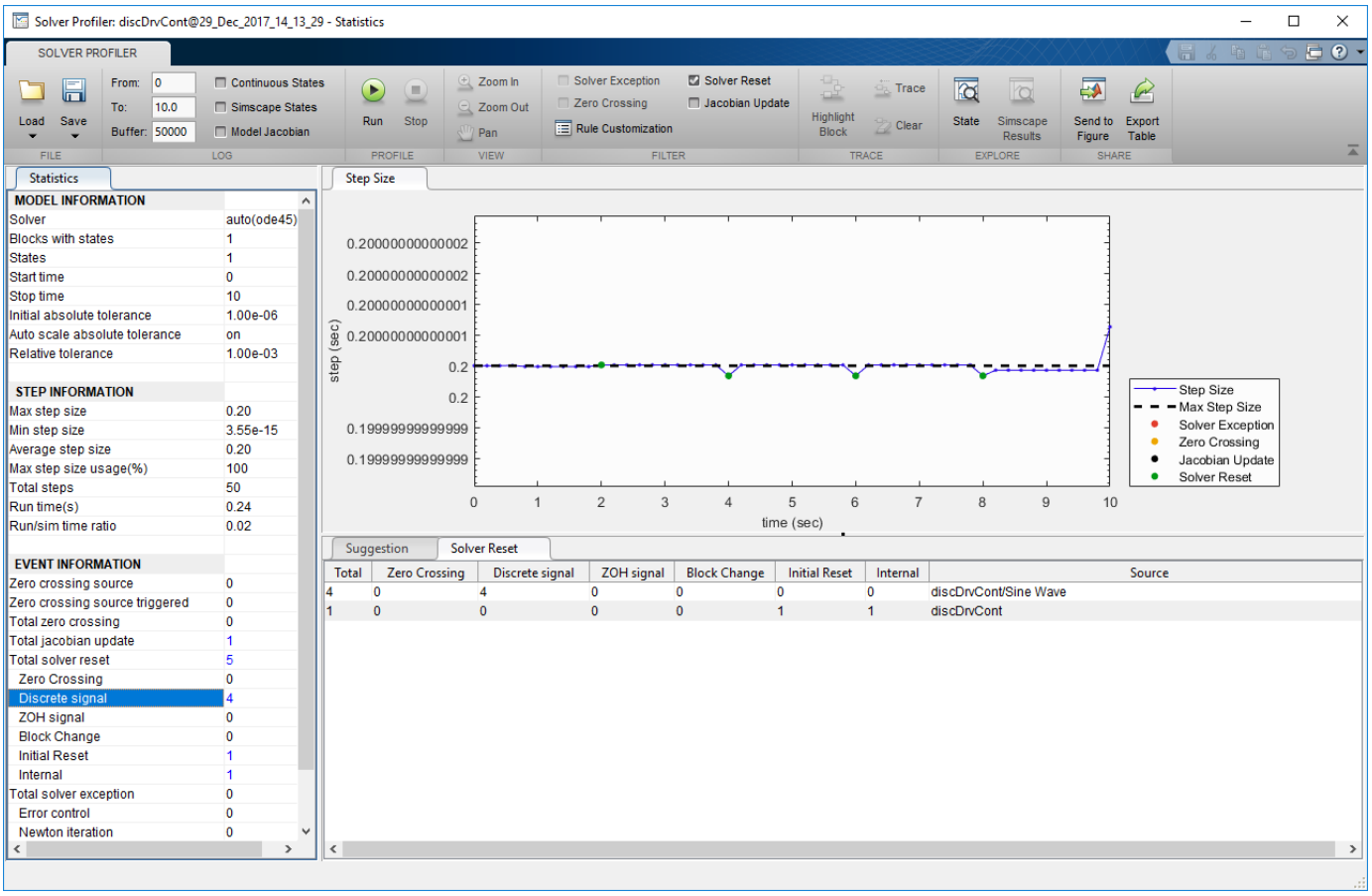


Discrete Signal

Solver resets also occur when your model has a discrete signal driving a block with continuous states, as seen in this example model.

The discrete Sine Wave block outputs a 1 rad/s sine wave with a discrete sample time $t_s = 2$.

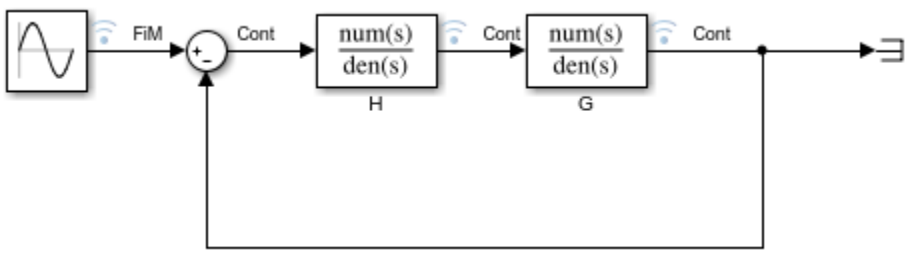
The Solver Profiler report shows that four solver resets occur due to discrete signals driving a continuous block. These resets occur at every execution of the Sine Wave block.



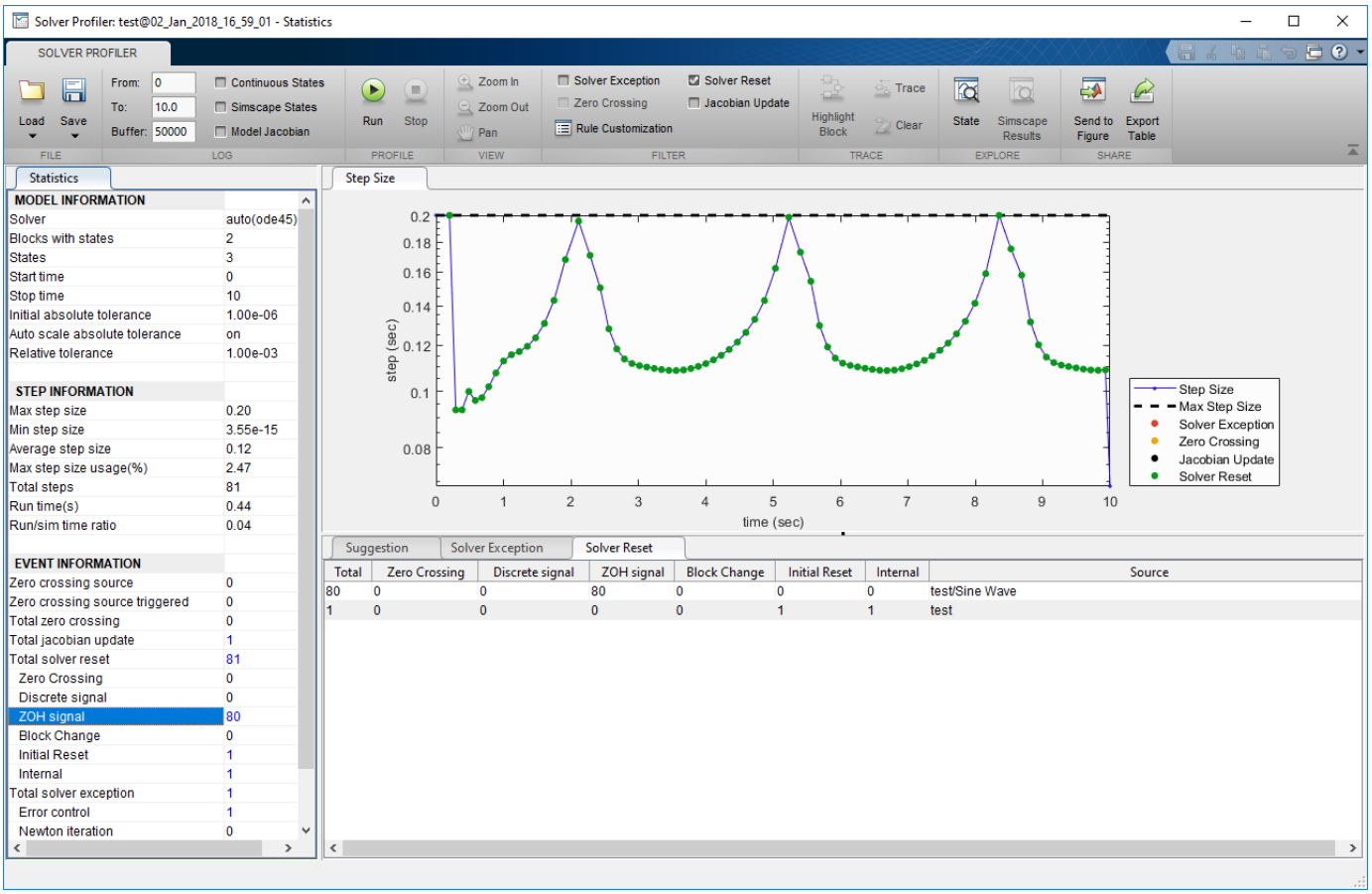
ZOH Signal

This type of solver reset occurs when a block output is not executed during a trial or minor time step and gets updated only during the integration or major time step. As a result, the block output changes discontinuously from one major time step to the other. As a result the solver resets. See “Fixed-in-Minor-Step” on page 7-14 sample time.

This example model shows a simple plant and controller that tracks a sinusoidal reference signal. The source of this signal is a Sine Wave block whose sample time is specified to be fixed-in-minor-step.



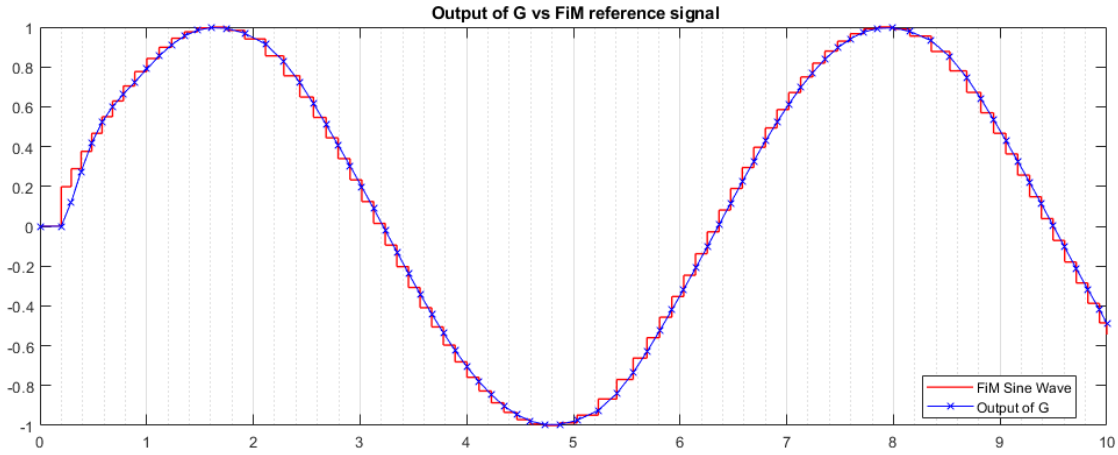
Observe from the results of the profiling session that there are `80` **ZOH Signal** solver resets in the simulation.



Note When you select **Continuous States** for logging or enable the `SaveStates` parameter, the derivative of a continuous block driven by a Fixed-in-Minor-Step signal is `NaN`.

This is because the driving “Fixed-in-Minor-Step” on page 7-14 block updates its value only at every major time step. The signal driven to the continuous block is therefore discontinuous and the state is non differentiable.

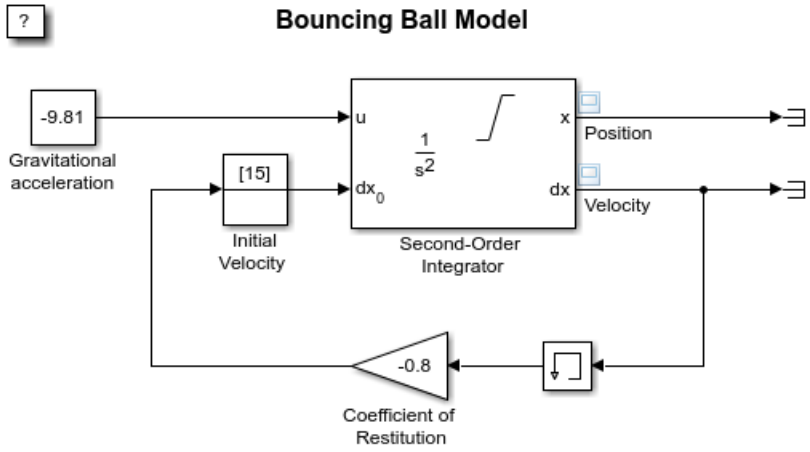
The plot shows the outputs of the Sine Wave and Integrator blocks in the example model.



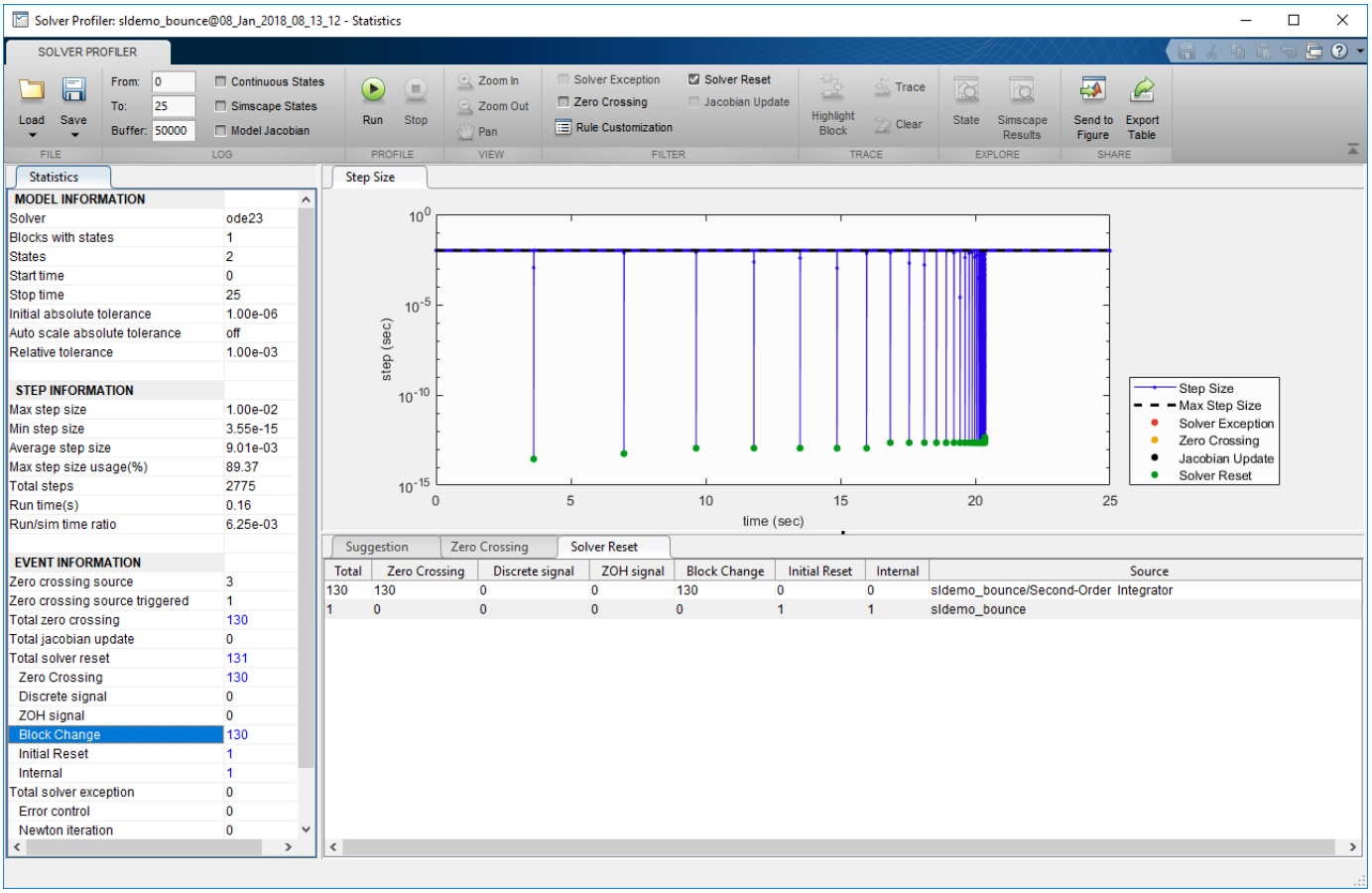
Tip Solver resets caused due to ZOH signals are serious compared to solver reset events caused by discrete signals and can significantly slow down the simulation. This is because ZOH signal reset events occur at every major step of the solver. Reconfigure your simulation to avoid these resets, if needed.

Block Signal

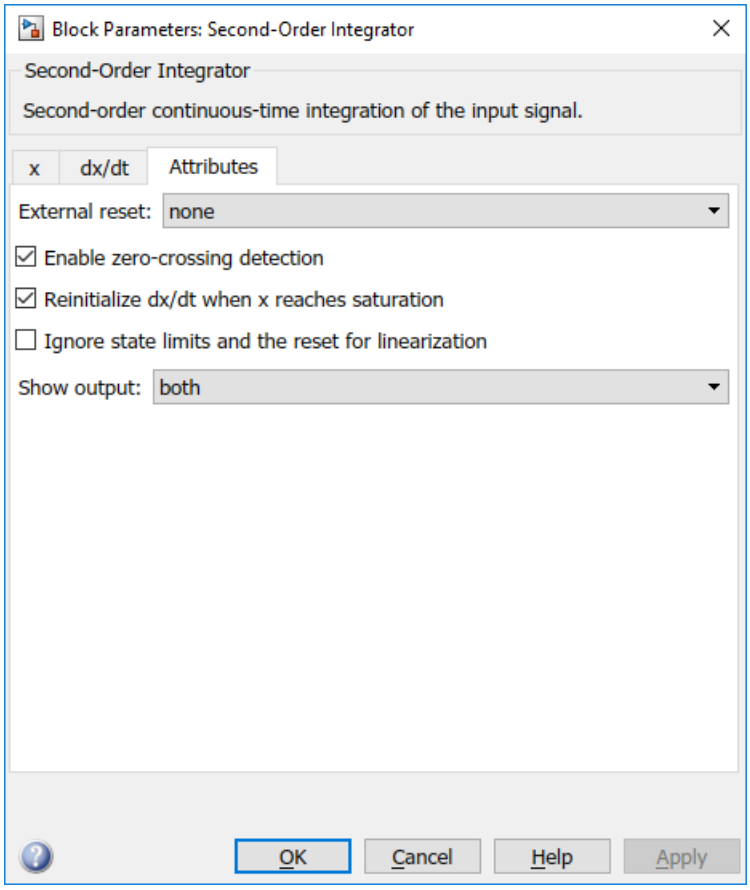
Sometimes, the block can be configured to reset the solver when certain conditions are satisfied during its execution. Consider the `sldemo_bounce`model of a bouncing ball which can be found in the “Capture the Velocity of a Bouncing Ball with the Memory Block” example.



Observe from the profiling results that there are `130` solver resets triggered by a block. A solver reset event can have multiple causes. In this case, zero crossings and block changes cause a solver reset event.



One setting that causes the **Block Change** solver reset event is the **Reinitialize dx/dt when x reaches saturation** parameter. This setting is necessary to properly simulate the dynamics of a bouncing ball.



Initial Reset

When you run a simulation, the solver needs to reset its parameters for the first time. This event is shown as the **Initial Reset** event in the Solver Profiler report. It occurs once at the start of the simulation.

Internal

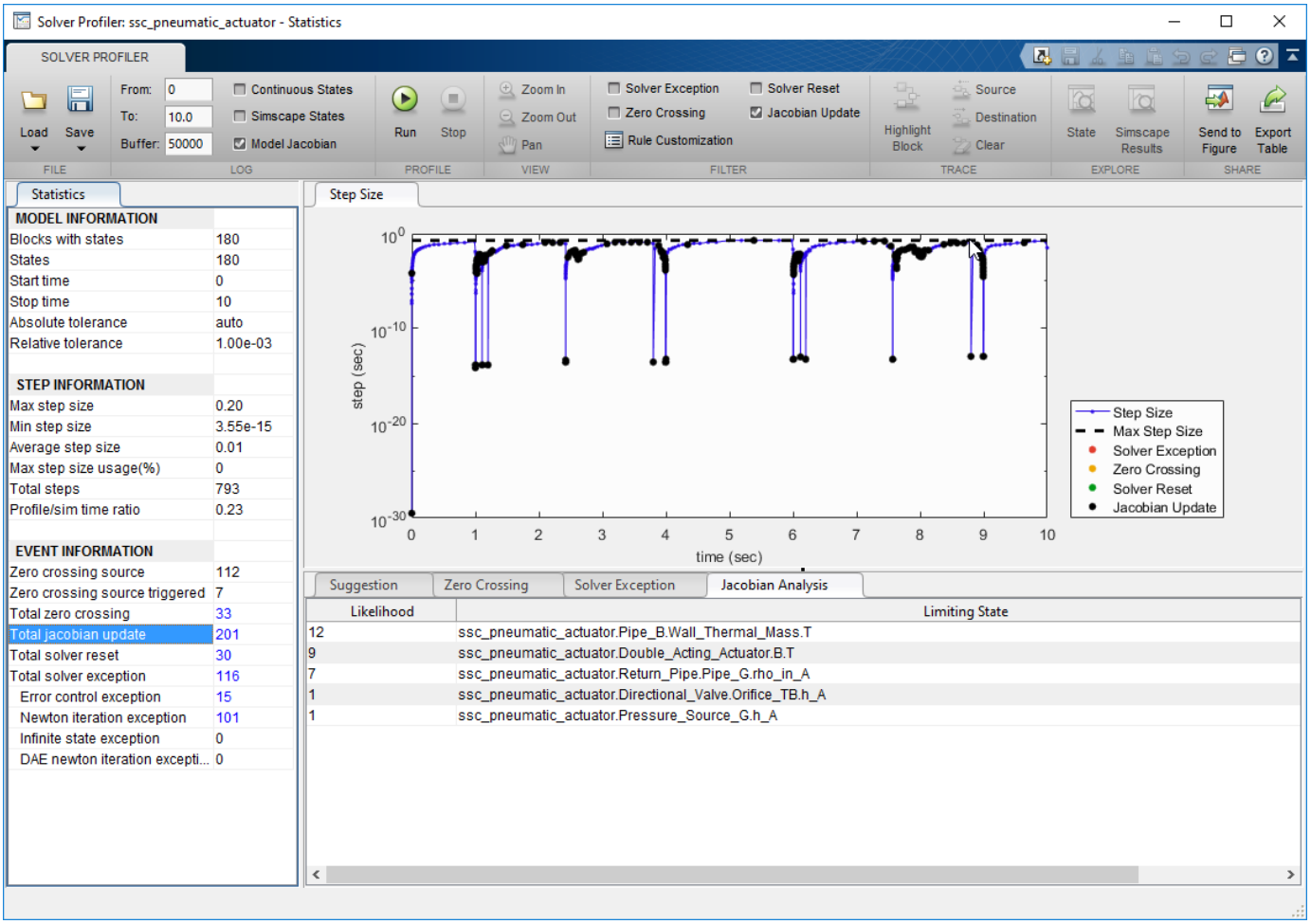
There are some reset events that are internal to the Simulink engine. These reset events are necessary for the engine to correctly configure itself for accurate simulation.

Jacobian Logging and Analysis

The Solver Profiler supports Jacobian logging and analysis for implicit solvers only.

You can select the **Model Jacobian** check box in the Solver Profiler to log updates to the Jacobian matrix for the model. When you run the Solver Profiler on a model, you can see the logs in the **Jacobian Analysis** tab.

The **Jacobian Analysis** tab indicates the states in the model that are likely slowing down the solver.



To investigate the cause of each limiting state, select a row in the **Jacobian Analysis** tab and click **Highlight Block**.

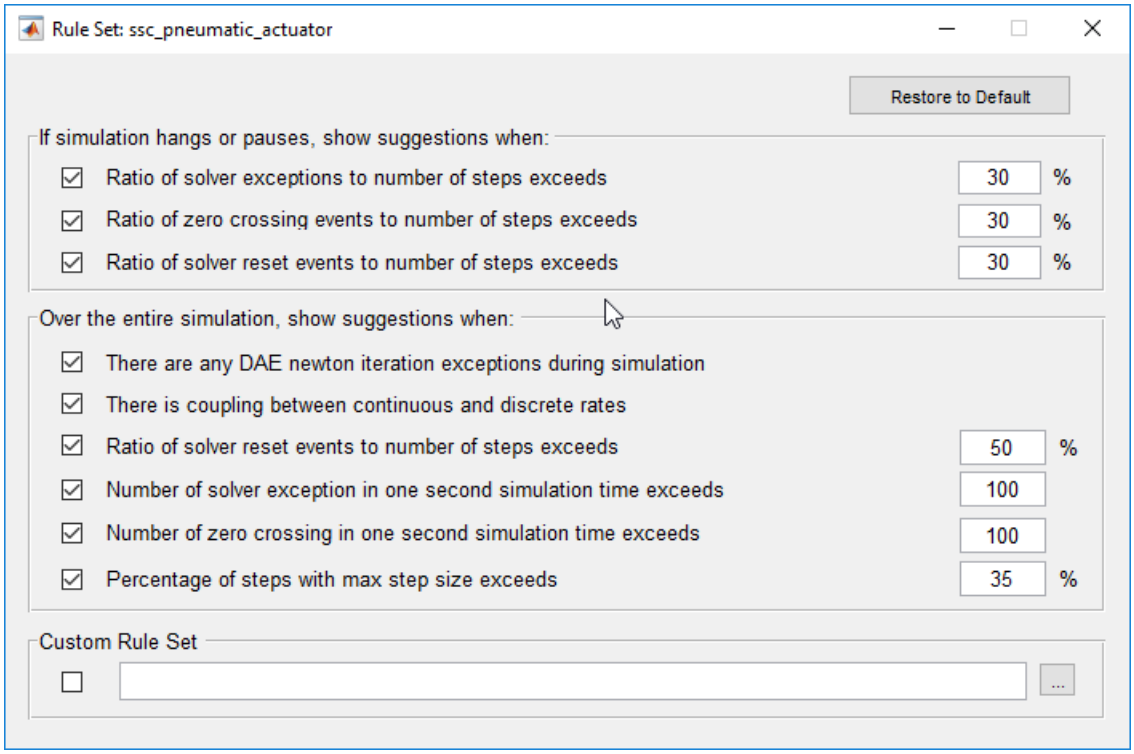
Modify Solver Profiler Rules

You can customize the suggestions that appear in the Solver Profiler in these ways:

- Change the thresholds of the Solver Profiler rules that detect failure patterns.
- Select which rules to apply during a profiling run.
- Author your own rule set as a MATLAB script.

Change Thresholds of Profiler Rules

Click **Rule Customization** in the Solver Profiler to access the rule set. You can change the thresholds for most of these rules and also select which rules you want to apply selectively during a simulation run.



Develop Profiler Rule Set

You can override the settings on the **Rule Set** dialog box by specifying a custom rule set.

Create a rule set as a MATLAB script and specify the path to the script in the **Custom Rule Set** section of the **Rule Set** dialog box.

A simple rule set example looks as follows:

```
function diagnosticsString = customRule(profilerData)
        if isempty(profilerData.zcEvents)
            diagnosticsString{1} = 'No zero crossing event detected.';
        else
```

```
            diagnosticsString{1} = 'Zero-crossing events detected.';
        end
end
```

The input to the function is an array of structures called `profilerData`. This array of structures organizes all the information that the Solver Profiler collects during a profiling run. It contains the following substructures.

Substructure	Fields
`stateInfo`: Stores information on block states	• `name`: Block name • `value`: State values • `blockIdx`: Block ID
`blockInfo`: Cross-reference of blocks and state IDs	• `name`: Block name • `stateIdx`: State ID
`zcSrcInfo`: Stores information on blocks causing zero crossing events	• `name`: Block name • `blockIdx`: Block ID
`zcEvents`: Cross-reference of the time stamps of zero crossing events and the corresponding state IDs	• `t`: Event timestamp • `srcIdx`: Block ID
`exceptionEvents`: Cross-reference of exception event timestamps, the ID of the corresponding state that caused the event, and the cause.	• `t`: Event timestamp • `stateIdx`: State ID • `cause`: Cause of exception
`resetTime`: Stores timestamps of solver resets.	None
`tout`: Stores simulation times.	None

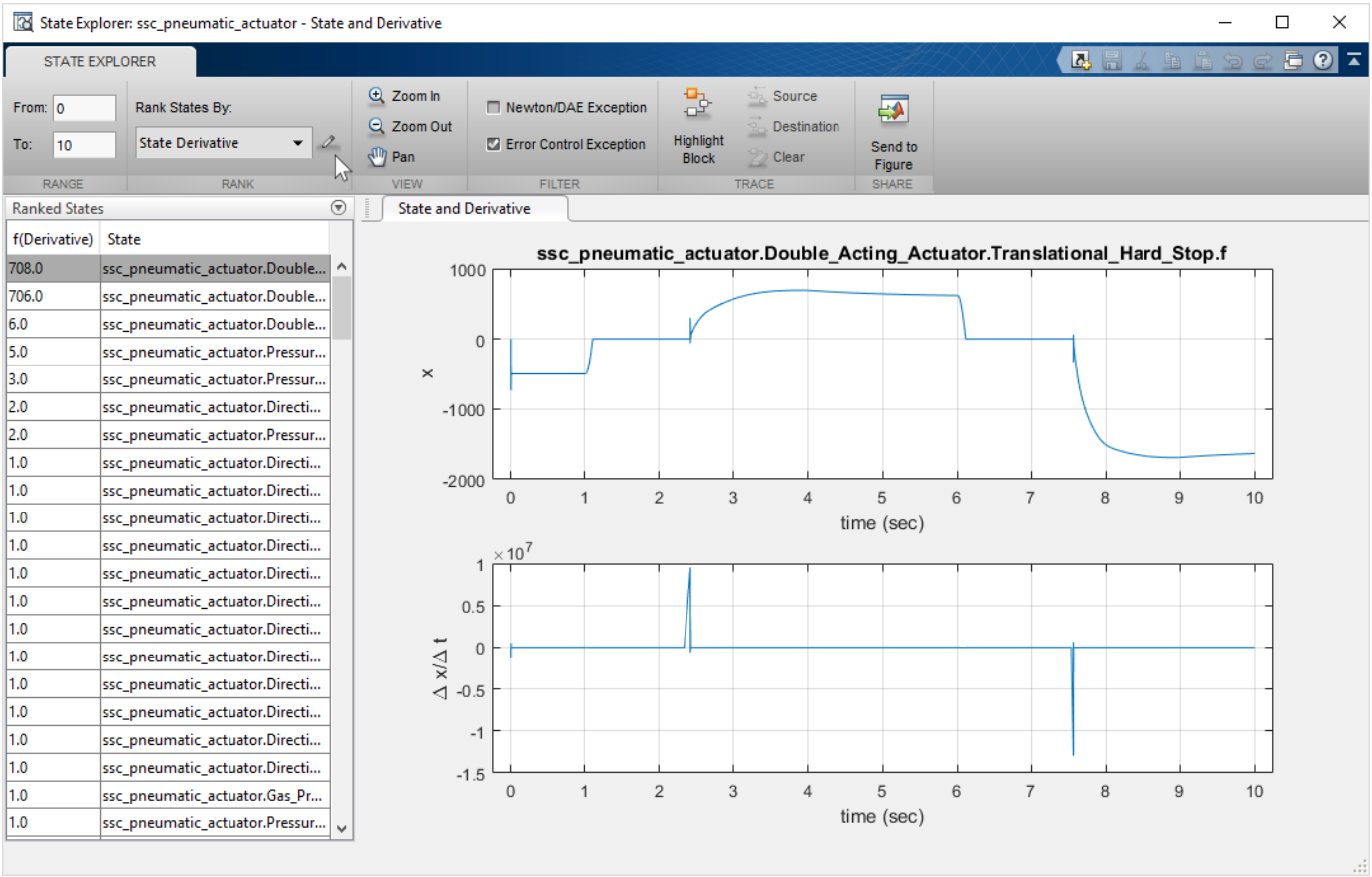
Customize State Ranking

If you log continuous states in the Solver Profiler, you can open the State Explorer to investigate each continuous state. The State Explorer ranks continuous states by the following metrics:

- State Derivative
- Newton/DAE Exception
- State Value
- Error Control Exception
- State Name
- State Chatter

In addition to these ranking metrics, you can write and upload your own algorithm to determine how continuous states are ranked in the State Explorer.

1 Click the edit button next to the **Rank States By** dropdown.



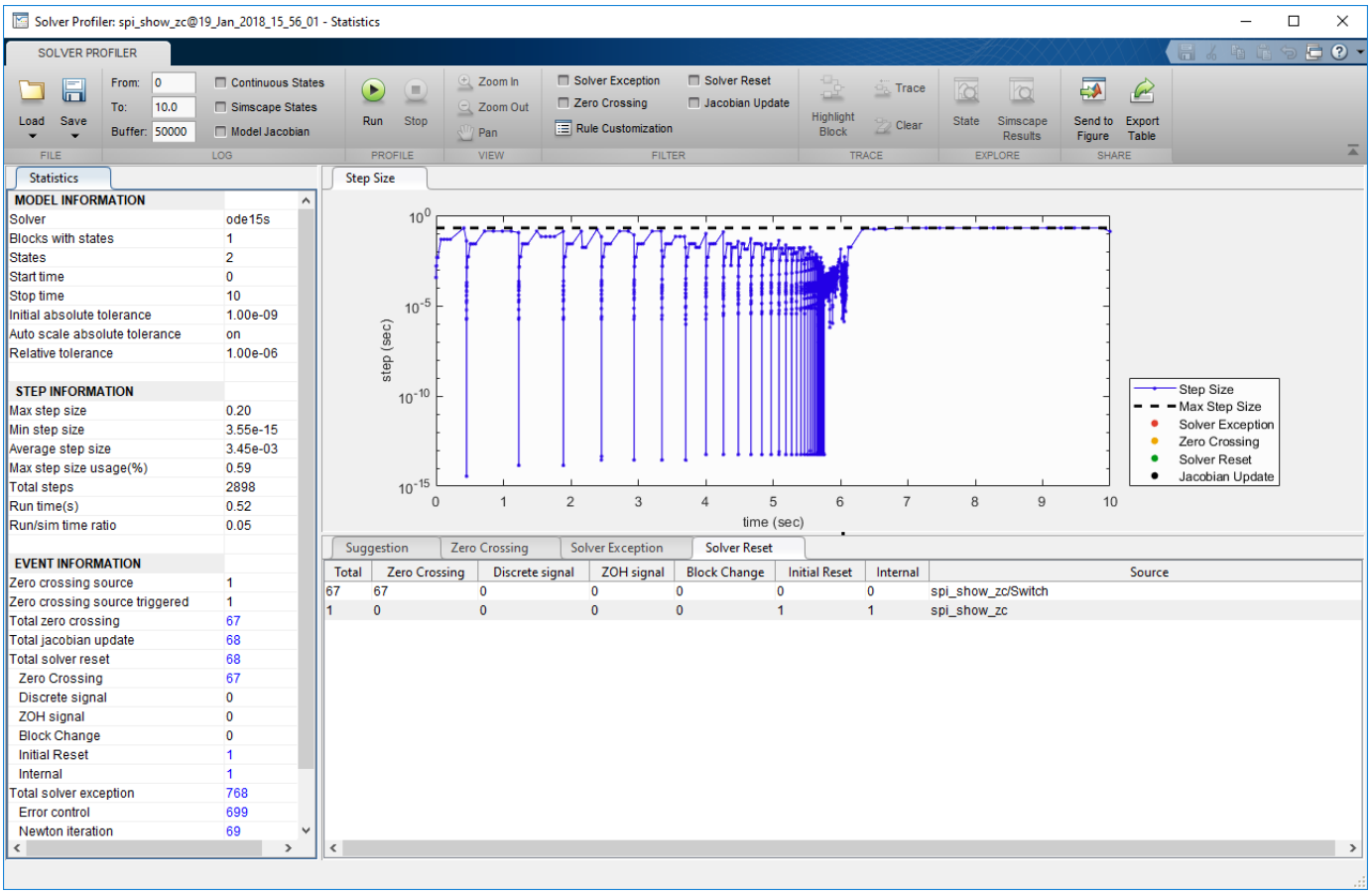
2 In the Custom Algorithm dialog box that appears, click **Add** and upload a MATLAB script that contains your ranking algorithm.

A simple algorithm that ranks states by value looks as follows:

Note The structures referenced in this example organize information that the Solver Profiler collects during a profiling run. For more information on the structures, see “Develop Profiler Rule Set” on page 33-23.

```
function [index,score] = customRank(sd,tl,tr)

% Allocate storage for index and score list
nStates = length(sd.stateInfo);
index = l:nStates;
score = zeros(nStates,l);

% Loop through each state to calculate score
for i = l:nStates
    x = sd.stateInfo(i).value;
    % apply time range constraints
    x = x(sd.tout>=tl & sd.tout<=tr);
    if max(x) > 1
        score(i) = 1;
    else
        score(i) = 0;
    end
end

% Rank the states
[score, order] = sort(score, 'descend');
index = index(order);

end
```

Solver Profiler Interface



The Solver Profiler enables you to:

- Start a profiling session, save a session, or open a saved session.
- Customize and filter the results displayed.
- Interact with the model and trace and highlight the states that contain exceptions.
- Open Simscape Explorer or States Explorer to analyze simulation data further.
- Customize profiler suggestions using your own rules.

Statistics Pane

The statistics pane displays information on model parameters, including:

- **Average step size** — A measure of how fast the solver advances. It is calculated as the total simulation time divided by the number of steps the solver used. It is bounded by the model configuration parameters **Max step size** and **Min step size**.
- **Max step size usage** — The percentage of maximum step sizes used by the solver among all step sizes.

- **Zero crossing** — A solver-specific event that affects model dynamics. During simulation, the solver detects zero crossing. Zero crossing detection incurs computation cost. For more information, see “Zero-Crossing Detection” on page 3-10.
- **Solver reset** — An event that causes the solver to reset its parameters. Solver reset detection incurs computation cost. The solver reset statistics are broken down into **Zero Crossing**, **Discrete Signal**, **ZOH Signal**, **Block Change**, **Initial Reset**, and **Internal** solver reset events. For more information, see “Solver Resets” on page 33-15
- **Solver exception** — An event that renders the solver unable to solve model states to meet accuracy specifications. To solve model states accurately, the solver has to run a few adjusted trials, which incur computation cost.
- **Error control exception** — An event where a solution obtained by the solver has an error that is greater than the tolerance specification.
- **Newton iteration exception** — An event specific to implicit solvers. Newton iterations do not converge after a few trials.
- **Infinite state exception** — An event where one or more states solved by the solver are infinite.
- **DAE newton iteration exception** — An event specific to implicit solvers for Simscape models. The Newton iteration does not converge even though the solver violates the minimum step size constraint.

Suggestions and Exceptions Pane

The suggestions and exceptions pane displays information on exceptions, including:

- **Zero crossing** — A solver-specific event that affects model dynamics. During simulation, the solver detects zero crossing. Zero crossing detection incurs computation cost. For more information, see “Zero-Crossing Detection” on page 3-10.
- **Solver reset** — An event that causes the solver to reset its parameters. Solver reset detection incurs computation cost. The solver reset statistics are broken down into **Zero Crossing**, **Discrete Signal**, **ZOH Signal**, **Block Change**, **Initial Reset**, and **Internal** solver reset events. For more information, see “Solver Resets” on page 33-15
- **Solver exception** — An event that renders the solver unable to solve model states to meet accuracy specifications. To solve model states accurately, the solver has to run a few adjusted trials, which incur computation cost.
- **Error control exception** — An event where a solution obtained by the solver has an error that is greater than the tolerance specification.
- **Newton iteration exception** — An event specific to implicit solvers. Newton iterations do not converge after a few trials.
- **Infinite state exception** — An event where one or more states solved by the solver are infinite.
- **DAE newton iteration exception** — An event specific to implicit solvers for Simscape models. The Newton iteration does not converge even though the solver violates the minimum step size constraint.

See Also

Related Examples

- “Examine Model Dynamics Using Solver Profiler” on page 33-2

- “Understand Profiling Results” on page 33-5

Simulink Debugger

Introduction to the Debugger

With the debugger, you run your simulation method by method. You can stop after each method to examine the execution results. In this way, you can pinpoint problems in your model to specific blocks, parameters, or interconnections.

Note Methods are functions that the Simulink software uses to solve a model at each time step during the simulation. Blocks are made up of multiple methods. “Block execution” in this documentation is shorthand for “block methods execution.” Block diagram execution is a multi-step operation that requires execution of the different block methods in all the blocks in a diagram at various points during the process of solving a model at each time step during simulation, as specified by the simulation loop.

The debugger has both a graphical and a command-line user interface. The graphical interface allows you to access the most commonly used features of the debugger. The command-line interface gives you access to all of the capabilities in the debugger. If you can use either to perform a task, the documentation shows you first how to use the graphical interface, “Debugger Graphical User Interface” on page 34-3, and then the command-line interface, “Debugger Command-Line Interface” on page 34-8.

All functions such as `atrace` and `ashow` can only be used within the debugger.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Run a Simulation Step by Step” on page 34-15

More About

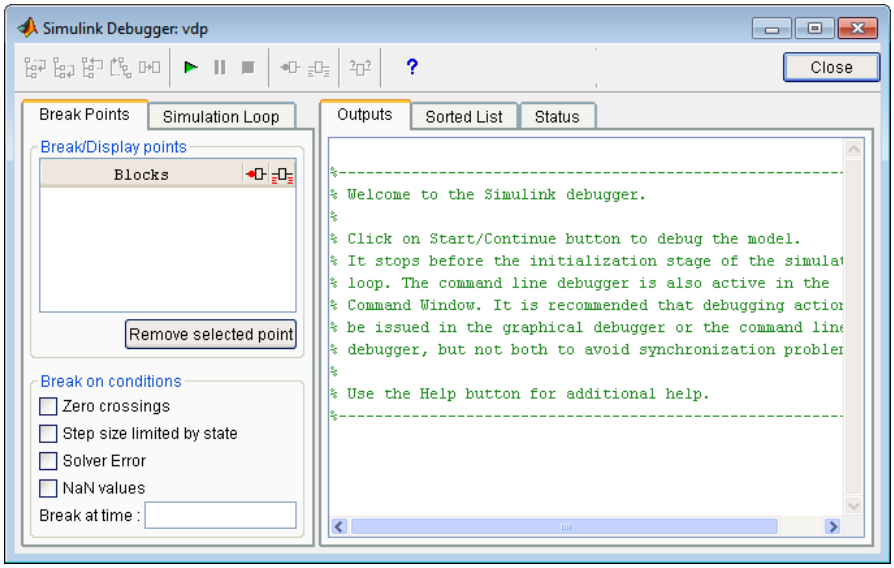
- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Debugger Graphical User Interface

In this section...

Displaying the Graphical Interface

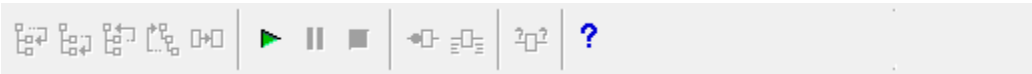
On the **Debug** tab, select **Breakpoints List > Debug Model** to display the debugger graphical interface.



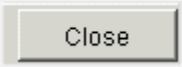
Note The debugger graphical user interface does not display state or solver information. The command line interface does provide this information. See “Display System States” on page 34-26 and “Display Solver Information” on page 34-26.

Toolbar

The debugger toolbar appears at the top of the debugger window.

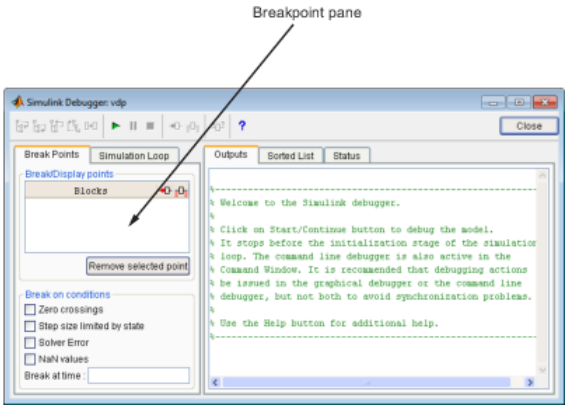


From left to right, the toolbar contains the following command buttons:

Button	Purpose
	Step into the next method (see “Stepping Commands” on page 34-16 for more information on this command, and the following stepping commands).
	Step over the next method.
	Step out of the current method.
	Step to the first method at the start of next time step.
	Step to the next block method.
	Start or continue the simulation.
	Pause the simulation.
	Stop the simulation.
	Break before the selected block.
	Display inputs and outputs of the selected block when executed (same as `trace gcb`).
	Display the current inputs and outputs of selected block (same as `probe gcb`).
	Display help for the debugger.
	Close the debugger.

Breakpoints Pane

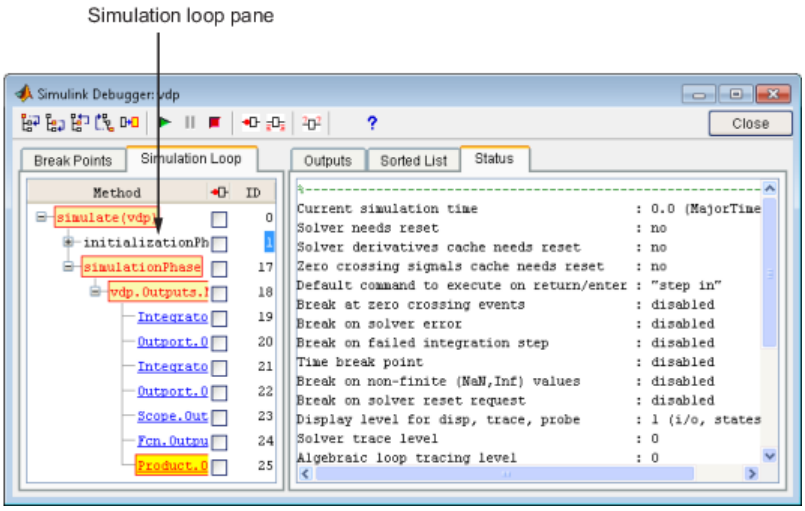
To display the **Breakpoints** pane, select the **Break Points** tab on the debugger window.



The **Breakpoints** pane allows you to specify block methods or conditions at which to stop a simulation. See “Set Breakpoints” on page 34-19 for more information.

Simulation Loop Pane

To display the **Simulation Loop** pane, select the **Simulation Loop** tab on the debugger window.



The **Simulation Loop** pane contains three columns:

- Method
- Breakpoints
- ID

Method Column

The **Method** column lists the methods that have been called thus far in the simulation as a method tree with expandable/collapsible nodes. Each node of the tree represents a method that calls other methods. Expanding a node shows the methods that the block method calls. Clicking a block method name highlights the corresponding block in the model diagram.

Whenever the simulation stops, the debugger highlights the name of the method where the simulation has stopped as well as the methods that invoked it. The highlighted method names indicate the current state of the method call stack.

Breakpoints Column

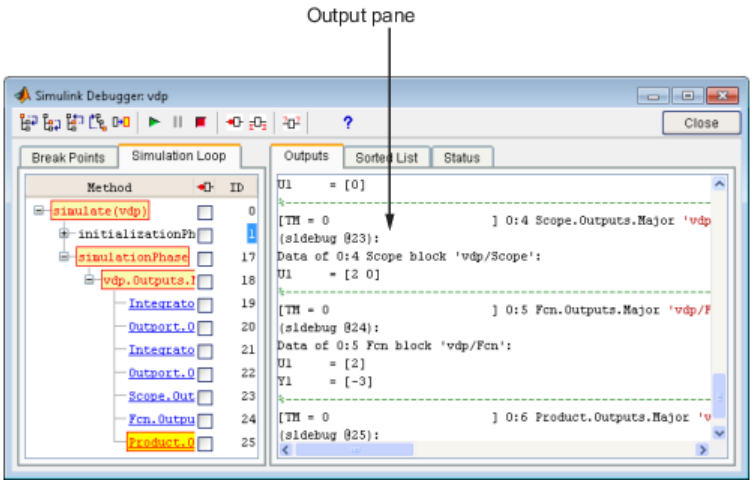
The breakpoints column consists of check boxes. Selecting a check box sets a breakpoint at the method whose name appears to the left of the check box. See “Setting Breakpoints from the Simulation Loop Pane” on page 34-20 for more information.

ID Column

The ID column lists the IDs of the methods listed in the **Methods** column. See “Method ID” on page 34-8 for more information.

Outputs Pane

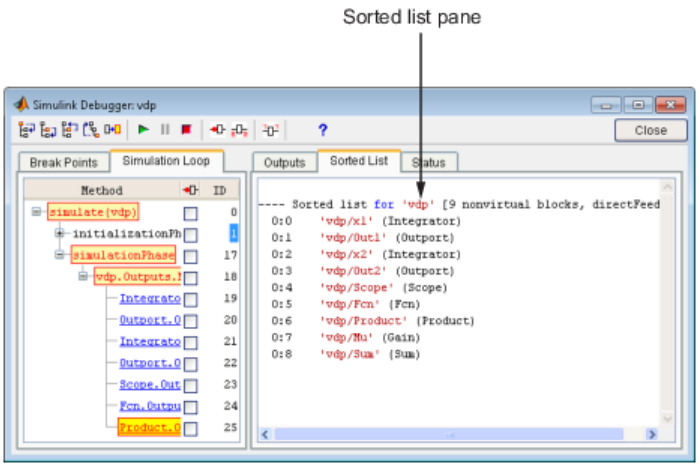
To display the **Outputs** pane, select the **Outputs** tab on the debugger window.



The Outputs pane displays the same debugger output that would appear in the MATLAB command window if the debugger were running in command-line mode. The output includes the debugger command prompt and the inputs, outputs, and states of the block at whose method the simulation is currently paused (see “Block Data Output” on page 34-16). The command prompt displays current simulation time and the name and index of the method in which the debugger is currently stopped (see “Block ID” on page 34-8).

Sorted List Pane

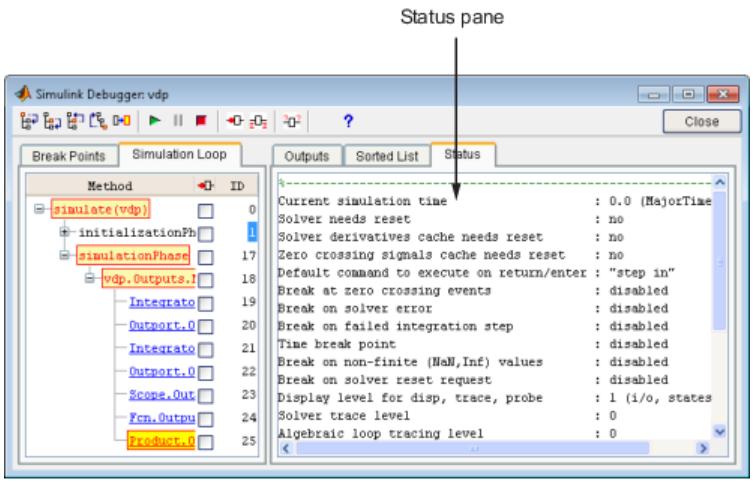
To display the **Sorted List** pane, select the **Sorted List** tab on the debugger window.



The **Sorted List** pane displays the sorted lists for the model being debugged. See “Display Model’s Sorted Lists” on page 34-28 for more information.

Status Pane

To display the **Status** pane, select the **Status** tab on the debugger window.



The **Status** pane displays the values of various debugger options and other status information.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Run a Simulation Step by Step” on page 34-15

More About

- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Debugger Command-Line Interface

In this section...

Controlling the Debugger

In command-line mode, you control the debugger by entering commands at the debugger command line in the MATLAB Command Window. To enter commands at the debugger command line, you must start the debugger programmatically and not through the GUI. Use `sldebug` for this purpose. The debugger accepts abbreviations for debugger commands. For more information on debugger commands, see “Simulink Debugger”.

Note You can repeat some commands by entering an empty command (i.e., by pressing the **Enter** key) at the command line.

Method ID

Some of the Simulink software commands and messages use method IDs to refer to methods. A method ID is an integer assigned to a method the first time the method is invoked. The debugger assigns method IDs sequentially, starting with 0.

Block ID

Some of the debugger commands and messages use block IDs to refer to blocks. Block IDs are assigned to blocks while generating the model's sorted lists during the compilation phase of the simulation. A block ID has the form `sysIdx:blkIdx`, where `sysIdx` is an integer identifying the system that contains the block (either the root system or a nonvirtual subsystem) and `blkIdx` is the position of the block in the system's sorted list. For example, the block ID `0:1` refers to the first block in the model's root system. The `slist` command shows the block ID for each debugged block in the model.

Accessing the MATLAB Workspace

You can enter any MATLAB expression at the `sldebug` prompt. For example, suppose you are at a breakpoint and you are logging time and output of your model as `tout` and `yout`. The following command creates a plot.

```
(sldebug ...) plot(tout, yout)
```

You cannot display the value of a workspace variable whose name is partially or entirely the same as that of a debugger command by entering it at the debugger command prompt. You can, however, use the `eval` command to work around this problem. For example, use `eval('s')` to determine the value of `s` rather than `step` the simulation.

See Also

Related Examples

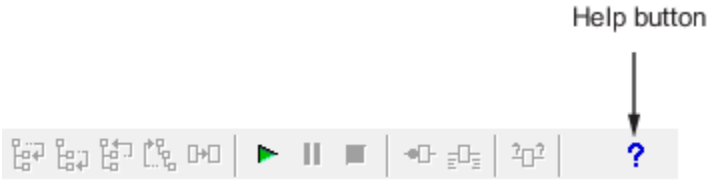
- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Run a Simulation Step by Step” on page 34-15

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Online Help” on page 34-10

Debugger Online Help

You can get online help on using the debugger by clicking the **Help** button on the debugger toolbar. Clicking the **Help** button displays help for the debugger in the MATLAB product Help browser.



In command-line mode, you can get a brief description of the debugger commands by typing `help` at the debug prompt.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Run a Simulation Step by Step” on page 34-15

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8

Start the Simulink Debugger

In this section...

You can start the debugger from either a Simulink model window or from the MATLAB Command Window.

Starting from a Model Window

1. In a model window, on the **Debug** tab, select **Breakpoints List > Debug Model**.

   The debugger graphical user interface opens. See “Debugger Graphical User Interface” on page 34-3.
2. Continue selecting toolbar buttons.

Note When running the debugger in graphical user interface (GUI) mode, you must explicitly start the simulation. For more information, see “Start a Simulation” on page 34-13.

Note When starting the debugger from the GUI, you cannot enter debugger commands in the MATLAB command window. For this, you must start the debugger from the command window using the `sim` or `sldebug` commands.

Starting from the Command Window

1. In the MATLAB Command Window, enter either

   - the `sim` command. For example, enter

     ```
     sim('vdp', 'StopTime', '10', 'debug', 'on')
     ```
   - or the `sldebug` command. For example, enter

     ```
     sldebug 'vdp'
     ```

   In both cases, the example model `vdp` loads into memory, starts the simulation, and stops the simulation at the first block in the model execution list.
2. The debugger opens and a debugger command prompt appears within the MATLAB command window. Continue entering debugger commands at this debugger prompt.

See Also

Related Examples

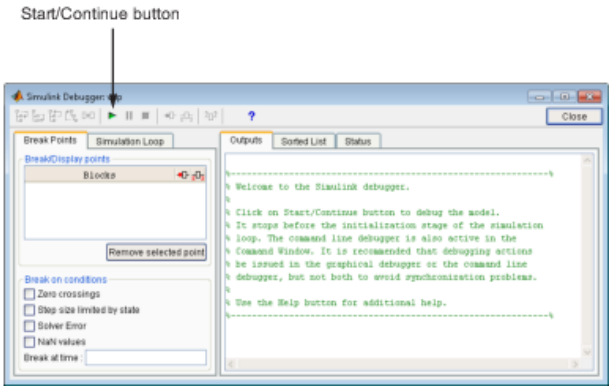
- “Start a Simulation” on page 34-13
- “Run a Simulation Step by Step” on page 34-15
- “Set Breakpoints” on page 34-19

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Start a Simulation

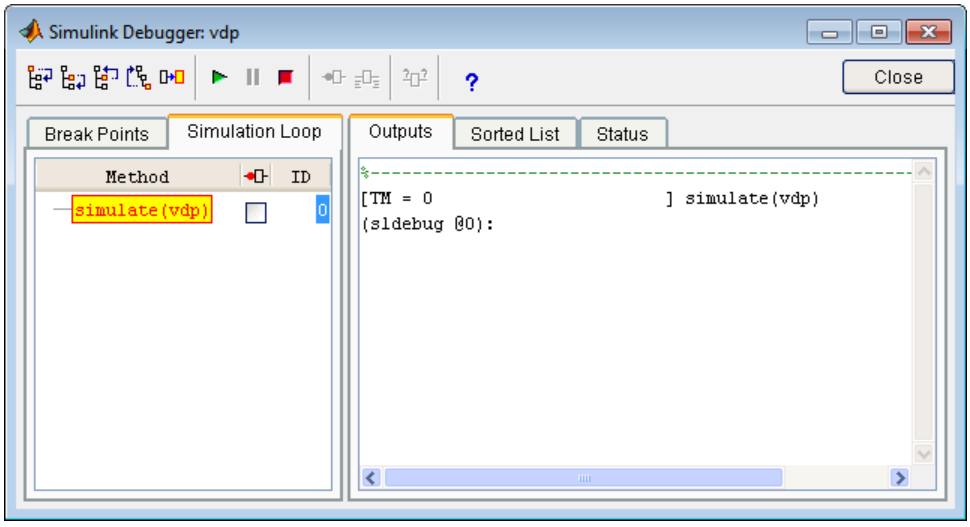
To start the simulation, click the **Start/Continue** button on the debugger toolbar.



The simulation starts and stops at the first simulation method that is to be executed. It displays the name of the method in its **Simulation Loop** pane. At this point, you can

- Set breakpoints.
- Run the simulation step by step.
- Continue the simulation to the next breakpoint or end.
- Examine data.
- Perform other debugging tasks.

The debugger displays the name of the method in the Simulation Loop pane, as shown in the following figure:



The following sections explain how to use the debugger controls to perform these debugging tasks.

Note When you start the debugger in GUI mode, the debugger command-line interface is also active in the MATLAB Command Window. However, to prevent synchronization errors between the graphical and command-line interfaces, you should avoid using the command-line interface.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Run a Simulation Step by Step” on page 34-15
- “Set Breakpoints” on page 34-19
- “Display Information About the Simulation” on page 34-24
- “Display Information About the Model” on page 34-28

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Run a Simulation Step by Step

In this section...

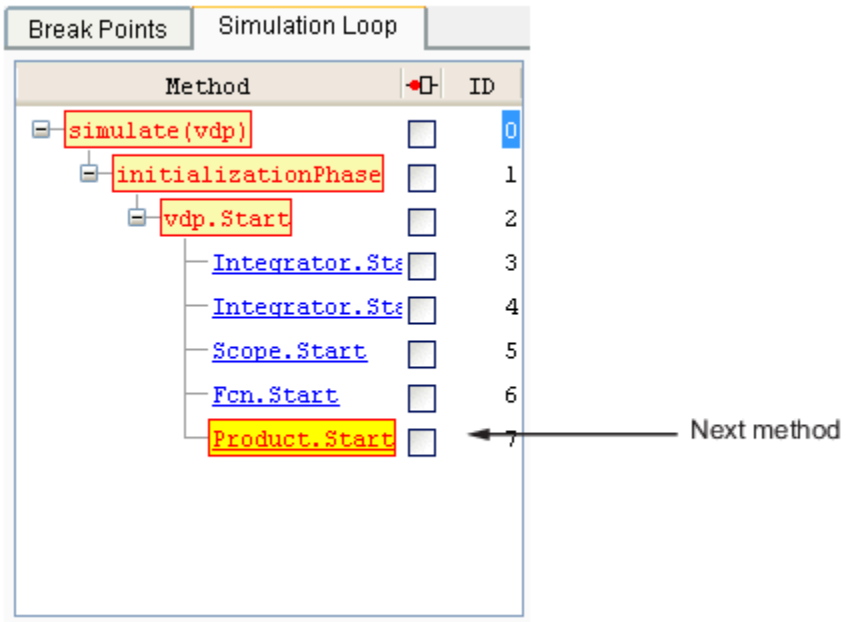
Introduction

The debugger provides various commands that let you advance a simulation from the method where it is currently suspended (the next method) by various increments (see “Stepping Commands” on page 34-16). For example, you can advance the simulation

- Into or over the next method
- Out of the current method
- To the top of the simulation loop.

After each advance, the debugger displays information that enables you to determine the point to which the simulation has advanced and the results of advancing the simulation to that point.

For example, in GUI mode, after each step command, the debugger highlights the current method call stack in the **Simulation Loop** pane. The call stack comprises the next method and the methods that invoked the next method either directly or indirectly. The debugger highlights the call stack by highlighting the names of the methods that make up the call stack in the **Simulation Loop** pane.



In command-line mode, you can use the `where` command to display the method call stack.

Block Data Output

After executing a block method, the debugger prints any or all of the following block data in the debugger **Output** panel (in GUI mode) or, if in command line mode, the MATLAB Command Window:

- `Un = v`

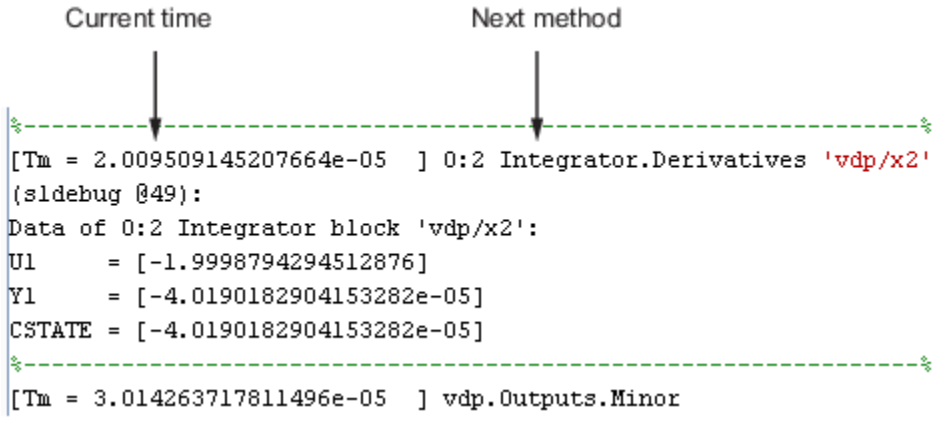
  where `v` is the current value of the block's `n`th input.
- `Yn = v`

  where `v` is the current value of the block's `n`th output.
- `CSTATE = v`

  where `v` is the value of the block's continuous state vector.
- `DSTATE = v`

  where `v` is the value of the block's discrete state vector.

The debugger also displays the current time, the ID and name of the next method to be executed, and the name of the block to which the method applies in the MATLAB Command Window. The following example illustrates typical debugger outputs after a step command.



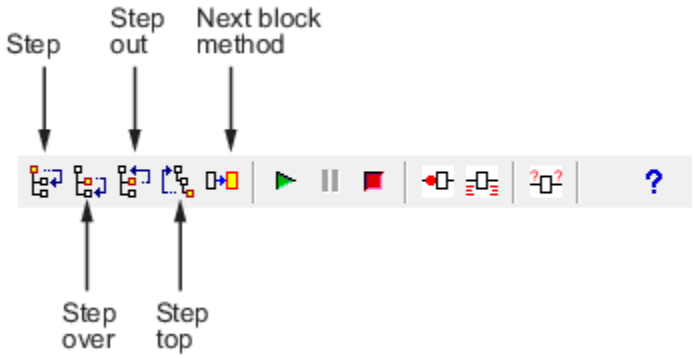
Stepping Commands

Command-line mode provides the following commands for advancing a simulation incrementally:

This command...	Advances the simulation...
`step [in into]`	Into the next method, stopping at the first method in the next method or, if the next method does not contain any methods, at the end of the next method
`step over`	To the method that follows the next method, executing all methods invoked directly or indirectly by the next method
`step out`	To the end of the current method, executing any remaining methods invoked by the current method

This command...	Advances the simulation...
`step top`	To the first method of the next time step (i.e., the top of the simulation loop)
`step blockmth`	To the next block method to be executed, executing all intervening model- and system-level methods
`next`	Same as `step over`

Buttons in the debugger toolbar allow you to access these commands in GUI mode.



Clicking a button has the same effect as entering the corresponding command at the debugger command line.

Continuing a Simulation

In GUI mode, the **Stop** button turns red when the debugger suspends the simulation for any reason. To continue the simulation, click the **Start/Continue** button. In command-line mode, enter `continue` to continue the simulation. By default, the debugger runs the simulation to the next breakpoint (see “Set Breakpoints” on page 34-19) or to the end of the simulation, whichever comes first.

Running a Simulation Nonstop

The `run` command lets you run a simulation to the end of the simulation, skipping any intervening breakpoints. At the end of the simulation, the debugger returns you to the command line. To continue debugging a model, you must restart the debugger.

Note The GUI mode does not provide a graphical version of the `run` command. To run the simulation to the end, you must first clear all breakpoints and then click the **Start/Continue** button.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Set Breakpoints” on page 34-19
- “Display Information About the Simulation” on page 34-24

- “Display Information About the Model” on page 34-28
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Set Breakpoints

In this section...

About Breakpoints

The debugger allows you to define stopping points called breakpoints in a simulation. You can then run a simulation from breakpoint to breakpoint, using the debugger `continue` command. The debugger lets you define two types of breakpoints: unconditional and conditional. An unconditional breakpoint occurs whenever a simulation reaches a method that you specified previously. A conditional breakpoint occurs when a condition that you specified in advance arises in the simulation.

Breakpoints are useful when you know that a problem occurs at a certain point in your program or when a certain condition occurs. By defining an appropriate breakpoint and running the simulation via the `continue` command, you can skip immediately to the point in the simulation where the problem occurs.

Note When you stop a simulation at a breakpoint of a MATLAB S-function in the debugger, to exit MATLAB, you must first quit the debugger.

Setting Unconditional Breakpoints

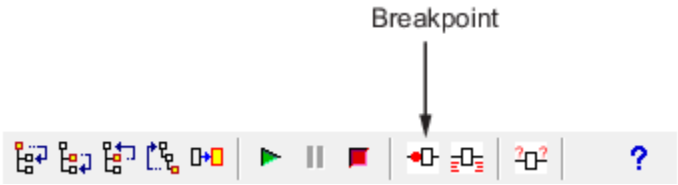
You can set unconditional breakpoints from the:

- Debugger toolbar
- **Simulation Loop** pane
- MATLAB product Command Window (command-line mode only)

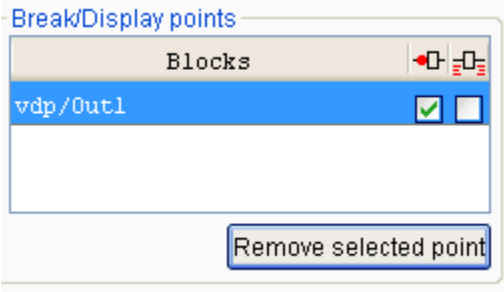
Setting Breakpoints from the Debugger Toolbar

To enable the **Breakpoint** button,

1. Simulate the model.
2. Click the **Step over current method** button until `simulationPhase` is highlighted.
3. Click the **Step into current method** button.



The debugger displays the name of the selected block in the **Break/Display points** panel of the **Breakpoints** pane.



Note Clicking the **Breakpoint** button on the toolbar sets breakpoints on the invocations of a block's methods in major time steps.

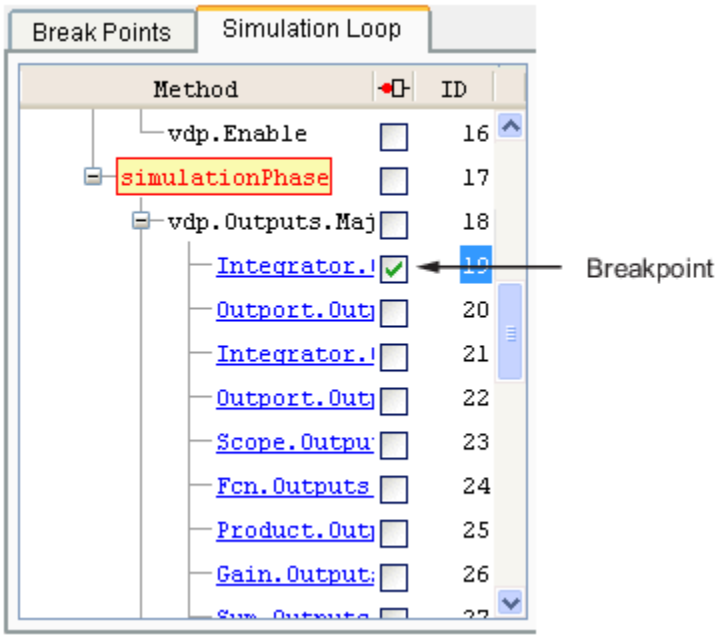
You can temporarily disable the breakpoints on a block by deselecting the check box in the breakpoints column of the panel. To clear the breakpoints on a block and remove its entry from the panel,

1. Select the entry.
2. Click the **Remove selected point** button on the panel.

Note You cannot set a breakpoint on a virtual block. A virtual block is purely graphical: it indicates a grouping or relationship among a model's computational blocks. The debugger warns you if you try to set a breakpoint on a virtual block. You can get a listing of a model's nonvirtual blocks, using the `slist` command (see “Displaying a Model's Nonvirtual Blocks” on page 34-29).

Setting Breakpoints from the Simulation Loop Pane

To set a breakpoint at a particular invocation of a method displayed in the Simulation Loop pane, select the check box next to the method's name in the breakpoint column of the pane.



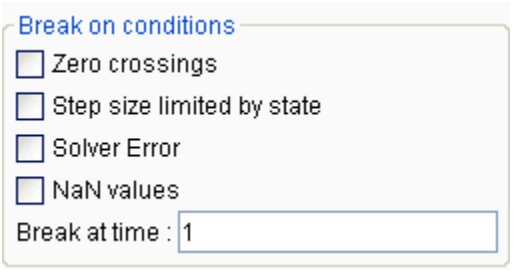
To clear the breakpoint, deselect the check box.

Setting Breakpoints from the Command Window

In command-line mode, use the `break` and `bafter` commands to set breakpoints before or after a specified method, respectively. Use the `clear` command to clear breakpoints.

Setting Conditional Breakpoints

You can use either the **Break on conditions** controls group on the debugger **Breakpoints** pane



or the following commands (in command-line mode) to set conditional breakpoints.

This command...	Causes the Simulation to Stop...
`tbreak [t]`	At a simulation time step
`ebreak`	At a recoverable error in the model
`nanbreak`	At the occurrence of an underflow or overflow (`NaN`) or infinite (`Inf`) value
`xbreak`	When the simulation reaches the state that determines the simulation step size
`zcbreak`	When a zero crossing occurs between simulation time steps

Setting Breakpoints at Time Steps

To set a breakpoint at a time step, enter a time in the debugger **Break at time** field (GUI mode) or enter the time using the `tbreak` command. This causes the debugger to stop the simulation at the `Outputs.Major` method of the model at the first time step that follows the specified time. For example, starting `vdp` in debug mode and entering the commands

```
tbreak 2
continue
```

causes the debugger to halt the simulation at the `vdp.Outputs.Major` method of time step `2.078` as indicated by the output of the `continue` command.

```
%------------------------------------------------------------------%
[Tm = 2.034340153847549      ] vdp.Outputs.Minor
(sldebug @37):
```

Breaking on Nonfinite Values

Selecting the debugger **NaN values** option or entering the `nanbreak` command causes the simulation to stop when a computed value is infinite or outside the range of values that is supported by the machine running the simulation. This option is useful for pinpointing computational errors in a model.

Breaking on Step-Size Limiting Steps

Selecting the **Step size limited by state** option or entering the `xbreak` command causes the debugger to stop the simulation when the model uses a variable-step solver and the solver encounters a state that limits the size of the steps that it can take. This command is useful in debugging models that appear to require an excessive number of simulation time steps to solve.

Breaking at Zero Crossings

Selecting the **Zero crossings** option or entering the `zcbreak` command causes the simulation to halt when a nonsampled zero crossing is detected in a model that includes blocks where zero crossings can arise. After halting, the ID, type, and name of the block in which the zero crossing was detected is displayed. The block ID (`s:b:p`) consists of a system index `s`, block index `b`, and port index `p` separated by colons (see “Block ID” on page 34-8).

For example, setting a zero-crossing break at the start of execution of the `zeroxing` example model,

```
>> sldebug zeroxing
%------------------------------------------------------------------
%
[TM = 0                      ] zeroxing.Simulate
(sldebug @0): >> zcbreak
Break at zero crossing events              : enabled
```

and continuing the simulation

```
(sldebug @0): >> continue
```

results in a zero-crossing break at

```
Interrupting model execution before running mdlOutputs at the left post of
(major time step just before) zero crossing event detected at the following location:
  6[-0]  0:5:2  Saturate  'zeroxing/Saturation'
%------------------------------------------------------------------%
```

```
[TzL= 0.3435011087932808     ] zeroxing.Outputs.Major
(sldebug @16): >>
```

If a model does not include blocks capable of producing nonsampled zero crossings, the command prints a message advising you of this fact.

Breaking on Solver Errors

Selecting the debugger **Solver Errors** option or entering the `ebreak` 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

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Display Information About the Simulation” on page 34-24
- “Display Information About the Model” on page 34-28
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

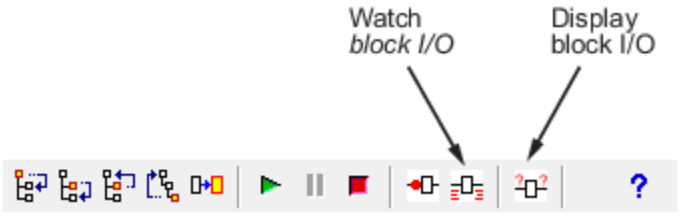
- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Display Information About the Simulation

In this section...

Display Block I/O

The debugger allows you to display block I/O by clicking the appropriate buttons on the debugger toolbar



or by entering the appropriate debugger command.

This command...	Displays a Blocks I/O...
`probe`	Immediately
`disp`	At every breakpoint any time execution stops
`trace`	Whenever the block executes

Note The two debugger toolbar buttons, Watch Block I/O ( ) and Display Block I/O ( ) correspond, respectively, to `trace gcb` and `probe gcb`. The `probe` and `disp` commands do not have a one-to-one correspondence with the debugger toolbar buttons.

Displaying I/O of a Selected Block

To display the I/O of a block, select the block and click in GUI mode or enter the `probe` command in command-line mode. In the following table, the `probe gcb` command has a corresponding toolbar button. The other commands do not.

Command	Description
`probe`	Enter or exit `probe` mode. Typing any command causes the debugger to exit `probe` mode.
`probe gcb`	Display I/O of selected block. Same as .
`probe s:b`	Print the I/O of the block specified by system number `s` and block number `b`.

The debugger prints the current inputs, outputs, and states of the selected block in the debugger **Outputs** pane (GUI mode) or the Command Window of the MATLAB product.

The `probe` command is useful when you need to examine the I/O of a block whose I/O is not otherwise displayed. For example, suppose you are using the `step` command to run a model method by method. Each time you step the simulation, the debugger displays the inputs and outputs of the current block. The `probe` command lets you examine the I/O of other blocks as well.

Displaying Block I/O Automatically at Breakpoints

The `disp` command causes the debugger to display a specified block's inputs and outputs whenever it halts the simulation. You can specify a block by entering its block index and entering `gcb` as the `disp` command argument. You can remove any block from the debugger list of display points, using the `undisp` command. For example, to remove `block 0:0`, enter `undisp 0:0`.

Note Automatic display of block I/O at breakpoints is not available in the debugger GUI mode.

The `disp` command is useful when you need to monitor the I/O of a specific block or set of blocks as you step through a simulation. Using the `disp` command, you can specify the blocks you want to monitor and the debugger will then redisplay the I/O of those blocks on every step. Note that the debugger always displays the I/O of the current block when you step through a model block by block, using the `step` command. You do not need to use the `disp` command if you are interested in watching only the I/O of the current block.

Watching Block I/O

To watch a block, select the block and click  in the debugger toolbar or enter the `trace` command. In GUI mode, if a breakpoint exists on the block, you can set a watch on it as well by selecting the check box for the block in the watch column  of the **Break/Display points** pane. In command-line mode, you can also specify the block by specifying its block index in the `trace` command. You can remove a block from the debugger list of trace points using the `untrace` command.

The debugger displays a watched block's I/O whenever the block executes. Watching a block allows you obtain a complete record of the block's I/O without having to stop the simulation.

Display Algebraic Loop Information

The `atrace` command causes the debugger to display information about a model's algebraic loops (see “Algebraic Loop Concepts” on page 3-27) each time they are solved. The command takes a single argument that specifies the amount of information to display.

This 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 the Jacobian matrix used to solve the loop
`atrace 4`	Level 3 plus intermediate solutions of the loop variable

Display System States

The `states` debug command lists the current values of the system states in the MATLAB Command Window. For example, the following sequence of commands shows the states of the bouncing ball example (`sldemo_bounce`) after its first, second, and third time steps. However, before entering the debugger, open the Configuration Parameters dialog box and clear the **Block reduction** and **Signal storage reuse** check boxes.

```
 sldebug sldemo_bounce
%--------------------------------------------------------------------%
[TM = 0                      ] simulate(sldemo_bounce)
(sldebug @0): >> step top
%--------------------------------------------------------------------%
[TM = 0                      ] sldemo_bounce.Outputs.Major
(sldebug @16): >> next
%--------------------------------------------------------------------%
[TM = 0                      ] sldemo_bounce.Update
(sldebug @23): >> states

Continuous States:
Idx  Value                    (system:block:element  Name   'BlockName')
  0  10                       (0:4:0  CSTATE  'sldemo_bounce/Second-Order  Integrator')
  1. 15                       (0:4:1)

(sldebug @23): >> next
%--------------------------------------------------------------------%
[Tm = 0                      ] solverPhase
(sldebug @26): >> states

Continuous States:
Idx  Value                    (system:block:element  Name   'BlockName')
  0  10                       (0:4:0  CSTATE  'sldemo_bounce/Second-Order  Integrator')
  1. 15                       (0:4:1)

(sldebug @26): >> next
%--------------------------------------------------------------------%
[TM = 0.01                   ] sldemo_bounce.Outputs.Major
(sldebug @16): >> states

Continuous States:
Idx  Value                    (system:block:element  Name   'BlockName')
  0  10.1495095               (0:4:0  CSTATE  'sldemo_bounce/Second-Order  Integrator')
  1. 14.9019                  (0:4:1)
```

Display Solver Information

The `strace` command allows you to pinpoint problems in solving a models differential equations that can slow down simulation performance. Executing this command causes the debugger to display solver-related information at the command line of the MATLAB product when you run or step through a simulation. The information includes the sizes of the steps taken by the solver, the estimated integration error resulting from the step size, whether a step size succeeded (i.e., met the accuracy requirements that the model specifies), the times at which solver resets occur, etc. If you are concerned about the time required to simulate your model, this information can help you to decide whether the solver you have chosen is the culprit and hence whether choosing another solver might shorten the time required to solve the model.

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11

- “Start a Simulation” on page 34-13
- “Set Breakpoints” on page 34-19
- “Display Information About the Model” on page 34-28
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Display Information About the Model

In this section...

Display Model’s Sorted Lists

In GUI mode, the debugger **Sorted List** pane displays lists of blocks for a models root system and each nonvirtual subsystem. Each list lists the blocks that the subsystems contains sorted according to their computational dependencies, alphabetical order, and other block sorting rules. In command-line mode, you can use the `slist` command to display a model's sorted lists.

```
---- Sorted list for 'vdp' [11 nonvirtual block(s), directFeed=0]
     Total number of tasks = 2
- Sorted list of task index [0], 10 nonvirtual block(s)
  (0)0:1   'vdp/x1' (Integrator)
           Input ports:  [0]
           Output ports: [0]
  (0)0:2   'vdp/Out1' (Outport)
           Input ports:  [0]
           Output ports: []
  (0)0:3   'vdp/x2' (Integrator)
           Input ports:  [0]
           Output ports: [0]
  (0)0:4   'vdp/Out2' (Outport)
           Input ports:  [0]
           Output ports: []
  (0)0:5   'vdp/Scope' (Scope)
           Input ports:  [0]
           Output ports: []
  (0)0:6   'vdp/Square' (Math)
           Input ports:  [0]
           Output ports: [0]
  (0)0:7   'vdp/Sum1' (Sum)
           Input ports:  [0 1]
           Output ports: [0]
  (0)0:8   'vdp/Product' (Product)
           Input ports:  [0 1]
           Output ports: [0]
  (0)0:9   'vdp/Mu' (Gain)
           Input ports:  [0]
           Output ports: [0]
  (0)0:10  'vdp/Sum' (Sum)
           Input ports:  [0 1]
           Output ports: [0]

- Sorted list of task index [1], 1 nonvirtual block(s)
  (1)0:1   'vdp/Constant' (Constant)
           Input ports:  []
           Output ports: [0]


 ----- Task Index Legend -----
 Task Index [0]: Cont
```

```
 Task Index [1]: Constant
 ----------------------------
```

These displays include the block index for each command. You can use them to determine the block IDs of the models blocks. Some debugger commands accept block IDs as arguments.

Identifying Blocks in Algebraic Loops

If a block belongs to an algebraic list, the `slist` command displays an algebraic loop identifier in the entry for the block in the sorted list. The identifier has the form

```
algId=s#n
```

where `s` is the index of the subsystem containing the algebraic loop and `n` is the index of the algebraic loop in the subsystem. For example, the following entry for an Integrator block indicates that it participates in the first algebraic loop at the root level of the model.

```
0:1 'test/ss/I1' (Integrator, tid=0) [algId=0#1, discontinuity]
```

When the debugger is running, you can use the `ashow` command at the debugger command-line interface to highlight the blocks and lines that make up an algebraic loop. See “Displaying Algebraic Loops” on page 34-30 for more information.

Display a Block

To determine the block in a models diagram that corresponds to a particular index, enter `bshow s:b` at the command prompt, where `s:b` is the block index. The `bshow` command opens the system containing the block (if necessary) and selects the block in the systems window.

Displaying a Model’s Nonvirtual Systems

The `systems` command displays a list of the nonvirtual systems in the model that you are debugging. For example, the `sldemo_clutch` model contains the following systems:

```
open_system('sldemo_clutch')
set_param(gcs, 'OptimizeBlockIOStorage','off')
sldebug sldemo_clutch
(sldebug @0): %-------------------------------------------------------------------%
[TM = 0                     ] simulate(sldemo_clutch)
(sldebug @0): >> systems
 0   'sldemo_clutch'
 1   'sldemo_clutch/Locked'
 2   'sldemo_clutch/Unlocked'
```

Note The `systems` command does not list subsystems that are purely graphical. That is, subsystems that the model diagram represents as Subsystem blocks but that are solved as part of a parent system. are not listed. In Simulink models, the root system and triggered or enabled subsystems are true systems. All other subsystems are virtual (that is, graphical) and do not appear in the listing from the `systems` command.

Displaying a Model's Nonvirtual Blocks

The `slist` command displays a list of the nonvirtual blocks in a model. The listing groups the blocks by system. For example, the following sequence of commands produces a list of the nonvirtual blocks in the Van der Pol (`vdp`) example model.

```
sldebug vdp
%-------------------------------------------------------------------%
[TM = 0                       ] simulate(vdp)
sldebug @0): >> slist

---- 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)
```

Note The `slist` command does not list blocks that are purely graphical. That is, blocks that indicate relationships between or groupings among computational blocks.

Displaying Blocks with Potential Zero Crossings

The `zclist` command displays a list of blocks in which nonsampled zero crossings can occur during a simulation. For example, `zclist` displays the following list for the clutch sample model:

```
(sldebug @0): >> zclist
  0  0:4:0    F  HitCross  'sldemo_clutch/Friction Mode Logic/Lockup
Detection/Velocities Match'
  1  0:4:1    F
  2  0:10:0    F  Abs  'sldemo_clutch/Friction Mode Logic/Lockup
Detection/Required Friction for Lockup/Abs'
  3  0:12:0    F  RelationalOperator  'sldemo_clutch/Friction Mode
Logic/Lockup Detection/Required Friction for Lockup/Relational Operator'
  4  0:19:0    F  Abs  'sldemo_clutch/Friction Mode Logic/Break Apart
Detection/Abs'
  5  0:20:0    F  RelationalOperator  'sldemo_clutch/Friction Mode
Logic/Break Apart Detection/Relational Operator'
  6  2:3:0    F  Signum  'sldemo_clutch/Unlocked/slip direction'
```

Displaying Algebraic Loops

The `ashow` command highlights a specified algebraic loop or the algebraic loop that contains a specified block. To highlight a specified algebraic loop, enter `ashow s#n`, where `s` is the index of the system (see “Identifying Blocks in Algebraic Loops” on page 34-29) that contains the loop and `n` is the index of the loop in the system. To display the loop that contains the currently selected block, enter `ashow gcb`. To show a loop that contains a specified block, enter `ashow s:b`, where `s:b` is the block's index. To clear algebraic-loop highlighting from the model diagram, enter `ashow clear`.

Displaying Debugger Status

In GUI mode, the debugger displays the settings of various debug options, such as conditional breakpoints, in its **Status** panel. In command-line mode, the `status` command displays debugger settings. For example, the following sequence of commands displays the initial debug settings for the `vdp` model:

```
sim('vdp', 'StopTime', '10', 'debug', 'on')
%-------------------------------------------------------------------%
[TM = 0                       ] simulate(vdp)
(sldebug @0): >> status
%-------------------------------------------------------------------%
```

```
Current simulation time                    : 0.0 (MajorTimeStep)
Solver needs reset                         : no
Solver derivatives cache needs reset       : no
Zero crossing signals cache needs reset    : no
Default command to execute on return/enter : ""
Break at zero crossing events              : disabled
Break on solver error                      : disabled
Break on failed integration step           : disabled
Time break point                           : disabled
Break on non-finite (NaN,Inf) values       : disabled
Break on solver reset request              : disabled
Display level for disp, trace, probe       : 1 (i/o, states)
Solver trace level                         : 0
Algebraic loop tracing level               : 0
Animation Mode                             : off
Execution Mode                             : Normal
Display level for etrace                   : 0 (disabled)
Break points                               : none installed
Display points                             : none installed
Trace points                               : none installed
```

See Also

Related Examples

- “Start the Simulink Debugger” on page 34-11
- “Start a Simulation” on page 34-13
- “Set Breakpoints” on page 34-19
- “Display Information About the Simulation” on page 34-24
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “Debugger Graphical User Interface” on page 34-3
- “Debugger Command-Line Interface” on page 34-8
- “Debugger Online Help” on page 34-10

Accelerating Models

What Is Acceleration?

Acceleration is a mode of operation in the Simulink product that you can use to speed up the execution of your model. The Simulink software includes two modes of acceleration: accelerator mode and the rapid accelerator mode. Both modes replace the normal interpreted code with compiled target code. Using compiled code speeds up simulation of many models, especially those where run time is long compared to the time associated with compilation and checking to see if the target is up to date.

The accelerator mode works with any model, but performance decreases if a model contains blocks that do not support acceleration. The accelerator mode supports the Simulink debugger and profiler. These tools help with debugging and determining relative performance of various parts of your model. For more information, see “Run Accelerator Mode with the Simulink Debugger” on page 35-25 and “How Profiler Captures Performance Data” on page 31-5.

The rapid accelerator mode works with only those models containing blocks that support code generation of a standalone executable. For this reason, rapid accelerator mode does not support the debugger or profiler. However, this mode generally results in faster execution than the accelerator mode. When used with dual-core processors, the rapid accelerator mode runs Simulink and the MATLAB technical computing environment from one core while the rapid accelerator target runs as a separate process on a second core.

For more information about the performance characteristics of the Accelerator and rapid accelerator modes, and how to measure the difference in performance, see “Comparing Performance” on page 35-27.

To optimize your model and achieve faster simulation automatically using Performance Advisor, see “Automated Performance Optimization”.

To employ modeling techniques that help achieve faster simulation, see “Manual Performance Optimization”.

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19

More About

- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10
- “Comparing Performance” on page 35-27

How Acceleration Modes Work

In this section...

Overview

The Accelerator and rapid accelerator modes use portions of the Simulink Coder product to create an executable.

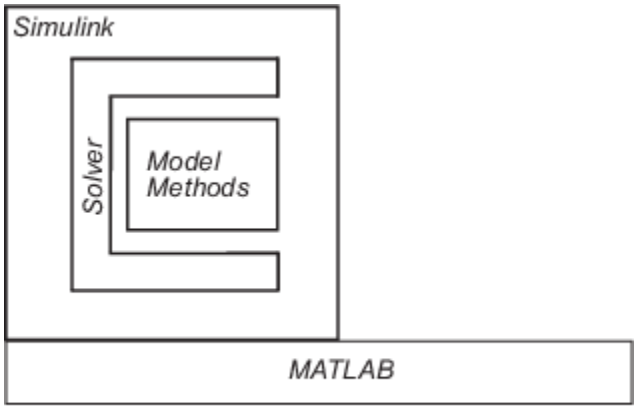
The Accelerator and rapid accelerator modes replace the interpreted code normally used in Simulink simulations, shortening model run time.

Although the acceleration modes use some Simulink Coder code generation technology, you do not need the Simulink Coder software installed to accelerate your model.

Note The code generated by the accelerator and rapid accelerator modes is suitable only for speeding the simulation of your model. Use Simulink Coder to generate code for other purposes.

Normal Mode

In normal mode, the MATLAB technical computing environment is the foundation on which the Simulink software is built. Simulink controls the solver and model methods used during simulation. Model methods include such things as computation of model outputs. Normal mode runs in one process.



One Process

Accelerator Mode

By default, the accelerator mode uses Just-in-Time (JIT) acceleration to generate an execution engine in memory instead of generating C code or MEX files. You can also have your model fall back to the classic accelerator mode, in which Simulink generates and links code into a C-MEX S-function.

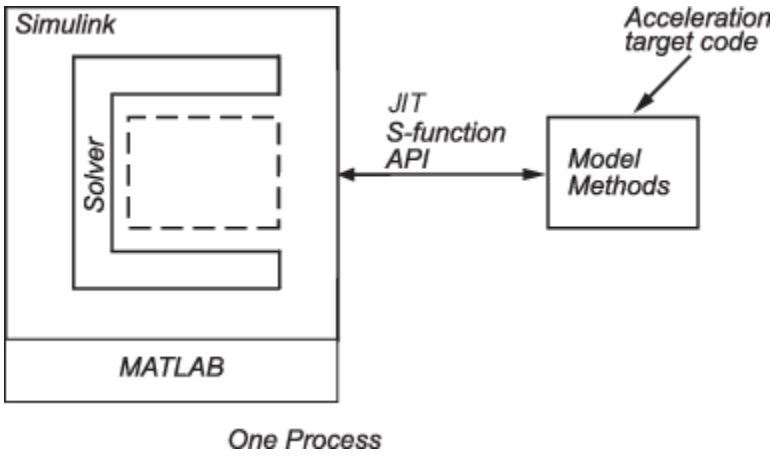
In the accelerator mode, the model methods are separate from the Simulink software and are part of the *acceleration target code*, which is used in later simulations.

Simulink checks that the acceleration target code is up to date before reusing it. For more information, see “Code Regeneration in Accelerated Models” on page 35-7 .

There are two modes of operation in accelerator mode.

Just-In-Time Accelerator Mode

In this default mode, Simulink generates an execution engine in memory for the top-level model only and not for referenced models. As a result, a C compiler is not required during simulation.



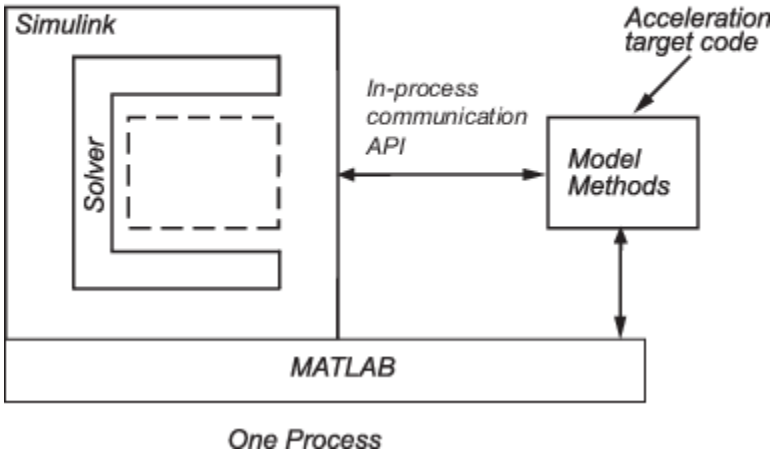
Because the acceleration target code is in memory, it is available for reuse as long as the model is open. Simulink also serializes the acceleration target code so that the model does not need rebuilding when it is opened.

Classic Accelerator Mode

If you want to simulate your model using the classic, C-code generating, accelerator mode, run the following command:

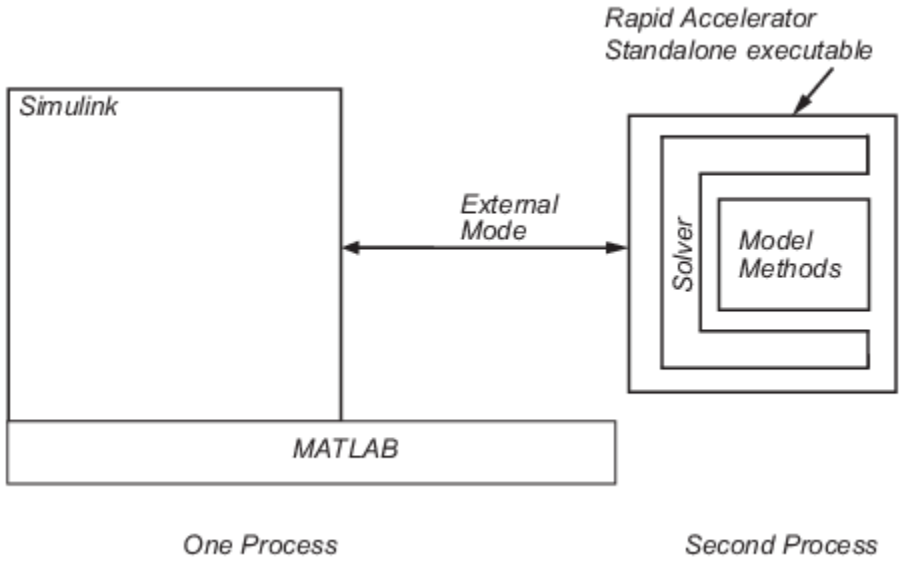
```
set_param(0, 'GlobalUseClassicAccelMode', 'on');
```

In this mode, Simulink generates and links code into a shared library, which communicates with the Simulink software. The target code executes in the same process as MATLAB and Simulink.



Rapid Accelerator Mode

The rapid accelerator mode creates a *Rapid Accelerator standalone executable* from your model. This executable includes the solver and model methods, but it resides outside of MATLAB and Simulink. It uses external mode (see “External Mode Communication” (Simulink Coder)) to communicate with Simulink.



MATLAB and Simulink run in one process, and if a second processing core is available, the standalone executable runs there.

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19

More About

- “Choosing a Simulation Mode” on page 35-10
- “Code Regeneration in Accelerated Models” on page 35-7
- “Comparing Performance” on page 35-27

Code Regeneration in Accelerated Models

In this section...

Changing the structure of your model causes the Rapid Accelerator mode to regenerate the standalone executable, and for the Accelerator mode to regenerate the target code and update (overwrite) the existing MEX-file. Changing the value of a tunable parameter does not trigger a rebuild.

Determine If the Simulation Will Rebuild

The Accelerator and Rapid Accelerator modes use a checksum to determine if the model has changed, indicating that the code should be regenerated. The checksum is an array of four integers computed using an MD5 checksum algorithm based on attributes of the model and the blocks it contains.

1. Use the `Simulink.BlockDiagram.getChecksum` command to obtain the checksum for your model. For example:

   ```
   cs1 = Simulink.BlockDiagram.getChecksum('myModel');
   ```

2. Obtain a second checksum after you have altered your model. The code regenerates if the new checksum does not match the previous checksum.
3. Use the information in the checksum to determine why the simulation target rebuilt.

For a detailed explanation of this procedure, see the example model `slAccelDemoWhyRebuild`.

Parameter Tuning in Rapid Accelerator Mode

In model rebuilds, Rapid Accelerator Mode handles block diagram and runtime parameters differently from other parameters.

Tuning Block Diagram Parameters

You can change some block diagram parameters during simulation without causing a rebuild. Tune these parameters using the `set_param` command or using the **Model Configuration Parameters** dialog box. These block diagram parameters include:

Solver Parameters		
AbsTol	MaxNumMinSteps	RelTol
ConsecutiveZCsStepRelTol	MaxOrder	StartTime
ExtrapolationOrder	MaxStep	StopTime
InitialStep	MinStep	ZCDetectionTol
MaxConsecutiveMinStep	OutputTimes	
MaxConsecutiveZCs	Refine	

Loading and Logging Parameters		
ConsistencyChecking	MinStepSizeMsg	SaveTime
Decimation	OutputOption	StateSaveName
FinalStateName	OutputSaveName	TimeSaveName
LimitDataPoints	SaveFinalState	
LoadExternalInput	SaveFormat	
MaxConsecutiveZCsMsg	SaveOutput	
MaxDataPoints	SaveState	

Tuning Runtime Parameters

To tune runtime parameters for maximum acceleration in Rapid Accelerator mode, follow this procedure which yields better results than using `set_param` for the same purpose:

1. Collect the runtime parameters in a runtime parameter structure while building a rapid accelerator target executable using the `Simulink.BlockDiagram.buildRapidAcceleratorTarget` function.
2. To change the parameters, use the `Simulink.BlockDiagram.modifyTunableParameters` function.
3. To specify the modified parameters to the `sim` command, use the `RapidAcceleratorParameterSets` and `RapidAcceleratorUpToDateCheck` parameters.

All other parameter changes can necessitate a rebuild of the model.

Parameter Changes	Passed Directly to `sim` command	Passed Graphically via Block Diagram or via `set_param` command
Runtime	Does not require rebuild	Can require rebuild
Block diagram (logging parameters)	Does not require rebuild	Does not require rebuild

For information about parameter tunability limitations with accelerated simulation modes, see “Tunability Considerations and Limitations for Other Modeling Goals” on page 37-36.

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19
- “How to Improve Performance in Acceleration Modes” on page 35-30
- “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38

More About

- “What Is Acceleration?” on page 35-2

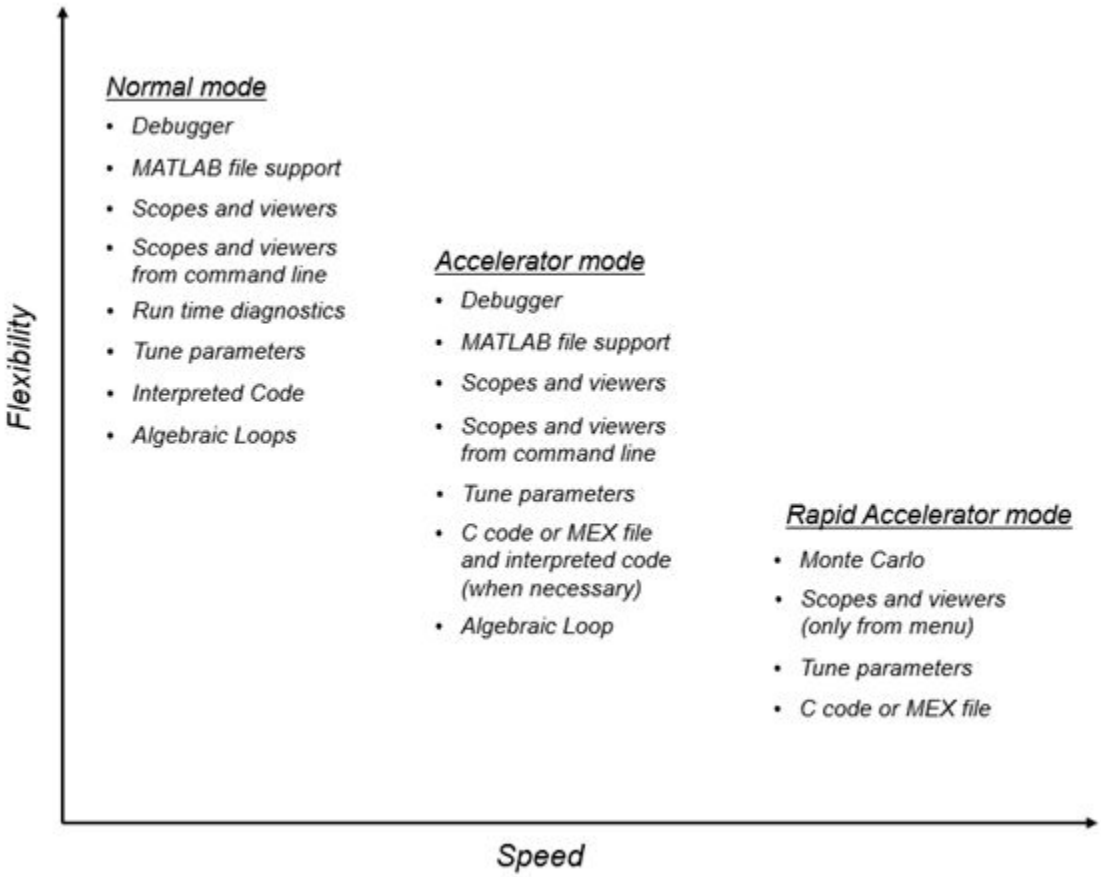
- “Choosing a Simulation Mode” on page 35-10
- “How Acceleration Modes Work” on page 35-3
- “Comparing Performance” on page 35-27

Choosing a Simulation Mode

In this section...

Simulation Mode Tradeoffs

In general, you must trade off simulation speed against flexibility when choosing either Accelerator mode or Rapid Accelerator mode instead of Normal mode.



Normal mode offers the greatest flexibility for making model adjustments and displaying results, but it runs the slowest.

Accelerator mode lies between Normal and Rapid Accelerator modes in performance and in interaction with your model. Accelerator mode does not support most runtime diagnostics.

Rapid Accelerator mode runs the fastest, but this mode does not support the debugger or profiler, and works only with those models for which C code or MEX file is available for all of the blocks in the model.

Note An exception to this rule occurs when you run multiple simulations, each of which executes in less than one second in Normal mode. For example:

```
for i=1:100
sim(model); % executes in less than one second in Normal mode
end
```

For this set of conditions, you will typically obtain the best performance by simulating the model in Normal mode.

Tip To gain additional flexibility, consider using model referencing to componentize your model. If the top-level model uses Normal mode, then you can simulate a referenced model in a different simulation mode than you use for other portions of a model. During the model development process, you can choose different simulation modes for different portions of a model. For details, see “Choose Simulation Modes for Model Hierarchies” on page 8-39.

Comparing Modes

The following table compares the characteristics of Normal mode, Accelerator mode, and Rapid Accelerator mode.

If you want to...	Then use this mode...		
	Normal	Accelerator	Rapid Accelerator
Performance			
Run your model in a separate address space			✓
Efficiently run batch and Monte Carlo simulations			✓
Model Adjustment			
Change model parameters such as solver, stop time without rebuilding	✓	✓	✓
Change block tunable parameters such as gain	✓	✓	✓
For more information on configuration set parameters which can be modified without requiring rebuild, see “Code Regeneration in Accelerated Models” on page 35-7			
Model Requirement			
Accelerate your model even if C code or MEX file are not used for all blocks		✓	
Support Interpreted MATLAB Function blocks	✓	✓	
Support Non-Inlined MATLAB language or Fortran S-Functions	✓	✓	
Permit algebraic loops in your model	✓	✓	
Have your model work with the debugger or profiler	✓	✓	
Have your model include C++ code	✓	✓	
Data Display			

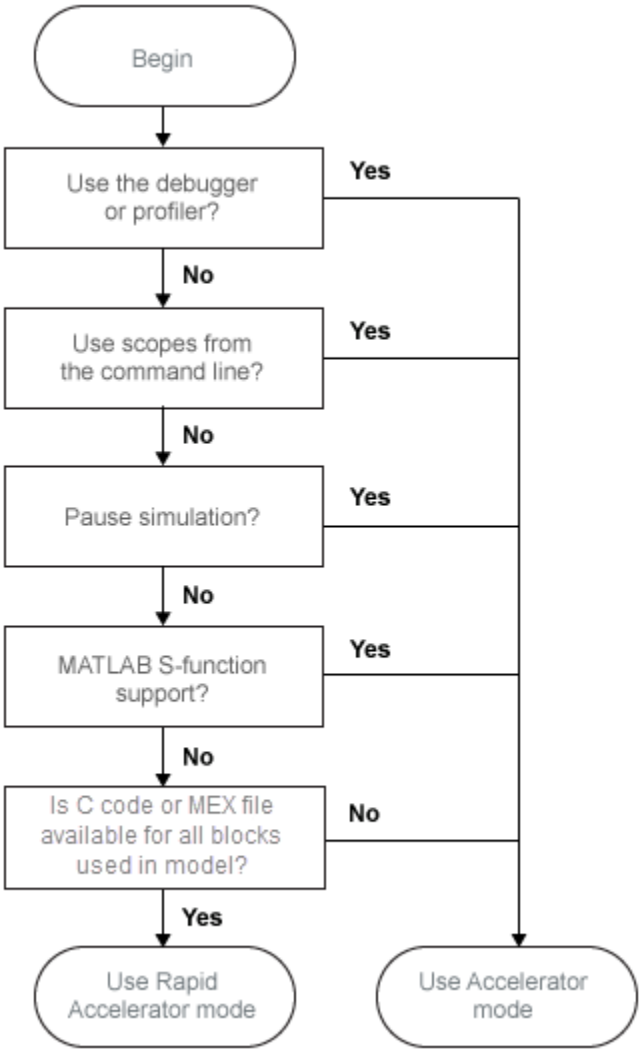
If you want to...	Then use this mode...		
	Normal	Accelerator	Rapid Accelerator
Use scopes and signal viewers	✓	✓	See “Behavior of Scopes and Viewers with Rapid Accelerator Mode” on page 35-16
Use scopes and signal viewers when running your model from the command line	✓	✓	

Note Scopes and viewers do not update if you run your model from the command line in Rapid Accelerator mode.

Decision Tree

Use this decision tree to select between Accelerator, or Rapid Accelerator modes.

See “Comparing Performance” on page 35-27 to understand how effective the accelerator modes will be in improving the performance of your model.



See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Interact with the Acceleration Modes Programmatically” on page 35-22

More About

- “Code Regeneration in Accelerated Models” on page 35-7
- “How Acceleration Modes Work” on page 35-3

Design Your Model for Effective Acceleration

In this section...

Select Blocks for Accelerator Mode

The Accelerator simulation mode runs the following blocks as if you were running Normal mode because these blocks do not generate code for the accelerator build. Consequently, if your model contains a high percentage of these blocks, the Accelerator mode may not increase performance significantly. All of these Simulink blocks use interpreted code.

- Display
- From File
- From Workspace
- Inport (root level only)
- Interpreted MATLAB Function
- Outport (root level only)
- Scope
- To File
- To Workspace
- XY Graph

Note In some instances, Normal mode output might not precisely match the output from Accelerator mode because of slight differences in the numerical precision between the interpreted and compiled versions of a model.

The following blocks can cause poor simulation runtime performance when run in the default JIT Accelerator mode.

- Transport Delay
- Variable Transport Delay

Select Blocks for Rapid Accelerator Mode

Blocks that do not support code generation (such as SimEvents) or blocks that generate code only for a specific target cannot be simulated in Rapid Accelerator mode.

Additionally, Rapid Accelerator mode does not work if your model contains any of the following blocks:

- Interpreted MATLAB Function
- Device driver S-functions, such as blocks from the Simulink Real-Time product, or those targeting Freescale™ MPC555

Note In some instances, Normal mode output might not precisely match the output from Rapid Accelerator mode because of slight differences in the numerical precision between the interpreted and compiled versions of a model.

Control S-Function Execution

Note In the default JIT Accelerator mode, inlining of user-written TLC S-Functions is not supported. If you run a model containing TLC S-Functions in the JIT Accelerator mode, there is a possibility of the execution speed reducing. The code generation speed, however, will be high due to JIT acceleration.

Inlining S-functions using the Target Language Compiler increases performance with the classic Accelerator mode by eliminating unnecessary calls to the Simulink API. By default, however, the classic Accelerator mode ignores an inlining TLC file for an S-function, even though the file exists. The Rapid Accelerator mode always uses the TLC file if one is available.

A device driver S-Function block written to access specific hardware registers on an I/O board is one example of why this behavior was chosen as the default. Because the Simulink software runs on the host system rather than the target, it cannot access the targets I/O registers and so would fail when attempting to do so.

To direct the classic Accelerator mode to use the TLC file instead of the S-function MEX-file, specify `SS_OPTION_USE_TLC_WITH_ACCELERATOR` in the `mdlInitializeSizes` function of the S-function, as in this example:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* Code deleted */
ssSetOptions(S, SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}
```

The Rapid Accelerator mode will make use of the MEX file if the S-Function’s C file is not present in the same folder.

Note to use the .c or .cpp code for your S-Function, ensure that they are in the same folder as the S-Function MEX-file, otherwise, you can include additional files to an S-function or bypass the path limitation by using the rtwmakecfg.m file. For more information, see Use rtwmakecfg.m API to Customize Generated Makefiles (Simulink Coder).

Accelerator and Rapid Accelerator Mode Data Type Considerations

- Accelerator mode supports fixed-point signals and vectors up to 128 bits.
- Rapid Accelerator mode supports fixed-point parameters up to 128 bits.

- Rapid Accelerator mode supports fixed-point root inputs up to 32 bits
- Rapid Accelerator mode supports root inputs of Enumerated data type
- Rapid Accelerator mode does not support fixed-point data for the From Workspace block.
- Rapid Accelerator mode ignores the selection of the **Log fixed-point data as a fi object** (FixptAsFi) check box for the To Workspace block.
- Rapid Accelerator mode supports bus objects as parameters.
- The Accelerator mode and Rapid Accelerator mode store integers as compactly as possible.
- Fixed-Point Designer does not collect min, max, or overflow data in the Accelerator or Rapid Accelerator modes.
- Accelerator mode supports a limited set of runtime diagnostics, including the assertion block.
- Rapid Accelerator mode supports a limited set of runtime diagnostics, including the assertion block.

Behavior of Scopes and Viewers with Rapid Accelerator Mode

Running the simulation from the command line or the menu determines the behavior of scopes and viewers in Rapid Accelerator mode.

Scope or Viewer Type	Simulation Run from Menu	Simulation Run from Command Line
Simulink Scope blocks	Same support as Normal mode	• Logging is supported • Scope window is not updated
Simulink signal viewer scopes	Graphics are updated, but logging is not supported	Not supported
Other signal viewer scopes	Support limited to that available in External mode	Not supported
Signal logging	Supported, with limitations listed in “Signal Logging in Rapid Accelerator Mode” on page 72-42	Supported, with limitations listed in “Signal Logging in Rapid Accelerator Mode” on page 72-42.
Multirate signal viewers	Not supported	Not supported
Stateflow Chart blocks	Same support for chart animation as Normal mode	Not supported

Rapid Accelerator mode does not support multirate signal viewers such as the DSP System Toolbox spectrum scope or the Communications Toolbox™ scatterplot, signal trajectory, or eye diagram scopes.

Note Although scopes and viewers do not update when you run Rapid Accelerator mode from the command line, they do update when you run the model from the menu. “Run Acceleration Mode from the User Interface” on page 35-19 shows how to run Rapid Accelerator mode from the menu. “Interact with the Acceleration Modes Programmatically” on page 35-22 shows how to run the simulation from the command line.

Factors Inhibiting Acceleration

- You cannot use the Accelerator or Rapid Accelerator mode if your model:
  - Passes array parameters to MATLAB S-functions that are not numeric, logical, or character arrays, are sparse arrays, or that have more than two dimensions.
  - Uses Fcn blocks containing trigonometric functions having complex inputs.
- In some cases, changes associated with external or custom code do not cause Accelerator or Rapid Accelerator simulation results to change. These include:
  - TLC code
  - S-function source code, including rtwmakecfg.m files
  - Integrated custom code
  - S-Function Builder

  In such cases, consider force regeneration of code for a top model. Alternatively, you can force regeneration of top model code by deleting code generation folders, such as slprj or the generated model code folder.

  Note With JIT acceleration, the acceleration target code is in memory. It is therefore available for reuse as long as the model is open, even if you delete the `slprj` folder.

Rapid Accelerator Mode Limitations

- Rapid Accelerator mode does not support:
  - Algebraic loops.
  - Targets written in C++.
  - Interpreted MATLAB Function blocks.
  - Noninlined MATLAB language or Fortran S-functions. You must write S-functions in C or inline them using the Target Language Compiler (TLC) or you can also use the MEX file. For more information, see “Write Fully Inlined S-Functions” (Simulink Coder).
  - Debugger or Profiler.
  - Run time objects for Simulink.RunTimeBlock and Simulink.BlockCompOutputPortData blocks.
- Model parameters must be one of these data types:
  - `boolean`
  - `uint8` or `int8`
  - `uint16` or `int16`
  - `uint32` or `int32`
  - `single` or `double`
  - Fixed-point
  - Enumerated
- You cannot pause a simulation in Rapid Accelerator mode.
- If a Rapid Accelerator build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.

- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In these cases, you must regenerate the code for the model. See “Code Regeneration in Accelerated Models” on page 35-7 for more information.
- For root inports, Rapid Accelerator mode supports only base as the `Srcworkspace`.
- For root inports, when you specify the minimum and maximum values that the block should output, Rapid Accelerator mode does not recognize these limits during simulation.
- In Rapid Accelerator mode, To File or To Workspace blocks inside function-call subsystems do not generate any logging files if the function-call port is connected to Ground or unconnected.
- Rapid Accelerator mode does not support systems that run RHEL / CentOS 6.x or 7.x.

Reserved Keywords

Certain words are reserved for use by the Simulink Coder code language and by Accelerator mode and Rapid Accelerator mode. These keywords must not appear as function or variable names on a subsystem, or as exported global signal names. Using the reserved keywords results in the Simulink software reporting an error, and the model cannot be compiled or run.

The keywords reserved for the Simulink Coder product are listed in “Construction of Generated Identifiers” (Simulink Coder). Additional keywords that apply only to the Accelerator and Rapid accelerator modes are:

muDoubleScalarAbs	muDoubleScalarCos	muDoubleScalarMod
muDoubleScalarAcos	muDoubleScalarCosh	muDoubleScalarPower
muDoubleScalarAcosh	muDoubleScalarExp	muDoubleScalarRound
muDoubleScalarAsin	muDoubleScalarFloor	muDoubleScalarSign
muDoubleScalarAsinh	muDoubleScalarHypot	muDoubleScalarSin
muDoubleScalarAtan,	muDoubleScalarLog	muDoubleScalarSinh
muDoubleScalarAtan2	muDoubleScalarLog10	muDoubleScalarSqrt
muDoubleScalarAtanh	muDoubleScalarMax	muDoubleScalarTan
muDoubleScalarCeil	muDoubleScalarMin	muDoubleScalarTanh

See Also

Related Examples

- “Perform Acceleration” on page 35-19
- “How to Improve Performance in Acceleration Modes” on page 35-30

More About

- “What Is Acceleration?” on page 35-2
- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10

Perform Acceleration

In this section...

Customize the Build Process

Compiler optimizations are off by default. This results in faster build times, but slower simulation times. You can optimize the build process toward a faster simulation.

1. From the **Simulation** menu, select **Model Configuration Parameters**.
2. In the Configuration Parameters dialog box, from the **Compiler optimization level** drop-down list, select `Optimizations on (faster runs)`.

   Code generation takes longer with this option, but the model simulation runs faster.
3. Select **Verbose accelerator builds** to display progress information using code generation, and to see the compiler options in use.

Changing the Location of Generated Code

By default, the Accelerator mode places the generated code in a subfolder of the working folder called `slprj/accel/`*modelname* (for example, `slprj/accel/f14`). To change the name of the folder into which the Accelerator Mode writes generated code:

1. In the Simulink Editor window, on the **Modeling** tab, select **Environment > Simulink Preferences**.

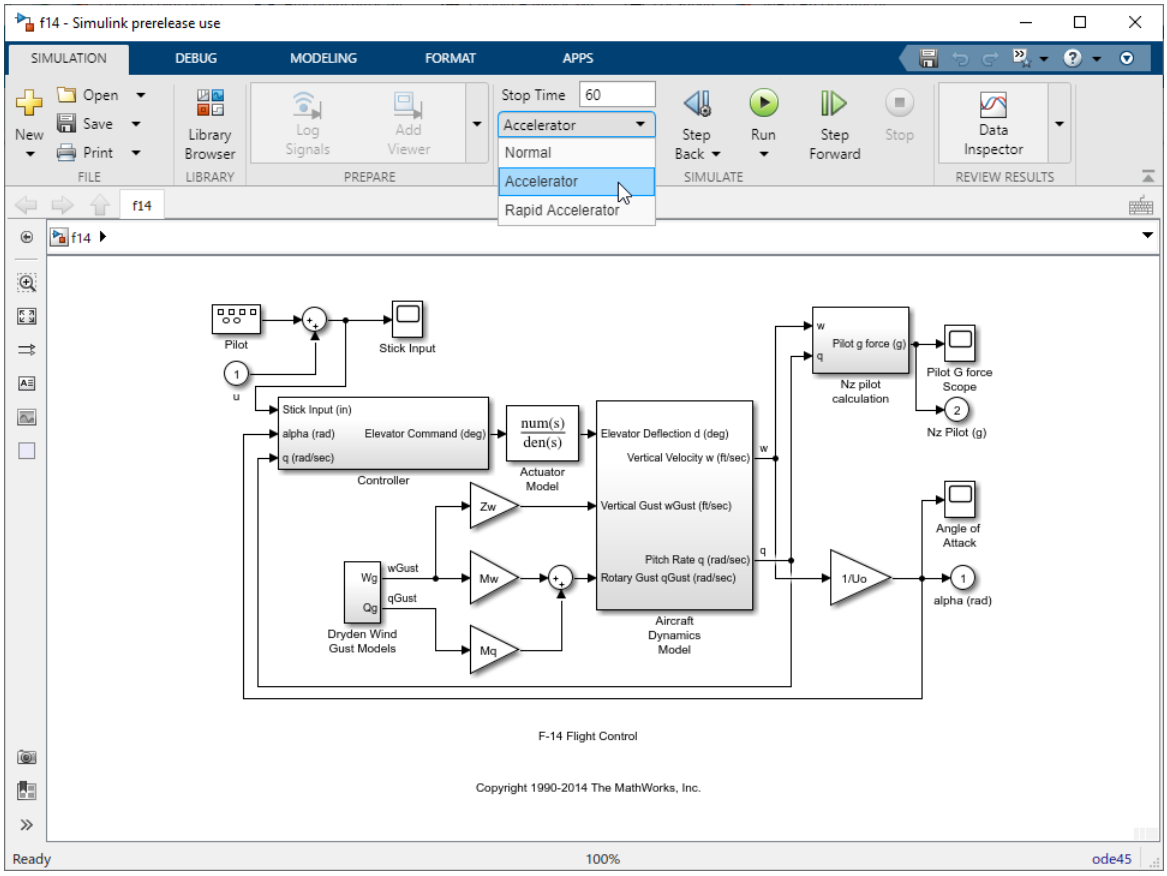
   The Simulink Preferences window appears.
2. In the Simulink Preferences window, navigate to the **Simulation cache folder** parameter.
3. Enter the absolute or relative path to your subfolder and click **Apply**.

Run Acceleration Mode from the User Interface

To accelerate a model, first open it, and then on the **Simulation** tab, in the **Simulate** section, select `Accelerator` or `Rapid Accelerator` from the drop-down menu. Then start the simulation.

The following example shows how to accelerate the already opened `f14` model using the Accelerator mode:

1. On the **Simulation** tab, in the **Simulate** section, select `Accelerator` from the drop-down menu.



2 On the **Simulation** tab, click **Run**.

The Accelerator and Rapid Accelerator modes first check to see if code was previously compiled for your model. If code was created previously, the Accelerator or Rapid Accelerator mode runs the model. If code was not previously built, they first generate and compile the C code, and then run the model.

For an explanation of why these modes rebuild your model, see “Code Regeneration in Accelerated Models” on page 35-7.

The Accelerator mode places the generated code in a subfolder of the working folder called `slprj/accel/`*modelname* (for example, `slprj/accel/f14`). If you want to change this path, see “Changing the Location of Generated Code” on page 35-19.

The Rapid Accelerator mode places the generated code in a subfolder of the working folder called `slprj/raccel/`*modelname* (for example, `slprj/raccel/f14`).

Note The warnings that blocks generate during simulation (such as divide-by-zero and integer overflow) are not displayed when your model runs in Accelerator or Rapid Accelerator mode.

Making Run-Time Changes

A feature of the Accelerator and Rapid Accelerator modes is that simple adjustments (such as changing the value of a Gain or Constant block) can be made to the model while the simulation is still

running. More complex changes (for example, changing from a `sin` to `tan` function) are not allowed during run time.

The Simulink software issues a warning if you attempt to make a change that is not permitted. The absence of a warning indicates that the change was accepted. The warning does not stop the current simulation, and the simulation continues with the previous values. If you wish to alter the model in ways that are not permitted during run time, you must first stop the simulation, make the change, and then restart the simulation.

In general, simple model changes are more likely to result in code regeneration when in Rapid Accelerator mode than when in Accelerator mode.

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Interact with the Acceleration Modes Programmatically” on page 35-22
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “How Acceleration Modes Work” on page 35-3
- “Code Regeneration in Accelerated Models” on page 35-7

Interact with the Acceleration Modes Programmatically

In this section...

Why Interact Programmatically?

You can build an accelerated model, select the simulation mode, and run the simulation from the command prompt or from MATLAB script. With this flexibility, you can create Accelerator mode MEX-files in batch mode, allowing you to build the C code and executable before running the simulations. When you use the Accelerator mode interactively at a later time, it will not be necessary to generate or compile MEX-files at the start of the accelerated simulations.

Build JIT Accelerated Execution Engine

With the `accelbuild` command, you can build a JIT accelerated execution engine without actually simulating the model. For example, to build an Accelerator mode simulation of `myModel`:

```
accelbuild myModel
```

Control Simulation

You can control the simulation mode from the command line prompt by using the `set_param` command:

```
set_param('modelname','SimulationMode','mode')
```

The simulation mode can be `normal`, `accelerator`, `rapid`, or `external`.

For example, to simulate your model with the Accelerator mode, you would use:

```
set_param('myModel','SimulationMode','accelerator')
```

However, a preferable method is to specify the simulation mode within the `sim` command:

```
simOut = sim('myModel', 'SimulationMode', 'accelerator');
```

You can use `bdroot` to set parameters for the currently active model (that is, the active model window) rather than *`modelname`* if you do not wish to explicitly specify the model name.

For example, to simulate the currently opened system in the Rapid Accelerator mode, you would use:

```
simOut = sim(bdroot,'SimulationMode','rapid');
```

Simulate Your Model

You can use `set_param` to configure the model parameters (such as the simulation mode and the stop time), and use the `sim` command to start the simulation:

```
sim('modelname', 'ReturnWorkspaceOutputs', 'on');
```

However, the preferred method is to configure model parameters directly using the `sim` command, as shown in the previous section.

You can substitute `gcs` for *`modelname`* if you do not want to explicitly specify the model name.

Unless target code has already been generated, the `sim` command first builds the executable and then runs the simulation. However, if the target code has already been generated and no significant changes have been made to the model (see “Code Regeneration in Accelerated Models” on page 35-7 for a description), the `sim` command executes the generated code without regenerating the code. This process lets you run your model after making simple changes without having to wait for the model to rebuild.

Simulation Example

The following sequence shows how to programmatically simulate `myModel` in Rapid Accelerator mode for 10,000 seconds.

First open `myModel`, and then type the following in the Command Window:

```
simOut = sim('myModel', 'SimulationMode', 'rapid'...
'StopTime', '10000');
```

Use the `sim` command again to simulate after making a change to your model. If the change is minor (adjusting the gain of a gain block, for instance), the simulation runs without regenerating code.

Customize the Acceleration Build Process

You can programmatically control the Accelerator mode and Rapid Accelerator mode build process and the amount of information displayed during the build process. See “Customize the Build Process” on page 35-19 for details on why doing so might be advantageous.

Controlling the Build Process

Use `SimCompilerOptimization` to set the degree of optimization used by the compiler when generating code for acceleration. The permitted values are `on` or `off`. The default is `off`.

Enter the following at the command prompt to turn on compiler optimization:

```
set_param('myModel', 'SimCompilerOptimization', 'on')
```

When `SimCompilerOptimization` is set to `on` in JIT accelerated mode, the simulation time for some models improves, while the build time can become slower.

Controlling Verbosity During Code Generation

Use the `AccelVerboseBuild` parameter to display progress information during code generation. The permitted values are `on` or `off`. The default is `off`.

Enter the following at the command prompt to turn on verbose build:

```
set_param('myModel', 'AccelVerboseBuild', 'on')
```

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10
- “Code Regeneration in Accelerated Models” on page 35-7

Run Accelerator Mode with the Simulink Debugger

In this section...

Advantages of Using Accelerator Mode with the Debugger

The Accelerator mode can shorten the length of your debugging sessions if you have large and complex models. For example, you can use the Accelerator mode to simulate a large model and quickly reach a distant break point.

For more information, see “Accelerator Mode” on page 35-4.

Note You cannot use the Rapid Accelerator mode with the debugger.

How to Run the Debugger

To run your model in the Accelerator mode with the debugger:

1. On the **Simulation** tab, in the **Simulate** section, select `Accelerator` from the drop-down menu.
2. At the command prompt, enter:

   ```
   sldebug modelname
   ```

3. At the debugger prompt, set a time break:

   ```
   tbreak 10000
   continue
   ```

4. Once you reach the breakpoint, use the debugger command `emode` (execution mode) to toggle between Accelerator and Normal mode.

When to Switch Back to Normal Mode

You must switch to Normal mode to step through the simulation by blocks, and when you want to use the following debug commands:

- `trace`
- `break`
- `zcbreak`
- `nanbreak`

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19
- “Interact with the Acceleration Modes Programmatically” on page 35-22

More About

- “What Is Acceleration?” on page 35-2
- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10

Comparing Performance

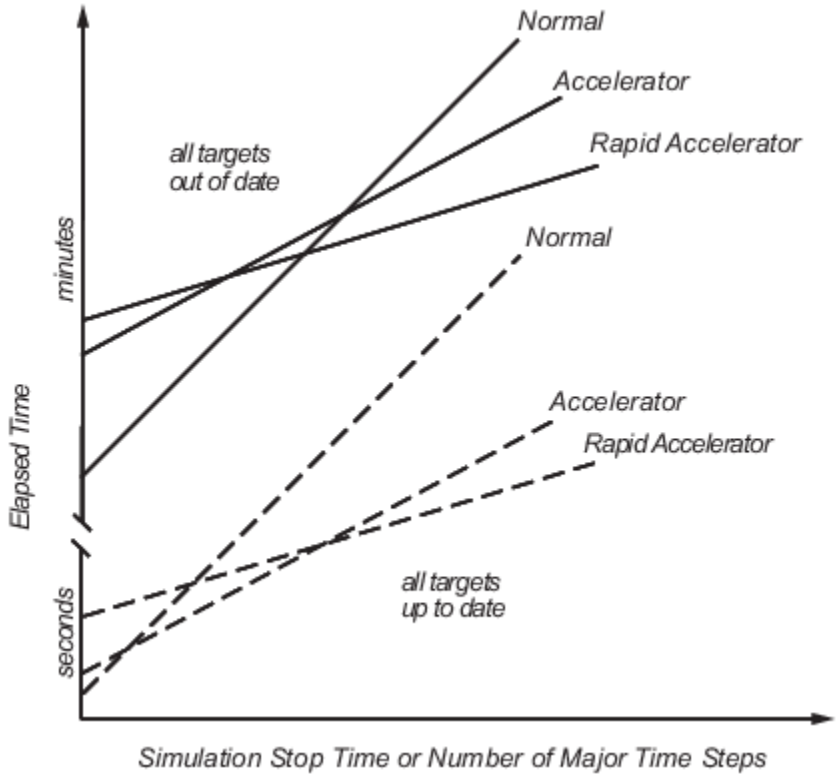
In this section...

Performance of the Simulation Modes

The Accelerator and Rapid Accelerator modes give the best speed improvement compared to Normal mode when simulation execution time exceeds the time required for code generation. For this reason, the Accelerator and Rapid Accelerator modes generally perform better than Normal mode when simulation execution times are several minutes or more. However, models with a significant number of Stateflow or MATLAB Function blocks might show only a small speed improvement over Normal mode because in Normal mode these blocks also simulate through code generation.

Including tunable parameters in your model can also increase the simulation time.

The figure shows in general terms the performance of a hypothetical model simulated in Normal, Accelerator, and Rapid Accelerator modes.



Performance When the Target Must Be Rebuilt

The solid lines in the figure show performance when the target code must be rebuilt (“all targets out of date”). For this hypothetical model, the time scale is on the order of minutes, but it could be longer for more complex models.

As generalized in the figure, the time required to compile the model in Normal mode is less than the time required to build either the Accelerator target or Rapid Accelerator executable. It is evident from the figure that for small simulation stop times Normal mode results in quicker overall simulation times than either Accelerator mode or Rapid Accelerator mode.

The crossover point where Accelerator mode or Rapid Accelerator mode result in faster execution times depends on the complexity and content of your model. For instance, those models running in Accelerator mode containing large numbers of blocks using interpreted code (see “Select Blocks for Accelerator Mode” on page 35-14) might not run much faster than they would in Normal mode unless the simulation stop time is very large. Similarly, models with a large number of Stateflow Chart blocks or MATLAB Function blocks might not show much speed improvement over Normal mode unless the simulation stop times are long. You can accelerate models with Stateflow Chart blocks or MATLAB Function blocks through code generation.

For illustration purposes, the graphic represents a model with a large number of Stateflow Chart blocks or MATLAB Function blocks. The curve labeled “Normal” would have much smaller initial elapsed time than shown if the model did not contain these blocks.

Performance When the Targets Are Up to Date

As shown by the broken lines in the figure (“all targets up to date”) the time for the Simulink software to determine if the Accelerator target or the Rapid Accelerator executable are up to date is significantly less than the time required to generate code (“all targets out of date”). You can take advantage of this characteristic when you wish to test various design tradeoffs.

For instance, you can generate the Accelerator mode target code once and use it to simulate your model with a series of gain settings. This is an especially efficient way to use the Accelerator or Rapid Accelerator modes because this type of change does not result in the target code being regenerated. This means the target code is generated the first time the model runs, but on subsequent runs the Simulink code spends only the time necessary to verify that the target is up to date. This process is much faster than generating code, so subsequent runs can be significantly faster than the initial run.

Because checking the targets is quicker than code generation, the crossover point is smaller when the target is up to date than when code must be generated. This means subsequent runs of your model might simulate faster in Accelerator or Rapid Accelerator mode when compared to Normal mode, even for short stop times.

Measure Performance

You can use the `tic`, `toc`, and `sim` commands to compare Accelerator mode or Rapid Accelerator mode execution times to Normal mode.

1. Open your model.
2. On the **Simulation** tab, in the **Simulate** section, select `Normal` from the drop-down menu.
3. Use the `tic`, `toc`, and `sim` commands at the command line prompt to measure how long the model takes to simulate in Normal mode:

   ```
   tic,[t,x,y]=sim('myModel',10000);toc
   ```

   `tic` and `toc` work together to record and return the elapsed time and display a message such as the following:

   ```
   Elapsed time is 17.789364 seconds.
   ```

4. On the **Simulation** tab, in the **Simulate** section, select `Accelerator` or `Rapid Accelerator` from the drop-down menu. Build an executable for the model by clicking **Run**. The acceleration modes use this executable in subsequent simulations as long as the model remains structurally unchanged. “Code Regeneration in Accelerated Models” on page 35-7 discusses the things that cause your model to rebuild.
5. Rerun the compiled model at the command prompt:

   ```
   tic,[t,x,y]=sim('myModel',10000);toc
   ```

6. The elapsed time displayed shows the run time for the accelerated model. For example:

   ```
   Elapsed time is 12.419914 seconds.
   ```

The difference in elapsed times (5.369450 seconds in this example) shows the improvement obtained by accelerating your model.

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Perform Acceleration” on page 35-19
- “Interact with the Acceleration Modes Programmatically” on page 35-22
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25
- “How to Improve Performance in Acceleration Modes” on page 35-30

More About

- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10

How to Improve Performance in Acceleration Modes

In this section...

Techniques

To get the best performance when accelerating your models:

- Verify that the Configuration Parameters dialog box settings are as follows:

Set...	To...
Solver data inconsistency	none
Array bounds exceeded	none
Signal storage reuse	selected

- Disable Stateflow debugging and animation.
- When logging large amounts of data (for instance, when using the Workspace I/O, To Workspace, To File, or Scope blocks), use decimation or limit the output to display only the last part of the simulation.
- Customize the code generation process to improve simulation speed. For details, see “Customize the Build Process” on page 35-19.

C Compilers

On computers running the Microsoft Windows operating system, the Accelerator and Rapid Accelerator modes use the default 64-bit C compiler supplied by MathWorks to compile your model. If you have a C compiler installed on your PC, you can configure the `mex` command to use it instead. You might choose to do this if your C compiler produces highly optimized code since this would further improve acceleration.

Note For an up-to-date list of 32- and 64-bit C compilers that are compatible with MATLAB software for all supported computing platforms, see:

`https://www.mathworks.com/support/compilers/current_release/`

See Also

Related Examples

- “Design Your Model for Effective Acceleration” on page 35-14
- “Interact with the Acceleration Modes Programmatically” on page 35-22
- “Run Accelerator Mode with the Simulink Debugger” on page 35-25

More About

- “How Acceleration Modes Work” on page 35-3
- “Choosing a Simulation Mode” on page 35-10
- “Comparing Performance” on page 35-27

Managing Blocks

Working with Blocks

Nonvirtual and Virtual Blocks

When creating models, you need to be aware that Simulink blocks fall into two basic categories: nonvirtual blocks and virtual blocks. Nonvirtual blocks play an active role in the simulation of a system. If you add or remove a nonvirtual block, you change the model's behavior. Virtual blocks, by contrast, play no active role in the simulation; they help organize a model graphically. Some Simulink blocks are virtual in some circumstances and nonvirtual in others. Such blocks are called conditionally virtual blocks. The table lists Simulink virtual and conditionally virtual blocks.

Block Name	Condition Under Which Block Is Virtual
Bus Assignment	Virtual if input bus is virtual.
Bus Creator	Virtual if output bus is virtual.
Bus Selector	Virtual if input bus is virtual.
Demux	Always virtual.
Enable	Virtual unless connected directly to an Outport block.
From	Always virtual.
Goto	Always virtual.
Goto Tag Visibility	Always virtual.
Ground	Always virtual.
Inport	Virtual *unless* the block resides in a conditionally executed or atomic subsystem *and* has a direct connection to an Outport block.
Mux	Always virtual.
Outport	Virtual when the block resides within any subsystem block (conditional or not), and does *not* reside in the root (top-level) Simulink window.
Selector	Virtual only when **Number of input dimensions** specifies `1` and **Index Option** specifies `Select all`, `Index vector (dialog)`, or `Starting index (dialog)`.
Signal Specification	Always virtual.
Subsystem	Virtual unless the block is conditionally executed or the **Treat as atomic unit** check box is selected. You can check if a block is virtual with the `IsSubsystemVirtual` block property. See “Block-Specific Parameters”.
Terminator	Always virtual.
Trigger	Virtual when the output port is *not* present.

See Also

More About

- “Specify Block Properties” on page 36-4
- “Display Port Values for Debugging” on page 36-16
- “Control and Display Execution Order” on page 36-25

- “Access Block Data During Simulation” on page 36-37
- “Block Libraries”

Specify Block Properties

For each block in a model, you can set general block properties, such as:

- A description of the block
- The block execution order
- A block annotation
- Block callback functions

To set block properties, use the Property Inspector. You can set properties in the **Properties** and **Info** tabs of the Property Inspector when the block is selected. Alternatively, you can use the Block Properties dialog box. For more information on setting properties, see “Add Blocks and Set Parameters” on page 1-13.

Set Block Annotation Properties

In the Property Inspector, use the **Block Annotation** section to display the values of selected block parameters in an annotation. The annotation appears below the block icon.

Enter the text of the annotation in the text box. You can use a block property token in the annotation. The value for the property replaces the token in the annotation in the model. To display a list of tokens that you can use in an annotation, type % in the text box. The parameters that are valid for the selected block appear. See “Common Block Properties” and “Block-Specific Parameters”.

Suppose that you specify the following annotation text and tokens for a Product block:

```
Multiplication = %<Multiplication>
Sample time = %<SampleTime>
```

In the Simulink Editor, the block displays this annotation:

You can also create block annotations programmatically. See “Create Block Annotations Programmatically” on page 36-5.

Specify Block Callbacks

Use the **Callbacks** section to specify block callbacks. Callbacks are MATLAB commands that execute when a specific model action occurs, such as when you select or delete a block. For more information on callbacks, see “Callbacks for Customized Model Behavior” on page 4-44.

1. Select the block whose callback you want to set.
2. In **Properties** tab of the Property Inspector, in the **Callbacks** section, select the function that you want to assign the callback to. For example, select `OpenFcn` to specify a behavior for double-clicking a block.

3 In the text box, enter the command that you want to execute when that block function occurs.

After you assign a callback to a function, the function displays an asterisk next to it in the list. The asterisks helps you to see the functions that have callbacks assigned to them.

Note After you add an `OpenFcn` callback to a block, double-clicking the block does not open the block dialog box. Also, the block parameters do not appear in the Property Inspector when the block is selected. To set the block parameters, select **Block Parameters** from the block context menu.

Set a Block Callback Programmatically

This example shows how to use the `OpenFcn` callback to execute MATLAB scripts when you double-click a block. For example, in a MATLAB script you can define variables or open a plot of simulated data.

To create a callback programmatically, select the block to which you want to add this property. Then, at the MATLAB command prompt, enter a command in this form:

```
set_param(gcb,'OpenFcn','myfunction')
```

In this example, `myfunction` represents a valid MATLAB command or a MATLAB script on your MATLAB search path.

Specify Block Execution Priority and Tag

In the **Advanced Properties** section of the block properties, you can specify the block priority and identify the block by assigning a value to the **Tag** property.

- **Priority** — Specify the execution priority of the block relative to other blocks in the model.
- **Tag** — Specify an identifier for the block. Specify text to assign to the block Tag parameter. Setting this property is useful for finding the block in the model by searching or programmatically using `find_system`. See “Exploring the Model Hierarchy”.

Use Block Description to Identify a Block

The **Info** tab displays information about the block type. The block author provides this description.

You can also enter a description in the **Description** box to provide information about the block instance.

- If you add a description, you can set up your model display so that the description appears in a tooltip when you hover over the block. To enable this tooltip, on the **Debug** tab, select **Information Overlays > Description in Tooltip** .
- The **Description** property can help you to find a block by searching. See **Simulink Editor**.

Create Block Annotations Programmatically

You can use a block `AttributesFormatString` parameter to display specified block parameter values below the block. “Common Block Properties” and “Block-Specific Parameters” describe the parameters that a block can have. Use the Simulink `set_param` function to set this parameter to the attributes format that you want.

The attributes format can be any text that has embedded parameter names. An embedded parameter name is a parameter name preceded by `%<` and followed by `>`, for example, `%<priority>`. Simulink displays the attributes format text below the block icon, replacing each parameter name with the corresponding value. You can use line-feed characters (`\n`) to display each parameter on a separate line. For example, select a Gain block and enter this command at the MATLAB command prompt:

```
set_param(gcb,'AttributesFormatString','pri=%<priority>\ngain=%<Gain>')
```

The Gain block displays this block annotation:

If a parameter value is not text or an integer, `N/S` (for not supported) appears in place of the value. If the parameter name is not valid, Simulink displays `???` in place of the value.

See Also

More About

- “Add Blocks and Set Parameters” on page 1-13
- “Callbacks for Customized Model Behavior” on page 4-44

Format a Model

As you build a model, you can adjust block positions, change block and background color, place block names and ports on any side of the block, adjust fonts, and add elements that help to improve model readability. These changes can help to organize the model visually and help others understand the model when you share it.

You can make these types of changes to your model format:

- Improve the model layout. See “Improve Model Layout” on page 36-7.
- Flip or rotate blocks or groups of blocks. These adjustments help blocks to fit in the model and connect with other blocks. See “Flip or Rotate Blocks” on page 36-7.
- Reposition or hide block names and move ports to any side of the block. See “Manage Block Names and Ports” on page 36-10.
- Add colors to blocks and to the background. See “Specify Model Colors” on page 36-12.
- Adjust aesthetics by changing fonts and deepening the intensity of drop shadows. See “Specify Fonts in Models” on page 36-12 and “Increase Drop Shadow Depth” on page 36-13.
- Surround groups of blocks with a box to show that the blocks are related. See “Box and Label Areas of a Model” on page 36-13.
- Copy formatting from a block, line, or area to another model element. See “Copy Formatting Between Model Elements” on page 36-15.
- Document a model using text, image, and math annotations. See “Describe Models Using Notes and Annotations” on page 4-3.
- Annotate a block. See “Set Block Annotation Properties” on page 36-4.
- Change the block icon, for example, display a graphic on the block. Use a mask to achieve this effect. A mask also enables you to design a custom interface for a block. To learn about masks, see “Masking Fundamentals” on page 39-2.

Improve Model Layout

To improve your diagram layout and appearance, in the **Format** tab, click **Auto Arrange**. This option:

- Aligns blocks in your model from left to right, starting with inputs and ending with outputs.
- Resizes blocks, such as the Constant block, to display long parameter values.
- Standardizes block size among similar blocks.
- Straightens signal lines by moving blocks.

Alternatively, you can try improving the shapes of signal lines. To try to improve the shape of a single line, select it and, from the action bar, select **Auto-route Line**. The line redraws if a better route between model elements is possible.

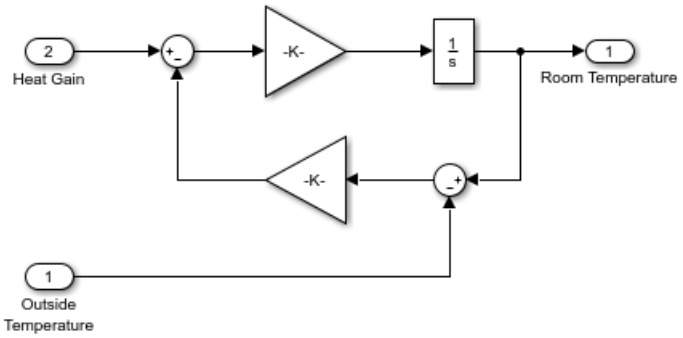
You can try to improve multiple lines using **Auto-route Lines**. To access **Auto-route Lines** from the action bar, select either a block or multiple model elements using a drag box.

Flip or Rotate Blocks

You can change the orientation of a block or a group of blocks by rotating in 90-degree increments or by flipping. Rotate or flip blocks to fit better in the model, for example, in feedback loops. You might

also need to rotate so that input ports align with output ports or to make better use of the model canvas.

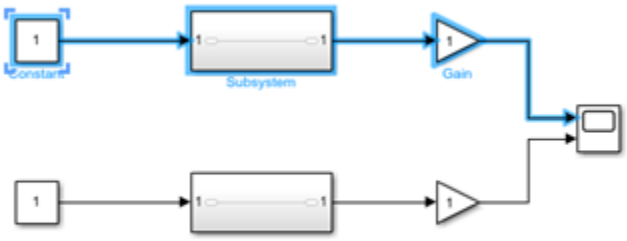
The figure shows a Gain block flipped to simplify a feedback loop diagram.



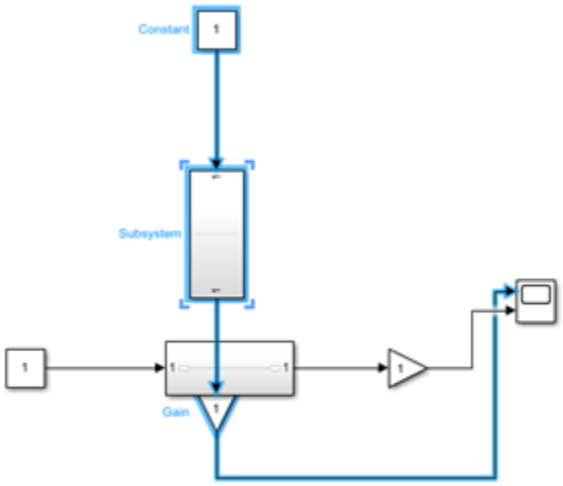
- To rotate or flip a block, select the block, and then, on the **Format** tab, click a **Rotate** icon in the **Arrange** section. You can rotate clockwise (**Ctrl+R**) or counterclockwise (**Ctrl+Shift+R**). Or click a **Flip** icon in the **Arrange** section. You can flip left-right or up-down, based on the orientation of a block's ports. For example, if the ports are on the sides, the block flips left to right.
- Blocks rotate when you place them on a signal line that has an orientation other than left to right. For example, if the signal goes from bottom to top and you place a block on it, the block rotates with its ports up.
- To rotate or flip a group of blocks, select multiple blocks, and then click the rotate or flip icon. The rotation or flip occurs as a group. The group of blocks flip only left to right when you flip blocks as a group.

  After you rotate or flip a group of blocks, you can improve the readability of the model by editing the signal lines. The **Auto Arrange** option might improve the appearance of signals. (For information on rotation with multiple ports, see “Port Location After Rotating or Flipping” on page 36-9.)

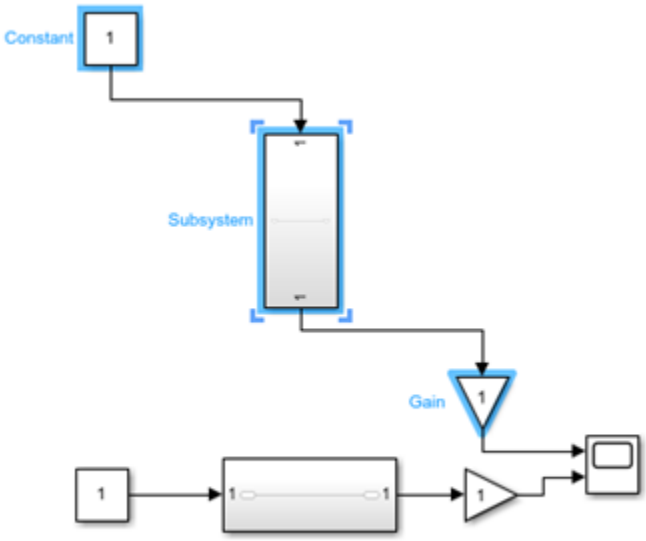
  For example, suppose you rotate these selected blocks clockwise.



  After you rotate the blocks and move them to fix overlapping, the model looks like this.



With the selection in place, in the toolstrip, click the **Format** tab and select **Auto Arrange** to improve the appearance of signal lines.

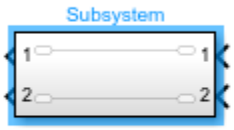


Port Location After Rotating or Flipping

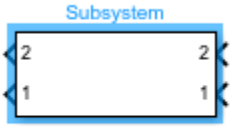
Rotating moves block ports from the sides to top and bottom or the reverse, depending on the placement of the ports. The resulting positions of the block ports depend on the block port rotation type.

Rotating can reposition the ports on some blocks to maintain left-to-right or top-to-bottom port numbering order. A block whose ports are reordered after a rotation have the default port rotation type. This policy helps to maintain the left-right and top-down block diagram orientation convention

used in control system modeling applications. Blocks by default use this rotation policy. The figure shows the effect of clockwise rotation on a block with the default port rotation policy.



A masked block can specify for ports to keep their order after rotation (see “Port rotation”). These blocks have a physical port rotation type. This policy helps when designing blocks to use in mechanical and hydraulic systems modeling and other applications where diagrams do not have a preferred orientation. The figure shows the effect of clockwise rotation on a block with a physical port rotation type.



Flipping a block moves the ports to the opposite side of the block, creating a mirror image, regardless of port rotation type.

Manage Block Names and Ports

You can manage block names by displaying or hiding them and by changing their location on the block.

Note Copying and pasting blocks whose names follow numeric order (Gain1, Gain2, and so on) creates names that follow standard sorting order conventions for ASCII characters. This sorting order can result in a sequence of numbers on the block names that is hard to understand. If the numbering scheme is important to you, name your blocks explicitly such that copying and pasting them creates names that follow a typical reading order. To do so, use a leading zero in the block names, for example Gain001, Gain002, and so on.

Hide or Display Block Names

The Simulink Editor names blocks when you create them. The first occurrence of the block is the library block name, for example, Gain. The next occurrence is the block name with a number appended. Each new block increments the number, for example, Gain1, Gain2, and so on. These names are called automatic names. By default, the Editor hides these names.

You can choose whether to hide or display block names. You can:

- Display all the hidden automatic names. On the **Format** tab, select **Auto > Hide Automatic Block Names** to clear the option.
- Temporarily display a hidden automatic block name by selecting the block.
- Name the block explicitly, for example, by its purpose in the model. The **Hide Automatic Names** setting does not affect blocks that you name explicitly. To name a block, select it, double-click the name, and type the new name.

In addition, you can explicitly hide or display any block name. Explicitly hidden or displayed blocks are not affected by the **Hide Automatic Block Names** setting. To explicitly hide or display a block name, select the block, then on the **Format** tab, select **Auto** and then select:

- **On** to always display the block name.
- **Off** to always hide the block name.
- **Auto** to return to the default state. If the block has an automatic name, **Hide Automatic Block Names** affects the block.

To display and hide block names programmatically, use `set_param` with the `'HideAutomaticNames'` option for models and the `'HideAutomaticName'` and `'ShowName'` options for blocks. For more information on these parameters, see “Common Block Properties”. The table shows how these parameters interact.

`'ShowName'` (block setting)	`'HideAutomaticName'` (block setting)	`'HideAutomaticNames'` (model setting)	Result
`'off'`	Any	Any	Name is hidden
`'on'`	`'on'`	`'on'`	Name is hidden
`'on'`	`'off'`	Any	Name is shown
`'on'`	`'on'`	`'off'`	Name is shown

Move Block Names

By default, block names appear below blocks whose ports are on the sides and to the left on blocks whose ports are on the top and bottom. To change the location of a block name, you can:

- Drag the block name to any side of the block.
- Select the block and then on the **Format** tab click **Flip Name**.

Move Ports

You can put ports on any side in any order on these blocks:

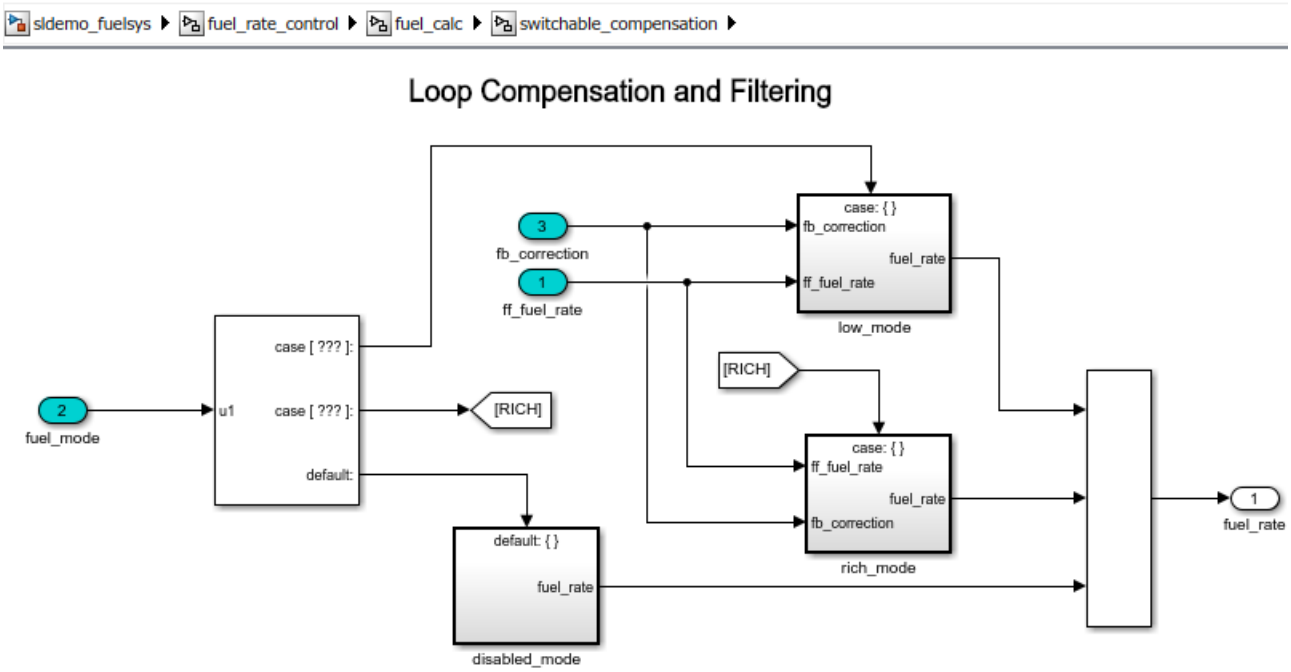
- Model block
- Subsystem block
- Subsystem Reference block
- Stateflow Chart
- Stateflow Truth Table
- Stateflow State Transition Table

You can move ports by clicking and dragging the port. For Subsystem blocks, the port index will automatically be renumbered after each move.

Specify Model Colors

You can specify the outline and interior colors of any block, and you can change the background color for any system in a model. You can also change text color and background color for annotations and fill color for areas.

This subsystem uses color to identify the input ports.



- To change outline color on a block, text color in an annotation, or interior color for an area, select the element, and then, on the **Format** tab, select a color from the **Foreground** menu. Changing the foreground color of a block also changes the color of its output signals.
- To change interior color on a block or background color in an annotation, select the element, and then, on the **Format** tab, select a color from the **Background** menu.
- To change a background color in a system, open the system, and then on the **Format** tab, select a color from the **Background** menu.

You can select a color from the menu or select **Custom** to open the color picker and define your own color.

You can also use the Property Inspector to change color for an area or an annotation. To specify colors programmatically, see “Specify Colors Programmatically” on page 1-6

Specify Fonts in Models

Change font family, style, and size for any model element to make your model easier to read or to comply with company standards. You can modify the font for selected blocks, signal labels, areas, and annotations. Some blocks display text according to the font style settings, and some blocks have fixed fonts and styles. To increase the font size of those blocks, zoom in.

You can also change the default font for the model. The default font affects any elements whose font you have not changed and any new elements you create. If you want to use the same default font in

all new models, change the default model font in your default template. See “Use Customized Settings When Creating New Models” on page 1-9.

- To change the font of a block, signal label, area, or annotation, select the element, and then specify font information on the **Format** tab.
- To change the default fonts for the model, on the **Format** tab, click the **Font Properties** button arrow, then click **Fonts for Model**. Use the Font Styles dialog box to specify the font information.

You can also use the Property Inspector to change font for an area or an annotation.

Select Font Dialog Box on Linux Machines

On Linux machines configured for English, the **Font style** list in the Select Font dialog box can appear out of order and in another language in some fonts. If the characters in your **Font style** list appear in another language, set the `LANG` environment variable to `en_US.utf-8` before you start MATLAB. For example, at a Linux terminal, enter:

```
setenv LANG en_US.utf-8
matlab
```

Increase Drop Shadow Depth

By default, blocks have a drop shadow. To make the block stand out more against the background, you can increase the depth of the drop shadow.

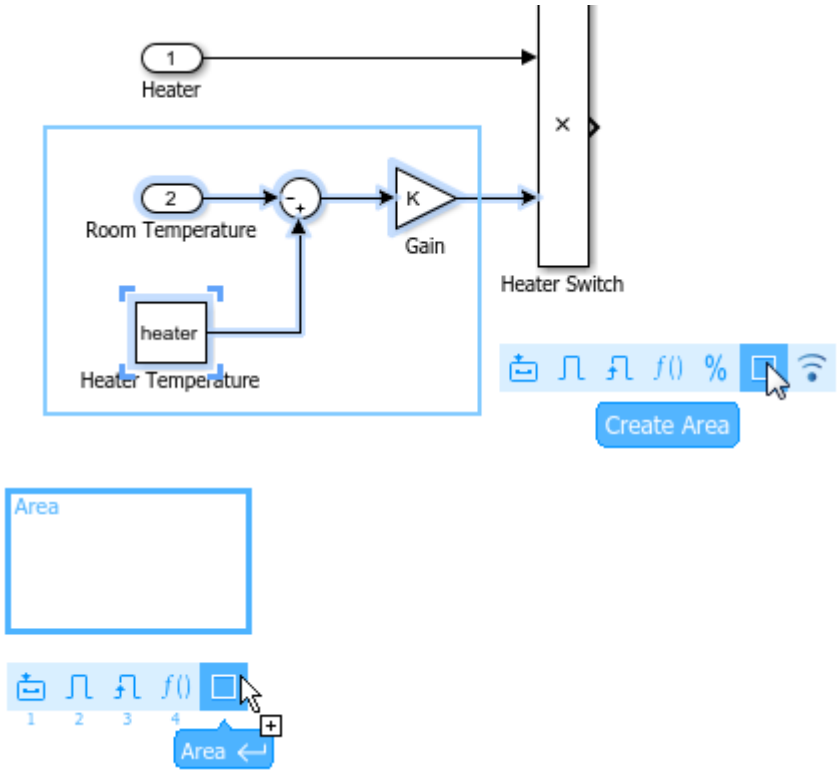
Select the blocks whose drop shadow depth you want to increase, and then on the **Format** tab click **Shadow**.

Tip To remove the default drop shadow for all blocks, select the Simulink Editor preference **Use classic diagram theme**.

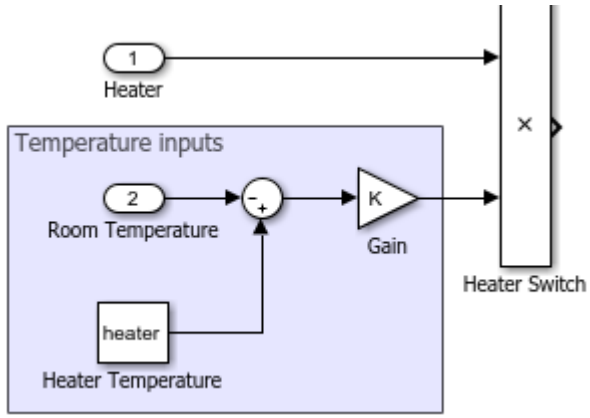
Box and Label Areas of a Model

Add an area to your model to visually group related model elements in a box. An area can move with the blocks it surrounds. You can add text to an area to briefly describe or label the area.

1. Drag a box around the area of interest in the model. Or, drag on a blank area of the canvas to draw the area shape.
2. From the action bar, select the option to create an area.



3 Type the name of the area. The name appears in the upper-left corner of the area.



To enter the name later, select the area, click the **?**, and start typing, or use the **Name** property in the Property Inspector.

4 Optionally, add a description of the area contents using the Property Inspector.

5 To move the area and its contents, drag the area near the border.

Tip To move an area without moving its contents, hold **Alt** (**option** on a Mac) and drag.

Convert Area to a Subsystem

An area is similar to a subsystem in that it is a way to group related blocks. However, a subsystem creates a hierarchy, replacing multiple blocks in a model with a single block. You can initially group

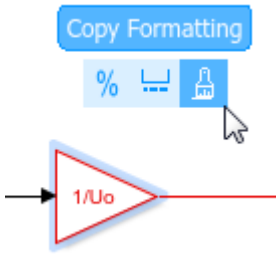
related blocks in an area and later decide to put those blocks in a subsystem by converting the area. The resulting subsystem has the same name, blocks, description, and requirements traceability information as the area.

To convert an area to a subsystem, right-click the area and select **Create Subsystem from Area**.

Copy Formatting Between Model Elements

If you have applied formatting to a block, signal line, or area in a model, you can copy the formatting and apply it to another model element. Examples of formatting include font changes, foreground and background color, and drop shadow effects.

1. Select the block, line, or area whose formatting you want to copy.
2. From the action bar, select **Copy Formatting**. The cursor becomes a paintbrush.



3. Using the paintbrush, click each element that you want to copy the formatting to.
4. To cancel the paintbrush cursor, click a blank spot on the canvas or press **Esc**.

See Also

More About

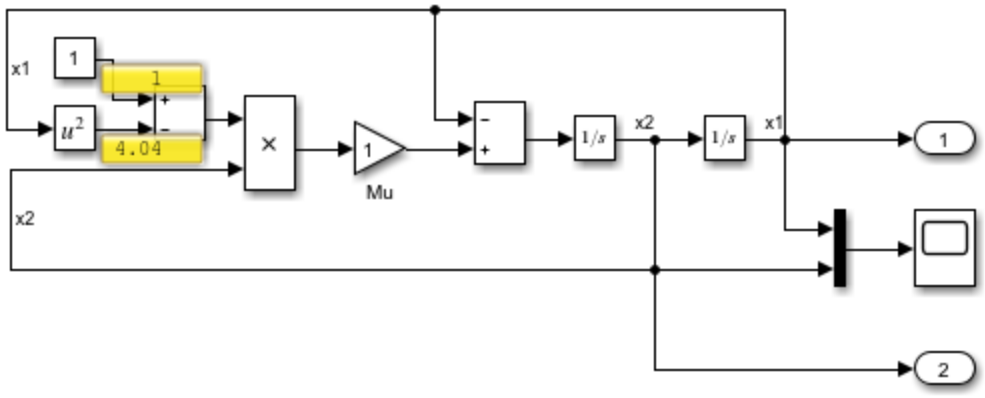
- “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61
- “Arrange Model Layouts Automatically” on page 1-5
- “Use Customized Settings When Creating New Models” on page 1-9
- “Set Block Annotation Properties” on page 36-4
- “Build and Edit a Model Interactively” on page 1-8
- “Programmatic Modeling Basics” on page 1-2

Display Port Values for Debugging

In this section...

Display Port Values for Easy Debugging

For many blocks whose signals carry data, Simulink can display signal values (block output) as port value labels (similar to tool tips) on the block diagram during and after a simulation. Port value labels display block output values when Simulink runs block output methods. This model shows a port value label for the ports on the Constant and Math blocks, output values of `1` and `4.04`.

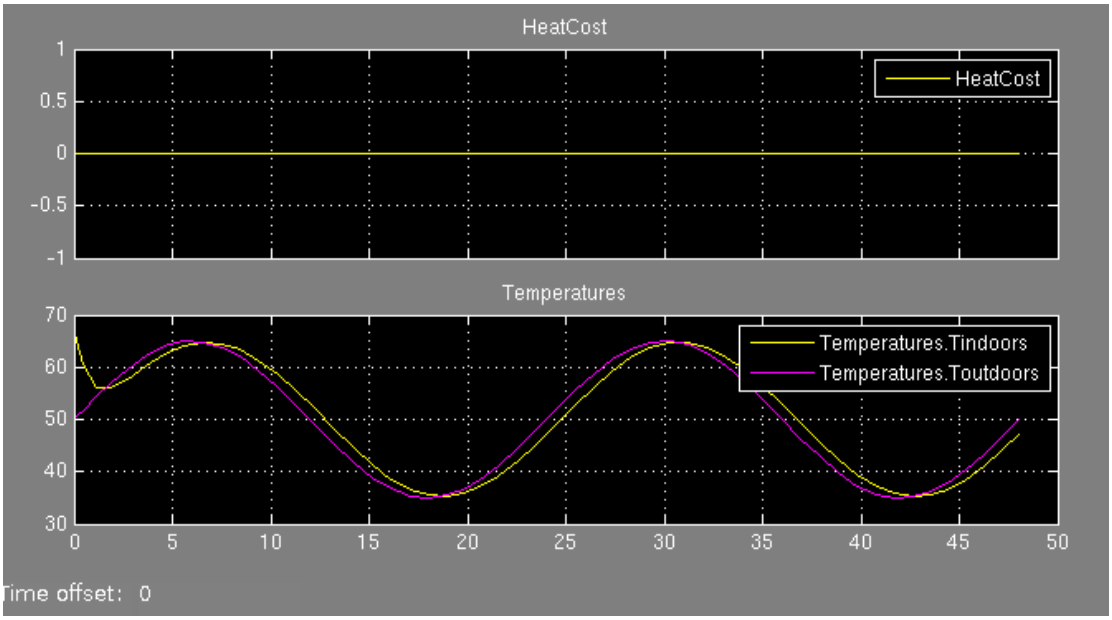


If the port value label appears empty, this means that no port value is currently available. For example, toggling a port value label on a continuous block when paused during simulation does not display any values in the label.

Port value labels are also empty when you have not yet simulated the model. This is because the block output methods do not run when the model does not simulate.

If you toggle or hover on a block that Simulink optimizes out of a simulation (such as a virtual subsystem block), while you simulate, the model displays the text `optimized`.

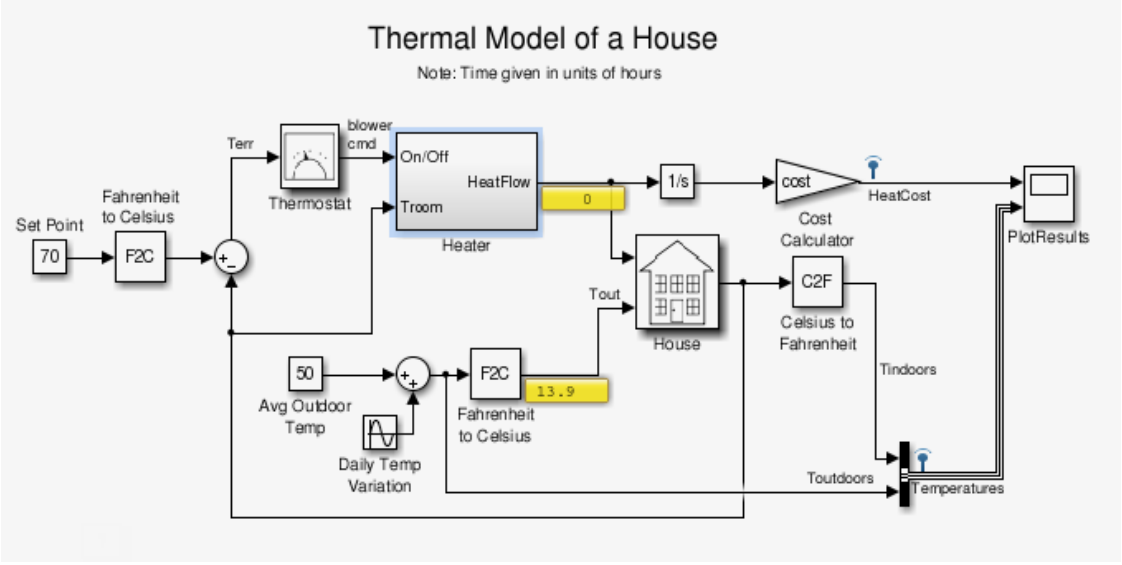
Displaying port value data tips can help during interactive debugging of a model. For example, the figure shows the output of a thermal model for a house.



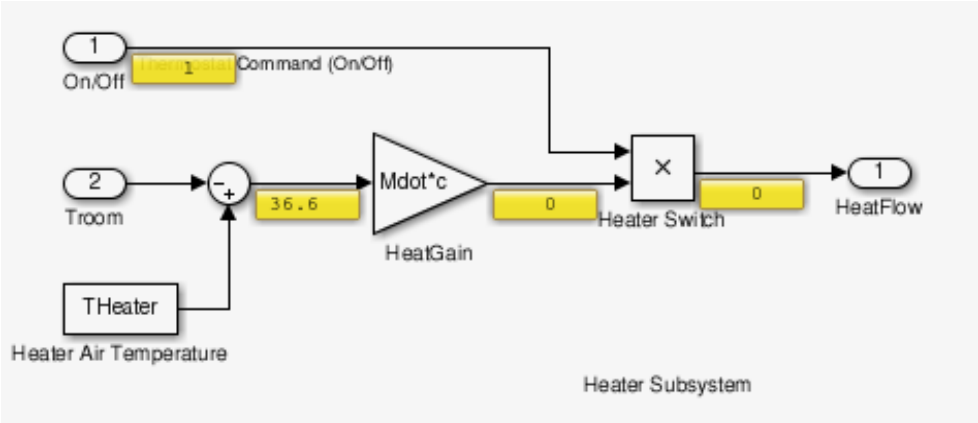
These results suggest a problem with the model because:

- The heating cost is `0` at all temperatures.
- The temperature inside the house matches ambient temperature almost exactly.

In such cases, debugging the blocks in the model interactively can help isolate the error. Port value labels provide information at the output of every block in the model. So in this example, if you step forward using Simulation Stepper, you can see that the output of the Heater subsystem is `0` at every time step.



To learn more, you can enable port value labels for blocks inside the Heater subsystem. Using Simulation Stepper, if you step forward again to display the values, you can see that there is an issue with the HeatGain block. The output is constant at `0`.



This technique helps you isolate the issue.

To simplify debugging, you can turn on and off port value labels during simulation. Besides providing useful information for debugging, port value displays can help you monitor a signal value during simulation. However, these labels are not saved with a model.

For nonnumeric data display, Simulink uses these values:

Message	Explanation
`action`	The signal executes action subsystems.
`fcn-call`	The signal is a function-call signal, e.g., Function Call Generator output.
`ground`	The signal is coming from a Ground block.
`not a data signal`	The signal does not contain valid data, e.g., the signal is from a block that is commented out.

In some cases:

- The port value display may not be able to acquire the value signal or
- The signal's value cannot be easily displayed

In such cases, Simulink uses these values.

Message	Explanation
...	The signal dimension exceeds the maximum number of elements Simulink can display. For more information, see "Display Port Values for a Model" on page 36-22.
(no message)	The simulation data available is insufficient. Step forward or press play to obtain more data.
`click to add signals`	You have enabled a port value label on a bus. However, you have not selected a signal to display. Click the label to select bus signals.
`inaccessible`	Simulink cannot obtain the port value. For an example, see "Signal Storage Reuse" on page 36-23.
`[m*n]`	This is a nonvector signal. Simulink cannot display the actual values of the matrix. It displays the matrix dimension instead.

Message	Explanation
`no data yet`	This message appears when: • The simulation data is not available. Start the simulation to see values. • If the model contains subsystems (for example, an enabled subsystem) and model references and they are not executed during a simulation.
`not used`	Simulink cannot obtain the signal value due to optimization.
`removed`	Simulink cannot obtain the signal value due to block reduction.
`optimized`	Simulink cannot obtain the signal value due to optimization. In Normal mode, this message appears for blocks with **Conditional input branch execution** enabled. For more information, see “Conditionally Executed Subsystems Overview” on page 10-3.
`unavailable`	The simulation data available is insufficient. For example, see “Simulation Stepper” on page 36-24.

Note You can force a value label to display the signal value by designating the signal as a test point. Use the **Properties** dialog box to do this.

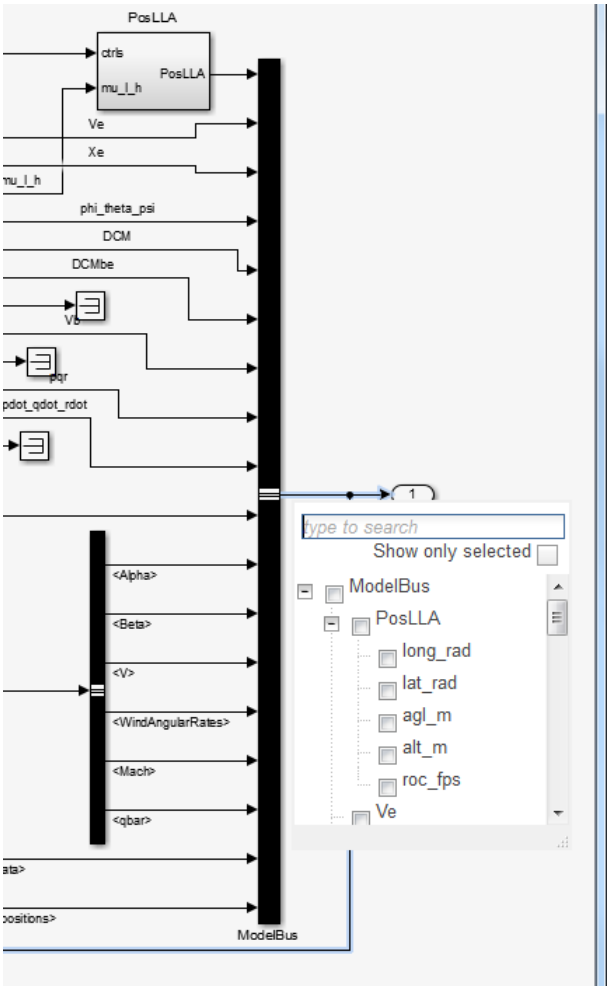
Display Value for a Specific Port

To display the value of a specific port or port values for a block before simulation, select one or more signals, right-click the selection, and select **Show Value Label of Selected Port**.

By default, Simulink displays the value of a signal when you click it during simulation. You can control this behavior. On the **Debug** tab, select  **> Options**, then in the Value Label Display Options dialog box, select **Enable by default during simulation**.

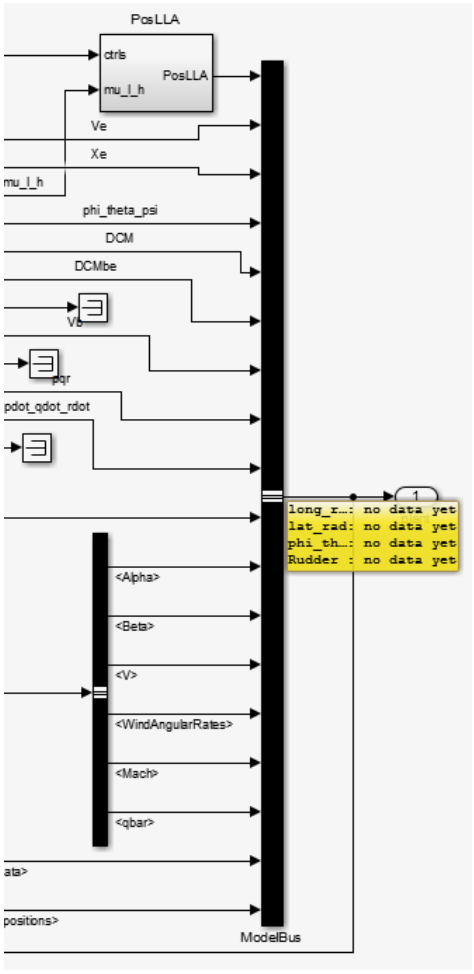
Note To remove all data tips, on the **Debug** tab, select  **> Remove Value Displays**.

For bus signals, the **Show Value Label of Selected Port** option opens a dialog box where you can select from all signals in the bus. For example, in this model, you can see the dialog box for all signals that are contained in ModelBus.

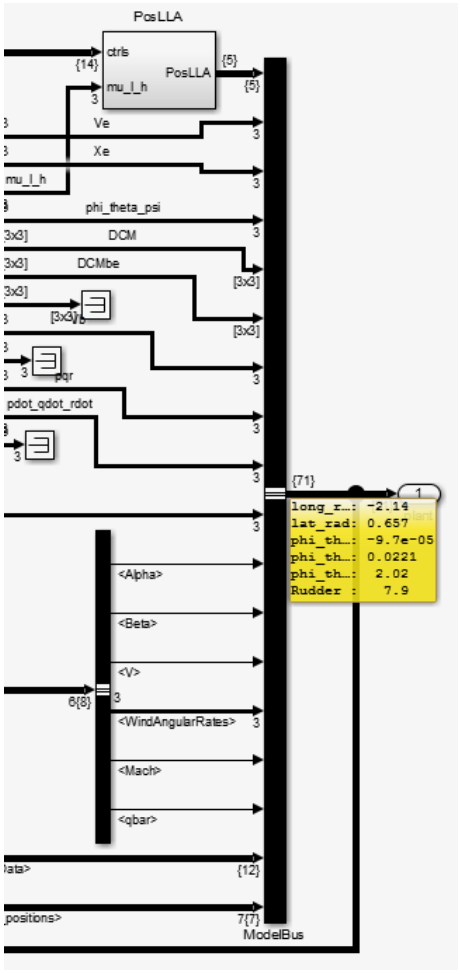


You can search for a signal by name or filter through the hierarchy. Select a parent signal to include all of the signals it contains. You can also filter the display to view only those signals you have selected.

Click anywhere outside the dialog box to close it. The port value label appears. The label has no data; it displays values when you simulate the model.



When you simulate the model, the port value label displays the names and values of the signals you chose. To change the signals to display, click on the port value label to reopen the dialog. You can also click on another signal to display its value.



Note Simulink does not save the values of a signal when you remove the port value label.

Display Port Values for a Model

Specify port value display formatting and the frequency of updates. The Value Label Display Options dialog box controls these settings on the entire model.

1. In the model whose port values you want to display, on the **Debug** tab, select > **Options**.
2. In the Value Label Display Options dialog box, specify your preferences for:
   - The display options, including font size, the refresh frequency, and the number of elements displayed for vector signals with signal widths greater than 1
   - The display mode
   - Floating-point or fixed-point format

Port Value Display Limitations

Performance

Enabling the hovering option for a model or setting at least one block to **Toggle Value Labels When Clicked** slows down simulation.

Accelerated Modes

Port values work in Normal and Accelerator modes only. They do not work in Rapid Accelerator and External modes. The table shows how accelerator modes affect the display of port values.

Accelerated Mode	Port Values
Accelerator	• Signals not optimized in Accelerator mode display port values as in Normal mode. Signals optimized in Accelerator mode display port values as `optimized`. For more information, see “Display Port Values for Easy Debugging” on page 36-16. • Model reference blocks simulated in Accelerator mode do not get their port value displays updated.
Rapid Accelerator	Incompatible. The limitation exists whether the model or its parent specifies accelerated simulation. For more information, see “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” (Simulink Coder).

Signal Storage Reuse

If the output port buffer of a block is shared with another block through the optimization of signal storage reuse, the port value displays as `inaccessible`. You can disable signal storage reuse using the **Signal storage reuse** check box. However, disabling signal storage reuse increases the memory used during simulation.

Signal Data Types

- Simulink displays the port value for ports connected to most kinds of signals, including signals with built-in data types (such as `double`, `int32`, or `Boolean`), `DYNAMICALLY_TYPED`, and several other data types.
- Simulink shows the floating format for only noncomplex signal value displays.
- Simulink displays the port value of fixed point data types based on the converted double value.
- Simulink does not display data for signals with some composite data types, such as bus signals.

Subsystems

- You cannot display port values for subsystems contained in a variant subsystem when there are no signal lines connecting to them. In such cases, during simulation, Simulink automatically determines block connectivity based on the active variant. However, you can display port values within the subsystems contained in the variant subsystem. You can also display values on signal lines outside of the variant subsystem.
- When you disable a conditionally executed subsystem, the port value display for a signal that goes into an Outport block displays the value of the Outport block, depending on the **Output when disabled** setting.

- Simulink does not display data for the ports of an enabled subsystem that is not enabled.

Simulation Stepper

If you do not enable port value display when stepping forward, the display will not be available when stepping back. When stepping back, if the port value is unavailable, the `unavailable` label is displayed.

Refine Factor

Port value displays do not honor refine factor values (**Configuration Parameters > Data Import/Export > Additional parameters > Refine factor**) because Simulink updates port value displays only during major time steps.

Signal Specification Block and Inport Block

When you display port values on Signal Specification and Inport blocks in a subsystem, the value that is driving the blocks displays instead of the block values.

Command-Line Simulations

For efficiency, Simulink does not support port value displays during a command-line simulation using the `sim` command.

Merge Block

Simulink does not display the input values to the merge block. To see this value, refer to the source block.

Command Line Interface

You cannot specify port value displays through the command line interface.

Non-Simulink signals

You cannot place port values on non-Simulink signals, such as Simscape or SimEvents signals. This limitation applies to conditional breakpoints as well.

See Also

More About

- “Nonvirtual and Virtual Blocks” on page 36-2
- “Specify Block Properties” on page 36-4
- “Format a Model” on page 36-7
- “Control and Display Execution Order” on page 36-25
- “Access Block Data During Simulation” on page 36-37
- “Block Libraries”

Control and Display Execution Order

The execution order for a model is the order in which Simulink invokes the block output methods during simulation. Simulink determines this order during a model update, which you can initiate by clicking **Update Model** on the **Modeling** tab. Simulink also updates the model during simulation.

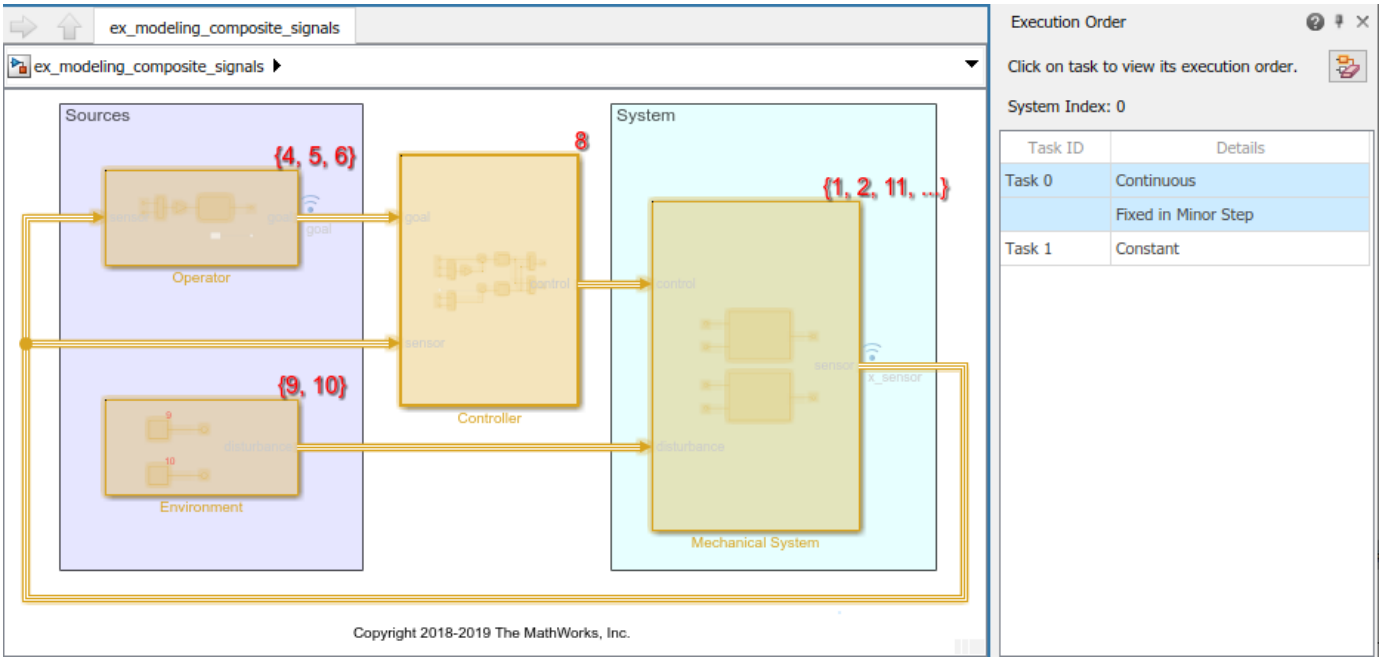
You cannot set the execution order, but you can assign priorities to nonvirtual blocks to indicate their execution order relative to other blocks in the corresponding block diagram. Simulink tries to honor block priority settings unless there is a conflict with data dependencies. To confirm the results of the priorities you have set or to debug your model, display and review the execution order of your nonvirtual blocks and subsystems.

Note For more information about block methods and execution, see:

- "Block Methods"
- "Conditionally Executed Subsystems Overview" on page 10-3

Execution Order Viewer

On the **Debug** tab, select **Information Overlays > Execution Order**. The Execution Order viewer opens in a pane on the right side of the Simulink Editor.



The Execution Order viewer displays a list of tasks for the current system.

Each **System Index** value corresponds to a group of blocks that execute independently from other blocks, such as the blocks in a nonvirtual subsystem. Within a model, the **System Index** values are unique. In model reference hierarchies, the same **System Index** value may be used multiple times, but it remains unique within each model in that hierarchy. To determine whether subsystems are part of the same system within a model, compare their **System Index** values.

Each task listed in the **Task ID** column corresponds to a group of blocks that share a sample rate. For fixed-step sizes, the **Treat each discrete rate as a separate task** configuration parameter determines whether Simulink executes blocks with discrete rates in one or multiple tasks.

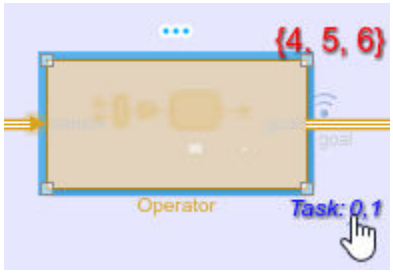
Selecting a task in the Execution Order viewer highlights the blocks that belong to the task and displays their execution order. By default, Simulink selects the first task that corresponds to the active block diagram.

When the active block diagram does not contain any blocks that execute as part of a given task, you cannot select the task in the Execution Order viewer. The active block diagram may contain virtual blocks, such as Inport blocks, that are part of this task.

To hide the highlighting and execution order, click the **Clear Highlighting** button  .

Navigation from Blocks to Tasks

To display the tasks in which a block executes, click on a block.



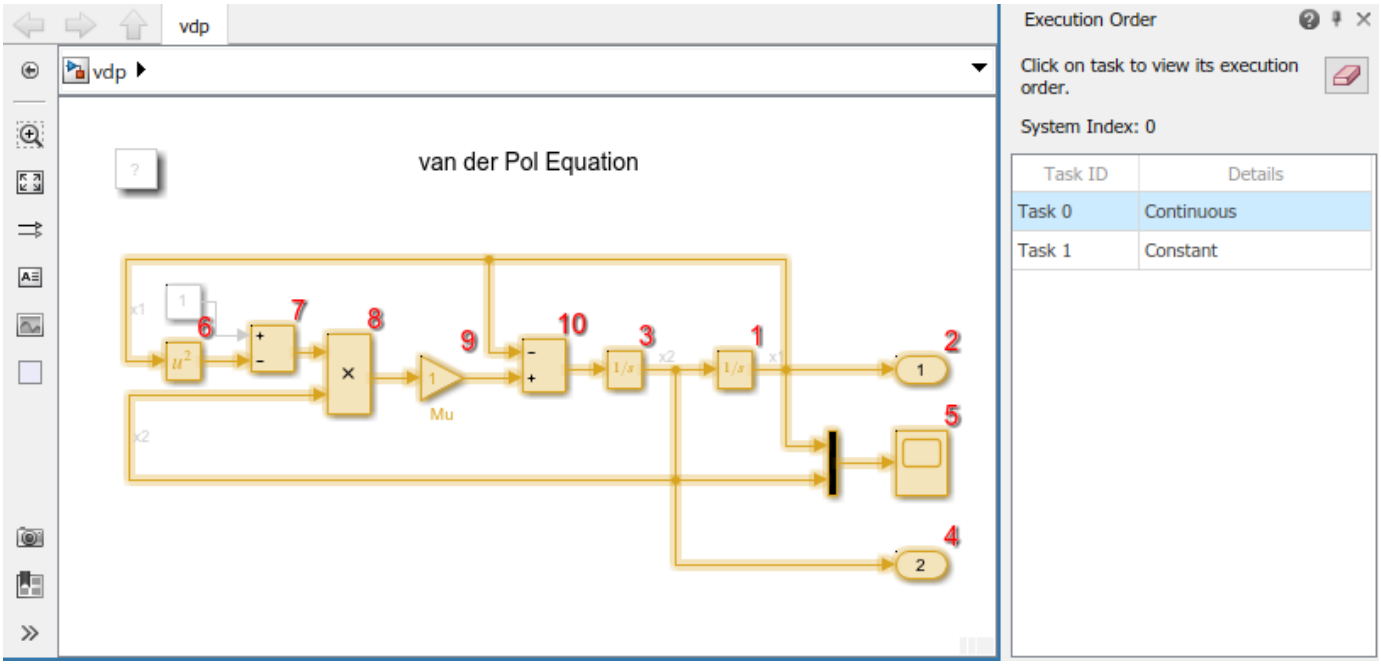
The task numbers in the label are links that you can click to select the corresponding task in the Execution Order viewer. When a model has many tasks, these links provide an alternative to scrolling through the list of tasks in the Execution Order viewer.

To compare tasks among blocks, select multiple blocks.

Execution Order Notation

Simulink displays a number at the top-right corner of each nonvirtual block. These numbers indicate the order in which the blocks execute. The first block to execute has the lowest execution order, which is usually `1`. The displayed execution order may skip numbers, but the blocks always execute in order of the visible numbers. Suppose a task displays execution orders 1, 2, and 4. The block labeled 1 executes before the block labeled 2, which executes before the block labeled 4.

For example, in the `vdp` model, the block execution order ranges from `1` to `9`, with each nonvirtual block receiving an execution order.

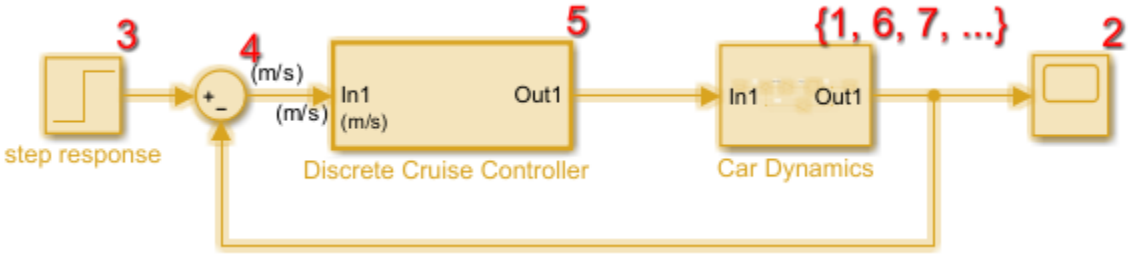


Virtual and Nonvirtual Subsystems

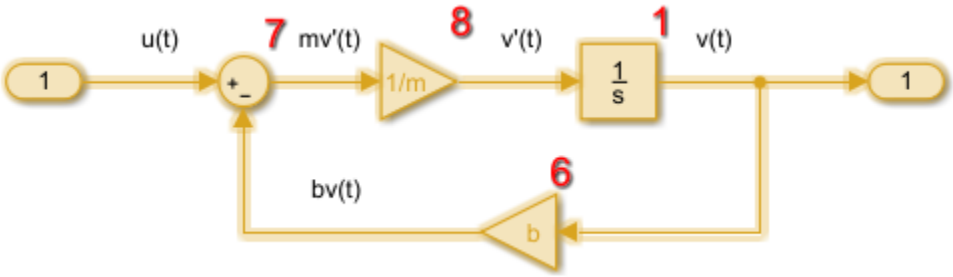
Virtual Subsystem blocks exist only graphically and do not execute. Consequently, they are not part of the execution order. The blocks inside a virtual subsystem have an execution order in the context of the root-level model. For virtual subsystems, block execution order within the subsystem is listed in curly brackets {}.

Nonvirtual Subsystem blocks exist functionally and execute as a unit. They have a single execution order and a different system index than the parent model. The blocks inside a nonvirtual subsystem have their own execution orders, which are independent of the parent model.

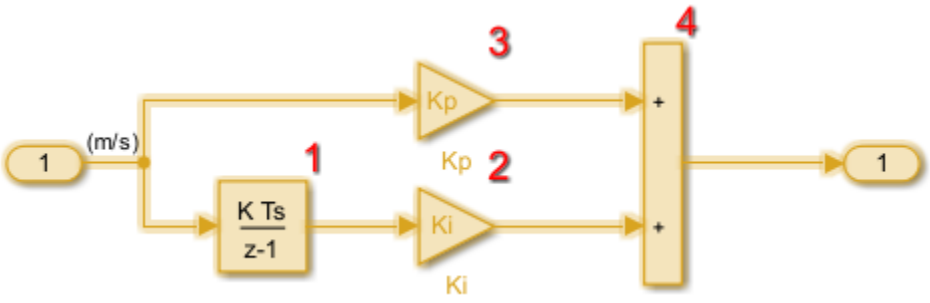
For example, the following model contains a virtual subsystem named Car Dynamics and an atomic, nonvirtual subsystem named Discrete Cruise Controller.



The virtual Car Dynamics subsystem shows a list of execution orders within curly brackets for the blocks it contains. The blocks it contains execute at the root level. The Integrator block executes first and sends its output to the Scope block in the root-level model, which executes second.



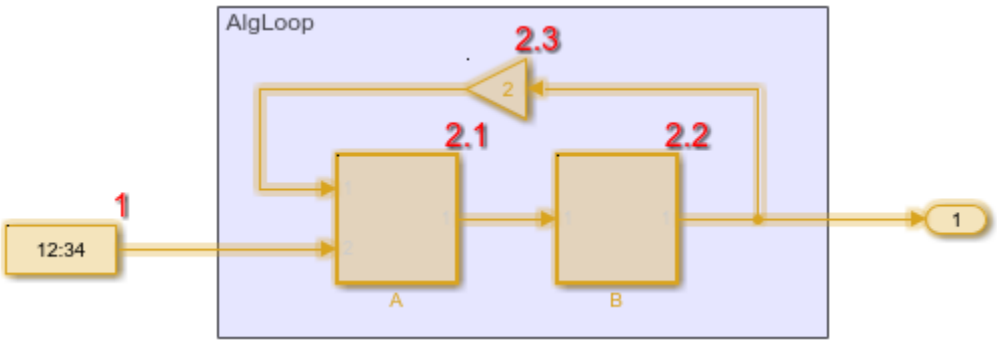
The nonvirtual Discrete Cruise Controller subsystem has a single execution order (5), which indicates that the subsystem and the blocks within it are the fifth to execute relative to the blocks at the root level.



Note Depending on your model configuration, Simulink can insert hidden, nonvirtual subsystems in your model. As a result, the visible blocks inside the hidden Subsystem block can have a system index that is different from the current system index. For example, if you select the **Conditional input branch execution** configuration parameter, Simulink creates hidden, nonvirtual subsystems, which can affect the sorted execution order.

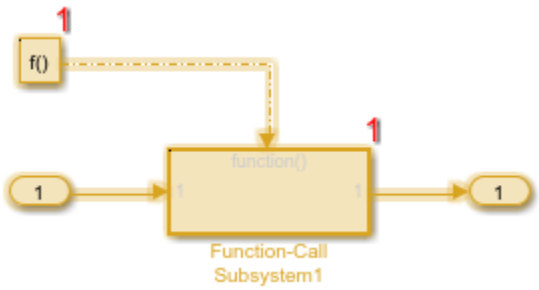
Algebraic Loops

Blocks within an algebraic loop are moved into a hidden nonvirtual subsystem. The execution order of the hidden subsystem is determined within the context of the other blocks; then, the execution order of the blocks within the hidden subsystem is determined.

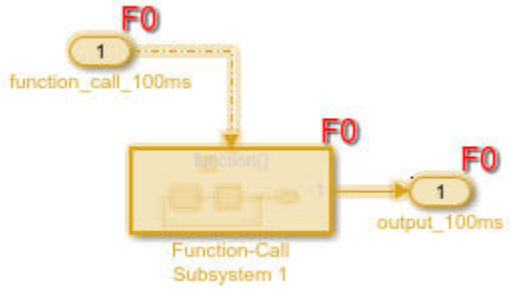


Function-Call and Action Subsystems

For function-call and action subsystems, the execution of the subsystem is tied to the execution of the initiator. The subsystem and its initiator therefore share an execution order.



At the root-level of export-function models, function-call execution orders have an F prefix.

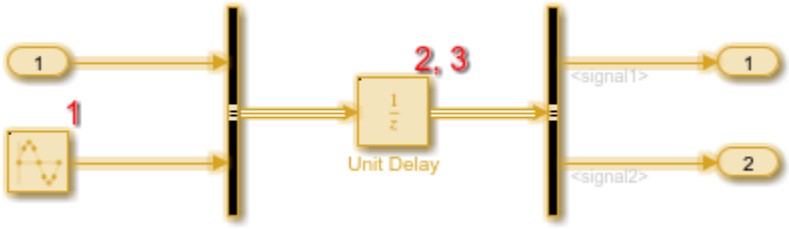


For more information, see “Export-Function Models Overview” on page 10-97.

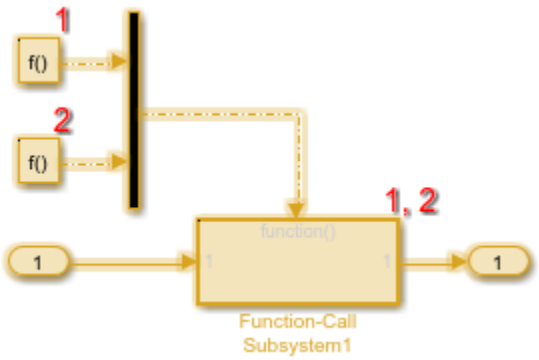
Buses and Multiple Initiators

A block has multiple execution orders when the block executes multiple times based on different execution paths to that block. For example:

- A block connected to a bus has an execution order corresponding to each signal that the bus contains.



- A function-call or action subsystem with multiple initiators has an execution order corresponding to each initiator.



How Simulink Determines Execution Order

Simulink uses task-based sorting to set the execution order of blocks and ports based on their derived sample time information. Task-based sorting provides an efficient and simpler process for determining the execution order of blocks.

With task-based sorting:

- Tasks are sorted individually based on sample time.
- Multiple sorted lists are generated instead of one flattened, sorted list of blocks across all tasks.
- Rate transition handling is simplified.
- False data dependency violations involving blocks in different tasks are avoided.
- Code generation results are in efficient rate groupings.
- One subsystem can belong to different sorted lists in multiple tasks.

Direct-Feedthrough Ports Impact on Execution Order

To ensure that the execution order reflects data dependencies among blocks, Simulink categorizes block input ports according to the dependency of the block outputs on the block input ports. An input port whose current value determines the current value of one of the block outputs is a *direct-feedthrough* port. Examples of blocks that have direct-feedthrough ports include:

- Gain
- Product
- Sum

Examples of blocks that have non-direct-feedthrough inputs include:

- Integrator — Output is a function of its state.
- Constant — Does not have an input.
- Memory — Output depends on its input from the previous time step.

Rules for Determining Block Execution Order

To sort blocks, Simulink uses the following rules:

- If a block drives the direct-feedthrough port of another block, the block must appear in the execution order ahead of the block that it drives.

This rule ensures that the direct-feedthrough inputs to blocks are valid when Simulink invokes block methods that require current inputs.

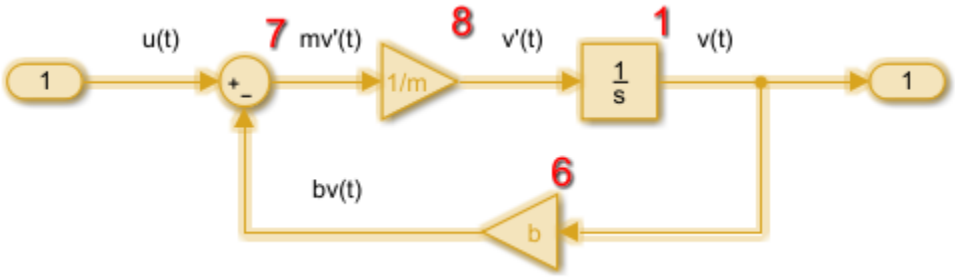
- Blocks that do not have direct-feedthrough inputs can appear anywhere in the execution order as long as they precede any direct-feedthrough blocks that they drive.

  Placing all blocks that do not have direct-feedthrough ports at the beginning of the execution order satisfies this rule. This arrangement allows Simulink to ignore these blocks during the sorting process.

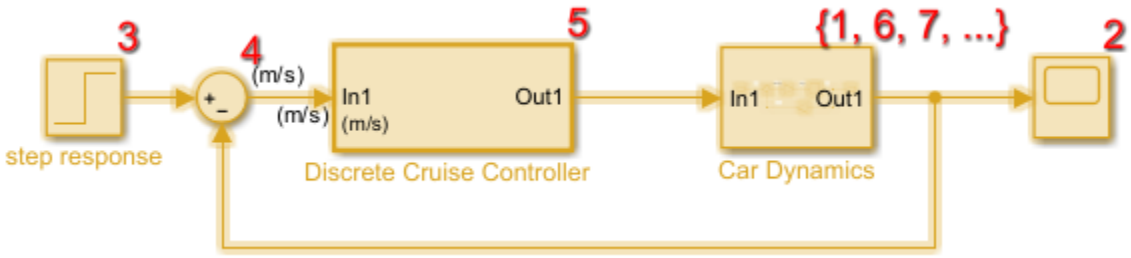
Applying these rules results in the execution order. Blocks without direct-feedthrough ports appear at the beginning of the list in no particular order. These blocks are followed by blocks with direct-feedthrough ports arranged such that they can supply valid inputs to the blocks which they drive.

The following model illustrates this result. The following blocks do not have direct-feedthrough and therefore appear at the beginning of the execution order of the root-level system:

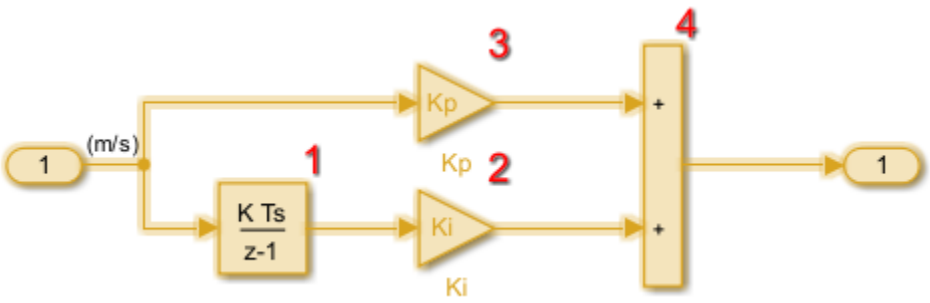
- Integrator block in the Car Dynamics virtual subsystem



- Speed block in the root-level model



Inside the Discrete Cruise Controller subsystem, all the Gain blocks, which have direct-feedthrough ports, run before the Sum block that they drive.



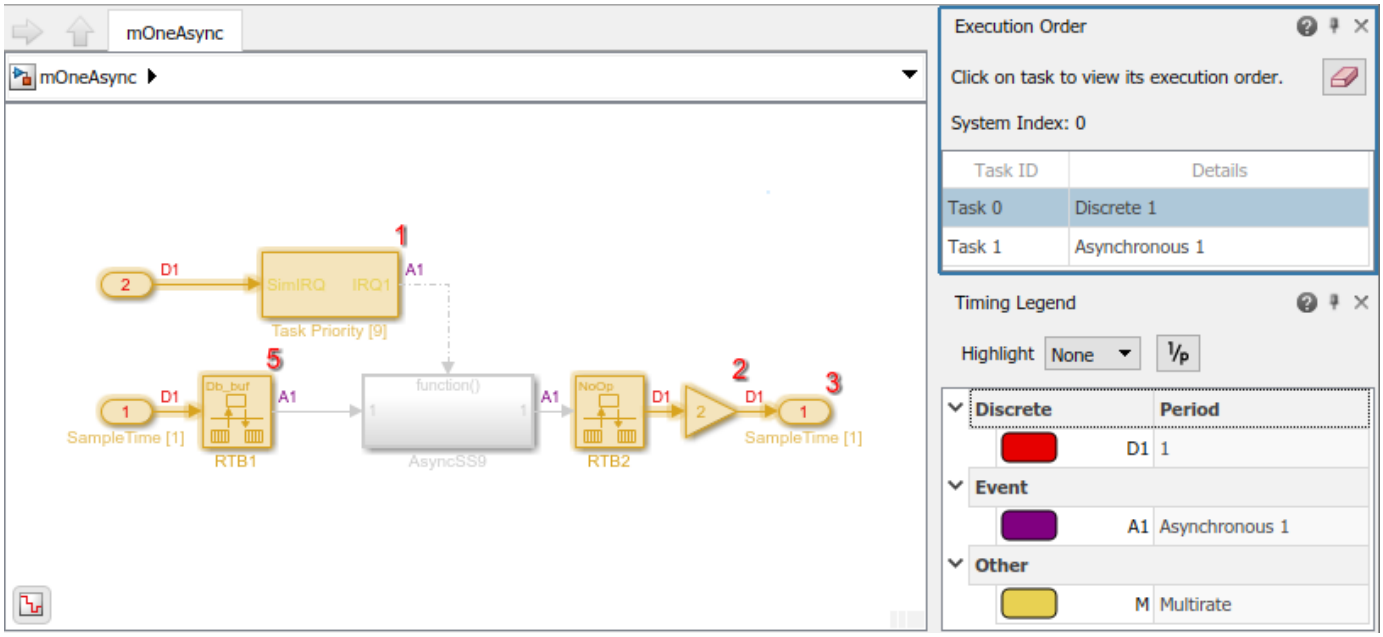
Rules for Determining Block Execution Order in Models Containing Asynchronous Tasks

In simulation, asynchronous function call initiators have the highest priority when determining block sorting order within a task. For more information, see “Asynchronous Sample Time” on page 7-15 and “Rate Transitions and Asynchronous Blocks” (Simulink Coder).

To determine the block execution order in models containing asynchronous sample times, Simulink uses the following rules:

- If an asynchronous function call initiator is triggered by a discrete rate shared by the corresponding rate transition block, then in the task of the common discrete rate, the asynchronous function call initiator will be sorted first.

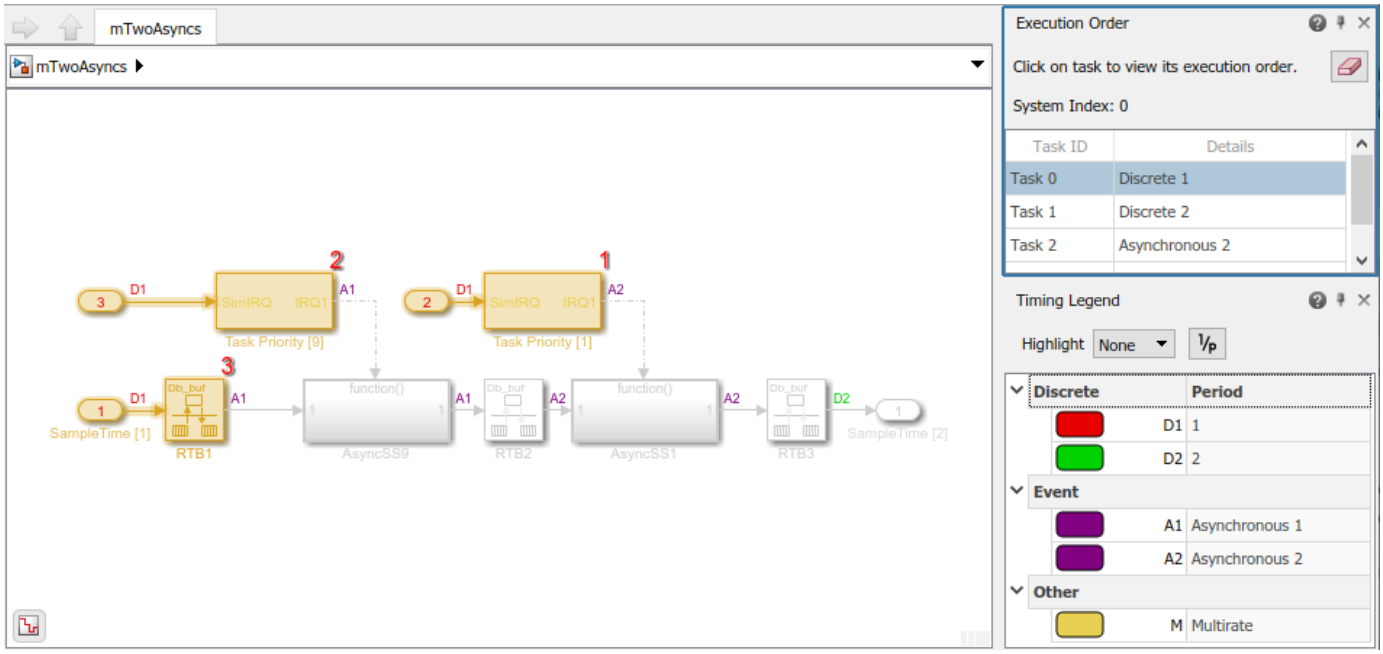
  For example, in the below model, there is one asynchronous function call initiator that is triggered by the discrete rate `D1`. Within the discrete task, the asynchronous function call initiator is sorted first. The Rate Transition blocks convert between the asynchronous rate, `A1`, and the discrete rate, `D1`.



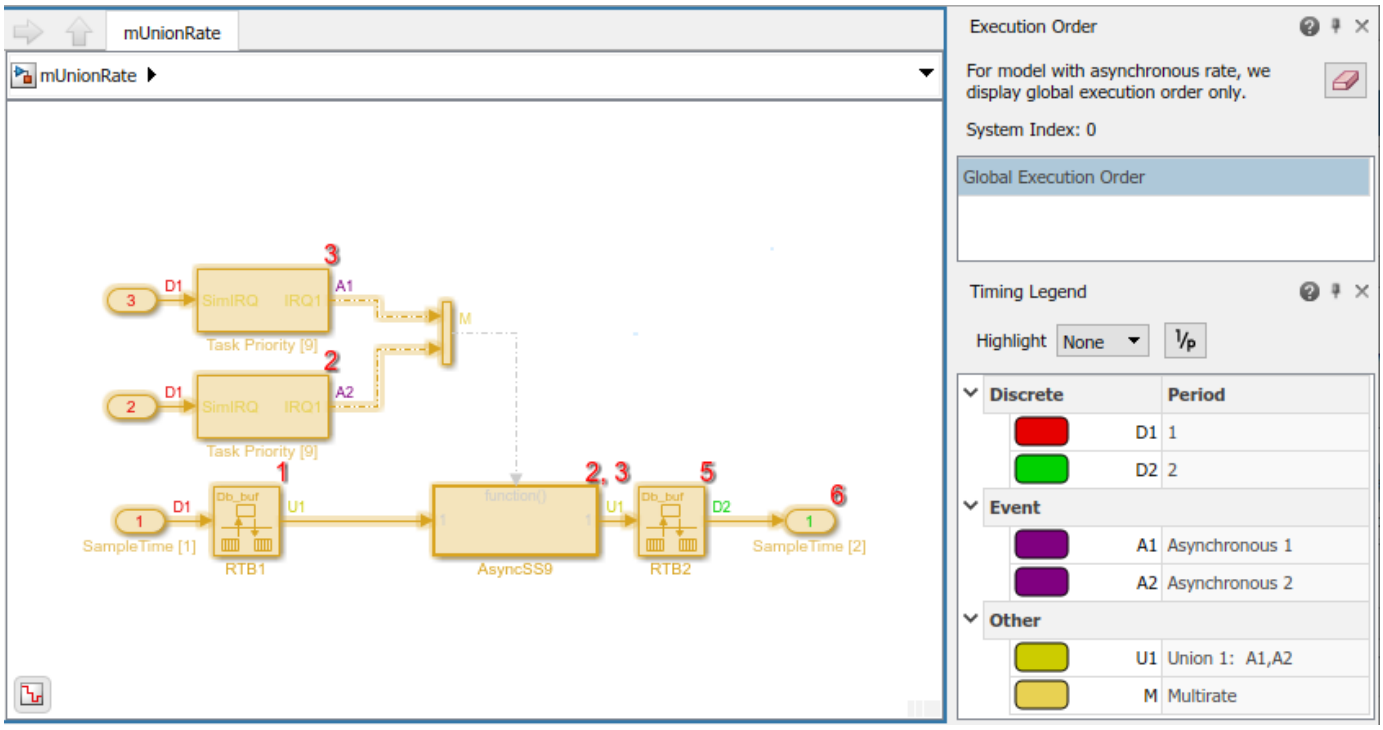
- If a Rate Transition block is reduced to `NoOp`, the Rate Transition block does not execute, and its position in the sorted order within the discrete task is transferred to the upstream or downstream discrete block to which it is connected.

  For example, in the above model, `RTB2` is reduced to `NoOp`, so its position in the sorted order is transferred to the downstream Gain block.
- If two asynchronous function call initiators share a common discrete rate task, then asynchronous function call initiator with higher priority executes first within the discrete rate task.

  For example, in the below model, two asynchronous function call initiators are triggered by the same discrete rate, `D1`. The one with higher task priority is sorted first.



- The union of asynchronous sample times is not supported by task-based sorting. Models containing the union of multiple asynchronous sample times default to a global execution order, in which all blocks are sorted within a single task.

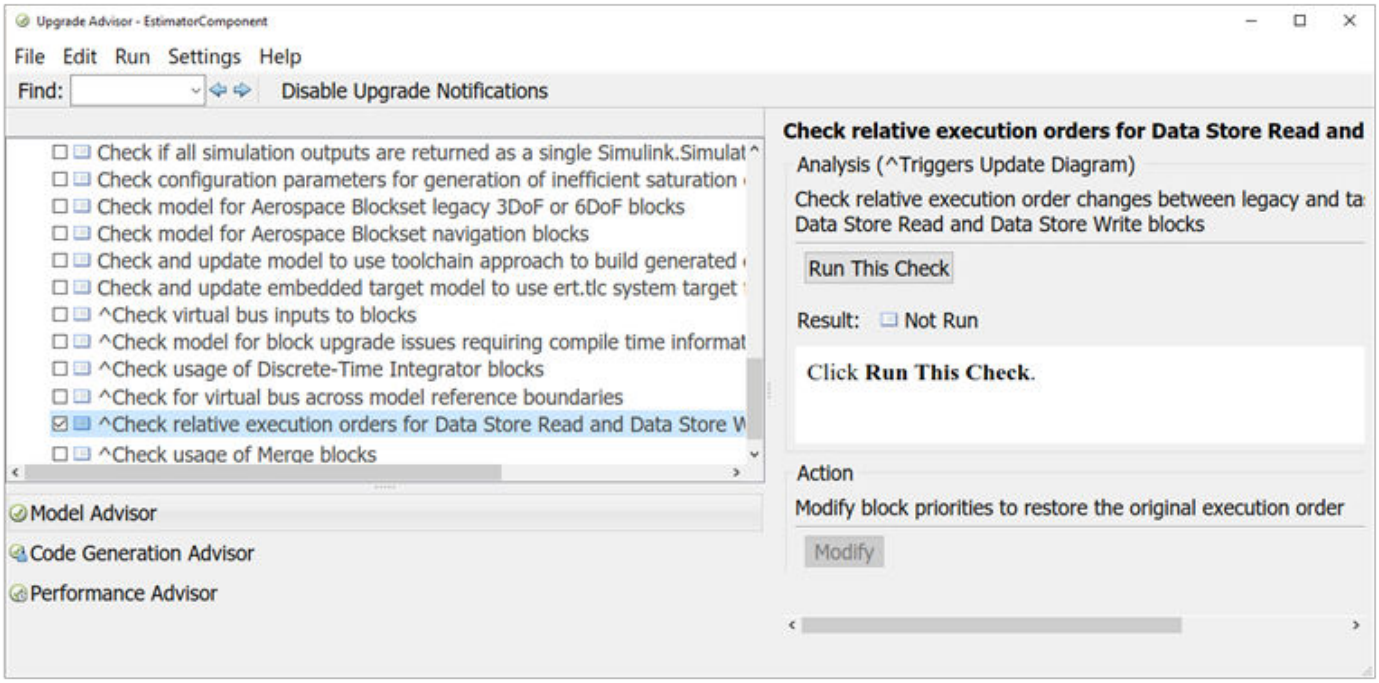


Checks for Execution Order Changes Involving Data Store Memory Blocks

Model Upgrades

Using task-based sorting instead of legacy (block) sorting can change the relative execution order involving Data Store Memory blocks. An Upgrade Advisor check detects the changes and provides an option to keep the original execution order when upgrading your model from earlier versions of Simulink.

1. Open the Upgrade Advisor. On the **Modeling** tab, select **Model Advisor > Upgrade Advisor**.
2. Select the check box for **Check relative execution orders for Data Store Read and Data Store Write blocks**.

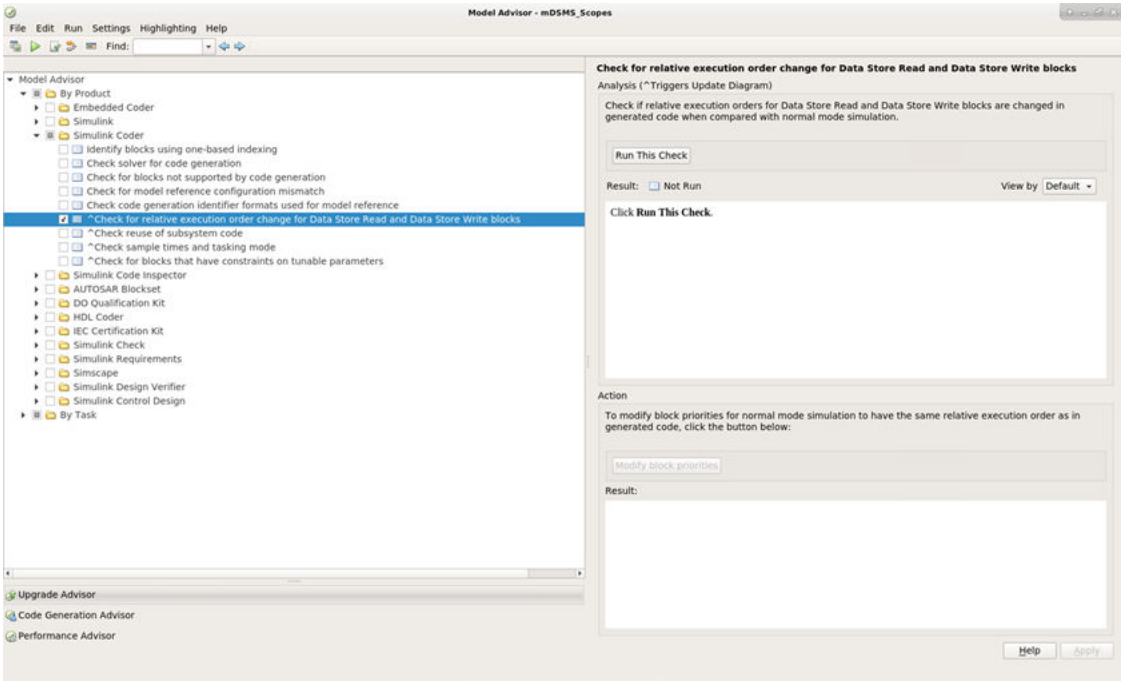


3. Click **Run This Check**.
4. Review any changes in the **Result** table. If you want to keep the original execution order, click **Modify**.

Code Generation

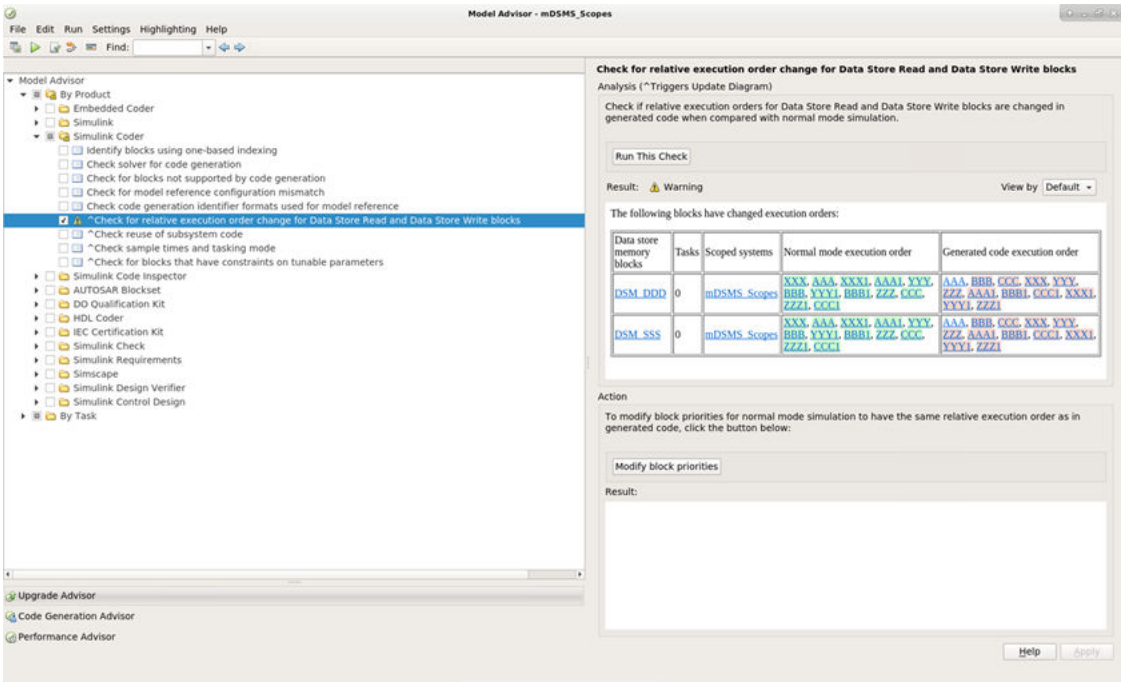
When a model is compiled for code generation, the relative execution order of Data Store Read and Data Store Write blocks can differ in the generated code from the order in normal simulation mode. A Model Advisor check detects these differences and provides an option to change the execution order in normal simulation mode to match the order in the generated code.

1. Open the Model Advisor. On the **Modeling** tab, select **Model Advisor > By Product > Simulink Coder**.
2. Select the check box for **Check for relative execution order change for Data Store Read and Data Store Write blocks**.



3 Click **Run This Check**.

4 Review any changes in the **Result** table. If there are discrepancies listed and you want to change the execution order in normal simulation to conform with the execution order in generated code, click **Modify block priorities**.



See Also

More About

- “Nonvirtual and Virtual Blocks” on page 36-2
- “Specify Block Properties” on page 36-4
- “Format a Model” on page 36-7
- “Display Port Values for Debugging” on page 36-16
- “Access Block Data During Simulation” on page 36-37

Access Block Data During Simulation

In this section...

About Block Run-Time Objects

Simulink provides an application programming interface, called the block run-time interface, that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to access block run-time data from the MATLAB command line, the Simulink Debugger, and from Level-2 MATLAB S-functions (see “Write Level-2 MATLAB S-Functions” in the online Simulink documentation).

Note You can use this interface even when the model is paused or is running or paused in the debugger.

The block run-time interface consists of a set of Simulink data object classes (see “Data Objects” on page 67-58) whose instances provide data about the blocks in a running model. In particular, the interface associates an instance of `Simulink.RunTimeBlock`, called the block's run-time object, with each nonvirtual block in the running model. A run-time object's methods and properties provide access to run-time data about the block's I/O ports, parameters, sample times, and states.

Access a Run-Time Object

Every nonvirtual block in a running model has a `RuntimeObject` parameter whose value, while the simulation is running, is a handle for the run-time object of the block. This allows you to use `get_param` to obtain a block's run-time object. For example, the following statement

```
rto = get_param(gcb,'RuntimeObject');
```

returns the run-time object of the currently selected block. Run-time object data is read-only. You cannot use run-time objects to change a block’s parameters, input, output, and state data.

Note Virtual blocks (see “Nonvirtual and Virtual Blocks” on page 36-2) do not have run-time objects. Blocks eliminated during model compilation as an optimization also do not have run-time objects (see “Block reduction”). A run-time object exists only while the model containing the block is running or paused. If the model is stopped, `get_param` returns an empty handle. When you stop a model, all existing handles for run-time objects become empty.

Listen for Method Execution Events

One application for the block run-time API is to collect diagnostic data at key points during simulation, such as the value of block states before or after blocks compute their outputs or derivatives. The block run-time API provides an event-listener mechanism that facilitates such

applications. For more information, see the documentation for the `add_exec_event_listener` command. For an example of using method execution events, enter

```
sldemo_msfcn_lms
```

at the MATLAB command line. This Simulink model contains the S-function `adapt_lms.m`, which performs a system identification to determine the coefficients of an FIR filter. The S-function's `PostPropagationSetup` method initializes the block run-time object's DWork vector such that the second vector stores the filter coefficients calculated at each time step.

In the Simulink model, double-clicking on the annotation below the S-function block executes its `OpenFcn`. This function first opens a figure for plotting the FIR filter coefficients. It then executes the function `add_adapt_coef_plot.m` to add a `PostOutputs` method execution event to the S-function's block run-time object using the following lines of code.

```
% Add a callback for PostOutputs event
blk = 'sldemo_msfcn_lms/LMS Adaptive';

h   = add_exec_event_listener(blk, ...
         'PostOutputs', @plot_adapt_coefs);
```

The function `plot_adapt_coefs.m` is registered as an event listener that is executed after every call to the S-function's `Outputs` method. The function accesses the block run-time object's DWork vector and plots the filter coefficients calculated in the `Outputs` method. The calling syntax used in `plot_adapt_coefs.m` follows the standard needed for any listener. The first input argument is the S-function's block run-time object, and the second argument is a structure of event data, as shown below.

```
function plot_adapt_coefs(block, ei) %#ok<INUSD>
%
% Callback function for plotting the current adaptive filtering
% coefficients.

stemPlot  = get_param(block.BlockHandle,'UserData');

est = block.Dwork(2).Data;
set(stemPlot(2),'YData',est);
drawnow('expose');
```

Synchronizing Run-Time Objects and Simulink Execution

You can use run-time objects to obtain the value of a block output and display in the MATLAB Command Window by entering the following commands.

```
rto = get_param(gcb,'RuntimeObject')
rto.OutputPort(1).Data
```

However, the displayed data may not be the true block output if the run-time object is not synchronized with the Simulink execution. Simulink only ensures the run-time object and Simulink execution are synchronized when the run-time object is used either within a Level-2 MATLAB S-function or in an event listener callback. When called from the MATLAB Command Window, the run-time object can return incorrect output data if other blocks in the model are allowed to share memory.

To ensure the `Data` field contains the correct block output, open the Configuration Parameters dialog box, and then clear the **Signal storage reuse** check box (see “Signal storage reuse” (Simulink Coder)).

See Also

More About

- “Nonvirtual and Virtual Blocks” on page 36-2
- “Specify Block Properties” on page 36-4
- “Format a Model” on page 36-7
- “Display Port Values for Debugging” on page 36-16
- “Control and Display Execution Order” on page 36-25

Working with Block Parameters

Set Block Parameter Values

Blocks have numeric parameters that determine how they calculate output values. To control the calculations that blocks perform, you can specify parameter values. For example, a Gain block has a **Gain** parameter, and a Transfer Fcn block has multiple parameters that represent the transfer function coefficients.

You can use numbers, variables, and expressions to set block parameter values. Choose a technique based on your modeling goals. For example, you can:

- Share parameter values between blocks and models by creating variables.
- Control parameter characteristics such as data type and dimensions by creating parameter objects.
- Model an algorithm as code by using mathematical expressions.

Set block parameters using the **Parameters** tab in the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**), or the block dialog box. For more information, see “Add Blocks and Set Parameters” on page 1-13. To set block sample times, see “Specify Sample Time” on page 7-3.

Tip You can use the Model Explorer to make batch changes to many block parameter values at once. For more information, see **Model Explorer**.

Programmatically Access Parameter Values

To programmatically access block parameter values, use the `get_param` and `set_param` functions. You can use this technique to:

- Construct a model programmatically.
- Adjust parameter values during a simulation run when you simulate a model programmatically.

To sweep parameter values between simulation runs by using a script, use `Simulink.SimulationInput` objects instead of `get_param` and `set_param`. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

Suppose you create a model named `myModel` that contains a Constant block named My Constant. Next, you use the block dialog box to set the **Constant value** parameter to `15`. To programmatically return the parameter value, use the function `get_param`. You specify the block path and the equivalent programmatic parameter name, `Value`.

```
paramValue = get_param('myModel/My Constant','Value')
```

```
paramValue =

15
```

To programmatically change the value, for example to `25`, use the function `set_param`. Use the character vector `'25'` as the input to the function.

```
set_param('myModel/My Constant','Value','25')
```

For a list of programmatic names of block parameters, see “Block-Specific Parameters”.

For more information about programmatic simulation, see “Run Simulations Programmatically” on page 26-2.

To avoid using the `get_param` and `set_param` functions, use the name of a MATLAB variable or `Simulink.Parameter` object as the parameter value, and change the value of the variable or object at the command prompt. See “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9.

Specify Parameter Values

Goal	Block Parameter Value	Description

Store the parameter value in the model file.	`2.3`
	
	`[1.2 2.3 4.5; 7.9 8.7 6.5]`
	
	`2 + 3i`
		Literal numeric value. Specify a scalar, vector, matrix, or multidimensional array. Use `i` to specify complex values.

• Access the parameter value without having to locate or identify the block in the model. • Change the parameter value without having to modify the model file. • Share the parameter value between blocks or between models. • Identify the parameter by a specific name when sweeping or tuning the value.	`myVar`	MATLAB variable that exists in a workspace. For more information, see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9.
• Avoid name clashes between workspace variables. • Organize parameter values using hierarchies and meaningful names. • Reduce the number of workspace variables that a model uses.	`myParam.a.SpeedVect`	Field of parameter structure. For more information, see “Organize Related Block Parameter Definitions in Structures” on page 37-19.
Use a portion of a matrix or array variable. For example, set the parameters of a n-D Lookup Table block.	`myMatrixParam(:,2)`	Index operation.

Goal	Block Parameter Value	Description
• Define parameter characteristics, such as data type, complexity, units, allowed value range, and dimensions, separately from the parameter value. • Define a system constant with custom documentation. • Create a tunable parameter in the generated code. • Set the value of a variable to a mathematical expression involving constants and other variables.	`myParam`	Parameter object. For more information, see “Use Parameter Objects” on page 37-4.

• Express a parameter value as a mathematical relationship between known physical constants instead of as an unidentifiable literal number.
• Reduce block population in a model.
• Model an obscure or trivial calculation by using code instead of blocks.
• Use MATLAB operators and functions to perform calculations.
• Write a custom MATLAB function that calculates parameter values.
	`5^3.2 - 1/3`
	
	`myParam * myOtherparam + sin(0.78*pi)`
	
	`myFun(15.23)`
		Expression or custom function.
		
		For more information, see “Use Mathematical Expressions, MATLAB Functions, and Custom Functions” on page 37-5.

Specify a block parameter value by using a data type other than `double`.	`15.23`
	
	`single(15.23)`
	
	`myParam`
		Typed or untyped expression, numeric MATLAB variable, or parameter object.
		
		For more information about controlling parameter data types, see “Control Block Parameter Data Types” on page 37-44.

Use Parameter Objects

Parameter objects are `Simulink.Parameter` objects and objects of the subclasses that you create. The parameter object exists in a workspace such as the base workspace or a data dictionary.

You can use parameter objects to define system constants. For example, use a parameter object to represent the radius of the Earth. Use the properties of the object to specify the physical units and to document the purpose of the value.

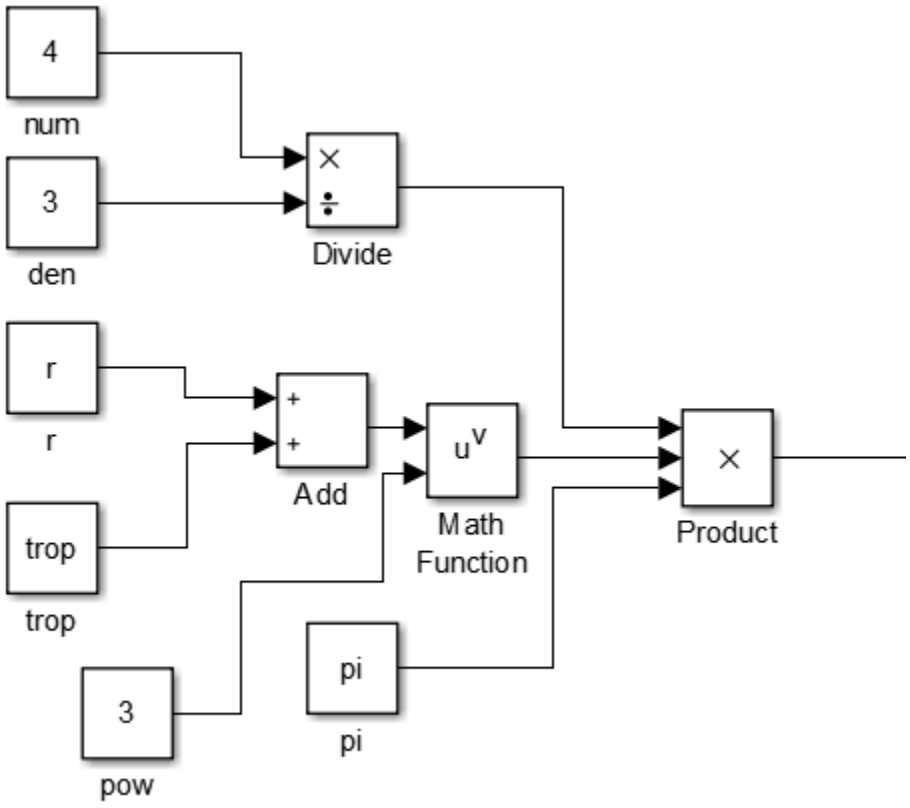
Create parameter objects to prepare your model for code generation. You can configure parameter objects to appear as tunable global variables in the generated code. You can also control the parameter data type through the object.

To create and use parameter objects in models, see “Data Objects” on page 67-58. For information about using variables to set block parameter values, see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9.

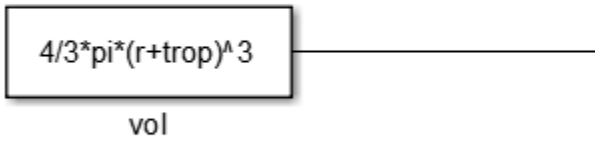
Use Mathematical Expressions, MATLAB Functions, and Custom Functions

You can set a block parameter value to an expression that calls MATLAB functions and operators such as `sin` and `max`. You can also call your own custom functions that you write on the MATLAB path.

Suppose that a section of your block algorithm uses variables to calculate a single constant number used by the rest of the algorithm. You can perform the calculation by creating multiple blocks.



Instead, create a single Constant block that uses an expression written in MATLAB code. This technique reduces the size of the block algorithm and improves readability.



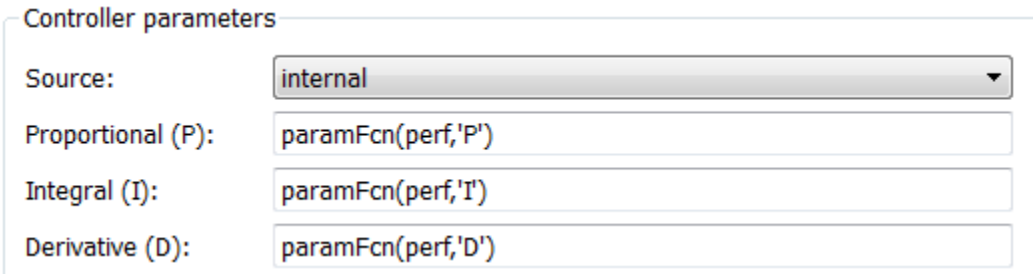
You can model a complicated portion of an algorithm by using an expression instead of many blocks. To operate on an existing signal, use a mathematical expression as the value of a parameter in an algorithmic block, such as the **Gain** parameter of a Gain block.

With expressions, you can also call your custom functions to set block parameter values. Suppose that you write a MATLAB function that calculates optimal P, I, and D parameters for a control algorithm by accepting a single input number.

paramFcn.m

```
function paramValue = paramFcn(performance,parameter)
%PARAMVALUE = paramFcn(PERFORMANCE,PARAMETER) returns the P, I, or D
%value PARAMVALUE that you request using PARAMETER. Specify performance
%criteria using PERFORMANCE.

% Initialize output.
paramValue = [];

% Calculate parameter value.
switch parameter
    case 'P'
        paramValue = performance*3;
    case 'I'
        paramValue = sqrt(performance)/0.175;
    case 'D'
        paramValue = performance^0.25;

% Validation
```

You can parameterize a PID Controller block by using the function to set the parameter values.



To make the best use of expressions, consider these tips:

- If you use variables and parameter objects, you can explicitly model the algebraic relationships between the real-world quantities that the variables and objects represent. Use expressions in parameter objects as described in “Set Variable Value by Using a Mathematical Expression” on page 37-10.
- While you edit an expression in a block parameter value, to navigate to the documentation for a function, use the button ⋮ next to the parameter value. You can also navigate to the source code of a custom function.

Considerations for Other Modeling Goals

Choose a technique to set block parameter values based on your modeling goals.

Goal	Features or Products	Best Practice
Run multiple simulations quickly.	`Simulink.SimulationInput` objects and the `sim` function	Use variables or parameter objects to set block parameter values. This technique helps you to assign meaningful names to the parameters and to avoid having to identify or locate the blocks in the model. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.
Sweep parameter values during testing.	Simulink Test	Use variables or parameter objects to set block parameter values. Use iterations and parameter overrides to run multiple tests. See “Parameter Overrides” (Simulink Test) and “Test Iterations” (Simulink Test).
Estimate and optimize parameter values.	Simulink Design Optimization™	Use variables or parameter objects to set block parameter values. To estimate or optimize a parameter that uses a data type other than `double`, use a parameter object to separate the value from the data type. For parameter estimation, see “Parameter Estimation” (Simulink Design Optimization). For response optimization, see “Optimize Model Response” (Simulink Design Optimization).

Goal	Features or Products	Best Practice
Generate code from a model. Simulate an external program through SIL/PIL or External mode simulations.	Simulink Coder	Use parameter objects to set block parameter values. This technique helps you to declare and identify tunable parameters in the generated code and to control parameter data types. See “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder). When you use expressions to set block parameter values, avoid using operators and functions that result in loss of tunability in the generated code. See “Tunable Expression Limitations” (Simulink Coder).

See Also

`set_param`

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Organize Related Block Parameter Definitions in Structures” on page 37-19
- “Tune and Experiment with Block Parameter Values” on page 37-31
- “Block-Specific Parameters”

Share and Reuse Block Parameter Values by Creating Variables

To set a block parameter value, such as the **Gain** parameter of a Gain block, you can use numeric variables that you create and store in workspaces such as the base workspace, a model workspace, or a Simulink data dictionary. You can use the variable to set multiple parameter values in multiple blocks, including blocks in different models. To change the values of the block parameters, you change the value of the variable in the workspace.

Using a variable to set a block parameter value also enables you to:

- Change the parameter value without having to modify the model file (if you store the variable outside the model workspace).
- Identify the parameter by a specific, meaningful name when sweeping or tuning the value.

For basic information about setting block parameter values, see “Set Block Parameter Values” on page 37-2.

Reuse Parameter Values in Multiple Blocks and Models

You can create a numeric MATLAB variable in a workspace, such as the base workspace or a data dictionary, and use it to specify one or more block parameter values.

If a block parameter value is set to a simple numeric expression, you can create a variable for that expression in the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). Click ⋮ in the right side of the cell that corresponds to the value, then select **Create variable**. In the Create New Data dialog box, set the name and location for the new variable, then click **Create**. The cell now displays the new variable.

You can also create a variable to represent a constant that is used in multiple expressions. The example model `sldemo_fuelsys` represents the fueling system of a gasoline engine. A subsystem in the model, `feedforward_fuel_rate`, calculates the fuel demand of the engine by using the constant number `14.6`, which represents the ideal (stoichiometric) ratio of air to fuel that the engine consumes. Two blocks in the subsystem use the number to set the values of parameters. In this example, to share the number between the blocks, you create a variable named `myParam`.

1. Open the model.

   ```
   sldemo_fuelsys
   ```

2. In the model, on the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, inspect the **Parameters** tab.
3. In the model, navigate into the subsystem.

   ```
   open_system(...
       'sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate')
   ```

4. In the Model Data Editor, in the **Filter contents** box, enter `14.6`.

   The data table contains two rows, which correspond to the **Constant value** parameters of two of the Constant blocks in the subsystem.
5. Use the **Value** column to replace the literal number `14.6` with `myParam`. Perform the replacement for both parameters.
6. In the **Filter contents** box, enter `myParam`.

7. While editing the value of one of the parameters, click the action button ⋮ and select **Create**.
8. In the **Create New Data** dialog box, set **Value** to `14.6` and click **Create**.

   The variable, `myParam`, appears in the base workspace.

Because the variable exists in the base workspace, you can use it in multiple models. However, when you end your MATLAB session, you lose the contents of the base workspace. Consider permanently storing the variable in a model workspace or data dictionary.

Define a System Constant

To define a system constant, such as a variable that represents the radius of the Earth, consider creating a `Simulink.Parameter` object instead of a numeric MATLAB variable. Parameter objects allow you to specify physical units and custom documentation as well as other characteristics.

- To create and use parameter objects in models, see “Data Objects” on page 67-58.
- Typically, the value of a system constant influences the values of other parameters and signals through mathematical relationships. To model these relationships explicitly, set the values of the dependent data by using expressions. See “Set Variable Value by Using a Mathematical Expression” on page 37-10.

Set Variable Value by Using a Mathematical Expression

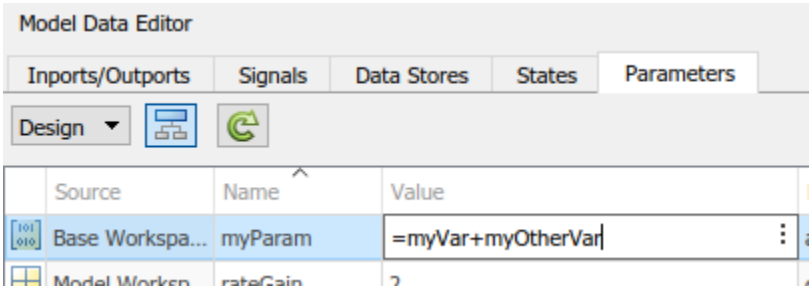
You can set the value of a variable to an expression involving literal numbers and other variables. With expressions, you can:

- Express the value as a relationship between known physical constants instead of as an unidentifiable literal number.
- Explicitly model algebraic dependencies between parameter data. When you change the values of independent data, you do not need to remember to adjust the values of dependent data.

General Technique

Convert the variable to a `Simulink.Parameter` object. Then, set the `Value` property of the object by using an expression:

- Interactively — For example, with the Model Data Editor or the Model Explorer, precede the expression with an equals sign, `=`. The figure shows how to specify the expression `myVar + myOtherVar`.



- Programmatically — Use the `slexpr` function, specifying the expression in a character vector or string. For example, to set the value of a parameter object named `myParam` to the expression `myVar + myOtherVar`:

  ```
  myParam.Value = slexpr('myVar + myOtherVar')
  ```

Explicitly Model Algebraic Relationship Between Variables

The example `sldemo_metro` (see "Exploring the Solver Jacobian Structure of a Model") models a system of three identical, pointlike metronomes suspended from a moving platform. Blocks in the model use these MATLAB variables from the base workspace:

- `m` — Mass of each metronome, initial value 0.1 kg
- `r` — Length of each metronome, initial value 1.0 m
- `J` — Moment of inertia of each metronome, initial value 0.1 kg/m^2

These variables share an algebraic relationship: the moment of inertia of each metronome is equal to the mass times the length squared. In this example, you record this relationship in the value of `J`.

1. Open the model.

   ```
   sldemo_metro
   ```
2. Update the block diagram. A model callback creates the variables in the base workspace.
3. To prevent the callback from overwriting changes that you make to the variables, for this example, remove the callback code.

   ```
   set_param('sldemo_metro','InitFcn','')
   ```
4. In the model, on the **Modeling** tab, click **Model Data Editor**.
5. On the Model Data Editor **Parameters** tab, activate the **Change scope** button.

   The blocks that use the variables are in the subsystems, so you must configure the Model Data Editor to show data in subsystems.
6. Click the **Show/refresh additional information** button.

   The data table contains rows that correspond to the variables in the base workspace.
7. In the **Filter contents** box, enter `J`.
8. In the data table, find the row that corresponds to `J`. In the **Value** column, set the value of the variable to `Simulink.Parameter(J)`.

   Simulink converts `J` to a `Simulink.Parameter` object.
9. In the **Value** column, set the value of the parameter object to `=m*r^2`.
10. Optionally, simulate the model with different metronome masses and lengths. As you change the values of `m` and `r`, you do not have to remember to correct the value of `J`.

Limitations and Considerations for Other Modeling Goals

- If the expression contains fixed-point data or data of an enumerated type, the expression can operate on only one variable or object.
- You cannot set the data type (`DataType` property) of the parameter object that uses the expression to `auto` (the default) and set the data types of parameter objects that appear in the expression to a value other than `auto`. For example, in the expression `J = m*r^2`, you cannot set the data type of `J` to `auto` and the data types of `m` and `r` to `single`.

  - To retain the benefits of `auto` (described in “Context-Sensitive Data Typing” on page 37-44) for the object that uses the expression, set the data types of the objects in the expression to `auto`. In other words, use `auto` for all of the involved objects. The objects in the expression acquire the same data type as the object that uses the expression.
  - To use a value other than `auto` for an object that appears in the expression, set the data types of all dependent parameter objects to a value other than `auto`. In other words, do not use `auto` for any involved objects.

    You must use the same data type for all objects used in the expression.
- If you have Simulink Coder and Embedded Coder licenses, you can generate code that initializes a global variable by using the expression. However, the code generator can preserve the expression only if it conforms to certain requirements. See “Expression Preservation” (Simulink Coder).

Control Scope of Parameter Values

The scope of a variable is the set of models and blocks that can use the variable. For example, variables that you create in the base workspace have global scope because all blocks in all open models can use the variables. Variables that you store in a model workspace have limited scope because only the blocks in the host model can use the variables.

You cannot create two variables that have the same name in the same scope. Controlling the scope of a variable helps you to avoid name conflicts and establish clear ownership of the variable.

The table describes the different ways that you can control the scope of a reusable parameter value.

Scope	Technique
All open models	Create a variable in the base workspace.
One or more targeted models	Create a variable in a data dictionary. To reuse the variable in multiple models, create a referenced dictionary. See “What Is a Data Dictionary?” on page 74-2
One model, including all subsystems in the model	Create a variable in the model workspace. See “Model Workspaces” on page 67-119.
Multiple blocks inside a subsystem, including blocks in nested subsystems	Mask the subsystem and create a mask parameter instead of a workspace variable. To prevent blocks inside a subsystem from using workspace variables, in the subsystem block dialog box, set **Permit Hierarchical Resolution** to `None`. This technique allows you to use the same name to create both a variable in a workspace and a mask parameter in the subsystem mask. The blocks in the subsystem can use only the mask parameter. For information about subsystems, see Subsystem. For information about masking, see “Masking Fundamentals” on page 39-2.

To avoid name conflicts when you have a large model with many variables in the same scope, consider packaging the variables into a single structure. For more information, see “Organize Related Block Parameter Definitions in Structures” on page 37-19.

For basic information about how blocks use the variable names that you specify, see “Symbol Resolution” on page 67-127.

Permanently Store Workspace Variables

Variables that you create in the base workspace do not persist between MATLAB sessions. However, you can store the variables in a MAT-file or script file, and load the file whenever you open the model using a model callback. A model callback is a set of commands that Simulink executes when you interact with a model in a particular way, such as opening the model. You can use a callback to load variables when you open the model. Use this technique to store variables while you learn about Simulink and experiment with models.

1. In a model that contains a Gain block, set the value of the **Gain** parameter to `K`.
2. At the command prompt, create a variable `K` in the base workspace.

   ```
   K = 27;
   ```
3. In the Workspace browser, right-click the variable and select **Save As**.

   To save multiple variables in one file, select all of the target variables in the Workspace browser, and then right-click any of the selected variables.
4. In the dialog box, set **Save as type** to `MATLAB Script`. Set **File name** to `loadvar` and click **Save**.

   The script file `loadvar.m` appears in your current folder. You can open the file to view the command that creates the variable `K`.
5. In the model, on the **Modeling** tab, select **Model Settings** > **Model Properties**.
6. In the **Callbacks** tab of the Model Properties dialog box, select `PreLoadFcn` as the callback that you want to define. In the **Model pre-load function** pane, enter `loadvar` and click **OK**.
7. Save the model.

The next time that you open the model, the `PreloadFcn` callback loads the variable `K` into the base workspace. You can also save the variable to a MAT-file, for example `loadvar.mat`, and set the model callback to `load loadvar`.

To learn about callbacks, see “Callbacks for Customized Model Behavior” on page 4-44. To programmatically define a callback for loading variables, see “Programmatically Store Workspace Variables for a Model” on page 37-13.

When you save variables to a file, you must save the changes that you make to the variables during your MATLAB session. To permanently store variables for a model, consider using a model workspace or a data dictionary instead of a MAT-file or script file. For more information about permanently storing variables, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.

Programmatically Store Workspace Variables for a Model

In the example above, you define a model callback that creates variables when you open a model. You can programmatically save the variable and set the model callback.

1. At the command prompt, create the variable K in the base workspace.

   ```
   K = 27;
   ```
2. Save the variable to a new script file named `loadvar.m`.

   ```
   matlab.io.saveVariablesToScript('loadvar.m','K')
   ```
3. Set the model callback to load the script file.

   ```
   set_param('mymodel','PreloadFcn','loadvar')
   ```
4. Save the model.

   ```
   save_system('myModel')
   ```

The function `matlab.io.saveVariablesToScript` saves variables to a script file. To save variables to a MAT-file, use the function `save`. To programmatically set model properties such as callbacks, use the function `set_param`.

Manage and Edit Workspace Variables

When you use variables to set block parameter values, you store the variables in a workspace or data dictionary. You can use the command prompt, the Model Explorer, and the Model Data Editor to create, move, copy, and edit variables. You can also determine where a variable is used in a model, list all of the variables that a model uses, and list all of the variables that a model does not use. For more information, see “Create, Edit, and Manage Workspace Variables” on page 67-106.

Package Shared Breakpoint and Table Data for Lookup Tables

To share breakpoint vectors or table data between multiple n-D Lookup Table, Prelookup, and Interpolation Using Prelookup blocks, consider storing the data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects instead of MATLAB variables or `Simulink.Parameter` objects. This technique improves model readability by clearly identifying the data as parts of a lookup table and explicitly associating breakpoint data with table data.

Store Standalone Lookup Table in Simulink.LookupTable Object

A standalone lookup table consists of a set of table data and one or more breakpoint vectors. You do not share the table data or any of the breakpoint vectors with other lookup tables.

When you share a standalone lookup table, you use all of the table and breakpoint data together in multiple n-D Lookup Table blocks. To store this data in a `Simulink.LookupTable` object:

1. Create the object in a workspace or data dictionary. For example, at the command prompt, enter:

   ```
   myLUTObj = Simulink.LookupTable;
   ```
2. Use the properties of the object to store the values of the table and breakpoint data.
3. Use the properties of the object to configure a unique name for the structure type in the generated code. In the property dialog box, under **Struct Type definition**, specify **Name**.
4. In the n-D Lookup Table blocks, set **Data specification** to `Lookup table object`.
5. To the right of **Data specification**, set **Name** to the name of the `Simulink.LookupTable` object.

For ways to create and configure `Simulink.LookupTable` objects, see `Simulink.LookupTable`

Store Shared Data in Simulink.LookupTable and Simulink.Breakpoint Objects

When you use Prelookup and Interpolation Using Prelookup blocks to more finely control the lookup algorithm, you can share breakpoint vectors and sets of table data. For example, you can share a breakpoint vector between two separate sets of table data. With this separation of the breakpoint data from the table data, you can share individual parts of a lookup table instead of sharing the entire lookup table.

To store breakpoint and table data:

1. Create a `Simulink.LookupTable` object for each unique set of table data. Create a `Simulink.Breakpoint` object for each unique breakpoint vector, including breakpoint vectors that you do not intend to share.
2. Use the properties of the objects to store the values of the table and breakpoint data.
3. Configure the `Simulink.LookupTable` objects to refer to the `Simulink.Breakpoint` objects for breakpoint data. In the `Simulink.LookupTable` objects, set **Specification** to `Reference`. Specify the names of the `Simulink.Breakpoint` objects.
4. In the Interpolation Using Prelookup blocks, set **Specification** to `Lookup table object`. Set **Name** to the name of a `Simulink.LookupTable` object.

   In the Prelookup blocks, set **Specification** to `Breakpoint object`. Set **Name** to the name of a `Simulink.Breakpoint` object.

The example model `fxpdemo_lookup_shared_param` contains two Prelookup and two Interpolation Using Prelookup blocks. Configure the blocks so that each combination of a Prelookup and an Interpolation Using Prelookup block represents a unique lookup table. Share the breakpoint vector between the two lookup tables. In this case, each lookup table has unique table data but shared breakpoint data.

1. Open the example model.
2. In the Prelookup block dialog box, set **Specification** to `Breakpoint object`. Set **Name** to `sharedBkpts`.
3. Click the button ⋮ next to the value of the **Name** parameter. Select **Create Variable**.
4. In the **Create New Data** dialog box, set **Value** to `Simulink.Breakpoint` and click **Create**.

   A `Simulink.Breakpoint` object appears in the base workspace.
5. In the property dialog box for `sharedBkpts`, specify **Value** as a vector such as `[1 2 3 4 5 6 7 8 9 10]`. Click **OK**.
6. In the Prelookup block dialog box, click **OK**.
7. In the Prelookup1 block dialog box, set **Specification** to `Breakpoint object`. Set **Name** to `sharedBkpts`.
8. In the Interpolation Using Prelookup block dialog box, set **Specification** to `Lookup table object`. Set **Name** to `dataForFirstTable`.
9. Click the button ⋮ next to the value of the **Name** parameter. Select **Create Variable**.
10. In the **Create New Data** dialog box, set **Value** to `Simulink.LookupTable` and click **Create**.

    A `Simulink.LookupTable` object appears in the base workspace.
11. In the property dialog box for `dataForFirstTable`, specify **Value** as a vector, such as `[10 9 8 7 6 5 4 3 2 1]`.

12. Set **Specification** to `Reference`.
13. In the table under **Specification**, set **Name** to `sharedBkpts` and click **OK**.
14. In the Interpolation Using Prelookup block dialog box, click **OK**.
15. Configure the Interpolation Using Prelookup1 block to use a `Simulink.LookupTable` object named `dataForSecondTable`. In the object property dialog box, specify **Value** as a vector, such as `[0 0.5 1 1.5 2 2.5 3 3.5 4 4.5]`. Configure the object to refer to `sharedBkpts` for the breakpoint data.

The model now represents two unique lookup tables:

- A combination of `sharedBkpts` and `dataForFirstTable`.
- A combination of `sharedBkpts` and `dataForSecondTable`.

These lookup tables share the same breakpoint data through `sharedBkpts`.

See Also

Related Examples

- “Create, Edit, and Manage Workspace Variables” on page 67-106
- “Data Objects” on page 67-58
- “Organize Related Block Parameter Definitions in Structures” on page 37-19
- “Set Block Parameter Values” on page 37-2

Parameter Interfaces for Reusable Components

You can use subsystems, referenced models, and custom library blocks as reusable components in other models. For guidelines to help you decide how to componentize a system, see “Choose Among Types of Model Components” on page 22-4.

Typically, a reusable algorithm requires that numeric block parameters, such as the **Gain** parameter of a Gain block, either:

- Use the same value in all instances of the component.
- Use a different value in each instance of the component. Each value is instance specific.

By default, if you use a literal number or expression to set the value of a block parameter, the parameter uses the same value in all instances of the component. If you set multiple block parameter values by using a MATLAB variable, `Simulink.Parameter` object, or other parameter object in a workspace or data dictionary, these parameters also use the same value in all instances of the component.

Referenced Models

If you use model referencing to create a reusable component, to set parameter values that are specific to each instance, configure model arguments for the referenced model. When you instantiate the model by adding a Model block to a different model, you set the values of the arguments in the Model block. When you add another Model block to the same parent model or to a different model, you can set different values for the same arguments. Optionally, if you create more than two instances, you can set the same value for some of the instances and different values for the other instances.

If a model has many model arguments, consider packaging the arguments into a single structure. Instead of configuring many arguments, configure the structure as a single argument. Without changing the mathematical functionality of the component, this technique helps you to reduce the number of model argument values that you must set in each instance of the component.

For more information about model arguments, see “Parameterize Instances of a Reusable Referenced Model” on page 8-64.

Subsystems

If you use subsystems or custom libraries to create reusable components, to set parameter values that are specific to each instance, use masks, mask parameters, and parameter promotion. When you instantiate the component in a model, you set the values of the mask parameters in the Subsystem block. When you instantiate the component again in the same model or a different model, you can set different values for the same mask parameters. Optionally, if you create more than two instances, you can set the same value for some of the instances and different values for the other instances.

If the subsystem has many mask parameters, consider packaging the parameters into a single structure. Instead of configuring many mask parameters, configure the structure as a single parameter. Without changing the mathematical functionality of the component, this technique helps you to reduce the number of mask parameter values that you must set in each instance of the component.

For more information about subsystems, see Subsystem. For more information about custom block libraries, see “Design and Create a Custom Block” on page 40-12. For more information about

masks, see “Masking Fundamentals” on page 39-2. For more information about structures, see “Organize Related Block Parameter Definitions in Structures” on page 37-19.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9
- “Set Block Parameter Values” on page 37-2

Organize Related Block Parameter Definitions in Structures

When you use numeric MATLAB variables to set block parameter values in a model, large models can accumulate many variables, increasing the effort of maintenance and causing the variable names to grow in length.

Instead, you can organize these parameter values into structures. Each structure is a single variable and each field of the structure stores a numeric parameter value. You can assign meaningful names to the structures, substructures, and fields to indicate the purpose of each value.

Use structures to:

- Reduce the number of workspace variables that you must maintain.
- Avoid name conflicts between workspace variables.

  You cannot create two variables that have the same name in the same scope, such as in the base workspace. When you create structures, you must provide each field a name, but multiple structures can each contain a field that uses the same name. Therefore, you can use each structure and substructure as a namespace that prevents the field names from conflicting with each other and with other variable names in the same scope.
- Logically group sets of block parameter values. For example, use nested structures to clearly identify the parameter values that each subsystem or referenced model uses.

If you use mask parameters or model arguments to pass parameter values to the components of a system, you can use structures to reduce the number of individual mask parameters or model arguments that you must maintain. Instead of passing multiple variables, you can pass a single structure variable.

For basic information about creating and manipulating MATLAB structures, see Structures. For basic information about setting block parameter values in a model, see “Set Block Parameter Values” on page 37-2.

To use structures to initialize bus signals, see “Specify Initial Conditions for Bus Signals” on page 76-57.

Create and Use Parameter Structure

This example shows how to create and use a parameter structure in a model.

The example model `f14` uses multiple variables from the base workspace to set block parameter values. For example, when you open the model, it creates the variables `Zw`, `Mw`, and `Mq` in the base workspace. To organize these variables into a single structure variable:

1. At the command prompt, open the example model.

   ```
   f14
   ```

2. At the command prompt, create the parameter structure `myGains`. Set the field values by using the values of the target variables.

   ```
   myGains.Zw = Zw;
   myGains.Mw = Mw;
   myGains.Mq = Mq;
   ```

3. In the Model Explorer, on the **Model Hierarchy** pane, click **Base Workspace**. In the **Contents** pane, right-click the variable `Mq` and select **Find Where Used**.
4. In the **Select a system** dialog box, click the node **f14** and click **OK**. Click **OK** when asked about updating the diagram.
5. In the **Contents** pane, right-click the row corresponding to the block labeled Gain1 and select **Properties**. The Gain1 block dialog box opens.
6. Change the value of the **Gain** parameter from `Mq` to `myGains.Mq` and click **OK**.
7. In the **Contents** pane, right-click the row corresponding to the Transfer Fcn.1 block and select **Properties**.
8. Change the value of the **Denominator coefficients** parameter from `[1,-Mq]` to `[1,-myGains.Mq]` and click **OK**.
9. In the **Model Hierarchy** pane, click **Base Workspace**. Use **Find Where Used** to locate the blocks that use the variables `Mw` and `Zw`. In the block dialog boxes, replace the references to the variable names according to the table.

Variable Name	Replacement Name
`Mw`	`myGains.Mw`
`Zw`	`myGains.Zw`

10. Clear the old variables.

```
clear Zw Mw Mq
```

Each of the modified block parameters now uses a field of the `myGains` structure. The numeric value of each structure field is equal to the value of the corresponding variable that you cleared.

You can migrate a model to use a single parameter structure instead of multiple workspace variables. For an example, see “Migration to Structure Parameters”.

Store Data Type Information in Field Values

To use a structure or array of structures to organize parameter values that use a data type other than `double`, you can explicitly specify the type when you create the structure. When you create the structure, use typed expressions such as `single(15.23)` to specify the field values.

```
myParams.Gain = single(15.23);
```

If you want to change the field value later, you must remember to explicitly specify the type again. If you do not specify the type, the field value uses the data type `double` instead:

```
myParams.Gain = 15.23;
% The field 'Gain' now uses the data type 'double' instead of 'single'.
```

To preserve the type specification, you can use subscripted assignment to assign a new value to the field:

```
% Assign value of type 'single'.
myParams.Gain = single(15.23);

% Assign new value while retaining type 'single'.
myParams.Gain(:) = 11.79;
```

To match a fixed-point data type, set the field value by using an `fi` object.

Control Field Data Types and Characteristics by Creating Parameter Object

A `Simulink.Parameter` object allows you to separate the value of a block parameter from its data type. If you use a parameter object to store a structure or array of structures, you can create a `Simulink.Bus` object to use as the data type of the entire structure.

You can use the bus object and the parameter object to explicitly control:

- The data type of each field. When you use this technique, you do not have to remember to use typed expressions or subscripted assignment to set the field values.
- The complexity, dimensions, and units of each field.
- The minimum and maximum value of each field if the field represents a tunable parameter value.
- The shape of the entire structure. The shape of the structure is the number, names, and hierarchy of fields.
- The tunability of the structure in the code that you generate from the model.

1 Create a parameter structure `myParams`.

```
myParams = struct(...
    'SubsystemA',struct(...
        'Gain',15.23,...
        'Offset',89,...
        'Init',0.59),...
    'SubsystemB',struct(...
        'Coeffs',[5.32 7.99],...
        'Offset',57,...
        'Init1',1.76,...
        'Init2',2.76)...
);
```

2 Use the function `Simulink.Bus.createObject` to create `Simulink.Bus` objects that represent the structure and substructures.

```
Simulink.Bus.createObject(myParams)
```

Because `myParams` contains two unique substructures, the function creates three `Simulink.Bus` objects: one named `slBus1` to represent the parent structure `myParams`, one named `SubsystemA` for the substructure `SubsystemA`, and one named `SubsystemB` for the substructure `SubsystemB`.

3 Rename the bus object `slBus1` as `myParamsType`.

```
myParamsType = slBus1;
clear slBus1
```

4 Store the structure `myParams` in a `Simulink.Parameter` object.

```
myParams = Simulink.Parameter(myParams);
```

The `Value` property of the parameter object contains the structure.

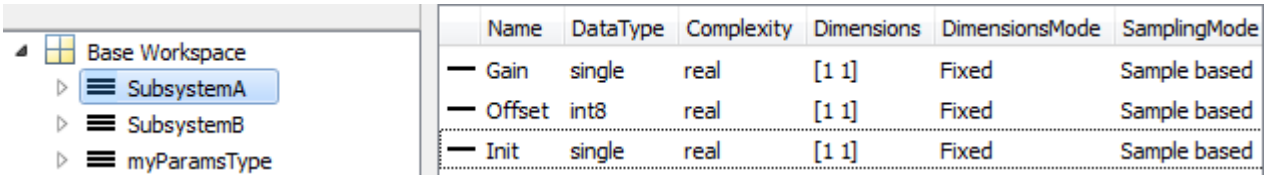
5 Set the data type of the parameter object to the bus object `myParamsType`.

```
myParams.DataType = 'Bus: myParamsType';
```

6 Open the Bus Editor to view the bus objects.

```
buseditor
```

7 In the **Model Hierarchy** pane, click the node SubsystemA. In the **Contents** pane, set the field data types according to the figure.



8 Optionally, set the field data types for the substructure `SubsystemB`.

The parameter object `myParams` stores the parameter structure. The data type of the parameter object is the bus object `myParamsType`. Prior to simulation and code generation, the parameter object casts the field values to the data types that you specified in the bus object.

To use one of the fields to set a block parameter value, specify an expression such as `myParams.SubsystemB.Init1`.

To access the field values at the command prompt, use the `Value` property of the parameter object. Because the bus object controls the field data types, you do not need to use a typed expression to set the field value.

```
myParams.Value.SubsystemA.Gain = 12.79;
```

The bus object strictly controls the field characteristics and the shape of the structure. For example, if you set the value of the two-element field `myParams.SubsystemB.Coeffs` to a three-element array, the model generates an error when you set a block parameter value. To change the dimensions of the field, modify the element `Coeffs` in the bus object `SubsystemB`.

To manipulate bus objects after you create them, see “Create and Specify Simulink.Bus Objects” on page 76-46 and “Save Simulink.Bus Objects” on page 76-47.

Match Field Data Type with Signal Data Type

Suppose that you use the field `myParams.SubsystemA.Gain` to set the value of the **Gain** parameter in a Gain block. If you want the data type of the field to match the data type of the output signal of the block, you cannot rely on context-sensitive data typing (see “Context-Sensitive Data Typing” on page 37-44). Consider using a `Simulink.AliasType` or a `Simulink.NumericType` object to set the data type of the field and the signal. If you do not use a data type object, you must remember to change the data type of the field whenever you change the data type of the signal.

1 At the command prompt, create a `Simulink.AliasType` object that represents the data type `single`.

```
myType = Simulink.AliasType;
myType.BaseType = 'single';
```

2 In the Gain block dialog box, on the **Signal Attributes** tab, set **Output data type** to `myType`.

3 At the command prompt, open the Bus Editor.

```
buseditor
```

4 In the **Model Hierarchy** pane, select the bus object `SubsystemA`. In the **Contents** pane, set the data type of the field `Gain` to `myType`.

Now, both the output signal of the Gain block and the structure field `myParams.SubsystemA.Gain` use the data type that you specify by using the `BaseType` property of `myType`.

For more information about data type objects, see `Simulink.AliasType` and `Simulink.NumericType`.

Manage Structure Variables

To create, modify, and inspect a variable whose value is a structure, you can use the Variable Editor. For more information, see “Modify Structure and Array Variables Interactively” on page 67-107.

Define Parameter Hierarchy by Creating Nested Structures

To further organize block parameter values, create a hierarchy of nested structures.

For example, suppose that you create subsystems named `SubsystemA` and `SubsystemB` in your model. You use variables such as `Offset_SubsystemA` and `Offset_SubsystemB` to set block parameter values in the subsystems.

```
Gain_SubsystemA = 15.23;
Offset_SubsystemA = 89;
Init_SubsystemA = 0.59;

Coeffs_SubsystemB = [5.32 7.99];
Offset_SubsystemB = 57;
Init1_SubsystemB = 1.76;
Init2_SubsystemB = 2.76;
```

Create a parameter structure that contains a substructure for each subsystem. Use the values of the existing variables to set the field values.

```
myParams = struct(...
    'SubsystemA',struct(...
        'Gain',Gain_SubsystemA,...
        'Offset',Offset_SubsystemA,...
        'Init',Init_SubsystemA),...
    'SubsystemB',struct(...
        'Coeffs',Coeffs_SubsystemB,...
        'Offset',Offset_SubsystemB,...
        'Init1',Init1_SubsystemB,...
        'Init2',Init2_SubsystemB)...
);
```

The single structure variable `myParams` contains all of the parameter information for the blocks in the subsystems. Because each substructure acts as a namespace, you can define the `Offset` field more than once.

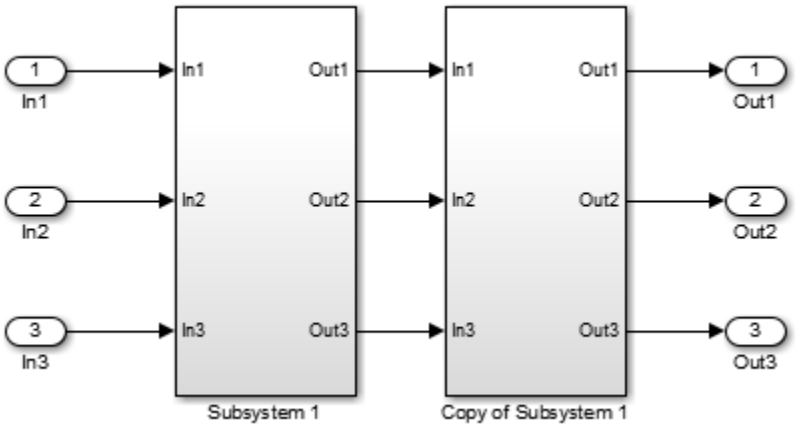
To use the `Offset` field from the substructure `SubsystemB` as the value of a block parameter, specify the parameter value in the block dialog box as the expression `myParams.SubsystemB.Offset`.

Group Multiple Parameter Structures into an Array

To organize parameter structures that have similar characteristics, you can create a single variable whose value is an array of structures. This technique helps you to parameterize a model that contains

multiple instances of an algorithm, such as a library subsystem or a referenced model that uses model arguments.

Suppose that you create two identical subsystems in a model.



Suppose that the blocks in each subsystem require three numeric values to set parameter values. Create an array of two structures to store the values.

```
myParams(1).Gain = 15.23;
myParams(1).Offset = 89;
myParams(1).Init = 0.59;

myParams(2).Gain = 11.93;
myParams(2).Offset = 57;
myParams(2).Init = 2.76;
```

Each structure in the array stores the three parameter values for one of the subsystems.

To set the value of a block parameter in one of the subsystems, specify an expression that references a field of one of the structures in the array. For example, use the expression `myParams(2).Init`.

Organize Parameter Values for Reusable Components and Iterative Algorithms

You can also partition an array of structures in a For Each Subsystem block. This technique helps you to organize workspace variables when a model executes an algorithm repeatedly, for example by iterating the algorithm over a vector signal. For an example, see “Repeat an Algorithm Using a For Each Subsystem” on page 76-81.

If you use model arguments to specify different parameter values across multiple instances of a referenced model, you can use arrays of structures to organize the model argument values. In the referenced model workspace, create a structure variable and configure the model to use the structure as a model argument. Use the fields of the structure to set block parameter values in the model. Then, create an array of structures in the base workspace or a data dictionary to which the parent model or models are linked. In the parent model or models, use each of the structures in the array as the value of the model argument in a Model block. Each structure in the array stores the parameter values for one instance of the referenced model.

The example model `sldemo_mdlref_datamngt` contains three instances (Model blocks) of the masked referenced model `sldemo_mdlref_counter_datamngt`. The base workspace variables

`IC1`, `IC2`, `Param1`, and `Param2` are `Simulink.Parameter` objects whose values are structures. The parent model uses these variables to set the values of mask parameters on the Model blocks. Since `IC1` is structurally identical to `IC2`, and `Param1` to `Param2`, you can combine these four structures into two arrays of structures.

1. Open the example parent model.

   ```
   sldemo_mdlref_datamngt
   ```

   The model creates the four `Simulink.Parameter` objects in the base workspace.
2. Open the example referenced model.

   ```
   sldemo_mdlref_counter_datamngt
   ```

   The model workspace defines two model arguments, `CounterICs` and `CounterParams`, whose values are structures. The blocks in the model use the fields of these structures to set parameter values.
3. In the model `sldemo_mdlref_datamngt`, open the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). In the Model Data Editor, inspect the **Parameters** tab.
4. In the model, click one of the Model blocks.

   The Model Data Editor highlights rows that correspond to two mask parameters on the selected Model block. The block uses the mask parameters to set the values of the two model arguments defined by the referenced model, `sldemo_mdlref_counter_datamngt`. Each Model block uses a different combination of the four parameter objects from the base workspace to set the argument values.
5. In the Model Data Editor **Value** column, click one of the cells to begin editing the value of the corresponding mask parameter (for example, `IC1`). Next to the parameter value, click the action button ⋮ and select **Open**. The property dialog box for the parameter object opens.
6. In the property dialog box, next to the **Value** box, click the action button and select **Open Variable Editor**.

   The Variable Editor shows that the parameter object stores a structure. The structures in `Param2` and `IC2` have the same fields as the structures in `Param1` and `IC1` but different field values.
7. At the command prompt, combine the four parameter objects into two parameter objects whose values are arrays of structures.

   ```
   % Create a new parameter object by copying Param1.
   Param = Param1.copy;

   % Use the structure in Param2 as the second structure in the new object.
   Param.Value(2) = Param2.Value;
   % The value of Param is now an array of two structures.

   % Delete the old objects Param1 and Param2.
   clear Param1 Param2

   % Create a new parameter object by copying IC1.
   % Use the structure in IC2 as the second structure in the new object.
   IC = IC1.copy;
   IC.Value(2) = IC2.Value;
   clear IC1 IC2
   ```
8. In the parent model, in the Model Data Editor, use the **Value** column to replace the values of the mask parameters according to the table

Previous Value	New Value
Param1	Param(1)
IC1	IC(1)
Param2	Param(2)
IC2	IC(2)

Each Model block sets the value of the model argument `CounterICs` by using one of the structures in the array `IC`. Similarly, each block sets the value of `CounterParams` by using one of the structures in `Param`.

Enforce Uniformity in an Array of Structures

All of the structures in an array of structures must have the same hierarchy of fields. Each field in the hierarchy must have the same characteristics throughout the array. You can use a parameter object and a bus object to enforce this uniformity among the structures.

To use a parameter object to represent an array of parameter structures, set the value of the object to the array of structures:

```
% Create array of structures.
myParams(1).Gain = 15.23;
myParams(1).Offset = 89;
myParams(1).Init = 0.59;
myParams(2).Gain = 11.93;
myParams(2).Offset = 57;
myParams(2).Init = 2.76;

% Create bus object.
Simulink.Bus.createObject(myParams);
myParamsType = slBus1;
clear slBus1

% Create parameter object and set data type.
myParams = Simulink.Parameter(myParams);
myParams.DataType = 'Bus: myParamsType';
```

To use one of the fields to set a block parameter value, specify an expression such as `myParams(2).Offset`.

To access the field values at the command prompt, use the `Value` property of the parameter object.

```
myParams.Value(2).Offset = 129;
```

Create a Structure of Constant-Valued Signals

You can use a structure in a Constant block to create a single bus signal that transmits multiple numeric constants. For more information, see Constant. For information about bus signals, see “Virtual Bus” on page 76-2.

Considerations Before Migrating to Parameter Structures

- Before you migrate a model to use parameter structures, discover all of the blocks in the target model and in other models that use the variables that you intend to replace.

For example, suppose two blocks in a model use the workspace variable `myVar`. If you create a structure `myParams` with a field `myVar`, and set the parameter value in only one of the blocks to `myParams.myVar`, the other block continues to use the variable `myVar`. If you delete `myVar`, the model generates an error because the remaining block requires the deleted variable.

To discover all of the blocks that use a variable:

1. Open all models that might use the variable. If the models are in a model reference hierarchy, you can open only the top model.
2. In the Model Data Editor or in the Model Explorer **Contents** pane, right-click the variable and select **Find Where Used**. The Model Explorer displays all of the blocks that use the variable.

You can discover variable usage only in models that are open. Before you migrate to parameter structures, open all models that might use the target variables. For more information about determining variable usage in a model, see “Finding Blocks That Use a Specific Variable” on page 67-111.

Alternatively, you can refrain from deleting `myVar`. However, if you change the value of the `myParams.myVar` structure field, you must remember to change the value of `myVar` to match.

- You can combine multiple separate variables or parameter objects (such as `Simulink.Parameter`) into a structure that you store in a single variable or parameter object (to combine parameter objects, see “Combine Existing Parameter Objects Into a Structure” on page 37-27). However, the resulting variable or object acts as a single entity. As a result, you cannot apply different code generation settings, such as storage classes, to individual fields in the structure.

Combine Existing Parameter Objects Into a Structure

When you use parameter objects to set block parameter values (for example, so you can apply storage classes), to combine the objects into a single structure:

1. Create a MATLAB structure and store it in a variable. To set the field values, use the parameter values that each existing parameter object stores.
2. Convert the variable to a parameter object. Create and use a `Simulink.Bus` object as the data type of the parameter object (see “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21).
3. Choose a storage class to apply to the resulting parameter object. You can choose only one storage class, which applies to the entire structure.
4. Transfer parameter metadata, such as the `Min` and `Max` properties of the existing parameter objects, to the corresponding properties of the `Simulink.BusElement` objects in the bus object.

For example, suppose you have three individual parameter objects.

```
coeff = Simulink.Parameter(17.5);
coeff.Min = 14.33;
coeff.DataType = 'single';
coeff.StorageClass = 'ExportedGlobal';

init = Simulink.Parameter(0.00938);
init.Min = -0.005;
init.Max = 0.103;
init.DataType = 'single';
```

```
init.StorageClass = 'Model default';

offset = Simulink.Parameter(199);
offset.DataType = 'uint8';
offset.StorageClass = 'ExportedGlobal';
```

1. Create a structure variable.

   ```
   myParams.coeff = coeff.Value;
   myParams.init = init.Value;
   myParams.offset = offset.Value;
   ```

2. Convert the variable to a parameter object.

   ```
   myParams = Simulink.Parameter(myParams);
   ```

3. Create a bus object and use it as the data type of the parameter object.

   ```
   Simulink.Bus.createObject(myParams.Value);
   paramsDT = copy(slBus1);

   myParams.DataType = 'Bus: paramsDT';
   ```

4. Transfer metadata from the old parameter objects to the bus elements in the bus object.

   ```
   % coeff
   paramsDT.Elements(1).Min = coeff.Min;
   paramsDT.Elements(1).DataType = coeff.DataType;

   % init
   paramsDT.Elements(2).Min = init.Min;
   paramsDT.Elements(2).Max = init.Max;
   paramsDT.Elements(2).DataType = init.DataType;

   % offset
   paramsDT.Elements(3).DataType = offset.DataType;
   ```

   To help you write a script that performs this transfer operation, you can use the `properties` function to find the properties that the bus elements and the old parameter objects have in common. To list the structure fields so that you can iterate over them, use the `fieldnames` function.

5. Apply a storage class to the parameter object.

   ```
   myParams.StorageClass = 'ExportedGlobal';
   ```

Now, you can use the fields of `myParams`, instead of the old parameter objects, to set the block parameter values.

Parameter Structures in the Generated Code

You can configure parameter structures to appear in the generated code as structures and arrays of structures. For information about generating code with parameter structures, see “Organize Data into Structures in Generated Code” (Simulink Coder).

Parameter Structure Limitations

- The value of a field that you use to set a block parameter must be numeric or of an enumerated type. The value of a field can be a real or complex scalar, vector, or multidimensional array.
- If the value of any of the fields of a structure is a multidimensional array, you cannot tune any of the field values during simulation.
- All of the structures in an array of structures must have the same hierarchy of fields. Each field in the hierarchy must have the same characteristics throughout the array:

  - Field name
  - Numeric data type, such as `single` or `int32`
  - Complexity
  - Dimensions

  Suppose that you define an array of two structures.

  ```
  paramStructArray = ...
  [struct('sensor1',int16(7),'sensor2',single(9.23)) ...
   struct('sensor1',int32(9),'sensor2',single(11.71))];
  ```

  You cannot use any of the fields in a block parameter because the field `sensor1` uses a different data type in each structure.
- Parameter structures do not support context-sensitive data typing in the generated code. If the parameter structure is tunable in the code, the fields of the structure use the numeric data types that you specify by using either typed expressions or a `Simulink.Bus` object. If you do not use typed expressions or a `Simulink.Bus` object, the fields of the structure use the `double` data type.

Package Shared Breakpoint and Table Data for Lookup Tables

When you share data between lookup table blocks, consider using `Simulink.LookupTable` and `Simulink.Breakpoint` objects instead of structures to store and group the data. This technique improves model readability by clearly identifying the data as parts of a lookup table and explicitly associating breakpoint data with table data. See “Package Shared Breakpoint and Table Data for Lookup Tables” on page 37-14.

Create Parameter Structure According to Structure Type from Existing C Code

You can create a parameter structure that conforms to a `struct` type definition that your existing C code defines. Use this technique to:

- Replace existing C code with a Simulink model.
- Integrate existing C code for simulation in Simulink (for example, by using the Legacy Code Tool). For an example, see “Integrate C Function Whose Arguments Are Pointers to Structures”.
- Generate C code (Simulink Coder) that you can compile with existing C code into a single application. For an example, see “Exchange Structured and Enumerated Data Between Generated and External Code” (Embedded Coder).

In MATLAB, store the parameter structure in a parameter object and use a bus object as the data type (see “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21). To create the bus object according to your C-code `struct` type, use the `Simulink.importExternalCTypes` function.

See Also

Related Examples

- “Detailed Workflow for Managing Data with Model Reference”
- “Set Block Parameter Values” on page 37-2
- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- Structures
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 37-56

Tune and Experiment with Block Parameter Values

As you construct a model you can experiment with block parameters, such as the coefficients of a Transfer Fcn block, to help you decide which blocks to use. You can simulate the model with different parameter values, and capture and observe the simulation output.

You can change the values of most numeric block parameters during a simulation. To observe the effects, you can visualize the simulation output in real time. This technique allows you to quickly test parameter values while you develop an algorithm. You can visually:

- Tune and optimize control parameters.
- Calibrate model parameters.
- Test control robustness under different conditions.

When you begin a simulation, Simulink first updates the model diagram. This operation can take time for larger models. To test parameter values without repeatedly updating the model diagram, you can tune the parameter values during a single simulation run.

Alternatively, to avoid updating the model diagram, use Fast Restart. For more information about Fast Restart, see “Get Started with Fast Restart” on page 81-5.

If you cannot visually analyze the simulation output in real time, or if you must run many simulations, consider using a programmatic approach to sweeping parameter values. You can capture the simulation output data and perform analysis later. For more information, see “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

For basic information about accessing and setting block parameter values, see “Set Block Parameter Values” on page 37-2.

Iteratively Adjust Block Parameter Value Between Simulation Runs

This example shows how to prototype a model by changing block parameter values between simulation runs. You can experiment with parameter values and observe simulation results to help you decide which blocks to use and how to build your model.

The example model `sldemo_fuelsys` represents the fueling system of a gasoline engine. A subsystem in the model, `feedforward_fuel_rate`, calculates the fuel demand of the engine by using the constant number `14.6`, which represents the ideal (stoichiometric) ratio of air to fuel that the engine consumes. Two blocks in the subsystem use the number to set the values of parameters.

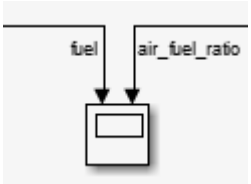
Suppose that you want to change the design value of the ideal air-to-fuel ratio from `14.6` to `17.5` to observe the effect on the fuel demand. To store the design value in the model, you can modify the value in the block dialog boxes. Alternatively, you can store the value in a variable with a meaningful name, which allows you to reuse the value in the two blocks.

To observe the change in simulation outputs by changing the value in the block dialog boxes:

1. Open the example model.

   ```
   sldemo_fuelsys
   ```

2. Set the model simulation time from `2000` to `50` for a faster simulation.
3. In the model, open the Scope block dialog box.



4. Simulate the model. Resize the window in the Scope dialog box to see all of the simulation results.

   The scope display shows that throughout the simulation, the `fuel` signal oscillates between approximately `0.9` and `1.6`. The `air_fuel_ratio` signal quickly climbs to `15` without overshoot.

5. In the model, open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, inspect the **Parameters** tab.

6. In the model or at the command prompt, navigate to the target subsystem.

```
open_system(...
    'sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate')
```

7. In the Model Data Editor, use the **Value** column to change the **Constant value** (`Value`) parameter of the Constant block labeled `rich` from `1/(14.6*0.8)` to `1/(17.5*0.8)`.

8. Similarly, change the **Constant value** parameter of the block labeled `normal` from `1/14.6` to `1/17.5`.

9. Simulate the model.

   The scope display shows that the signals now respond differently.

To replace the literal values in the block dialog boxes with a numeric variable:

1. Use the Model Data Editor to set the value of the `normal` Constant block to `1/mixture`.

2. Set the value of the `rich` block to `1/(mixture*0.8)`.

3. While editing the `rich` value, next to `1/(mixture*0.8)`, click the action button ⋮ and select **Create**.

4. In the **Create New Data** dialog box, set **Value** to `17.5` and click **Create**.

   The numeric variable `mixture` appears in the base workspace with value `17.5`. Between simulation runs, you can change the value of `mixture` in the base workspace instead of changing the parameter values in the block dialog boxes.

Tune Block Parameter Value During Simulation

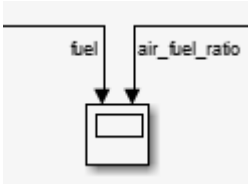
This example shows how to observe the effect of changing a block parameter value during a simulation. This technique allows you to avoid updating the model diagram between simulation runs and to interactively test and debug your model.

The example model `sldemo_fuelsys` contains a Constant block, `Throttle Command`, that represents the throttle command. To observe the effect of increasing the magnitude of the command during simulation:

1. Open the example model.

```
sldemo_fuelsys
```

2. In the model, open the Scope block dialog box.



3. Begin a simulation.

   The model is configured to simulate 2000 seconds. During the simulation, the values of the `fuel` and `air_fuel_ratio` signals appear on the scope graph in real time.
4. In the model, when the status bar indicates approximately `1000` (1000 seconds), click the Pause button to pause the simulation.
5. In the scope display, the **fuel** graph plots the simulation output prior to the pause time.
6. In the model, on the **Modeling** tab, click **Model Data Editor**.
7. In the Model Data Editor, select the **Parameters** tab.
8. In the model, select the `Throttle Command` block.
9. In the Model Data Editor, select the `rep_seq_y` row. Make sure that you do not select the `rep_seq_t` row as well.
10. For the `rep_seq_y` row, change the value in the **Value** column from `[10 20 10]` to `[10 30 10]`.
11. Click the Step Forward button to advance the simulation step by step. Click the button about 15 times or until you see a change in the **fuel** graph in the scope display.

    The plot of the signal `fuel` indicates a sharp increase in fuel demand that corresponds to the increased throttle command.
12. In the model, resume the simulation by clicking the Continue button.

    The scope display shows the significant periodic increase in fuel demand, and the periodic reduction in the air-to-fuel ratio, throughout the rest of the simulation.

During the simulation, you must update the model diagram after you change the value of a workspace variable. For more information about updating the model diagram, see “Update Diagram and Run Simulation” on page 1-38.

Prepare for Parameter Tuning and Experimentation

- Use workspace variables to set block parameter values.

  To access the value of a block parameter, such as the **Constant value** parameter of a Constant block, you must navigate to the block in the model and open the block dialog box, search for the block by using the Model Explorer, or use the function `set_param` at the command prompt.

  Alternatively, if you set the block parameter value by creating a workspace variable, you can change the value of the variable by using the command prompt, the MATLAB Workspace browser, or the Model Explorer. You can also create a variable to set the same value for multiple block parameters. When you change the variable value, all of the target block parameters use the new value. For more information about accessing and setting block parameter values, see “Set Block Parameter Values” on page 37-2.
- Visualize simulation output.

To observe simulation output in real time while you tune block parameter values, you can use blocks in a model such as the Scope block. You can also capture simulation output at the end of a simulation run, and view the data in the Simulation Data Inspector. For more information, see “Decide How to Visualize Simulation Data” on page 30-2.

- Specify value ranges for block parameters that you expect to tune during simulation.

  If you expect another person to use your model and tune the parameter, you can control the allowed tuning values by specifying a range. Also, it is a best practice to specify value ranges for all fixed-point block parameters that you expect to tune. To specify block parameter value ranges, see “Specify Minimum and Maximum Values for Block Parameters” on page 37-52.
- Control simulation duration and pace.

  A simulation run can execute so quickly that you cannot tune block parameter values. Also, if you want to change a parameter value at a specific simulation time, you must learn to control the simulation pace. You can configure the simulation to run for a specific duration or to run forever, and pause and advance the simulation when you want to. For more information, see “Simulate a Model Interactively” on page 25-2 and “Run Simulations Programmatically” on page 26-2.

Interactively Tune Using Dashboard Blocks

You can tune block parameter values by adding blocks from the Dashboard library to your model. Dashboard blocks allow you to adjust the parameter values of other blocks, and to observe simulation output in real time, by interacting with knobs, switches, and readouts that mimic the appearance of industrial controls. You can interact with the Dashboard blocks without having to locate the target block parameters in the model. For more information, see “Tune and Visualize Your Model with Dashboard Blocks” on page 29-164.

Which Block Parameters Are Tunable During Simulation?

Nontunable block parameters are parameters whose values you cannot change during simulation. For example, you cannot tune the **Sample time** block parameter. If a parameter is nontunable, you cannot change its value during simulation by changing the value in the block dialog box or by changing the value of a workspace variable.

Nontunable block parameters include:

- Sample times.
- Parameters that control the appearance or structure of a block such as the number of inputs of a Sum block.
- Priority, which allows you to control block execution order.
- Parameters that control the block algorithm, such as the **Integrator method** parameter of a Discrete-Time Integrator block.

To determine whether a block parameter is tunable during simulation, use one of these techniques:

- Begin a simulation and open the block dialog box. If the value of the target block parameter is gray during simulation, you cannot tune the parameter.
- At the command prompt, determine whether the flags `read-write` and `read-only-if-compiled` describe the parameter.

1. Select the block in the model.
2. At the command prompt, use the function `get_param` to return information about the block dialog box parameters. The function returns a structure that has a field for each parameter in the block dialog box.

   ```
   paramInfo = get_param(gcb,'DialogParameters');
   ```

   Alternatively, rather than locating and selecting the block in the model, you can replace `gcb` with the block path, such as `'myModel/mySubsystem/myBlock'`.
3. View the information about the target block parameter. For example, to view the information about the **Sample time** parameter of a block, view the value of the field `SampleTime`, which is also a structure.

   ```
   paramInfo.SampleTime

   ans =

           Prompt: 'Sample time:'
             Type: 'string'
             Enum: {}
       Attributes: {'read-write'  'read-only-if-compiled'  'dont-eval'}
   ```
4. Inspect the structure's `Attributes` field, whose value is a cell array of character vectors. If the flag `read-write` appears in the cell array, you can modify the parameter value. However, if the flag `read-only-if-compiled` also appears in the cell array, you cannot modify the parameter value during simulation.

If you use masks to create custom interfaces for blocks and subsystems, you can control the tunability of individual mask parameters. If you use model arguments to parameterize referenced models, you can tune the value of each model argument in each Model block.

Why Did the Simulation Output Stay the Same?

If the output of your simulation does not change after you change a parameter value, use these troubleshooting techniques:

- Locate the definition of a workspace variable.

  If you use a workspace variable to set block parameter values, determine where the variable definition resides. For example, if you define a variable `myVar` in a model workspace and use it to set a block parameter value in the model, you cannot change the parameter value by changing the value of a variable named `myVar` in the base workspace. You must access the variable definition in the model workspace.

  To locate the definition of a variable, while editing the value of a block parameter that uses the variable, click the nearby action button ⋮ and select **Explore**. A dialog box opens, such as the Model Explorer, which displays the definition of the variable in the appropriate workspace. For more information about how models use variables, see "Symbol Resolution" on page 67-127.
- Specify value ranges for fixed-point parameters that you want to tune during simulation.

  If the block parameter you want to tune uses a fixed-point data type with best-precision scaling, specify a minimum and maximum value for the parameter so that Simulink can calculate and apply an appropriate scaling. If you do not specify a value range, Simulink might apply a scaling that

excludes the tuning values that you want to use. To specify value ranges, see “Specify Minimum and Maximum Values for Block Parameters” on page 37-52.

- Update the model diagram during a simulation run. If you use a workspace variable to set the value of one or more block parameters, after you change the value of the variable during a simulation, you must update the model diagram.

  To learn how to update a model diagram, see “Update Diagram and Run Simulation” on page 1-38.

Tunability Considerations and Limitations for Other Modeling Goals

Referenced Models

When you use Model blocks, these parameter tunability limitations apply:

- If you set the simulation mode of a Model block to an accelerated mode or if you simulate the parent model in an accelerated mode, you cannot tune block parameters in the referenced model during simulation. However, if the referenced model uses variables in the base workspace or a data dictionary to set parameter values, you can tune the values of the variables.
- Suppose you use a MATLAB variable or `Simulink.Parameter` object in a model workspace to set the value of a block parameter in a model. If you use a Model block to refer to this model:
  - And you set the simulation mode of the Model block to an accelerated mode or simulate the parent model in an accelerated mode, you cannot change the value of the variable or object during the simulation.
  - When you simulate the parent model in an accelerated mode, changing the value of the variable or object between simulation runs causes Simulink to regenerate code.
  - And you use additional Model blocks to refer to the model multiple times in the parent model, you can choose a different simulation mode for each Model block. If at least one block uses normal simulation mode and any other block uses a different simulation mode, you cannot change the value of the variable or object during simulation. Also, when you simulate the parent model with fast restart on, you cannot change the value of the variable or object between fast-restart simulation runs.

  As a workaround, move the variable or object to the base workspace or a data dictionary.

Accelerator and SIL/PIL Simulations

These tunability limitations apply to accelerator, rapid accelerator, SIL, and PIL simulations:

- Suppose you use a MATLAB variable or `Simulink.Parameter` object in a model workspace to set the value of a block parameter in a model. If you use the `sim` function to simulate the model in rapid accelerator mode and set the `RapidAcceleratorUpToDateCheck` pair argument to `'off'`, you cannot use the `RapidAcceleratorParameterSets` pair argument to specify different values for the variable or object. The structure returned by `Simulink.BlockDiagram.buildRapidAcceleratorTarget` does not contain information about the variable or object.
- If a block parameter value references workspace variables, you cannot change the block parameter value during rapid accelerator simulation, such as by using the function `set_param`. Instead, you can tune the values of the referenced variables.

  Alternatively, use parameter sets to tune runtime parameters in between rapid accelerator simulations. For more information, see “Tuning Runtime Parameters” on page 35-8.

For more information about parameter tunability during accelerated simulations, see “Tuning Runtime Parameters” on page 35-8 and “sim in parfor with Rapid Accelerator Mode” on page 26-12. For more information about parameter tunability during SIL and PIL simulations, see “Tunable Parameters and SIL/PIL” (Embedded Coder).

Fast Restart

For more information about parameter tunability when you use fast restart, see “Get Started with Fast Restart” on page 81-5 .

Code Generation and Simulation of External Programs

Parameters that are tunable during simulation can appear as nontunable inlined parameters in the generated code. If you simulate an external program by using SIL, PIL, or External mode simulation, parameter tunability during the simulation and between simulation runs can depend on your code generation settings.

To control parameter tunability in the generated code, you can adjust the code generation settings for a model by using the configuration parameter **Default parameter behavior**. You can also adjust settings for individual MATLAB variables, `Simulink.Parameter` objects, and other parameter objects. For more information, see “Preserve Variables in Generated Code” (Simulink Coder).

Stateflow Charts

To debug a Stateflow chart by changing data during simulation, see “Debugging Stateflow Charts” (Stateflow).

See Also

`set_param`

Related Examples

- “Specify Minimum and Maximum Values for Block Parameters” on page 37-52
- “Parameter Tuning in Rapid Accelerator Mode” on page 35-7
- “Create, Edit, and Manage Workspace Variables” on page 67-106
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 37-56

Optimize, Estimate, and Sweep Block Parameter Values

When you sweep one or more parameters, you change their values between simulation runs, and compare and analyze the output signal data from each run. Use parameter sweeping to tune control parameters, estimate unknown model parameters, and test the robustness of a control algorithm by taking into consideration uncertainty in the real-world system.

You can sweep block parameter values or the values of workspace variables that you use to set the parameter values. Use the **Parameters** tab on the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**), the command prompt, or scripts to change parameter values between simulation runs.

If you want to repeatedly change the value of a block parameter, consider creating a variable in a workspace. You can use the Model Explorer or programmatic commands to change the value of the variable instead of locating or identifying the block in the model. Also, several features and products that facilitate parameter optimization, estimation, and sweeping require that you set block parameter values by creating workspace variables.

To learn how to manipulate parameter values during the iterative process of creating a model, see “Tune and Experiment with Block Parameter Values” on page 37-31.

For basic information about accessing and setting block parameter values as you design a model, see “Set Block Parameter Values” on page 37-2. For basic information about programmatically simulating a model, such as by using a script, see “Run Simulations Programmatically” on page 26-2.

Sweep Parameter Value and Inspect Simulation Results

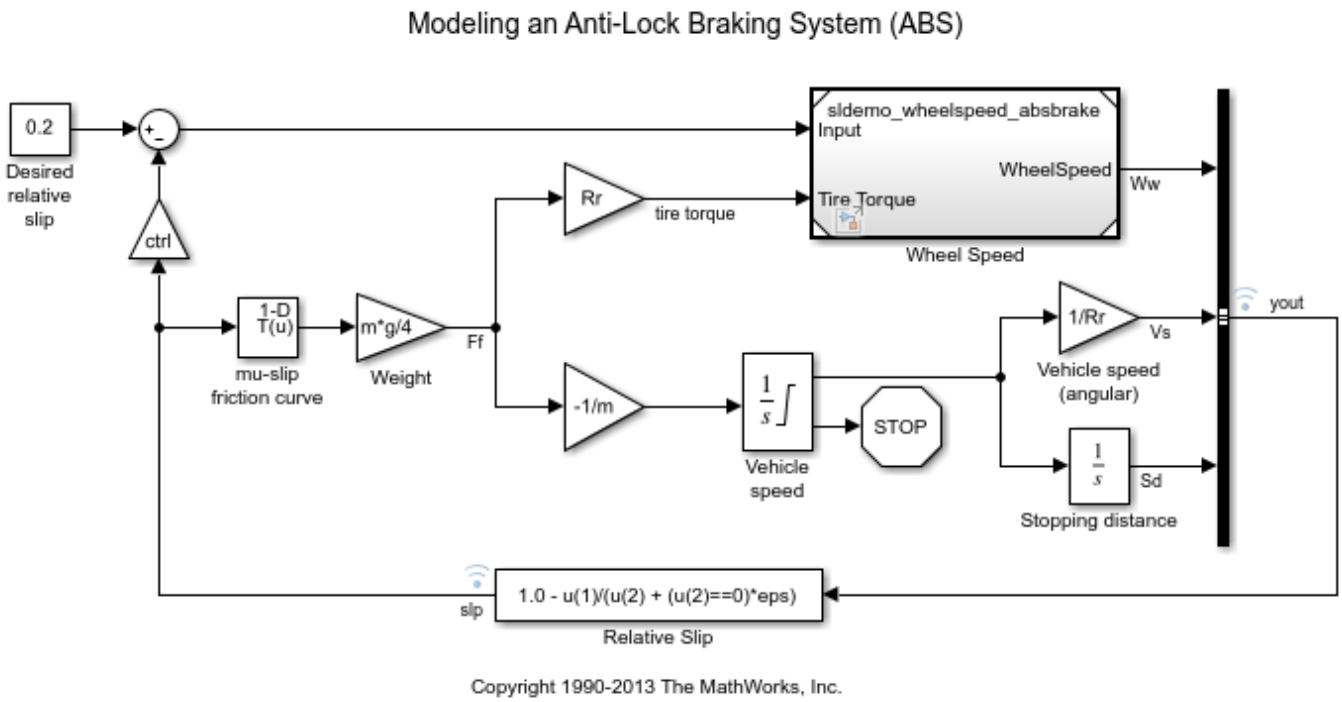
This example shows how to change a block parameter value between multiple programmatic simulation runs. Use this technique to determine an optimal parameter value by comparing the output signal data of each run.

The example model `sldemo_absbrake` uses a Constant block to specify a slip setpoint for an anti-lock braking system. Simulate the model with two different slip setpoint values, 0.24 and 0.25, and compare the output wheel speed of each simulation run.

To store the setpoint value, create a variable in the base workspace. This technique enables you to assign a meaningful name to the value.

Open the example model.

```
open_system('sldemo_absbrake');
```



On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, select the **Signals** tab.

Set the **Change view** drop-down list to `Instrumentation`. The **Log Data** column shows that the signals `yout` (which is a virtual bus) and `slp` are configured for logging. When you simulate the model, you can collect and later inspect the values of these signals by using the Simulation Data Inspector.

In the Model Data Editor, select the **Parameters** tab. Set **Change view** to `Design`.

In the model, select the Constant block labeled `Desired relative slip`. The Model Data Editor highlights the row that corresponds to the **Constant value** parameter of the block.

Use the **Value** column to set the parameter value to `relSlip`.

While editing the value, next to `relSlip`, click the action button (with three vertical dots) and select **Create**.

In the Create New Data dialog box, set **Value** to `0.2` and click **Create**. A variable, whose value is `0.2`, appears in the base workspace. The model now acquires the relative slip setpoint from this variable.

Alternatively, you can use these commands at the command prompt to create the variable and configure the block:

```
relSlip = 0.2;
set_param('sldemo_absbrake/Desired relative slip','Value','relSlip')
```

At the command prompt, create an array to store the two experimentation values for the relative slip setpoint, `0.24` and `0.25`.

```
relSlip_vals = [0.24 0.25];
```

Create a `Simulink.SimulationInput` object for each simulation that you want to run (in this case, two). Store the objects in a single array variable, `simIn`. Use the `setVariable` method of each object to identify each of the two experimentation values.

```
for i = 1:length(relSlip_vals)
    simIn(i) = Simulink.SimulationInput('sldemo_absbrake');
    simIn(i) = setVariable(simIn(i),'relSlip',relSlip_vals(i));
end
```

Use the `sim` function to simulate the model. Optionally, store the output in a variable named `simOutputs`.

```
simOutputs = sim(simIn);
```

```
[26-Aug-2020 09:08:32] Running simulations...
[26-Aug-2020 09:08:34] Completed 1 of 2 simulation runs
[26-Aug-2020 09:08:35] Completed 2 of 2 simulation runs
```

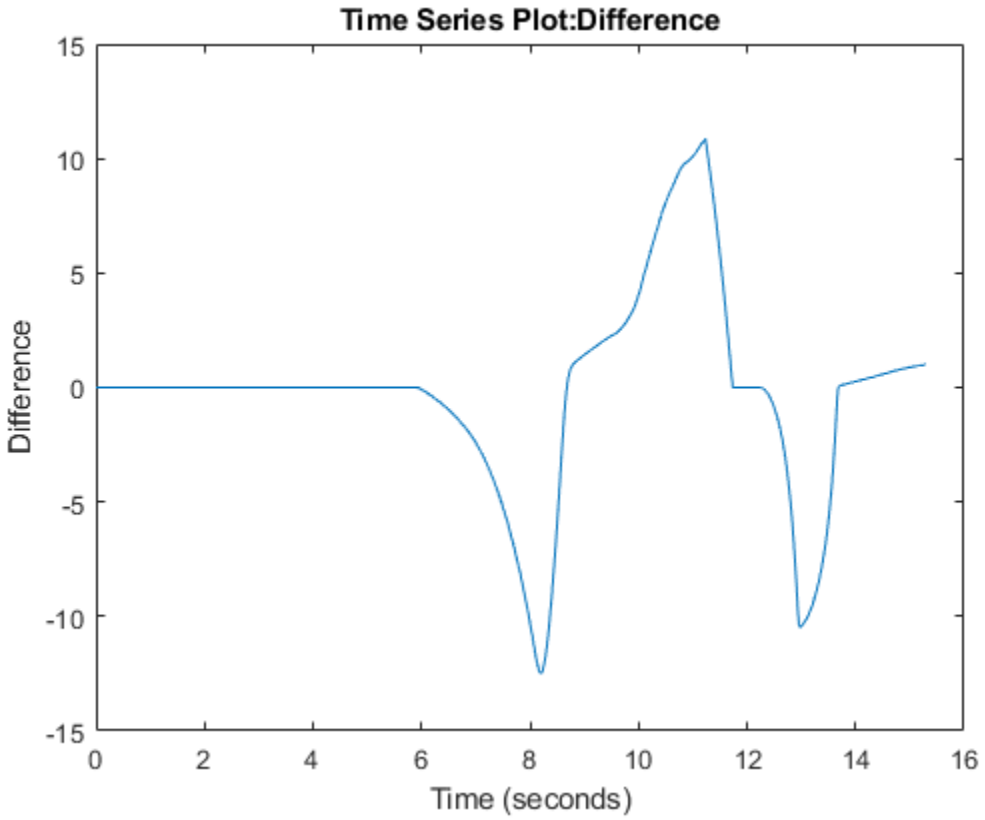
The model streams the logged signals, `yout` and `slp`, to the Simulation Data Inspector. You can view the signal data in the Simulation Data Inspector.

Compare the output data of the two latest simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs();
runResult = Simulink.sdi.compareRuns(runIDs(end-1), runIDs(end));
```

Plot the difference between the values of the `Ww` signal (which is an element of the virtual bus signal `yout`) by specifying the result index `1`.

```
signalResult = getResultByIndex(runResult,1);
plot(signalResult.Diff);
```



Store Sweep Values in Simulink.SimulationInput Objects

When you write a script to run many simulations, create an array of `Simulink.SimulationInput` objects (one object for each simulation that you want to run). Use the `setVariable` and `setBlockParameter` methods of each object to identify the parameter values to use for the corresponding simulation run. With this technique, you avoid having to use the `set_param` function to modify block parameter values and assignment commands to modify workspace variable values between simulation runs.

For more information about using `Simulink.SimulationInput` objects to run multiple simulations, see `sim`.

Sweep Nonscalars, Structures, and Parameter Objects

If you use nonscalar variables, structure variables, or `Simulink.Parameter` objects to set block parameter values, use the `setVariable` method of each `Simulink.SimulationInput` object. Refer to the examples in the table.

Scenario	Example
MATLAB variable, `myArray`, whose value is an array. You want to set the third element in the array (assuming one-based indexing).	`setVariable(simIn,'myArray(3)',15.23)`

Scenario	Example
MATLAB variable, `myStruct`, that has a field named `field1`.	`setVariable(simIn,'myStruct.field1',15.23)`
Parameter object, `myParam`, whose `Value` property is a scalar.	`setVariable(simIn,'myParam.Value',15.23)`
Parameter object, `myArrayParam`, whose `Value` property is an array. You want to set the third element in the array.	`setVariable(simIn,'myArrayParam.Value(3)',15.23)`
Parameter object, `myStructParam`, whose `Value` property is a structure. The structure has a field named `field1`.	`setVariable(simIn,'myStructParam.Value.field1',15.23)`

Sweep Value of Variable in Model Workspace

If you use the model workspace to store variables, when you use the `setVariable` method of a `Simulink.SimulationInput` object to modify the variable value, use the `Workspace` pair argument to identify the containing model:

```
setVariable(simIn,'myVar',15.23,'Workspace','myModel')
```

Capture and Visualize Simulation Results

Each simulation run during a parameter sweep produces outputs, such as signal values from Outport blocks and from logged signals.

You can capture these outputs in variables and objects for later analysis. For more information, see “Export Simulation Data” on page 72-2.

To visualize simulation output data so you can compare the effect of each parameter value, see “Decide How to Visualize Simulation Data” on page 30-2.

Improve Simulation Speed

To perform many simulations that each use different parameter values, you can use accelerated simulation modes. For larger models, accelerated simulations take less time to execute than normal simulations. If you also have Parallel Computing Toolbox, you can use the multiple cores of your processor to simultaneously execute simulations. Use arguments of the `sim` and `parsim` functions.

To improve the simulation speed of your model by using accelerated simulations and other techniques, see “Optimize Performance”. For examples and more information, see “Run Multiple Simulations” on page 27-2.

Sweep Parameter Values to Test and Verify System

If you have Simulink Test, you can confirm that your model still meets requirements when you use different parameter values. Parameter overrides and test iterations enable you to set different parameter values for each test case. For more information, see “Parameter Overrides” (Simulink Test) and “Test Iterations” (Simulink Test).

Estimate and Calibrate Model Parameters

If you have Simulink Design Optimization, you can estimate model parameter values so that simulation outputs closely fit the data that you measure in the real world. Use this technique to estimate the real-world values of parameters in a plant model, which represents the dynamics of a real-world system, when you cannot directly measure the values. This estimation improves the accuracy of the plant model. For more information, see “Estimate Parameters from Measured Data” (Simulink Design Optimization).

Tune and Optimize PID and Controller Parameters

If you have Simulink Control Design, you can use PID Tuner to tune the parameters of a PID Controller block. For more information, see “PID Controller Tuning in Simulink” (Simulink Control Design).

If you have Simulink Design Optimization, you can optimize control parameter values so that simulation outputs meet response requirements that you specify. For more information, see “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization).

See Also

Related Examples

- “Data Objects” on page 67-58
- “Specify Minimum and Maximum Values for Block Parameters” on page 37-52
- “Sweep Variant Control Using Parallel Simulation” on page 26-15
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 37-56

Control Block Parameter Data Types

A block parameter, such as the **Gain** parameter of a Gain block, has a data type in the same way that a signal has a data type (see “Control Signal Data Types” on page 67-6). MATLAB variables, `Simulink.Parameter` objects, and other parameter objects that you use to set block parameter values also have data types. Control block parameter data types to:

- Accurately simulate the execution of your control algorithm on hardware.
- Generate efficient code.
- Integrate the generated code with your custom code.
- Avoid using data types that your target hardware does not support.

Reduce Maintenance Effort with Data Type Inheritance

By default, block parameters, numeric MATLAB variables that use the data type `double`, and `Simulink.Parameter` objects acquire a data type through inherited and context-sensitive data typing. For example, if the input and output signals of a Gain block use the data type `single`, the **Gain** parameter typically uses the same data type. If you use a `Simulink.Parameter` object to set the value of the block parameter, by default, the object uses the same data type as the parameter. You can take advantage of this inheritance to avoid explicitly specifying data types for parameters.

Some inheritance rules choose a parameter data type other than the data type that the corresponding signals use. For example, suppose that:

- The input and output signals of a Gain block use fixed-point data types with binary-point-only scaling.
- On the **Parameter Attributes** tab, **Parameter data type** is set to `Inherit: Inherit via internal rule` (the default).
- On the **Parameter Attributes** tab, you specify a minimum and maximum value for the parameter by using **Parameter minimum** and **Parameter maximum**.

The data type setting `Inherit: Inherit via internal rule` can cause the block to choose a different data type, with a different word length or scaling, than the data type that the signals use. The minimum and maximum values that you specified for the parameter influence the scaling that the block chooses.

When you select internal rules (`Inherit: Inherit via internal rule`) to enable Simulink to choose data types, before simulating or generating code, configure the characteristics of your target hardware. The internal rules can use these settings to choose data types that yield efficient generated code.

Context-Sensitive Data Typing

When you use a MATLAB variable or `Simulink.Parameter` object to set the value of a block parameter, you can configure the variable or parameter object to use context-sensitive data typing. When you simulate or generate code, the variable or parameter object uses the same data type as the block parameter. With this technique, you can match the data type of the variable or parameter object with the data type of the block parameter. To control the data type of the block parameter and the variable or object, you specify only the data type of the block parameter.

To use context-sensitive data typing, set the value of a MATLAB variable to a `double` value. For a `Simulink.Parameter` object, set the `Value` property by using a `double` value and set the `DataType` property to `auto` (the default).

Context-Sensitive Data Typing for Structure Fields

As described in “Organize Related Block Parameter Definitions in Structures” on page 37-19, you can organize multiple block parameter values into a structure.

The fields of parameter structures do not support context-sensitive data typing. However, to match the data type of a field with the data type of another data item in a model, you can use a bus object and a data type object.

1. Use a `Simulink.Bus` object as the data type of the structure.
2. Use a `Simulink.AliasType` or `Simulink.NumericType` object as the data type of the element in the bus object and as the data type of the target data item.

Techniques to Explicitly Specify Parameter Data Types

Many blocks supported for discrete-time simulation and code generation (such as those in the built-in Discrete library) enable you to explicitly specify parameter data types. For example, in an **n-D Lookup Table** block dialog box, on the **Data Types** tab, you can specify a data type for the lookup table data by using the **Table data** parameter. In a Gain block dialog box, use the **Parameter Attributes** tab to set **Parameter data type**, which controls the data type of the **Gain** parameter.

Some blocks, such as those in the Continuous library, do not enable you to specify parameter data types. These block parameters use internal rules to choose a data type. To indirectly control the data type of such a parameter, apply the data type to a `Simulink.Parameter` object instead.

When you use a `Simulink.Parameter` object or other parameter object to set the value of a block parameter, you can use the `DataType` property of the object to specify a data type.

If you use model arguments, you can specify a data type:

- For the model argument that you store in the model workspace.
- With some blocks (such as those in the Discrete library), for the block parameter that uses the model argument.
- For the argument value that you specify in a Model block.

The default settings for these data types typically use inheritance and context-sensitive data typing. For example, the default value of the `DataType` property of a `Simulink.Parameter` object is `auto`, which causes the parameter object to acquire a data type from the block parameter or parameters that use the object.

To explicitly specify a data type, you can use the Data Type Assistant in block dialog boxes and property dialog boxes. For information about the Data Type Assistant, see “Specify Data Types Using Data Type Assistant” on page 67-30.

Use the Model Data Editor for Batch Editing

Using the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), you can specify the same data type for multiple block parameters simultaneously. On the **Parameters** tab, set the **Change view** drop-down list to `Design` and specify data types by using the **Data Type** column.

For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Calculate Best-Precision Fixed-Point Scaling for Tunable Block Parameters

When you apply fixed-point data types to a model, you can use the Data Type Assistant and the Fixed-Point Tool to calculate best-precision scaling for tunable block parameters. A block parameter, `Simulink.Parameter` object, or other parameter object is tunable if it appears in the generated code as a variable stored in memory.

The chosen scaling must accommodate the range of values that you expect to assign to the parameter. To enable the tools to calculate an appropriate scaling, specify the range information in the block or in a parameter object. Then, use one of these techniques to calculate the scaling:

- Use the Fixed-Point Tool to autoscale the entire model or subsystem. The tool can propose and apply fixed-point data types for data items including block parameters, `Simulink.Parameter` objects, signals, and states.
- Configure individual block parameters or parameter objects to calculate their own scaling.

  When you later change value ranges for parameters, this technique enables you or the model to recalculate the scaling without having to autoscale the entire model. However, if changing the value range of the parameter also changes the value range of an associated signal, you must manually calculate and apply a new scaling for the signal or use the Fixed-Point Tool to autoscale the model or subsystem.

For basic information about fixed-point data types, block parameters, and other tools and concepts, use the information in the table.

Topic	More Information
Fixed-point data types and scaling	“Fixed-Point Numbers in Simulink” (Fixed-Point Designer)
How to specify value range information for block parameters and parameter objects	“Specify Minimum and Maximum Values for Block Parameters” on page 37-52
How to use the Data Type Assistant	“Specify Data Types Using Data Type Assistant” on page 67-30
Tunability and block parameter representation in the generated code	“How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder) and “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Autoscale Entire Model by Using the Fixed-Point Tool

You can use the Fixed-Point Tool to autoscale data items in your model, including tunable parameters and signals whose values depend on those parameters. If you use this technique:

- To configure parameters as tunable, use parameter objects (for example, `Simulink.Parameter`) instead of the Model Parameter Configuration dialog box. The Fixed-Point Tool can autoscale parameter objects, but cannot autoscale numeric variables that you select through the Model Parameter Configuration dialog box.

If your model already uses the Model Parameter Configuration dialog box, use the `tunablevars2parameterobjects` function to create parameter objects instead.

- When you use `Simulink.Parameter` objects to set block parameter values, specify the value range information in the objects instead of the blocks. The Fixed-Point Tool uses the range information in each object to propose a data type for that object.
- To enable the tool to autoscale parameter values that you store as fields of a structure, use a `Simulink.Bus` object as the data type of the entire structure. Specify the range information for each field by using the `Min` and `Max` properties of the corresponding element in the bus object. The tool can then apply a data type to each element by using the `DataType` property.

  To use a bus object as the data type of a parameter structure, see “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21.
- Before you apply the data types that the Fixed-Point Tool proposes, clear the proposals for parameters and parameter objects whose data types you do not want the tool to change. For example, clear the proposals for these entities:

  - Parameter objects that you import into the generated code from your own handwritten code by applying a storage class such as `ImportedExtern`.
  - `Simulink.Parameter` model arguments in a model workspace.

    Alternatively, before autoscaling the model, consider replacing these parameter objects with numeric MATLAB variables to prevent the Fixed-Point Tool from autoscaling them.

    Allowing the tool to autoscale model arguments can increase the risk of unintentional data type mismatches between the model argument values (which you specify in Model blocks in a parent model), the model arguments in the model workspace, and the client block parameters in the model.
  - Parameter objects whose `DataType` property is set to `auto` (context-sensitive). Clear the proposals if you want the parameter objects to continue to use context-sensitive data typing.

For more information about using the Fixed-Point Tool to autoscale `Simulink.Parameter` objects, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

Calculate Best-Precision Scaling for Individual Parameters

You can configure a block parameter or `Simulink.Parameter` object to calculate its own best-precision scaling. First, specify value range information for the target parameter or parameter object. Then, use the Data Type Assistant or the function `fixdt` to apply a data type to the parameter or object. Use these techniques when you do not want to use the Fixed-Point Tool to autoscale the model.

Enable Block Parameter to Automatically Calculate Best-Precision Scaling

You can enable the parameters of some blocks (typically blocks in the Discrete library) to automatically calculate best-precision fixed-point scaling. Use this technique to store the range and data type information in the model instead of a parameter object. When you use this technique, if you later change the range information, the block parameter automatically recalculates best-precision scaling.

In the block dialog box, use the function `fixdt` to specify a fixed-point data type with unspecified scaling. For example, use best-precision scaling for lookup table data, and store the data in a 16-bit word:

1. On the **Data Types** tab of an n-D Lookup Table block, under the **Minimum** and **Maximum** columns, specify a value range for the elements of the table data.
2. Under the **Data Type** column, set the table data type to `fixdt(1,16)`.
3. If you use a tunable `Simulink.Parameter` object to set the value of the table data parameter, set the `DataType` property of the object to `auto`. In the generated code, the parameter object uses the same scaling as the block parameter.

When you simulate or generate code, the lookup table data uses a signed 16-bit fixed-point data type whose binary-point scaling depends on the range information that you specify. The calculated scaling allows the fixed-point type to represent values that lie within the range. If you later change the minimum or maximum values, the block recalculates the scaling when you simulate or generate code.

Calculate Scaling for Parameter Object

If you use a `Simulink.Parameter` object to set the values of multiple block parameters, and if the block parameters use different data types (including different fixed-point scaling), you cannot set the `DataType` property of the object to `auto` (the default). Instead, you can calculate best-precision fixed-point scaling for the parameter object by specifying range and data type information in the object. You can also use this technique to store range and data type information in a parameter object instead of a block dialog box. When you use this technique, if you later change the range information, you must recalculate the best-precision scaling by using the Data Type Assistant.

Suppose that you create a parameter object to represent the value `15.25`, and that the design range of the value is between `0.00` and `32.00`. To calculate best-precision scaling, use the Data Type Assistant.

1. At the command prompt, create a parameter object in the base workspace whose value is `15.25`.

   ```
   myParam = Simulink.Parameter(15.25);
   ```
2. In the MATLAB Workspace browser, double-click the object `myParam`. The property dialog box opens.
3. Specify range information in the object. For example, set **Minimum** to `0.00` and **Maximum** to `32.00`.
4. Set **Data type** to `fixdt(0,16,0)`.
5. Expand the Data Type Assistant and click **Calculate Best-Precision Scaling**.

   The data type changes from `fixdt(0,16,0)` to `fixdt(0,16,10)`.

The calculated scaling (a fraction length of 10 bits) enables the fixed-point data type to represent parameter values that lie within the range that you specified.

If you specify range and data type information in a parameter object, consider removing the range and data type information from the blocks that use the object. Some tools, such as the Fixed-Point Tool, ignore the range information that you specify in the block and use only the information in the parameter object. Removing the information from the block prevents confusion and user errors.

For example, on the **Parameter Attributes** tab of a Gain block dialog box, set **Parameter minimum** and **Parameter maximum** to `[]`. Set **Parameter data type** to an inherited data type such as `Inherit: Inherit from 'Gain'` so that the block parameter uses the same data type as the parameter object.

Detect Numerical Accuracy Issues Due to Quantization and Overflow

When the data type of a block parameter, MATLAB variable, or parameter object cannot represent the value that you specify, the data type quantizes the value, compromising numerical accuracy. For example, the 32-bit floating-point data type `single` (`float` in C code) cannot exactly represent the parameter value `1.73`. When the real-world value of a data item lies outside the range of values that the data type can represent, overflow can cause loss of information.

To detect these issues, use the diagnostic configuration parameters under **Configuration Parameters > Diagnostics > Data Validity > Parameters**. Set the values of these diagnostic configuration parameters to `warning` or `error`:

- **Detect downcast**
- **Detect precision loss**
- **Detect underflow**
- **Detect overflow**

Reuse Custom C Data Types for Parameter Data

In a model, you can create parameter data that conform to custom C data types, such as structures, that your existing C code defines. Use these data to:

- Replace existing C code with a Simulink model.
- Integrate C code for simulation in Simulink (for example, by using the Legacy Code Tool).
- Prepare to generate code (Simulink Coder) that you can integrate with existing code.

Use these techniques to match your custom data types:

- For a structure type, create a `Simulink.Bus` object. Use the object as the data type for a structure that you store in a `Simulink.Parameter` object. See “Organize Related Block Parameter Definitions in Structures” on page 37-19.
- For an enumeration, create an enumeration class and use it as the data type for block parameters. See “Use Enumerated Data in Simulink Models” on page 68-6.
- To match a `typedef` statement that represents an alias of a primitive, numeric data type, use a `Simulink.AliasType` object as the data type for block parameters. See `Simulink.AliasType`.

To create these classes and objects, you can use the function `Simulink.importExternalCTypes`.

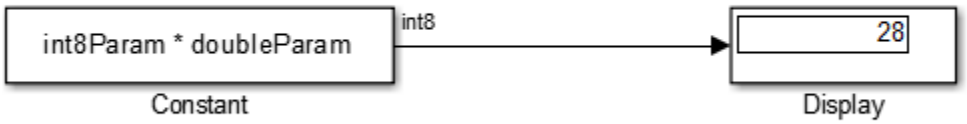
If a MATLAB Function block or Stateflow chart in your model uses an imported enumeration or structure type, configure the model configuration parameters to include (`#include`) the type definition from your external header file. See “Control Imported Bus and Enumeration Type Definitions” on page 44-124 (for a MATLAB Function block) and “Access Custom Code Variables and Functions in Stateflow Charts” (Stateflow) and “Integrate Custom Structures in Stateflow Charts” (Stateflow) (for a chart).

Data Types of Mathematical Expressions

If you specify a block parameter using a mathematical expression, the block determines the final parameter data type using a combination of MATLAB and Simulink data typing rules.

Suppose that you define two parameter objects `int8Param` and `doubleParam`, and use the objects to specify the **Constant value** parameter in a Constant block.

```
int8Param = Simulink.Parameter(3);
int8Param.DataType = 'int8';

doubleParam = Simulink.Parameter(9.36);
doubleParam.DataType = 'double';
```



The Constant block determines the data type of the **Constant value** parameter using these steps:

1. Each parameter object casts the specified numeric value to the specified data type.

Parameter object	Data type	Numeric value	Result
`int8Param`	`int8`	3	`int8(3)`
`doubleParam`	`double`	9.36	`double(9.36)`

2. The block evaluates the specified expression, `int8Param * doubleParam`, using MATLAB rules.

   An expression that involves a `double` data type and a different type returns a result of the different type. Therefore, the result of the expression `int8(3) * double(9.36)` is `int8(28)`.

If you use an expression to set the value of a parameter object (such as `Simulink.Parameter`), parameter objects used in the expression follow different data typing rules. The `auto` setting of the `DataType` property has a slightly different meaning. See “Set Variable Value by Using a Mathematical Expression” on page 37-10.

Block Parameter Data Types in the Generated Code

For more information about controlling parameter data types in the generated code, see “Parameter Data Types in the Generated Code” (Simulink Coder).

See Also

Related Examples

- “Set Block Parameter Values” on page 37-2
- “Specify Minimum and Maximum Values for Block Parameters” on page 37-52
- “Tune and Experiment with Block Parameter Values” on page 37-31

- “Data Validity Diagnostics Overview”

Specify Minimum and Maximum Values for Block Parameters

You can protect your model design by preventing block parameters from using values outside of a range. For example, if the value of a parameter that represents the angle of an aircraft aileron cannot feasibly exceed a known magnitude, you can specify a design maximum for the parameter in the model.

Fixed-Point Designer enables Simulink to use your range information to calculate best-precision fixed-point scaling for:

- Tunable parameters.
- Signals, by taking into consideration the range of values that you intend to assign to tunable parameters.

For basic information about block parameters, see “Set Block Parameter Values” on page 37-2.

Specify Parameter Value Ranges

When you specify a value range for a block parameter, typically, you can choose to store the information in the block (the model file) or in an external variable or parameter object. Choose a technique based on your modeling goals.

- Use other parameters of the same block, if available. For example, you can control the value range of the **Gain** parameter of a Gain block by using the **Parameter minimum** and **Parameter maximum** parameters in the **Parameter Attributes** tab in the block dialog box. For other blocks, such as n-D Lookup Table and PID Controller, use the **Data Types** tab.

  Use this technique to:

  - Store the range information in the model file.
  - Store the range information when you store fixed-point data type information in the block (for example, by setting the **Parameter data type** parameter of a Gain block to a fixed-point type, including best-precision scaling). This technique more clearly associates the range information with the data type information.
- Use parameter objects (for example, `Simulink.Parameter`) to set the parameter value. You can specify the range information in the object, instead of the block, by using the `Min` and `Max` properties of the object.

  Use this technique to:

  - Specify range information for blocks that cannot store minimum or maximum information, for example, many blocks in the Continuous library.
  - Specify range information for a single value that you share between multiple block parameters (see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9). Instead of using a numeric MATLAB variable, use a parameter object so that you can specify the `Min` and `Max` properties.
  - Store the range information when you store fixed-point data type information in a parameter object (by setting the `DataType` property to a fixed-point type instead of `auto`). This technique more clearly associates the range information with the data type information.

  If you specify the range information in a parameter object, consider removing the range information from the block. For example, on the **Parameter Attributes** tab of a Gain block dialog

box, set **Parameter minimum** and **Parameter maximum** to `[]`. Some tools, such as the Fixed-Point Tool, use the range information that you specify in the block only if you do not specify the range information in the parameter object. If you specify the range information in the parameter object, the tools ignore the range information that you specify in the block.

For basic information about creating and using data objects, see “Data Objects” on page 67-58.

Specify Valid Range Information

Specify a minimum or maximum as an expression that evaluates to a scalar, real number with `double` data type. For example, you can specify a minimum value for the **Gain** parameter in a Gain block by setting **Parameter minimum**:

- A literal number such as `98.884`. Implicitly, the data type is `double`.
- A numeric workspace variable (see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9) whose data type is `double`. Use this technique to share a minimum or maximum value between multiple data items.

  However, you cannot use variables to set the `Min` or `Max` properties of a parameter object.

To leave the minimum or maximum of a block parameter or parameter object unspecified, use an empty matrix `[]`, which is the default value.

Specify Range Information for Nonscalar Parameters

If the value of a block parameter is a vector or matrix, the range information that you specify applies to each element of the vector or matrix. If the value of any of the elements is outside the specified range, the model generates an error.

If the value of a block parameter is a structure or a field of a structure, specify range information for the structure fields by creating a `Simulink.Parameter` object whose data type is a `Simulink.Bus` object. Specify the range information by using the properties of the signal elements in the bus object. For more information, see “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21.

Specify Range Information for Complex-Valued Parameters

If the value of a block parameter is complex (i), the range information that you specify applies separately to the real part and to the imaginary part of the complex number. If the value of either part of the number is outside the range, the model generates an error.

Specify Ranges for Multiple Parameters by Using the Model Data Editor

Using the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), you can specify value ranges for multiple block parameters and variables with a searchable, sortable table. On the **Parameters** tab, set the **Change view** drop-down list to `Design` and specify values in the **Min** and **Max** columns.

For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Restrict Allowed Values for Block Parameters

To protect your design by preventing block parameters from using values outside of a range, you can specify the minimum and maximum information by using other parameters of the same block. If you

or your users set the value of the target parameter outside the range that you specify, the model generates an error.

Whether a block allows you to specify a value range for a parameter, consider using a parameter object (for example, `Simulink.Parameter`) to set the value of the target parameter. Use the properties of the object to specify the range information. This technique helps you to specify range information for a variable that you use to set multiple block parameter values.

Specify Range Information for Tunable Fixed-Point Parameters

When you use fixed-point data types in your model, you can enable Simulink to choose a best-precision scaling for block parameters and `Simulink.Parameter` objects. If you intend to tune such a parameter by changing its value during simulation or during execution of the generated code, the fixed-point scaling chosen by Simulink must accommodate the range of values that you expect to assign to the parameter.

Also, if you expect to change the value of a parameter, signal data types in the model must accommodate the corresponding expanded range of possible signal values. If you use the Fixed-Point Tool to propose and apply fixed-point data types for a model, to allow the tool to accurately autoscale the signals, specify range information for tunable parameters.

To specify range information for tunable parameters, see “Calculate Best-Precision Fixed-Point Scaling for Tunable Block Parameters” on page 37-46. To learn how the Fixed-Point Tool autoscales signals by taking into account the value ranges of tunable parameters, see “Derive Ranges for Simulink.Parameter Objects” (Fixed-Point Designer).

Unexpected Errors or Warnings for Data with Greater Precision or Range than double

When a data item (signal or parameter) uses a data type other than `double`, before comparison, Simulink casts the data item and each design limit (minimum or maximum value that you specify) to the nondouble data type. This technique helps prevent the generation of unnecessary, misleading errors and warnings.

However, Simulink stores design limits as `double` before comparison. If the data type of the data item has higher precision than `double` (for example, a fixed-point data type with a 128-bit word length and a 126-bit fraction length) or greater range than `double`, and `double` cannot exactly represent the value of a design limit, Simulink can generate unexpected warnings and errors.

If the nondouble type has higher precision, consider rounding the design limit to the next number furthest from zero that `double` can represent. For example, suppose that a signal generates an error after you set the maximum value to `98.8847692348509014`. At the command prompt, calculate the next number furthest from zero that `double` can represent.

```
format long
98.8847692348509014 + eps(98.8847692348509014)

ans =

  98.884769234850921
```

Use the resulting number, `98.884769234850921`, to replace the maximum value.

Optimize Generated Code

If you have Embedded Coder, Simulink Coder can optimize the code that you generate from the model by taking into account the minimum and maximum values that you specify for signals and parameters. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Embedded Coder).

See Also

Related Examples

- “Control Block Parameter Data Types” on page 37-44
- “Tune and Experiment with Block Parameter Values” on page 37-31

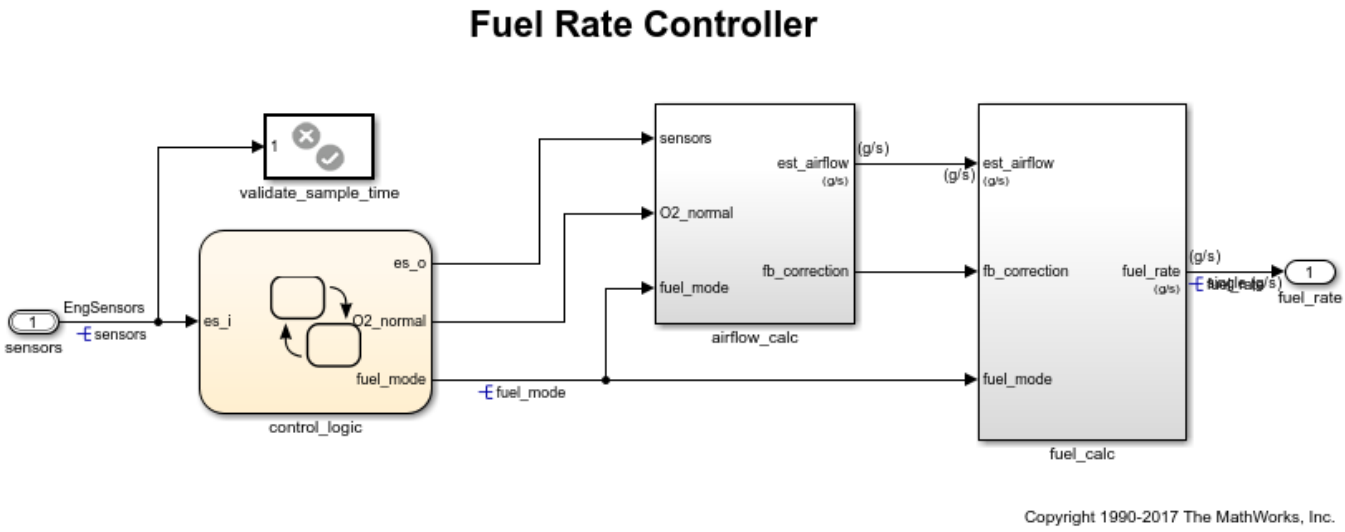
Switch Between Sets of Parameter Values During Simulation and Code Execution

To store multiple independent sets of values for the same block parameters, you can use an array of structures. To switch between the parameter sets, create a variable that acts as an index into the array, and change the value of the variable. You can change the value of the variable during simulation and, if the variable is tunable, during execution of the generated code.

Explore Example Model

Open the example model.

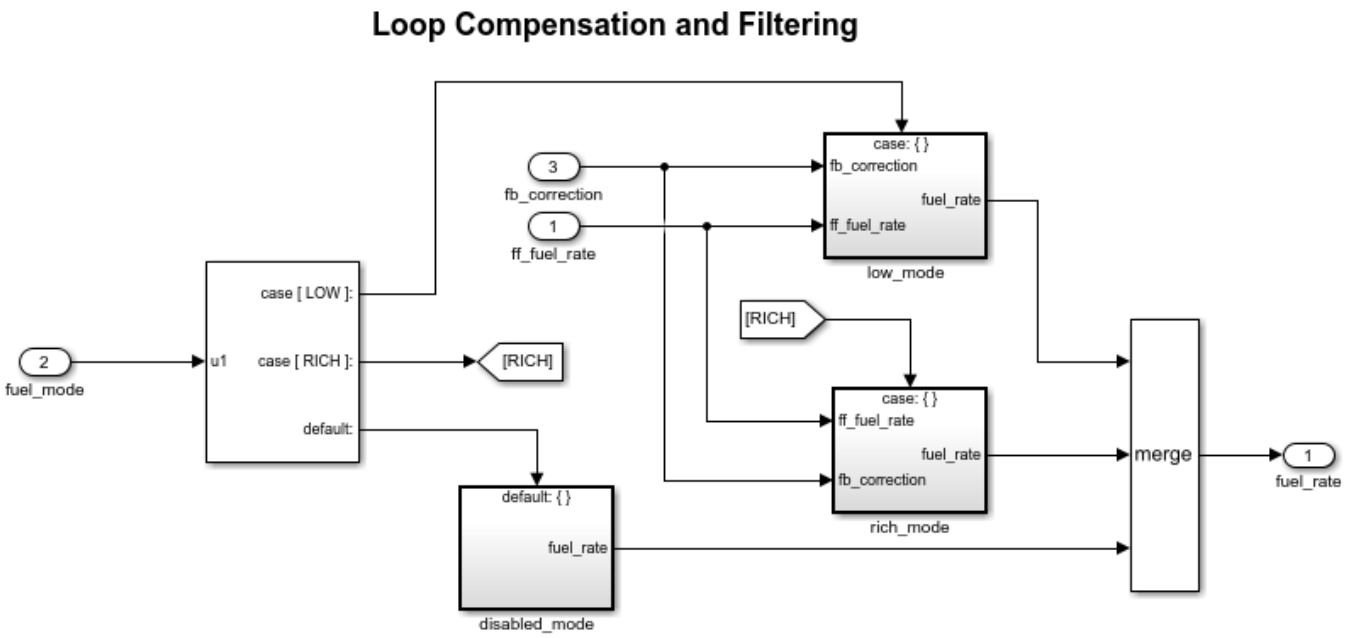
```
open_system('sldemo_fuelsys_dd_controller')
```



This model represents the fueling system of a gasoline engine. The output of the model is the rate of fuel flow to the engine.

Navigate to the `switchable_compensation` nested subsystem.

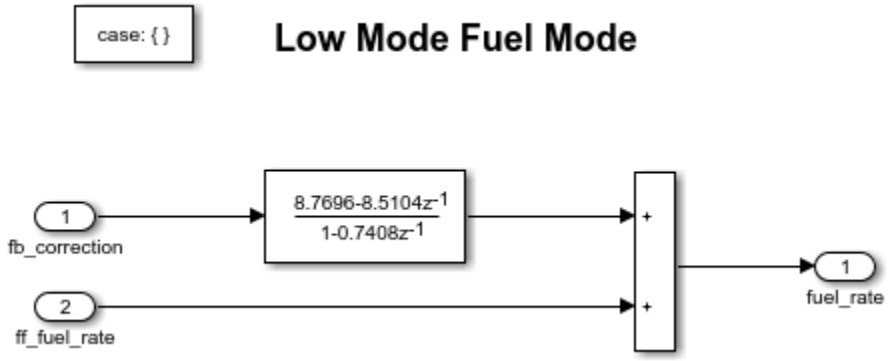
```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation'])
```



This subsystem corrects and filters noise out of the fuel rate signal. The subsystem uses different filter coefficients based on the fueling mode, which the control logic changes based on sensor failures in the engine. For example, the control algorithm activates the `low_mode` subsystem during normal operation. It activates the `rich_mode` subsystem in response to sensor failure.

Open the `low_mode` subsystem.

```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation/low_mode'])
```



The Discrete Filter block filters the fuel rate signal. In the block dialog box, the **Numerator** parameter sets the numerator coefficients of the filter.

The sibling subsystem `rich_mode` also contains a Discrete Filter block, which uses different coefficients.

Update the model diagram to display the signal data types. The input and output signals of the block use the single-precision, floating-point data type `single`.

In the lower-left corner of the model, click the model data badge, then click the **Data Dictionary** link. The data dictionary for this model, `sldemo_fuelsys_dd_controller.sldd`, opens in the Model Explorer.

In the Model Explorer **Model Hierarchy** pane, select the **Design Data** node.

In the **Contents** pane, view the properties of the `Simulink.NumericType` objects, such as `s16En15`. All of these objects currently represent the single-precision, floating-point data type `single`. The model uses these objects to set signal data types, including the input and output signals of the Discrete Filter blocks.

Suppose that during simulation and execution of the generated code, you want each of these subsystems to switch between different numerator coefficients based on a variable whose value you control.

Store Parameter Values in Array of Structures

Store the existing set of numerator coefficients in a `Simulink.Parameter` object whose value is a structure. Each field of the structure stores the coefficients for one of the Discrete Filter blocks.

```
lowBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/low_mode/Discrete Filter'];
richBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/rich_mode/Discrete Filter'];
params.lowNumerator = eval(get_param(lowBlock,'Numerator'));
params.richNumerator = eval(get_param(richBlock,'Numerator'));
params = Simulink.Parameter(params);
```

Copy the value of `params` into a temporary variable. Modify the field values in this temporary structure, and assign the modified structure as the second element of `params`.

```
temp = params.Value;
temp.lowNumerator = params.Value.lowNumerator * 2;
temp.richNumerator = params.Value.richNumerator * 2;
params.Value(2) = temp;
clear temp
```

The value of `params` is an array of two structures. Each structure stores one set of filter coefficients.

Create Variable to Switch Between Parameter Sets

Create a `Simulink.Parameter` object named `Ctrl`.

```
Ctrl = Simulink.Parameter(2);
Ctrl.DataType = 'uint8';
```

In the `low_mode` subsystem, in the Discrete Filter block dialog box, set the **Numerator** parameter to the expression `params(Ctrl).lowNumerator`.

```
set_param(lowBlock,'Numerator','params(Ctrl).lowNumerator');
```

In the Discrete Filter block in the `rich_mode` subsystem, set the value of the **Numerator** parameter to `params(Ctrl).richNumerator`.

```
set_param(richBlock,'Numerator','params(Ctrl).richNumerator');
```

The expressions select one of the structures in `params` by using the variable `Ctrl`. The expressions then dereference one of the fields in the structure. The field value sets the values of the numerator coefficients.

To switch between the sets of coefficients, you change the value of `Ctrl` to the corresponding index in the array of structures.

Use Bus Object as Data Type of Array of Structures

Optionally, create a `Simulink.Bus` object to use as the data type of the array of structures. You can:

- Control the shape of the structures.
- For each field, control characteristics such as data type and physical units.
- Control the name of the `struct` type in the generated code.

Use the function `Simulink.Bus.createObject` to create the object and rename the object as `paramsType`.

```
Simulink.Bus.createObject(params.Value)
paramsType = slBus1;
clear slBus1
```

You can use the `Simulink.NumericType` objects from the data dictionary to control the data types of the structure fields. In the bus object, use the name of a data type object to set the `DataType` property of each element.

```
paramsType.Elements(1).DataType = 's16En15';
paramsType.Elements(2).DataType = 's16En7';
```

Use the bus object as the data type of the array of structures.

```
params.DataType = 'Bus: paramsType';
```

Use Enumerated Type for Switching Variable

Optionally, use an enumerated type as the data type of the switching variable. You can associate each of the parameter sets with a meaningful name and restrict the allowed values of the switching variable.

Create an enumerated type named `FilterCoeffs`. Create an enumeration member for each of the structures in `params`. Set the underlying integer value of each enumeration member to the corresponding index in `params`.

```
Simulink.defineIntEnumType('FilterCoeffs',{'Weak','Aggressive'},[1 2])
```

Use the enumerated type as the data type of the switching variable. Set the value of the variable to `Aggressive`, which corresponds to the index 2.

```
Ctrl.Value = FilterCoeffs.Aggressive;
```

Add New Objects to Data Dictionary

Add the objects that you created to the data dictionary `sldemo_fuelsys_dd_controller.sldd`.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
sectObj = getSection(dictObj,'Design Data');
addEntry(sectObj,'Ctrl',Ctrl)
```

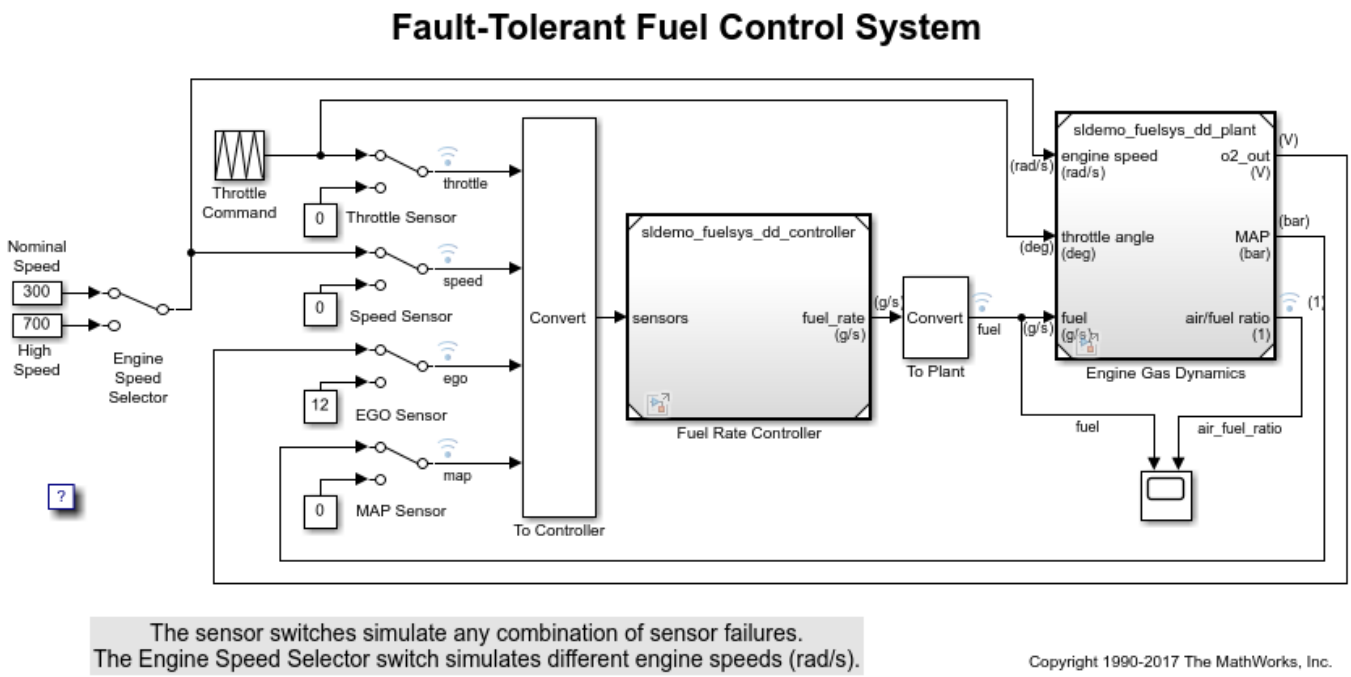
```
addEntry(sectObj,'params',params)
addEntry(sectObj,'paramsType',paramsType)
```

You can also store enumerated types in data dictionaries. However, you cannot import the enumerated type in this case because you cannot save changes to `sldemo_fuelsys_dd_controller.sldd`. For more information about storing enumerated types in data dictionaries, see “Enumerations in Data Dictionary” on page 74-12.

Switch Between Parameter Sets During Simulation

Open the example model `sldemo_fuelsys_dd`, which references the controller model `sldemo_fuelsys_dd_controller`.

```
open_system('sldemo_fuelsys_dd')
```



Set the simulation stop time to `Inf` so that you can interact with the model during simulation.

Begin a simulation run and open the Scope block dialog box. The scope shows that the fuel flow rate (the `fuel` signal) oscillates with significant amplitude during normal operation of the engine.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. Set the value of `Ctrl` to `FilterCoeffs.Weak`.

Update the `sldemo_fuelsys_dd` model diagram. The scope shows that the amplitude of the fuel rate oscillations decreases due to the less aggressive filter coefficients.

Stop the simulation.

Generate and Inspect Code

If you have Simulink Coder software, you can generate code that enables you to switch between the parameter sets during code execution.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. In the **Contents** pane, set **Column View** to `Storage Class`.

Use the **StorageClass** column to apply the storage class `ExportedGlobal` to `params` so that the array of structures appears as a tunable global variable in the generated code. Apply the same storage class to `Ctrl` so that you can change the value of the switching variable during code execution.

Alternatively, to configure the objects, use these commands:

```
tempEntryObj = getEntry(sectObj,'params');
params = getValue(tempEntryObj);
params.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,params);

tempEntryObj = getEntry(sectObj,'Ctrl');
Ctrl = getValue(tempEntryObj);
Ctrl.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,Ctrl);
```

Generate code from the controller model.

```
rtwbuild('sldemo_fuelsys_dd_controller')
```

```
### Starting build procedure for: sldemo_fuelsys_dd_controller
### Successful completion of code generation for: sldemo_fuelsys_dd_controller

Build Summary

Top model targets built:

Model                          Action          Rebuild Reason
=========================================================================================================
sldemo_fuelsys_dd_controller  Code generated  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 45.54s
```

In the code generation report, view the header file `sldemo_fuelsys_dd_controller_types.h`. The code defines the enumerated data type `FilterCoeffs`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_types.h');
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_
#define DEFINED_TYPEDEF_FOR_FilterCoeffs_

typedef enum {
  Weak = 1,                            /* Default value */
  Aggressive
} FilterCoeffs;

#endif
```

The code also defines the structure type `paramsType`, which corresponds to the `Simulink.Bus` object. The fields use the single-precision, floating-point data type from the model.

```
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_paramsType_',...
    '#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_paramsType_
#define DEFINED_TYPEDEF_FOR_paramsType_

typedef struct {
  real32_T lowNumerator[2];
  real32_T richNumerator[2];
} paramsType;

#endif
```

View the source file `sldemo_fuelsys_dd_controller.c`. The code uses the enumerated type to define the switching variable `Ctrl`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller.c');
rtwdemodbtype(file,'FilterCoeffs Ctrl = Aggressive;',...
    '/* Block signals (default storage) */',1,0)
```

```
FilterCoeffs Ctrl = Aggressive;        /* Variable: Ctrl
                                        * Referenced by:
                                        *   '<S12>/Discrete Filter'
                                        *   '<S13>/Discrete Filter'
                                        */
```

The code also defines the array of structures `params`.

```
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Variable: params',1,1)
```

```
/* Exported block parameters */
paramsType params[2] = { {
    { 8.7696F, -8.5104F },

    { 0.0F, 0.2592F }
  }, { { 17.5392F, -17.0208F },

    { 0.0F, 0.5184F }
  } } ;                                /* Variable: params
```

The code algorithm in the model `step` function uses the switching variable to index into the array of structures.

To switch between the parameter sets stored in the array of structures, change the value of `Ctrl` during code execution.

Close the connections to the data dictionary. This example discards unsaved changes. To save the changes, use the `'-save'` option.

```
Simulink.data.dictionary.closeAll('sldemo_fuelsys_dd_controller.sldd','-discard')
```

See Also

Related Examples

- “Tune and Experiment with Block Parameter Values” on page 37-31
- “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)
- “Organize Related Block Parameter Definitions in Structures” on page 37-19
- “Access Structured Data Through a Pointer That External Code Defines” (Embedded Coder)

Working with Lookup Tables

About Lookup Table Blocks

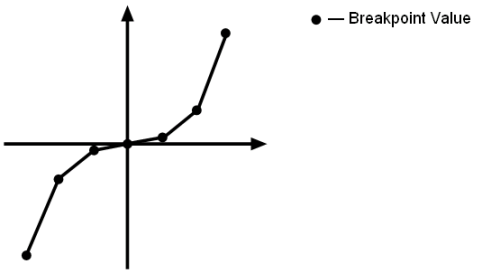
A lookup table is an array of data that maps input values to output values, thereby approximating a mathematical function. Given a set of input values, a lookup operation retrieves the corresponding output values from the table. If the lookup table does not explicitly define the input values, Simulink can estimate an output value using interpolation, extrapolation, or rounding, where:

- An interpolation is a process for estimating values that lie between known data points.
- An extrapolation is a process for estimating values that lie beyond the range of known data points.
- A rounding is a process for approximating a value by altering its digits according to a known rule.

A lookup table block uses an array of data to map input values to output values, approximating a mathematical function. Given input values, Simulink performs a “lookup” operation to retrieve the corresponding output values from the table. If the lookup table does not define the input values, the block estimates the output values based on nearby table values.

The following example illustrates a one-dimensional lookup table that approximates the function $y = x^3$. The lookup table defines its output (y) data discretely over the input (x) range `[-3, 3]`. The following table and graph illustrate the input/output relationship:

x	-3	-2	-1	0	1	2	3
y	-27	-8	-1	0	1	8	27



An input of -2 enables the table to look up and retrieve the corresponding output value (-8). Likewise, the lookup table outputs 27 in response to an input of 3.

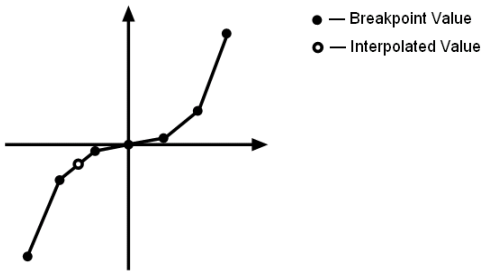
When the lookup table block encounters an input that does not match any of the table's x values, it can interpolate or extrapolate the answer. For instance, the lookup table does not define an input value of -1.5; however, the block can linearly interpolate the nearest adjacent data points (x_i, y_i) and (x_{i+1}, y_{i+1}). For example, given these two points:

- (x_i, y_i) is (-2, -8)
- (x_{i+1}, y_{i+1}) is (-1, -1)

The lookup table estimates and returns a value of -4.5.

Linearly interpolated value

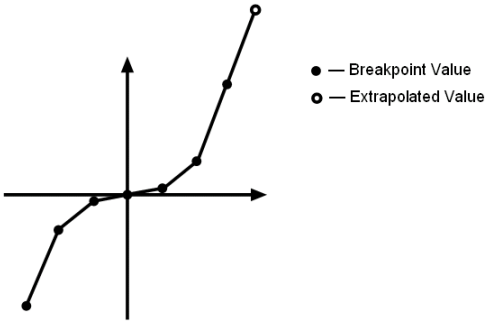
x	-3	-2	-1.5	-1	0	1	2	3
y	-27	-8	-4.5	-1	0	1	8	27



Similarly, although the lookup table does not include data for *x* values beyond the range of `[-3, 3]`, the block can extrapolate values using a pair of data points at either end of the table. Given an input value of 4, the lookup table block linearly extrapolates the nearest data points (2, 8) and (3, 27) to estimate an output value of 46.

Linearly extrapolated value

x	-3	-2	-1	0	1	2	3	4
y	-27	-8	-1	0	1	8	27	46



Since table lookups and simple estimations can be faster than mathematical function evaluations, using lookup table blocks might result in speed gains when simulating a model. Consider using lookup tables in lieu of mathematical function evaluations when:

- An analytical expression is expensive to compute.
- No analytical expression exists, but the relationship has been determined empirically.

Simulink provides a broad assortment of lookup table blocks, each geared for a particular type of application. The sections that follow outline the different offerings, suggest how to choose the lookup table best suited to your application, and explain how to interact with the various lookup table blocks.

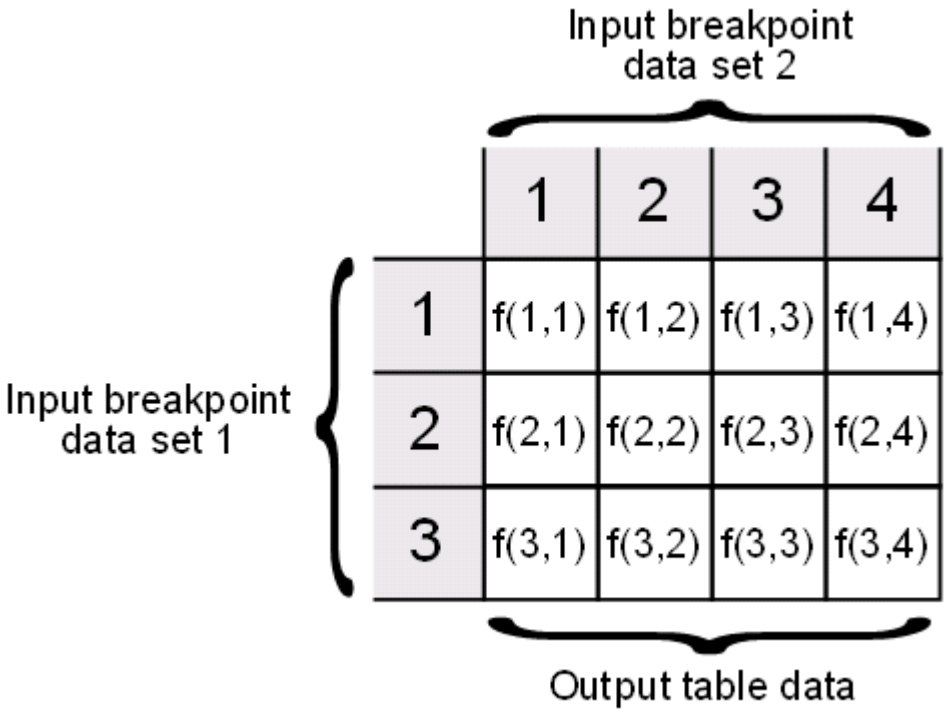
See Also

More About

- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5

Anatomy of a Lookup Table

The following figure illustrates the anatomy of a two-dimensional lookup table. Vectors or breakpoint data sets and an array, referred to as table data, constitute the lookup table.



A breakpoint is a single element of a breakpoint data set. A breakpoint represents a particular input value to which a corresponding output value in the table data is mapped. Each breakpoint data set is an index of input values for a particular dimension of the lookup table. The array of table data serves as a sampled representation of a function evaluated at the breakpoint values. Lookup table blocks use breakpoint data sets to relate a table's input values to the output values that it returns.

A breakpoint data set is a vector of input values that indexes a particular dimension of a lookup table. A lookup table uses breakpoint data sets to relate its input values to the output values that it returns.

Table data is an array that serves as a sampled representation of a function evaluated at a lookup table's breakpoint values. A lookup table uses breakpoint data sets to index the table data, ultimately returning an output value.

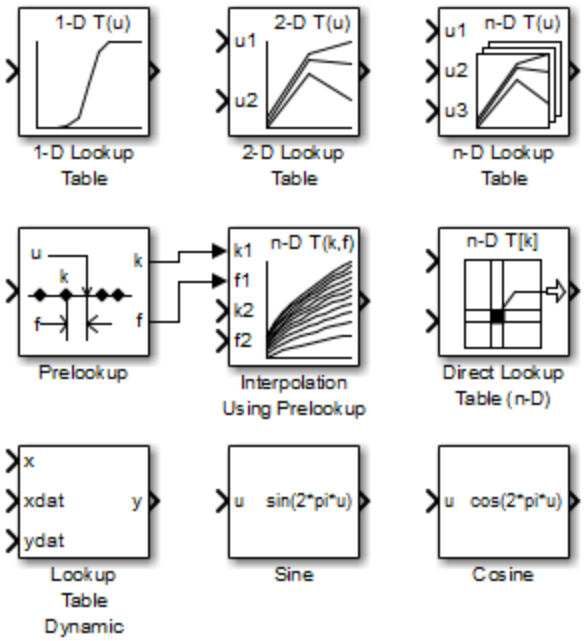
See Also

More About

- “About Lookup Table Blocks” on page 38-2
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20

Lookup Tables Block Library

Several lookup table blocks appear in the Lookup Tables block library.



The following table summarizes the purpose of each block in the library.

Block Name	Description
1-D Lookup Table	Approximate a one-dimensional function.
2-D Lookup Table	Approximate a two-dimensional function.
n-D Lookup Table	Approximate an N-dimensional function.
Prelookup	Compute index and fraction for Interpolation Using Prelookup block.
Interpolation Using Prelookup	Use precalculated index and fraction values to accelerate approximation of N-dimensional function.
Direct Lookup Table (n-D)	Index into an N-dimensional table to retrieve the corresponding outputs.
Lookup Table Dynamic	Approximate a one-dimensional function using a dynamically specified table.
Sine	Use a fixed-point lookup table to approximate the sine wave function.
Cosine	Use a fixed-point lookup table to approximate the cosine wave function.

See Also

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Edit Lookup Tables” on page 38-20

- “Guidelines for Choosing a Lookup Table” on page 38-7

Guidelines for Choosing a Lookup Table

In this section...

Data Set Dimensionality

In some cases, the dimensions of your data set dictate which of the lookup table blocks is right for your application. If you are approximating a one-dimensional function, consider using either the 1-D Lookup Table or Lookup Table Dynamic block. If you are approximating a two-dimensional function, consider the 2-D Lookup Table block. Blocks such as the n-D Lookup Table and Direct Lookup Table (n-D) allow you to approximate a function of *N* variables.

Data Set Numeric and Data Types

The numeric and data types of your data set influence the decision of which lookup table block is most appropriate. Although all lookup table blocks support real numbers, the Direct Lookup Table (n-D), 1-D Lookup Table, 2-D Lookup Table, and n-D Lookup Table blocks also support complex table data. All lookup table blocks support integer and fixed-point data in addition to `double` and `single` data types.

Note For the Direct Lookup Table (n-D) block, fixed-point types are supported for the table data, output port, and optional table input port.

Data Accuracy and Smoothness

The desired accuracy and smoothness of the data returned by a lookup table determine which of the blocks you should use. Most blocks provide options to perform interpolation and extrapolation, improving the accuracy of values that fall between or outside of the table data, respectively. For instance, the Lookup Table Dynamic block performs linear interpolation and extrapolation, while the n-D Lookup Table block performs either linear, cubic spline interpolation and extrapolation, or Akima spline interpolation and extrapolation. In contrast, the Direct Lookup Table (n-D) block performs table lookups without any interpolation or extrapolation. You can achieve a mix of interpolation and extrapolation methods by using the Prelookup block with the Interpolation Using Prelookup block.

Dynamics of Table Inputs

The dynamics of the lookup table inputs impact which of the lookup table blocks is ideal for your application. The blocks use a variety of index search methods to relate the lookup table inputs to the table's breakpoint data sets. Most of the lookup table blocks offer a binary search algorithm, which performs well if the inputs change significantly from one time step to the next. The 1-D Lookup Table,

2-D Lookup Table, n-D Lookup Table, and Prelookup blocks offer a linear search algorithm. Using this algorithm with the option that resumes searching from the previous result performs well if the inputs change slowly. Some lookup table blocks also provide a search algorithm that works best for breakpoint data sets composed of evenly spaced breakpoints. You can achieve a mix of index search methods by using the Prelookup block with the Interpolation Using Prelookup block.

Efficiency of Performance

When the efficiency with which lookup tables operate is important, consider using the Prelookup block with the Interpolation Using Prelookup block. These blocks separate the table lookup process into two components — an index search that relates inputs to the table data, followed by an interpolation and extrapolation stage that computes outputs. These blocks enable you to perform a single index search and then reuse the results to look up data in multiple tables. Also, the Interpolation Using Prelookup block can perform sub-table selection, where the block interpolates a portion of the table data instead of the entire table. For example, if your 3-D table data constitutes a stack of 2-D tables to be interpolated, you can specify a selection port input to select one or more of the 2-D tables from the stack for interpolation. A full 3-D interpolation has 7 sub-interpolations but a 2-D interpolation requires only 3 sub-interpolations. As a result, significant speed improvements are possible when some dimensions of a table are used for data stacking and not intended for interpolation. These features make table lookup operations more efficient, reducing computational effort and simulation time.

Summary of Lookup Table Block Features

Use the following table to identify features that correspond to particular lookup table blocks, then select the block that best meets your requirements.

Feature	1-D Lookup Table	2-D Lookup Table	Lookup Table Dynamic	n-D Lookup Table	Direct Lookup Table (n-D)	Prelookup	Interpolation Using Prelookup
Interpolation Methods							
Flat	•	•	•	•			•
Nearest	•	•		•			•
Linear			•				
Linear point-slope	•	•		•			•
Linear Lagrange	•	•		•			•
Cubic spline	•	•		•			
Akima spline	•	•		•			
Extrapolation Methods							
Clip	•	•	•	•		•	•
Linear	•	•	•	•		•	•
Cubic spline	•	•		•			
Akima spline	•	•					

Feature	1-D Lookup Table	2-D Lookup Table	Lookup Table Dynamic	n-D Lookup Table	Direct Lookup Table (n-D)	Prelookup	Interpolation Using Prelookup
Numeric & Data Type Support							
Complex	•	•		•	•		
Double, Single	•	•	•	•	•	•	•
Integer	•	•	•	•	•	•	•
Fixed point	•	•	•	•	•	•	•
Index Search Methods							
Binary	•	•	•	•		•	
Linear	•	•		•		•	
Evenly spaced points	•	•		•	•	•	
Start at previous index	•	•		•		•	
Miscellaneous							
Sub-table selection					•		•
Dynamic breakpoint data						•	
Dynamic table data			•		•		•
Input range checking	•	•		•	•	•	•

See Also

More About

- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20

Enter Breakpoints and Table Data

In this section...

Entering Data in a Block Parameter Dialog Box

This example shows how to populate a 1-D Lookup Table block using the parameter dialog box. The lookup table in this example approximates the function $y = x^3$ over the range `[-3, 3]`.

1. Copy a 1-D Lookup Table block from the Lookup Tables block library to a Simulink model.
2. In the 1-D Lookup Table block dialog box, enter the table dimensions and table data in the specified fields of the dialog box:

   - Set **Number of table dimensions** to `1`.
   - Set **Table data** to `[-27 -8 -1 0 1 8 27]`.

   Alternatively, to use an existing lookup table (`Simulink.LookupTable`) object, select **Data specification** > **Lookup table object**.
3. Enter the breakpoint data set using either of two methods:

   - To specify evenly spaced data points, set **Breakpoint specification** to `Even spacing`. Set **First point** to `-3` and **Spacing** to `1`. The block calculates the number of evenly spaced breakpoints based on the table data.
   - To specify breakpoint data explicitly, set **Breakpoint specification** to `Explicit values` and set **Breakpoints 1** to `[-3:3]`.

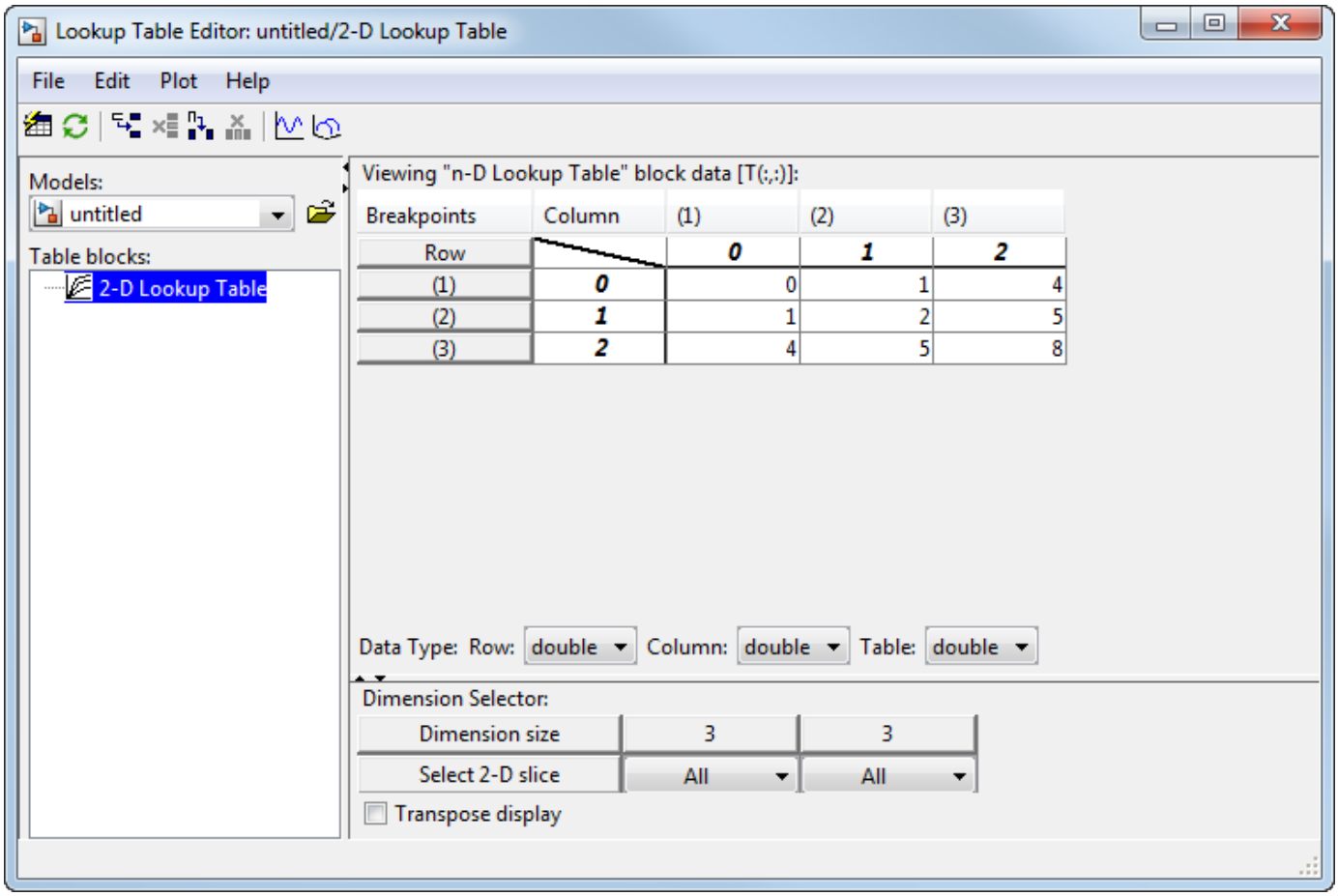
Entering Data in the Lookup Table Editor

Use the following procedure to populate a 2-D Lookup Table block using the Lookup Table Editor. In this example, the lookup table approximates the function $z = x^2 + y^2$ over the input ranges `x = [0, 2]` and `y = [0, 2]`.

1. Copy a 2-D Lookup Table block from the Lookup Tables block library to a Simulink model.
2. Open the Lookup Table Editor by selecting **Lookup Table Editor** from the **Modeling** tab or by clicking **Edit table and breakpoints** on the dialog box of the 2-D Lookup Table block.
3. Under **Viewing "n-D Lookup Table" block data**, enter the breakpoint data sets and table data in the appropriate cells. To change data, click a cell, enter the new value, and press **Enter**.

   - In the cells associated with the **Row Breakpoints**, enter each of the values `[0 1 2]`.
   - In the cells associated with the **Column Breakpoints**, enter each of the values `[0 1 2]`.
   - In the table data cells, enter the values in the array `[0 1 4; 1 2 5; 4 5 8]`.

   The Lookup Table Editor looks like this:



4 In the Lookup Table Editor, select **File** > **Update Block Data** to update the data in the 2-D Lookup Table block.

5 Close the Lookup Table Editor.

Entering Data Using Inports of the Lookup Table Dynamic Block

Use the following procedure to populate a Lookup Table Dynamic block using that block's inports. In this example, the lookup table approximates the function $y = 3x^2$ over the range `[0, 10]`.

1 Copy a Lookup Table Dynamic block from the Lookup Tables block library to a Simulink model.

2 Copy the blocks needed to implement the equation $y = 3x^2$ to the Simulink model:

- One Constant block to define the input range, from the Sources library
- One Math Function block to square the input range, from the Math Operations library
- One Gain block to multiply the signal by 3, also from the Math Operations library

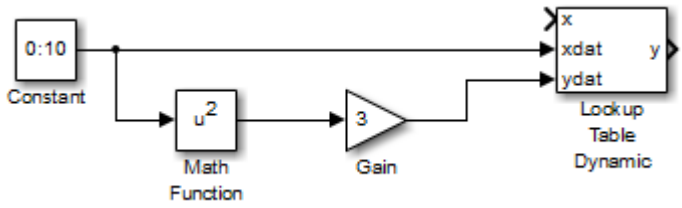
3 Assign the following parameter values to the Constant, Math Function, and Gain blocks using their dialog boxes:

Block	Parameter	Value
Constant	**Constant value**	`0:10`

Block	Parameter	Value
Math Function	**Function**	`square`
Gain	**Gain**	3

4 Input the breakpoint data set to the Lookup Table Dynamic block by connecting the output port of the Constant block to the input port of the Lookup Table Dynamic block labeled **xdat**. This signal is the input breakpoint data set for *x*.

5 Input the table data to the Lookup Table Dynamic block by branching the output signal from the Constant block and connecting it to the Math Function block. Then connect the Math Function block to the Gain block. Finally, connect the Gain block to the input port of the Lookup Table Dynamic block labeled **ydat**. This signal is the table data for *y*.



See Also

Lookup Table Dynamic | n-D Lookup Table

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7

Characteristics of Lookup Table Data

In this section...

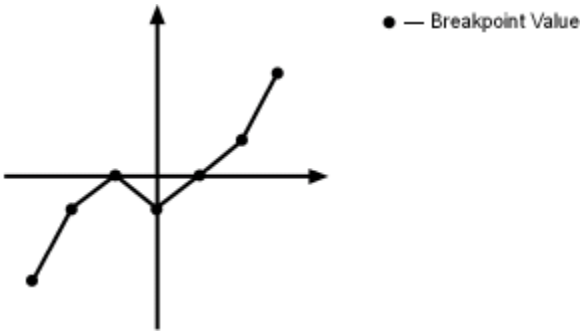
Sizes of Breakpoint Data Sets and Table Data

The following constraints apply to the sizes of breakpoint data sets and table data associated with lookup table blocks:

- The memory limitations of your system constrain the overall size of a lookup table.
- Lookup tables must use consistent dimensions so that the overall size of the table data reflects the size of each breakpoint data set.

To illustrate the second constraint, consider the following vectors of input and output values that create the relationship in the plot.

```
Vector of input values:   [-3 -2 -1  0 1 2 3]
Vector of output values:  [-3 -1  0 -1 0 1 3]
```



In this example, the input and output data are the same size (1-by-7), making the data consistently dimensioned for a 1-D lookup table.

The following input and output values define the 2-D lookup table that is graphically shown.

```
Row index input values:    [1 2 3]
Column index input values: [1 2 3 4]
Table data:                [11 12 13 14; 21 22 23 24; 31 32 33 34]
```

	1	2	3	4
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34

In this example, the sizes of the vectors representing the row and column indices are 1-by-3 and 1-by-4, respectively. Consequently, the output table must be of size 3-by-4 for consistent dimensions.

Monotonicity of Breakpoint Data Sets

The first stage of a table lookup operation involves relating inputs to the breakpoint data sets. The search algorithm requires that input breakpoint sets be strictly monotonically increasing, that is, each successive element is greater than its preceding element. For example, the vector

```
A = [0  0.5  1  1.9  2.1  3]
```

is a valid breakpoint data set as each element is larger than its predecessors.

Note Although a breakpoint data set is strictly monotonic in `double` format, it might not be so after conversion to a fixed-point data type.

Formulation of Evenly Spaced Breakpoints

You can represent evenly spaced breakpoints in a data set by using one of these methods.

Formulation	Example	When to Use This Formulation
`[first_value:spacing:last_value]`	`[10:10:200]`	The lookup table does *not* use `double` or `single`.
`first_value + spacing * [0:(last_value-first_value)/spacing]`	`1 + (0.02 * [0:450])`	The lookup table uses `double` or `single`.

Because floating-point data types cannot precisely represent some numbers, the second formulation works better for `double` and `single`. For example, use `1 + (0.02 * [0:450])` instead of `[1:0.02:10]`. For a list of lookup table blocks that support evenly spaced breakpoints, see “Summary of Lookup Table Block Features” on page 38-8.

Among other advantages, evenly spaced breakpoints can make the generated code division-free and reduce memory usage. For more information, see:

- `fixpt_evenspace_cleanup` in the Simulink documentation
- “Effects of Spacing on Speed, Error, and Memory Usage” (Fixed-Point Designer)
- “Identify questionable fixed-point operations” (Embedded Coder)

Tip Do not use the MATLAB `linspace` function to define evenly spaced breakpoints. Simulink uses a tighter tolerance to check whether a breakpoint set has even spacing. If you use `linspace` to define breakpoints for your lookup table, Simulink considers the breakpoints to be unevenly spaced.

See Also

More About

- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20

- “Guidelines for Choosing a Lookup Table” on page 38-7
- “Summary of Lookup Table Block Features” on page 38-8

Methods for Approximating Function Values

In this section...

About Approximating Function Values

The second stage of a table lookup operation involves generating outputs that correspond to the supplied inputs. If the inputs match the values of indices specified in breakpoint data sets, the block outputs the corresponding values. However, if the inputs fail to match index values in the breakpoint data sets, Simulink estimates the output. In the block parameter dialog box, you can specify how to compute the output in this situation. The available lookup methods are described in the following sections.

Interpolation Methods

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. Most lookup table blocks have the following interpolation methods available:

- `Flat` — Disables interpolation and uses the rounding operation titled `Use Input Below`. For more information, see “Rounding Methods” on page 38-18.
- `Nearest` — Disables interpolation and returns the table value corresponding to the breakpoint closest to the input. If the input is equidistant from two adjacent breakpoints, the breakpoint with the higher index is chosen.
- `Linear point-slope` — Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input. This is the equation for linear point-slope, where x is the input data, y is the output table data (x_i,y_i is the coordinate of the table data), and f is the fraction. For more information on x_i,y_i, see “About Lookup Table Blocks” on page 38-2.

  $$f = \frac{x - x_i}{x_{i+1} - x_i}$$

  $$y = y_i + f(y_{i+1} - y_i)$$
- `Cubic spline` — Fits a cubic spline to the adjacent breakpoints, and returns the point on that spline corresponding to the input.
- `Linear Lagrange` — Fits a line between the adjacent breakpoints using first-order Lagrange interpolation, and returns the point on that line corresponding to the input. This is the equation for linear Lagrange, where x is the input data, y is the output table data, and f is the fraction. f is constrained to range from 0 to less than 1 (`[0,1)`). For more information on x and y, see “About Lookup Table Blocks” on page 38-2.

  $$f = \frac{x - x_i}{x_{i+1} - x_i}$$

$$y = (1 - f)y_i + f y_{i+1}$$

If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

- `Akima spline` — Fits an Akima spline to the adjacent breakpoints, and returns the point on that spline corresponding to the input. The interpolation method works only with the `Akima spline` extrapolation method. The modified Akima cubic Hermite interpolation method has these properties:
  - It produces fewer undulations than Cubic spline.
  - It is more efficient for real-time applications than Cubic spline.
  - Unlike Cubic spline, it does not produce overshoots.
  - Unlike Cubic spline, it supports nonscalar signals.

Note The Lookup Table Dynamic block does not let you select an interpolation method. The `Interpolation-Extrapolation` option in the **Lookup Method** field of the block parameter dialog box performs linear interpolation.

Each interpolation method includes a trade-off between computation time and the smoothness of the result. Although rounding is quickest, it is the least smooth. Linear interpolation is slower than rounding but generates smoother results, except at breakpoints where the slope changes. Cubic spline interpolation is the slowest method but produces smooth results. Akima spline produces the smoothest results.

Extrapolation Methods

When an input falls outside the range of a breakpoint data set, the block extrapolates the output value from a pair of values at the end of the breakpoint data set. Most lookup table blocks have the following extrapolation methods available:

- `Clip` — Disables extrapolation and returns the table data corresponding to the end of the breakpoint data set range. This does not provide protection against out-of-range values.
- `Linear` — If the interpolation method is `Linear`, this extrapolation method fits a line between the first or last pair of breakpoints, depending on whether the input is less than the first or greater than the last breakpoint. If the interpolation method is `Cubic spline` or `Akima spline`, this extrapolation method fits a linear surface using the slope of the interpolation at the first or last break point, depending on whether the input is less than the first or greater than the last breakpoint. The extrapolation method returns the point on the generated linear surface corresponding to the input.

  If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.
- `Cubic spline` — Fits a cubic spline to the first or last pair of breakpoints, depending if the input is less than the first or greater than the last breakpoint, respectively. This method returns the point on that spline corresponding to the input.

- `Akima spline` — Fits an Akima spline to the first or last pair of breakpoints, depending if the input is less than the first or greater than the last breakpoint, respectively. This method returns the point on that spline corresponding to the input.

Note The Lookup Table Dynamic block does not let you select an extrapolation method. The `Interpolation-Extrapolation` option in the **Lookup Method** field of the block parameter dialog box performs linear extrapolation.

In addition to these methods, some lookup table blocks, such as the n-D Lookup Table block, allow you to select an action to perform when encountering situations that require extrapolation. For instance, you can specify that Simulink generate either a warning or an error when the lookup table inputs are outside the ranges of the breakpoint data sets. To specify such an action, select it from the **Diagnostic for out-of-range input** list on the block parameter dialog box.

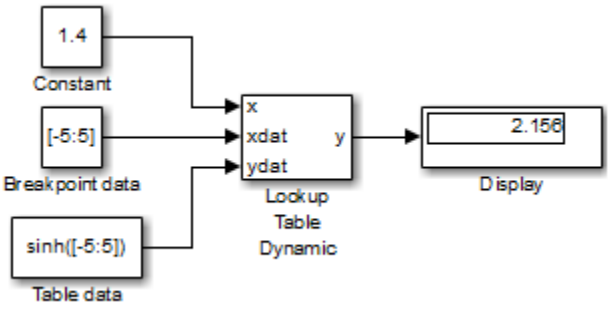
Rounding Methods

If an input falls between breakpoint values or outside the range of a breakpoint data set and you do not specify interpolation or extrapolation, the block rounds the value to an adjacent breakpoint and returns the corresponding output value. For example, the Lookup Table Dynamic block lets you select one of the following rounding methods:

- `Use Input Nearest` — Returns the output value corresponding to the nearest input value.
- `Use Input Below` — Returns the output value corresponding to the breakpoint value that is immediately less than the input value. If no breakpoint value exists below the input value, it returns the breakpoint value nearest the input value.
- `Use Input Above` — Returns the output value corresponding to the breakpoint value that is immediately greater than the input value. If no breakpoint value exists above the input value, it returns the breakpoint value nearest the input value.

Example Output for Lookup Methods

In the following model, the Lookup Table Dynamic block accepts a vector of breakpoint data given by `[-5:5]` and a vector of table data given by `sinh([-5:5])`.



The Lookup Table Dynamic block outputs the following values when using the specified lookup methods and inputs.

Lookup Method	Input	Output	Comment
`Interpolation-Extrapolation`	`1.4`	`2.156`	N/A
	`5.2`	`83.59`	N/A
`Interpolation-Use End Values`	`1.4`	`2.156`	N/A
	`5.2`	`74.2`	The block uses the value for `sinh(5.0)`.
`Use Input Above`	`1.4`	`3.627`	The block uses the value for `sinh(2.0)`.
	`5.2`	`74.2`	The block uses the value for `sinh(5.0)`.
`Use Input Below`	`1.4`	`1.175`	The block uses the value for `sinh(1.0)`.
	`-5.2`	`-74.2`	The block uses the value for `sinh(-5.0)`.
`Use Input Nearest`	`1.4`	`1.175`	The block uses the value for `sinh(1.0)`.

See Also

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7

Edit Lookup Tables

In this section...

You can edit a lookup table using:

- Lookup Table block dialog box
- Lookup Table Editor

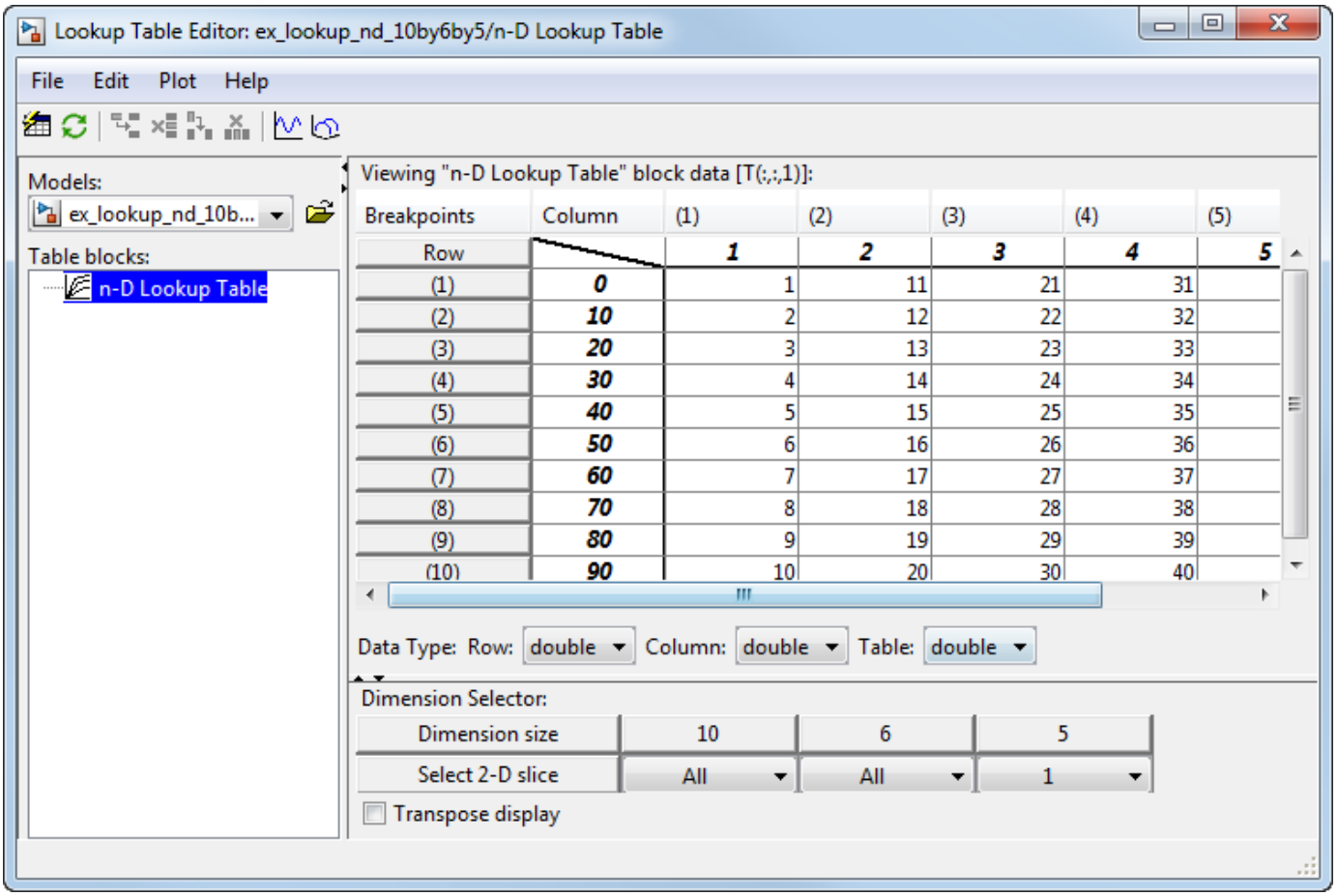
To edit the lookup table in a block:

1. Open the subsystem that contains the lookup table block.
2. Open the lookup table block’s dialog box.
3. In the Table and Breakpoints tab, edit the **Table data** and relevant **Breakpoints** parameters as needed.

With the Lookup Table Editor, you can skip these steps and edit the desired lookup table without navigating to the block that uses it. However, you cannot use the Lookup Table Editor to change the dimensions of a lookup table. You must use the block parameter dialog box for this purpose.

Edit N-Dimensional Lookup Tables

If the lookup table of the block currently selected in the Lookup Table Editor tree view has more than two dimensions, the table view displays a two-dimensional slice of the lookup table.



The **Dimension Selector** specifies which slice currently appears and lets you select another slice. The selector consists of a 2-by-`N` array of controls, where `N` is the number of dimensions in the lookup table. Each column corresponds to a dimension of the lookup table. The first column corresponds to the first dimension of the table, the second column to the second dimension of the table, and so on. The **Dimension size** row of the selector array displays the size of each dimension. The **Select 2-D slice** row specifies which dimensions of the table correspond to the row and column axes of the slice and the indices that select the slice from the remaining dimensions.

To select another slice of the table, specify the row and column axes of the slice in the first two columns of the **Select 2-D slice** row. Then select the indices of the slice from the pop-up index lists in the remaining columns.

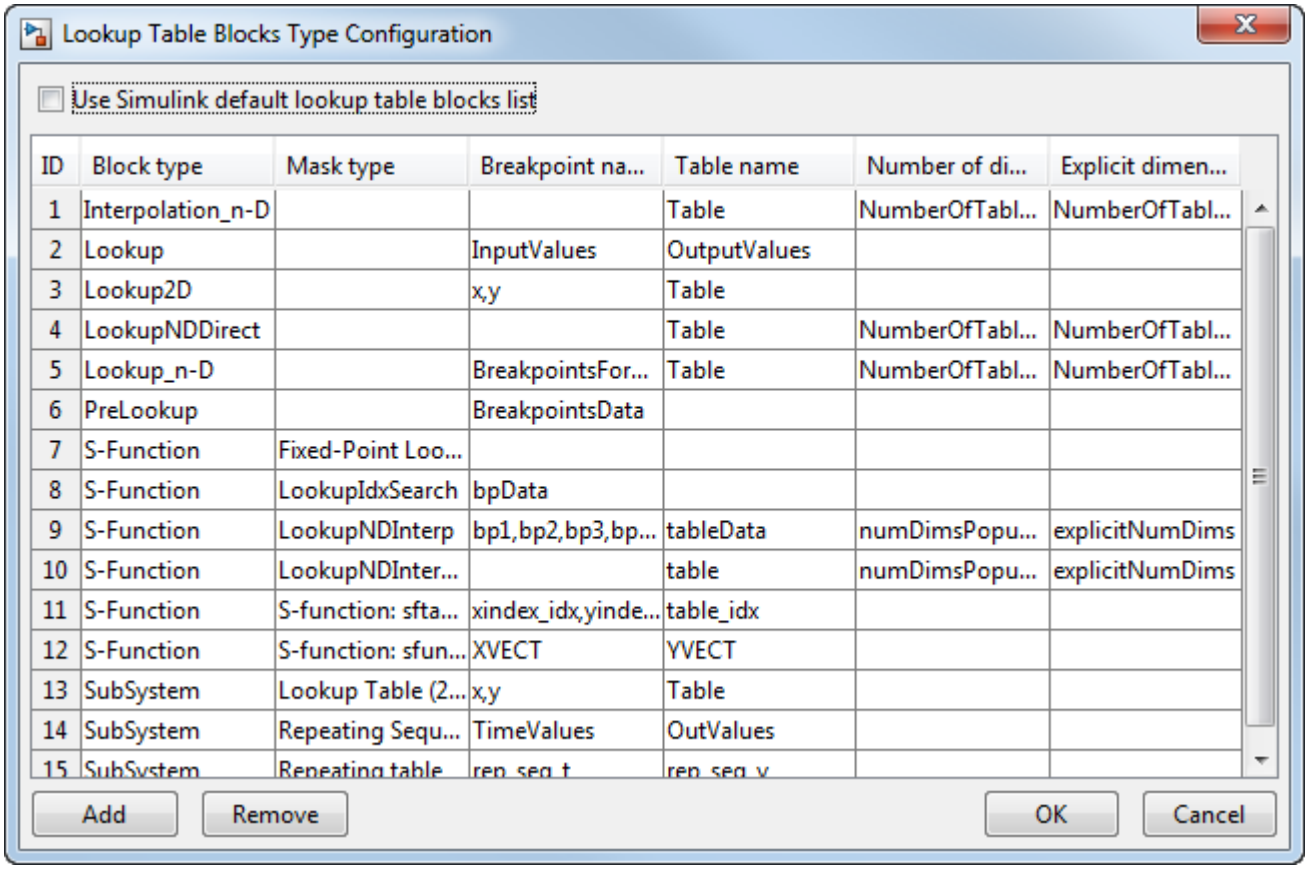
For example, the following selector displays slice (:,:,1) of a 3-D lookup table, as shown under Dimension Selector in the Lookup Table Editor.

To transpose the table display, select the **Transpose display** check box.

Edit Custom Lookup Table Blocks

You can use the Lookup Table Editor to edit custom lookup table blocks that you have created. To do this, you must first configure the Lookup Table Editor to recognize the custom lookup table blocks in your model.

1 Select **File** > **Configure**. The Lookup Table Blocks Type Configuration dialog box appears.



The dialog box displays a table of the lookup table block types that the Lookup Table Editor currently recognizes. This table includes the standard blocks. Each row of the table displays key attributes of a lookup table block type.

2 Click **Add** on the dialog box. A new row appears at the bottom of the block type table.

3 Enter information for the custom block in the new row under these headings.

Field Name	Description
Block type	Block type of the custom block. The block type is the value of the block's `BlockType` parameter.
Mask type	Mask type of the custom block. The mask type is the value of the block's `MaskType` parameter.
Breakpoint name	Names of the block parameters that store the breakpoints.
Table name	Name of the block parameter that stores the table data.
Number of dimensions	Leave empty.
Explicit dimensions	Leave empty.

4 Click **OK**.

To remove a custom lookup table block type from the list that the Lookup Table Editor recognizes, select the custom entry in the table of the Lookup Table Blocks Type Configuration dialog box and click **Remove**. To remove all custom lookup table block types, select the **Use Simulink default lookup table blocks list** check box at the top of the dialog box.

See Also

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Guidelines for Choosing a Lookup Table” on page 38-7

Import Lookup Table Data from MATLAB

In this section...

You can import table and breakpoint data from variables in the MATLAB workspace by referencing them in the **Table and Breakpoints** tab of the dialog box. The following examples show how to import and export standard format and non-standard format data from the MATLAB workspace.

Import Standard Format Lookup Table Data

Suppose you specify a 3-D lookup table in your n-D Lookup Table block.

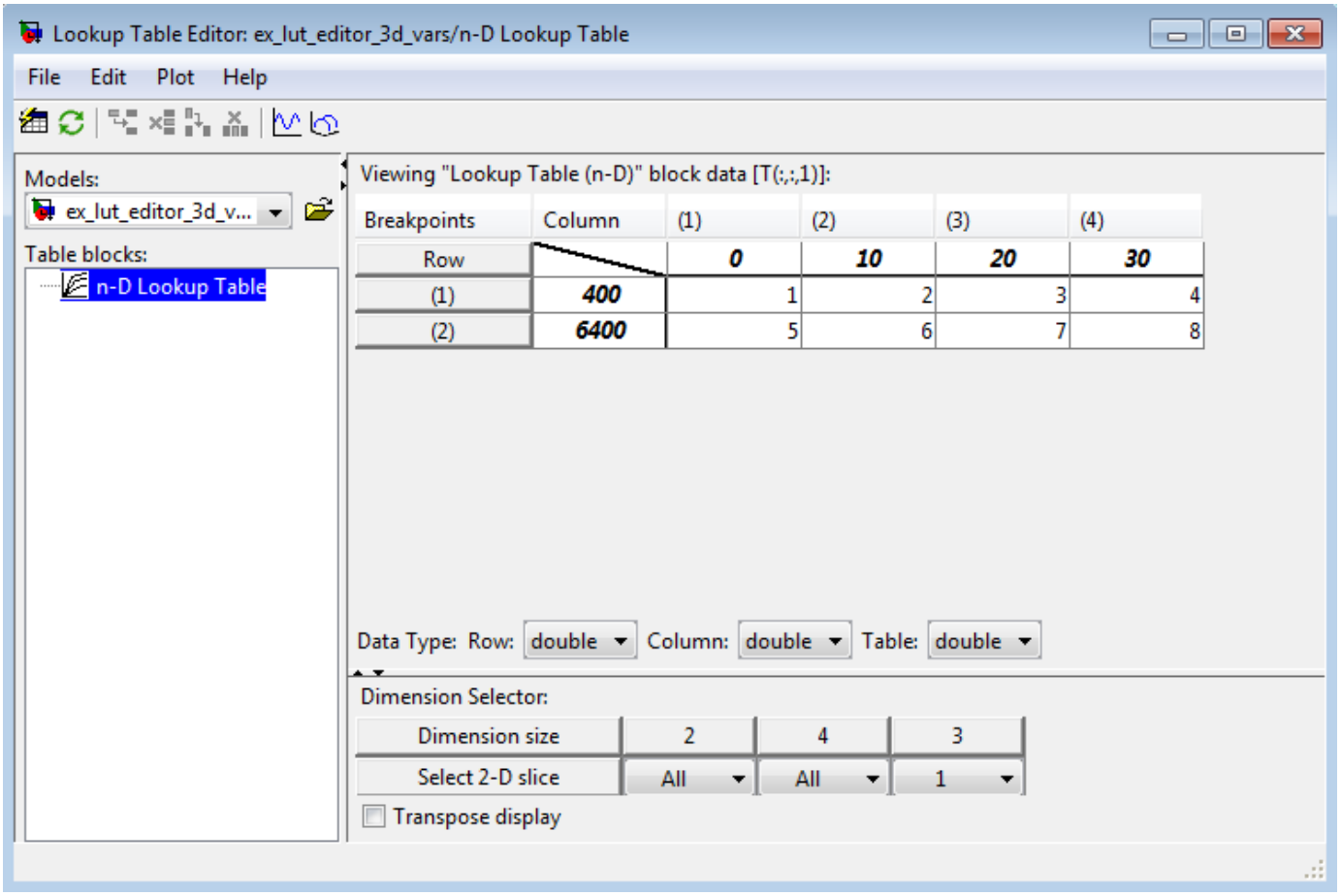
Create workspace variables to use as breakpoint and table data for the lookup table.

```
table3d_map = zeros(2,4,3);
table3d_map(:,:,1) = [     1     2     3     4;      5     6     7     8];
table3d_map(:,:,2) = [    11    12    13    14;     15    16    17    18];
table3d_map(:,:,3) = [   111   112   113   114;   115   116   117   118];
bp3d_z =[  0    10    20];
bp3d_x =[     0    10    20    30];
bp3d_y =[   400  6400];
```

Open the n-D Lookup Table block dialog box, and enter the following parameters in the Table and Breakpoints tab:

- Table data: table3d_map
- Breakpoints 1: bp3d_y
- Breakpoints 2: bp3d_x
- Breakpoints 3: bp3d_z

Click **Edit table and breakpoints** to open the Lookup Table Editor and show the data from the workspace variables.



Propagate Standard Format Lookup Table Data

When you make changes to your lookup table data, consider propagating the changes back to the MATLAB workspace variables the data was imported from using **File > Update Block Data**.

You can also use the Lookup Table Editor to edit the table data and breakpoint data set of `Simulink.LookupTable` and the breakpoint data set of `Simulink.Breakpoint` objects and propagate the changes back to the object.

Suppose you make a change to the lookup table variables imported from the MATLAB workspace variables in “Import Standard Format Lookup Table Data” on page 38-24. For example, change the value of the data in (1,1,1) from `1` to `33`. To propagate this change back to `table3d_map` in the workspace, select **File > Update Block Data**. Click **Yes** to confirm that you want to overwrite `table3d_map`.

Import Nonstandard Format Lookup Table Data

Suppose you specify a 3-D lookup table in your n-D Lookup Table block. Create workspace variables to use as breakpoint and table data for the lookup table. The variable for table data, `table3d_map_custom`, is a two-dimensional matrix.

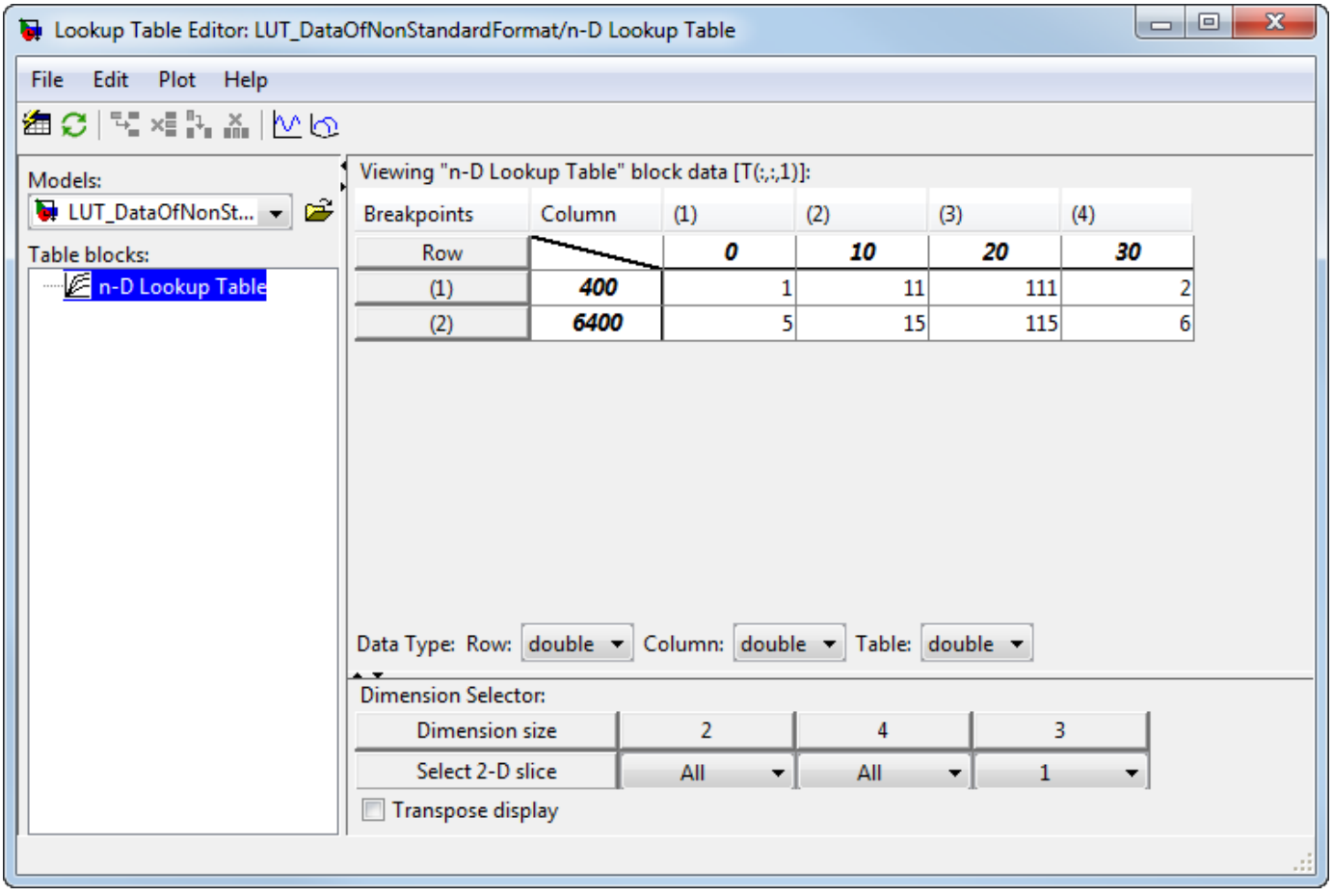
```
table3d_map_custom = zeros(6,4);
table3d_map_custom = [     1     2     3     4;      5     6     7     8;
```

```
11      12     13      14;        15      16      17     18;
111   112    113    114;     115    116    117    118];
bp3d_z =[  0    10    20];
bp3d_x =[  0    10    20    30];
bp3d_y =[  400  6400];
```

Open the n-D Lookup Table block dialog box, and enter the following parameters in the Table and Breakpoints tab. Transform `table3d_map_custom` into a three-dimensional matrix for the table data input using the `reshape` command.

- Table data: `reshape(table3d_map_custom,[2,4,3])`
- Breakpoints 1: `bp3d_y`
- Breakpoints 2: `bp3d_x`
- Breakpoints 3: `bp3d_z`

Click **Edit table and breakpoints** to open the Lookup Table Editor and show the data from the workspace variables.



Change 1 to 33 in the Lookup Table Editor. The Lookup Table Editor records your changes by maintaining a copy of the table. To restore the variable values from the MATLAB workspace, select **File > Reload Block Data**. To update the MATLAB workspace variables with the edited data, select **File > Update Block Data** in the Lookup Table Editor. You cannot propagate the change to `table3d_map_custom`, the workspace variable that contains the nonstandard table data for the n-D

Lookup Table block. To propagate the change, you must register a customization function that resides on the MATLAB search path. For details, see “Propagate Nonstandard Format Lookup Table Data” on page 38-27.

Propagate Nonstandard Format Lookup Table Data

This example shows how to propagate changes from the Lookup Table Editor to workspace variables of nonstandard format. Suppose your Simulink model from “Import Nonstandard Format Lookup Table Data” on page 38-25 has a three-dimensional lookup table that gets its table data from the two-dimensional workspace variable `table3d_map_custom`. Update the lookup table in the Lookup Table Editor and propagate these changes back to `table3d_map_custom` using a customization function.

1 Create a file named `sl_customization.m` with these contents.

```
function sl_customization(cm)
cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle{end+1} = ...
@myGetTableConvertInfoFcn;
end
```

In this function:

- The argument `cm` is the handle to a customization manager object.
- The handle `@myGetTableConvertInfoFcn` is added to the list of function handles in the cell array for `cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle`. You can use any alphanumeric name for the function whose handle you add to the cell array.

2 In the same file, define the `myGetTableConvertInfoFcn` function.

```
function blkInfo = myGetTableConvertInfoFcn(blk,tableStr)
        blkInfo.allowTableConvertLocal = true;
        blkInfo.tableWorkSpaceVarName = 'table3d_map_custom';
        blkInfo.tableConvertFcnHandle = @myConvertTableFcn;
end
```

The `myGetTableConvertInfoFcn` function returns the `blkInfo` object containing three fields.

- `allowTableConvertLocal` — Allows table data conversion for a block.
- `tableWorkSpaceVarName` — Specifies the name of the workspace variable that has a nonstandard table format.
- `tableConvertFcnHandle` — Specifies the handle for the conversion function.

When `allowTableConvertLocal` is set to `true`, the table data for that block is converted to the nonstandard format of the workspace variable whose name matches `tableWorkSpaceVarName`. The conversion function corresponds to the handle that `tableConvertFcnHandle` specifies. You can use any alphanumeric name for the conversion function.

3 In the same file, define the `myConvertTableFcn` function. This function converts a three-dimensional lookup table of size *Rows* * *Columns* * *Height* to a two-dimensional variable of size *(Rows*Height)* * *Columns*.

```
% Converts 3-dimensional lookup table from Simulink format to
% nonstandard format used in workspace variable
function cMap = myConvertTableFcn(data)

% Determine the row and column number of the 3D table data
    mapDim = size(data);
    numCol = mapDim(2);
```

```
    numRow = mapDim(1)*mapDim(3);
    cMap = zeros(numRow, numCol);
   % Transform data back to a 2-dimensional matrix
    cMap = reshape(data,[numRow,numCol]);
end
```

4. Put `sl_customization.m` on the MATLAB search path. You can have multiple files named `sl_customization.m` on the search path. For more details, see “Behavior with Multiple Customization Functions” on page 38-28.
5. Refresh Simulink customizations at the MATLAB command prompt.

   ```
   sl_refresh_customizations
   ```
6. Open the Lookup Table Editor for your lookup table block and select **File > Update Block Data**. Click **Yes** to overwrite the workspace variable `table3d_map_custom`.
7. Check the value of `table3d_map_custom` in the base workspace.

   ```
   table3d_map_custom =

       33     2     3     4
        5     6     7     8
       11    12    13    14
       15    16    17    18
      111   112   113   114
      115   116   117   118
   ```

   The change in the Lookup Table Editor has propagated to the workspace variable.

Note If you do not overwrite the workspace variable `table3d_map_custom`, you are prompted to replace it with numeric data. Click **Yes** to replace the expression in the **Table data** field with numeric data. Click **No** if you do not want your Lookup Table Editor changes for the table data to appear in the block dialog box.

Behavior with Multiple Customization Functions

At the start of a MATLAB session, Simulink loads each `sl_customization.m` customization file on the path and executes the `sl_customization` function. Executing each function establishes the customizations for that session.

When you select **File > Update Block Data** in the Lookup Table Editor, the editor checks the list of function handles in the cell array for `cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle`. If the cell array contains one or more function handles, the `allowTableConvertLocal` property determines whether changes in the Lookup Table Editor can be propagated.

- If the value is set to `true`, then the table data is converted to the nonstandard format in the workspace variable.
- If the value is set to `false`, then table data is not converted to the nonstandard format in the workspace variable.
- If the value is set to `true` and another customization function specifies it to be `false`, the Lookup Table Editor reports an error.

See Also

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7
- “Import Lookup Table Data from Excel” on page 38-30

Import Lookup Table Data from Excel

This example shows how to use the MATLAB `xlsread` function in a Simulink model to import data into a lookup table.

1. Save the Excel file in a folder on the MATLAB path.
2. Open the model containing the lookup table block and in the **Modeling** tab, select **Model Settings**.
3. In the Model Properties dialog box, in the **Callbacks** tab, click **PostLoadFcn** callback in the model callbacks list.
4. Enter the code to import the Excel Spreadsheet data in the text box. Use the MATLAB `xlsread` function, as shown in this example for a 2-D lookup table.

```
% Import the data from Excel for a lookup table
data = xlsread('MySpreadsheet','Sheet1');
% Row indices for lookup table
breakpoints1 = data(2:end,1)';
% Column indices for lookup table
breakpoints2 = data(1,2:end);
% Output values for lookup table
table_data = data(2:end,2:end);
```

5. Click **OK**.

After you save your changes, the next time you open the model, Simulink invokes the callback and imports the data.

See Also

More About

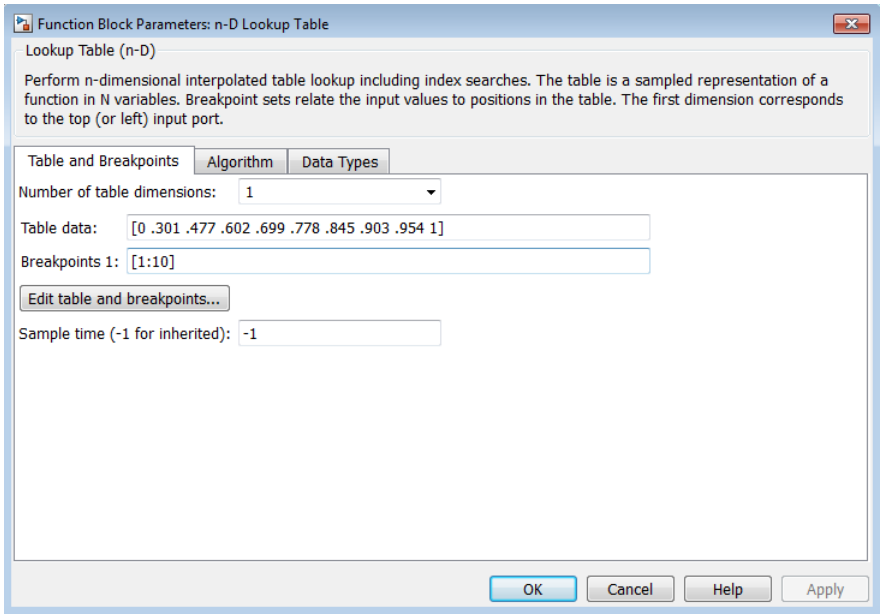
- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7
- “Import Lookup Table Data from MATLAB” on page 38-24

Create a Logarithm Lookup Table

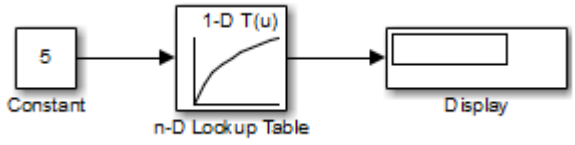
Suppose you want to approximate the common logarithm (base 10) over the input range [1, 10] without performing an expensive computation. You can perform this approximation using a lookup table block as described in the following procedure.

1. Copy the following blocks to a Simulink model:
   - One Constant block to input the signal, from the Sources library
   - One n-D Lookup Table block to approximate the common logarithm, from the Lookup Tables library
   - One Display block to display the output, from the Sinks library
2. Assign the table data and breakpoint data set to the n-D Lookup Table block:
   a. In the **Number of table dimensions** field, enter `1`.
   b. In the **Table data** field, enter `[0 .301 .477 .602 .699 .778 .845 .903 .954 1]`.
   c. In the **Breakpoints 1** field, enter `[1:10]`.
   d. Click **Apply**.

   The dialog box looks something like this:



3. Double-click the Constant block to open the parameter dialog box, and change the **Constant value** parameter to `5`. Click **OK** to apply the changes and close the dialog box.
4. Connect the blocks as follows.



5 Start simulation.

The following behavior applies to the n-D Lookup Table block.

Value of the Constant Block	Action by the n-D Lookup Table Block	Example of Block Behavior	
		Input Value	Output Value
Equals a breakpoint	Returns the corresponding output value	5	0.699
Falls between breakpoints	Linearly interpolates the output value using neighboring breakpoints	7.5	0.874
Falls outside the range of the breakpoint data set	Linearly extrapolates the output value from a pair of values at the end of the breakpoint data set	10.5	1.023

For the n-D Lookup Table block, the default settings for **Interpolation method** and **Extrapolation method** are both `Linear`.

See Also

n-D Lookup Table

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7
- “Import Lookup Table Data from MATLAB” on page 38-24

Prelookup and Interpolation Blocks

The following examples show the benefits of using Prelookup and Interpolation Using Prelookup blocks.

Action	Benefit	Example
Use an index search to relate inputs to table data, followed by an interpolation and extrapolation stage that computes outputs	Enables reuse of index search results to look up data in multiple tables, which reduces simulation time	For more information, see “Using the Prelookup and Interpolation Blocks”.
Set breakpoint and table data types explicitly	Lowers memory required to store: • Breakpoint data that uses a smaller type than the input signal • Table data that uses a smaller type than the output signal	For more information, see “Saving Memory in Prelookup and Interpolation Blocks by Using Smaller Data”.
	Provides easier sharing of: • Breakpoint data among Prelookup blocks • Table data among Interpolation Using Prelookup blocks	For more information, see “Sharing Parameters in Prelookup and Interpolation Blocks” (Fixed-Point Designer).
	Enables reuse of utility functions in the generated code	For more information, see “Shared Utility Functions for Prelookup Blocks” (Fixed-Point Designer).
Set the data type for intermediate results explicitly	Enables use of higher precision for internal computations than for table data or output data	For more information, see “High Precision Calculations in Interpolation Block” (Fixed-Point Designer).

See Also

Interpolation Using Prelookup | Prelookup

More About

- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7

Optimize Generated Code for Lookup Table Blocks

In this section...

Remove Code That Checks for Out-of-Range Inputs

By default, generated code for the following lookup table blocks include conditional statements that check for out-of-range breakpoint or index inputs:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup
- Interpolation Using Prelookup

To generate code that is more efficient, you can remove the conditional statements that protect against out-of-range input values.

Block	Check Box to Select
1-D Lookup Table	**Remove protection against out-of-range input in generated code**
2-D Lookup Table	
n-D Lookup Table	
Prelookup	
Interpolation Using Prelookup	**Remove protection against out-of-range index in generated code**

Selecting the check box on the block dialog box improves code efficiency because there are fewer statements to execute. However, if you are generating code for safety-critical applications, you should not remove the range-checking code.

To verify the usage of the check box, run the following Model Advisor checks and perform the recommended actions.

Model Advisor Check	When to Run the Check
By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code	For code efficiency
By Product >Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks	For safety-critical applications

For more information about the Model Advisor, see “Check Your Model Using the Model Advisor” on page 5-2 in the Simulink documentation.

Optimize Breakpoint Spacing in Lookup Tables

When breakpoints in a lookup table are tunable, the spacing does not affect efficiency or memory usage of the generated code. When breakpoints are *not* tunable, the type of spacing can affect the following factors.

Factor	Even Power of 2 Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
Execution speed	The execution speed is the fastest. The position search and interpolation are the same as for evenly-spaced data. However, to increase speed a bit more for fixed-point types, a bit shift replaces the position search, and a bit mask replaces the interpolation.	The execution speed is faster than that for unevenly-spaced data because the position search is faster and the interpolation uses a simple division.	The execution speed is the slowest of the different spacings because the position search is slower, and the interpolation requires more operations.
Error	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be smaller because approximating a function with nonuniform curvature requires fewer points to achieve the same accuracy.
ROM usage	Uses less command ROM, but more data ROM.	Uses less command ROM, but more data ROM.	Uses more command ROM, but less data ROM.
RAM usage	Not significant.	Not significant.	Not significant.

Follow these guidelines:

- For fixed-point data types, use breakpoints with even, power-of-2 spacing.
- For non-fixed-point data types, use breakpoints with even spacing.

To identify opportunities for improving code efficiency in lookup table blocks, run the following Model Advisor checks and perform the recommended actions:

- **By Product > Embedded Coder > Identify questionable fixed-point operations**
- **By Product > Embedded Coder > Identify blocks that generate expensive saturation and rounding code**

For more information about the Model Advisor, see “Check Your Model Using the Model Advisor” on page 5-2 in the Simulink documentation.

Reduce Data Copies for Lookup Table Blocks

When you use workspace variables to store table and breakpoint data for Lookup Table blocks, and then configure these variables for tunability, you can avoid data copies by using the same data type for the block parameter and variable. Workspace variables include numeric MATLAB variables and `Simulink.Parameter` objects that you store in a workspace, such as the base workspace, or in a

data dictionary. If the data type of the variable is smaller than the data type of the block parameter, the generated code implicitly casts the data type of the variable to the data type of the block parameter. This implicit cast requires a data copy which can potentially significantly increase RAM consumption and slow down code execution speed for large vectors or matrices.

For more information, see “Parameter Data Types in the Generated Code” (Embedded Coder) and “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

Efficient Code for Row-Major Array Layout

To generate efficient code for row-major array layout, select the model configuration parameter **Math and Data Types > Use algorithms optimized for row-major array layout**. The row-major algorithms perform with the best speed and memory usage when operating on table data with row-major array layout. Similarly, the default column-major algorithms perform best with column-major array layout. Consider using the algorithm that is optimized for the specified array layout to achieve best performance. For example, use row-major algorithms when array layout is set as row-major during code generation.

Array Layout	Algorithm	Cache-Friendly Algorithm
Column-major	Column-major	Recommended
Row-major	Row-major	Recommended
Row-major	Column-major	Not recommended
Column-major	Row-major	Not recommended

For more information, see “Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder).

See Also

Interpolation Using Prelookup | Prelookup | n-D Lookup Table

More About

- “Check Your Model Using the Model Advisor” on page 5-2
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7
- “Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder)

Row-Major Algorithm in Existing Models Containing Lookup Table Blocks

The Direct Lookup Table (n-D), Interpolation Using Prelookup, and n-D Lookup Table blocks have algorithms that work with row-major array layouts or column-major array layouts. This capability requires a Simulink Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

Prior to R2018b, lookup table blocks supported only column-major array layouts. When selecting algorithms optimized for row-major array layout for a model previously configured for algorithms optimized for column-major array layout, you may need to preserve lookup table block semantics. For example, if a model contains lookup table blocks configured like these:

- An Interpolation Using Prelookup block configured with a subtable selection before interpolation.
- A Direct Lookup Table (n-D) configured for a vector or 2-D matrix output

Use the `permute` function to rearrange the dimensions of the array and preserve the semantics.

For an example of preserving semantics by using table permutations, see “Direct Lookup Table Algorithm for Row-Major Array Layout” (Simulink Coder) .

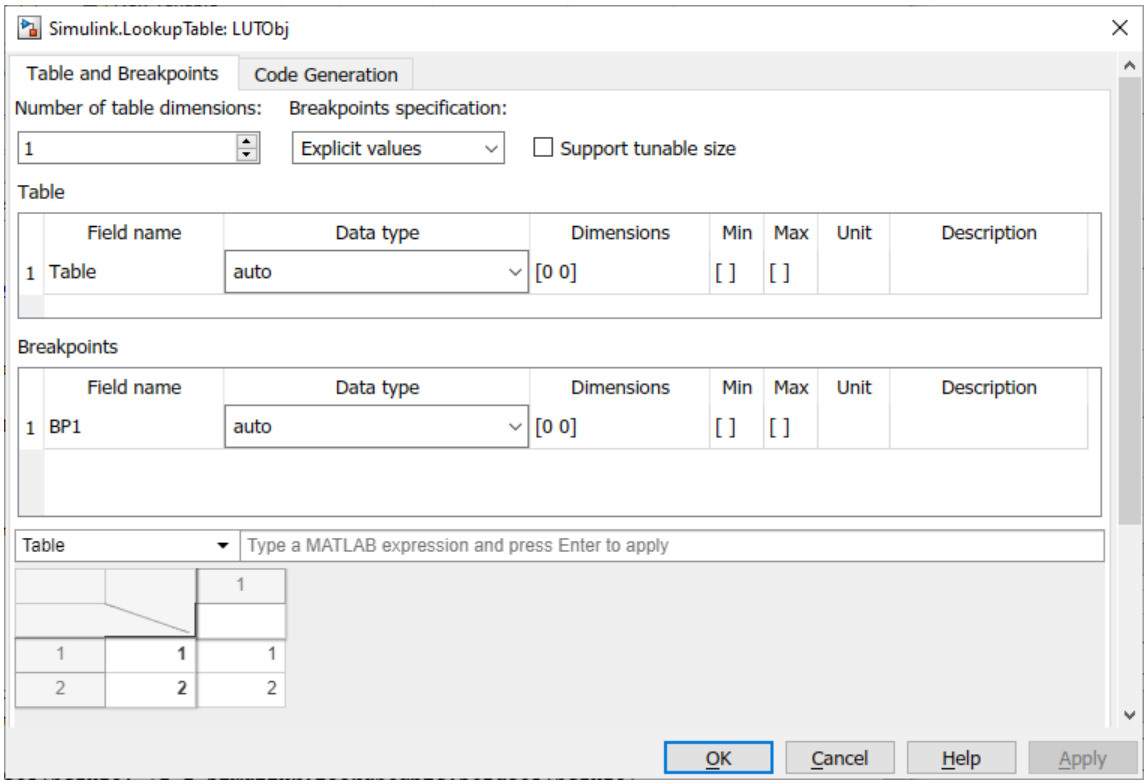
For an example of converting column-major arrays to row-major arrays see “Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder).

See Also

Direct Lookup Table (n-D) | Interpolation Using Prelookup | n-D Lookup Table

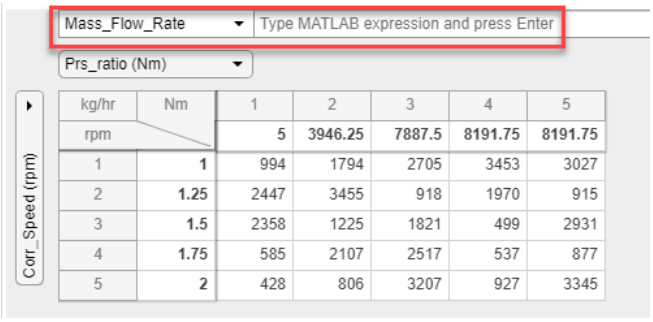
View Simulink.LookupTable Object Data Using the Property Dialog Box Tabular Interface

You can use the `Simulink.LookupTable` property dialog box tabular interface to view to view lookup table and breakpoint data .

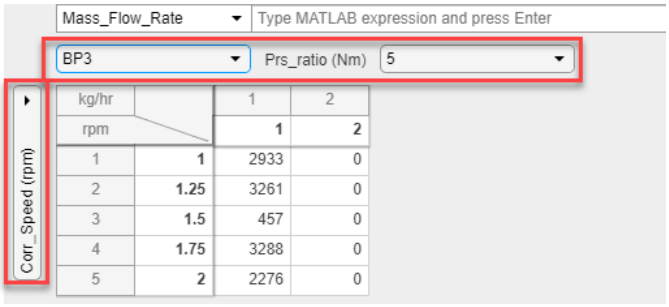


When the **Breakpoints specification** property is set to `Explicit values`, use the tabular interface and MATLAB expression text box at the bottom of the property dialog box to:

- View, add, manipulate, and remove table and breakpoint data.
- Create or modify table or breakpoint data using MATLAB expressions.



- View and edit 2-D slices of data from multiple dimensions.



This topic describes:

- Supported data types for the `Simulink.LookupTable` object property dialog box tabular Interface
- How to create, edit, and view table and breakpoint data in the tabular interface
- How to view multidimensional 2-D slices
- How the tabular interface handles data overflows
- How the tabular interface handles invalid data
- Supported keyboard shortcuts

Simulink.LookupTable Object Property Dialog Box Data Type Support

The `Simulink.LookupTable` object property dialog box supports these data types:

- Built-in data types (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `double`, `single`, `boolean`)
- Fixed-point data types
- `uint64`, `int64`

It does not support other data types that Simulink supports.

Create Simulink.LookupTable Objects

To start working with a `Simulink.LookupTable` object, create one at the MATLAB command line:

```
LUTObj = Simulink.LookupTable;
```

How to Open the Simulink.LookupTable Object Property Dialog Box

To work with the `Simulink.LookupTable` object, start its property dialog box in one of these ways from the MATLAB Command Window:

- In the workspace, double-click the `Simulink.LookupTable` object.
- In the Model Explorer, double-click the object in the specific workspace for the object.
- In the command line, use the `open` function, for example:

  ```
  open LUTObj
  ```

Create Table and Breakpoint Data

Generate data and manipulate it using standard table editing actions. You can generate lookup table data in multiple ways, such as in the workspace, from Microsoft Excel, and so forth, and copy that data into the tabular area. You can also generate data from within the property dialog box using MATLAB expressions.

This example describes how to create data in the MATLAB Command Window workspace and how to set up the property dialog box using an example with that data.

1 To create table and breakpoint data, at the MATLAB command line, type:

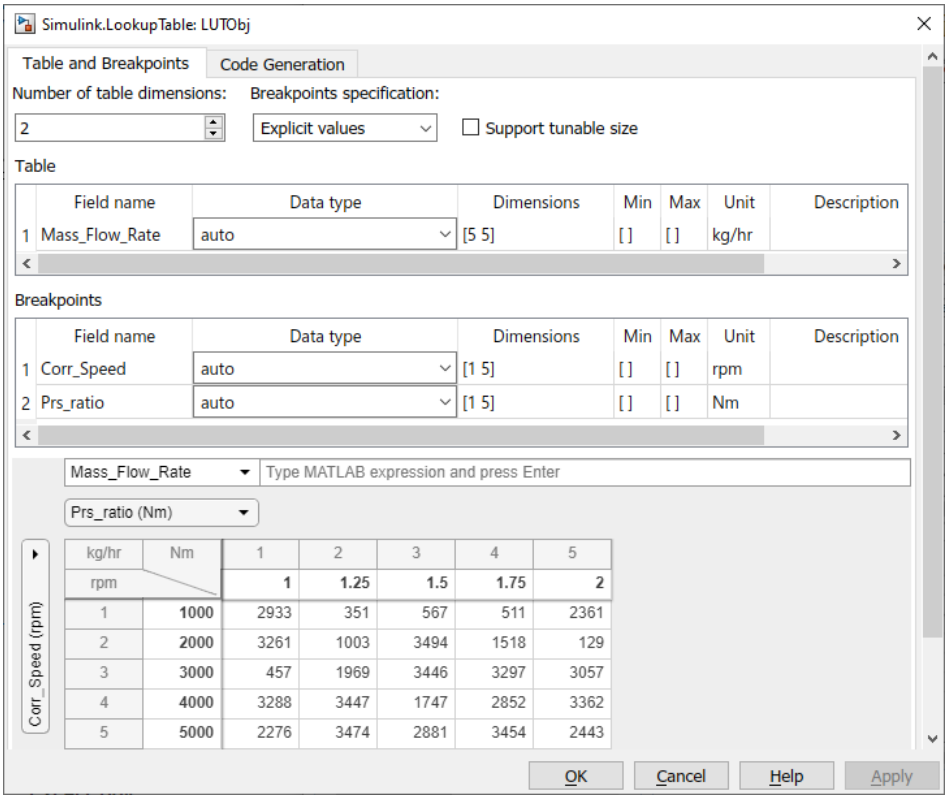
```
LUTObj.Breakpoints(1).Value = fi(linspace(1000,5000,5),1,16,2);
LUTObj.Breakpoints(2).Value = single(linspace(1,2,5));
LUTObj.Table.Value = uint16(rand(5,5)*60*60);
```

2 Set up the **Unit** and **Field name** properties for the object.

```
LUTObj.Breakpoints(1).Unit = 'rpm';
LUTObj.Breakpoints(2).Unit = 'Nm';
LUTObj.Table.Unit = 'kg/hr';
LUTObj.Breakpoints(1).FieldName = 'Corr_Speed';
LUTObj.Breakpoints(2).FieldName = 'Prs_ratio';
LUTObj.Table.FieldName = 'Mass_Flow_Rate';
```

3 Open the property dialog box.

```
open LUTObj;
```



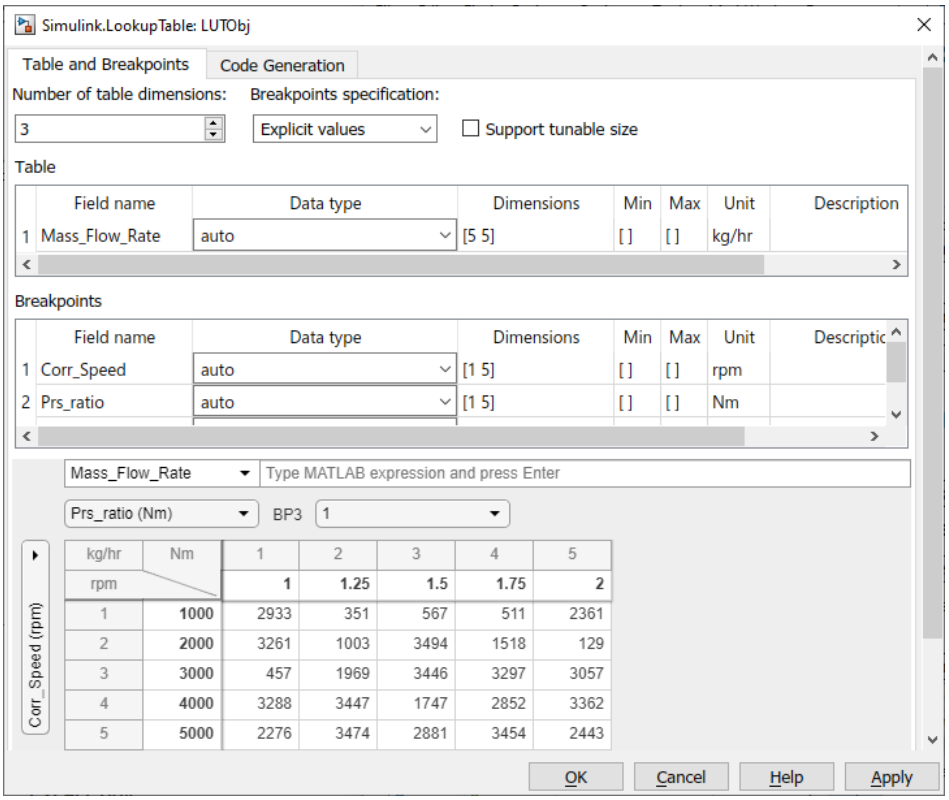
4 From within the tabular area, you can perform typical table edits on particular breakpoint data, such as copying and pasting. When you are done, click **Apply**.

View Multidimensional Slices of Data

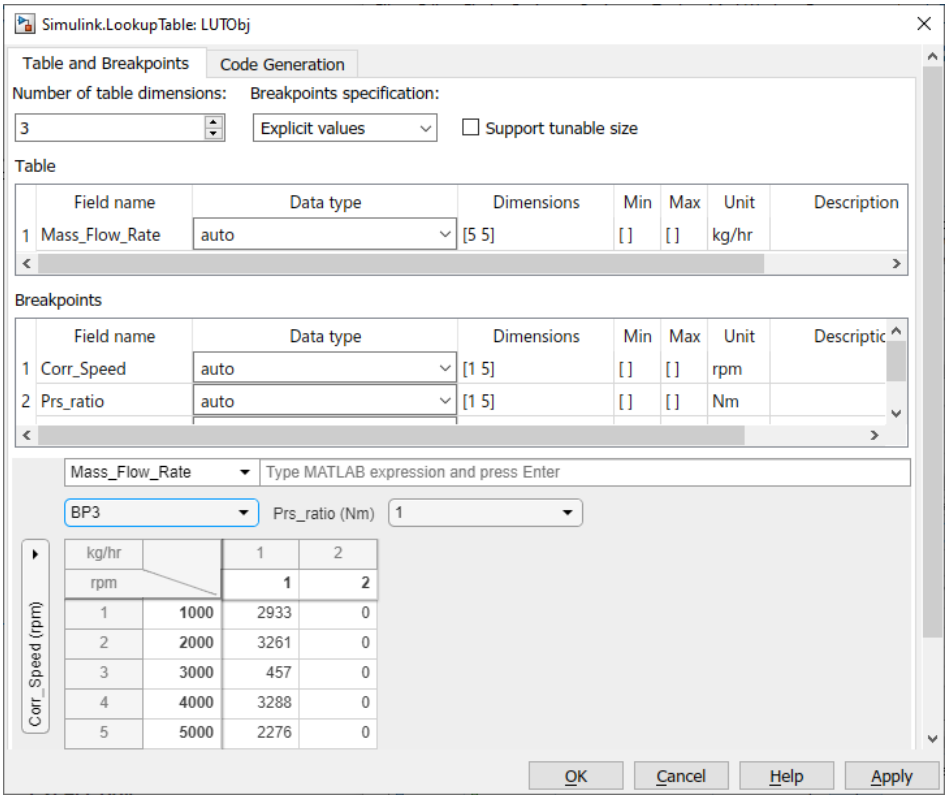
The Lookup Table property dialog box lets you view and edit 2-D slices of data. This example describes how to change the number of dimensions for the example of a multidimensional slice view.

To view a multidimensional slice, use the dropdown lists under the MATLAB expression parameter. There is a dropdown list for each breakpoint. To view a two-dimensional slice, choose a combination of the breakpoint data from the vertical and horizontal dropdown lists.

1 In the property dialog box, change **Number of table dimensions** to 3.



2 In the tabular area at the bottom, change the horizontal breakpoint slice to BP3. Observe the changed view of the slice of data.

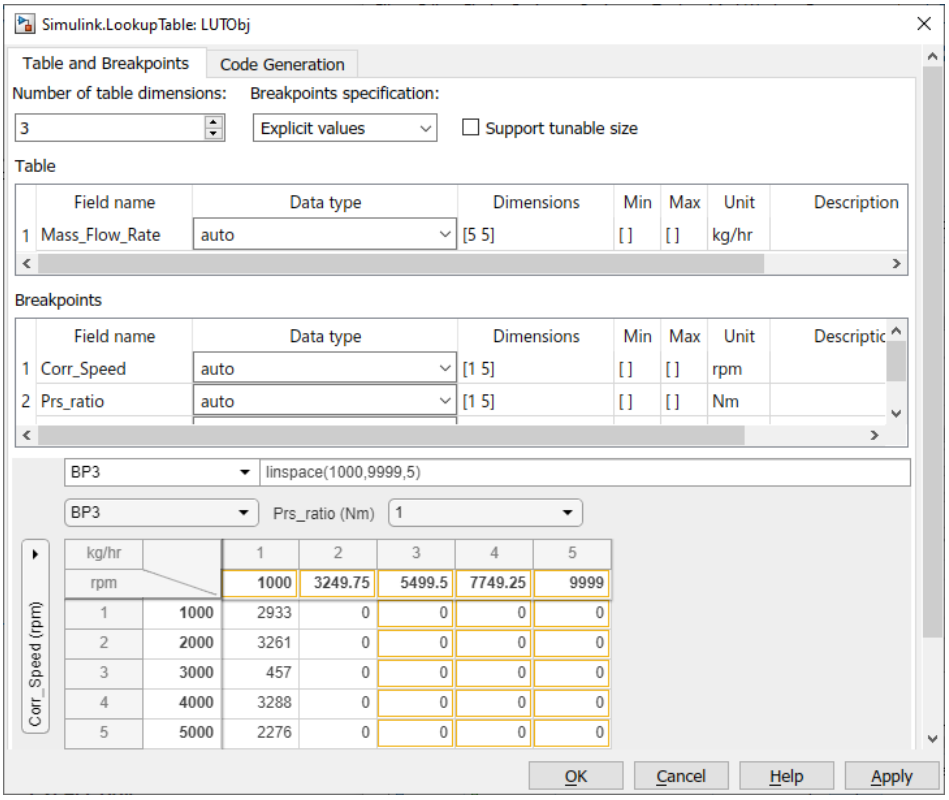


Edit Table and Breakpoint Data with MATLAB Expressions

Edit table and breakpoint data using standard table editing actions, such as cutting or copying and pasting, or directly editing table cells. You can also edit table and breakpoint data with MATLAB expressions.

This example shows how to replace the table and breakpoint using a MATLAB expression.

1. In the property dialog box, in the tabular area at the bottom, select BP3 from the horizontal dimension dropdown list.
2. In the MATLAB expression field, enter `linspace(1000,9999,5)`.
3. Observe the changed view of the slice of data.

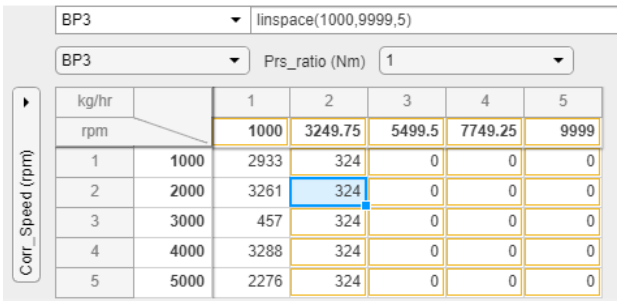


Edit Table and Breakpoint Data

You can perform standard table edits in the property dialog box tabular area. This example shows how to apply the same value to multiple cells.

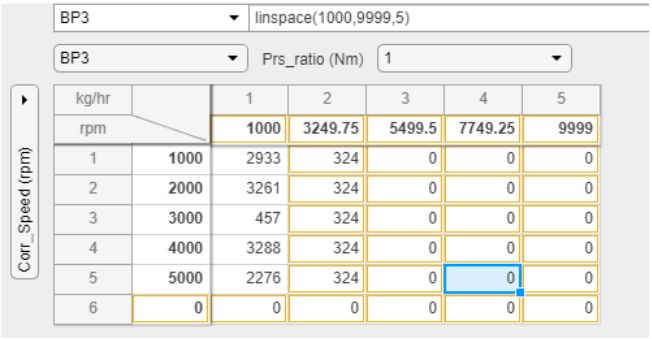
Replace a column of data with the same value.

1. In the property dialog box tabular area, select row 1, column 2 of the table data.
2. Select the entire column by dragging down the column.
3. Enter 324.
4. Press **Ctrl+Enter**.
5. Observe that the entire column of data is now 324.



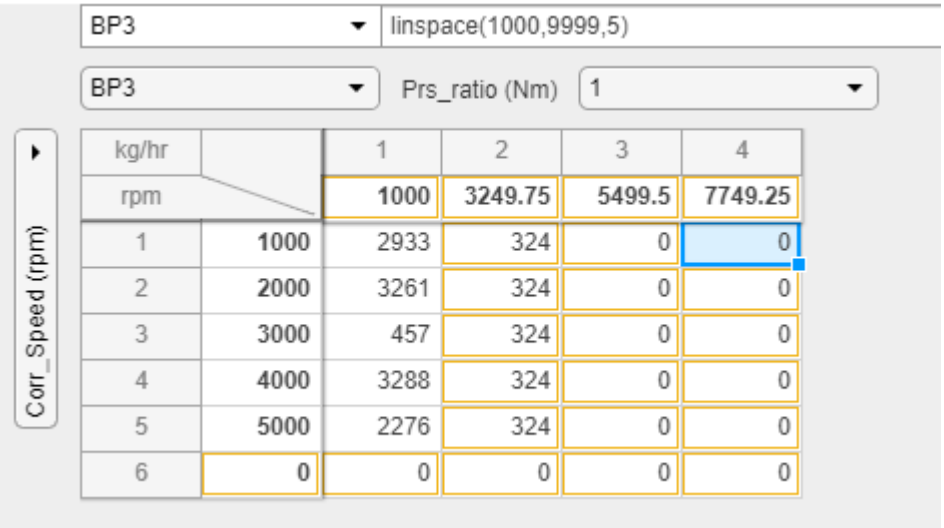
Add a row to the table. You can add or remove a row only after or from the current last row of the table.

1 In the property dialog box tabular area, right-click the last row of the table and select **Add new row**.



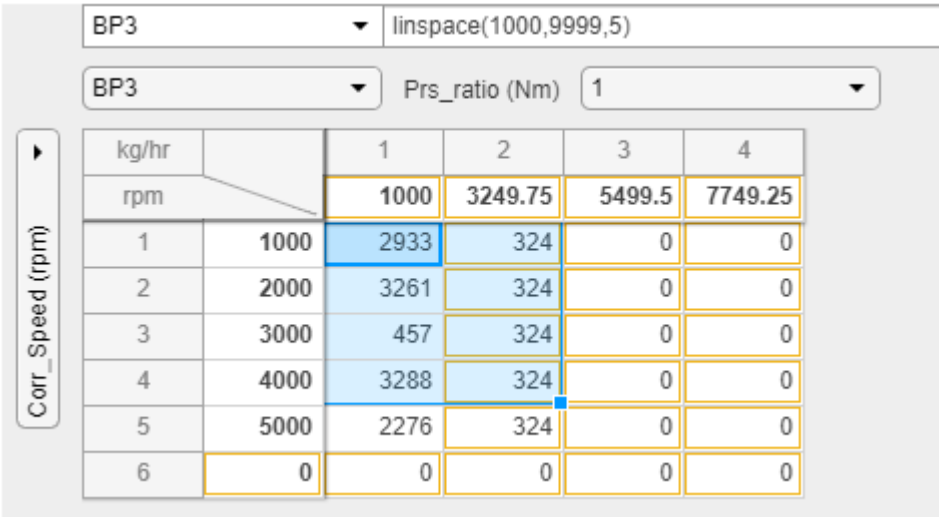
Delete a column in the table. You can add or remove a column only after or from the current last row of the table.

1 In the property dialog box tabular area, right-click the last column of the table and select **Remove right-most column**.

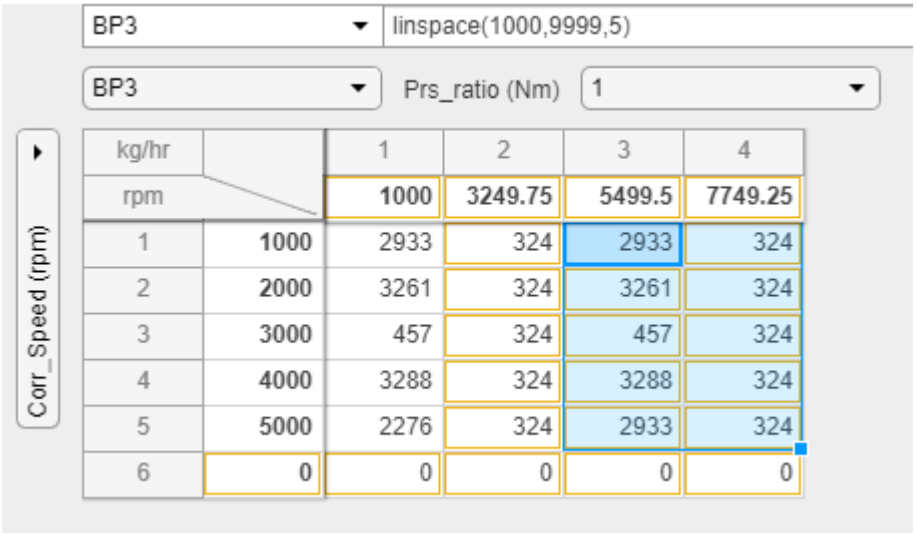


Select and paste a region in the table.

1 In the property dialog box tabular area, select the top-left cell of your selection and drag to the bottom-right of your selection.



2 Select another area of the table. Paste the selected data with **Ctrl+V**.



Overflow Handling

The `Simulink.LookupTable` object property dialog box handles data overflows in the tabular area by automatically changing values to ones that do not cause overflows. For example, if you enter a value of `70000` in a cell for a data type of `uint16`, the property dialog box automatically changes the value to `65535`, which is the maximum number for `uint16`. For a list of supported `Simulink.LookupTable` object property dialog box data types, see “Simulink.LookupTable Object Property Dialog Box Data Type Support” on page 38-39.

Data Validation

The `Simulink.LookupTable` object property dialog box performs data validation when you enter a table cell value and press **Enter**. For example, if you enter `NaN`, `Inf` or `-Inf` as a table cell value, the cell is outlined in red. Hover over the cell to see the error `Value must be numeric`. You must correct all `NaN`, `Inf`, and `-Inf` errors before continuing. After correcting invalid data, click **Apply**

and check that the updated data is correct. Correct and apply all `NaN` and `Inf` errors before continuing.

The MATLAB expression area also validates expressions. For example, if you enter an invalid expression, the text box is outlined in red and displays an error message on an expected expression. If you enter an expression for fixed-point data with a bias, the software evaluates and, as necessary, saturates the value to the nearest available fixed-point value before displaying the corrected value in the table.

Simulink.LookupTable Object Property Dialog Box Tabular Interface Shortcuts

Table Navigation

Action	Key or Keys
Move to the table cell above current active cell.	**Up Arrow**
Move to table cell under current active cell.	**Down Arrow**
Move to the table cell to the right of the current active cell.	**Right Arrow** or **Tab**
Move to the table cell to the left side of current active cell.	**Left Arrow** or **Tab+Shift**
Move to the first table cell in a row.	**Home**
Move to the last table cell in a row.	**End**
Move to the first table cell in a column.	**Ctrl+Home**
Move to the last table cell in a column.	**Ctrl+End**

Selection

Action	Key or Keys
Select all.	**Ctrl+A**
Extend selection of the table cell above.	**Shift+Up Arrow**
Extend selection of the table cell underneath.	**Shift+Down Arrow**
Extend selection of the table cell to the right.	**Shift+Right Arrow**
Extend selection of the table cell to the left.	**Shift+Left Arrow**
Select all table cells in the row to the left, including the current cell.	**Shift+Home**
Select all table cells in the row to the right, including the current cell.	**Shift+End**
Select all cells from the column to the top of the table, including the current cell.	**Ctrl+Shift+Home**
Select all cells from the column to the bottom of the table, including the current cell.	**Ctrl+Shift+End**

Editor

Action	Key or Keys
Open or close table cell editor.	**Enter**
Open table cell editor.	**F2**
Cancel editing and close table cell editor.	**Esc**
Clear table cell.	**Backspace** or **Delete**
Copy table cell content.	**Ctrl+C**
Cut table cell content.	**Ctrl+X**
Paste table cell content.	**Ctrl+V**
Fill all selected table cells with edited cell value.	**Ctrl+Enter**
Undo.	**Ctrl+Z**
Redo.	**Ctrl+Y**

Context Menu Navigation

Action	Key or Keys
Move the selection to the next option in context menu.	**Down Arrow**
Move the selection to the previous option in context menu.	**Up Arrow**
Select option from context menu.	**Enter**

See Also

`Simulink.LookupTable`

Update Lookup Table Blocks to New Versions

In this section...

Comparison of Blocks with Current Versions

In R2011a, the following lookup table blocks were replaced with newer versions in the Simulink library:

Block	Changes	Enhancements

Lookup Table	• Block renamed as 1-D Lookup Table
	• Icon changed
		• Default integer rounding mode changed from `Floor` to `Simplest`
		• Support for the following features:
		  • Specification of parameter data types different from input or output signal types
		  • Reduced memory use and faster code execution for nontunable breakpoints with even spacing
		  • Cubic-spline interpolation and extrapolation
		  • Table data with complex values
		  • Fixed-point data types with word lengths up to 128 bits
		  • Specification of data types for fraction and intermediate results
		  • Specification of index search method
		  • Specification of diagnostic for out-of-range inputs

Block	Changes	Enhancements

Lookup Table (2-D)	• Block renamed as 2-D Lookup Table
	• Icon changed
		• Default integer rounding mode changed from `Floor` to `Simplest`
		• Support for the following features:
		  • Specification of parameter data types different from input or output signal types
		  • Reduced memory use and faster code execution for nontunable breakpoints with even spacing
		  • Cubic-spline interpolation and extrapolation
		  • Table data with complex values
		  • Fixed-point data types with word lengths up to 128 bits
		  • Specification of data types for fraction and intermediate results
		  • Specification of index search method
		  • Specification of diagnostic for out-of-range inputs
		• Check box for **Require all inputs to have the same data type** now selected by default

Lookup Table (n-D)	• Block renamed as n-D Lookup Table • Icon changed	• Default integer rounding mode changed from `Floor` to `Simplest`

Compatibility of Models with Older Versions of Lookup Table Blocks

When you load existing models that contain the Lookup Table, Lookup Table (2-D), and Lookup Table (n-D) blocks, those versions of the blocks appear. The current versions of the lookup table blocks appear only when you drag the blocks from the Simulink Library Browser into new models.

If you use the `add_block` function to add the Lookup Table, Lookup Table (2-D), or Lookup Table (n-D) blocks to a model, those versions of the blocks appear. If you want to add the *current* versions of the blocks to your model, change the source block path for `add_block`:

Block	Old Block Path	New Block Path
Lookup Table	`simulink/Lookup Tables/Lookup Table`	`simulink/Lookup Tables/1-D Lookup Table`
Lookup Table (2-D)	`simulink/Lookup Tables/Lookup Table (2-D)`	`simulink/Lookup Tables/2-D Lookup Table`
Lookup Table (n-D)	`simulink/Lookup Tables/Lookup Table (n-D)`	`simulink/Lookup Tables/n-D Lookup Table`

How to Update Your Model

To update your model to use current versions of the lookup table blocks, follow these steps:

Step	Action	Reason
1	Run the Upgrade Advisor.	Identify blocks that do not have compatible settings with the 1-D Lookup Table and 2-D Lookup Table blocks.
2	For each block that does not have compatible settings: • Decide how to address each warning. • Adjust block parameters as needed.	Modify each Lookup Table or Lookup Table (2-D) block to ensure compatibility with the current versions.
3	Repeat steps 1 and 2 until you are satisfied with the results of the Upgrade Advisor check.	Ensure that block replacement works for the entire model.

After block replacement, the block names that appear in the model remain the same. However, the block icons match the ones for the 1-D Lookup Table and 2-D Lookup Table blocks. For more information about the Upgrade Advisor, see “Model Upgrades”.

What to Expect from the Model Advisor Check

The Model Advisor check groups all Lookup Table and Lookup Table (2-D) blocks into three categories:

- Blocks that have compatible settings with the 1-D Lookup Table and 2-D Lookup Table blocks
- Blocks that have incompatible settings with the 1-D Lookup Table and 2-D Lookup Table blocks
- Blocks that have repeated breakpoints

Blocks with Compatible Settings

When a block has compatible parameter settings, automatic block replacement can occur without backward incompatibilities.

Lookup Method in the Lookup Table or Lookup Table (2-D) Block	Parameter Settings After Automatic Block Replacement	
	Interpolation	Extrapolation
`Interpolation-Extrapolation`	`Linear`	`Linear`
`Interpolation-Use End Values`	`Linear`	`Clip`
`Use Input Below`	`Flat`	Not applicable

Depending on breakpoint spacing, one of two index search methods can apply.

Breakpoint Spacing in the Lookup Table or Lookup Table (2-D) Block	Index Search Method After Automatic Block Replacement
Not evenly spaced	`Binary search`
Evenly spaced and tunable	A prompt appears, asking you to select `Binary search` or `Evenly spaced points`.
Evenly spaced and not tunable	

Blocks with Incompatible Settings

When a block has incompatible parameter settings, the Model Advisor shows a warning and a recommended action, if applicable.

- If you perform the recommended action, you can avoid incompatibility during block replacement.
- If you use automatic block replacement without performing the recommended action, you might see numerical differences in your results.

Incompatibility Warning	Recommended Action	What Happens for Automatic Block Replacement
The **Lookup Method** is `Use Input Nearest` or `Use Input Above`. The replacement block does not support these lookup methods.	Change the lookup method to one of the following options: • `Interpolation - Extrapolation` • `Interpolation - Use End Values` • `Use Input Below`	The **Lookup Method** changes to `Interpolation - Use End Values`. In the replacement block, this setting corresponds to: • **Interpolation** set to `Linear` • **Extrapolation** set to `Clip` You also see a message that explains possible numerical differences.
The **Lookup Method** is `Interpolation - Extrapolation`, but the input and output are not the same floating-point type. The replacement block supports linear extrapolation only when all inputs and outputs are the same floating-point type.	Change the extrapolation method or the port data types of the block.	
The block uses small fixed-point word lengths, so that interpolation uses only one rounding operation. The replacement block uses two rounding operations for interpolation.	None	You see a message that explains possible numerical differences.

Blocks with Repeated Breakpoints

When a block has repeated breakpoints, the Model Advisor recommends that you change the breakpoint data and rerun the check. You cannot perform automatic block replacement for blocks with repeated breakpoints.

See Also

Interpolation Using Prelookup | Prelookup | n-D Lookup Table

More About

- “Model Upgrades”
- “About Lookup Table Blocks” on page 38-2
- “Anatomy of a Lookup Table” on page 38-4
- “Lookup Tables Block Library” on page 38-5
- “Edit Lookup Tables” on page 38-20
- “Guidelines for Choosing a Lookup Table” on page 38-7

Working with Block Masks

Masking Fundamentals

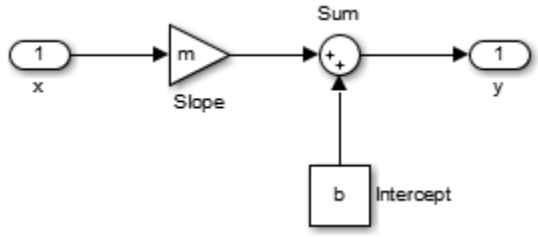
A mask is a custom interface for a block that hides the block content, making it appear as an atomic block with its own icon and parameter dialog box. It encapsulates the block logic, provides controlled access to the block data, and simplifies the graphical appearance of a model.

When you mask a block, a mask definition is created and saved along with the block. A mask changes only the block interface, and not the underlying block characteristics. You can provide access to one or more underlying block parameters by defining corresponding mask parameters on the mask.

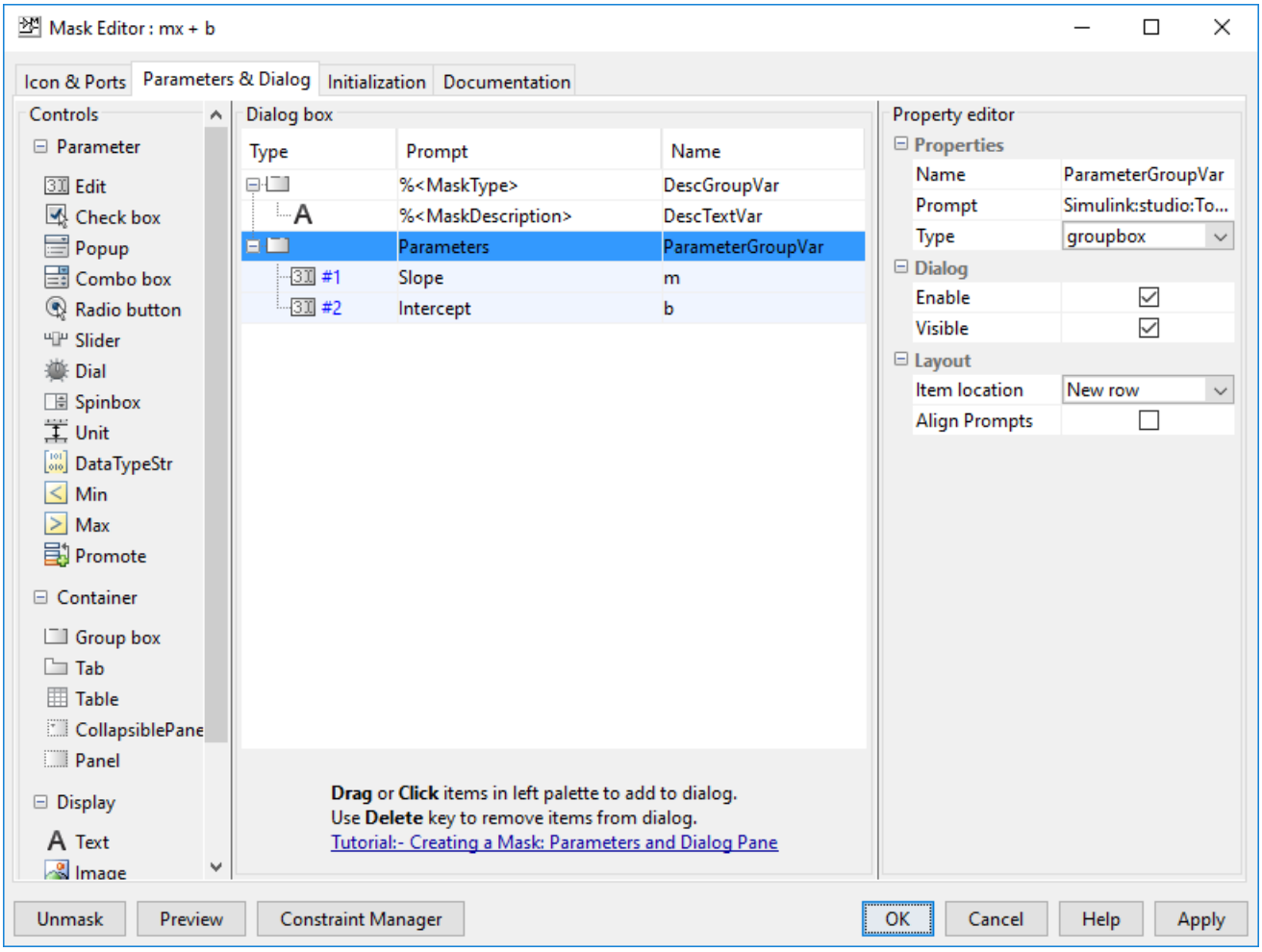
Mask a Simulink block to:

- Display a meaningful icon on a block
- Provide a customized dialog box for the block
- Provide a dialog box that enables you to access only select parameters of the underlying blocks
- Provide users customized description that is specific to the masked block
- Initialize parameters using MATLAB code

Consider the model masking_example that represents the equation of line `y = mx + b`.

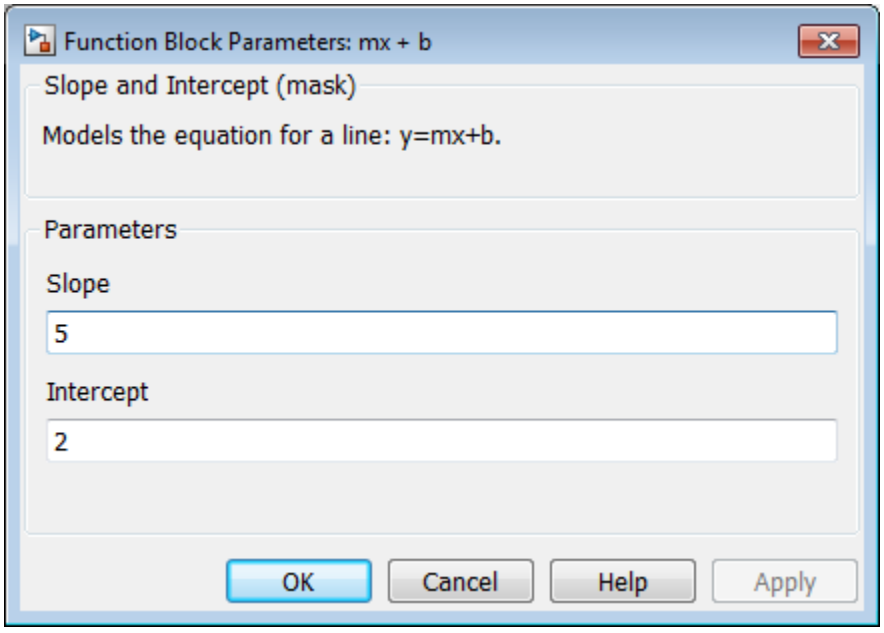


Each block has its own dialog box, making it complicated to specify the values for the line equation variables. To simplify the user interface, a mask is applied on the top-level subsystem block.



Here the variable `m` represents slope and the variable `b` represents the intercept for the line equation `y = mx + b`.

The mask dialog box displays the fields for **Slope** and **Intercept** that are internally mapped to variables `m` and `b`.



Masked blocks do not support content preview. To preview the contents of a subsystem, see “Preview Content of Model Components” on page 1-33.

Tip For masking examples, see Simulink Masking Examples. The examples are grouped by type. In an example model:

- To view the mask definition, double-click the View Mask block.
- To view the mask dialog box, double-click the block.

Examples of few blocks that cannot be masked are:

- Scope blocks
- Simulink Function block
- Initialize, Terminate and Reset Function blocks
- Gauge blocks

Masking Terminology

Term	Description
Mask icon	The masked block icon generated using drawing commands. Mask icon can be static or change dynamically with underlying block parameter values.

Term	Description
Mask parameters	The parameters that are defined in the Mask Editor and appear on the mask dialog box. Setting the value of a mask parameter on the mask dialog box sets the corresponding block parameter value.
Mask initialization code	MATLAB Code that initializes a masked block or reflects current parameter values. Add mask initialization code in the **Initialization** pane of the Mask Editor dialog box. For example, add initialization code to set a parameter value automatically.
Mask dialog callback code	MATLAB Code that runs in the base workspace when the value of a mask parameter changes. Use callback code to change the appearance of a mask dialog box dynamically and reflect current parameter values. For example, enable visible parameters on the dialog box.
Mask documentation	Description and usage information for a masked block defined in the Mask Editor.
Mask dialog box	A dialog box that contains fields for setting mask parameter values and provides mask description.
Mask workspace	Masks that define mask parameters or contain initialization code have a mask workspace. This workspace stores the evaluated values of the mask parameters and temporary values used by the mask.

See Also

More About

- “Create Block Masks”
- “Create a Simple Mask” on page 39-6
- “Mask Editor Overview”
- Set mask parameters
- Creating a Mask: Masking Fundamentals (3 min, 46 sec)

Create a Simple Mask

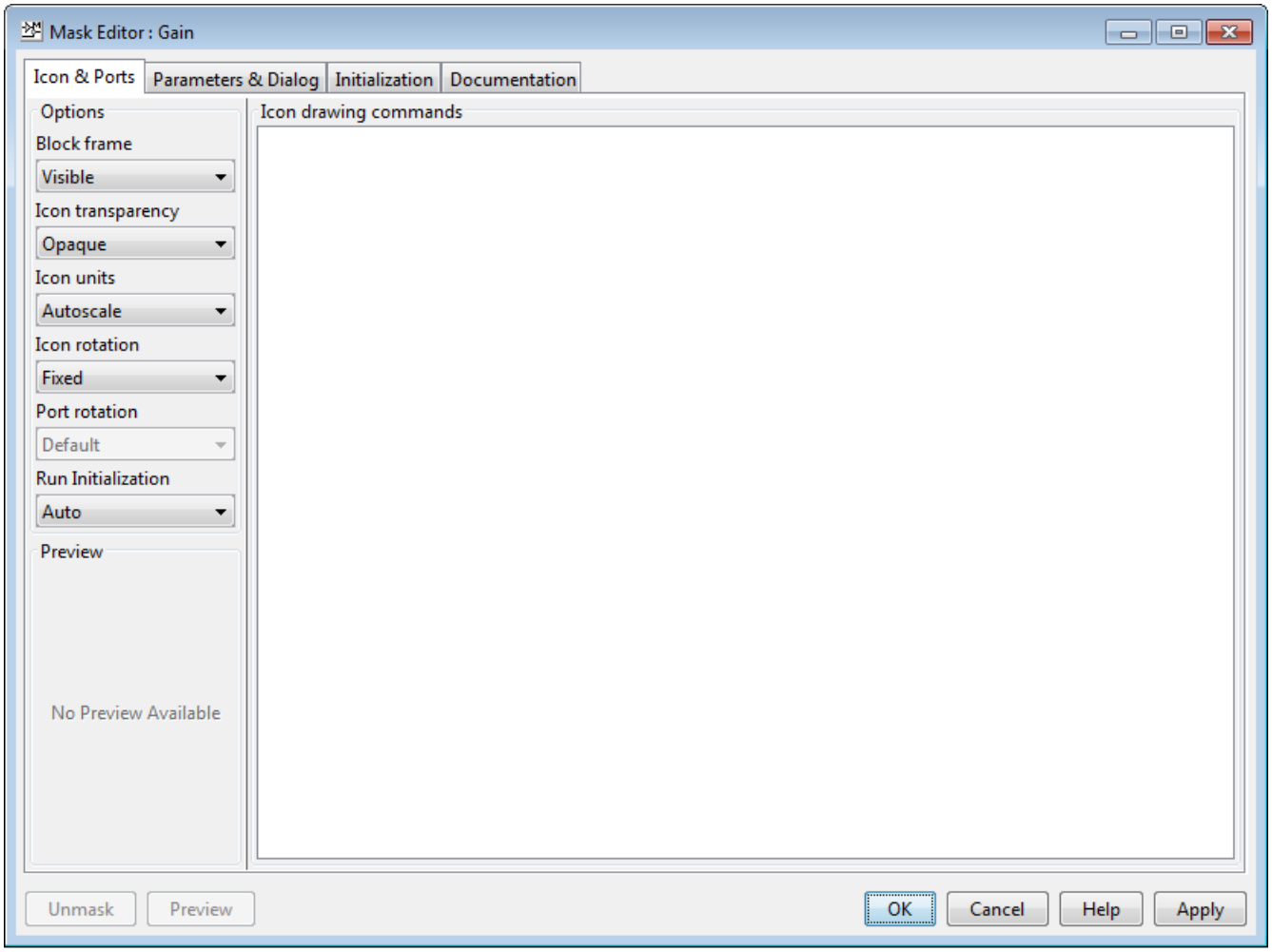
You can mask a block interactively by using the Mask Editor or mask it programmatically. This example describes how to mask a block by using the **Mask Editor**. To mask a block programmatically, see “Control Masks Programmatically” on page 39-29.

For masking examples, see Simulink Masking Examples.

Step 1: Open Mask Editor

1. Open the model in which you want to mask a block. For example, open `subsystem_example`.

   This model contains a Subsystem block that models the equation for a line: `y = mx + b`.
2. Select the Subsystem block and on the **Subsystem** tab, in the **Mask** group,click **Create Mask**.



Step 2: Define the Mask

The **Mask Editor** contains four tabs that enable you to define the block mask and customize the dialog box for the mask.

For detailed information on each pane, see “Mask Editor Overview”.

Icon & Ports Tab

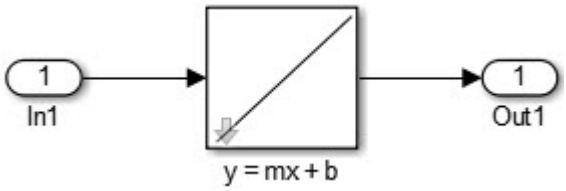
Use this tab to create an icon for the block mask. You can use the **Options** pane on the left to specify icon properties and icon visibility.

Add an image to the block mask.

1. In the **Block frame** drop-down box, select **Visible**.
2. In the **Icon transparency** drop-down box, select **Opaque**.
3. In the **Icon units** drop-down box, select **Autoscale**.
4. To restrict the icon rotation, select **Fixed** from the **Icon rotation** list.
5. In the **Icon drawing commands** text box, type,

   ```
   x = [0 0.5 1 1.5];y = [0 0.5 1 1.5];
   % An example to defines the variables x and y
   plot(y,x) % Command to plot the graph
   ```

   For more information on drawing command syntax, see “Icon drawing commands”.
6. To save the changes, click **Apply**. To preview the block mask icon without exiting the **Mask Editor**, click **Preview**



Note For detailed information, see “Icon & Ports Pane”.

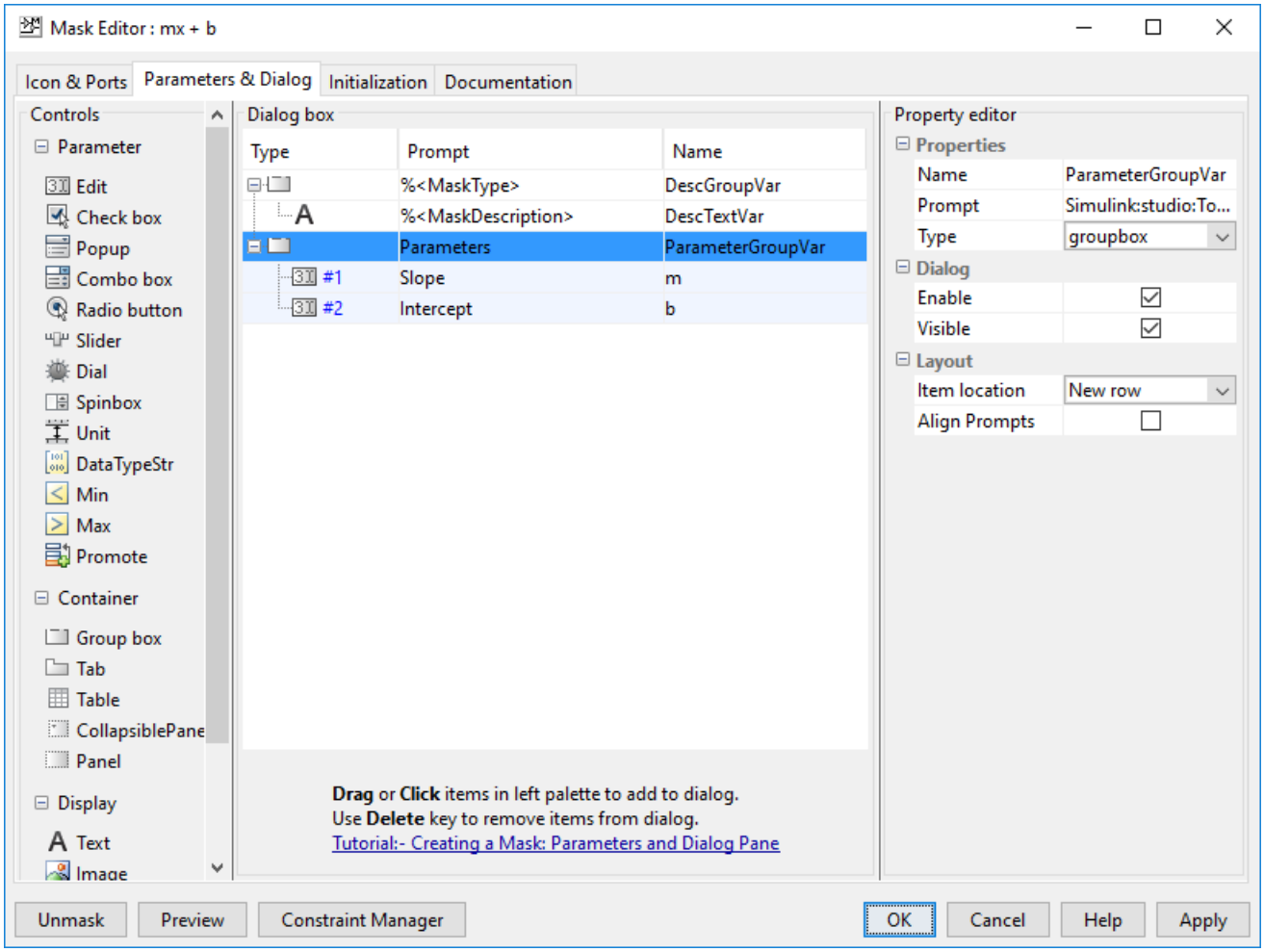
You can create static or dynamic block mask icons. For more information, see “Draw Mask Icon” on page 39-17 and slexMaskDisplayAndInitializationExample.

Parameters & Dialog Tab

Use this tab to add controls like parameters, displays, and action items to the mask dialog box.

To add **Edit** boxes to the block mask.

1. In the left pane, under **Parameter**, click **Edit** twice to add two new rows in the **Dialog box** pane.
2. Type `Slope` and `Intercept` in the **Prompt** column for the two **Edit** parameters. The value that you enter in **Prompt** column appears on the mask dialog box. Similarly, type `m` and `b` in the **Name** column. The value you enter in **Name** column is the mask parameter name. The mask parameter name must be a valid MATLAB name.
3. In the right pane, under **Property editor**, provide values in the **Properties**, **Dialog**, and **Layout** sections.
4. Click **Apply**.



5 To preview the mask dialog box without exiting the Mask Editor, click **Preview**.

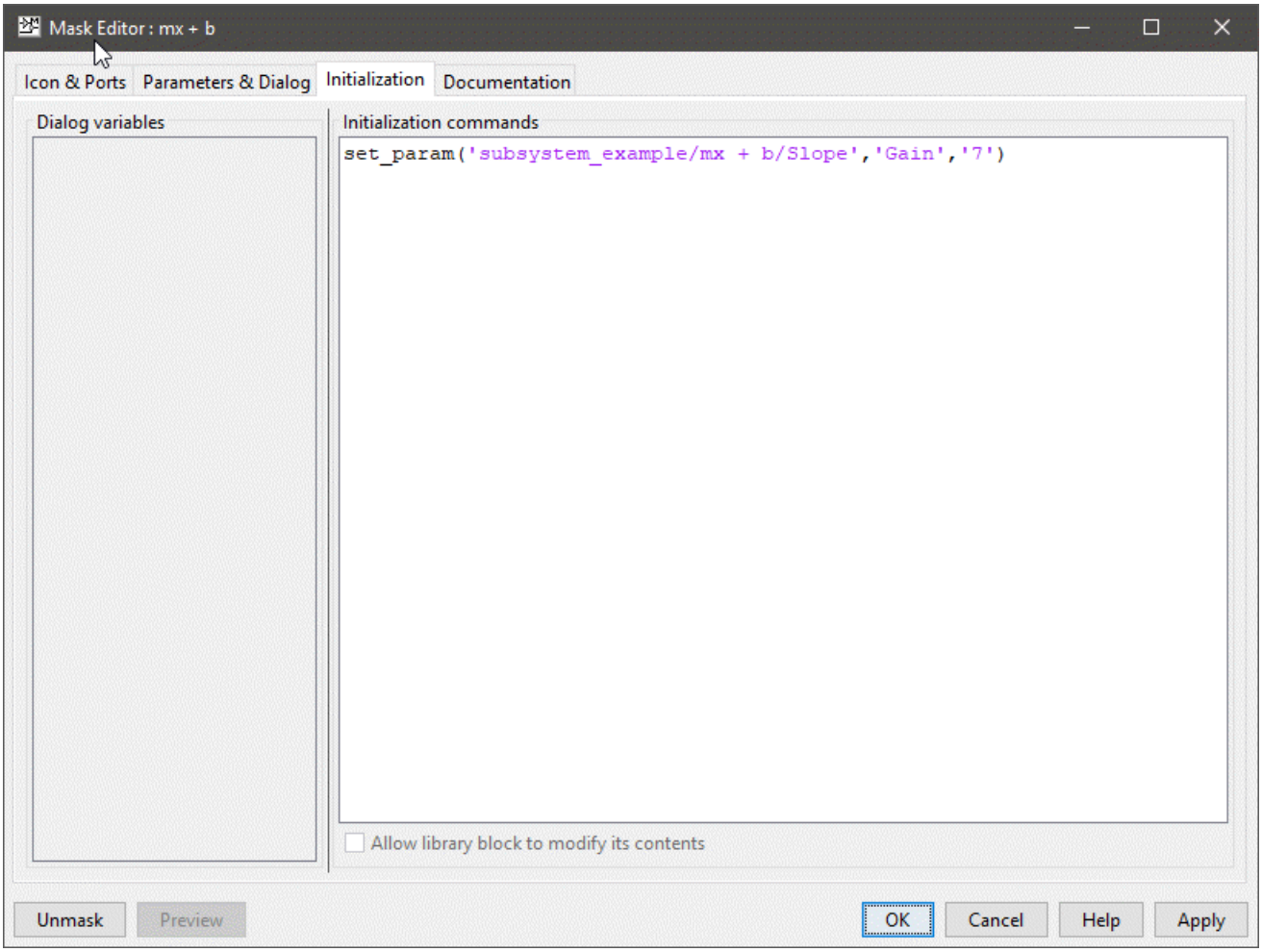
For detailed information, see “Parameters & Dialog Pane”.

Note A Simulink mask parameter cannot reference another parameter on the same mask.

Initialization Tab

Use this pane to specify MATLAB code to control the mask parameters. For example,you can provide a predefined value for a mask parameter.

Consider the equation `y = mx + b` in the example. To set the value of the child block corresponding to 'm', you can use the `set_param` function in the initialization pane.



Note For detailed information, see “Initialization Pane”.

Documentation Tab

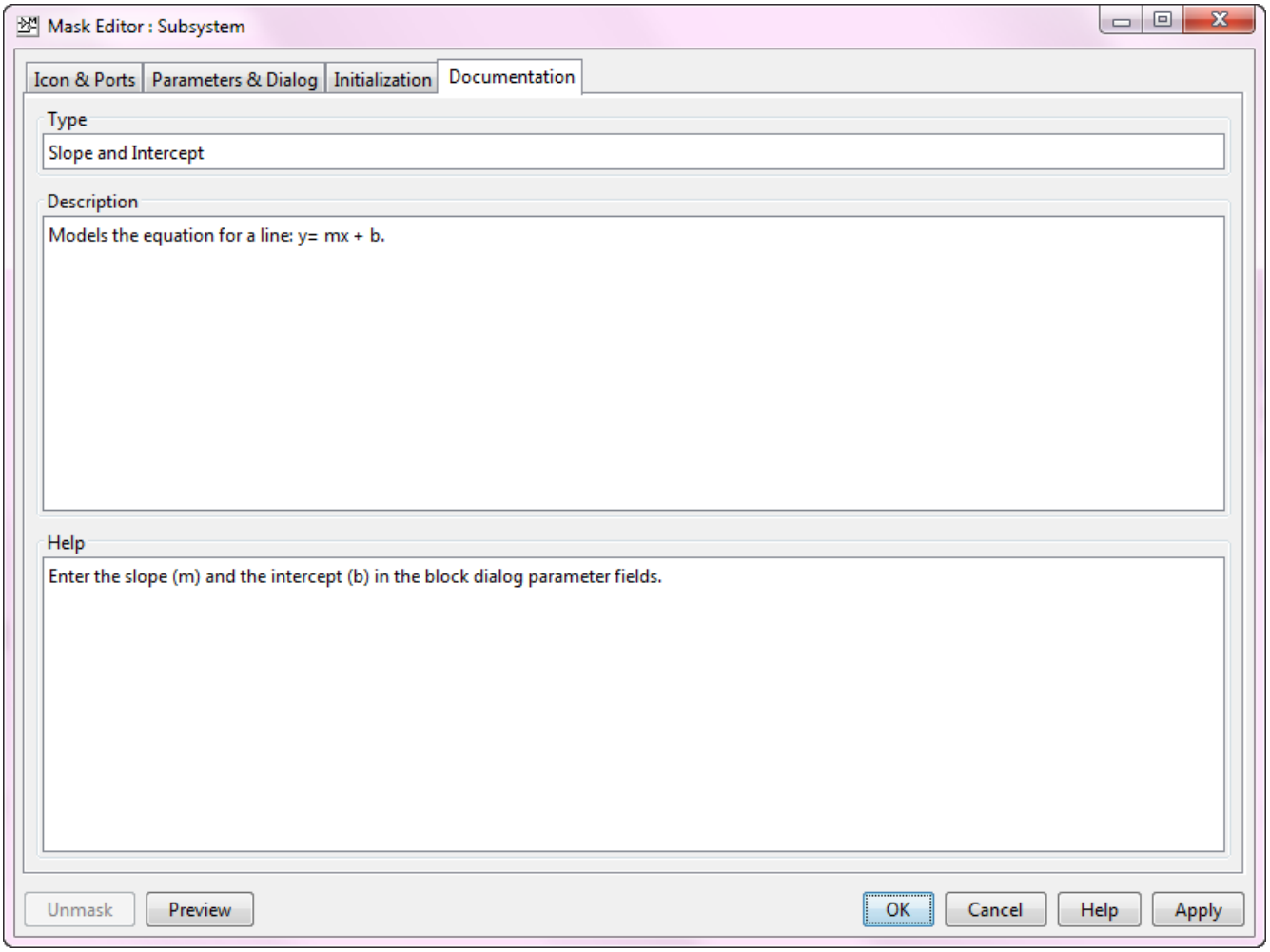
Use this tab to add a name, description, and additional information for the mask.

The **Documentation** tab contains these fields:

1. **Type**: You can add a name for the block mask in this box. The mask name appears on top of the mask dialog box. You cannot add new lines.
2. **Description**: You can add a description for the block mask in this box. By default, the description is displayed below the mask name, and it can contain new lines and spaces.
3. **Help**. You can add additional information for the block mask in this box. You click **Help** on the mask dialog box, this information is displayed. You can use plain text, HTML and graphics, URLs, and `web` or `eval` commands to add information in the **Help** field.

   After you have added information in the **Mask Editor**, click **Apply** or **OK**.

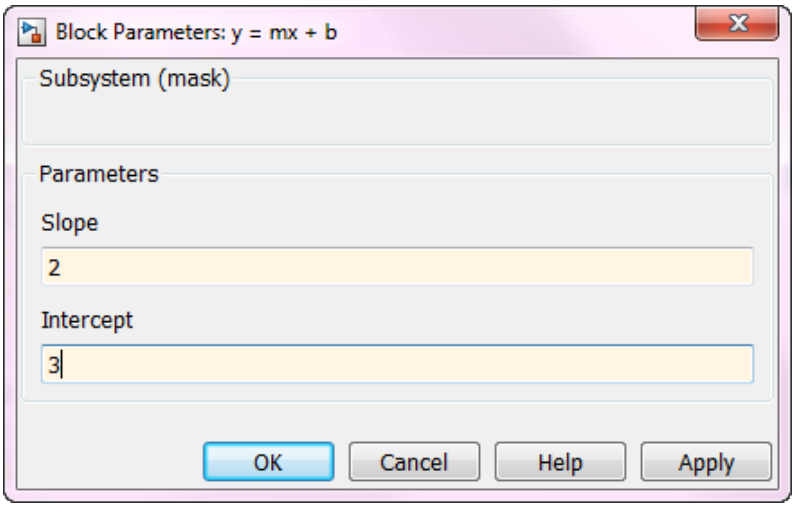
   The block is now masked.



Note For detailed information, see “Documentation Pane”.

Step 3: Operate on Mask

1. You can preview the mask and choose to unmask the block or edit the block mask.
2. Double-click the masked block.



The mask dialog box appears.

3. Type values in the `Slope` and `Intercept` boxes of the mask dialog box. To view the output, simulate the model.
4. Click **OK**.
5. To edit the mask definition, select the subsystem block and click **Edit Mask** from the Subsystem tab in the Toolstrip. For more information, see “Manage Existing Masks” on page 39-12.
6. Select the masked block and on the **Subsystem Block** tab, in the **Mask** group, click **Look Under Mask** to view:

   - The blocks inside the masked subsystem
   - The built-in block dialog box of a masked block
   - The base mask dialog box of a linked masked block

See Also

More About

- “Create Block Masks”
- Creating a Mask: Masking Fundamentals (3 min, 46 sec)
- “Mask Editor Overview”
- “Masking Fundamentals” on page 39-2

Manage Existing Masks

Change a Block Mask

You can change an existing mask by reopening the Mask Editor and using the same techniques that you used to create the mask:

1. Select the masked block.
2. On the **Subsystem Block** tab, in the **Mask** group, click **Edit Mask**.

The Mask Editor reopens, showing the existing mask definition. Change the mask as needed. After you change a mask, click **Apply** to preserve the changes.

View Mask Parameters

To display a mask dialog box, double-click the block. Alternatively, select the block and on the **Block** tab, in the **Mask** group, click **Mask Parameters**.

Tip Each block has a block parameter dialog box. To view the block parameters dialog box of a masked block, right-click the masked block and select **Block Parameters (BlockType)**.

Look Under Block Mask

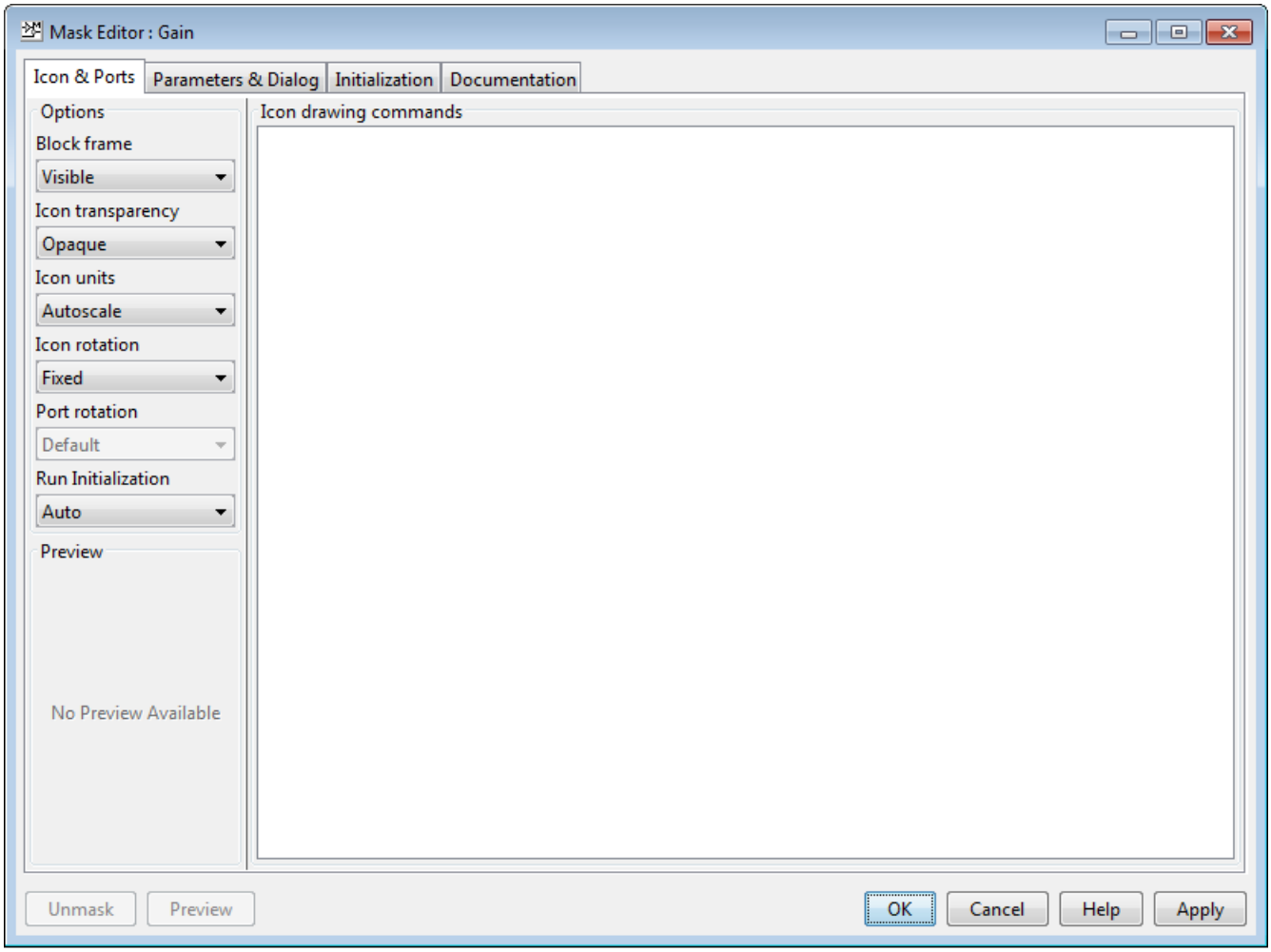
To see the block diagram under a masked Subsystem block, built-in block, or the model referenced by a masked model block, select the block and on the **Subsystem** tab, click **Look Under Mask**.

Remove Mask

To remove a mask from a block,

1. Select the block.
2. On the **Block** tab, in the **Mask** group,click **Edit Mask**.

   The Mask Editor opens and displays the existing mask, for example:



3 Click **Unmask** in the lower left corner of the Mask Editor.

The Mask Editor removes the mask from the block.

See Also

More About

- “Create a Simple Mask” on page 39-6

Mask Callback Code

In this section...

Add Mask Code

You can use MATLAB code to initialize a mask and to draw mask icons. Since the location of code affects model performance, add your code to reflect the functionality you need.

Purpose	Add in Mask Editor	Programmatic Specification
Initialize the mask	**Initialization** pane	`MaskInitialization` parameter
Draw mask icon	**Icon & Ports** pane	`MaskDisplay` parameter
Callback code for mask parameters	**Parameters & Dialog** pane	`MaskCallbacks` parameter

Execute Drawing Command

Place MATLAB code for drawing mask icons in the **Icon Drawing Commands** section of the **Icon & Ports** pane. Simulink executes these commands sequentially to redraw the mask icon when:

- Block is rendered first on the Mask Editor canvas.
- Mask parameters and values that depend on drawing commands change.
- Block appearance is altered due to rotation or other changes.

Note Placing MATLAB code for drawing mask icons in the **Initialization** pane affects model performance. This behavior is because Simulink redraws the icon each time the masked block is evaluated in the model.

Execute Initialization Command

Initialization commands for all masked blocks in a model run when you:

- Update the diagram
- Start simulation
- Start code generation
- Apply mask changes
- Change any of the parameters that define the mask, such as `MaskDisplay` and `MaskInitialization`, using the Mask Editor or `set_param`.
- Rotate or flip the masked block, if the icon depends on initialization commands.

- Cause the icon to be drawn or redrawn, and the icon drawing depends on initialization code.
- Change the value of a mask parameter by using the block dialog box or `set_param`.
- Copy the masked block within the same model or between different models.

When you open a model, Simulink locates visible masked blocks that reside at the top level of the model or in an open subsystem.

Simulink only executes the initialization commands for these visible masked blocks if they meet either of the following conditions:

- The masked block has icon drawing commands.

  Note Simulink does not initialize masked blocks that do not have icon drawing commands, even if they have initialization commands during model load.

- The masked subsystem belongs to a library and has the **Allow library block to modify its contents** parameter enabled.

When you load a model into memory without displaying it graphically, no initialization commands initially run for any masked blocks. See “Load a Model” on page 1-2 and `load_system` for information about loading a model without displaying it.

Note The non-tunable parameters of a masked block are not evaluated if the model is already compiled (initialized).

Execute Callback Code

Mask parameter callback codes are executed in a temporary workspace and not in the base workspace. If you need a variable created in the callback to be available later(not during callback processing), you must explicitly assign those variables to the base workspace.

Simulink executes the callback commands when:

- You open the mask dialog box. Callback commands execute sequentially, starting with the top mask dialog box.
- You modify a parameter value in the mask dialog box and then change the cursor location. For example, you press the **Tab** key or click into another field in the dialog box after changing the parameter value.
- You modify the parameter value by using the `set_param` command, the callback commands execute.
- You modify the parameter value, either in the mask dialog box or using `set_param`, and then apply the change by clicking **Apply** or **OK**. Mask initialization commands execute after callback commands. For more information, see “Initialization Pane”.
- You hover over a masked block to see the tool tip for the block, when the tool tip contains parameter names and values.

  Note Callback commands do not execute if the mask dialog box is open when the block tool tip appears.

- You update a diagram by pressing **Ctrl+D** or by clicking **Update Model** on the **Modeling** tab in the Simulink Editor.
- If you close a mask dialog box without saving the changes, the Callback command for parameters is executed sequentially.

Note Buttons on mask dialog box are unavailable when the callback code associated with the button is being executed.

For related Simulink example models, see:

- Sequence mask callbacks
- Unsafe mask callbacks
- Unsafe nested mask callbacks

See Also

More About

- “Create Block Masks”
- “Initialize Mask” on page 39-20

Draw Mask Icon

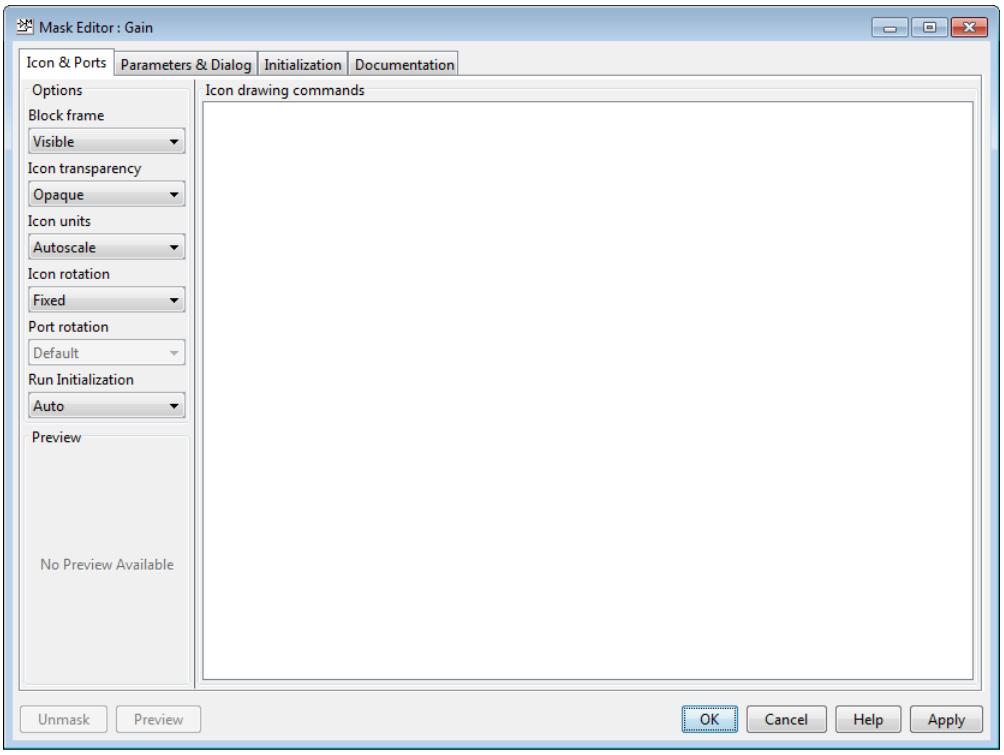
You can create icons that update when you change the mask parameters to reflect the purpose of the block. This example shows how to use drawing commands to create a mask icon.

In this section...

Draw Static Icon

A static mask icon remains unchanged, independent of the value of the mask parameters.

1. Select the masked block that requires the icon.
2. On the **Block** tab, in the **Mask** group, click **Edit Mask**.



3. In the **Icons & Ports** tab, enter this command in the **Icon Drawing commands** pane:

```
% Use specified image as mask icon
image('engine.jpg')
```

The image file must be on the MATLAB path.

For more examples of drawing command syntax, see “Icon drawing commands”.

Images in formats `.cur`, `.hdf4`, `.ico`, `.pcx`, `.ras`, `.xwd`, `.svg` cannot be used as block mask images. However, you can use images in these formats if you wrap the file name in the `imread()`

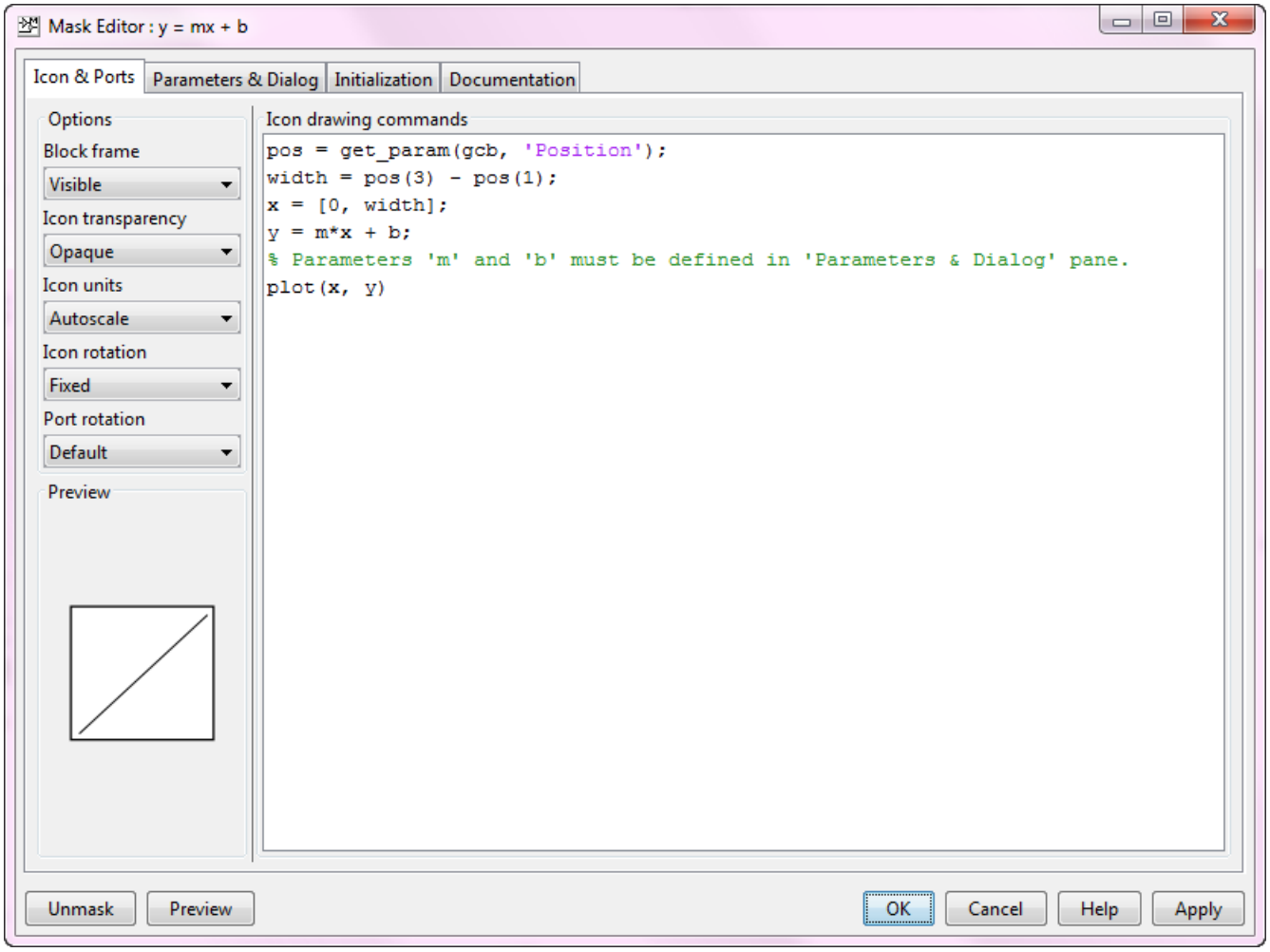
function and use the RGB triplet. Using the `imread()` function is not efficient. However, it is still supported for backward compatibility.

Draw Dynamic Icon

A dynamic icon changes with the values of the mask parameters. Use it to represent the purpose of the masked block.

1. Select the masked block that requires the icon.
2. On the **Block** tab, in the **Mask** group, click **Edit Mask**.

   The Mask Editor opens.



3. In the **Icons & Ports** tab, enter this command in the **Icon Drawing commands** pane:

```
pos = get_param(gcb, 'Position');
width = pos(3) - pos(1);
x = [0, width];
y = m*x + b;
% Parameters 'm' and 'b' must be defined in 'Parameters & Dialog' pane.
plot(x,y)
```

4 Under **Options**, set **Icon Units** to `Pixels`.

The drop-down lists under **Options** allow you to specify icon frame visibility, icon transparency, drawing context, icon rotation, and port rotation.

5 Click **Apply**. To view the icon generated, see model `masking_example`.

Note If Simulink cannot evaluate all commands in the **Icon Drawing commands** pane to generate an icon, three question marks (`? ? ?`) appear on the mask.

See `slexMaskDisplayAndInitializationExample` for more examples of icon drawing commands. This model shows how to draw:

- Static mask
- Dynamic shape mask
- Dynamic text mask
- Image mask

See Also

More About

- “Create Block Masks”

Initialize Mask

You can add MATLAB code in the Initialization pane of the Mask Editor to initialize a masked block. Simulink executes these initialization commands to initialize a masked subsystem at critical times, such as model loading and start of a simulation run. For more information, see “Execute Initialization Command” on page 39-14.

You can add mask initialization code for these cases:

- To specify the initial values of mask parameters. For example to specify an initial value of parameter `a`, type `a = 5` in the Initialization pane.
- To specify the value of a child block. For example,

  ```
  set_param('Child block Name','Parameter name','Parameter Value')
  ```
- To create a self-modifiable mask. For more information, see Self-Modifying Mask.

The initialization code of a masked subsystem can refer only to the variables in its local workspace.

When you reference a block with, or copy a block into, a model, the mask dialog box displays the specified default values. You cannot use mask initialization code to change mask parameter default values in a library block or any other block.

Ensure that the mask parameters used in the mask initialization code are defined. Errors in mask initialization are displayed when editing the mask initialization commands, but this is only possible if all the mask parameter values are evaluated without errors.

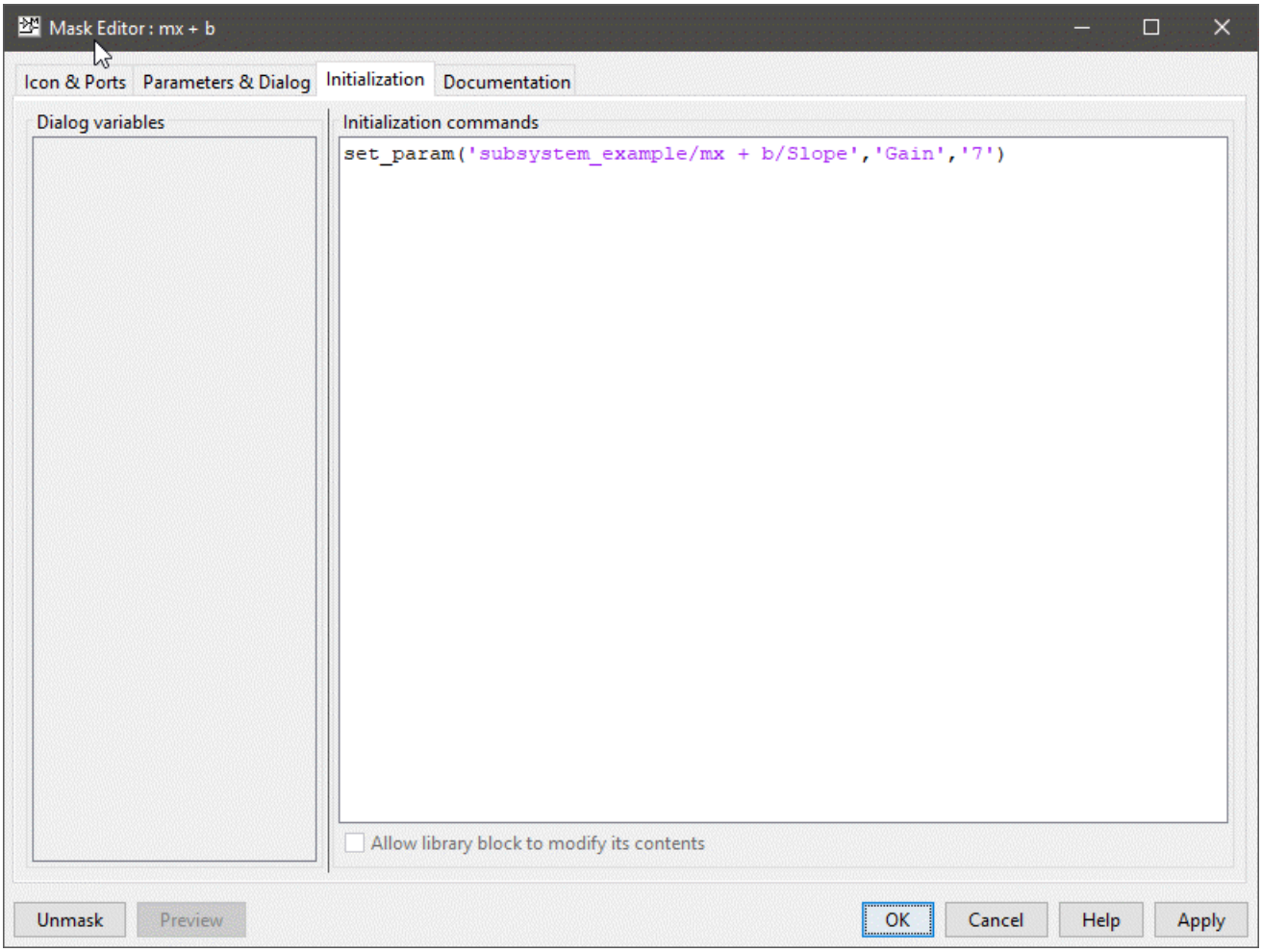
Note

- Blocks that contain initialization code do not work as expected when using model reference.
- When you use `set_param` in the mask initialization code of a Subsystem block, all the child blocks are also evaluated.

Use the Mask Editor **Initialization** pane to add MATLAB commands that initialize a masked block.

The **Initialization** pane contains these sections:

- **Dialog variables**
- **Initialization commands**



Dialog Variables

The **Dialog variables** section displays the names of the variables associated with the mask parameters of the masked block that are defined in the **Parameters** pane.

You can copy the name of a parameter from this list and paste it into the **Initialization commands** section.

You can change the name of the mask parameter variable in the list by double-clicking and editing the name.

Initialization Commands

You can add the initialization commands in this section. The initialization code must be a valid MATLAB expression, consisting of MATLAB functions and scripts, operators, and variables defined in the mask workspace. Initialization commands cannot access base workspace variables.

To avoid echoing results to the MATLAB Command Window, terminate initialization commands with a semicolon.

To view related examples, see

- Define mask display and initialization
- Use MATLAB graphics in masking

Mask Initialization Best Practices

Mask initialization commands must observe the following rules:

- Do not use initialization code to create dynamic mask dialog boxes (Dialog boxes whose appearance or control settings change depending on changes made to other control settings). Instead, use the mask callbacks that are intended for this purpose. For more information, see “Dynamic Mask Dialog Box” on page 39-39.
- Do not use initialization code to add or delete blocks during model load time.
- For nested masked subsystem, do not use `set_param` on a parent block from a child block. The child block mask and the parent block mask both could be initializing the same parameter of the block leading to unexpected behavior. For more information, see Unsafe Mask Callback Error.
- Do not use `set_param` commands on blocks that reside in another masked subsystem that you are initializing. Trying to set parameters of blocks in lower-level masked subsystems can trigger unresolved symbol errors if lower-level masked subsystems reference symbols defined by higher-level masked subsystems.

  Suppose, for example, a masked subsystem `A` contains a masked subsystem `B` which contains Gain block `C`, whose `Gain` parameter references a variable defined by `B`. Suppose also that subsystem `A` has initialization code that contains this command:

  ```
  set_param([gcb '/B/C'], 'SampleTime', '-1');
  ```

  Simulating or updating a model containing A causes an unresolved symbol error.
- You cannot use mask initialization code to create data objects. Data objects are objects of these classes:

  - `Simulink.Parameter` and subclasses
  - `Simulink.Signal` and subclasses
- Do not add initialization code to delete the same masked block.
- Use mask initialization code to control direct child blocks only.

Note Do not use mask initialization code to comment or uncomment a block.

See Also

More About

- “Create Block Masks”
- “Masking Fundamentals” on page 39-2
- “Mask Callback Code” on page 39-14
- Self-Modifying Interface Connector

Promote Parameter to Mask

In this section...

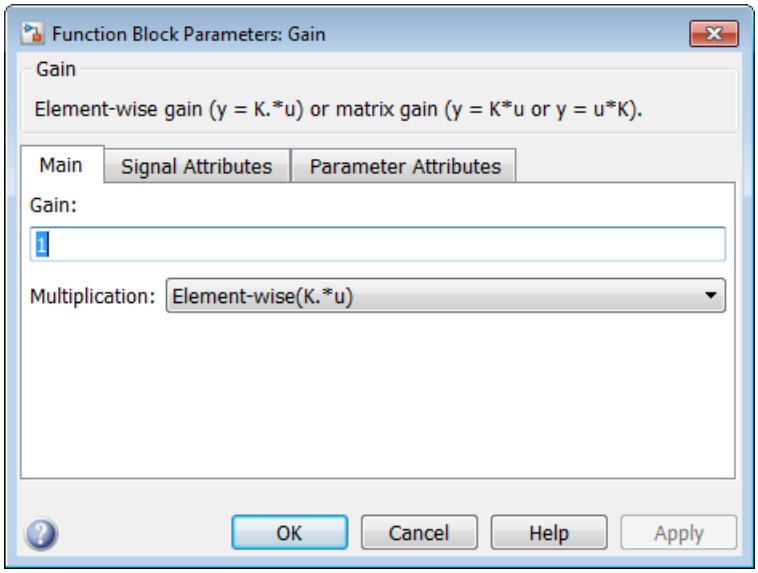
Blocks and subsystems can have multiple parameters associated with them. Block masks allow you to expose one or more of these parameters while hiding others from view. Promoting specific parameters to the mask block simplifies the interface and enables you to specify the parameters the user of the block can view and set.

You can use the **Promote** button on the Mask Editor to promote any underlying parameter of a block either to a block mask or to a subsystem mask. The promoted block parameter gets associated with a parameter in the mask, enabling you to edit the parameter value from the mask dialog box.

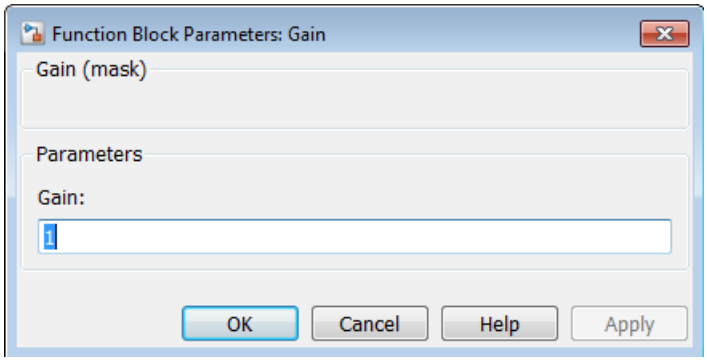
Promote parameters from the block dialog box to the mask:

- To customize the mask dialog box by moving the required parameters from the block dialog box to the mask dialog box.
- To reuse a library block at different instances of a model. For each instance of the library block, you can create individual mask dialog box by promoting parameters for each block.

Consider the block dialog box of the Gain block, which has parameters such as **Gain**, **Multiplication**.



To expose only the `Gain` parameter, mask the Gain block and promote the `Gain` parameter to the mask dialog box.



Similarly, you can mask a subsystem block and promote parameters to the mask from child blocks of the subsystem block. If the data type of subsystem child block parameters is same, you can associate a single mask parameter with multiple promoted parameters. For example, you can promote multiple **Gain** parameters in a subsystem to a single dialog box on your mask.

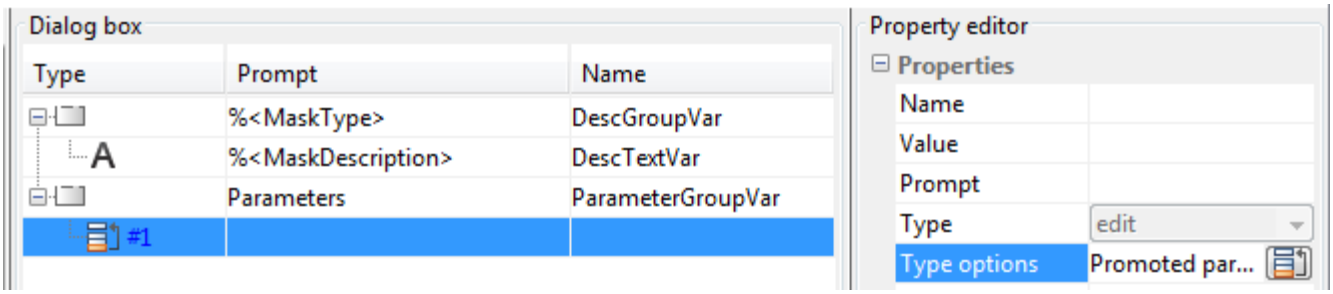
If the parameter is of data type `popup` or `DataType`, the options must also be the same for the parameters to be promoted together. The **Evaluate** attribute for all the parameters to be promoted must be similar.

For a related example, see Promote mask parameters

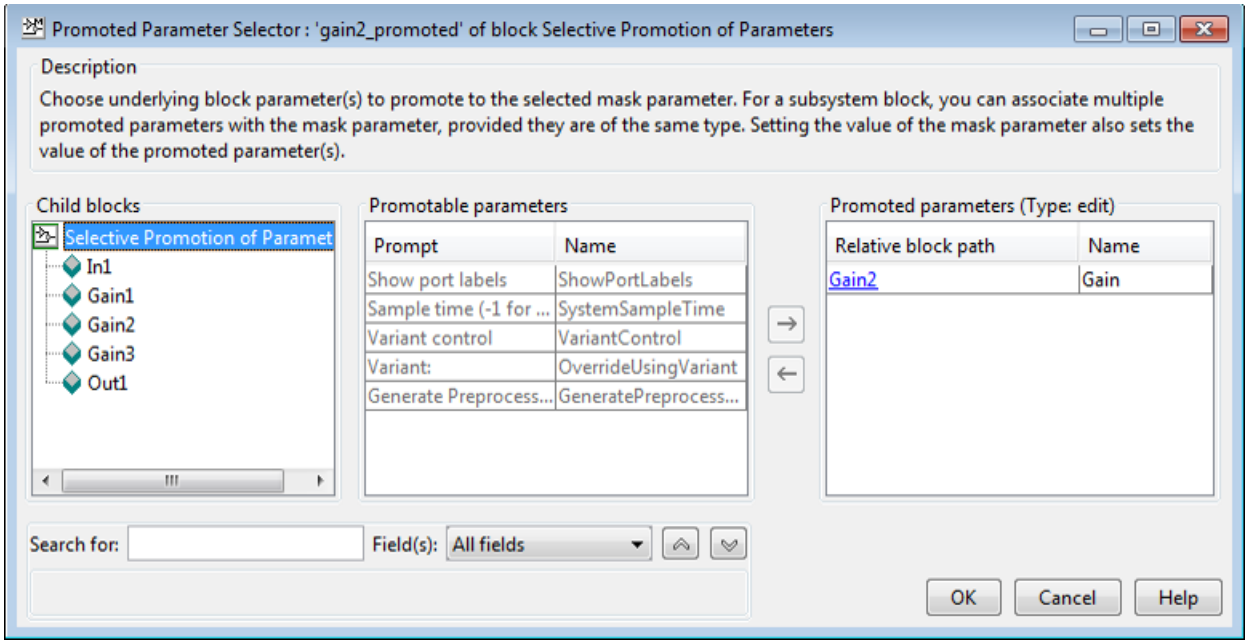
You can also change the attributes of a promoted parameter. For example, you can make a promoted parameter read only or hidden. For more information on attributes, see “Property editor”.

Promote Underlying Parameters to Block Mask

1. Select the block whose parameter you want to promote.
2. On the **Block** tab, click **Create Mask**.
3. In the **Mask Editor** dialog box, click the **Parameters & Dialog** tab.
4. In the **Controls** pane, click **Promote**.
5. In the **Property editor** pane, next to **Type options**, click  .



Use the **Promoted Parameter Selector** dialog box to select the parameters that you want to promote.



6 To add a parameter to the **Promoted parameters** list, select a parameter from the **Promotable parameters** table and click the **Add to promoted parameter list** button  .

To view the parameter properties such as **Type**, hover over the parameter name in the **Promotable parameter** pane.

Tip

- You can use the **Child blocks** list or the **Search** box to find underlying block parameters to promote.
- To prevent tuning of a property during simulation, you can disable the **Tunable** attribute while promoting a tunable parameter.

7 Click **OK**.

8 In the **Mask Editor** dialog box, edit the prompt names for the promoted parameters and click **OK**. You cannot edit the variable names. You can change the attributes of the promoted parameter in the Property editor section. For example, you can mark the promoted parameter as read-only, evaluate, hidden, and tunable. For more information, see “Property editor”.

9 Click **OK**. Look at the block mask. Only the parameters you promoted are available to set.

Note

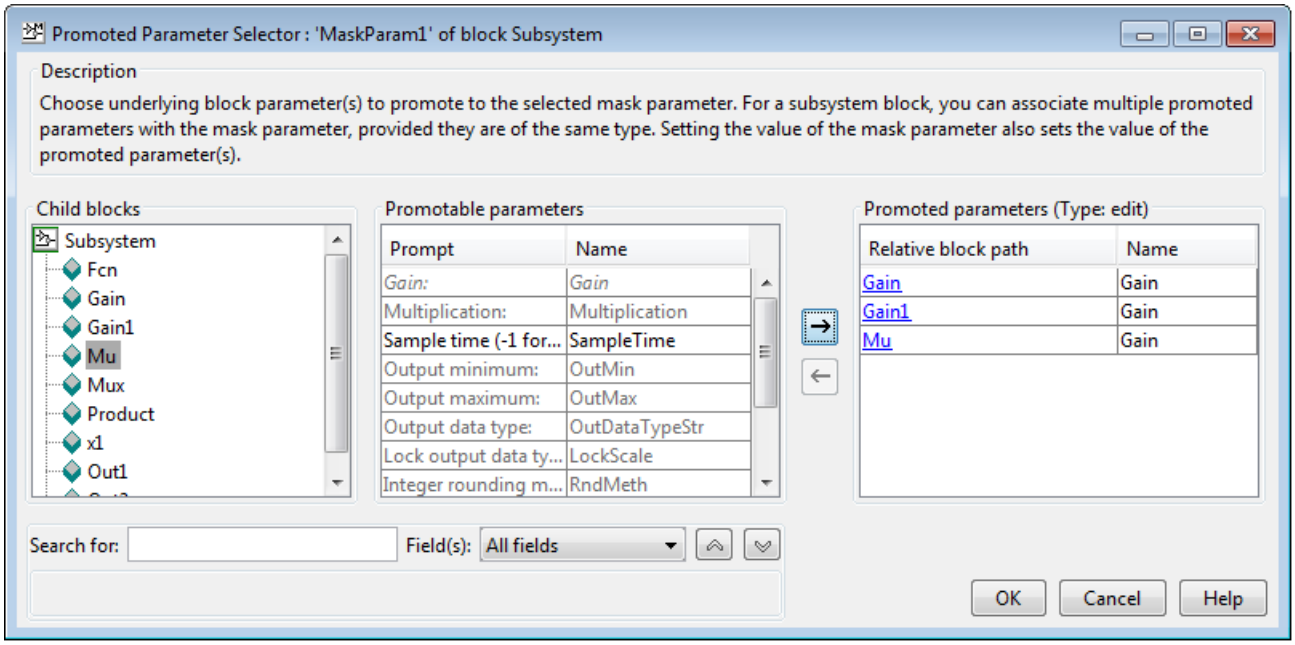
- You can use **Promote all**  to promote all parameters. **Promote all** is available for all block masks except for subsystem masks.
- To remove a promoted parameter, select the parameter and press **Delete** key.
- You cannot view or promote parameters of a nested masked or linked child block.

- Do not promote the parameters of built-in Simulink blocks, as these blocks may have internal callbacks associated with them.

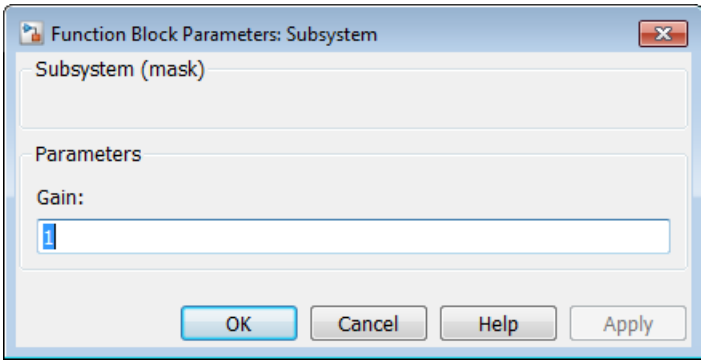
Promote Underlying Parameters to Subsystem Mask

1. Select the subsystem.
2. On the **Subsystem Block** tab, in the **Mask** group, click **Create Mask**.
3. In the **Mask Editor** dialog box, click the **Parameters & Dialog** tab.
4. In the **Controls** pane, click **Promote**.
5. In the **Property editor** pane, next to **Type options**, click  .
6. In the **Promoted Parameter Selector** dialog box, select the parameters that you want to promote.
7. To add a parameter to the **Promoted parameters** list, select a parameter from the **Promotable parameters** table and click the **Add to promoted parameter list** button  .

   You can add parameters of the same data type from different child blocks to the **Promoted parameters** list. For example, the `Gain` parameter from a different child block can be added to the **Promoted parameters** list to promote to the single `Gain` parameter on the mask.



8. Click **OK**.
9. In the **Mask Editor** dialog box, edit the prompt names for the promoted parameters and click **OK**. You cannot edit the variable names.
10. Click **OK**. Look at the block mask. Only the parameters you promoted are available to set.



Unresolved Promoted Parameter

When a promoted parameter is disconnected from the underlying block parameter, the promoted parameter is unresolved. Unresolved promoted parameters can cause the model to be erroneous as the promoted parameter cannot find the corresponding block parameter. Promoted parameters can become unresolved for any of these reasons:

- The underlying block is deleted.
- The underlying block is replaced with another block of same name but does not have the specified parameter.
- The underlying block is moved within another mask.

Best Practices

- Set the value of a promoted parameter only in the mask dialog box and not in the underlying block dialog box or from the command line.
- Parameters once promoted cannot be promoted again to any other mask.
- Do not edit the **Evaluate** attribute of the promoted parameter. This property is inherited from the block parameter.
- If you are promoting a nontunable parameter, do not edit the **Tunable** attribute.
- Parameters of a masked or linked child block cannot be viewed or promoted.
- Callbacks associated with a block parameter are promoted to the block mask and not to the subsystem mask. User-defined callbacks are sequentially executed after the dynamic dialog callback execute.

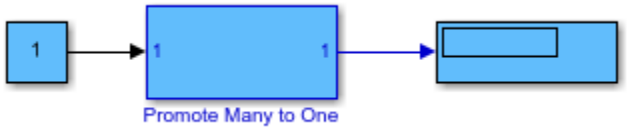
Promote Block Parameters on a Mask

You can use Parameter Promotion to promote any underlying parameter of a block either to a block mask or to a subsystem mask. This model contains a subsystem that has 3 Gain blocks (**Gain1**, **Gain2, and *Gain3**). The variable K represents the Gain parameter for these Gain blocks. You can promote only the Gain parameter of each of these Gain blocks to the block mask as a single parameter. When you do so, the parameter K is available on the mask for editing and its value will be applied to **Gain1** , **Gain2** , and **Gain3** blocks.

1. Select the Subsystem block.
2. On the **Subsystem Block** tab, in the **Mask** group, click **Create Mask/Edit Mask**.

3. In the Mask Editor dialog box, click the **Parameters & Dialog** tab.
4. In the **Controls** pane, click **Promote** .
5. In the **Property editor** pane, **Type options** field, click
6. In the **Promoted Parameter Selector** dialog box, select **Gain1** .
7. Select **Gain** from the **Promotable parameters** table and click the Add to promoted parameter list button. Similarly, add Gain parameter for **Gain2**.
8. Click **OK** .
9. In the Mask Editor dialog box, edit the prompt names for the **Gain** parameter. Here the Prompt used is **Common gain** .
10. Click **OK** to finish creating subsystem mask with many-to-one promotion.
11. Simulate the model. Notice that the value 4 is passed from the mask to the underlying block **Gain1**, **Gain2** , and **Gain3** . In this case, the output shows 64.

```
open_system('promote_block_param_to_mask');
```



See Also

More About

- “Create Block Masks”
- “Masking Fundamentals” on page 39-2
- “Parameter Interfaces for Reusable Components” on page 37-17

Control Masks Programmatically

In this section...

Simulink defines a set of parameters that help in setting and editing masks. To set and edit a mask from the MATLAB command line, you can use `Simulink.Mask` and `Simulink.MaskParameter` class methods. You can also use the `get_param` and `set_param` functions to set and edit masks. However, since these functions use delimiters that do not support Unicode (Non-English) characters it is recommended that you use methods of the `Simulink.Mask` and `Simulink.MaskParameter` class methods to control masks.

Use Simulink.Mask and Simulink.MaskParameter

Use methods of `Simulink.Mask` and `Simulink.MaskParameter` classes to perform the following mask operations:

- Create, copy, and delete masks
- Create, edit, and delete mask parameters
- Determine the block that owns the mask
- Get workspace variables defined for a mask

1 In this example the `Simulink.Mask.create` method is used to create a block mask:

```
maskObj = Simulink.Mask.create(gcb);
```

```
maskObj =
     Simulink.Mask handle
     Package: Simulink
     Properties:
                    Type: ''
             Description: ''
                    Help: ''
          Initialization: ''
          SelfModifiable: 'off'
                 Display: ''
               IconFrame: 'on'
              IconOpaque: 'on'
    RunInitForIconRedraw: 'off'
              IconRotate: 'none'
              PortRotate: 'default'
               IconUnits: 'autoscale'
              Parameters: []
  Methods, Events, Superclasses
```

2 In this example the mask object is assigned to variable `maskObj` using the `Simulink.Mask.get` method:

```
maskObj = Simulink.Mask.get(gcb)
```

```
maskObj =
    Simulink.Mask handle
```

```
Package: Simulink
Properties:
                Type: ''
         Description: ''
                Help: ''
      Initialization: ''
      SelfModifiable: 'off'
             Display: ''
           IconFrame: 'on'
          IconOpaque: 'on'
RunInitForIconRedraw: 'off'
          IconRotate: 'none'
          PortRotate: 'default'
           IconUnits: 'autoscale'
          Parameters: [1x1 Simulink.MaskParameter]
 Methods, Events, Superclasses
```

For examples of other mask operations, like creating and editing mask parameters and copying and deleting masks see `Simulink.Mask` and `Simulink.MaskParameter` .

Use get_param and set_param

The `set_param` and `get_param` functions have parameters for setting and controlling the mask. You can use these functions to set the mask of any block in the model or library based on a value passed from the MATLAB command line:

```
set_param(gcb,'MaskStyleString','edit,edit',...
'MaskVariables','maskparameter1=@1;maskparameter2=&2;',...
'MaskPromptString','Mask Parameter 1:|Mask Parameter 2:',...
'MaskValues',{'1','2'});

get_param(gcb,'MaskStyleString');

set_param(gcb,'MaskStyles',{'edit','edit'},'MaskVariables',...
'maskparameter1=@1;maskparameter2=&2;','MaskPrompts',...
{'Mask Parameter 1:','Mask Parameter 2:'},...
'MaskValueString','1|2');

get_param(gcb,'MaskStyles');
```

where

- `|` separates individual character vector values for the mask parameters.
- `@` indicates that the parameter field is evaluated.
- `&` indicates that the parameter field is not evaluated but assigned as a character vector.

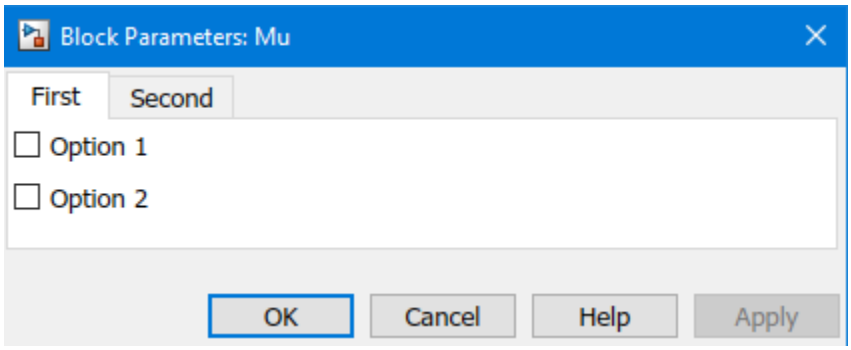
Note

- When you use `get_param` to get the `Value` of a mask parameter, Simulink returns the value that was last applied using the mask dialog. Values that you have entered into the mask dialog box but not applied are not reflected when you use the `get_param` command.
- To specify the value of a mask parameter programmatically, it is recommended to use `set_param` command on the **mask parameter** instead of using `set_param` on **MaskValues**.

To control the mask properties programmatically for a release before R2014a, see Mask Parameters.

Programmatically Create Mask Parameters and Dialogs

This example shows how to create this simple mask dialog, add controls to the dialog, and change the properties of the controls.

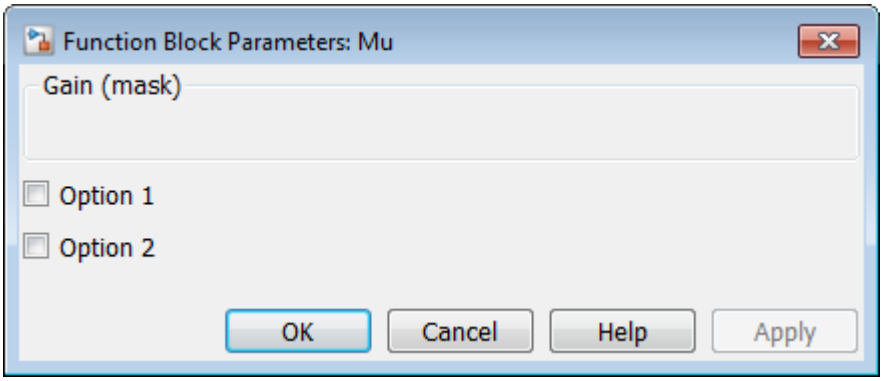


1. Create the mask for a block you selected in the model.

   ```
   maskobj = Simulink.Mask.create(gcb);
   ```

2. To customize the dialog and to use tabs instead of the default group, remove the **Parameters** group box.

   ```
   maskobj.removeDialogControl('ParameterGroupVar');
   ```

   Simulink preserves the child dialog controls– the two check boxes in this example– even when you delete the `ParametersGroupVar` group surrounding them. These controls are parameters, that cannot be deleted using dialog control methods.

   You can delete parameters using methods such as `removeAllParameters`, which belongs to the `Simulink.Mask` class.

3. Create a tab container and get its handle.

   ```
   tabgroup = maskobj.addDialogControl('tabcontainer','tabgroup');
   ```

4. Create tabs within this tab container.

   ```
   tab1 = tabgroup.addDialogControl('tab','tab1');
   tab1.Prompt = 'First';
   maskobj.addParameter('Type','checkbox','Prompt','Option 1',...
   ```

```
  'Name','option1','Container','tab1');
maskobj.addParameter('Type','checkbox','Prompt','Option 2',...
  'Name','option2','Container','tab1');

tab2 = tabgroup.addDialogControl('tab', 'tab2');
tab2.Prompt = 'Second';
tab3 = tabgroup.addDialogControl('tab','tab3');
tab3.Prompt = 'Third (invisible)';
```

Make the third tab invisible.

```
tab3.Visible = 'off'

tab3 =

  Tab with properties:

              Name: 'tab3'
            Prompt: 'Third (invisible)'
           Enabled: 'on'
           Visible: 'on'
     DialogControls: []
```

You can change the location and other properties of the parameters on the dialog by using the `Simulink.dialog.Control` commands.

For example, to change the dialog layout options, consider a Gain block with a Popup parameter named Parameter2 added. To set the dialog layout options of the parameter, you can use an instance of `Simulink.dialog.parameter.Popup` class. The following code shows how to set the prompt location in dialog layout:

```
a = Simulink.Mask.get('testmodel/Gain');
d = a.Parameters(2).DialogControl;
```

This lists all the properties of the popup parameter 'Parameter2':

```
d =

  Popup with properties:

                Name: 'Parameter2'
      PromptLocation: 'top'
                 Row: 'new'
    HorizontalStretch: 'on'
              Tooltip: 'Test'
```

Now, to set the `PromptLocation` property, use the command:

```
d.PromptLocation = 'left'
```

This sets the `PromptLocation` as `'left'`. The available values are `'left'` and `'top'`. The output confirms the change of the `PromptLocation` property value to `left`:

```
d =

  Popup with properties:

                Name: 'Parameter2'
```

```
       PromptLocation: 'left'
                  Row: 'new'
    HorizontalStretch: 'on'
              Tooltip: 'Test'
```

For more information on dialog controls and their properties, see `Simulink.dialog.Control`.

See Also

More About

- “Create Block Masks”
- “Masking Fundamentals” on page 39-2
- “Initialize Mask” on page 39-20

Pass Values to Blocks Under the Mask

A masked block can pass values to the block parameters under the mask. The underlying blocks use the passed values during simulation to execute the block logic.

A masked block has variables associated with mask parameters. These variables are stored in the mask workspace for a model and can correspond to a block parameter under the mask. When such a block is evaluated, the block variables look for matching values in the mask workspace to get a value.

The mapping of variables from the mask workspace to the base workspace must be correct. A correct mapping ensures that the right block variable is assigned the value that is passed from the mask.

Use any of these options to pass values to blocks under the mask:

- Parameter promotion (recommended)
- Mask initialization
- Referencing block parameters using variable names (For the Edit Parameter only)

Parameter Promotion

When you promote a block parameter to its mask, the block parameter becomes accessible from the mask dialog box, allowing you to pass a value for the block parameter. Parameter promotion ensures correct mapping of parameter values and is a recommended way to pass values to the block from the mask dialog box. Note that, when you promote parameters of an in-built block, the internal callbacks associated with these parameters are also inherited in the new parameter. For more information on promoting a parameter, see “Promote Parameter to Mask” on page 39-23.

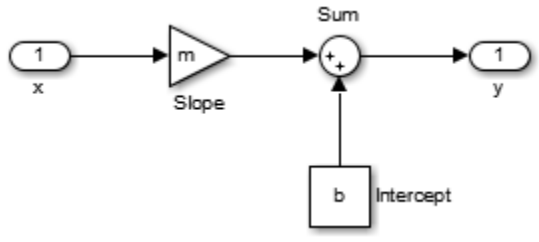
Mask Initialization

You can use MATLAB code in the Initialization pane of the Mask Editor to assign or pass values to the block parameters under the mask. You can assign a fixed value to a block parameter, specify an acceptable range for the input values, or specify a value for the child block. For more information, see “Initialize Mask” on page 39-20.

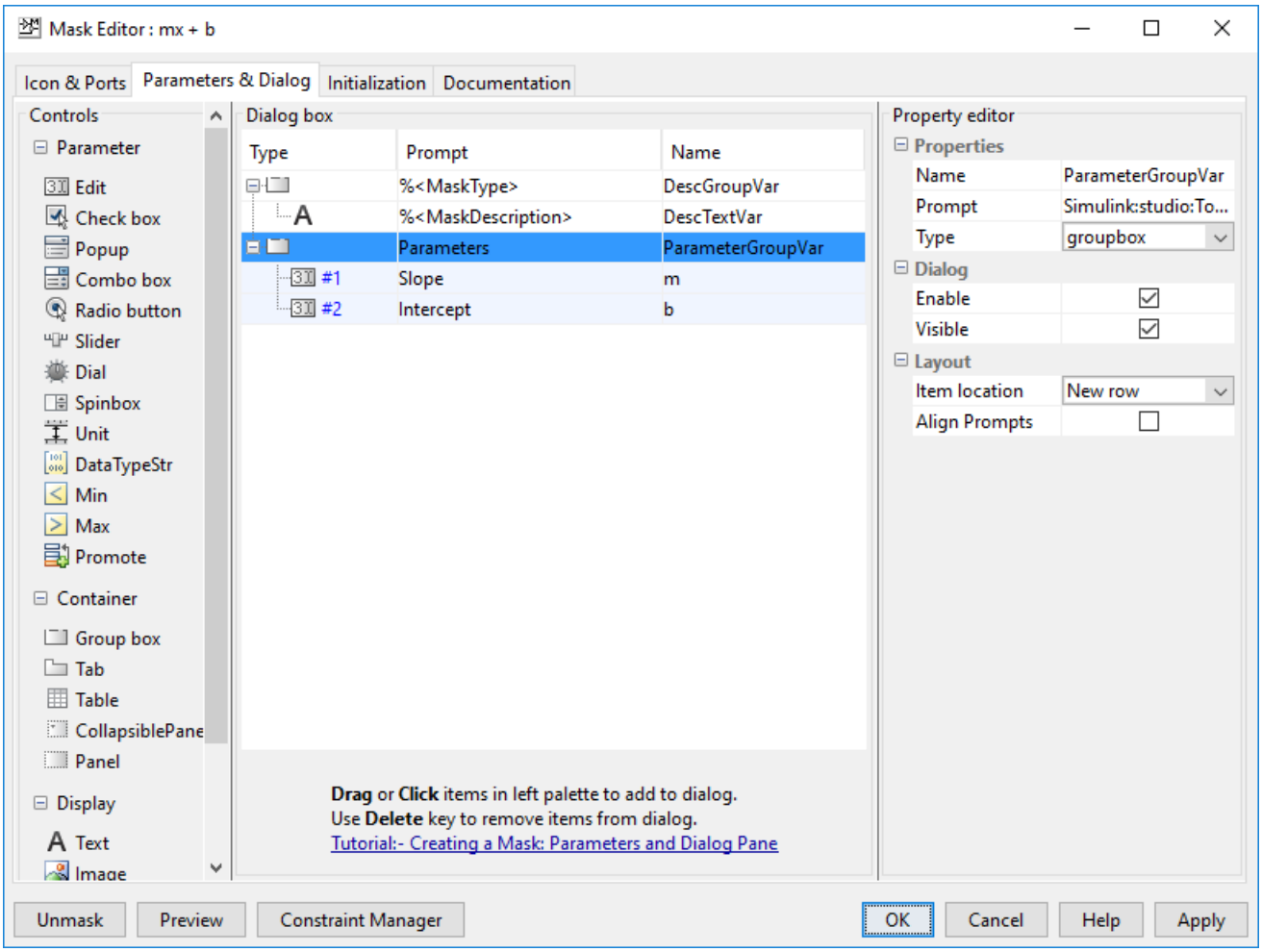
Referencing Block Parameters Using Variable Names

You can add an Edit parameter to the mask dialog box and pass values to the block parameters through it. The values that you provide for the Edit parameter in the mask dialog box automatically becomes associated with the block parameter, by using the techniques described in “Symbol Resolution” on page 67-127.

Consider the model `masking_example`, which contains a masked Subsystem block and governs the equation `y = mx + b`. Here, `m` and `b` are variables controlling the slope and intercept of the equation and are associated with the Gain and the Constant block, respectively.



The variables `m` and `b` are assigned to the mask parameters **Slope** and **Intercept**, respectively, as parameter names in the Mask Editor.



When you type values for **Slope** and **Intercept** in the mask dialog box, these values are internally assigned to the variables `m` and `b`. When the model is simulated, the Gain block and the Constant block search for numeric values of `m` and `b` and apply them to resolve the equation `y = mx + b`.

See Also

More About

- “Create Block Masks”
- “Masking Fundamentals” on page 39-2
- Creating a Mask: Masking Fundamentals (3 min, 46 sec)

Mask Linked Blocks

In this section...

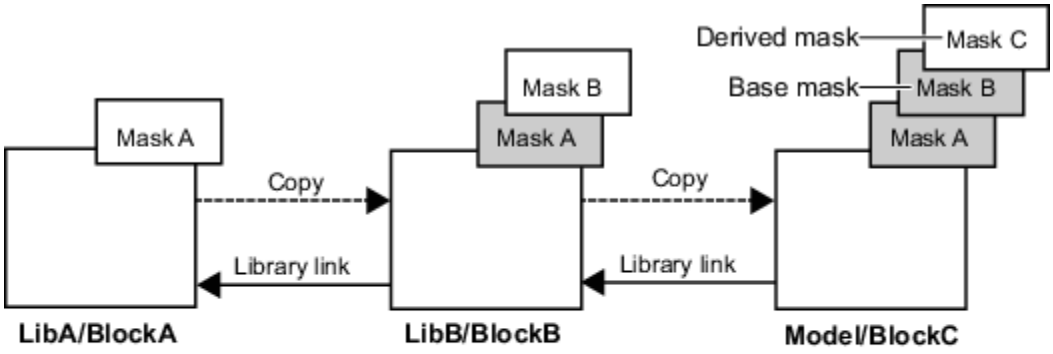
Simulink libraries can contain blocks that have masks. An example of this type of block is the Ramp block. These blocks become library links when copied to a model or another library. You can add a mask on this linked block. If this linked block is in a library and copied again, you can add another mask to this new linked block thus creating a stack of masks. Masking linked blocks allows you to add a custom interface to the link blocks similar to other Simulink blocks.

You can also apply a mask to a block, then include the block in a library. Masking a block that is later included in a library requires no special provisions. For more information, see “Create a Custom Library” on page 41-2.

The block mask that is present as part of the library is the base mask. A derived mask is the one created on top of the base mask.

For example, in the figure, Library A contains Block A, which has a Mask A. Block A is copied to Library B, and Mask B is added to it. When Block A is copied to Library B, a library link from Library B to Library A is created.

Block B is then copied to a model, and Mask C is added to it. This creates a library link from Block C to Library B. Block C now has Mask A, Mask B, and Mask C. Mask C is the derived mask and Mask B is the base mask.



For Block C:

- Mask parameter names are unique in the stack.
- You can set mask parameters for Mask B and Mask C.

- Mask B and Mask C inherit `MaskType` and `MaskSelfModifiable` parameters from Mask A.
- Mask initialization code for Mask C executes first, followed by Mask B and Mask A.
- Variables are resolved starting from the mask immediately above the current mask in the stack. If the current mask is the top mask, it follows the regular variable resolution rules.

Creating or changing a library block mask changes the block interface in all models that access the block using a library reference, but has no effect on instances of the block that exist as separate copies.

To view related example, see Use self-modifying library masks.

Guidelines for Mask Parameters

- You cannot use same names for the mask parameters. The exception is the `Promote` type mask parameter, for which the name is inherited and is the same as that of the parameter promoted to it.
- You cannot set mask parameters for masks below the base mask. Mask parameters for masks below the base mask are inherited from the library.

Mask Behavior for Masked, Linked Blocks

The following are some of the behaviors that are important to understand about masked, linked blocks.

- The `MaskType` and the `MaskSelfModifiable` parameters are inherited from the base mask.
- The mask display code for the derived mask executes first, followed by the display code for the masks below it until we come across a mask whose `MaskIconFrame` parameter is set to `opaque`.
- The mask initialization code for the derived mask executes first, followed by the initialization code for the masks below it.
- Variables are resolved starting from the mask immediately above the current mask in the stack. If the current mask is the top mask, the regular variable resolution rules apply.
- When you save a Simulink model or library containing a block with multiple masks, using **Save** > **Export Model to** > **Previous Version** on the **Simulation** tab, the `Sourceblock` parameter is modified to point to the library block having the bottom-most mask.
- The following occurs when you disable, break, or reestablish links to libraries:
  - If you disable the link to the library block, the entire mask hierarchy is saved to the model file so that the block can act as a standalone block.
  - If you break the link to the library block, the block becomes a standalone block.
  - If you reestablish the link after disabling by doing a restore, all changes to the mask are discarded. If you mask subsystems, you must reestablish the link after disabling by doing a push. When you do a push, subsystem changes are pushed to the library block and top mask changes are pushed to the immediate library.

Mask a Linked Block

Step 1: Create Custom Library with Masked Linked Block

1. In the **Simulink Library Browser**, click the arrow next to  and select **New Library**.
2. Open the Ramp block in the Library editor window.
3. Select the Ramp block and on the **Block** tab,in the **Mask** group, click **Create Mask**.

   The **Mask Editor** opens.
4. In the **Icon drawing commands** section of the **Icons & Ports** pane, type:

   ```
   plot ([0:10],[0,1:10])
   ```
5. In the **Parameter & Dialog** pane, select  **Promote** to promote the `Slope` and `Initial Output` parameters.
6. Click **OK**.
7. Rename the block to Derived Ramp block.

Step 2: Add a Mask to the Masked, Link Block

1. In the **Simulink Library Browser**, click the arrow next to  and select **New Model**. The Model editor window opens.
2. Drag the Derived Ramp block from the Library editor to the Model editor.

   The Derived Ramp block in the model has multiple masks on it. You can set parameters of the derived mask.

Step 3: View Masks Below the Top Mask

- Right-click the Derived Ramp block in the model and select **Mask > View Base Mask**. This opens the **Mask Editor** displaying the base mask definition.

See Also

More About

- “Create Block Masks”
- “Linked Blocks” on page 41-10

Dynamic Mask Dialog Box

In this section...

You can create dialogs for masked blocks whose appearance changes in response to user input. Features of masked dialog boxes that can change in this way include:

- Visibility of parameter controls — Changing a parameter can cause the control for another parameter to appear or disappear. The dialog expands or shrinks when a control appears or disappears, respectively.
- Enabled state of parameter controls — Changing a parameter can cause the control for another parameter to be enabled or disabled for input. A disabled control is grayed to indicate visually that it is disabled.
- Parameter values — Changing a mask dialog box parameter can cause related mask dialog box parameters to be set to appropriate values.

Note Mask parameter addition, deletion, or modification is restricted from mask callback.

Creating a dynamic masked dialog box entails using the Mask Editor with the `set_param` command. Specifically, you use the Mask Editor to define parameters of the dialog box, both static and dynamic. For each dynamic parameter, you enter a callback function that defines how the dialog box responds to changes to that parameter (see “Execute Callback Code” on page 39-15). The callback function can in turn use the `set_param` command to set mask parameters that affect the appearance and settings of other controls on the dialog box (see “Create Dynamic Mask Dialog Box” on page 39-39). Finally, you save the model or library containing the masked subsystem to complete the creation of the dynamic masked dialog box.

To view related example, see Create dynamic mask dialog boxes.

Show Parameter

The selected parameter appears on the mask dialog box only if this option is checked (the default).

Enable Parameter

Clearing this option grays the prompt of the selected parameter and disables the edit control of the prompt.

Create Dynamic Mask Dialog Box

This example shows how to create a mask dialog blocks whose appearance changes in response to your input.

You can set two parameters using this mask dialog box. The first parameter is a popup menu through which you select one of three gain values: `2`, `5`, or `User-defined`. Depending on the value that you select in this popup menu, an edit field for specifying the gain appears or disappears.

1. Select a subsystem and on the **Subsystem Block** tab, in the **Mask** group, click **Create Mask**.
2. Select the **Parameters & Dialog** pane on the Mask Editor.
3. Drag and drop a **Popup** parameter and select it in the **Dialog box** pane.
   a. In the **Prompt** field, enter `Gain`.
   b. In the **Name** field, enter `gainpopup`.
   c. In the Property editor pane, clear **Evaluate** so that Simulink uses the literal values you specify for the popup.
   d. In the **Type options** field, click the **Edit** button to enter these three values in the Popup Options dialog box:

      ```
      2
      5
      User-defined
      ```
4. Enter this code in the **Dialog callback** field:

```
% Get the mask parameter values. This is a cell
%   array of character vectors.
maskStr = get_param(gcb,'gainpopup');

% The pop-up menu is the first mask parameter.
%   Check the value selected in the pop-up
if strcmp(maskStr(1),'U'),

    % Set the visibility of both parameters on when
    %   User-defined is selected in the pop-up.

    set_param(gcb,'MaskVisibilities',{'on';'on'}),

else

    % Turn off the visibility of the Value field
    %   when User-defined is not selected.

    set_param(gcb,'MaskVisibilities',{'on';'off'}),

    % Set the character vector in the Values field equal to the
    % character vector selected in the Gain pop-up menu.

    %maskStr{2}=maskStr{1};
    set_param(gcb,'editvalue',maskStr);
end
```

5. Drag and drop an **Edit** parameter and select it in the **Dialog box** pane.
   a. In the **Prompt** field, enter `Value`.
   b. In the **Name** field, enter `editvalue`.
   c. In the Property editor pane, clear **Visible** so that Simulink turns off the visibility of this property by default.
6. Click **Apply**.

7 To open the mask dialog box, double-click the masked subsystem.

If you select `2` or `5` as the **Gain**, Simulink hides the **Value**. If you select `User-defined` as the **Gain** the **Value** is visible.

Set Up Nested Masked Block Parameters

If lower-level masked subsystems reference symbols defined by higher-level masked subsystems and you try to set parameters of blocks in lower-level masked subsystems, unresolved symbol errors can occur. Therefore, avoid using `set_param` commands to set parameters of blocks residing in masked subsystems that reside in the masked subsystem being initialized. Trying if lower-level masked subsystems reference symbols defined by higher-level masked subsystems.

Suppose, for example, a masked subsystem A contains masked subsystem B, which contains Gain block C, whose Gain parameter references a variable defined by B. Suppose also that subsystem A's initialization code contains this command:

```
set_param([gcb '/B/C'], 'SampleTime', '-1');
```

Simulating or updating a model containing A causes an unresolved symbol error.

See Also

More About

- “Create Block Masks”
- “Dynamic Masked Subsystem” on page 39-42

Dynamic Masked Subsystem

In this section...

Allow Library Block to Modify Its Contents

This check box is enabled only if the masked subsystem resides in a library. Checking this option allows the block initialization code to modify the contents of the masked subsystem (that is, it lets the code add or delete blocks and set the parameters of those blocks). Otherwise, an error is generated when a masked library block tries to modify its contents in any way. To set this option at the MATLAB prompt, select the self-modifying block and enter the following command.

```
set_param(gcb, 'MaskSelfModifiable', 'on');
```

Then save the block.

Create Self-Modifying Masks for Library Blocks

You can create masked library blocks that can modify their structural contents. These self-modifying masks allow you to:

- Modify the contents of a masked subsystem based on parameters in the mask dialog box or when the subsystem is initially dragged from the library into a new model.
- Vary the number of ports on a multiport S-Function block that resides in a library.

Simulink runs the mask-initialization code for a self-modifiable library block when you load the block. If the mask-initialization code controls the number of input/output ports for a block, mark the block as self-modifiable. Otherwise, the mask-initialization code will not execute and will not set the right number of ports, which will disconnect the block.

Creating Self-Modifying Masks Using the Mask Editor

To create a self-modifying mask using the Mask Editor:

1. Unlock the library (see “Lock and Unlock Libraries” on page 41-6).
2. Select the block in the library.
3. On the **Block** tab, in the **Mask** group, click **Edit Mask**. The Mask Editor opens.
4. In the Mask Editor **Initialization** pane, select the **Allow library block to modify its contents** option.
5. Enter the code that modifies the masked subsystem in the mask **Initialization** pane.

   Do not enter code that structurally modifies the masked subsystem in a dialog parameter callback (see “Add Mask Code” on page 39-14). Doing so triggers an error when you edit the parameter.
6. Click **Apply** to apply the change or **OK** to apply the change and close the Mask Editor.

7 Lock the library.

Creating Self-Modifying Masks from the Command Line

To create a self-modifying mask from the command line:

1 Unlock the library using the following command:

```
set_param(gcs,'Lock','off')
```

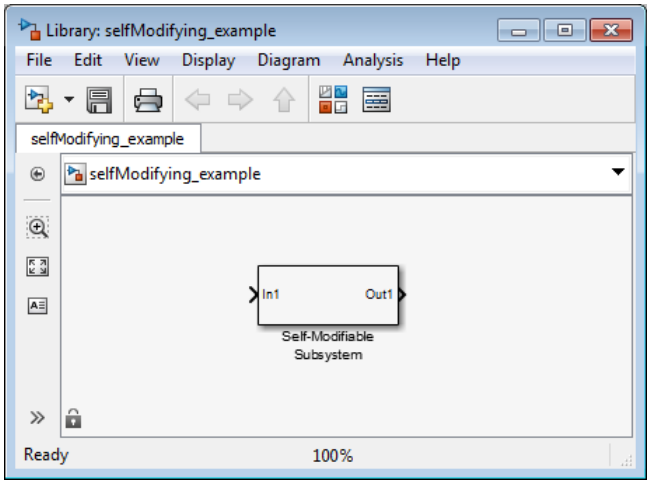
2 Specify that the block is self-modifying by using the following command:

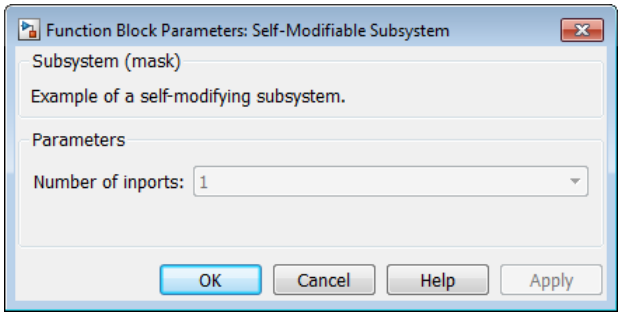
```
set_param(block_name,'MaskSelfModifiable','on')
```

where `block_name` is the full path to the block in the library.

Create Self-Modifying Mask

The library `selfModifying_example` contains a masked subsystem that modifies its number of input ports based on a selection made in the subsystem mask dialog box.

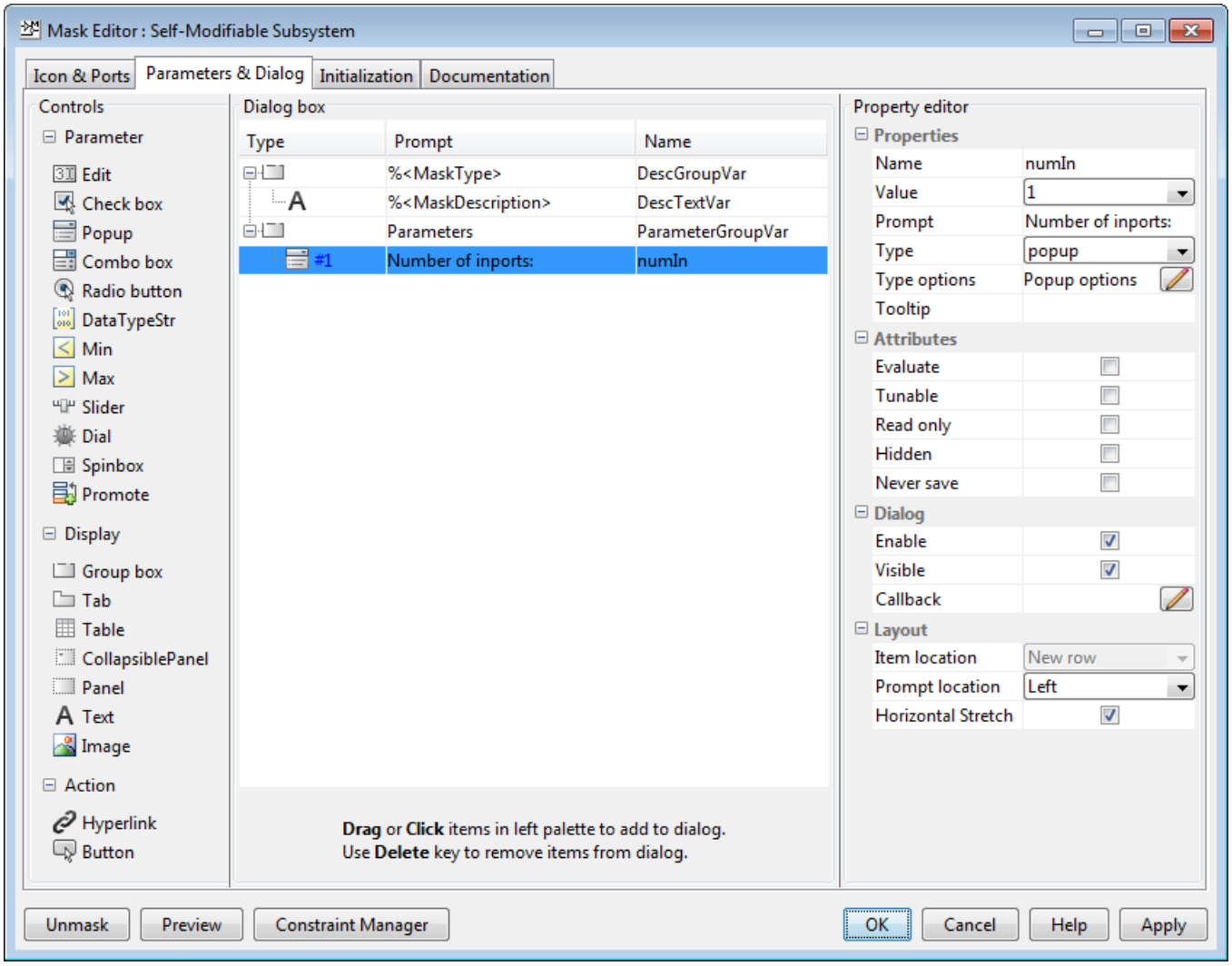




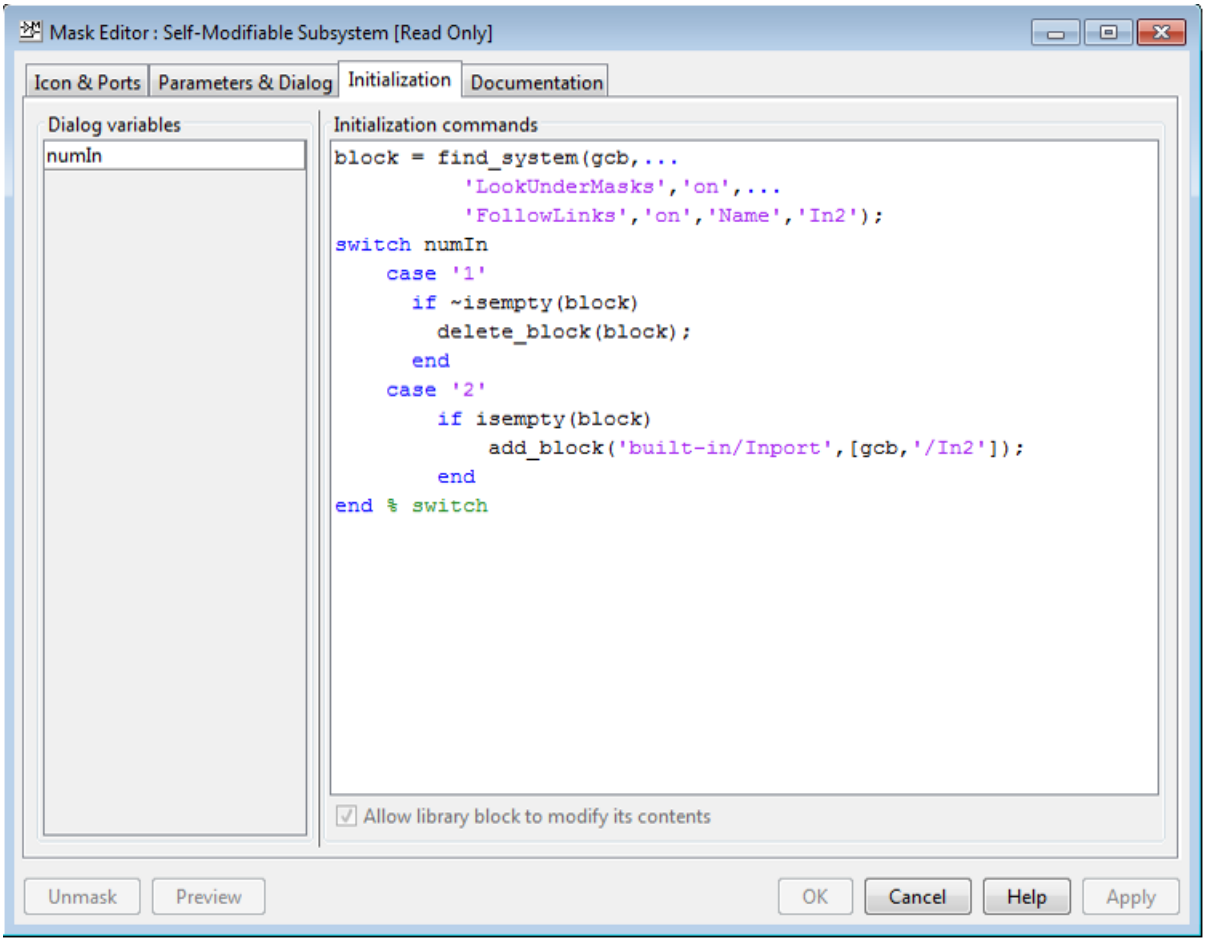
1 In the Library window, on the **Library** tab, click **Locked Library** to unlock the library.

2 On the **Subsystem Block** tab, in the **Mask** group, click **Edit Mask**. The Mask Editor opens.

3 The Mask Editor **Parameters & Dialog** pane defines a parameter `numIn` that stores the value for the **Number of inports** option. This mask dialog box callback adds or removes Input ports inside the masked subsystem based on the selection made in the **Number of inports** list.



4 To allow the dialog box callback to function properly, the **Allow library block to modify its contents** option on the Mask Editor **Initialization** pane is selected. If this option is not selected, copy of the library block could not modify their structural contents. Also, changing the selection in the **Number of inports** list would produce an error.



Passing Mask Parameter Values from Parent Subsystem to Child Block

You can pass mask parameter values from a parent subsystem to a child block in three ways:

- Using parameter promotion.
- Using the mask initialization code. This is done by using the `set_param` command on the child block.
- Having the child block parameters reference the parent mask parameter name. This is applicable only for the edit parameters.

See Also

More About

- “Create Block Masks”
- “Create a Simple Mask” on page 39-6
- “Initialize Mask” on page 39-20
- “Mask Linked Blocks” on page 39-36

- Self-Modifying Interface Connector

Debug Masks That Use MATLAB Code

In this section...

Code Written in Mask Editor

Debug initialization commands and parameter callbacks entered directly into the Mask Editor by:

- Removing the terminating semicolon from a command to echo its results to the MATLAB Command Window.
- Placing a `keyboard` command in the code to stop execution and give control to the keyboard.

Tip To stop debugging the mask initialization callback code when an error is encountered, use the command `dbstop if caught error`.

Code Written Using MATLAB Editor/Debugger

Note You cannot debug icon drawing commands using the MATLAB Editor/Debugger. For information on icon drawing commands syntax, see “Icon drawing commands”.

Debug initialization commands and parameter callbacks written in files using the MATLAB Editor/Debugger in the same way that you would with any other MATLAB program file.

When debugging initialization commands, you can view the contents of the mask workspace. However, when debugging parameter callbacks, you can only access the base workspace of the block. If you need the value of a mask parameter, use `get_param`.

See Also

More About

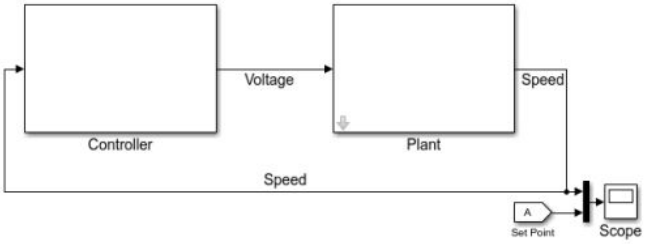
- “Initialize Mask” on page 39-20
- “Mask Callback Code” on page 39-14

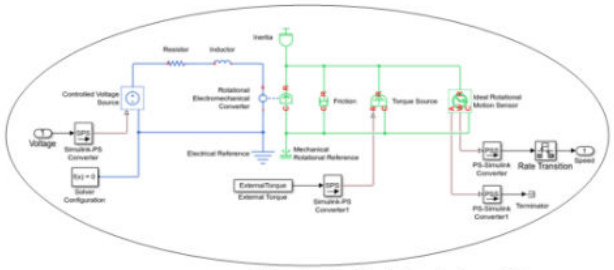
Introduction to System Mask

A model consists of multiple blocks, with each block containing its own parameter and block dialog box. Simulink enables you to mask a model. By masking a model you encapsulate the model to have its own mask parameter dialog box. You can customize the mask parameter dialog box. When you mask a model, the model arguments become the mask parameters. Referencing a masked model helps in having a better user interface for a model with ease of controlling the model parameters through the mask.

When you reference a masked model from a Model block, a mask is generated automatically on the Model block. The generated mask on the Model block is a copy of the model mask that it is referencing. You can reference a masked model from multiple Model block instances.

Consider a model that represents the DC motor equation. `Plant` in this model is a Model block that references a masked model, simplifying the user interface.





Components of 'Plant' Model block (Masked Model)

The `Plant` block contains the same mask as that of the masked model and the mask is uneditable. The mask can only be edited from the Mask Editor dialog box of the masked model.

See Also

More About

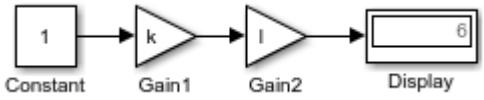
- “Create and Reference a Masked Model” on page 39-49
- “Control Model Mask Programmatically” on page 39-54
- “Masking Fundamentals” on page 39-2

Create and Reference a Masked Model

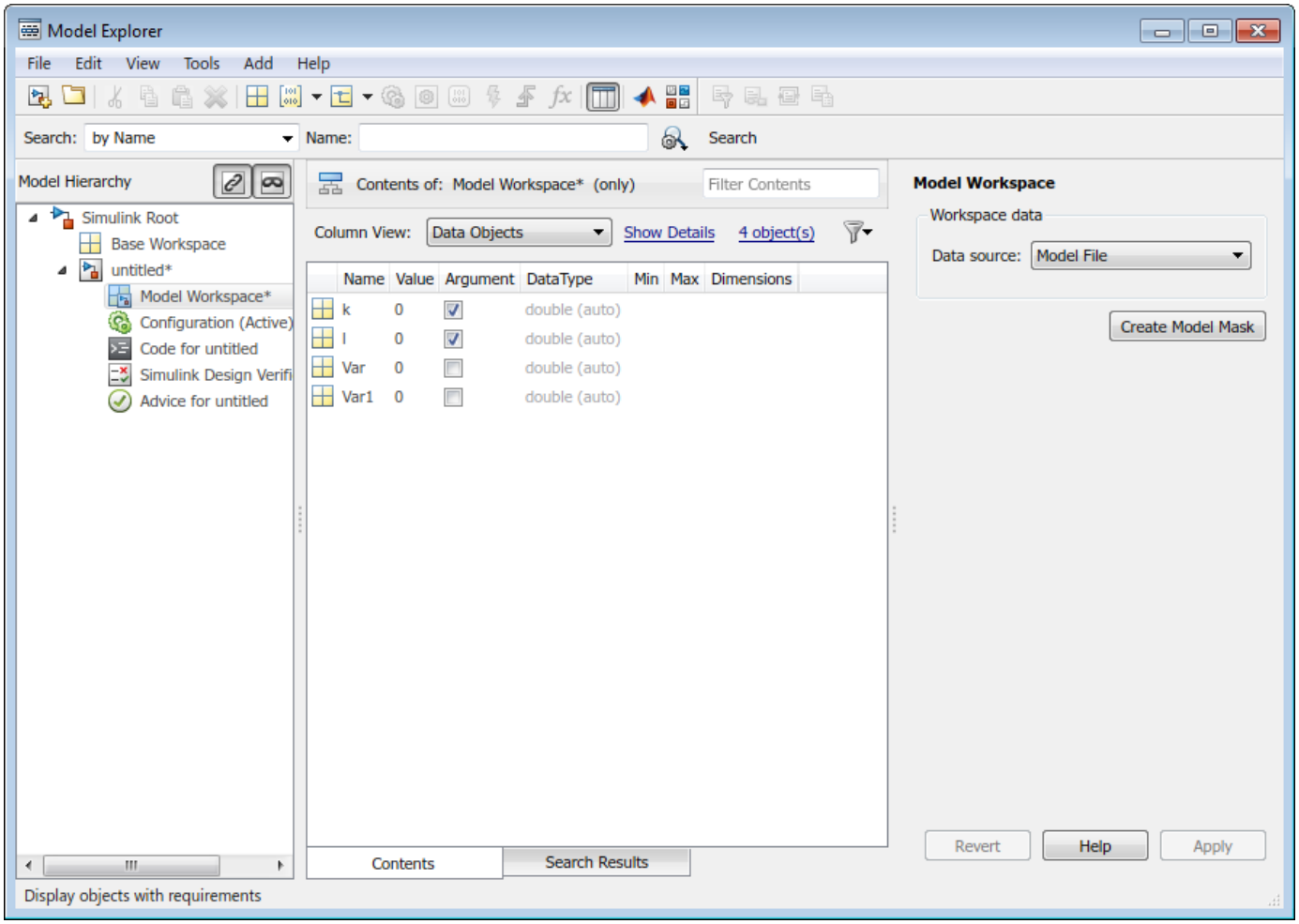
This example shows how to mask a model and reference the masked model from the Model block.

Step 1: Define Mask Arguments

1. Open the model in Simulink. For example, consider a simple model containing two Gain blocks, a Constant block, and a Display block.



2. On the **Modeling** tab, under **Design**, click **Model Workspace**. The Model Explorer dialog box opens.
3. Select **Add** > **MATLAB Variable**. A variable of data type `double` is created in the Model Workspace.
4. Select the **Argument** check box corresponding to the MATLAB variables to make it a model argument, for example, `k` and `l`.



Step 2: Create Model Mask

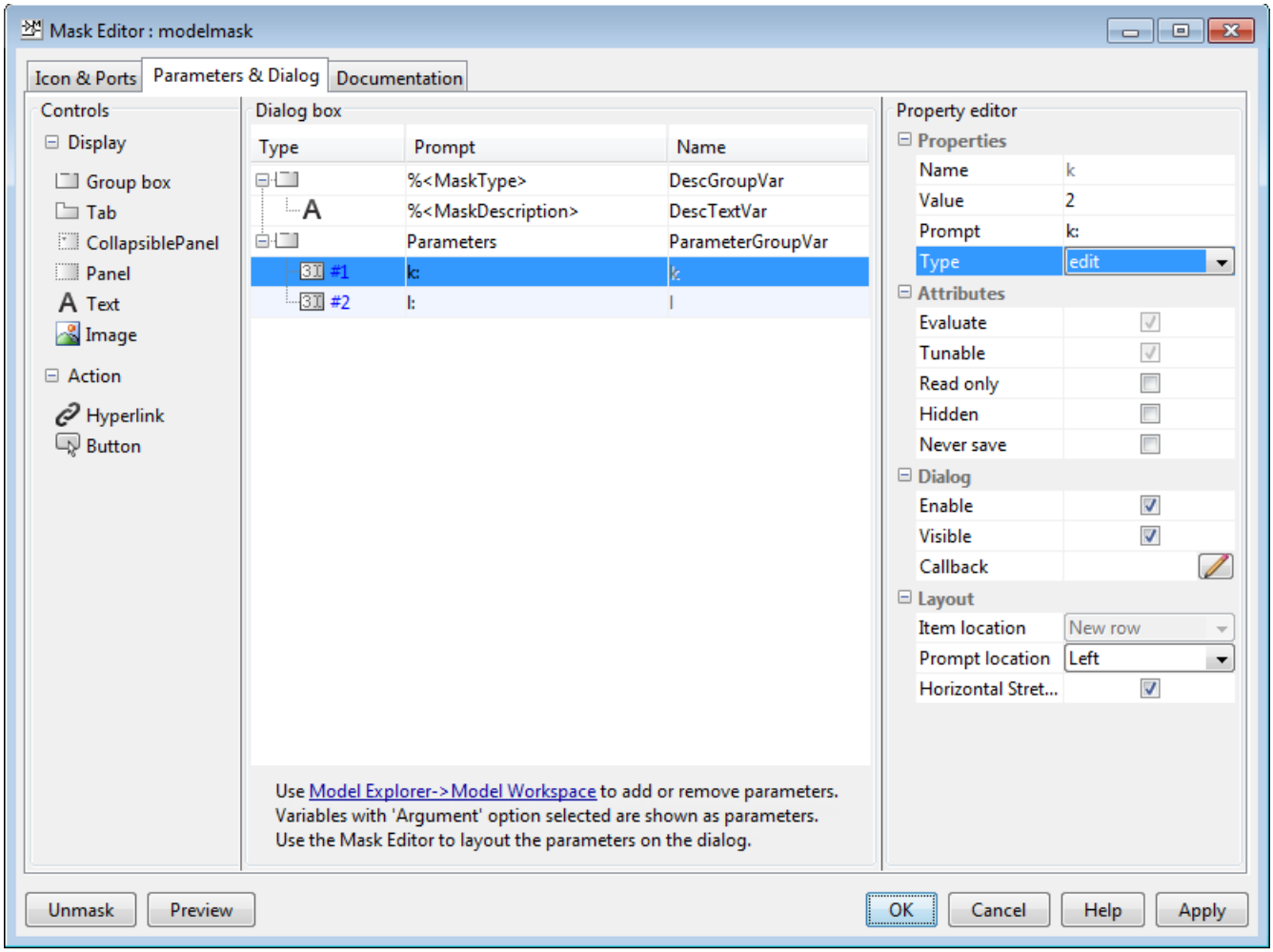
1 In the **Model Workspace** pane, click **Create Model Mask**.

Alternatively, in Simulink, on the **Modeling** tab, under **Component**, click **Create Model Mask**, or right-click the model, and select **Mask > Create Model Mask**.

The Mask Editor dialog box opens.

2 Click the **Parameters & Dialog** tab. The model arguments that you select in Model Explorer appear in the Mask Editor dialog box as mask parameters.

Tip Ensure that the model arguments you have selected in the Model Explorer dialog box are added as block parameters in the model. For example, the arguments `k` and `l` are passed to `Gain A` and `Gain B` blocks, respectively.



Note The Mask Editor dialog box for model mask does not contain the **Initialization** tab. Initialization code can alter the model and other model reference blocks, and thus affect the simulation results.

3 Select a mask parameter (`k` or `l`) on the **Dialog box** pane and edit its properties in the Property editor, as required. For example, you can change the prompt name, parameter type, value, or orientation.

By default, the **Edit** parameter type is assigned to a model mask parameter. You can change the parameter type by editing the **Type** property in the **Property editor** section.

Note

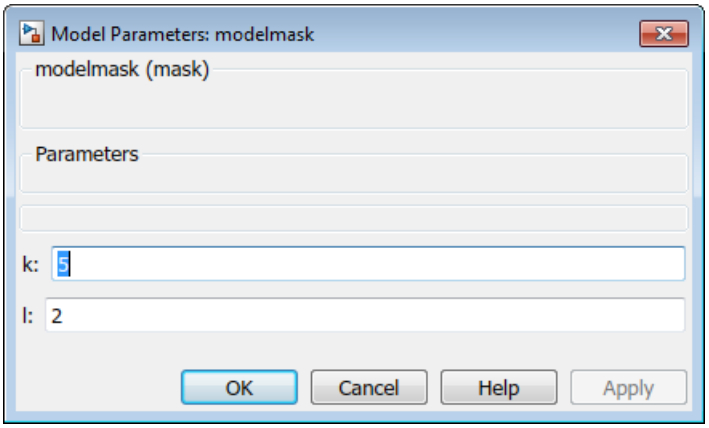
- Simulink supports only the **Edit**, **Slider**, **Dial**, and **Spinbox** parameter types for model mask.
- Model mask supports all types of display and action controls.

4 Click **OK**. The Simulink model is now masked and contains the model arguments as the mask parameter.

5 Save the model.

Step 3: View Model Mask Parameters

1 To view the mask parameter dialog box, on the **Block** tab, click **Mask Parameters**.



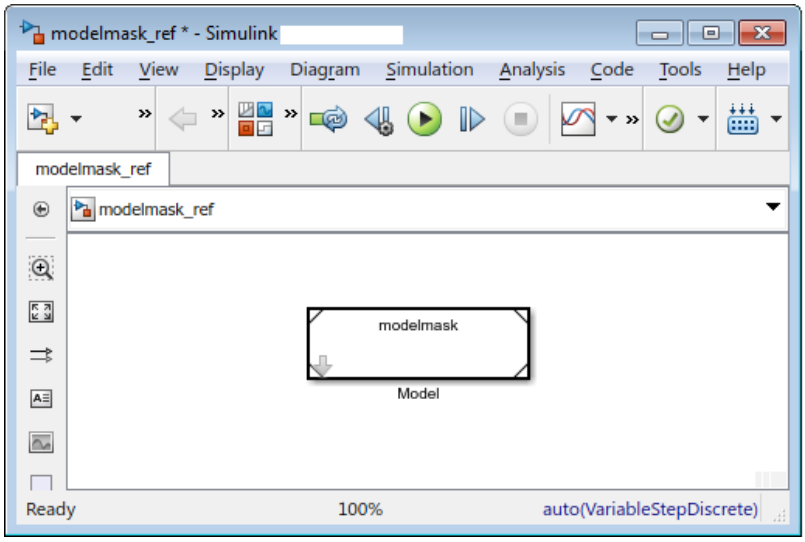
Tip To edit the model mask parameters, on the **Block** tab, click **Edit Mask**.

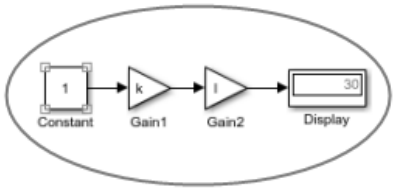
2 Save the masked model.

Step 4: Reference Masked Model

1 Open a blank model in Simulink and add the Model block from the library.

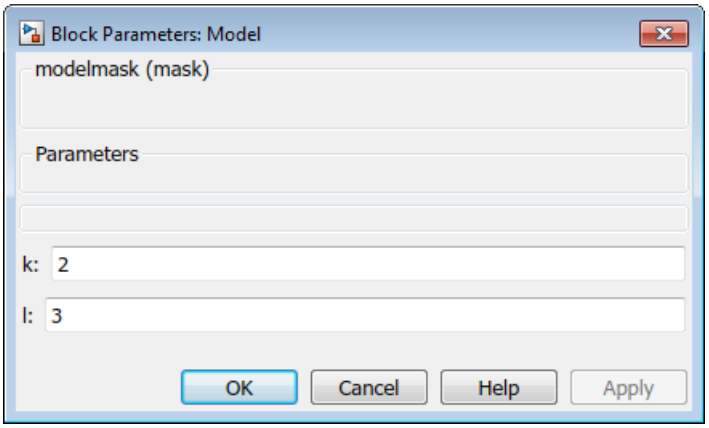
2 To reference the masked model from the Model block, specify the name of the masked model as the **Model name** in the **Block parameter** dialog box.



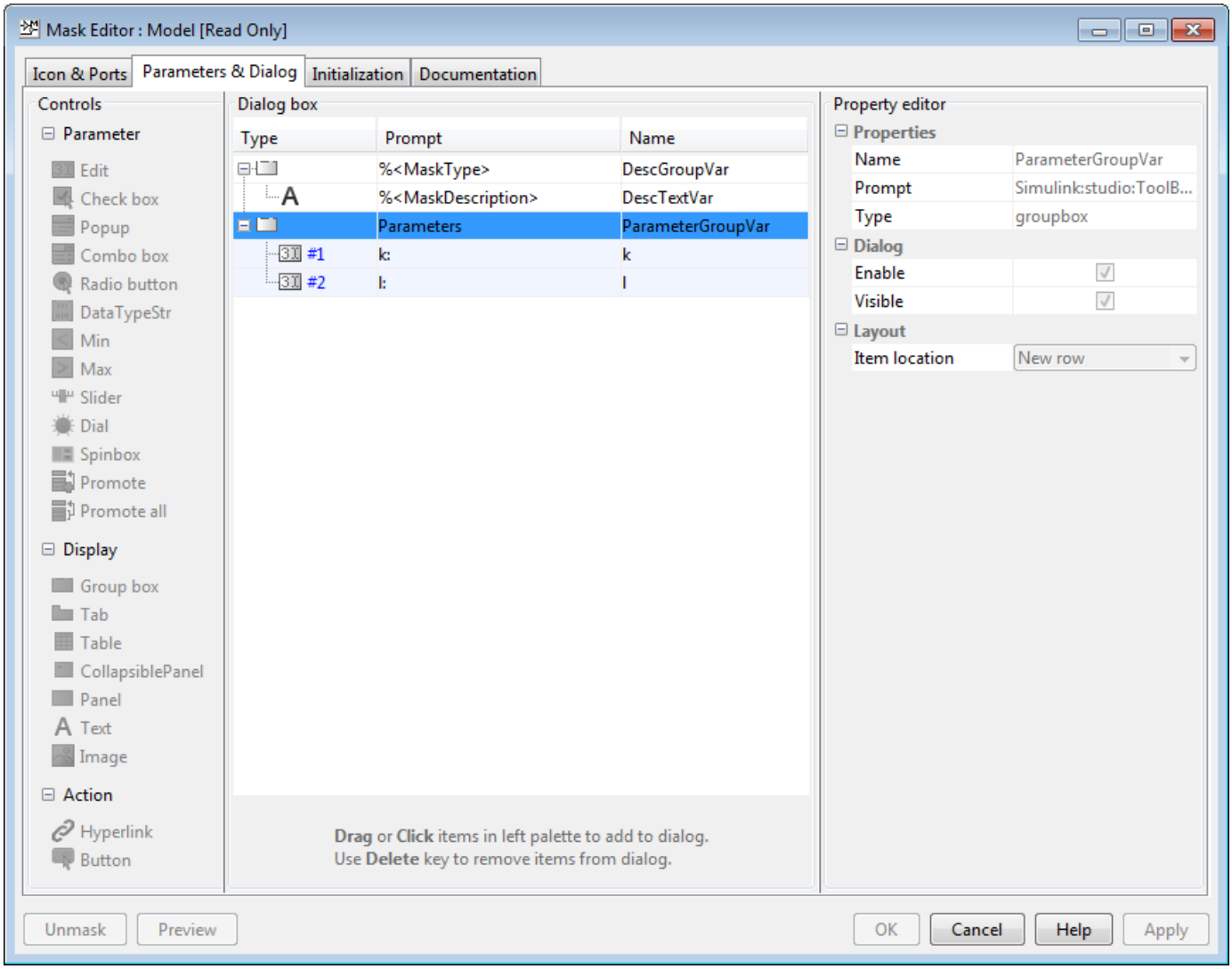


Masked model referenced by the Model block

3. To view the parameter dialog box of the referenced model, right-click the Model block, and in the context menu, click **Mask > Mask Parameters**. Alternatively, double-click the Model block.
4. Type 2 and 3 as the parameter values for `k` and `l` respectively.



5. Click **OK**.
6. Simulate the model and view the result on the display block.
7. To view the referenced model from the Model block, click **Mask > Look Under Mask**.
8. To view the mask, select the Model block and click **Mask > View Mask**. The **Mask Editor** dialog box opens. The Mask Editor dialog box displays the uneditable mask parameters of the referenced model.



See Also

More About

- “Introduction to System Mask” on page 39-48
- “Control Model Mask Programmatically” on page 39-54
- “Masking Fundamentals” on page 39-2

Control Model Mask Programmatically

Simulink defines a set of parameters to configure and edit a model mask.

Note Adding, removing, and renaming parameters on a model mask using these methods is not supported:

- `addParameter`
- `removeParameter`
- `removeAllParameters`
- `MaskParameter.Name`

Simulink.Mask.create

Use the `Simulink.Mask.create` method to create mask on a model. The syntax to mask a model is,

- Using the model name:

  ```
  Simulink.Mask.create(ModelName)
  ```

- Using the model handle

  ```
  ModelHandle = get_param(gcs,'Handle') %To get the model handle
  Simulink.Mask.create(ModelHandle) %To create mask using model handle
  ```

An example follows,

```
maskObj = Simulink.Mask.create('vdp');
```

```
                    Type: 'vdp'
             Description: 'The van der Pol Equation…'
                    Help: ''
          Initialization: ''
          SelfModifiable: 'off'
                 Display: ''
               IconFrame: 'on'
              IconOpaque: 'opaque'
    RunInitForIconRedraw: 'off'
              IconRotate: 'none'
              PortRotate: 'default'
               IconUnits: 'autoscale'
              Parameters: [0×0 Simulink.MaskParameter]
                BaseMask: [0×0 Simulink.Mask]
```

Simulink.Mask.get

Use the `Simulink.Mask.get` method to get the mask on a model as a mask object. The syntax to get the existing mask of a model is,

- Using the model name:

  ```
  Simulink.Mask.get(ModelName)
  ```

- Using the model handle

```
ModelHandle = get_param(gcs,'Handle') %To get the model handle
Simulink.Mask.get(ModelHandle) %To create mask using model handle
```

An example follows:

```
maskObj = Simulink.Mask.get('vdp');
```

```
                    Type: 'vdp'
             Description: 'The van der Pol Equation…'
                    Help: ''
          Initialization: ''
          SelfModifiable: 'off'
                 Display: ''
               IconFrame: 'on'
              IconOpaque: 'opaque'
    RunInitForIconRedraw: 'off'
              IconRotate: 'none'
              PortRotate: 'default'
               IconUnits: 'autoscale'
              Parameters: [0×0 Simulink.MaskParameter]
                BaseMask: [0×0 Simulink.Mask]
```

Note To get the model mask as a mask object in the mask callback, you can use `Simulink.Mask.get()` without passing a system name or system handle. Simulink does not require the system name (`gcb`) or the system handle (`gcs`) to query the mask object for the model mask.

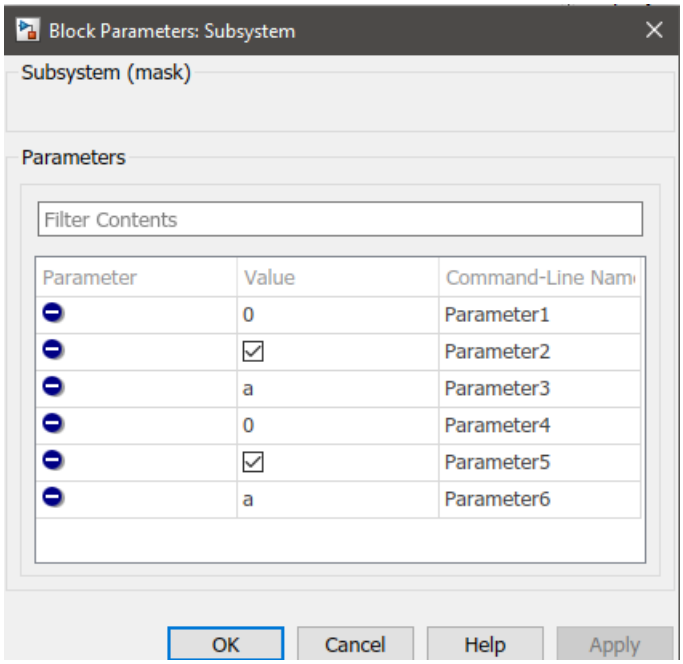
See Also

More About

- “Create and Reference a Masked Model” on page 39-49
- “Introduction to System Mask” on page 39-48
- “Masking Fundamentals” on page 39-2

Handling Large Number of Mask Parameters

The **Table** control in Mask Editor dialog box allows you to organize large number of mask parameters. The **Table** control can handle large (500+) number of mask parameters. You can include **Edit**, **Checkbox**, and **Popup** parameters within a **Table**.



You can also add large number of mask parameters in a **Table** programmatically. An example follows,

```
% Get mask object.
aMaskObj = Simulink.Mask.get(gcbh);

% Add Table controls to the mask.
aMaskObj.addDialogControl('Table', 'MyTable');

% Add parameters to table container.
for i = 1:length(Parameters) % To import values from an array called 'Parameters'
    aMaskObj.addParameter('Name', Parameters(i).Name, 'Type', Parameters(i).Type, 'Container', 'M
end
```

See Also

More About

- “Create Block Masks”
- “Create a Simple Mask” on page 39-6
- “Mask Editor Overview”

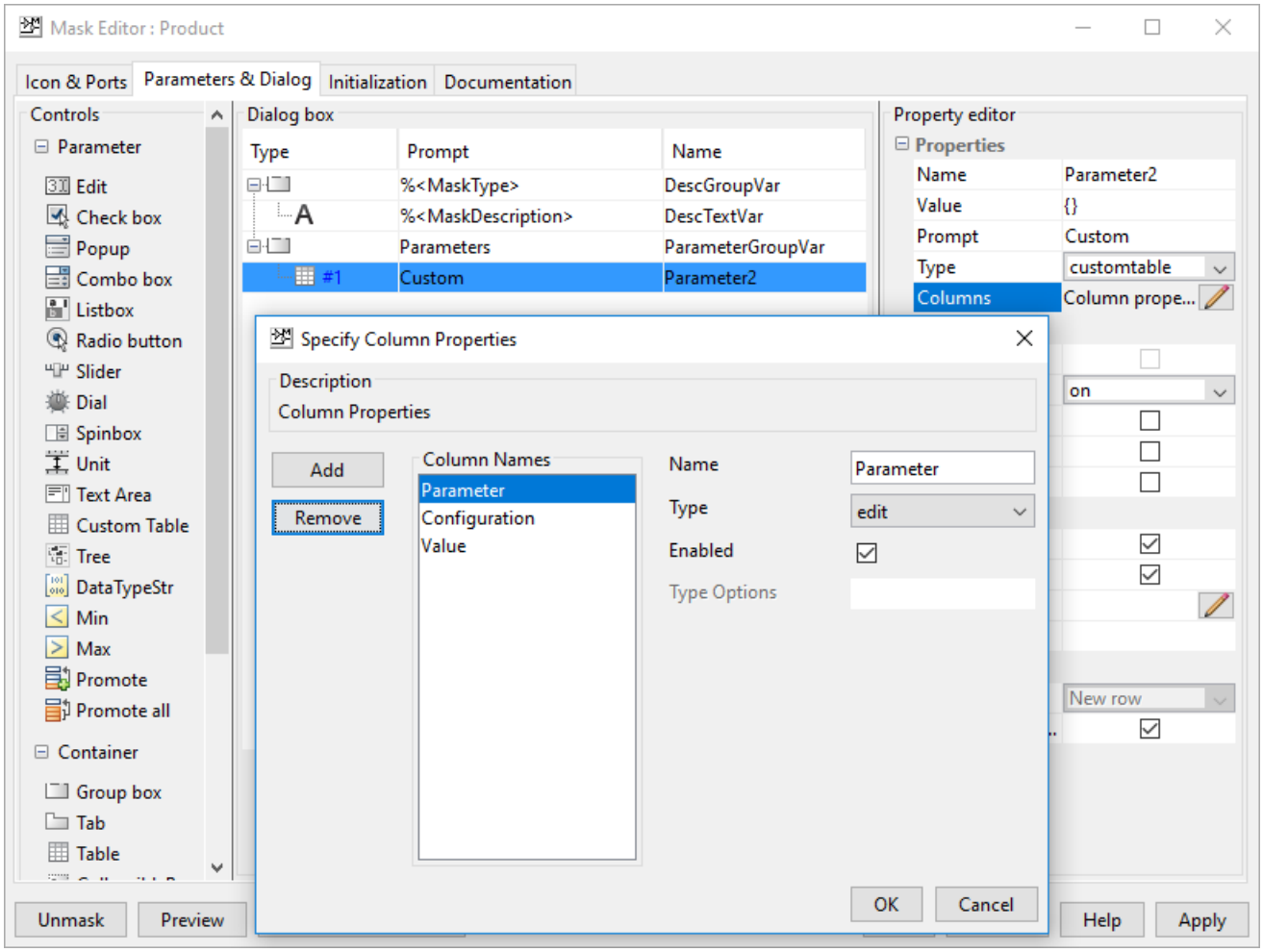
Customize Tables for Masked Blocks

The **Custom Table** parameter allows you to add customized table with structured data on a mask dialog box without writing custom codes. As a mask parameter, the custom table widget supports parameter promotion unlike the tables created using custom block dialog code.

You can provide input to the **Custom Table** parameter as a string in the cell array format.

Adding a Custom Table Parameter

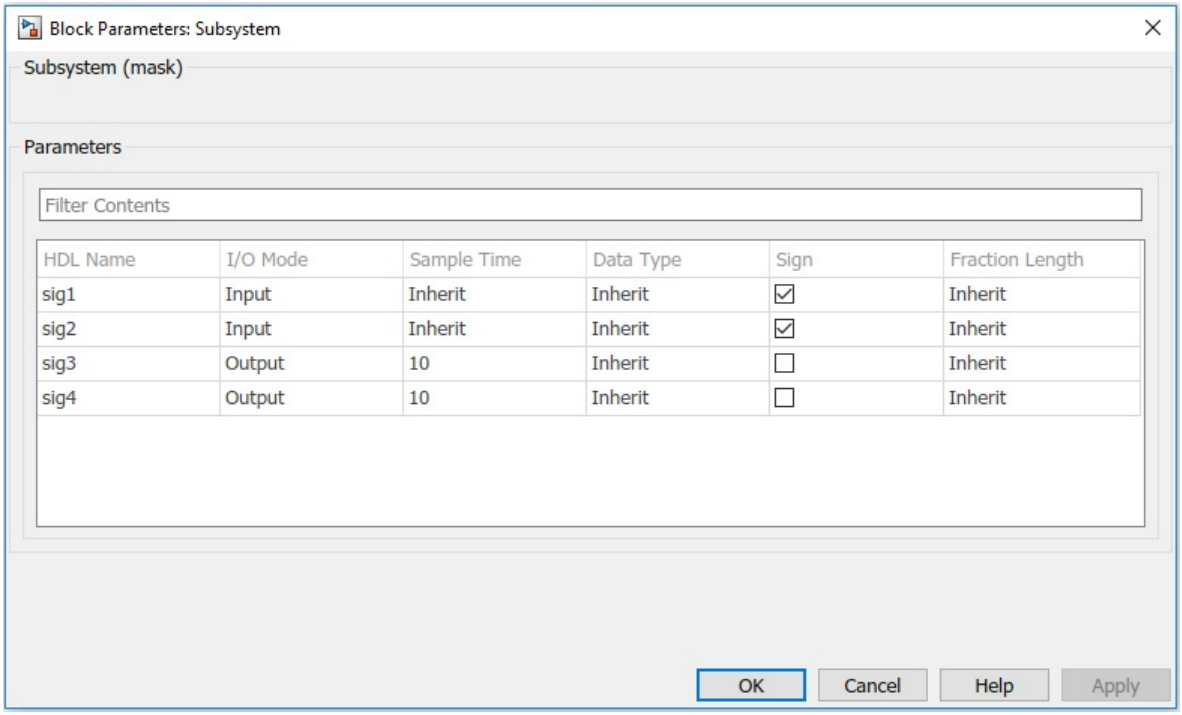
1. Open a Simulink model.
2. Select any block and on the **Block** tab, in the **Mask** group, click **Create Mask**. The Mask Editor opens.
3. In the **Parameters & Dialog** tab, click **Custom Table** to add it to the mask as a parameter. Specify a name and prompt for the parameter, as required.
4. In the **Property Editor** pane, click the Edit icon in the **Columns** field to specify the column properties like the number of columns, name of each column, and type of column. The available column types are **edit**, **checkbox**, and **popup**. Use the **Add** button to add a column and **Remove** button to remove an existing column.



5 Enter the values for the table in the **Value** field of **Property Editor** as a string in the cell array format. For example, if the value is given as:

```
{ 'sig1', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit';...
 'sig2', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit';...
 'sig3', 'Output', '10', 'Inherit', 'off', 'Inherit';...
 'sig4', 'Output', '10', 'Inherit', 'off', 'Inherit' }
```

The table created is:



6 Click **Apply** to save the changes to the mask. If you want to edit the table, right click and select **Mask > Edit Mask**.

See Also

More About

- “Control Custom Tables Programmatically” on page 39-59

Control Custom Tables Programmatically

You can add a custom table on the mask dialog box programmatically and control its properties using command line interface. For information on creating a custom table from the Mask Editor, see “Customize Tables for Masked Blocks” on page 39-57.

Add a Custom Table Parameter

You can add a custom table parameter to a mask dialog box using these commands:

```
% Mask Object
maskObj = Simulink.Mask.create(gcb);

% Add custom table parameter
tableParam = maskObj.addParameter( 'Name', 'myTable', 'Type', 'customtable' );

% Add values to the table
tableParam.Value = join( [ "{'sig1', 'Input', 'Inherit', 'Inherit', 'on',  'Inherit';",  ...
                           " 'sig2', 'Input', 'Inherit', 'Inherit', 'on',  'Inherit';",  ...
                           " 'sig3', 'Output', '10',     'Inherit', 'off', 'Inherit';",  ...
                           " 'sig4', 'Output', '10',     'Inherit', 'off', 'Inherit'}" ] );
```

Add Columns to a Table

You can add columns to a custom table using the `addColumn` command:

```
tableControl = maskObj.getDialogControl('myTable');
tableControl.addColumn( 'Name', 'HDL Name', 'Type', 'edit' );
tableControl.addColumn( 'Name', 'I/O Mode', 'Type', 'popup', 'TypeOptions', {'Input', 'Output'} )
tableControl.addColumn( 'Name', 'Sample Time', 'Type', 'edit' );
tableControl.addColumn( 'Name', 'Data Type', 'Type', 'popup', 'TypeOptions', {'Inherit', 'FixedPo
tableControl.addColumn( 'Name', 'Sign', 'Type', 'checkbox' );
tableControl.addColumn( 'Name', 'Fraction Length', 'Type', 'edit' );
```

Set and Get Table Properties

You can fetch the value of a cell if it had a change and set a new value for a cell in the table using these commands:

```
 % get values of the changed cell
changedCells = tableControl.getChangedCells();

% get value of a particular cell
tableControl.getValue( [rowIdx colIdx] );

% Set value for a particular cell
tableControl.setValue( [rowIdx colIdx], 'Value' );
```

Set and Get Cell Level Specifications

You can set and fetch the value of a particular cell in a custom table. The commands used are:

```
% set value for a particular table cell
tableControl.setTableCell( [rowIdx colIdx], 'Type', 'checkbox', 'Value', 'off', 'Enabled', 'off'
```

```
% get value from a particular table cell
tableCell = tableControl.getTableCell( [rowIdx colIdx] )

tableCell =

  CustomTableParamCellObject with properties:

          Value: 'Inherit'
           Type: 'popup'
        Enabled: 'off'
    TypeOptions: {4×1 cell}

tableCell.Value = 'Value'
```

Note The `setTableCell` and `getTableCell` APIs are expected to be used as part of mask parameter call backs while getting the number of rows in a table.

Edit Rows in a Custom Table

You can insert, remove, swap, and get the value of a specific row in a custom table. The commands used are:

```
% add a row to the table
tableControl.addRow( 'sig5', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit' )

% Insert a row at a specific location in the table
tableControl.insertRow( rowIndex, 'insertSig4', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit' )

% Remove a particular row
tableControl.removeRow( rowIndex )

% Swap two rows
tableControl.swapRows( rowIndex1, rowIndex2 )

tableControl.getSelectedRows()


ans =

     3     4
```

Edit Columns in a Custom Table

You can insert, remove, swap, and get the value of a specific column in a custom table. The commands used are:

```
% add a column to the table
tableControl.addColumn( 'Name', 'HDL Name', 'Type', 'edit' );

% Insert a column at a particular location in the table
tableControl.insertColumn( columnIndex, 'Name', 'HDL Name', 'Type', 'edit' );

% Remove a column from the table
tableControl.removeColumn( columnIndex );
```

```
tableControl.getColumn( columnIndex );
```

For example,

```
tableControl.getColumn( 4 )
```

```
ans =

  TableParamColumnInfo with properties:

          Name: 'Data Type'
          Type: 'popup'
       Enabled: 'on'
    TypeOptions: {4×1 cell}
```

Get and Set Table Parameter

You can use the `set_param` and `get_param` commands to set or get the values of the custom table parameter you created in a mask dialog box.

```
get_param( gcb, 'myTable' )
```

```
ans =
    '{ 'sig1', 'Input', 'Inherit', 'Inherit',...
 'on', 'Inherit'; 'sig2', 'Input', 'Inherit', 'Inherit',...
 'on', 'Inherit'; 'sig3', 'Output', '10', 'Inherit', 'off',...
 'Inherit'; 'sig4', 'Output', '10', 'Inherit', 'off', 'Inherit' }'
```

```
set_param( gcb, 'myTable', "{ 'sig1', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit' }" )
```

See Also

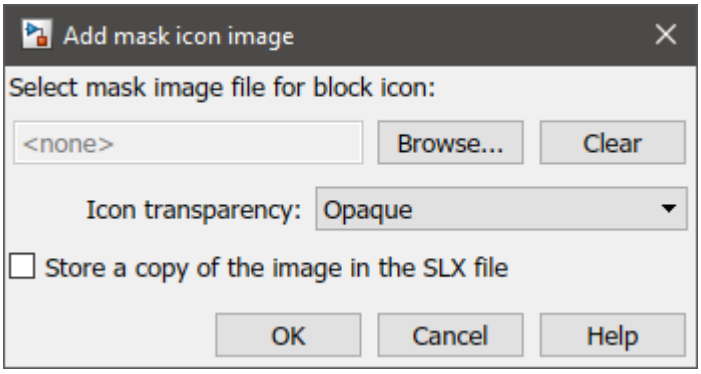
More About

- “Customize Tables for Masked Blocks” on page 39-57
- slexMaskParameterOptionsExample

Add Images in Masks

You can add images as icons on a Simulink mask and save them in the SLX file.

1. Select any masked block and on the **Block** tab, click **Add Image**.
2. In the **Add mask icon image** dialog box, click **Browse** to select an image from your local repository. You can also set the transparency using the **Icon transparency** field. Available options are: `Opaque`, `Transparent`, and `Opaque with ports`.



3. Select the **Store a copy of the image in the SLX file** check box if you want to store the mask image in the SLX file.

   Note You cannot store mask images in an MDL file.

4. Click **OK** to save your changes.

Store Mask Images Programmatically

- Convert mask image to internal for one block

  `Simulink.Mask.convertToInternalImage`
- Convert mask image to internal for the whole model

  `Simulink.Mask.convertToInternalImages`
- Convert mask image to external for one block

  `Simulink.Mask.convertToExternalImage`

See Also

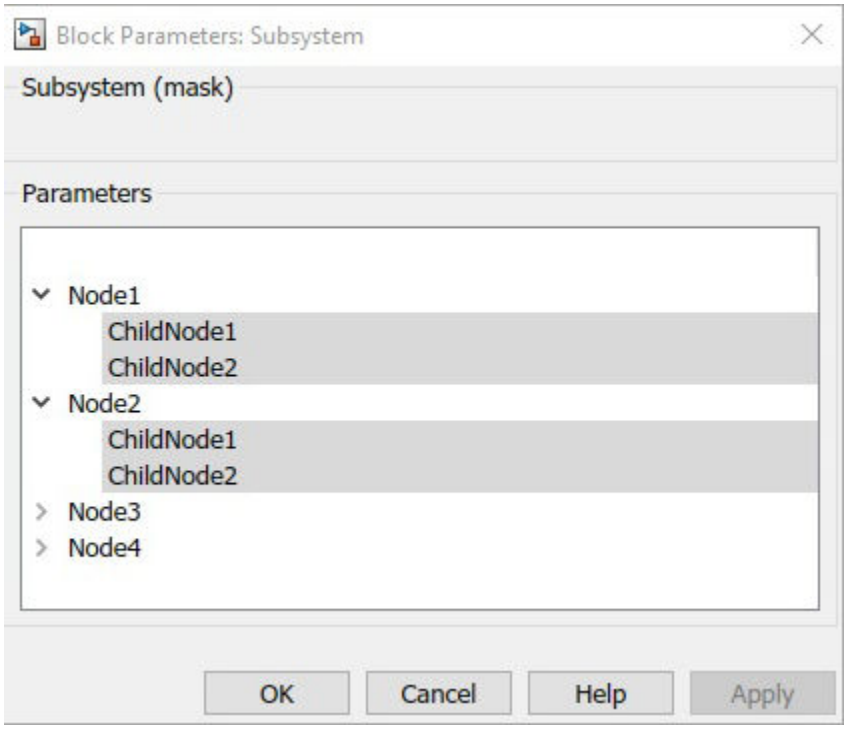
More About

- “Control Custom Tables Programmatically” on page 39-59

Create Hierarchical List in Mask Dialog

You can use the **Tree Control** option available in the **Display** section of the Mask Editor to create the hierarchical list of data in mask dialog box.

Consider a scenario in which you want to create a parent-child hierarchy on a mask dialog box as shown here:



To do so,

1. Right-click a block and in the content menu click **Mask > Create Mask** or **Mask > Edit Mask**.
2. In the **Parameters & Dialog** pane, click **Tree Control** from the **Display** section.
3. In the **Property editor** section, specify these in the **Tree Items** field:

   ```
   { 'Node1', {'ChildNode1', 'ChildNode2'}, 'Node2', {'ChildNode1',
   'ChildNode2'}, 'Node3', {'ChildNode1', 'ChildNode2'}, 'Node4',
   {'ChildNode1', 'ChildNode2'}}
   ```

4. Click **Apply**.

See Also

More About

- slexMaskingTreeControl
- “Mask Editor Overview”

Validating Mask Parameters Using Constraints

A mask can contain parameters that accept user input values. You can provide input values for mask parameters using the mask dialog box. Mask parameter constraints help you to create validations on a mask parameter without having to write your own validation code. Constraints ensure that the input for the mask parameter is within a specified range. For example, consider a masked Gain block. You can set a constraint where the input value must be between 1 and 10. If you provide an input that is outside the specified range, an error displays.

Create and Associate a Constraint

Launch Constraint Manager

Mask Editor contains a Constraint Manager with attributes and options to create your constraints. You can launch the Constraint Manager in two ways:

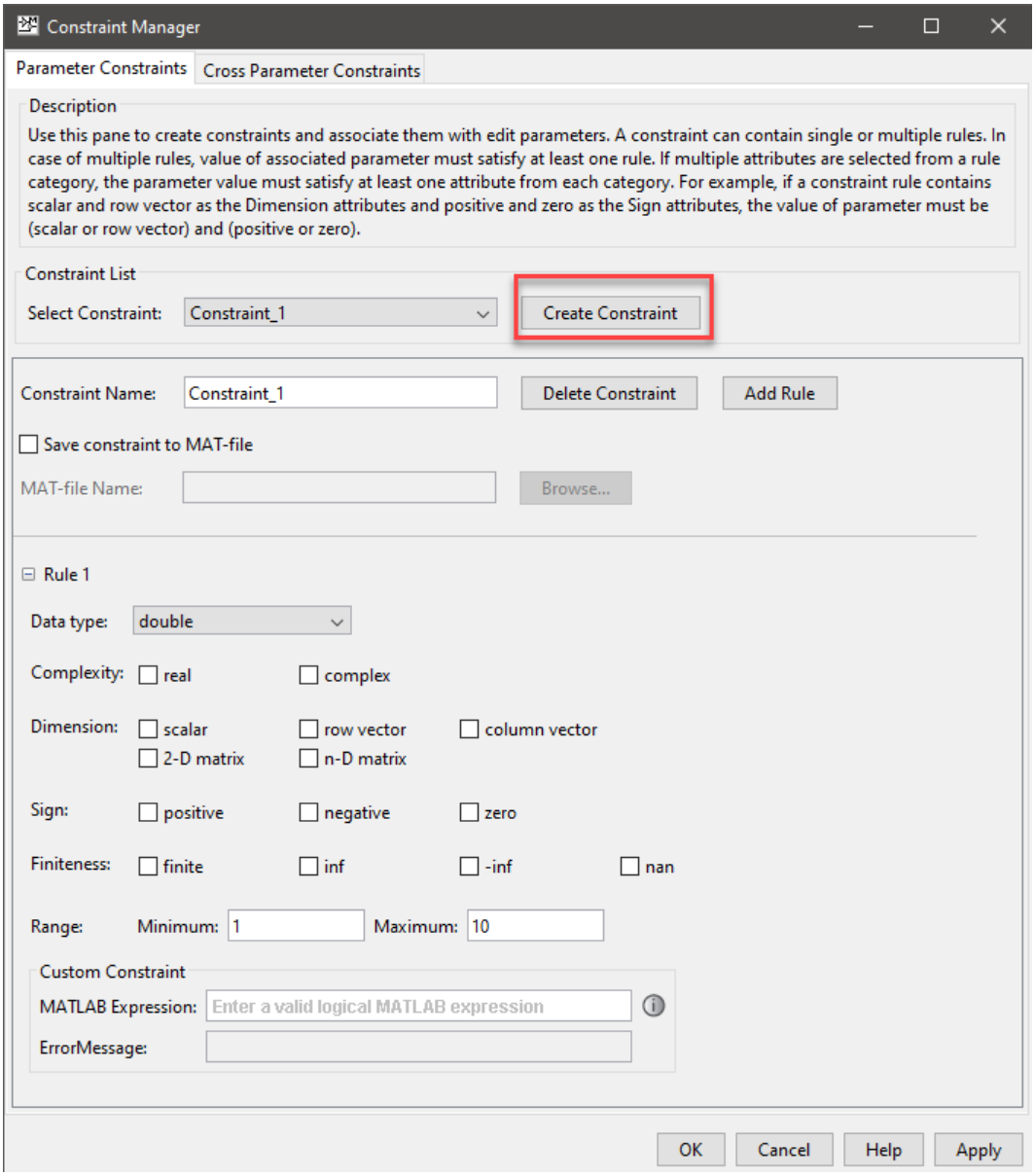
- Click the **Constraint Manager** button in Mask Editor
- While editing a parameter, select **Add New Constraint** from the `Constraint` drop-down menu under **Property Editor**.

Create a Constraint

You can create constraints according to your specification using the built-in attributes in the Constraint Manager. To create a constraint:

1. In the Constraint Manager, click **Create Constraint**.
2. Select attributes for the constraint in the **Rule** section. Depending on the data type selected the rule attributes change.

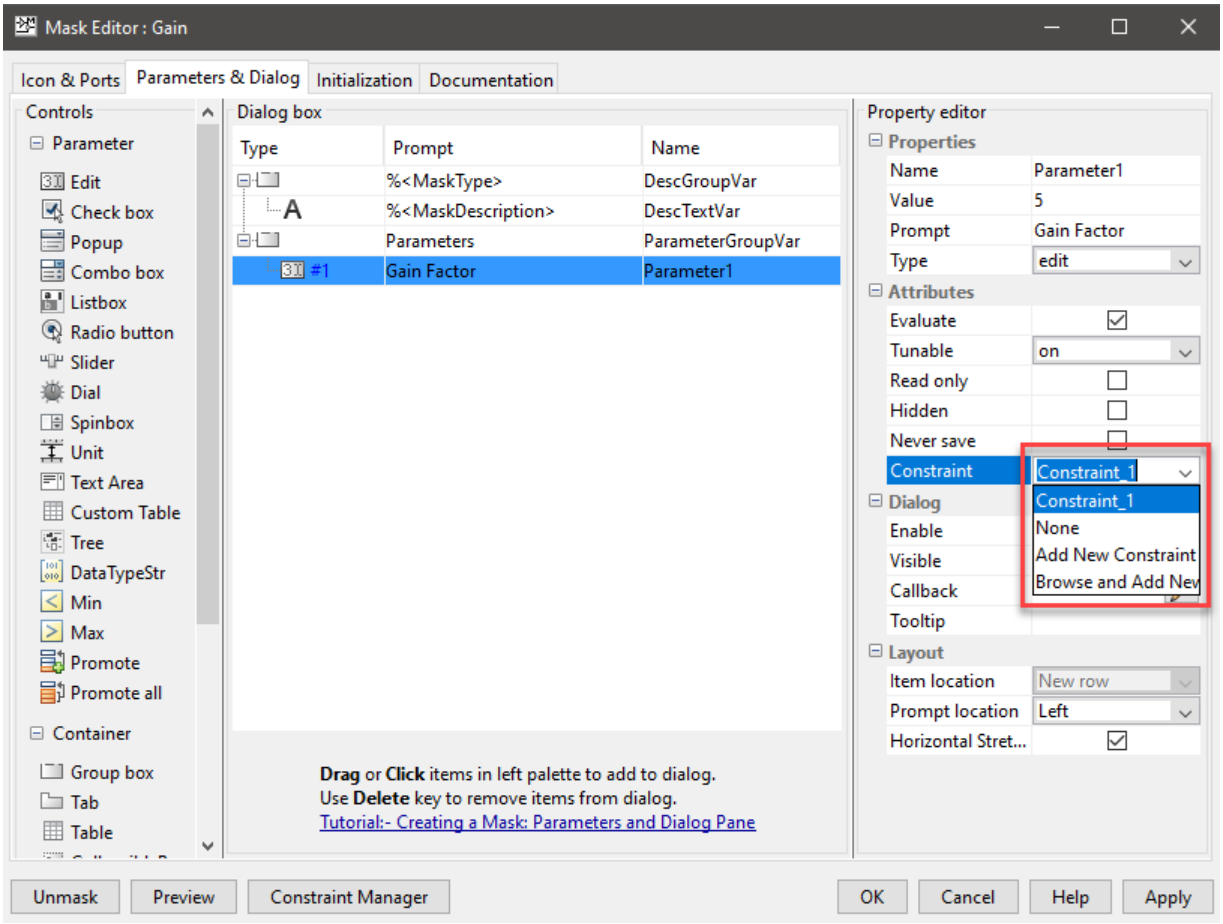
   For more details on rule attributes, see “Rule Attributes in Constraint Manager” on page 39-68.
3. Click **Apply** to create the constraint.



Associate the Constraint to a Mask Parameter

Once a constraint is created, you can associate it with any **Edit** or **Combobox** parameters in the **Mask Editor**.

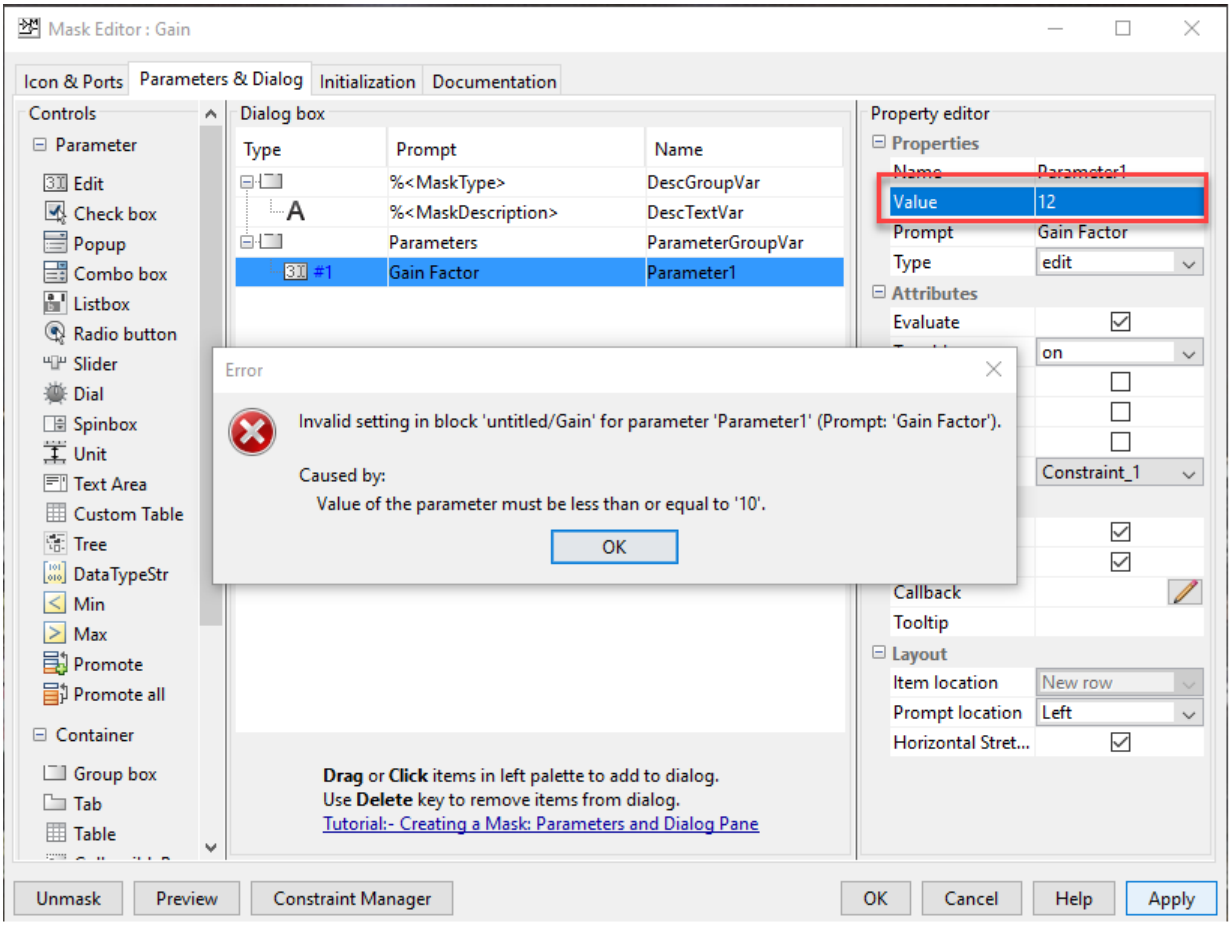
1. In the **Mask Editor**, select the parameter you want to associate a constraint with.
2. Select the constraint name from the **Constraint** drop-down menu.
3. Click **Apply** to associate the constraint.



Validate the Constraint

To check if the parameter is in adherence with the associated constraint:

1. Select a parameter with a constraint associated with it.
2. Provide the input values for the parameter in the Property editor. If the input is outside the specification for the associated constraint, an error displays.



Create a Cross-Parameter Constraint

Cross-parameter constraints are applied among two or more **Edit** or **Combobox** type mask parameters. You can use a cross parameter constraint when you want to specify scenarios such as, Parameter1 must be greater than Parameter2.

1. Launch Constraint Manager.
2. Click the **Cross-Parameter Constraints** tab.
3. Click **Create Constraint**. A new cross-parameter constraint is created with a default name (Constraint_1). You can change the constraint name.
4. Specify the following values for the new constraint:

   - Name – Specify a name for the constraint
   - MATLAB Expression – Specify a valid MATLAB expression. This expression is evaluated during edit time and simulation
   - Error Message – Specify the error message to be displayed when the constraint rule is not met. If no error message is specified, a default error message displays.
5. Click **Apply**.

Rule Attributes in Constraint Manager

Rule attributes available in the Constraint Manager to create constraints are shown in the table:

Rule Attribute	Attribute Value	Description
Data type	`double`, `single`, `numeric`, `integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `boolean`, `enum`, `fixdt`, `string`, `half`	Specify the acceptable data type of the mask parameter value. For example, if the data type specified is `uint8`, the acceptable value is in the range of uint8, that is, `0—255`.
Complexity	`real`, `complex`	Specify if the mask parameter value can be a real or complex number.
Dimensions	`scalar`, `row vector`, `column vector`, `2-D matrix`, `n-D matrix`	Specify the acceptable dimensions for the mask parameter.
Sign	`positive`, `negative`, `zero`	Specify if the input value can be positive, negative, or zero.
Finiteness	`finite`, `inf`, `-inf`, `NaN`	Specify the acceptable finiteness of mask parameter value.
Range	`Minimum`, `Maximum`	Specify the acceptable range of mask parameter value.
Custom Constraint	Valid MATLAB expression	Specify custom constraint for the mask parameter using a valid MATLAB expression. You can use the value token to parameterize the expression. During validation, the evaluated value of the parameter replaces the value token.
Custom Error Message	Character vector	Specify a custom error message for the custom constraint. You can specify the error message as character vector or as a message catalog ID.

See Also

More About

- slexMaskConstraints

Custom Constraints

If the constraint you need cannot be created with the built-in attributes, you can create your custom constraint by writing your own MATLAB expression. To create a custom constraint:

1. Launch Constraint Manager.
2. Click **Create Constraint**.
3. In the **Custom Constraint** section, enter a valid MATLAB expression in the **MATLAB Expression** field.

   You can use the `value` token to parameterize the expression. During validation, the evaluated value of the parameter replaces the `value` token. For example, if the MATLAB expression for a constraint is `value > 100` and is associated with the **Edit** type mask parameter, `Parameter1`, the MATLAB expression evaluates as `Parameter1 > 100`. This helps in assigning the constraint to multiple parameters.

   For example, to set a custom constraint for a Gain block with an edit mask parameter that must accept only even numbers, in the **MATLAB Expression** enter the MATLAB command:

   ```
   "mod(value,2) ==0"
   ```
4. Write the error message for your custom constraint in the **Error Message** field.
5. Click **Apply**.

See Also

More About

- “Validating Mask Parameters Using Constraints” on page 39-64

Shared Constraints

You can create and save parameter constraints in a MAT file. You can save a constraint in a MAT file either from the Mask Editor or programmatically. Multiple block masks can share a parameter constraint saved in a MAT file. To create a shared constraint:

1. Launch Constraint Manager.
2. Click **Create Constraint**.
3. Create a constraint with the required attributes.
4. Select the **Save constraint to MAT file** check box and specify the MAT file name.
5. Click **Apply**.

A shared constraint is listed in the Property editor section in the format `<MATFileName>:<ConstraintName>`. You can select any available constraint as required and associate it with an **Edit** parameter.

While creating shared constraints it is recommended that you:

- Save constraints to MAT files that are in the MATLAB path.
- Provide meaningful names to constraints so that other users can understand the nature of the constraint easily.

See Also

More About

- “Validating Mask Parameters Using Constraints” on page 39-64

Control Constraints Programmatically

You can create a custom constraint programmatically from the MATLAB command window

To create a custom constraint:

```
% Get mask constraint handle
paramConstraint = maskObj.getParameterConstraint('const1');

% Add rules to the constraint.
paramConstRule = paramConstraint.addParameterConstraintRule('CustomConstraint','mod(value,2) ==0
```

This creates a custom constraint:

```
ans =

  ParameterConstraintRules with properties:

           DataType: ''
          Dimension: {0×1 cell}
         Complexity: {0×1 cell}
               Sign: {0×1 cell}
         Finiteness: {0×1 cell}
            Minimum: ''
            Maximum: ''
    CustomConstraint: 'mod(value,2) ==0''
```

You can create and save constraints in a MAT file and load constraints from a MAT file programmatically from the MATLAB command window:

To save a constraint in a MAT file:

```
uint16Constraint = Simulink.Mask.Constraints;
uint16Constraint.Name = 'uint16Constraint';
uint16Constraint.addParameterConstraintRule('DataType', 'uint16');
save('constraintList.mat', 'uint16Constraint','-append');  % appends 'uint16Constraint'
save('constraintList.mat', 'uint16Constraint') % overwrites the MAT file with 'uint16Constraint'
```

Note It is recommended that the constraint name (for example, `uint16Constraint.Name = 'uint16Constraint'`) and the variable used for storing the constraint (for example, `uint16Constraint = Simulink.Mask.Constraints`) have the same name.

To load a constraint from a MAT file:

```
myConstraints = load(‘constraintList’) % loads constraints into the variable 'myConstraints'
```

See Also

More About

- “Validating Mask Parameters Using Constraints” on page 39-64

Define Measurement Units for Masked Blocks

Measurement units translate the amount of entities supplied to your computations. They are also crucial when diverse users using different systems of measurement are using the same equation. You can add measuring units to a Simulink model to enhance the usability and to avoid confusion while analyzing equations.

To specify measuring units for masked blocks, you can:

- Promote a `Unit` parameter from the underlying block to the mask. For more information, see “Promote Parameter to Mask” on page 39-23.
- Add a **Unit** parameter to the mask dialog box as shown in the following steps:

  1. Open a Simulink Model.
  2. Select the block to be masked and press **Ctrl+M**. The Mask Editor opens.
  3. In the Mask Editor, click the **Parameters & Dialog** tab.
  4. In the Parameters section, click **Unit**.
  5. Click the **Initialization** tab in the Mask Editor dialog box.
  6. To define the measurement unit for the **Unit** parameter, specify the initialization command by using the following syntax:

     ```
     set_param([gcb '/<blockname>'], '<unit parameter name>', get_param(gcb, '<unit parameter n
     ```

     For example, to associate the **Unit** parameter with the measuring unit of the Inport block, use:

     ```
     set_param([gcb '/In1'], 'Unit', get_param(gcb, 'Unit'));
     ```
  7. Click **Apply**, and then click **OK**. For more information, see slexMaskParameterOptionsExample.

See Also

More About

- “Masking Fundamentals” on page 39-2
- “Promote Parameter to Mask” on page 39-23

Masking Example Models

The Simulink Masking example models help you to understand and configure mask parameters, properties, and features. The examples are grouped by type. In an example model:

- To view the mask definition, double-click the View Mask block.
- To view the mask dialog box, double-click the block.

Goals	Example Models	Related Topics
Add Parameters control type to mask dialog box. For example, Evaluate, Tune, Add Image, Pop-up, Combo-box, Slider and Dial, Slider Range	Mask Parameters	“Mask Editor Overview”
• Add an opaque mask with visible port labels (**Icon Transparency**). • Specify Run Initialization instructions	Mask Icon Drawing	“Mask Editor Overview”
Use dialog layout options to: • Add horizontal stretch on mask dialog box • Group multiple parameters in to tabs • Create a dynamic pop-up parameter • Create a collapsible content panel in a mask dialog box • Create a table container to group multiple parameters	Dialog Layout Options	
Promote parameters to a mask	Mask Parameter Promotion	“Promote Parameter to Mask” on page 39-23
Execute Mask Callback	Mask Callbacks	“Mask Callback Code” on page 39-14
Display an Image as icon on a mask	Mask Display and Initialization	“Add Image to Block Icon” on page 45-62
• Make a parameter invisible in the mask • Disable a mask parameter	Dynamic Mask Dialog	
Set a mask to be self-modifying	Self-Modifying Library Masks	“Create Self-Modifying Mask” on page 39-43
Use MATLAB graphics to create a MATLAB GUI and use it as mask dialog	Handle Graphics in Masking	
Mask a variant subsystem	Masking Variant Blocks	

Goals	Example Models	Related Topics
Create a self-modifiable interface block	Self-Modifiable Interface Blocks	

There are certain bad practices, which when followed may result in unexpected behavior of the masking features. The following table shows some of the bad practices.

Bad Practice	Example Models
Use of Mask callbacks, where the callback modifies entities outside of its scope may result in unexpected behavior	Unsafe Mask Callback Error
Setting parameters outside of the hierarchical boundary in nested masks may lead to unexpected behavior	Nested Mask Error

See Also

More About

- “Create Block Masks”
- “Mask Editor Overview”
- “Masking Fundamentals” on page 39-2

Create a Custom Table in the Mask Dialog

This example shows how to create a custom table in a mask dialog. This model includes a Subsystem block with a mask on it. The mask has a callback that modifies the table content based on the values provided in the cells. The callback comes from an external supporting file. To create this table, start by adding a mask to the block or editing an existing mask.

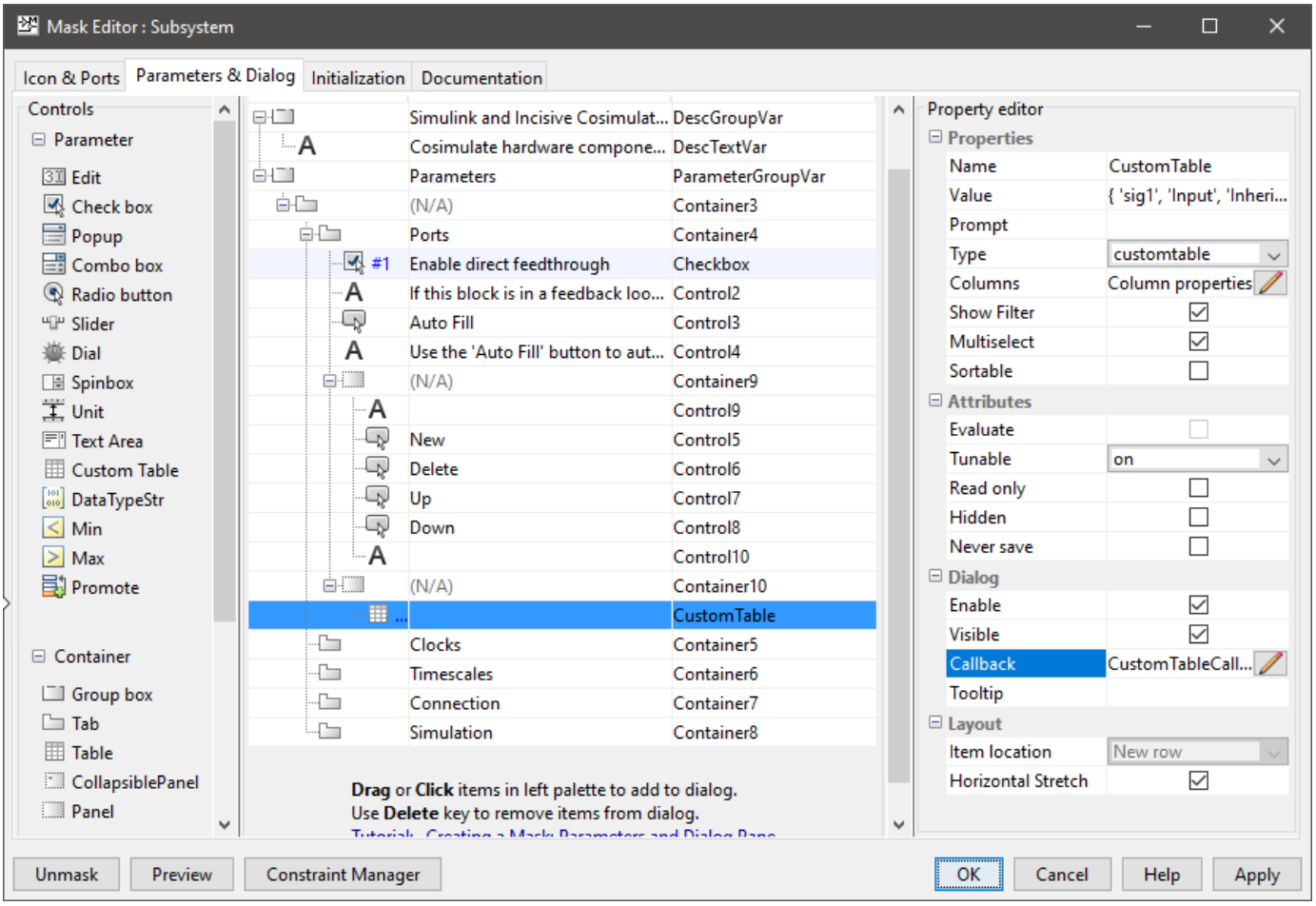
Add and Configure a Custom Table parameter

To add a Custom Table parameter, open the Mask Editor,go to the **Parameters & Dialog** tab and click **Custom Table**. Then, add values in the **Property editor** section. For this example, in the **Value** field, add:

```
{ 'sig1', 'Input', 'Inherit', 'Inherit', 'Signed', 'Inherit'; 'sig2',
'Input', 'Inherit', 'Inherit', 'Signed', 'Inherit'; 'sig3', 'Output', '10',
'Inherit', 'Signed', 'Inherit'; 'sig4', 'Output', '10', 'Inherit', 'Signed',
'Inherit' }.
```

In the **Columns** field, click the Edit icon and add the name of columns. The column names used in this example are: HDL Name, I/O Mode, Sample Time, Data Type, Sign, and Fraction Length.

In the **Dialog** section, edit the **Callback** field to add your callback. The callback used in this example is from an external file named `CustomTableCallback.m`. Enter the filename in the callback field. This callback defines how the values change in a cell based on the value from other cells and columns.



Add Buttons and Options to Control the Custom Table

In this example, four buttons are added to add a new row, delete a selected row, and move a row up or down. To add the buttons, in the **Action** section click the Button parameter four times. Name the buttons as **New**, **Delete**, **Up**, and **Down**. To configure the buttons, edit the **Callback** field in the Property editor and add the appropriate callbacks. The callback used for each button is:

New

```
maskObj = Simulink.Mask.get(gcb);
tableControl = maskObj.getDialogControl( 'CustomTable' );

hdlName =  'sig';
rowIndex = tableControl.getNumberOfRows();
hdlName = strcat( 'sig', num2str( rowIndex + 1 ) );
tableControl.addRow( hdlName, 'Input', 'Inherit', 'Inherit', 'Signed', 'Inherit' )
```

Delete

```
maskObj = Simulink.Mask.get(gcb);
tableControl = maskObj.getDialogControl( 'CustomTable' );

rowIndex = tableControl.getSelectedRows();
hdlName =  'sig';
```

```
if ( ~isempty(rowIndex) )
    tableControl.removeRow( rowIndex(1) );
end
```

Up

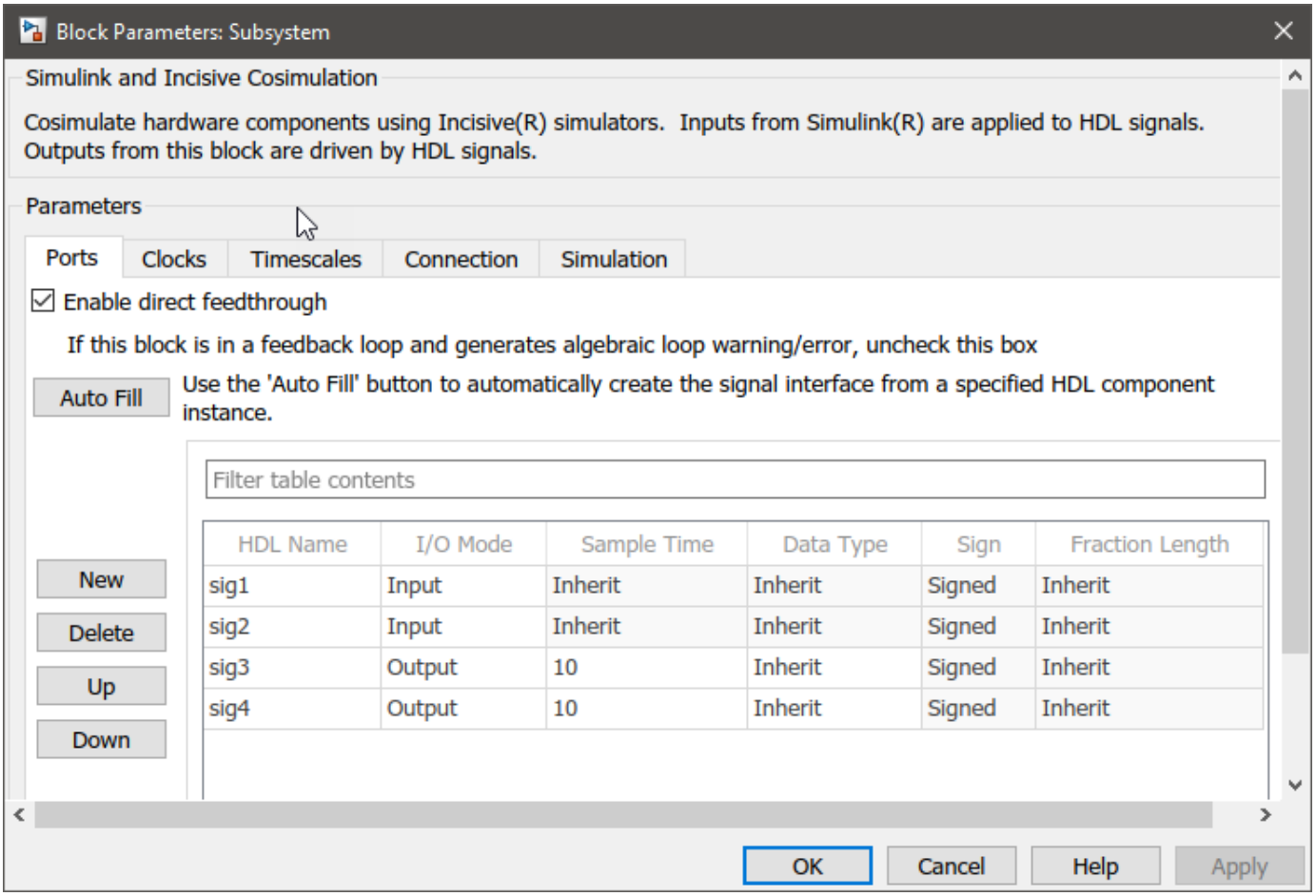
```
maskObj = Simulink.Mask.get(gcb);
tableControl = maskObj.getDialogControl( 'CustomTable' );

rowIndex = tableControl.getSelectedRows();

if ( ~isempty(rowIndex) )
    tableControl.swapRows( rowIndex(1)-1, rowIndex(1) );
end
```

Down

```
maskObj = Simulink.Mask.get(gcb);
tableControl = maskObj.getDialogControl( 'CustomTable' );

rowIndex = tableControl.getSelectedRows();

if ( ~isempty(rowIndex) )
    tableControl.swapRows( rowIndex(1)+1, rowIndex(1) );
end
```

In addition to these buttons, the table also has a checkbox to enable direct feedthrough and an autofill button that automatically creates the signal interface from a specified HDL component instance. To add these options, add a checkbox and a button control and add the appropriate configurations.

After adding all the values, click **OK** to save the changes. You can preview the table using the **Preview** button. This is a preview of the final table created in this example:


Block Parameters: Subsystem
Simulink and Incisive Cosimulation
Cosimulate hardware components using Incisive(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.
Parameters
Ports
Clocks
Timescales
Connection
Simulation
Enable direct feedthrough
If this block is in a feedback loop and generates algebraic loop warning/error, uncheck this box
Auto Fill
Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.
Filter table contents
HDL Name
I/O Mode
Sample Time
Data Type
Sign
Fraction Length
sig1 Input Inherit Inherit Signed Inherit
sig2 Input Inherit Inherit Signed Inherit
sig3 Output 10 Inherit Signed Inherit
sig4 Output 10 Inherit Signed Inherit
New
Delete
Up
Down
OK
Cancel
Help
Apply

Create a Block Mask Icon

You can create block icons on mask using various mask drawing commands. This model contains a collection of subsystem blocks, each showing a particular mask drawing command in use. To create a mask icon on a block, you must create a mask and add the icon drawing command to the mask in the mask editor. To create a mask:

1. Select a **Subsystem block** . For example, the Graph as Icon block.
2. On the **Subsystem Block** tab, in the **Mask** group, click **Create Mask/Edit Mask**.
3. In the Mask Editor dialog box, click the **Icon & Ports** tab.
4. Enter the command in the Icon drawing commands pane on the right. You can set the Run Initialization value to On, Off or Analyze based on the dependency on mask workspace.

Plotted function as Icon

This command let you plot a graph and set it as the mask icon. Select the subsystem Graph as Icon. In the Mask Editor > Icon and Ports tab, enter **plot([10 20 30 40], [10 20 10 15])**. Run Initialization can be set to Off as there is no mask workspace dependency.

Transfer Function as Icon

This command helps to set a Transfer function as the mask icon. Select the subsystem Transfer Function as Icon. In the Mask Editor > Icon and Ports tab, enter **droots([-1], [-4 -5], 4)**. Run Initialization can be set to Off as there is no mask workspace dependency.

Color patch as Icon

This command lets you to set a color patch as mask icon. Select the subsystem Color patch as Icon. In the Mask Editor > Icon and Ports tab, enter **patch([0 10 20 30 30 0], [10 30 20 25 10 10],[1 0 0])**. Run Initialization can be set to Off as there is no mask workspace dependency.

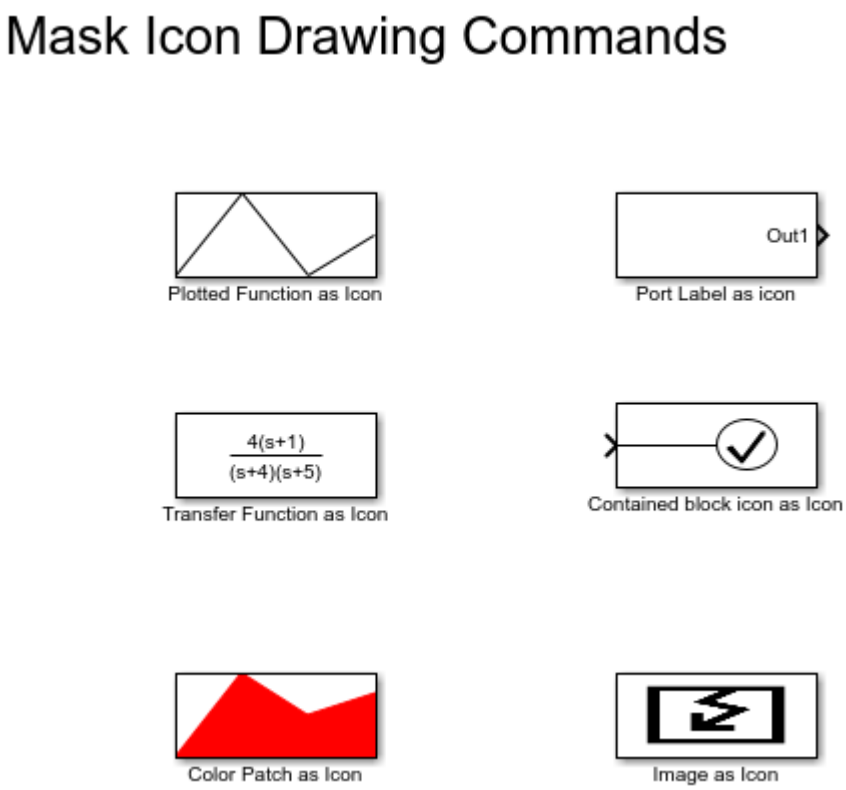
Image as Icon

This command lets you to set a color patch as mask icon. Select the subsystem Image as Icon. In the Mask Editor > Icon and Ports tab, enter **image('maskimage.png','center')**. Run Initialization is set to On. Make sure the image is available in the current folder in MATLAB.

Contained block icon as Icon

This command lets you promote the icon of a block contained in a subsystem to the subsystem mask. Select the subsystem Promoted block Icon . In the Mask Editor > Icon and Ports tab, enter **block_icon(Assertion)**.

```
open_system('mask_icon_drawing_example');
```

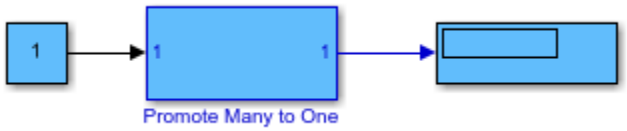

Mask Icon Drawing Commands
Plotted Function as Icon
Out1
Port Label as icon
4(s+1)
(s+4)(s+5)
Transfer Function as Icon
Contained block icon as Icon
Color Patch as Icon
Image as Icon

Promote Block Parameters on a Mask

You can use Parameter Promotion to promote any underlying parameter of a block either to a block mask or to a subsystem mask. This model contains a subsystem that has 3 Gain blocks (**Gain1**, **Gain2, and *Gain3**). The variable K represents the Gain parameter for these Gain blocks. You can promote only the Gain parameter of each of these Gain blocks to the block mask as a single parameter. When you do so, the parameter K is available on the mask for editing and its value will be applied to **Gain1** , **Gain2** , and **Gain3** blocks.

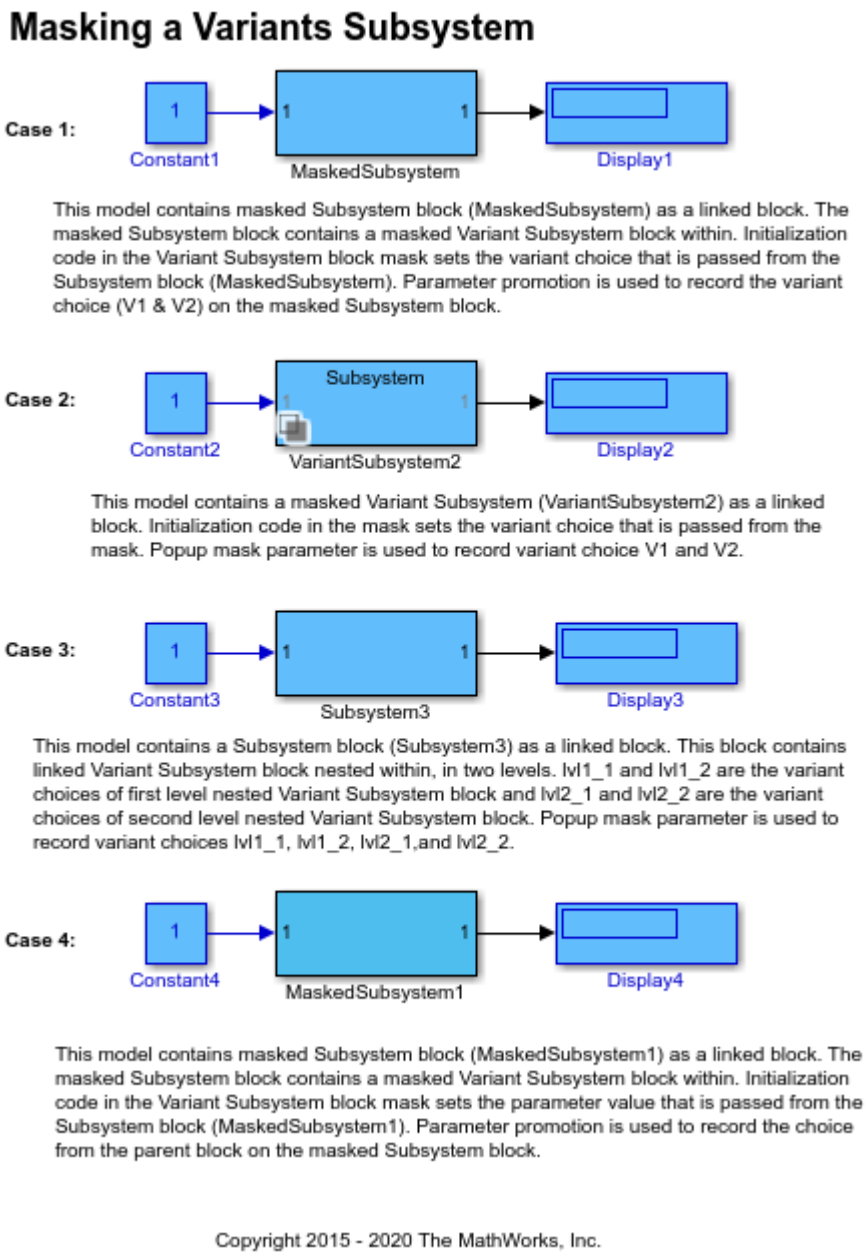
1. Select the Subsystem block.
2. On the **Subsystem Block** tab, in the **Mask** group, click **Create Mask/Edit Mask**.
3. In the Mask Editor dialog box, click the **Parameters & Dialog** tab.
4. In the **Controls** pane, click **Promote** .
5. In the **Property editor** pane, **Type options** field, click
6. In the **Promoted Parameter Selector** dialog box, select **Gain1** .
7. Select **Gain** from the **Promotable parameters** table and click the Add to promoted parameter list button. Similarly, add Gain parameter for **Gain2**.
8. Click **OK** .
9. In the Mask Editor dialog box, edit the prompt names for the **Gain** parameter. Here the Prompt used is **Common gain** .
10. Click **OK** to finish creating subsystem mask with many-to-one promotion.
11. Simulate the model. Notice that the value 4 is passed from the mask to the underlying block **Gain1**, **Gain2** , and **Gain3** . In this case, the output shows 64.

```
open_system('promote_block_param_to_mask');
```



Mask a Variant Subsystem

This example shows how to use a masked Variant Subsystem block in a Simulink model. Click the **Open Model** button located on the top right corner to view the related example model. This example model references masked library blocks.



When you mask a Variant Subsystem block, you can specify the variant choice from the mask dialog box. The variant choice that you specify on the mask dialog box is applied on the underneath Variant Subsystem block.

To pass the variant choice from the mask to the Variant Subsystem block, you can either use the `set_param` command or the parameter promotion option.

Let us consider the cases described in the example model.

- Case 1: The mask parameter promotion option is used to promote the Variant Subsystem block parameter to the mask. The Variant Subsystem block is wrapped within a masked Subsystem block. Initialization code (`set_param`) is used in the Variant Subsystem block to define the variant choice which is further passed on to the mask on the Subsystem block using parameter promotion. This promoted parameter records the variant choice specified from the masked Subsystem block.
- Case 2: The Popup mask parameter is used to create the choice option on the top level masked Subsystem block. This masked Subsystem block contains a Variant Subsystem block. Initialization code (`set_param`) is used in the Variant Subsystem block to define the variant choice. The value that you specify as a variant choice from the mask dialog box (**Popup** parameter) is transferred to the underneath Variant Subsystem block to set it choices.
- Case 3: This case is similar to `Case 2` with an additional layer of Variant Subsystem block in the model. Initialization code (`set_param`) in the Subsystem block is used to define the variant choice.
- Case 4: Parameter promotion is used to record the choice from the parent block on the masked Subsystem block. The masked Subsystem block contains a masked Variant Subsystem block within. Initialization code (`set_param`) in the Variant Subsystem block mask sets the parameter value that is passed from the Subsystem block (MaskedSubsystem1).

More About

- “Masking Fundamentals” on page 39-2
- “Initialization Function” on page 12-108

Creating Custom Blocks

Types of Custom Blocks

In this section...

MATLAB Function Blocks

A MATLAB Function block allows you to use the MATLAB language to define custom functionality. These blocks are a good starting point for creating a custom block if:

- You have an existing MATLAB function that models the custom functionality.
- You find it easier to model custom functionality using a MATLAB function than using a Simulink block diagram.
- The custom functionality does not include continuous or discrete dynamic states.

You can create a custom block from a MATLAB function using one of the following types of MATLAB function blocks.

- The Interpreted MATLAB Function block allows you to use a MATLAB function to define a SISO block.
- The MATLAB Function block allows you to define a custom block with multiple inputs and outputs that you can deploy to an embedded processor.

Each of these blocks has advantages in particular modeling applications. For example, you can generate code from models containing MATLAB Function blocks.

MATLAB System Blocks

A MATLAB System block allows you to use System objects written with the MATLAB language to define custom functionality. These blocks are a good starting point for creating a custom block if:

- You have an existing System object that models the custom functionality.
- You find it easier to model custom functionality using the MATLAB language than using a Simulink block diagram.
- The custom functionality includes discrete dynamic states.

Subsystem Blocks

Subsystem blocks allow you to build a Simulink diagram to define custom functionality. These blocks serve as a good starting point for creating a custom block if:

- You have an existing Simulink diagram that models custom functionality.

- You find it easier to model custom functionality using a graphical representation rather than using handwritten code.
- The custom functionality is a function of continuous or discrete system states.
- You can model the custom functionality using existing Simulink blocks.

Once you have a Simulink subsystem that models the required behavior, you can convert it into a custom block by:

1. Masking the block to hide the block contents and provide a custom block dialog box.
2. Placing the block in a library to prohibit modifications and allow for easily updating copies of the block.

For more information, see “Custom Libraries” and “Create Block Masks”.

C Caller Block

The C Caller block allows you to integrate C code into Simulink blocks. These blocks serve as a good starting point for creating a custom block if:

- You have existing C code that models custom functionality.
- Your C functions do not read or write global/static variables.
- You want blocks to easily integrate with other Simulink features, such as Simulink Coverage, Simulink Test, and Simulink Coder.
- You are not modeling dynamic systems.

S-Function Blocks

S-function blocks allow you to write MATLAB, C, or C++ code to define custom functionality. These blocks serve as a good starting point for creating a custom block if:

- You have existing MATLAB, C, or C++ code that models custom functionality.
- You use continuous or discrete dynamic states or other system behaviors that require access to the S-function API.
- You cannot model the custom functionality using existing Simulink blocks.

You can create a custom block from an S-function using one of the following types of S-function blocks.

- The Level-2 MATLAB S-Function block allows you to write your S-function using the MATLAB language. (See “Write Level-2 MATLAB S-Functions”). You can debug a MATLAB S-function during a simulation using the MATLAB debugger.
- The S-Function block allows you to write your S-function in C or C++, or to incorporate existing code into your model using a C MEX wrapper. (See “Implement C/C++ S-Functions”.)
- The S-Function Builder block assists you in creating a C MEX S-function or a wrapper function to incorporate legacy C or C++ code. (See “Implement C/C++ S-Functions”.)
- The Legacy Code Tool transforms existing C or C++ functions into C MEX S-functions. (See “Integrate C Functions Using Legacy Code Tool”.)

The S-function target in the Simulink Coder product automatically generates a C MEX S-function from a graphical subsystem. If you want to build your custom block in a Simulink subsystem, but

implement the final version of the block in an S-function, you can use the S-function target to convert the subsystem to an S-function. See “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” (Simulink Coder) in the Simulink Coder User's Guide for details and limitations on using the S-function target.

Masked Blocks

You can customize any block by adding a mask to it. A mask is a custom interface to the block. You can customize a block using a mask in many ways, such as:

- Change the block appearance.
- Hide some or all of the parameters from the user of the block.
- Customize block parameters.

To learn more about masked blocks, see “Create Block Masks”.

See Also

Interpreted MATLAB Function | Level-2 MATLAB S-Function | MATLAB Function | MATLAB System | S-Function | S-Function Builder | Simulink Function | Subsystem

More About

- “Comparison of Custom Block Functionality” on page 40-5
- “Design and Create a Custom Block” on page 40-12

Comparison of Custom Block Functionality

In this section...

When creating a custom block, consider:

- Does the block model continuous or discrete state behavior on page 40-6?
- Is the simulation performance on page 40-6 important?
- Do you need to generate code on page 40-8 for a model containing the custom block?

This table shows how each custom block type addresses the three concerns.

Modelling Considerations

Custom Block Type	Model State Dynamics on page 40-6	Simulation Performance on page 40-6	Code Generation on page 40-8
Interpreted MATLAB Function	No	Less fast	Not supported
Level-2 MATLAB S-function	Yes	Less fast	Requires a TLC file
MATLAB Function	No	Fast	Supported with exceptions
MATLAB System	Yes	Fast	Supported with exceptions
S-Function	Yes	Fast	Requires a TLC file or non-inline S-Function support
C Caller	No	Fast	Supported
S-Function Builder	Yes	Fast	Supported
Simulink Function	Yes	Fast	Supported
Subsystem	Yes	Fast	Supported

For detailed design of custom blocks, consider:

- Does the custom block need multiple input and output ports on page 40-9?
- What are the callback methods on page 40-10 to communicate with the Simulink engine and which custom blocks let you implement all or a subset of these callback methods?
- How important is the effect of the custom block on the speed of updating the Simulink diagram on page 40-9?

Model State Behavior

You need to model the state behavior for a block that requires some or all of its previous outputs to compute its current outputs. See “State variables” for more information.

Custom Block Type	Notes
Interpreted MATLAB Function, C Caller	Does not allow you to model state behavior.
MATLAB Function	Allows you to model a discrete state using `persistent` variables.
Level-2 MATLAB S-Function	Allows you to model both continuous and discrete state behavior using the `ContStates` or `Dwork` run-time object methods in combination with block callback methods. For a list of supported methods, see “Level-2 MATLAB S-Function Callback Methods” in “Write Level-2 MATLAB S-Functions”.
MATLAB System	Allows you to model discrete state behavior using `DiscreteState` properties of the System object, in combination with block callback methods. This block uses System object methods for callback methods: `mdlOutputs` (`stepImpl`, `outputImpl`), `mdlUpdate` (`updateImpl`), `mdlInitializeConditions` (`resetImpl`), `mdlStart` (`setupImpl`), `mdlTerminate` (`releaseImpl`). For more information see “What Are System Objects?”.
C MEX S-Function, S-Function Builder	Allows you to model both continuous and discrete state behavior in combination with block callback methods. For more information, see “Callback Methods for C MEX S-Functions”
Simulink Function	Communicates directly with the engine. You can model the state behavior using appropriate blocks from the continuous and discrete Simulink block libraries. When multiple calls to this function originate from different callers, the state values are also persistent between these calls. For more information, see “Call a Simulink Function block from multiple sites” on page 10-133.
Subsystem	Communicates directly with the engine. You can model the state behavior using appropriate blocks from the continuous and discrete Simulink block libraries.

Simulation Performance

For most applications, all custom block types provide satisfactory simulation performance. Use the Simulink profiler to get the actual performance indication. See “How Profiler Captures Performance Data” on page 31-5 for more information.

The two categories of performance indication are the interface cost and the algorithm cost. The interface cost is the time it takes to move data from the Simulink engine into the block. The algorithm cost is the time it takes to perform the algorithm that the block implements.

Custom Block Type	Notes
Interpreted MATLAB Function	Has a slower performance due to the interface, but has the same algorithm cost as a MATLAB function. When block data (such as inputs and outputs) is accessed or returned from an Interpreted MATLAB Function block, the Simulink engine packages this data into MATLAB arrays. This packaging takes additional time and causes a temporary increase in memory during communication. If you pass large amounts of data across this interface, such as frames or arrays, the performance can be substantially slow. Once the data has been converted, the MATLAB execution engine executes the algorithm. As a result, the algorithm cost is the same as for MATLAB function.
Level-2 MATLAB S-Function	Incurs the same algorithm costs as the Interpreted MATLAB Function block, but with a slightly higher interface cost. Since MATLAB S-Functions can handle multiple inputs and outputs, the packaging is more complicated than for the Interpreted MATLAB Function block. In addition, the Simulink engine calls the MATLAB execution engine for each block method you implement, whereas the Interpreted MATLAB Function block calls the MATLAB execution engine only for the `Outputs` method.
MATLAB Function	Performs simulation through code generation and incurs the same interface cost as other Simulink built-in blocks. The algorithm cost of this block is harder to analyze because of the block's implementation. On average, a function for this block and the MATLAB function run at about the same speed. If the MATLAB Function block has code that uses `coder.extrinsic` to call out to the MATLAB execution engine, it incurs all the costs that the MATLAB S-Function or Interpreted MATLAB Function block incur. Calling out to the MATLAB execution engine from a MATLAB Function block produces a warning to prevent you from doing so unintentionally. To reduce the algorithm cost, you can disable debugging for all MATLAB Function blocks.
MATLAB System	In the interpreted execution mode, performance is similar to that of the Level-2 MATLAB S-function because the model simulates the block using the MATLAB execution engine. In the code generation mode, performance is similar to that of the MATLAB Function because the model simulates the block using the generated code. For more information, see the MATLAB Function entry in this table.
C Caller	First time model simulation with could be slower due to parsing and building of the custom code. To speed up the simulation after the first compile, from **Configuration parameters**, select **Faster Builds** for faster compile and select **Faster Runs** for faster simulation.
C MEX S-Function	Simulates via the compiled code and incurs the same interface cost as Simulink built-in blocks. The algorithm cost depends on the complexity of the S-Function.
S-Function Builder	This block only builds an S-Function from the specifications and C code you supply. You can also use this block as a wrapper for the generated S-Function in models. The algorithm cost of this block compared to C MEX S-Function is incurred only from the wrapper.

Custom Block Type	Notes
Simulink Function, Subsystem	If included in a library, introduces no interface or algorithm costs beyond what would normally be incurred if the block existed as a regular subsystem in the model. Performance is proportional to the complexity of the algorithm implemented in the subsystem. If the subsystem is contained in a library, some cost is incurred when Simulink loads any unloaded libraries the first time the diagram is updated or readied for simulation. If all referenced library blocks remain unchanged, Simulink does not subsequently reload the library. Compiling the model becomes faster than if the model did not use libraries.

Code Generation

You need code generation if your model is part of a bigger system. Not all custom block types support code generation with Simulink Coder.

Custom Block Type	Notes
Interpreted MATLAB Function	Does not support code generation.
C Caller	Supports code generation.
Level-2 MATLAB S-Function	Generates code only if you implement the algorithm using a Target Language Compiler (TLC) function. In accelerated and external mode simulations, you can choose to execute the S-Function in the interpretive mode by calling back to the MATLAB execution engine without implementing the algorithm in TLC. If the MATLAB S-Function is `SimViewingDevice`, the Simulink Coder product automatically omits the block during code generation.
MATLAB Function, MATLAB System	Supports code generation. However, if your block calls out to the MATLAB execution engine, it will build with the Simulink Coder product only if the calls to the MATLAB execution engine do not affect the block outputs. Under this condition, the Simulink Coder product omits these calls from the generated C code. This feature allows you to leave visualization code in place, even when generating embedded code.
C MEX S-Function, S-Function Builder	Both supports code generation. • For non-inlined S-Functions, the Simulink Coder product uses the C MEX function during code generation. • In the case of C MEX S-Functions, if you need to either inline the S-Function or create a wrapper for handwritten code, you must write a TLC file for the S-Function. • In the case of S-Function Builder, you can choose the Generate wrapper TLC option to automatically generate a TLC file. See “S-Functions and Code Generation” (Simulink Coder) for more information.
Simulink Function	Supports code generation.
Subsystem	Supports code generation as long as the blocks contained within the subsystem support code generation. For more information, see “Control Generation of Functions for Subsystems” (Embedded Coder)

Multiple Input and Output Ports

These types of custom blocks support multiple input and output ports.

Custom Block Type	Notes
Interpreted MATLAB Function	Supports only a single input and a single output port.
MATLAB Function	Supports multiple input and output ports, including bus signals. See “How Structure Inputs and Outputs Interface with Bus Signals” on page 44-60 for more information.
MATLAB System	Supports multiple input and output ports, including bus signals. In addition, you can modify the number of input and output ports based on system object properties using the `getNumInputs` and `getNumOutputs` methods.
C Caller	Supports multiple input and output ports, including bus signals. Complex data type is not supported.
Level-2 MATLAB S-Function, C MEX S-Function, S-Function Builder	Supports multiple input and output ports. In addition, you can modify the number of input and output ports based on user-defined parameters. The C MEX S-Function and S-Function Builder support bus signals.
Simulink Function	Supports multiple input and output ports, including bus signals.
Subsystem	Supports multiple input and output ports, including bus signals. In addition, you can modify the number of input and output ports based on user-defined parameters. See “Self-Modifiable Linked Subsystems” on page 41-16 for more information.

Speed of Updating the Simulink Diagram

Simulink updates the diagram before every simulation and when requested by the user. Every block introduces some overhead into the diagram update process.

Custom Block Type	Notes
Interpreted MATLAB Function	Low diagram update cost.
MATLAB Function, C Caller	Simulation is performed through code generation, so this blocks can take a significant amount of time when first updated. However, because code generation is incremental, Simulink does not repeatedly update the block if the block and the signals connected to it have not changed.
MATLAB System	Faster than MATLAB Function because code is not generated to update the diagram. Since, code generation is incremental, Simulink does not repeatedly update the block if the block and the signals connected to it have not changed.
C MEX S-Function, Level-2 MATLAB S-Function	Incurs greater costs than other Simulink blocks only if it overrides methods executed when updating the diagram. If these methods become complex, they can contribute significantly to the time it takes to update the diagram. For a list of methods executed when updating the diagram, see the process view in “Simulink Engine Interaction with C S-Functions”. When updating the diagram, Simulink invokes all relevant methods in the model initialization phase up to, but not including, `mdlStart`.

Custom Block Type	Notes
Simulink Function, Subsystem	The speed is proportional to the complexity of the algorithm implemented in the subsystem. If the subsystem is contained in a library, some cost is incurred when Simulink loads any unloaded libraries the first time the diagram is updated or readied for simulation. If all referenced library blocks remain unchanged, Simulink does not subsequently reload the library. Compiling the model becomes faster than if the model does not use libraries.

Callback Methods

Simulink blocks communicate with the Simulink engine through block callback methods, which fully specify the behavior of blocks (except the Simulink Function block). Each custom block type allows you to implement a different set of callback methods. To learn how blocks interact with Simulink engine, see “Simulink Engine Interaction with C S-Functions”. This table uses “S-Function Callback Methods” names as equivalents.

Custom Block Type	Notes
Interpreted MATLAB Function, MATLAB Function, C Caller	All create a `mdlOutputs` method to calculate the value of outputs given the value of inputs. You cannot implement any other callback methods using one of these blocks and, therefore, cannot model state behavior.
Level-2 MATLAB S-Function	Allows implementation of a larger subset of callback methods, including methods you can use to model continuous and discrete states. For a list of supported methods, see “Level-2 MATLAB S-Function Callback Methods” in “Write Level-2 MATLAB S-Functions”.
MATLAB System	Uses System object methods for callback methods: `mdlOutputs` (`stepImpl`, `outputImpl`), `mdlUpdate` (`updateImpl`), `mdlInitializeConditions` (`resetImpl`), `mdlStart` (`setupImpl`), `mdlTerminate` (`releaseImpl`). For more information, see “Simulink Engine Interaction with System Object Methods” on page 45-24
C MEX S-Function	Allows implementation of a complete set of callback methods.
S-Function Builder	Allows implementation of `mdlOutputs`, `mdlDerivatives` and `mdlUpdate`.
Simulink Function	Packaged as a standalone function. Any caller to this function becomes part of one of the callback methods based on the caller’s location.
Subsystem	Communicates directly with the engine. You can model state behaviors using appropriate blocks from the continuous and discrete Simulink block libraries.

Comparing MATLAB S-Functions to MATLAB Functions for Code Generation

MATLAB S-functions and MATLAB functions for code generation have some fundamental differences.

- The Simulink Coder product can generate code for both MATLAB S-functions and MATLAB functions for code generation. However, MATLAB S-functions require a Target Language Compiler (TLC) file for code generation. MATLAB functions for code generation do not require a TLC file.
- MATLAB S-functions can use any MATLAB function whereas MATLAB functions for code generation are a subset of the MATLAB language. For a list of supported functions for code generation, see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2.

- MATLAB S-functions can model discrete and continuous state dynamics whereas MATLAB functions for code generation cannot model state dynamics.

Expanding Custom Block Functionality

You can expand the functionality of any custom block using callbacks and MATLAB graphics.

Block callbacks perform user-defined actions at specific points in the simulation. For example, the callback can load data into the MATLAB workspace before the simulation or generate a graph of simulation data at the end of the simulation. You can assign block callbacks to any of the custom block types. For a list of available callbacks and more information on how to use them, see “Specify Block Callbacks” on page 4-49.

App Designer, the MATLAB graphical user interface development environment, provides tools for easily creating custom user interfaces. See “App Building” for more information on using App Designer.

See Also

Simulink Function | Interpreted MATLAB Function | Level-2 MATLAB S-Function | MATLAB Function | MATLAB System | S-Function | S-Function Builder | Subsystem

More About

- “Types of Custom Blocks” on page 40-2
- “Design and Create a Custom Block” on page 40-12

Design and Create a Custom Block

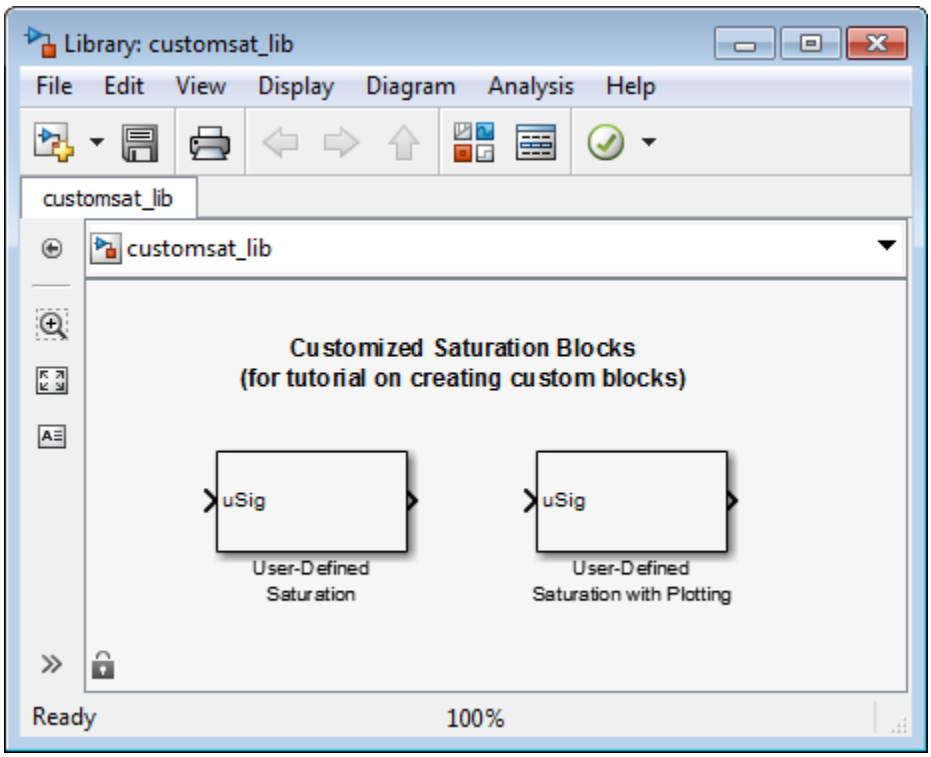
In this section...

How to Design a Custom Block

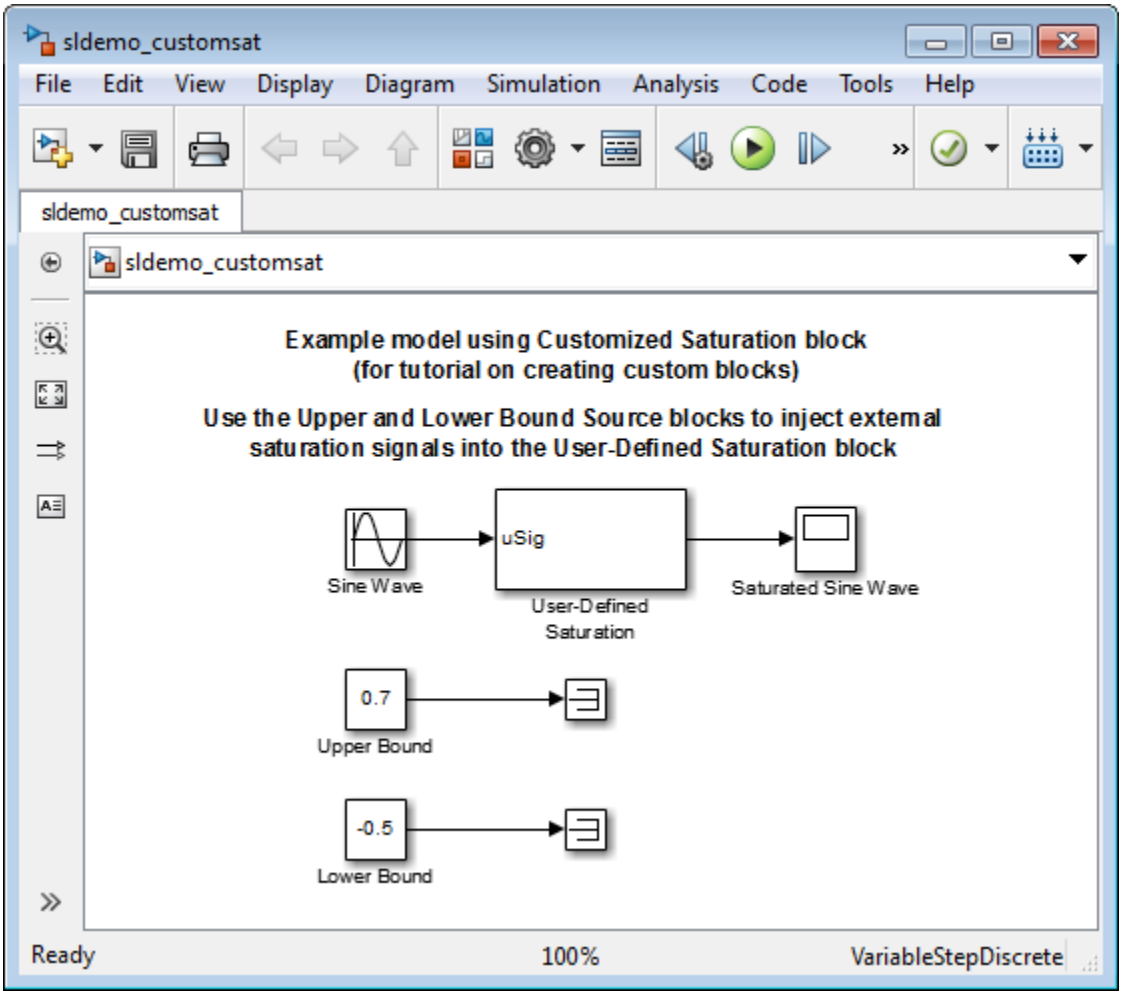
In general, use the following process to design a custom block:

1. “Defining Custom Block Behavior” on page 40-13
2. “Deciding on a Custom Block Type” on page 40-14
3. “Placing Custom Blocks in a Library” on page 40-17
4. “Adding a User Interface to a Custom Block” on page 40-18

Suppose you want to create a customized saturation block that limits the upper and lower bounds of a signal based on either a block parameter or the value of an input signal. In a second version of the block, you want the option to plot the saturation limits after the simulation is finished. The following tutorial steps you through designing these blocks. The library `ex_customsat_lib` contains the two versions of the customized saturation block.



The example model `sldemo_customsat` uses the basic version of the block.



Defining Custom Block Behavior

Begin by defining the features and limitations of your custom block. In this example, the block supports the following features:

- Turning on and off the upper or lower saturation limit.
- Setting the upper and/or lower limits via a block parameters.
- Setting the upper and/or lower limits using an input signal.

It also has the following restrictions:

- The input signal under saturation must be a scalar.
- The input signal and saturation limits must all have a data type of double.
- Code generation is not required.

Deciding on a Custom Block Type

Based on the custom block features, the implementation needs to support the following:

- Multiple input ports
- A relatively simple algorithm
- No continuous or discrete system states

Therefore, this tutorial implements the custom block using a Level-2 MATLAB S-function. MATLAB S-functions support multiple inputs and, because the algorithm is simple, do not have significant overhead when updating the diagram or simulating the model. See “Comparison of Custom Block Functionality” on page 40-5 for a description of the different functionality provided by MATLAB S-functions as compared to other types of custom blocks.

Parameterizing the MATLAB S-Function

Begin by defining the S-function parameters. This example requires four parameters:

- The first parameter indicates how the upper saturation limit is set. The limit can be off, set via a block parameter, or set via an input signal.
- The second parameter is the value of the upper saturation limit. This value is used only if the upper saturation limit is set via a block parameter. In the event this parameter is used, you should be able to change the parameter value during the simulation, i.e., the parameter is tunable.
- The third parameter indicates how the lower saturation limit is set. The limit can be off, set via a block parameter, or set via an input signal.
- The fourth parameter is the value of the lower saturation limit. This value is used only if the lower saturation limit is set via a block parameter. As with the upper saturation limit, this parameter is tunable when in use.

The first and third S-function parameters represent modes that must be translated into values the S-function can recognize. Therefore, define the following values for the upper and lower saturation limit modes:

- `1` indicates that the saturation limit is off.
- `2` indicates that the saturation limit is set via a block parameter.
- `3` indicates that the saturation limit is set via an input signal.

Writing the MATLAB S-Function

After you define the S-function parameters and functionality, write the S-function. The template `msfuntmpl.m` provides a starting point for writing a Level-2 MATLAB S-function. You can find a completed version of the custom saturation block in the file `custom_sat.m`. Save this file to your working folder before continuing with this tutorial.

This S-function modifies the S-function template as follows:

- The `setup` function initializes the number of input ports based on the values entered for the upper and lower saturation limit modes. If the limits are set via input signals, the method adds input ports to the block. The `setup` method then indicates there are four S-function parameters and sets the parameter tunability. Finally, the method registers the S-function methods used during simulation.

```
function setup(block)

% The Simulink engine passes an instance of the Simulink.MSFcnRunTimeBlock
% class to the setup method in the input argument "block". This is known as
% the S-function block's run-time object.

% Register original number of input ports based on the S-function
% parameter values

try % Wrap in a try/catch, in case no S-function parameters are entered
    lowMode    = block.DialogPrm(1).Data;
    upMode     = block.DialogPrm(3).Data;
    numInPorts = 1 + isequal(lowMode,3) + isequal(upMode,3);
catch
    numInPorts=1;
end % try/catch
block.NumInputPorts = numInPorts;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';

% Override output port properties
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';

% Register parameters. In order:
% -- If the upper bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The upper limit value. Should be empty if the upper limit is off or
%    set via an input signal
% -- If the lower bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The lower limit value. Should be empty if the lower limit is off or
%    set via an input signal
block.NumDialogPrms     = 4;
block.DialogPrmsTunable = {'Nontunable','Tunable','Nontunable', ...
    'Tunable'};

% Register continuous sample times [0 offset]
block.SampleTimes = [0 0];

%% -----------------------------------------------------------------
%% Options
%% -----------------------------------------------------------------
% Specify if Accelerator should use TLC or call back into
% MATLAB script
block.SetAccelRunOnTLC(false);

%% -----------------------------------------------------------------
%% Register methods called during update diagram/compilation
%% -----------------------------------------------------------------

block.RegBlockMethod('CheckParameters',      @CheckPrms);
block.RegBlockMethod('ProcessParameters',    @ProcessPrms);
block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Outputs',              @Outputs);
block.RegBlockMethod('Terminate',            @Terminate);
%end setup function
```

- The `CheckParameters` method verifies the values entered into the Level-2 MATLAB S-Function block.

```
function CheckPrms(block)

lowMode = block.DialogPrm(1).Data;
lowVal  = block.DialogPrm(2).Data;
```

```
upMode  = block.DialogPrm(3).Data;
upVal   = block.DialogPrm(4).Data;

% The first and third dialog parameters must have values of 1-3
if ~any(upMode == [1 2 3]);
    error('The first dialog parameter must be a value of 1, 2, or 3');
end

if ~any(lowMode == [1 2 3]);
    error('The first dialog parameter must be a value of 1, 2, or 3');
end

% If the upper or lower bound is specified via a dialog, make sure there
% is a specified bound. Also, check that the value is of type double
if isequal(upMode,2),
    if isempty(upVal),
        error('Enter a value for the upper saturation limit.');
    end
    if ~strcmp(class(upVal), 'double')
        error('The upper saturation limit must be of type double.');
    end
end

if isequal(lowMode,2),
    if isempty(lowVal),
        error('Enter a value for the lower saturation limit.');
    end
    if ~strcmp(class(lowVal), 'double')
        error('The lower saturation limit must be of type double.');
    end
end

% If a lower and upper limit are specified, make sure the specified
% limits are compatible.
if isequal(upMode,2) && isequal(lowMode,2),
    if lowVal >= upVal,
        error('The lower bound must be less than the upper bound.');
    end
end

%end CheckPrms function
```

- The `ProcessParameters` and `PostPropagationSetup` methods handle the S-function parameter tuning.

```
function ProcessPrms(block)

%% Update run time parameters
block.AutoUpdateRuntimePrms;

%end ProcessPrms function


function DoPostPropSetup(block)

%% Register all tunable parameters as runtime parameters.
block.AutoRegRuntimePrms;

%end DoPostPropSetup function
```

- The `Outputs` method calculates the block's output based on the S-function parameter settings and any input signals.

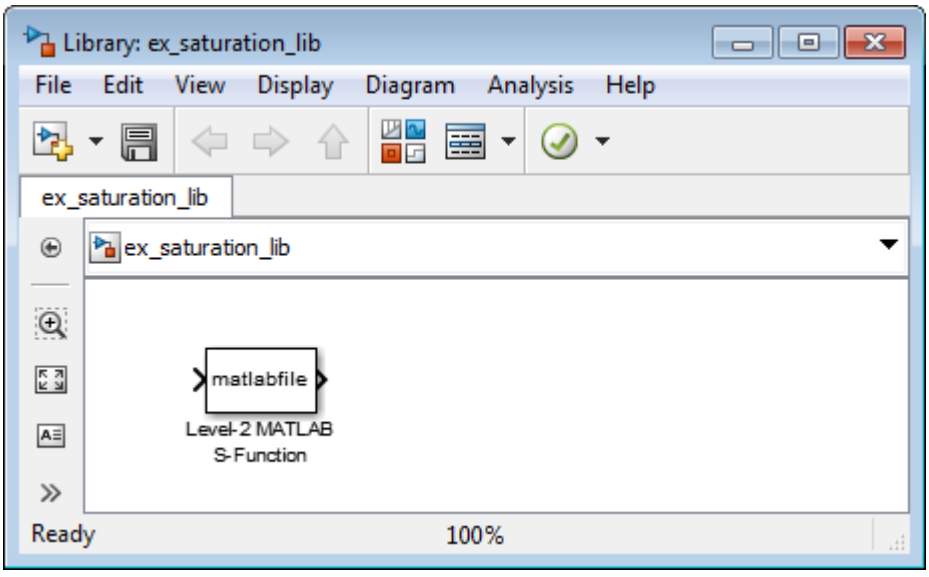
```
function Outputs(block)

lowMode    = block.DialogPrm(1).Data;
upMode     = block.DialogPrm(3).Data;
sigVal     = block.InputPort(1).Data;
lowPortNum = 2; % Initialize potential input number for lower saturation limit

% Check upper saturation limit
if isequal(upMode,2), % Set via a block parameter
```

```
    upVal = block.RuntimePrm(2).Data;
elseif isequal(upMode,3), % Set via an input port
    upVal = block.InputPort(2).Data;
    lowPortNum = 3; % Move lower boundary down one port number
else
    upVal = inf;
end

% Check lower saturation limit
if isequal(lowMode,2), % Set via a block parameter
    lowVal = block.RuntimePrm(1).Data;
elseif isequal(lowMode,3), % Set via an input port
    lowVal = block.InputPort(lowPortNum).Data;
else
    lowVal = -inf;
end

% Assign new value to signal
if sigVal > upVal,
    sigVal = upVal;
elseif sigVal < lowVal,
    sigVal=lowVal;
end

block.OutputPort(1).Data = sigVal;

%end Outputs function
```

Placing Custom Blocks in a Library

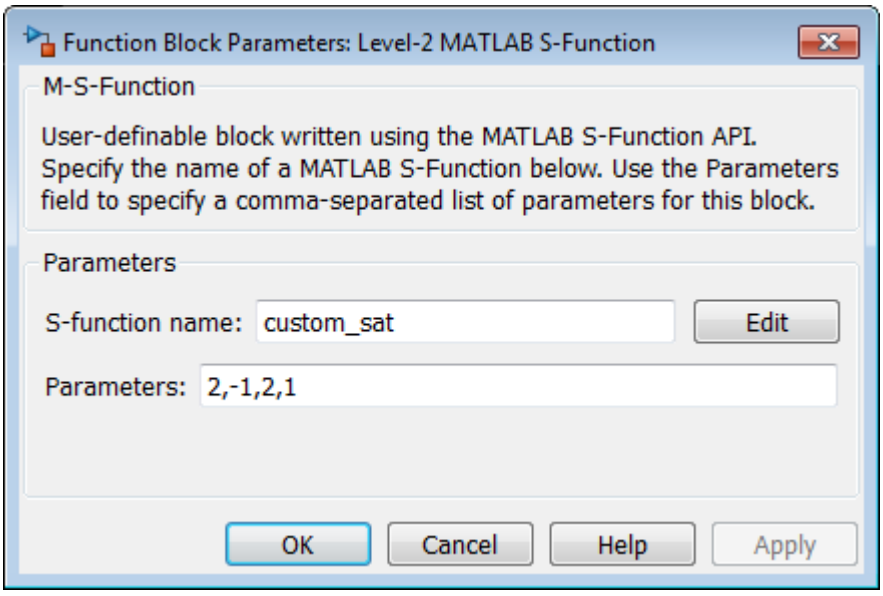
Libraries allow you to share your custom blocks with other users, easily update the functionality of copies of the custom block, and collect blocks for a particular project into a single location. This example places the custom saturation block into a library.

1. In the Simulink Editor, in the **Simulation** tab, select **New > Library**.
2. From the User-Defined Functions library, drag a Level-2 MATLAB S-Function block into your new library.



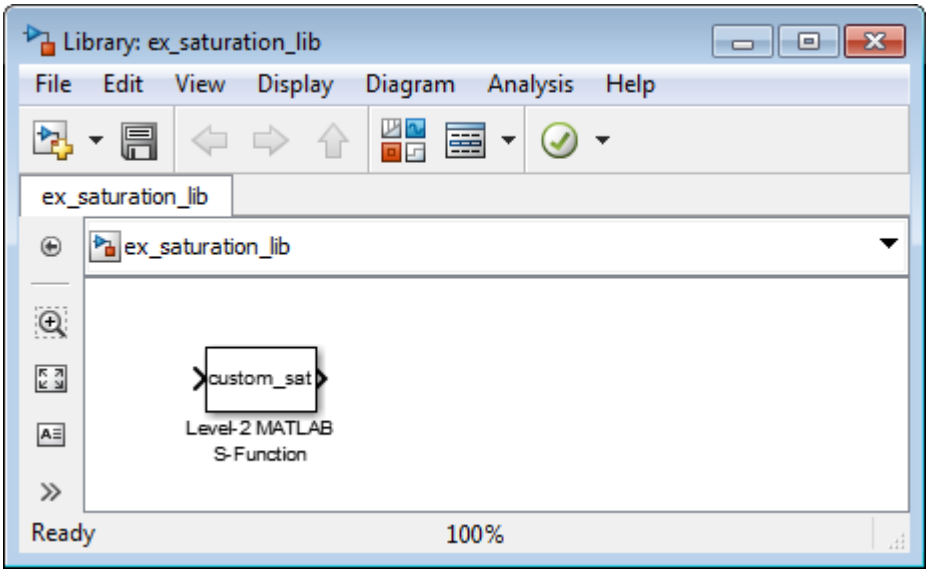
3. Save your library with the filename `saturation_lib`.
4. Double-click the block to open its Function Block Parameters dialog box.

5 In the **S-function name** field, enter the name of the S-function. For example, enter `custom_sat`. In the **Parameters** field enter `2,-1,2,1`.



6 Click **OK**.

You have created a custom saturation block that you can share with other users.



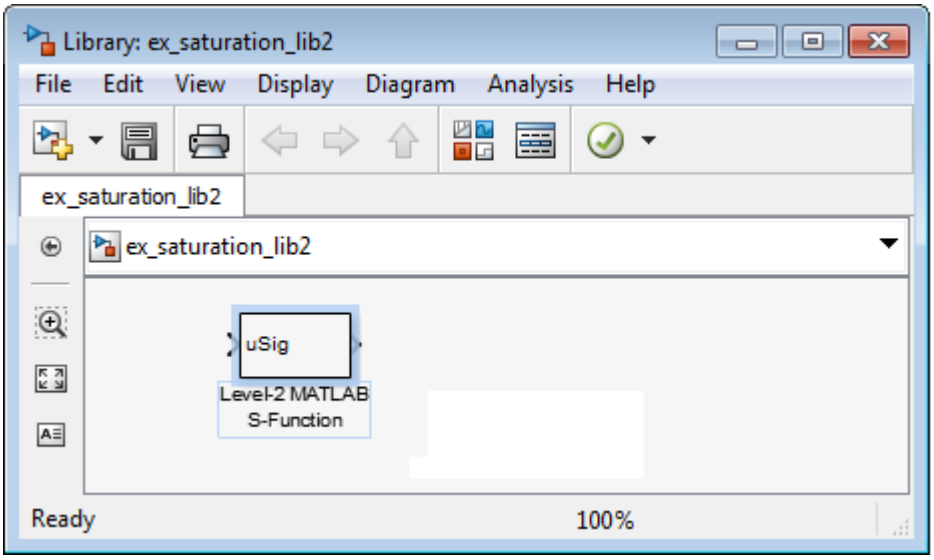
You can make the block easier to use by adding a customized user interface.

Adding a User Interface to a Custom Block

You can create a block dialog box for a custom block using the masking features of Simulink. Masking the block also allows you to add port labels to indicate which ports corresponds to the input signal and the saturation limits.

1. Open the library `saturation_lib` that contains the custom block you created,
2. Right-click the Level-2 MATLAB S-Function block and select **Mask > Create Mask**.
3. On the **Icon & Ports** pane in the **Icons drawing commands** box, enter `port_label('input',1,'uSig')`, and then click **Apply**.

   This command labels the default port as the input signal under saturation.



4. In the **Parameters & Dialog** pane, add four parameters corresponding to the four S-Function parameters. For each new parameter, drag a popup or edit control to the **Dialog box** section, as shown in the table. Drag each parameter into the Parameters group.

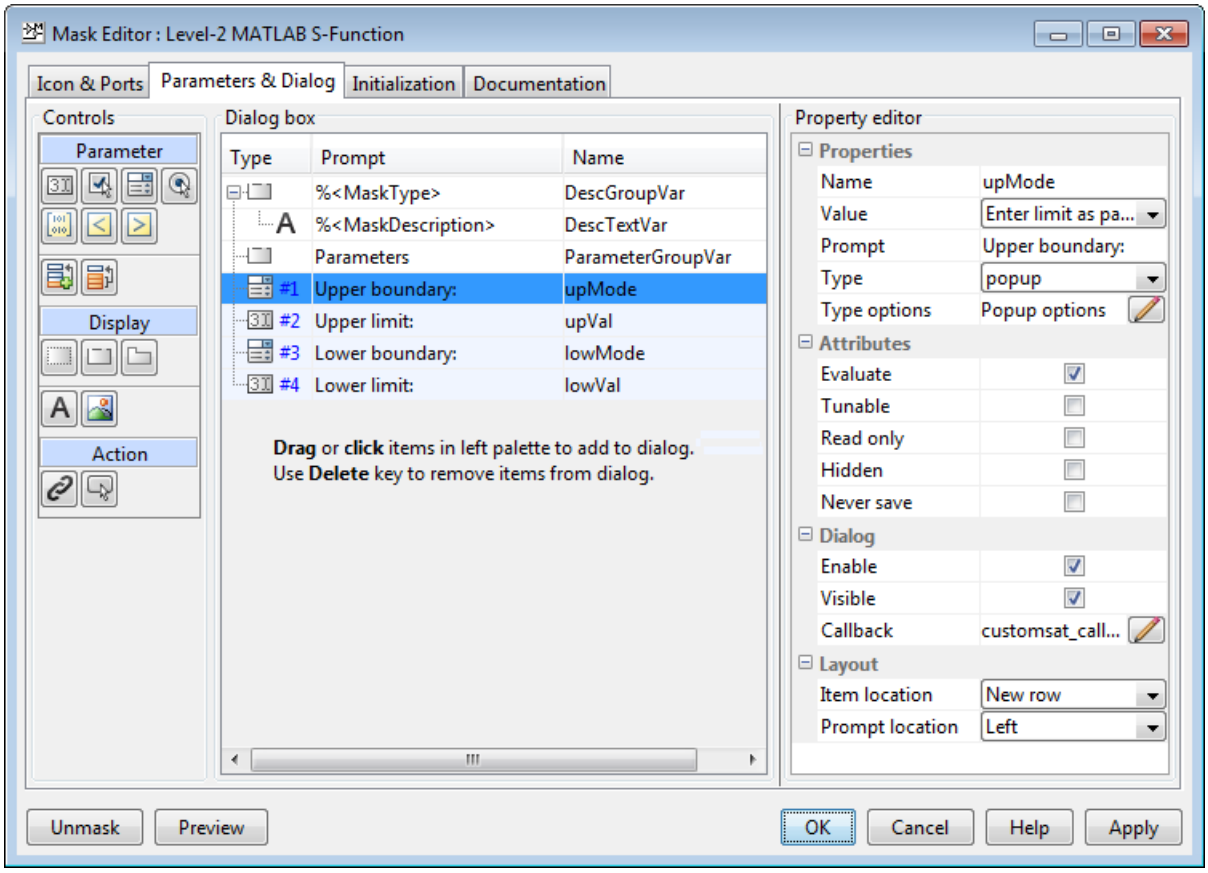
Type	Prompt	Name	Evaluate	Tunable	Popup options	Callback
`popup`	Upper boundary:	`upMode`	✓		No limit Enter limit as parameter Limit using input signal	`customsat_callback('upperbound_callback', gcb)`
`edit`	Upper limit:	`upVal`	✓	✓	N/A	`customsat_callback('upperparam_callback', gcb)`

Type	Prompt	Name	Evaluate	Tunable	Popup options	Callback
`popup`	Lower boundary:	`lowMode`	✓		No limit Enter limit as parameter Limit using input signal	`customsat_callback('lowerbound_callback', gcb)`

Type	Prompt	Name	Evaluate	Tunable	Popup options	Callback
edit	Lower limit:	lowVal	✓	✓	N/A	customsat_callback('lowerparam_callback', gcb)

The MATLAB S-Function script `custom_sat_final.m` contains the mask parameter callbacks. Save `custom_sat_final.m` to your working folder to define the callbacks in this example. This MATLAB script has two input arguments. The first input argument is a character vector indicating which mask parameter invoked the callback. The second input argument is the handle to the associated Level-2 MATLAB S-Function block.

The figure shows the completed **Parameters & Dialog** pane in the Mask Editor.



5 In the **Initialization** pane, select the **Allow library block to modify its contents** check box. This setting allows the S-function to change the number of ports on the block.

6 In the **Documentation** pane:

- In the **Mask type** field, enter

  ```
  Customized Saturation
  ```

- In the **Mask description** field, enter

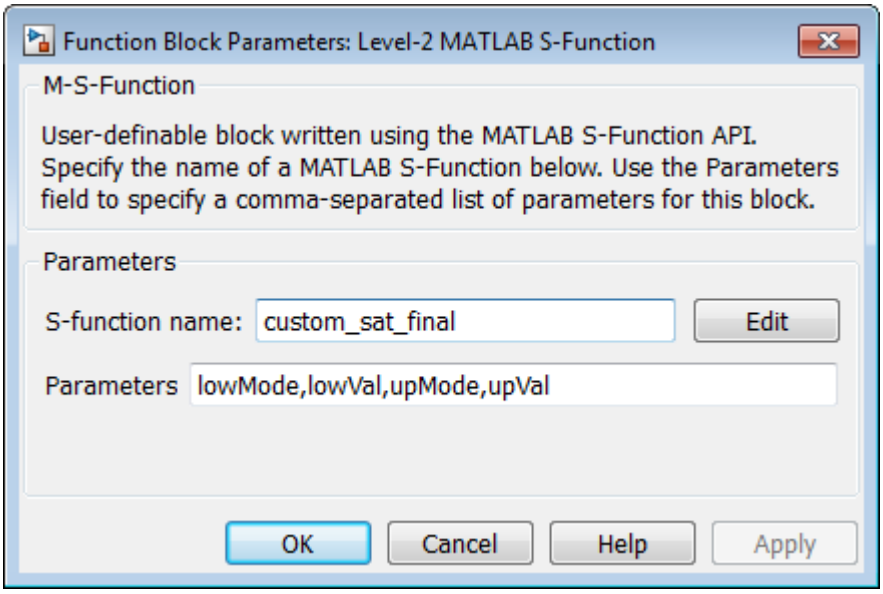
  ```
  Limit the input signal to an upper and lower saturation value
  set either through a block parameter or input signal.
  ```

7 Click **OK**.

8 To map the S-function parameters to the mask parameters, right-click the Level-2 MATLAB S-Function block and select **Mask > Look Under Mask**.

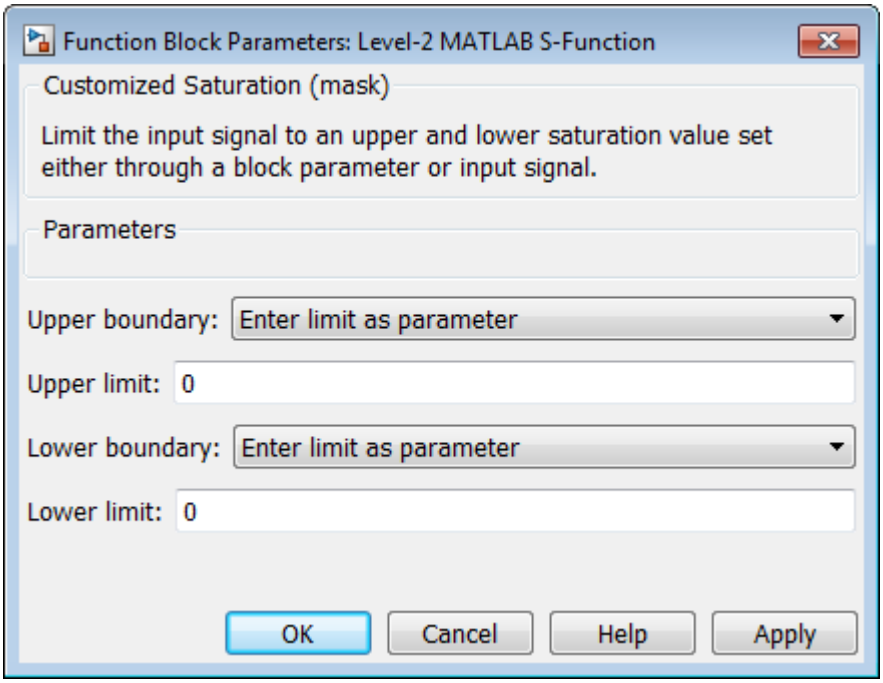
9 Change the **S-function name** field to `custom_sat_final` and the **Parameters** field to `lowMode,lowVal,upMode,upVal`.

The figure shows the Function Block Parameters dialog box after the changes.



10 Click **OK**. Save and close the library to exit the edit mode.

11 Reopen the library and double-click the customized saturation block to open the masked parameter dialog box.



To create a more complicated user interface, place a MATLAB graphics user interface on top of the masked block. The block `OpenFcn` invokes the MATLAB graphics user interface, which uses calls to `set_param` to modify the S-function block parameters based on settings in the user interface.

Writing the Mask Callback

The function `customsat_callback.m` contains the mask callback code for the custom saturation block mask parameter dialog box. This function invokes local functions corresponding to each mask parameter through a call to `feval`.

The following local function controls the visibility of the upper saturation limit's field based on the selection for the upper saturation limit's mode. The callback begins by obtaining values for all mask parameters using a call to `get_param` with the property name `MaskValues`. If the callback needed the value of only one mask parameter, it could call `get_param` with the specific mask parameter name, for example, `get_param(block,'upMode')`. Because this example needs two of the mask parameter values, it uses the `MaskValues` property to reduce the calls to `get_param`.

The callback then obtains the visibilities of the mask parameters using a call to `get_param` with the property name `MaskVisbilities`. This call returns a cell array of character vectors indicating the visibility of each mask parameter. The callback alters the values for the mask visibilities based on the selection for the upper saturation limit's mode and then updates the port label text.

The callback finally uses the `set_param` command to update the block's `MaskDisplay` property to label the block's input ports.

```
function customsat_callback(action,block)
% CUSTOMSAT_CALLBACK contains callbacks for custom saturation block

%   Copyright 2003-2007 The MathWorks, Inc.

%% Use function handle to call appropriate callback
feval(action,block)

%% Upper bound callback
function upperbound_callback(block)

vals = get_param(block,'MaskValues');
vis = get_param(block,'MaskVisibilities');
portStr = {'port_label(''input'',1,''uSig'')'};
switch vals{1}
    case 'No limit'
        set_param(block,'MaskVisibilities',[vis(1);{'off'};vis(3:4)]);
    case 'Enter limit as parameter'
        set_param(block,'MaskVisibilities',[vis(1);{'on'};vis(3:4)]);
    case 'Limit using input signal'
        set_param(block,'MaskVisibilities',[vis(1);{'off'};vis(3:4)]);
        portStr = [portStr;{'port_label(''input'',2,''up'')'}];
end
if strcmp(vals{3},'Limit using input signal'),
    portStr = [portStr;{['port_label(''input'',',num2str(length(portStr)+1), ...
        ',''low'')']}];
end
set_param(block,'MaskDisplay',char(portStr));
```

The final call to `set_param` invokes the `setup` function in the MATLAB S-function `custom_sat.m`. Therefore, the `setup` function can be modified to set the number of input ports based on the mask parameter values instead of on the S-function parameter values. This change to the `setup` function keeps the number of ports on the Level-2 MATLAB S-Function block consistent with the values shown in the mask parameter dialog box.

The modified MATLAB S-function `custom_sat_final.m` contains the following new `setup` function. If you are stepping through this tutorial, open the file and save it to your working folder.

```
%% Function: setup ===================================================
function setup(block)

% Register original number of ports based on settings in Mask Dialog
ud = getPortVisibility(block);
numInPorts = 1 + isequal(ud(1),3) + isequal(ud(2),3);

block.NumInputPorts = numInPorts;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';

% Override output port properties
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';

% Register parameters. In order:
% -- If the upper bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The upper limit value. Should be empty if the upper limit is off or
%    set via an input signal
% -- If the lower bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The lower limit value. Should be empty if the lower limit is off or
%    set via an input signal
block.NumDialogPrms     = 4;
block.DialogPrmsTunable = {'Nontunable','Tunable','Nontunable','Tunable'};

% Register continuous sample times [0 offset]
block.SampleTimes = [0 0];

%% -----------------------------------------------------------------
%% Options
%% -----------------------------------------------------------------
% Specify if Accelerator should use TLC or call back into
% MATLAB script
block.SetAccelRunOnTLC(false);

%% -----------------------------------------------------------------
%% Register methods called during update diagram/compilation
%% -----------------------------------------------------------------

block.RegBlockMethod('CheckParameters',      @CheckPrms);
block.RegBlockMethod('ProcessParameters',    @ProcessPrms);
block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Outputs',              @Outputs);
block.RegBlockMethod('Terminate',            @Terminate);
%endfunction
```

The `getPortVisibility` local function in `custom_sat_final.m` uses the saturation limit modes to construct a flag that is passed back to the `setup` function. The `setup` function uses this flag to determine the necessary number of input ports.

```
%% Function: Get Port Visibilities ====================================
function ud = getPortVisibility(block)

ud = [0 0];

vals = get_param(block.BlockHandle,'MaskValues');
switch vals{1}
    case 'No limit'
        ud(2) = 1;
    case 'Enter limit as parameter'
        ud(2) = 2;
    case 'Limit using input signal'
        ud(2) = 3;
```

```
end

switch vals{3}
    case 'No limit'
        ud(1) = 1;
    case 'Enter limit as parameter'
        ud(1) = 2;
    case 'Limit using input signal'
        ud(1) = 3;
end
```

Adding Block Functionality Using Block Callbacks

The User-Defined Saturation with Plotting block in `customsat_lib` uses block callbacks to add functionality to the original custom saturation block. This block provides an option to plot the saturation limits when the simulation ends. The following steps show how to modify the original custom saturation block to create this new block.

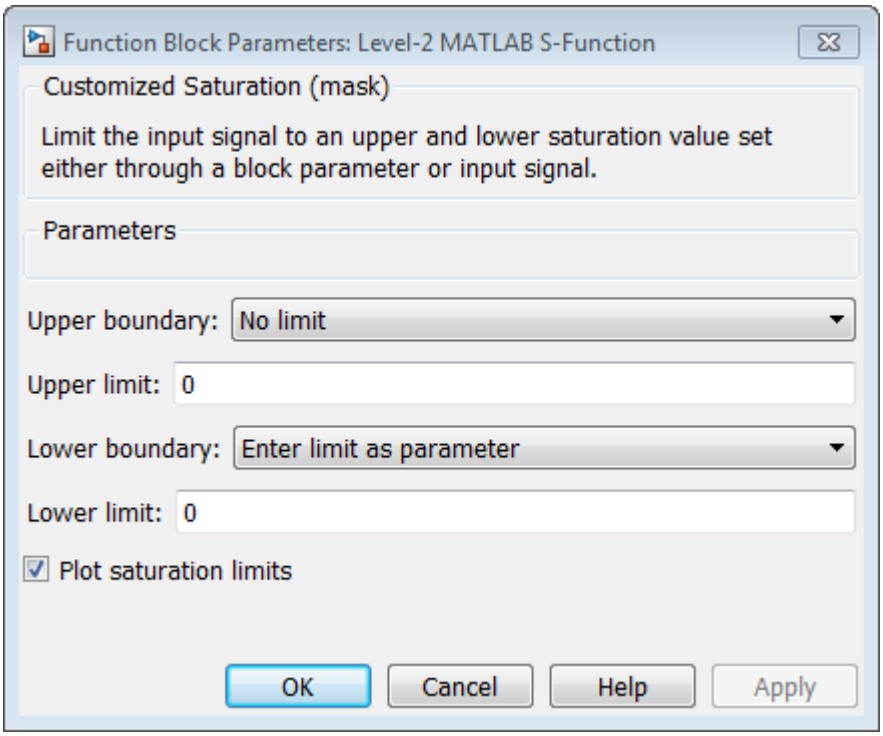
1 Add a check box to the mask parameter dialog box to toggle the plotting option on and off.

a Right-click the Level-2 MATLAB S-Function block in `saturation_lib` and select **Mask** + **Create Mask**.

b On the Mask Editor **Parameters** pane, add a fifth mask parameter with the following properties.

Prompt	Name	Type	Tunable	Type options	Callback
Plot saturation limits	`plotcheck`	`checkbox`	No	NA	`customsat_callback('plotsaturation',gcb)`

c Click **OK**.



2. Write a callback for the new check box. The callback initializes a structure to store the saturation limit values during simulation in the Level-2 MATLAB S-Function block `UserData`. The MATLAB script `customsat_plotcallback.m` contains this new callback, as well as modified versions of the previous callbacks to handle the new mask parameter. If you are following through this example, open `customsat_plotcallback.m` and copy its local functions over the previous local functions in `customsat_callback.m`.

```
%% Plotting checkbox callback
function plotsaturation(block)

% Reinitialize the block's userdata
vals = get_param(block,'MaskValues');
ud = struct('time',[],'upBound',[],'upVal',[],'lowBound',[],'lowVal',[]);

if strcmp(vals{1},'No limit'),
    ud.upBound = 'off';
else
    ud.upBound = 'on';
end

if strcmp(vals{3},'No limit'),
    ud.lowBound = 'off';
else
    ud.lowBound = 'on';
end

set_param(gcb,'UserData',ud);
```

3. Update the MATLAB S-function `Outputs` method to store the saturation limits, if applicable, as done in the new MATLAB S-function `custom_sat_plot.m`. If you are following through this example, copy the `Outputs` method in `custom_sat_plot.m` over the original `Outputs` method in `custom_sat_final.m`

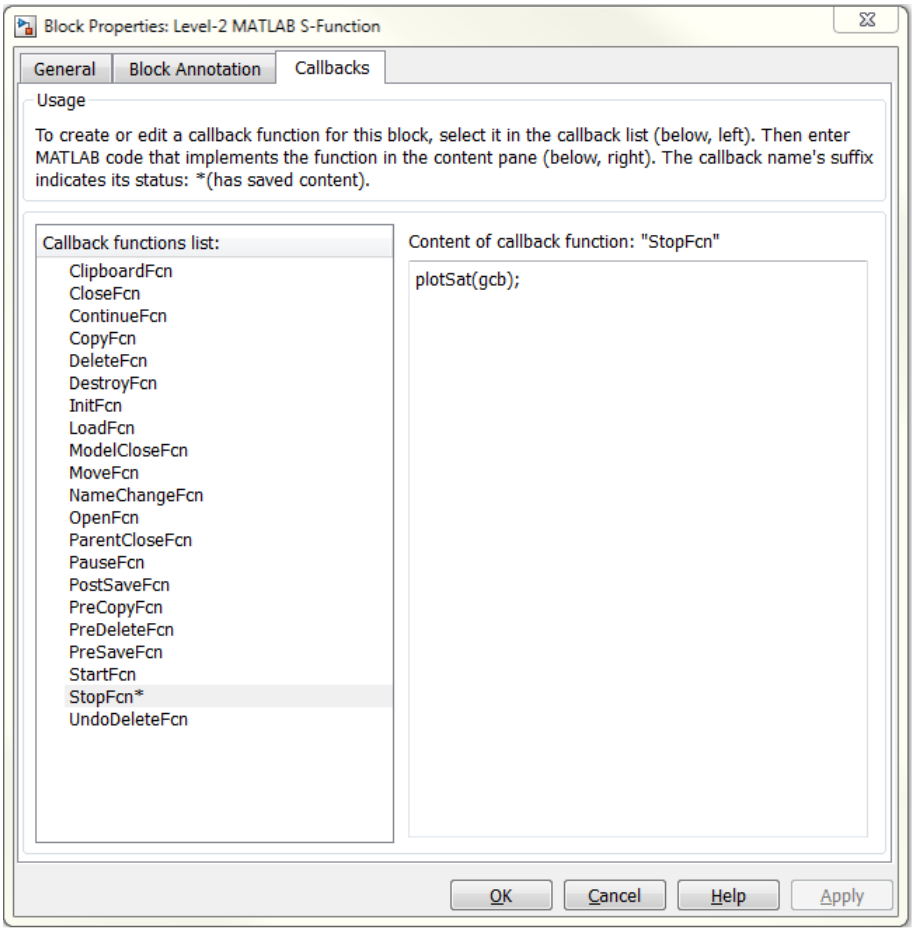
```
%% Function: Outputs ================================================
function Outputs(block)

lowMode    = block.DialogPrm(1).Data;
upMode     = block.DialogPrm(3).Data;
sigVal     = block.InputPort(1).Data;
vals = get_param(block.BlockHandle,'MaskValues');
plotFlag = vals{5};
lowPortNum = 2;

% Check upper saturation limit
if isequal(upMode,2)
    upVal = block.RuntimePrm(2).Data;
elseif isequal(upMode,3)
    upVal = block.InputPort(2).Data;
    lowPortNum = 3; % Move lower boundary down one port number
else
    upVal = inf;
end

% Check lower saturation limit
if isequal(lowMode,2),
    lowVal = block.RuntimePrm(1).Data;
elseif isequal(lowMode,3)
    lowVal = block.InputPort(lowPortNum).Data;
else
    lowVal = -inf;
end

% Use userdata to store limits, if plotFlag is on
if strcmp(plotFlag,'on');
    ud = get_param(block.BlockHandle,'UserData');
    ud.lowVal = [ud.lowVal;lowVal];
    ud.upVal = [ud.upVal;upVal];
    ud.time = [ud.time;block.CurrentTime];
    set_param(block.BlockHandle,'UserData',ud)
```

```
end

% Assign new value to signal
if sigVal > upVal,
    sigVal = upVal;
elseif sigVal < lowVal,
    sigVal=lowVal;
end

block.OutputPort(1).Data = sigVal;

%endfunction
```

4 Write the function `plotsat.m` to plot the saturation limits. This function takes the handle to the Level-2 MATLAB S-Function block and uses this handle to retrieve the block's `UserData`. If you are following through this tutorial, save `plotsat.m` to your working folder.

```
function plotSat(block)

% PLOTSAT contains the plotting routine for custom_sat_plot
%   This routine is called by the S-function block's StopFcn.

ud = get_param(block,'UserData');
fig=[];
if ~isempty(ud.time)
    if strcmp(ud.upBound,'on')
        fig = figure;
        plot(ud.time,ud.upVal,'r');
        hold on
    end
    if strcmp(ud.lowBound,'on')
        if isempty(fig),
            fig = figure;
        end
        plot(ud.time,ud.lowVal,'b');
    end
    if ~isempty(fig)
        title('Upper bound in red. Lower bound in blue.')
    end

    % Reinitialize userdata
    ud.upVal=[];
    ud.lowVal=[];
    ud.time = [];
    set_param(block,'UserData',ud);
end
```

5 Right-click the Level-2 MATLAB S-Function block and select **Properties**. The Block Properties dialog box opens. On the **Callbacks** pane, modify the `StopFcn` to call the plotting callback as shown in the following figure, then click **OK**.



See Also

More About

- “Types of Custom Blocks” on page 40-2
- “Comparison of Custom Block Functionality” on page 40-5

Working with Block Libraries

Create a Custom Library

Create a Library

You can create your own library and, optionally, add it to the Simulink Library Browser. You save a library as a `.SLX` file as you do a model. However, you cannot simulate in a library, and a library becomes locked for editing each time you close it. You must unlock a library before you make changes to it. See “Lock and Unlock Libraries” on page 41-6.

1. From the Simulink start page, select **Blank Library** and click **Create Library**.
2. Add blocks from models or libraries to the new library. Make the changes you want to the blocks, such as changing block parameters, adding masks, or adding blocks to subsystems.

   Subsystem names in a library hierarchy must be unique. For example, do not create a hierarchy such as `Subsystem_Name1/Subsystem_Name2/Subsystem_Name1`.
3. Add annotations or images. Right-click the ones you want to appear in the library in the Library Browser and select **Show in Library Browser**.
4. If you plan to add the library to the Library Browser, you can order the blocks and annotations in your library model. By default, they appear alphabetically in the Library Browser, with subsystems first, then blocks, and then annotations. The user of your library can use the Library Browser context menu to choose between viewing them in alphabetical order or the order you specified. When the user selects this option, the order in which they appear in your library model determines the order they appear on the grid in the library in the Library Browser.
5. If you want the library to appear in the Library Browser, enable the model property `EnableLBRepository` before you save the library.

   ```
   set_param(gcs,'EnableLBRepository','on');
   ```
6. Save the library.

   Where you save the library depends on how you plan to use it. If you want to add it to the Library Browser, save it to a folder on the MATLAB path or add the location to the MATLAB path. Otherwise, save it to a location where the models that use the blocks can access it.

If you want the library to appear in the Library Browser, you must also create a function `slblocks` on your MATLAB path that adds the library to the browser. For an example that shows complete steps for adding a library to the browser, see “Add Libraries to the Library Browser” on page 41-7.

Note To update the Library Browser with your custom libraries, right-click anywhere in the Library Browser library list and select **Refresh Library Browser**. Refreshing the Library Browser also updates the quick insert list to include the blocks in custom libraries currently in effect. The quick insert list lets you add blocks to a model without leaving the canvas. Click the canvas and start typing to add blocks from the quick insert list.

Blocks for Custom Libraries

Your library can contain the blocks you need, configured for your purposes. Subsystems, masked blocks, and charts in your library become linked blocks as instances in the model and stay updated if you change them in your library. Knowing about custom blocks is also useful when you create a library. See “Design and Create a Custom Block” on page 40-12.

You can create blocks in custom libraries with settings for specific purposes.

Create a Sublibrary

If your library contains many blocks, you can group the blocks into subsystems or separate sublibraries. To create a sublibrary, you create a library of the sublibrary blocks and reference the library from a Subsystem block in the parent library.

1. In the library you want to add a sublibrary to, add a Subsystem block.
2. Inside the Subsystem block, delete the default input and output ports.
3. If you want, create a mask for the subsystem that displays text or an image that conveys the sublibrary purpose.
4. In the subsystem block properties, set the `OpenFcn` callback to the name of the library you want to reference.

To learn more about masks, see “Create a Simple Mask” on page 39-6.

Prevent Library Block from Linking to Instance

You can configure a library block so the instances created from it are not linked blocks and are instead copies. Set the block’s `CopyFcn` callback.

```
set_param(gcbh,'LinkStatus','none');
```

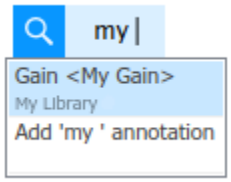
Include Block Description in Linked Block

To add a description that appears in the linked block, mask the library block and add the description in the **Documentation** pane of the mask. Descriptions added to the library block through the block’s properties do not appear on the linked block.

Configure Block with Keywords for Quick Insert

You can add one or more keywords to a block in your library. The keyword lets you add the block to your model from the quick insert prompt by entering the keyword or the block name.

For example, suppose you have a custom Gain block in your library. You can add the keyword `My Gain` to the block. Then, you can add the block to your model by entering `My Gain` at the quick insert prompt.



Note You cannot add keywords to the blocks from **Commonly Used Blocks** as they are a place to view frequently used blocks from various libraries. If you want to add keywords to a block available in **Commonly Used Blocks**, make sure that you set it in the library where the block is defined.

To add the keyword to the block in your library, use `set_param` with the `'BlockKeywords'` parameter. You can use a character vector, string scalar, or string array as the value. For example:

```
set_param(gcb,'BlockKeywords',{"My Gain","Your Gain"})
set_param(gcb,'BlockKeywords','My Integrator')
```

Note The supported special characters in keywords are `'&'`, `'('`, `')'`, `'+'`, `'@'`, `'!'`.

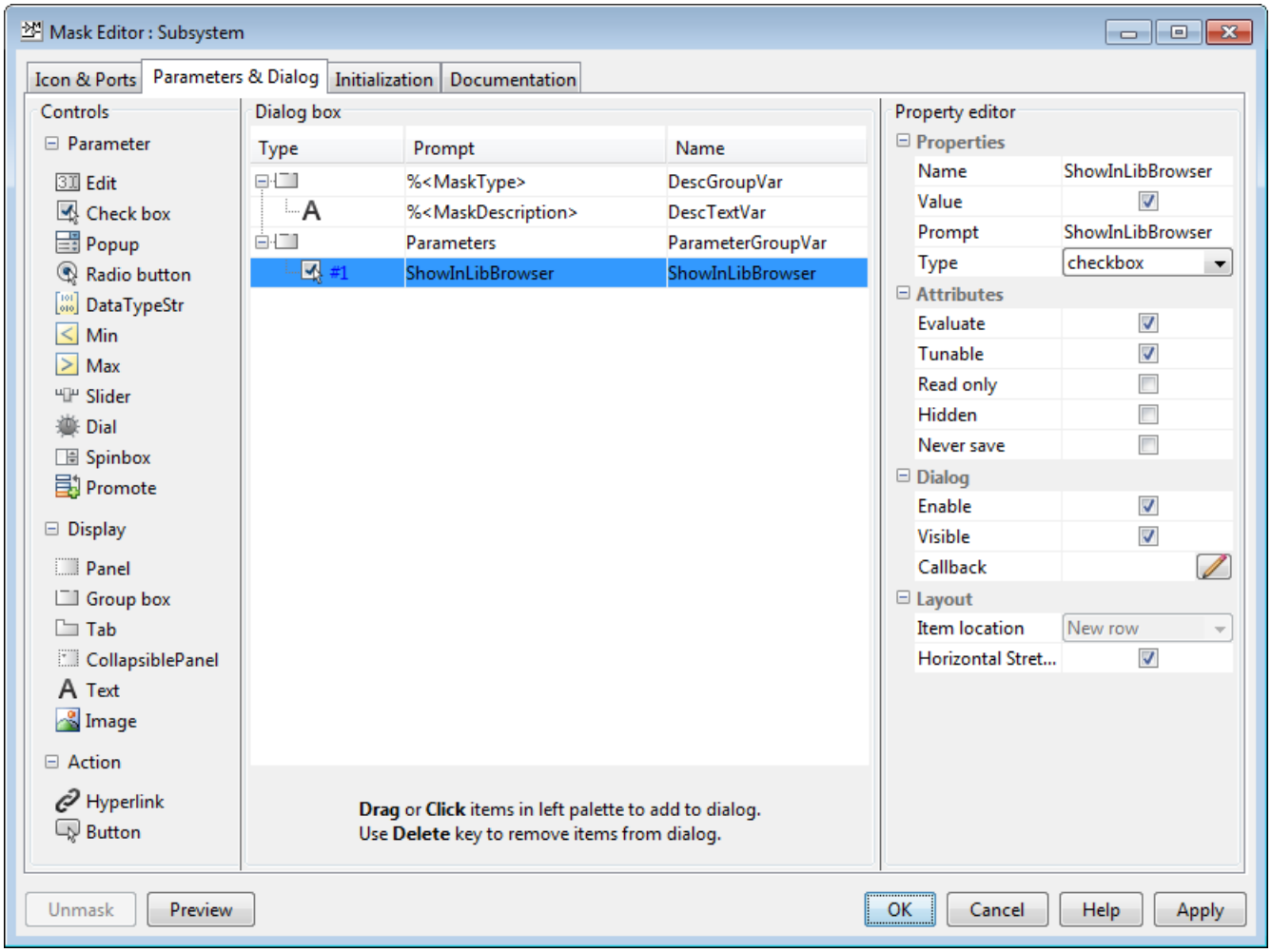
Note The Quick Insert also supports searching for blocks in languages other than English by using internationalized keywords.

Configure Subsystems with OpenFcn Callback for Library Browser

A common use of a Subsystem block in a custom library is to set the `OpenFcn` callback property to open a library, creating a library hierarchy. However, you can use the `OpenFcn` callback property of a Subsystem block for other purposes, for example to run MATLAB code or to open a link.

If your subsystem block in a library is empty and its `OpenFcn` callback contains code that performs an action other than point to a library or model, then you need to add a `'ShowInLibBrowser'` mask parameter to the subsystem to have it appear in the Library Browser.

1. Right-click the subsystem and select **Mask > Create Mask**. If the block already has a mask, select **Edit Mask** instead.
2. In the Mask Editor **Parameters & Dialog** tab, on the **Controls** pane, click **Check box**.
3. In the **Dialog box** pane, set the prompt and name for the new check box to `ShowInLibBrowser` and click **OK**.



Annotations in Custom Libraries

You can add annotations in your custom library and optionally have them appear in the Library Browser. For example, you can add an annotation that documents the library. You can also add annotations that the user of your library can add to their model from the Library Browser. Annotations can contain text and images or display an equation. Annotations can also perform an action when clicked. Learn more about annotations in “Describe Models Using Notes and Annotations” on page 4-3.

You can add callout lines from annotations to blocks in your library. However, the callouts do not appear in the Library Browser.

If you want the annotation to appear in the Library Browser, after you add it to your library, right-click it and select **Show in Library Browser**. If you want a description to appear in a tooltip when the user hovers over the annotation in the Library Browser, add the description to the annotation programmatically. At the MATLAB command prompt, enter:

```
set_param(annotationHandle,'Description','descriptionText)
```

To get the annotation handle, use `find_system`. This example gets all the annotations in the library `mylib`:

```
ann = find_system('mylib','FindAll','on','Type','annotation');
```

To get a specific annotation, turn on regular expression search and specify part of the annotation text with the `'Name'` argument:

```
ann = find_system('mylib2',FindAll','on','RegExp',...
'on','Type','annotation','Name','matchingText');
```

“Add Libraries to the Library Browser” on page 41-7 includes instructions for adding an annotation that appears in the Library Browser.

Lock and Unlock Libraries

When you close a library, it becomes locked for editing. When you next open it, unlock it if you want to make changes to it. Click the lock badge in the lower-left corner of the library to unlock it. Additionally, if you try to modify a locked library, a message prompts you to unlock it.

You can unlock a library programmatically. At the MATLAB command prompt, enter:

```
set_param('library_name','Lock','off');
```

To lock the library programmatically, enter:

```
set_param('library_name','Lock','on');
```

Prevent Disabling of Library Links

By default, a user of the blocks in your library can disable the link to library blocks. If you want to control editing of linked blocks and prevent the block user from disabling links, you can lock links to your library. Locking library links prevents the user from making any changes to the block instances.

- In your library, on the **Library** tab, click **Lock Links**.

To understand how the block user interacts with blocks from locked libraries, see “Lock Links to Blocks in a Library” on page 41-22.

See Also

More About

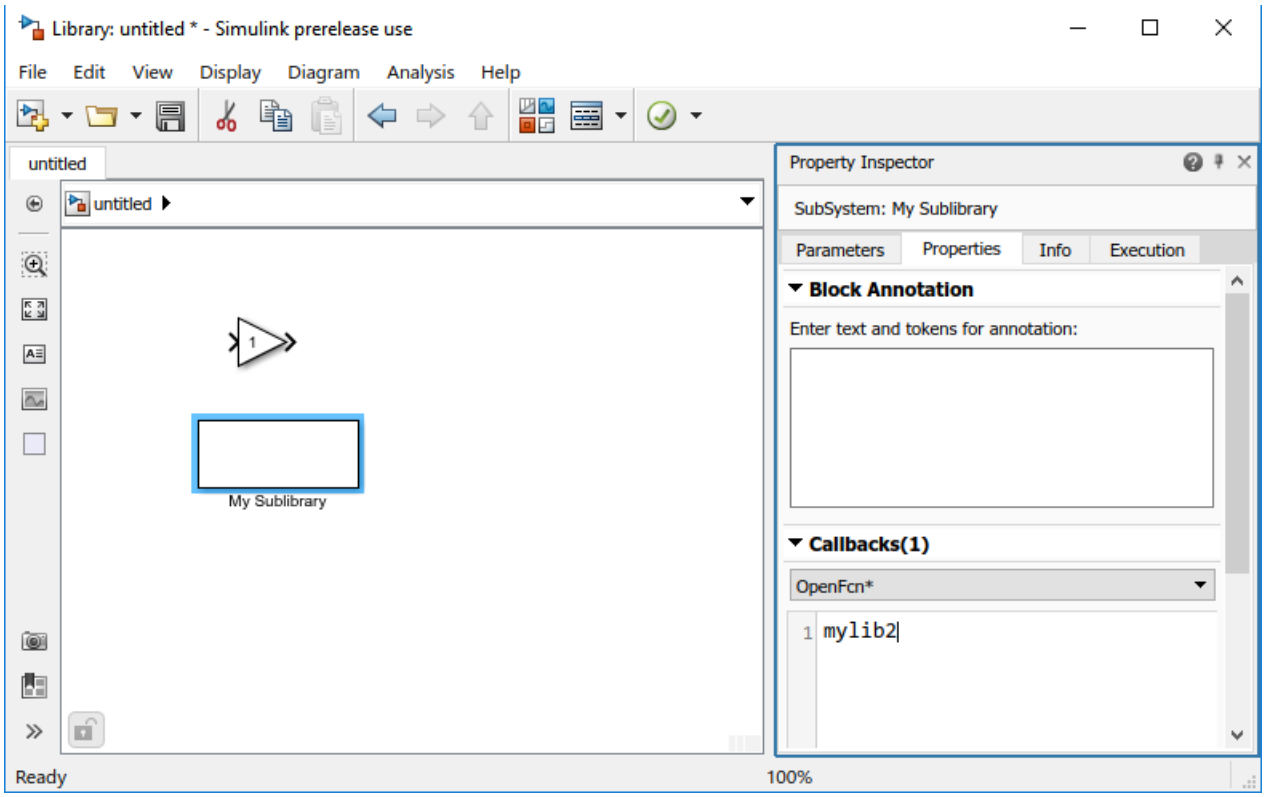
- “Choose Among Types of Model Components” on page 22-4
- “Design and Create a Custom Block” on page 40-12
- “Masking Fundamentals” on page 39-2
- “Describe Models Using Notes and Annotations” on page 4-3
- “Add Libraries to the Library Browser” on page 41-7
- “Linked Blocks” on page 41-10
- “Customize Library Browser Appearance” on page 78-19

Add Libraries to the Library Browser

This example shows how to create a block library and add it to the Simulink Library Browser. This example also shows how to add a sublibrary.

You create a function `slblocks` to specify information about your library. You can save the function as a `.m` or `.mlx` file. You cannot save it as a P-code file.

1. From the Simulink start page, select **Blank Library** and click **Create Library**.
2. Add a Gain block and a Subsystem block to the library. Remove the input and output port from the subsystem.
3. Name the Subsystem block My Sublibrary. To link it to a sublibrary, in the My Sublibrary properties, set the `OpenFcn` callback to `mylib2`.



4. At the MATLAB command prompt, enter this command to enable the model property `EnableLBRepository`. Your library can appear in the browser only if this property is on when you save your library.

```
set_param(gcs,'EnableLBRepository','on');
```

5. Save the library in a folder on the MATLAB path. For this example, name the library `mylib`.
6. Create another library `mylib2` and add some blocks to it.
7. At the MATLAB command prompt, enable the model property `EnableLBRepository` for the new library.

```
set_param(gcs,'EnableLBRepository','on');
```

8 Save `mylib2` to the same folder you saved `mylib` to.

You can close both libraries if you want.

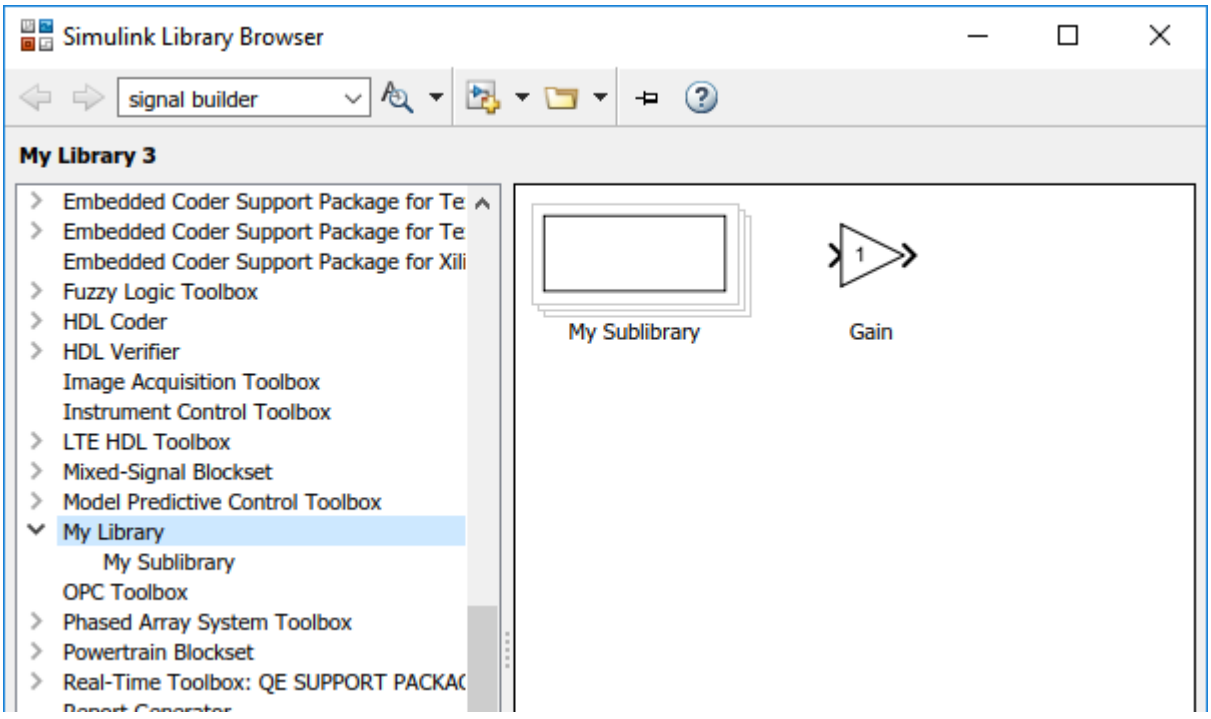
9 In MATLAB, right-click the folder you saved the library to and select **New File** > **Script**. Name the file `slblocks.m`.

10 Open `slblocks.m`. Add this function to it and save.

```
function blkStruct = slblocks
        % This function specifies that the library should appear
        % in the Library Browser
        % and be cached in the browser repository

        Browser.Library = 'mylib';
        % 'mylib' is the name of the library

        Browser.Name = 'My Library';
        % 'My Library' is the library name that appears
             % in the Library Browser

        blkStruct.Browser = Browser;
```

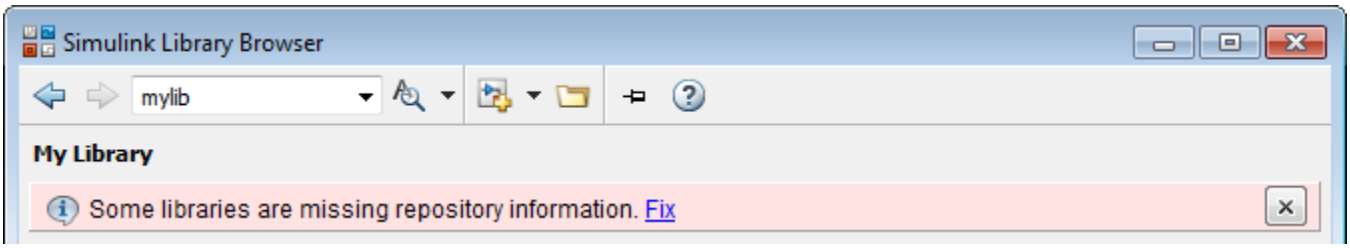
11 In the Library Browser, refresh to see the new library and sublibrary. Right-click the library list and select **Refresh Library Browser**.

The figure shows the example library `mylib` with the Library Browser name **My Library**.



Because of the callback you created, clicking My Sublibrary displays the contents of the `mylib2` library.

Note If you saved your library without setting `'EnableLBRepository'` to `'on'`, a message appears at the top of the Library Browser.



Click **Fix** and respond to the prompt as appropriate.

Specify Library Order in the Library List

You can specify the order of your library relative to the other libraries in the list by adding a `sl_customization.m` file to the MATLAB path and setting the sort priority of your library. For example, to see your library at the top of the list, you can set the sort priority to `-2`. By default, the sort priority of the Simulink library is `-1`. The other libraries have a sort priority of `0` by default, and these libraries appear below the Simulink library. Libraries with the same sort priority appear in alphabetical order.

This sample content of the `sl_customization.m` file places the new library at the top of the list of libraries.

```
function sl_customization(cm)
% Change the order of libraries in the Simulink Library Browser.
cm.LibraryBrowserCustomizer.applyOrder({'My Library',-2});
end
```

To make the customization take effect immediately, at the command prompt, enter:

```
sl_refresh_customizations
```

See Also

Related Examples

- “Create a Custom Library” on page 41-2
- “Customize Library Browser Appearance” on page 78-19

More About

- “Registering Customizations” on page 78-23

Linked Blocks

In this section...

When you add a masked library block or a Subsystem block from a Library to a Simulink model, a referenced instance of the library block is created. Such referenced instance of a library block is a called a linked block and contains link or path to the parent library block. The link or path allows the linked block to update when the library block is updated.

To optimize the performance, the child blocks (and the parameters) of a linked block are not saved with the Simulink model. When such a model is loaded and viewed, the child blocks are referenced from the parent library. If you change the parameter value of a child block of such a linked block, the changed information is saved as linked data in the model.

To locate the parent library block of a linked block, right-click the block and select **Library Link > Go To Library Link** (Ctrl + L). This option is available only for the blocks that are linked and not for the Simulink built-in blocks. To prevent unintentional disabling of library links, use the locked links option on the library. For more information, see “Lock Links to Blocks in a Library” on page 41-22.

Note The tooltip for a linked block shows the name of the referenced masked library block.

When you edit a library block (either in Simulink Editor or at the command line), Simulink updates the changes in the linked blocks. The outdated links are updated when you:

- Simulate or update the model.
- Use the `find_system` command.
- On the **Modeling** tab, select **Update Model > Refresh Blocks** (or press **Ctrl+K**).
- Load the model or library (only the visible links are updated).
- Use `get_param` to query the link status of the block (see “Control Linked Block Programmatically” on page 41-31).

  You can use the `LinkStatus` parameter or the `StaticLinkStatus` parameter to query the link status.

  - `LinkStatus`: First updates the linked block and then returns the link status.
  - `StaticLinkStatus`: Returns the link status without updating the linked block.

  Selective usage of `StaticLinkStatus` over `LinkStatus` can result in better Simulink performance. For more information on `StaticLinkStatus` and `LinkStatus`, see “Control Linked Block Programmatically” on page 41-31.

Rules for Linked Blocks

- You can change the values of a linked block parameter (including the existing mask) in the mask dialog box.
- To allow the library block initialization code to change the values of a linked block parameter, select the **Allow library block to modify its contents** check box in the **Initialization** pane of the library block.
- It is not recommended to set callback parameters for a linked block.
- If the reference library block of a linked block is a subsystem, you can make nonstructural changes such as changing the parameter value of the linked subsystem. To make structural changes to a linked block, disable the link of the linked block from its library block (See “Disable or Break Links to Library Blocks” on page 41-20).

Linked Block Terminology

Terminology	Definition
Parent library block	Library block from which the linked blocks are referenced.
Linked block	Reference instance of a library block that contains links or path to its parent library block.
Locked links	Prevents unintentional modification of a linked block. For more information, see “Lock Links to Blocks in a Library” on page 41-22.
Disabled links	Library links that are temporarily disconnected from their parent library block. For more information, see “Disable or Break Links to Library Blocks” on page 41-20.
Restore links	Restores the disabled link of a linked block to their parent library block. For more information, see “Restore Disabled Links” on page 41-24.
Break links	Permanently breaks the link of a linked block from its parent library block. For more information, see “Disable or Break Links to Library Blocks” on page 41-20.
Self-modifiable links	Linked block with the ability to have structural changes in a linked Subsystem block. For more information, see “Self-Modifiable Linked Subsystems” on page 41-16.
Parameterized links	Created when the parameter values of a linked block are modified using the MATLAB command prompt. For more information, see “Parameterized Links and Self-Modifiable Linked Subsystems” on page 41-13.
Forwarding Tables	Maps the old library block path to new library block path. For more information, see “Forwarding Tables” on page 41-34.

Terminology	Definition
Transformation function	Corrects the mismatch of parameters in the `InstanceData` of the new and old library links to ensure that the library links continue to work. For more information, see “Transformation Functions” on page 41-37.

See Also

`find_system`

More About

- “Choose Among Types of Model Components” on page 22-4
- “Display Library Links” on page 41-18
- “Control Linked Block Programmatically” on page 41-31
- “Fix Unresolved Library Links” on page 41-29

Parameterized Links and Self-Modifiable Linked Subsystems

You can use the MATLAB command prompt to change the value of a parameter in a linked block. Such parameter changes on the linked block result in parameterized links.

Similarly, you can also modify the structure of a linked Subsystem block without changing the parent library block. Such changes can be applied using the mask initialization code and is called self-modifiable linked subsystem.

Parameterized Links

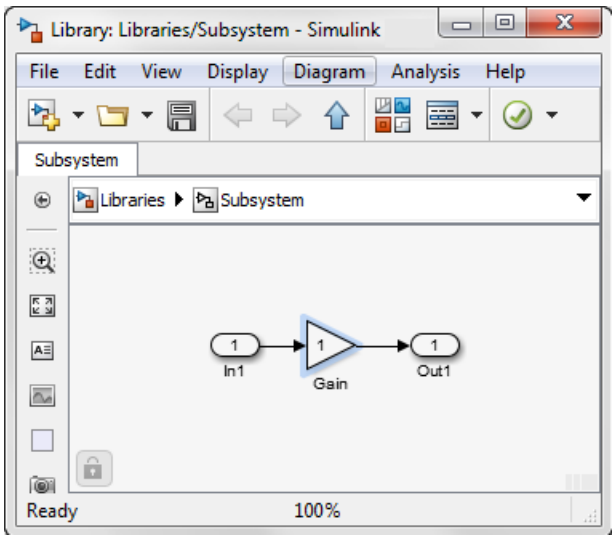
A parameterized link is created when you change the parameter values of the child blocks of a masked subsystem linked block.

A parameterized link allows you to have a different parameter value for the linked block and the parent library block. For such library blocks, the link to the parent block is still retained.

Note Changing the mask value of a parent library block does not create a parameterized link.

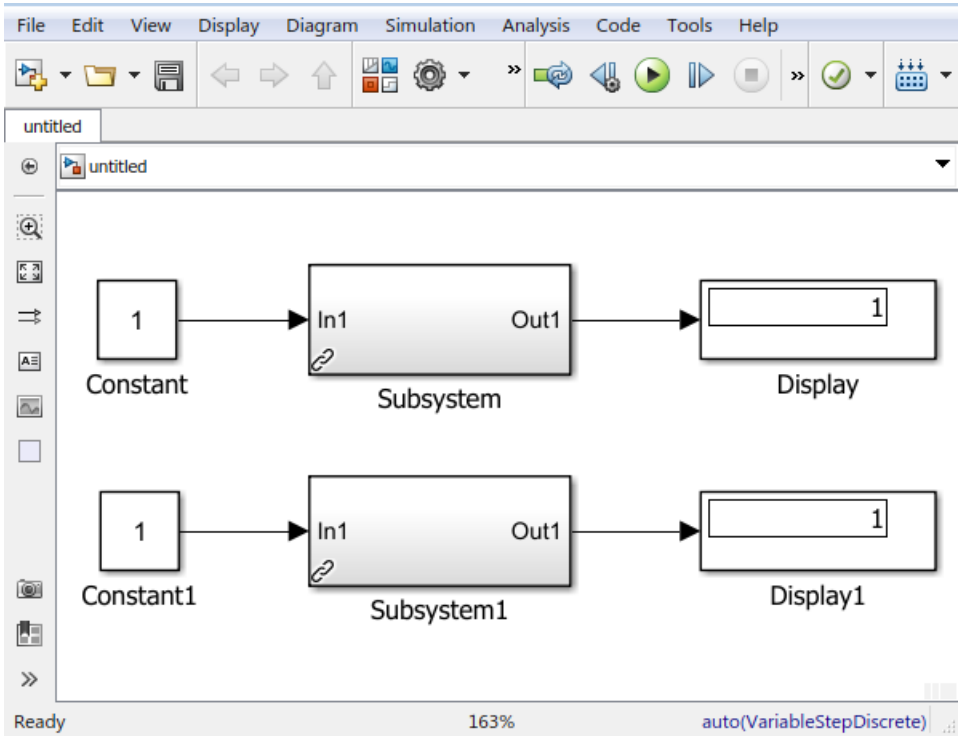
For example, you can use the `set_param` command to set a parameter value in the child blocks of a linked subsystem block. The `set_param` command overrides the parameter values of the child blocks of the subsystem linked block. Thus, differentiating the child block value from its parent library block and creating a parameterized link.

Consider a Subsystem library block (see “Subsystem Library Block” on page 41-13) that contains a Gain block within with its parameter value as `1`.



Subsystem Library Block

Use this Subsystem block as a linked block in a model.

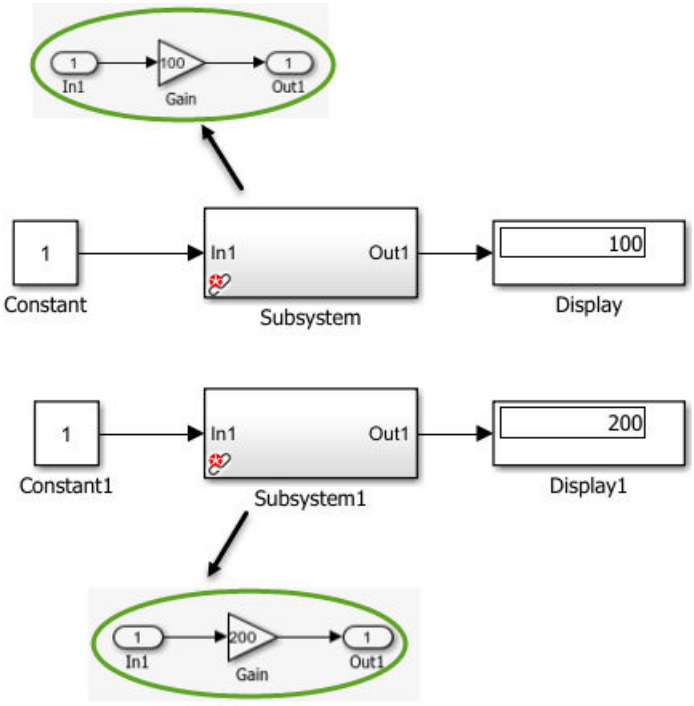


You can modify the parameter values of the child blocks of the linked block without changing the value of the parent library block. For example, you can change the parameter value of the Gain block within the `Subsystem` linked block.

To change the `Gain` parameter value of the Gain block within the `Subsystem` linked block to `100`, sequentially type these commands at MATLAB command prompt:

```
pathName = [ModelName,'/Gain_Subsystem1/Gain'];
set_param(pathName, 'Gain', '100')
```

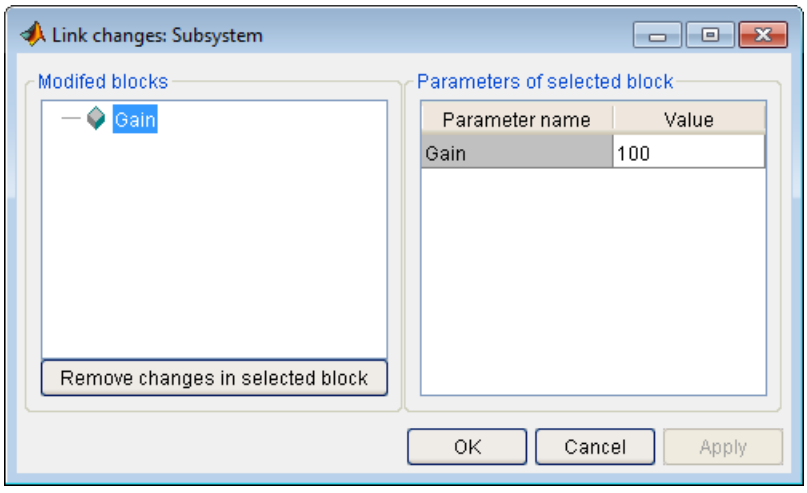
A parameterized link is now created, overriding the parameter value (see “Parameterized Linked Block” on page 41-15). Similarly, change the `Gain` parameter value of the `Subsystem1` linked block.



Parameterized Linked Block

When you save a model containing a parameterized link, Simulink saves the changes to a local copy of the Subsystem with the path to the parent library. When you reopen the model, Simulink copies the library block into the loaded model and applies the saved changes.

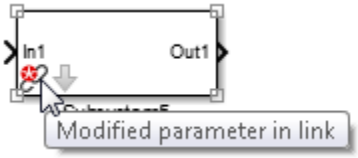
Note To view the parameterized changes on a block, right-click the block, and on the context menu, select **View Changes**. The **Link changes** dialog box opens displaying the list of modified blocks. You can also use this dialog box to remove parameterized changes from a block.



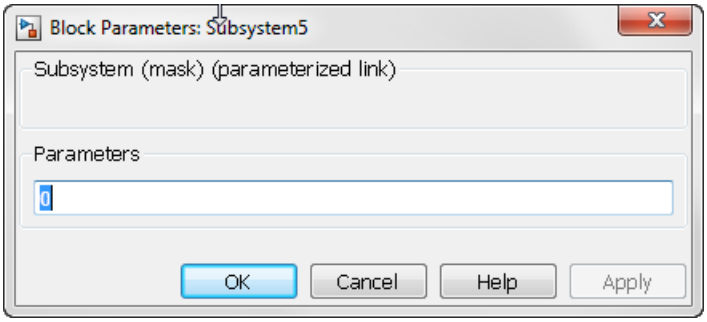
Identifying Parameterized Links

A parameterized link displays these identifications:

- The link badge of a parameterized link contains a black links with a red star icon, . For more information, see “Display Library Links” on page 41-18.
- The tooltip of a parameterized linked block displays **Modified parameter in link**.



- Block dialog box of a linked Subsystem block contains **parameterized link**.



Self-Modifiable Linked Subsystems

Tip We recommend using variant blocks over self-modifiable linked subsystems.

A self-modifiable linked subsystem is a linked block with the ability to have structural changes in the subsystem without disabling the link. A self-modifiable linked subsystem is created when you use a library block containing a self-modifiable mask as a linked block. You can use the mask initialization code to change the structural contents.

For more information, see “Dynamic Masked Subsystem” on page 39-42 and Self-Modifiable Mask on page 41-17.

See Also

More About

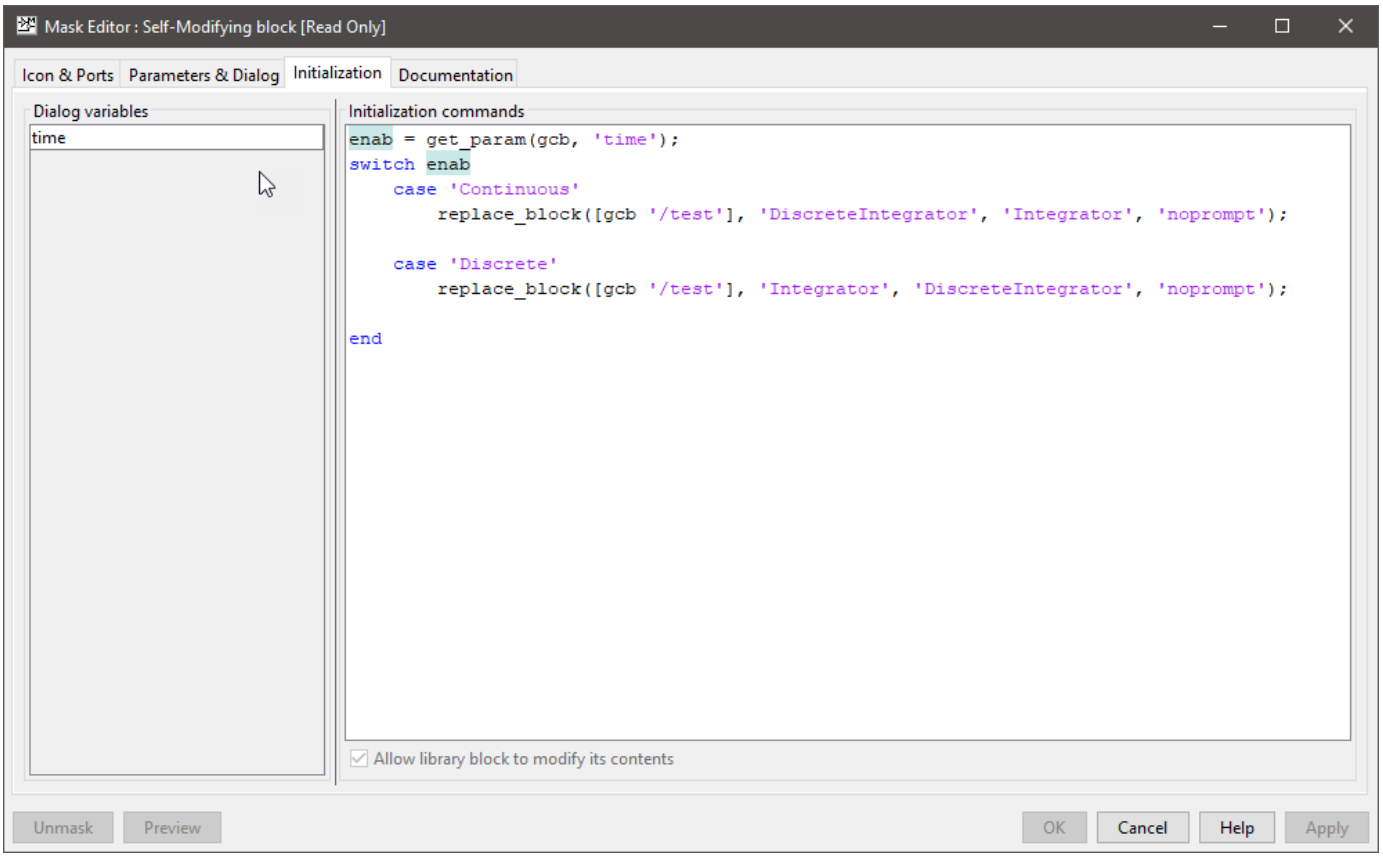
- “Linked Blocks” on page 41-10
- “Disable or Break Links to Library Blocks” on page 41-20
- “Restore Disabled Links” on page 41-24

Create a Self-Modifiable Library Block

A self-modifiable linked block is created when you add a self-modifiable masked subsystem block from the library to your model. Such linked blocks allow structural changes within the subsystem block.

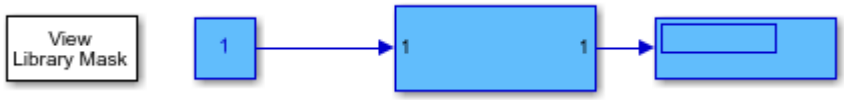
Observe that in this example if you change the 'Time domain' on the mask dialog box, the block within the Subsystem is replaced accordingly. For example, if you select the 'Time domain' as 'Discrete', the continuous integrator block is replaced with a discrete integrator block and vise-versa.

This dynamic change in the structure of the self-modifiable linked block is controlled using the MATLAB code that is added in the Initialization pane of the Mask Editor.



The structural changes take place only if the option 'Allow library block to modify its contents' available on the Initialization pane is selected.

```
open_system('self_modifiable_mask_example');
```



Display Library Links

Each linked block has a link badge associated with it. The badge makes it easier to identify the linked block in a model and also displays its link status.

To control the display of library link badges in a Simulink model, on the **Debug** tab, select **Information Overlays** and then one of these options:

- **Hide All Links** — Displays no links.
- **Disabled Links** — Displays only disabled links (the default for new models).
- **User-Defined Links** — Displays only links to user libraries.
- **Show All Links** — Displays all links.

If activated, link badges are displayed at the bottom left corner of a linked block. You can right-click the link badge to access link menu options.

The color and icon of the link badge indicates the status of the link.

Link Badge	Status
Black links 	Active link
Gray separated links 	Inactive link
Black links with a red star icon 	Active and modified (parameterized link)
White links, black background 	Locked link

See Also

More About

- “Linked Blocks” on page 41-10

- “Disable or Break Links to Library Blocks” on page 41-20
- “Restore Disabled Links” on page 41-24

Disable or Break Links to Library Blocks

Structural changes in a model include addition or deletion of blocks or adding ports while non-structural changes include changes in parameter value.

A linked block does not allow structural changes to it. You can disable the link of a linked block from its parent library block and perform required modifications. A disabled linked block behaves like a local instance of a block and allows you to make structural and nonstructural changes.

To disable a link, right-click the linked block and select **Library Link > Disable Link**. The **Disable Link** menu choice is made available only if the parent block of the linked block is disabled. If the menu option is grayed out, you must disable the link of the parent block first.

Alternatively, you can right-click the link icon in the graph on the bottom left corner of the canvas and click **Disable Link**. This action recursively disables the links of all the blocks up in the hierarchy.

To prevent unintentional disabling of a linked block, you can lock its links to the library. To lock a link, in the Library window, on the **Library** tab, click **Lock Links**. You can later choose to unlock the locked link by clicking **Links Locked**.

Note Simulink offers to disable the library links (unless the link is locked) when you try to make structural changes to a block that contains active library links.

Do not use `set_param` to make structural changes to an active link. The result of this type of change is undefined.

A disabled link of a linked block can be restored. For more information, see “Restore Disabled Links” on page 41-24.

Disabled links can cause merge conflicts and failure to update all instances of the same model component. In a hierarchy of links, you can accidentally disable all links without being aware of it, and only restore one link while leaving others disabled.

Break Links

You can permanently break links to the parent library. Before you break a library link, the link must first be disabled. When you break a link, the linked block is converted to a standalone block, and you cannot detect what the block linked to previously.

To break a link, use any of these options:

- For disabled links, right-click the linked block and select **Library Link > Break Link**.
- To copy and break links to multiple blocks simultaneously, select multiple blocks and then drag. The locked links are ignored and not broken.
- When you save the model, you can break links by supplying arguments to the `save_system` command. For more information, see `save_system`.

Note

- Some models can contain blocks from third-party libraries or optional Simulink block sets. Breaking the link for such models does not guarantee that you can run the model standalone. It is

possible that a library block invokes functions supplied with the library and hence can run only if the library is installed on the system running the model.

- Breaking a link can cause a model to fail when you install a new version of the library on a system. For example, if a model block invokes a function that is supplied from a library. If you break the link for such a block, the function can no longer be invoked from the model, causing simulation to fail. To avoid such problems, avoid breaking links to libraries.

See Also

“Linked Blocks” on page 41-10 | “Restore Disabled Links” on page 41-24

Lock Links to Blocks in a Library

You can lock links to a library. Lockable library links prevent unintentional disabling of these links. Lockable libraries ensure robust usage of mature stable libraries.

To lock links to a library, in the Library window, on the **Library** tab, click **Lock Links**. The link from the linked block to its parent library is now locked. When you refresh the model, you see a change in the link badge. The locked link badges have a black background.

Locked linked block links cannot be disabled from the parent library block. Such links can only be disabled from the command line by changing the `LinkStatus` to `inactive`. For more information, see “Control Linked Block Programmatically” on page 41-31.

The context menu of a locked linked block displays **Locked Library Link** and not **Library Link**. Also notice that the only enabled option on this menu is **Go To Library Block**.

If you open a locked linked block, the window title displays **Locked Link: *`blockname`***. The bottom left corner shows a lock icon and a link badge. You can also hover over the link badge to view lock information.

To unlock links from the Library window, on the **Library** tab, click **Links Locked**. If the library is locked, the option to lock or unlock the links is disabled. You must unlock the library to enable the option.

Rules for Locked Links

- Locked links cannot be edited. If you try to make a structural change to a locked link (such as editing the diagram), you see a message that you cannot modify the link because it is either locked or inside another locked link.
- The mask and block parameter dialogs are disabled for blocks inside locked links. For a resolved linked block with a mask, its parameter dialog is always disabled.
- You cannot parameterize locked links in the Model Editor.

- When you copy a block, the library status determines the status of a link (locked or not). If you copy a block from a library with locked links, the link is locked. If you later unlock the library links, any existing linked blocks do not change to unlocked links until you refresh the links.
- If you use sublibraries, when you lock links to a library, lock links to the sublibraries.

See Also

More About

- “Linked Blocks” on page 41-10

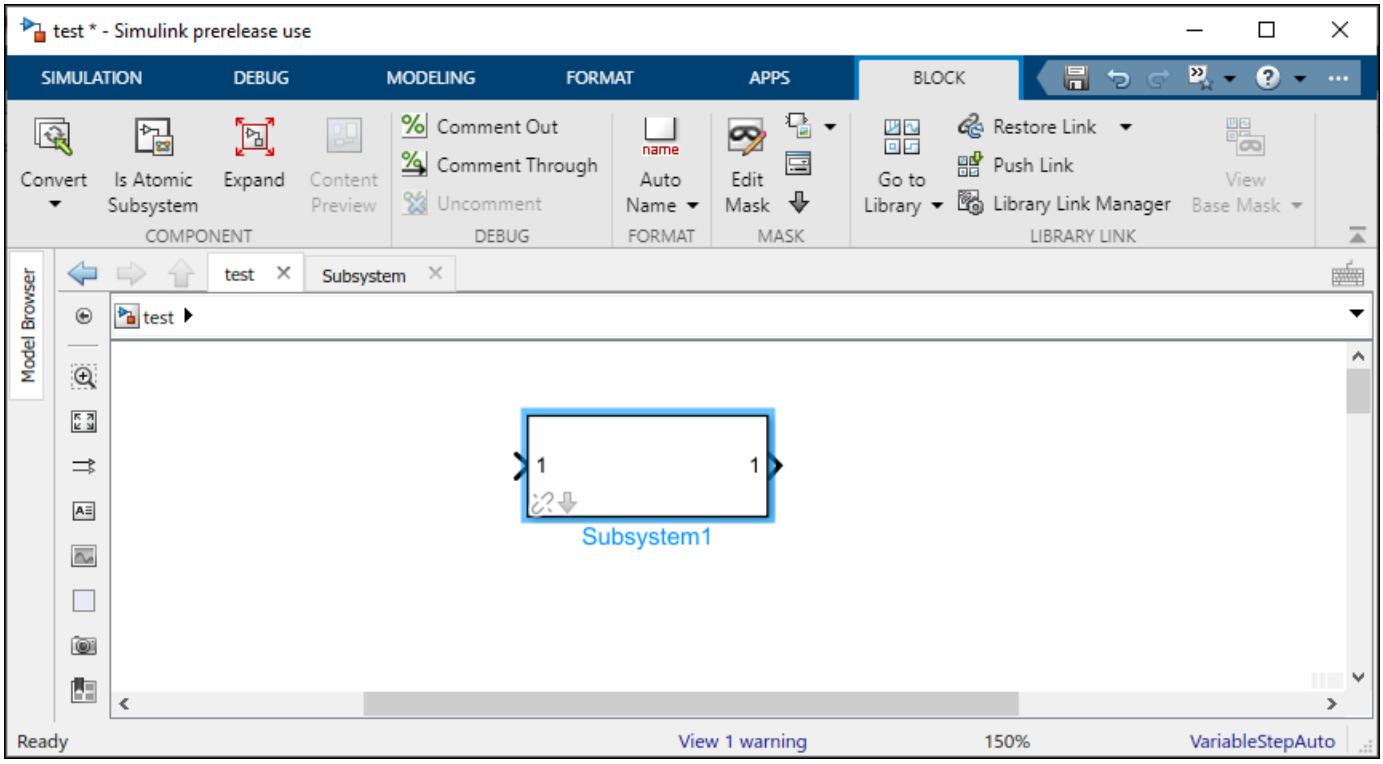
Restore Disabled Links

You can restore a disabled linked block or push a linked block value to the parent library block either individually or hierarchically.

Restore Disabled Links Individually

When you push or restore a disabled block in individual mode, the disabled or edited block is pushed to or restored from the library, preserving the changes inside the block without acting on the hierarchy. All other links are unaffected. You can restore or push individual disabled blocks in three ways:

- Select the disabled linked block and in the Simulink toolstrip, on the **Subsystem Block** tab, select **Push Link** or **Restore Link**.



- Right-click a disabled linked block and from the context menu, select **Library Links** and click **Push Link** to push the changes or **Restore Link** to restore the values from the parent library block.
- Right-click the link badge on the bottom-left corner of a disabled linked block and from the context menu, click **Push Links** to push the changes or **Restore Link** to restore the values from the parent library block.

Restore Disabled Links Hierarchically

When you push or restore disabled links in the hierarchy mode, the operation is applied to the whole hierarchy of links. To push or restore in hierarchy mode, use the Library Link Manager tool. The Library Link Manager displays all the disabled and parameterized linked blocks in the model.

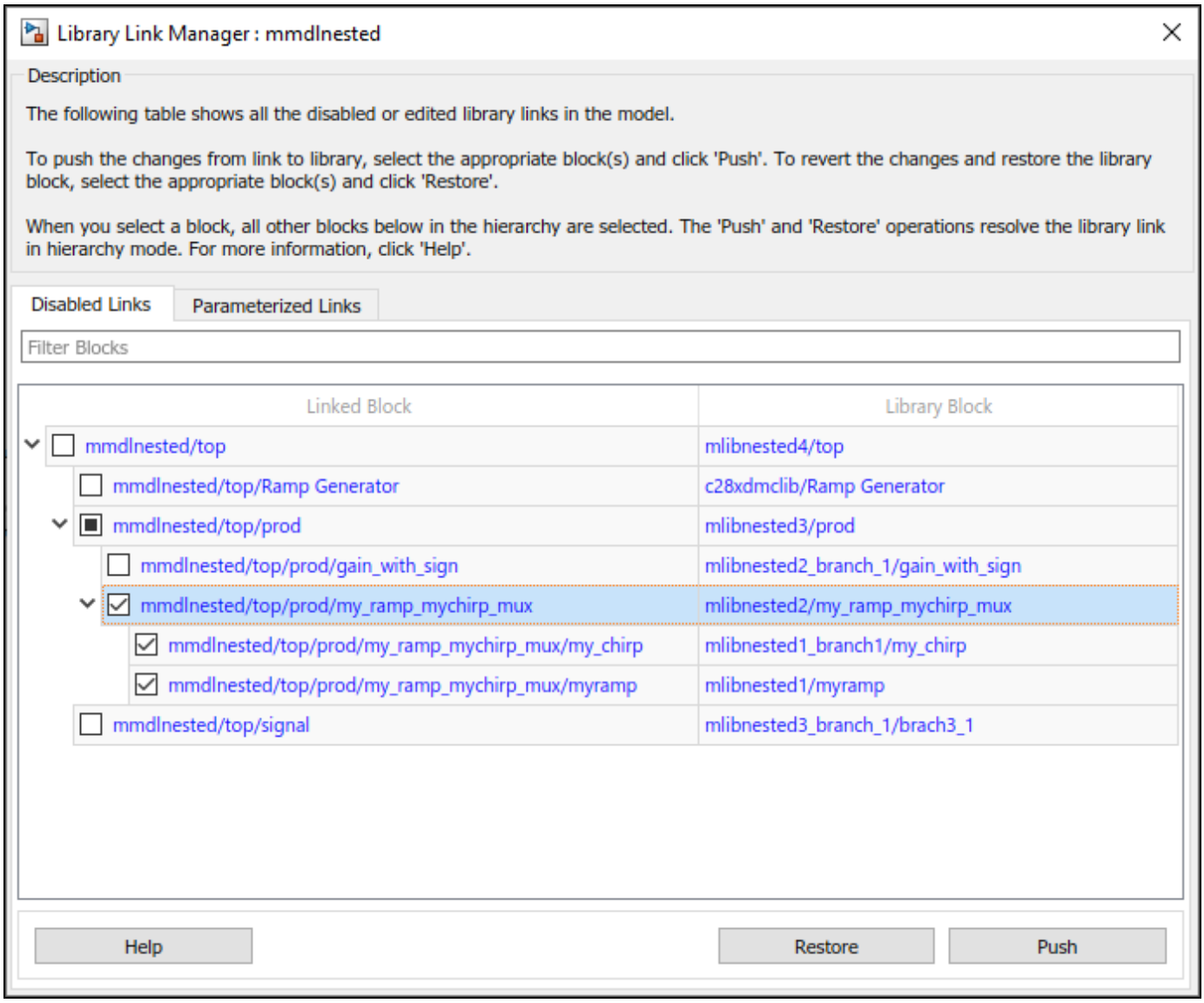
You can launch the Library Link Manager in two ways:

- In the Simulink toolstrip, on the **Modeling** Tab, under the **System Design** section, click **Library Link Manager**.

  or

- Select any disabled linked block and in the Simulink toolstrip, on the **Subsystem Block** tab, click **Library Link Manager**.

  The Library Link Manager window appears. It has two tabs showing disabled Links and parameterized links in the model. By default, the **Disabled Links** tab is selected.



The table shows two columns where the **Linked Block** column displays the list of linked blocks that have disabled links and the **Library Block** column displays the parent library block of the corresponding linked blocks.

To push or restore disabled blocks,

1. Select the disabled block or blocks as per your requirement.
2. Click **Push** to replace the version of the block in the library with the version in the model or **Restore** to replace the version of the linked block in the model with the version in the library.

You can use the **Filter Blocks** field to filter the displayed linked blocks based on your preference. This is useful when you have a huge number of blocks in the model.

Pushing or Restoring Link Hierarchies

Pushing a hierarchy of disabled links affects the disabled links inside and outside the hierarchy for a given link. If you push changes from a disabled link in the middle of a hierarchy, the inside links are pushed and the outside links are restored if unchanged. This operation does not affect outer (parent) links with changes unless you also explicitly selected them for push. The Library Link Manager starts to push from the lowest inside links and then moves up in the hierarchy.

For examples:

1. Link A contains link B and both have changes.
   - Push A. The Links Tool pushes both A and B.
   - Push B. The Links Tool pushes B and not A.
2. Link A contains link B. A has no changes, and B has changes.
   - Push B. The Links Tool pushes B and restores A. When parent links are unmodified, they are restored.

If you have a hierarchy of parameterized links, the Library Links Manager can manipulate only the top level.

Tip To compare files and view structural changes, in the **Modeling** tab, select **Compare > Compare Models**.

See Also

More About

- “Linked Blocks” on page 41-10
- “Restore Parameterized Links” on page 41-27

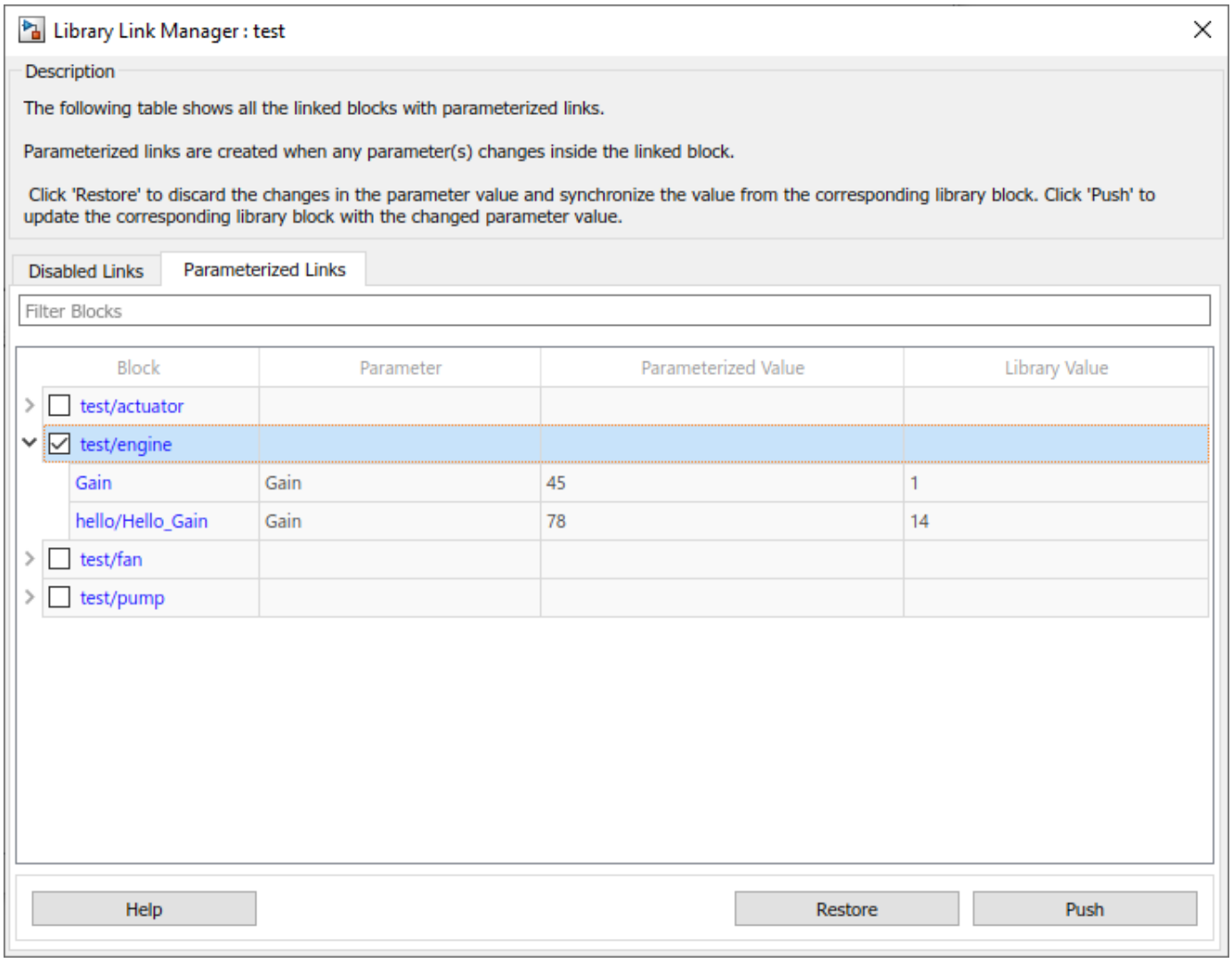
Restore Parameterized Links

A parameterized link is created when you change the parameter values of the child blocks of a masked subsystem linked block. You can choose to push the new values from the linked block to the parent block or restore the linked block with values from the parent block. You can push or restore parameterized links using the Library Links Manager.

To restore or push parameterized links:

1. Select the parameterized link and in the Simulink toolstrip, on the **Subsystem Block** tab, click **Library Link Manager**.

   The Library Link manager opens. Click the **Parameterized Links** tab to display all the parameterized link in the model.



The table has the following columns:

- **Block** - Displays the block name.

- **Parameter** - The name of the parameter in the block.
- **Parameterized Value** - The changed value of any edited parameter in the model.
- **Library Value** - The original value of the parameter in the parent library block.

2 Select the blocks you want to push or restore.

3 Click **Push** to replace the value of the block in the library with the parameterized value in the model or **Restore** to replace the version of the linked block in the model with the version in the library.

See Also

More About

- “Linked Blocks” on page 41-10
- “Restore Disabled Links” on page 41-24

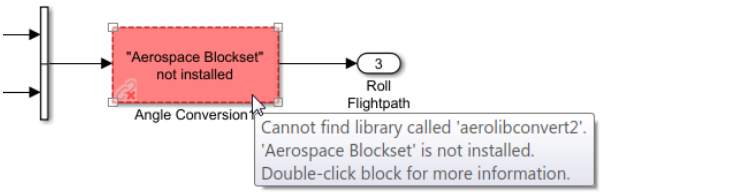
Fix Unresolved Library Links

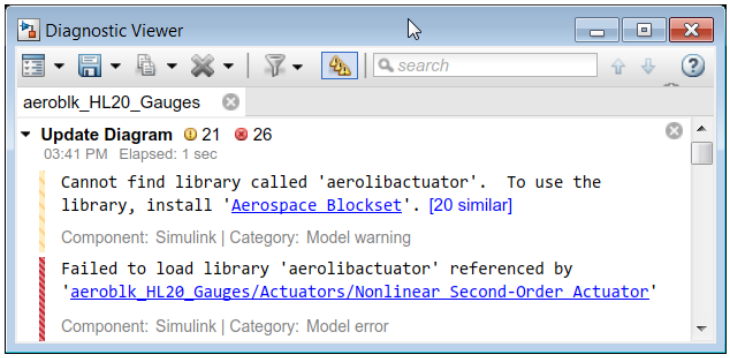
If Simulink is unable to find either the library block or the source library on your MATLAB path during linked block update, the link becomes unresolved. Simulink changes the appearance of these blocks.

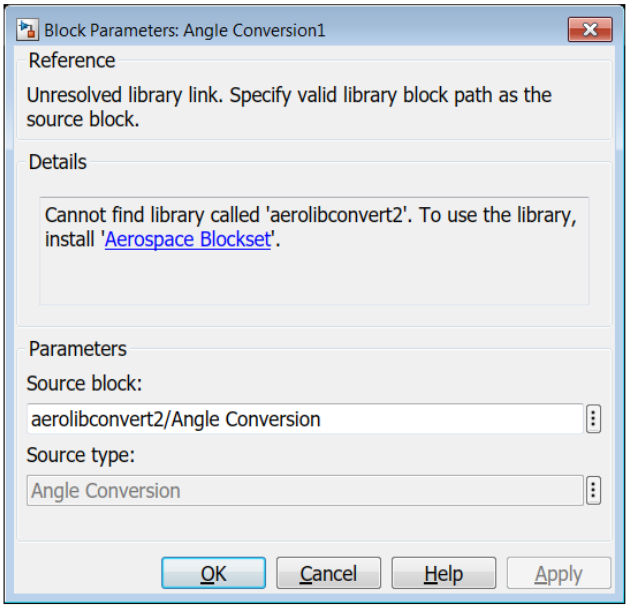
Simulink tries to help you find and install missing products that a model needs to run. If you open a model that contains built-in blocks or library links from missing products, you see labels and links to help you fix the problem.

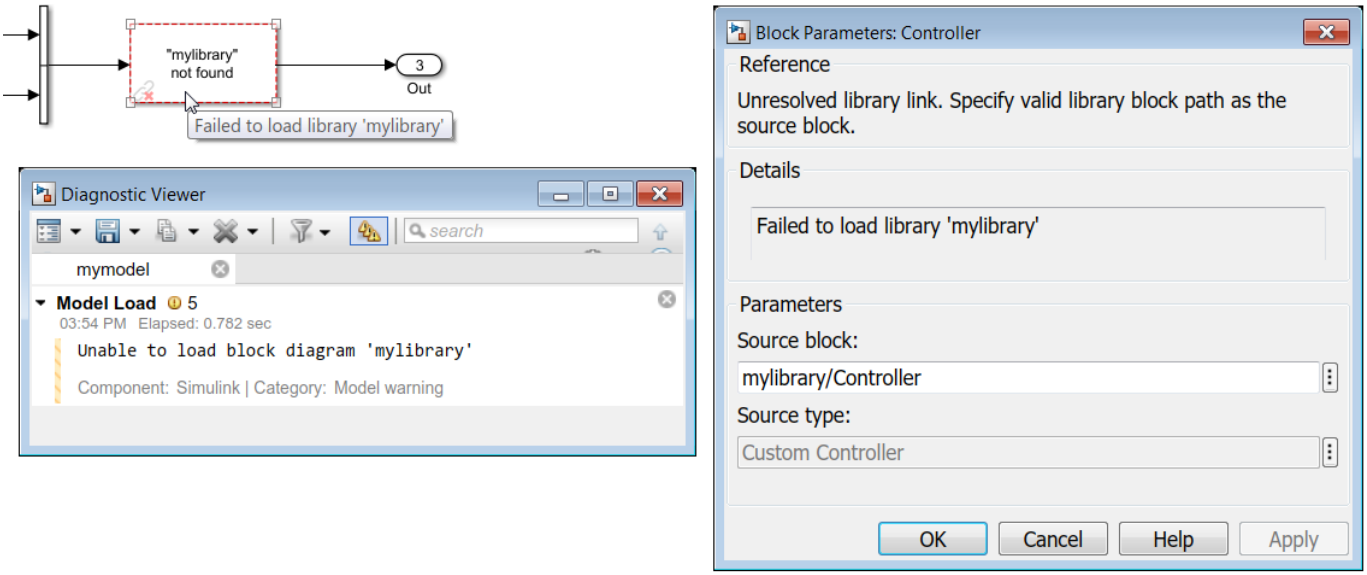
- Blocks are labeled with missing products (for example, **SimEvents not installed**)
- Tooltips include the name of the missing product
- Messages provide links to open Add-On Explorer and install the missing products

For unresolved library links, double-click the block to view details. Click the link to open Add-On Explorer and install the product.









To fix an unresolved link, you can:

- Double-click the unresolved block to open its dialog box (see the Unresolved Link block reference page). If a product is missing, click the link to open Add-On Explorer and install the product. Alternatively, correct the path name in the **Source block** field and click **OK**.
- Delete the unresolved block and copy the library block back into your model.
- Add the folder that contains the required library to the MATLAB path and then, on the **Modeling** tab, click **Update Model**.

See Also

Unresolved Link

More About

- “Linked Blocks” on page 41-10
- “Display Library Links” on page 41-18

Control Linked Block Programmatically

Linked Block Information

Use the `libinfo` command to get information about the linked blocks in the model. `libinfo` also provides information about the parent library blocks of a linked block.

For example, here is a model with linked blocks:

Masking variant blocks

When you execute the `libinfo(gcb)` command on this block:

```
 Block: 'slexMaskVariantExample/VariantSubsystem2' %Linked block
           Library: 'slexMaskingVariants_libraryblock' %Parent library block
    ReferenceBlock: 'slexMaskingVariants_libraryblock/VariantSubsystem2'
        LinkStatus: 'resolved' %Link status
```

The `ReferenceBlock` property gives the path of the library block to which a block links. You can change this path programmatically by using the `set_param` command. For example:

```
set_param('slexMaskVariantExample/VariantSubsystem2', 'ReferenceBlock', 'slexMaskVariantExample2/
```

Here, `slexMaskVariantExample/VariantSubsystem2` is the original library block path and `slexMaskVariantExample2/VariantSubsystem` is the new library block path.

Note It is not recommended to change the properties of a referenced block by using the `set_param` command in the mask initialization code or callback code of the same block. For such modeling patterns, you can use Variant blocks (Masking variant blocks) or use the `ReferenceBlock` parameter on the callback code or the mask initialization code of the parent block of the reference block.

Lock Linked Blocks

Use the `LockLinksToLibrary` command to lock or unlock a linked block in a library from the command line. When you set the value of `LockLinksToLibrary` to `on`, the linked block links to the library are locked.

```
set_param('MyLibraryName', 'LockLinksToLibrary', 'on') %Lock links
```

```
set_param('MyLibraryName', 'LockLinksToLibrary', 'off') %Unlock links
```

Link Status

All blocks have a `LinkStatus` parameter and a `StaticLinkStatus` parameter to indicate whether the block is a linked block.

Use `get_param(gcb, 'StaticLinkStatus')` to query the link status without updating the linked blocks. You can use `StaticLinkStatus` to query the status of a linked block that is either active or outdated.

Use `get_param` to send a query to get the value of the `LinkStatus`.

Get LinkStatus Value	Description
`none`	Block is not a linked block.
`resolved`	Resolved link.
`unresolved`	Unresolved link.
`implicit`	Block resides in library block and is itself not a link to a library block. Suppose that A is a link to a subsystem in a library that contains the Gain block. If you open A and select the Gain block, `get_param(gcb, 'LinkStatus')` returns `implicit`.
`inactive`	Disabled link.

Use `set_param` to set the `LinkStatus`.

Set LinkStatus Value	Description
`none`	Breaks link. Use `none` to break a link, for example, `set_param(gcb, 'LinkStatus', 'none')`.
`breakWithoutHierarchy`	Breaks links in place without breaking the nested parent hierarchy of link. For example, `set_param(gcb, 'LinkStatus', 'breakWithoutHierarchy')`.
`inactive`	Disables link. Use `inactive` to disable a link, for example, `set_param(gcb, 'LinkStatus', 'inactive')`.
`restore`	Restores an inactive or disabled link to a library block and discards any changes made to the local copy of the library block. For example, `set_param(gcb, 'LinkStatus', 'restore')` replaces the selected block with a link to a library block of the same type. It discards any changes in the local copy of the library block. `restore` is equivalent to `Restore Individual` in the Links Tool.
`propagate`	Pushes any changes made to the disabled link to the library block and re-establishes its link. `propagate` is equivalent to `Push Individual` in the Links Tool.
`restoreHierarchy`	Restores all disabled links in the hierarchy with their corresponding library blocks. `restoreHierarchy` is equivalent to `Restore` in hierarchy mode in the Links Tool.
`propagateHierarchy`	Pushes all links with changes in the hierarchy to their libraries. `propagateHierarchy` is equivalent to `Push` in the **Hierarchy** mode of the Links Tool. See “Restore Disabled Links” on page 41-24.

Note

- When you use `get_param` to query the link status of a block, the outdated block links also resolve.
- Using the `StaticLinkStatus` command to query the link status when `get_param` is being used in the callback code of a child block is recommended. `StaticLinkStatus` command does not resolve any outdated links.

If you call `get_param` on a block inside a library link, Simulink resolves the link wherever necessary. Executing `get_param` can involve loading part of the library and executing callbacks.

See Also

More About

- “Linked Blocks” on page 41-10
- “Display Library Links” on page 41-18

Forwarding Tables

If you edit an existing library block, you would want to ensure that the changes do not break the model when the model is saved with an older version of the library block. The type of edits in the library block can include, library path change, library block name change, or addition, removal, or renaming of parameters.

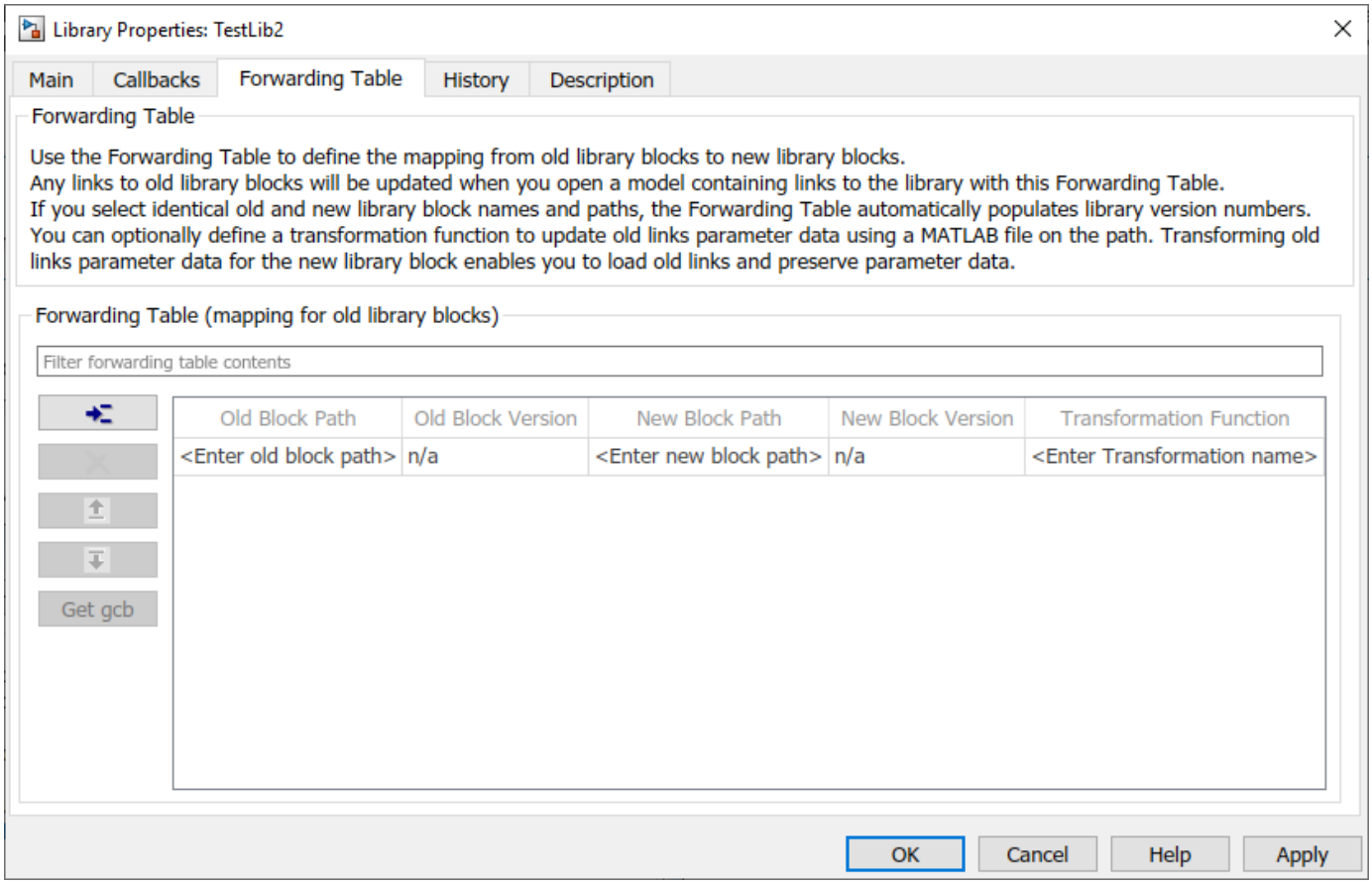
The Forwarding Table helps you to maintain compatibility of library blocks and ensures that the models continue to work. You can use the Forwarding Table to create a map between the old and the new library blocks without any loss of data or functionality. After specifying the mapping of old library blocks to new library blocks in the forwarding table, links to the old library blocks update automatically during model load. For example, if you rename or move a block in a library, you can use a forwarding table to update the models that have links to the old library block. Forwarding Table executes in the following scenarios:

- When a model is closed and re-opened.
- When an `add_block` or `replace_block` command is executed.

Create Forwarding Table

Note Models that have broken or disabled links cannot be updated using the Forwarding Table.

1. Open a locked library model.
2. In the Library window, on the **Library** tab, click **Locked Library**. The library is now unlocked for editing.
3. On the **Modeling** tab, click **Library Properties**. The **Library Properties** dialog box opens.
4. Click the **Forwarding Table** tab.
5. Click  (Add New Entry) button. A new row is added in the Forwarding Table.



6 Specify values in the **Old Block Path** and **New Block Path** columns. To get the path of a block, select the block in the model and click Get gcb.

7 In the **Version** column, you can choose to specify a version number for the library block.

If the old block name and the new block name are the same, the forwarding table populates the version number automatically. The initial value of the library version (`LibraryVersion`) is derived from the model version (`ModelVersion`) of the library at the time the library link is created. Any subsequent updated to the library block would update the library version to match the model version of the library.

Note

- Version number must be a numeric value.
- When the old and the new block paths are the same, the version number must be of the format `<major_version>.<minor_version>`. For example, while renaming a library block.
- The version number cannot have more than one dot notation. For example, a version number of `1.3` is acceptable whereas, version number `1.3.1` is not acceptable.
- When you use a Forwarding Table to move a library block from one library to another, the version number format is not critical.

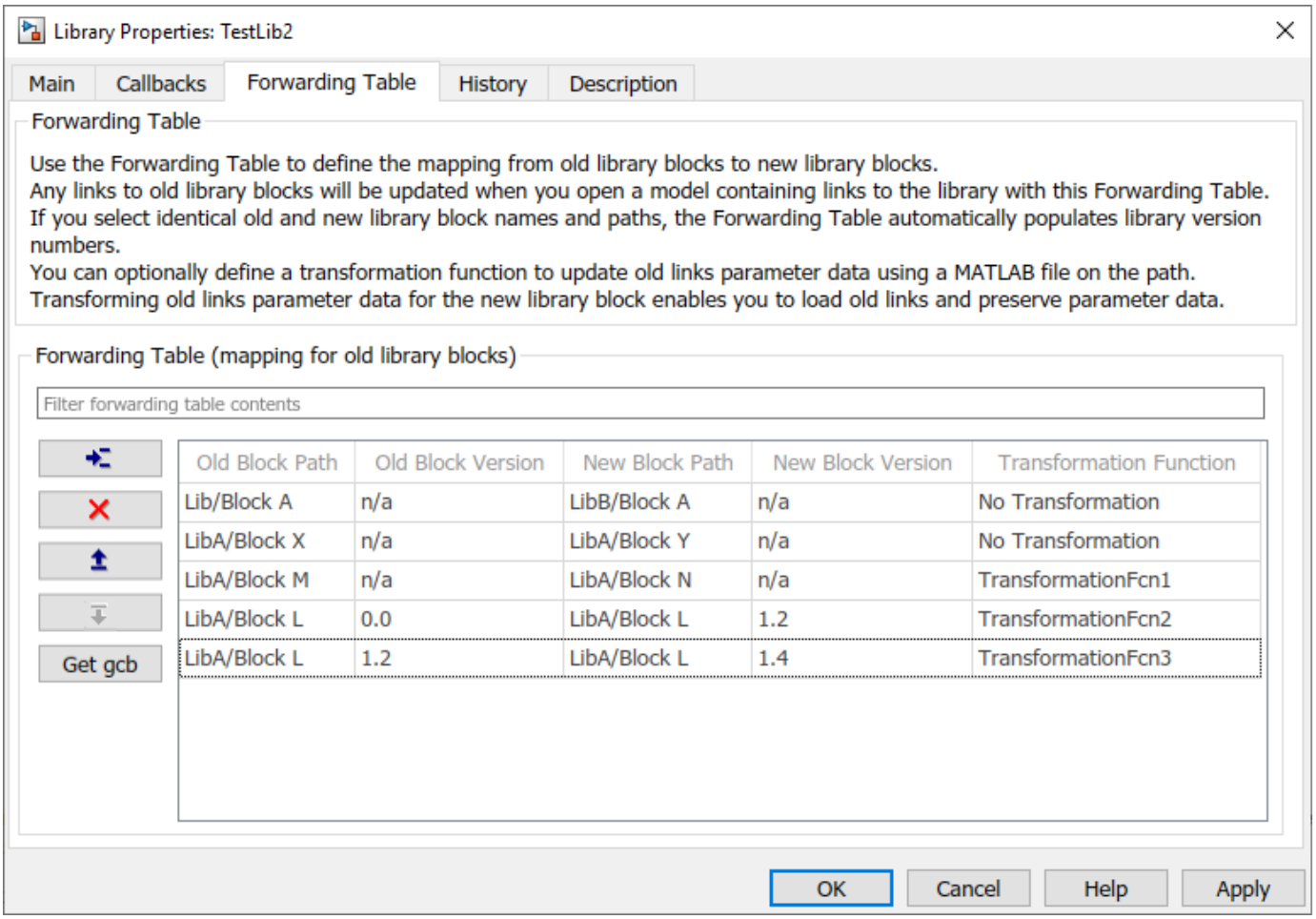
8 In the **Transformation Function** column, you can specify a MATLAB file that corrects the mismatch of parameter data between the old and the new link. Transforming old link parameter data for the new library block enables you to load old links and preserve parameter data. For more information, see “Transformation Functions” on page 41-37.

   If no transformation function is specified, the Transformation Function column displays **No Transformation** when you save the library.

9 To apply the changes and close the dialog box, click **OK**. The mapping of old path to new path is created in the Forwarding Table. The links to the old library blocks are updated automatically when you open a model containing links to the library.

Once the Forwarding Table is populated, you can use the search bar above the table to filter its contents. This filter is especially useful when the Forwarding Table has too many entries. You can sort the columns in the table in ascending or descending order. You can also group each of the columns by their values.

An example of user-defined Forwarding Table is as shown:



When you specify identical library block name and path for the older and the newer blocks, the Forwarding Table populates the version number automatically. For the first entry with identical names and path, the version number of the old block starts with 0, and the new version of the block is

set as the model version of the library. You can view the model version of the library under the **History** tab of the Forwarding Table.

A transformation function must be specified when the instance-specific parameters (`InstanceData`) have changed in the old and the new library block.

In this example,

- Block path for `Block A` changes from `LibA` to `LibB`.
- Block name for `Block X` changes to `Block Y` while the library path remains the same.
- Block name for `Block M` changed to `Block N`. A transformation function is specified to take care of the instance-specific changes.
- Block version and instance-specific parameter changed for `Block L`.

Create Forwarding Table Programmatically

At the command line, you can create a simple Forwarding Table specifying the old locations and new locations of blocks that have moved within the library or to another library. You associate a forwarding table with a library by setting its `ForwardingTable` parameter to a cell array of two-element cell arrays, each of which specifies the old and new path of a block that has moved. For example, the syntax to create a forwarding table and assign it to a library named `Lib1` is:

```
set_param('Lib1', 'ForwardingTable', {{'Lib1/A', 'Lib2/A'}
{'Lib1/B', 'Lib1/C'}});
```

where:

- `Lib1` is the library associated to the forwarding table.
- Block `A` is transferred from `Lib1` to `Lib2`.
- Block `B` is renamed to `C` in the same library.

Transformation Functions

A linked block instance is associated with instance-specific parameters called `InstanceData`. When you create versions of a library block, parameter sets can get added or removed from the `InstanceData`.

A transformation function corrects the mismatch of parameters in the `InstanceData` of the new and old library links thus ensuring that the library links continue to work.

You can define a transformation function using a MATLAB file on the path, then specify the function in the **Transformation Function** column of the **Forwarding Table**.

The new block path defined in the forwarding table overrides the values defined in the transformation function. If the new block path is a dynamic value that changes based on certain conditions, then the new block path must be only defined using the transformation function.

The syntax for transformation function must be:

```
function outData = TransformationFcn(inData)
```

where:

- `inData` is a structure with fields `ForwardingTableEntry` and `InstanceData`, and `ForwardingTableEntry` is also a structure.

- `outData` is a structure with fields `NewInstanceData` and `NewBlockPath`.

A general transformation function can have many local functions defined in it. The function calls the appropriate local functions based on old block names and versions. You can use this to combine multiple local functions into a single transformation function, to avoid having many transformation functions on the MATLAB path.

Consider the Compare to Constant block in Simulink Library. You must create versions of this block without changing the name and the block path but add the parameters to the newer library block.

The table displays the parameter difference in two versions of the Compare to Constant block.

Old Version	New Version

Block {
BlockType Reference
Name "Compare\nTo Constant"
Ports [1, 1]
Position [210, 60, 250, 100]
SourceBlock "fixpt_lib_4/Logic & Compa\nTo Constant"
SourceType "Fixed-Point Compare To Constant"
relop "=="
const "3.0"
}
	Block {
	BlockType Reference
	Name "Compare\nTo Constant"
	SID "15"
	Ports [1, 1]
	Position [125, 50, 155, 80]
	ZOrder -5
	LibraryVersion "1.271"
	SourceBlock "simulink/Logic and Bit\nOperat
	SourceType "Compare To Constant"
	relop "<="
	const "3.0"
	OutDataTypeStr "boolean"
	ZeroCross on
	}

The new version of the Compare to Constant block has additional parameters (`OutDataTypeStr` and `ZeroCross`) associated with it. For such cases, the transformation function must ensure that the additional parameters in the `InstanceData` are set so that the old library links work.

This example shows a transformation function for the Compare to Constant block to add the `OutDataTypestr` parameter with a value of `uint8`.

```
function [outData] = TransformationCompConstBlk(inData)
outData.NewBlockPath = ''; % No change in the library block path
outData.NewInstanceData = [];
instanceData = inData.InstanceData;
% Get the field type 'Name' from instanceData
[ParameterNames{1:length(instanceData)}] = instanceData.Name;

if (~ismember('OutDataTypeStr',ParameterNames))
    % OutDataTypeStr parameter is not present in old link. Add it and set value uint8
    instanceData(end+1).Name = 'OutDataTypeStr';
    instanceData(end).Value = 'uint8';
end

outData.NewInstanceData = instanceData;
```

Create Mask Parameter Aliases

If you rename a mask parameter, you must ensure that the existing MATLAB scripts that use the old parameter names, continues to work. To ensure the compatibility, you can create alias (alternate names) for a mask parameter name. Alias allows you to change the name of a mask parameter in a library block without having to recreate links to the block in existing models.

Consider a masked block that contains an **Edit** parameter. The mask parameter name for this **Edit** parameter is `p1`.

```
MaskObj = Simulink.Mask.get(gcb));
hEdit = MaskObj.getParameter('p1');

hEdit=
% MaskParameter with properties:

           Type: 'edit'
    TypeOptions: {0×1 cell}
           Name: 'p1'
         Prompt: 'p1'
          Value: '0'
       Evaluate: 'on'
        Tunable: 'on'
      NeverSave: 'off'
         Hidden: 'off'
        Enabled: 'on'
        Visible: 'on'
        ToolTip: 'on'
       Callback: ''
          Alias: ''
```

Notice that the **Edit** mask parameter does not have any alias name. To add an alias name for the mask parameter, you can set a value for the `Alias` mask parameter property.

```
MaskObj.Alias = 'pa'
```

You can either use the mask parameter name or the alias to do a function call on the mask parameter. For example, in this case you can either use `set_param(gcb, 'p1, '10)` (mask parameter name) `set_param(gcb, 'pa, '10)` (mask parameter alias) to set a value for the **Edit** mask parameter.

See Also

More About

- “Linked Blocks” on page 41-10

Integrate C Code in Simulink Models

Integrate C Code Using C Caller Blocks

In this section...

You can integrate new or existing C code into Simulink using the C Caller block. To create custom blocks in your Simulink models, the C Caller block allows you to call external C functions specified in external source code and libraries. The advantages of the C Caller block are:

- Automated integration of simple C functions
- Integration with Simulink Coverage, Simulink Test, and Simulink Design Verifier
- Integration with Simulink Coder

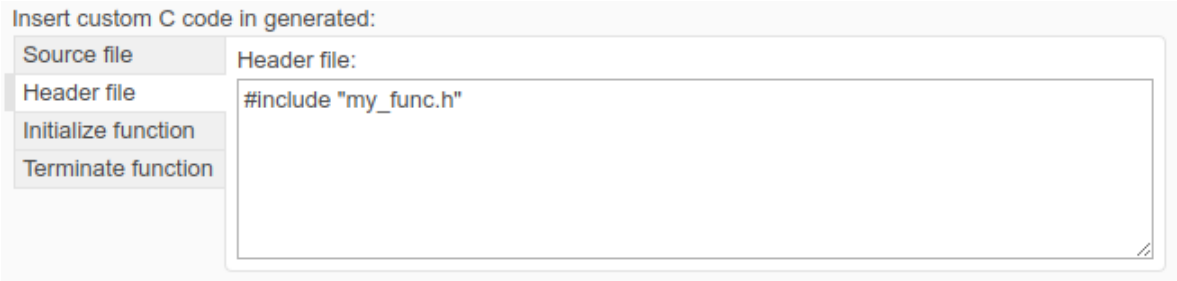
The C Caller block allows you to bring C algorithms into Simulink. To model dynamic systems, use the S-Function Builder instead. Next steps describe the workflow to integrate C code into Simulink using the C Caller block.

Specify Source Code and Dependencies

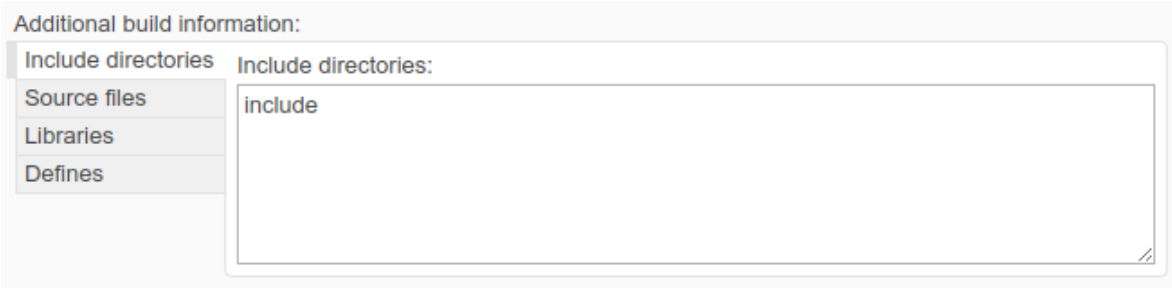
Specify your external source code file that contains your C functions.

1. From Simulink toolstrip, open the **Configuration Parameters**.
2. In the left pane, select **Simulation Target**.
3. To enable code parsing by the C Caller block, ensure that the **Import custom code** box is selected.

   The directories and file paths can be absolute and relative file paths to model directories or to the current working directory. See “Specify Relative Paths to Your Custom Code” (Stateflow).
4. Select **Header file** and enter the name of your header file with the `#include` tag.



5. Under **Additional build information**, select **Include directories**, and enter the folders where additional build information, such as header files, are stored.

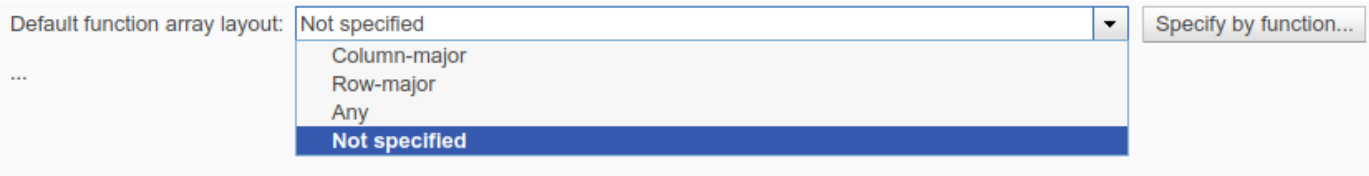


6 Select **Source files** and enter the path and the name of the source file. If the model and the source files are in different directories, enter the directory that contains the source file before the file name.

Note If a function is declared in the header file but not implemented in the source code, an empty stub function is automatically generated to simulate and compile the model.

Define Default Function Array Layout

You can specify the order of how your matrix data is stored in Simulink. Matrix data passed to and from your C functions is converted to the default function array layout you specify. If the function array layout is not specified, the matrix data is passed through the C Caller in the same order of your Simulink data, and computational errors may occur due to row-column major disarrangement. Ensure that you follow the same default function array layout for all Simulink data.



- **Column-Major** — The C Caller block handles Simulink data in column-major order. Suppose that you have a 3-by-3 matrix. In the C Caller block, this matrix is stored in this sequence: first column, second column, and third column.
- **Row-Major** — The C Caller block handles Simulink data in row-major order. Suppose that you have a 3-by-3 matrix. In the C Caller block, this matrix is stored in this sequence: first row, second row, and third row.
- **Any** — Array data can be stored both in column-major and row-major order in the C Caller block. As a result, you can generate code both in column-major and row-major settings.

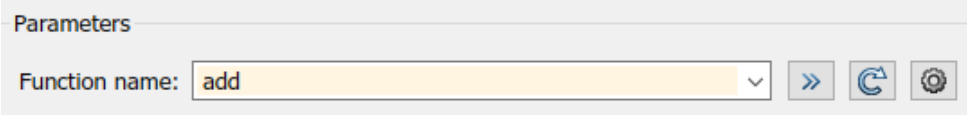
- **Not specified** — Array data can be stored in both column-major and row-major order. Compared to **Any** setting, you can only generate code in column-major setting.

To learn more about the row-major and column-major array layouts in Simulink, see “Default function array layout”.

1. Select an array layout option under **Default Array Function Layout**.
2. If you need to apply a specific array layout to some of the functions in your code, click **Specify by Function** to select these functions.
3. Click **Apply** to accept your changes.
4. Click **OK** to close the **Configuration Parameters**.

Call C Caller Block and Specify Ports

You can start your custom C code integration into Simulink by typing C Caller in the Simulink canvas. Alternatively, drag a C Caller block from the **Library Browser > User-Defined Functions**. Double-click the block to open the **Block Parameters** dialog box to see the names of your functions and port specifications.



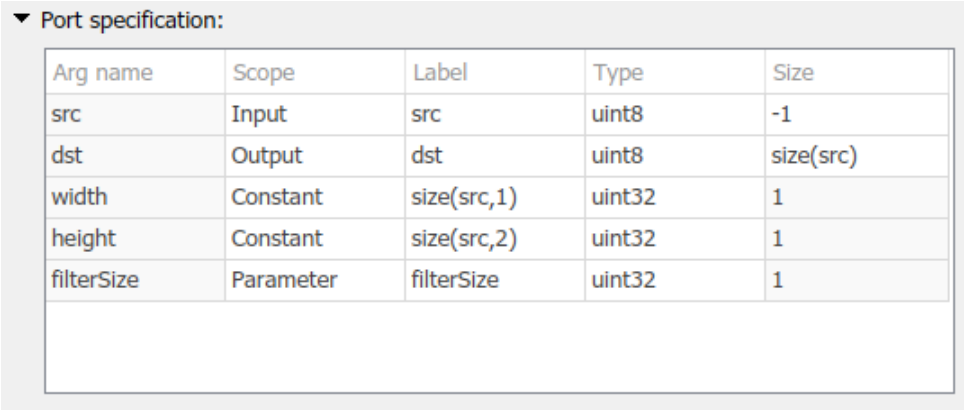
1. Click on the **Refresh custom code** to import your source code and its dependencies.
2. Your C functions are displayed under **Function Name**. If you can't see your full list of functions, click on the to reimport your source code.
3. To view function declarations or input/output variables to your functions in the header file, click the **Go to function declaration** to navigate the source files.
4. To change source files and their dependencies, or to define and select function array layouts, click the **Custom code settings** to open the **Simulation Target** tab in Configuration Parameters.

Map C Function Arguments to Simulink Ports

You can map C function arguments from your source code to Simulink ports using the **Port specification** table in the C Caller block and by creating a `FunctionPortSpecification` object through the command line . In your source code, the header file includes the C function arguments to be connected to Simulink ports.

```
extern void mean_filter(const unsigned char* src,
                        unsigned char* dst,
                        unsigned int width, unsigned int height,
                        unsigned int filterSize);
```

Port specification shows the details of your arguments and how they connect to your C Caller block in Simulink.



Name — Specifies the name of input and output arguments. **Name** is the function argument or parameter name as defined in your C functions from source code. This column is for reference purposes only.

Scope — Specifies how C function arguments map to the Simulink scope. Your arguments have default scopes depending on the function definition and you can change the scopes depending your function definition in the source code.

Simulink Scope	Scope to Block Mapping
`Input`	Block input port
`Output`	Block output port
`InputOutput`	Block input and output port
`Global`	Global variable used by the block
`Parameter`	Block tunable parameter
`Constant`	Constant value

When you have a constant qualifier definition such as `const double *u`, the argument can only be an input or a parameter. When there is no constant qualifier, the argument is an output by default, and you can change it to an `Input`, `InputOutput` or to a `Parameter` scope. In this case, ensure that the C function does not modify the memory pointed by the pointer. If the argument is of an `Output` type, every element pointed by this pointer should be reassigned in every call for this function.

C Argument	Simulink Scope
Function return	`Output`
`double u`	`Input`, `Parameter`, `Constant`

`double u[]`
 	
`double u[][2]`
 	
`double u[2][3]`
	`Output` (default), `Input`, `Parameter`

`double *u`	`Output` (default), `InputOutput`, `Input`, `Parameter`

C Argument	Simulink Scope

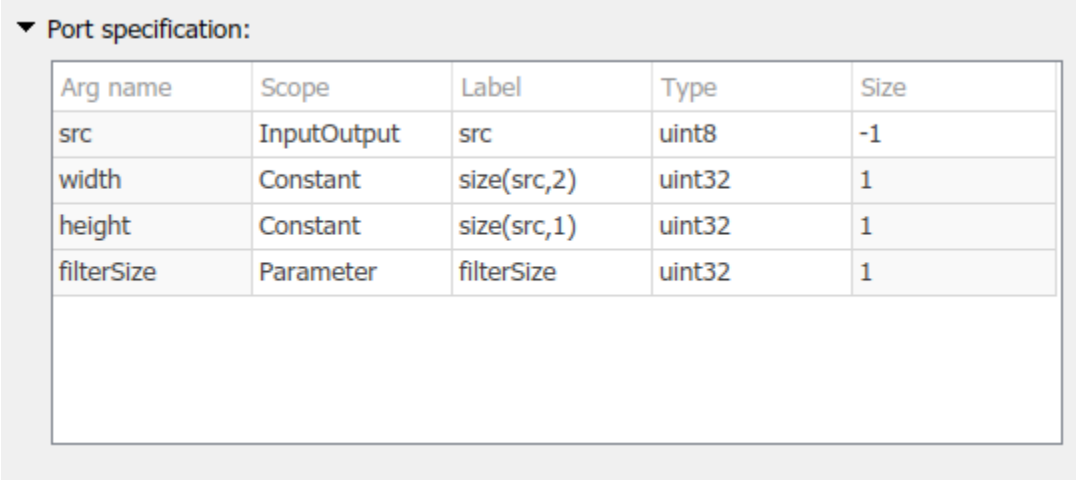
`const double *u`
`const double u[]`
`const double u[][2]`
`const double u[2][3]`
	`Input` (default), `Parameter`

Use the `InputOutput` port to map an input passed by a pointer in your C functions. Ports created using an `InputOutput` port have the same name for input and output ports. `InputOutput` ports enables reuse of buffer for input and output ports. This may optimize the memory use depending on the signal size and the block layout.

To map C function arguments to an `InputOutput` port, define the variable as a pointer in your function definitions.

```
extern void mean_filter(unsigned char* src,
                        unsigned int width, unsigned int height,
                        unsigned int filterSize);
```

Then, select the port specification to the `InputOutput` scope in the **Port Specification** table, and assign the resulting function output to the input variable in the custom function.



You can use global variables in your custom code map them to the appropriate Simulink scope. To enable the use of global variables in your model, select “Enable custom code globals as function interface” from **Model Settings > Configuration Parameters > Simulation Target**. You can map the global variables to an `Input`, `Output`, `InputOutput` or `Global` scope on the C Caller block. The availability of the these scopes depend on the use of the global variable in your custom code.

A `Global` scope enables you to transfer data between custom code and the C Caller block and lets you use the global variable during calculations on the block. Values transferred using `Global` scope are not visible on the block interface. This table shows example code snippets and their default and available ports.

Example Code	Simulink Scope

`double data;`
 	
`void foo(void)`
`    {`
`        int temp = data;`
`    }`
	Global variable data only reads the variable `data`. Available scopes are:
	
	`Input` (default)
	
	`Global`

`double data;`
 	
`void bar(void)`
`    {`
`        data = 0;`
`    }`
	Data is written to a global variable. Available scopes are:
	
	`Output` (default)
	
	`Global`
	
	`InputOutput`

`double data;`
 	
`void foo2(void)`
`    {`
`        data = data + 1;`
`    }`
	Data is both read and written on a global variable. Available scopes are:
	
	`Global` (default)
	
	`InputOutput`
	
	`Output`

Label — Indicates the label for the corresponding argument in a Simulink block. By default, your argument label is the same as the argument name, unless you change it.

Simulink Scope	Simulink Port Label
`input`, `output`	Port name
`inputoutput`	Port name in both input and output port
`Global`	Port name and global variable name
`parameter`	Parameter name
`constant`	Expression for the constant value. size expressions using input argument names, for example `size(in1,1)`

Type — Demonstrates the match between the Simulink data type and the C function argument data type.

C Argument Data Type	Simulink Data Type
signed char	int8
unsigned char	uint8
char	int8 or uint8, depending on the compiler
int*	int32
unsigned int*	uint32
short *	int16

C Argument Data Type	Simulink Data Type
long *	int32 or fixdt(1,64,0), depending on the operating system
float	single
double	double
int8_t*	int8
uint8_t*	int8
int16_t*	int16
uint16_t*	uint16
int32_t*	int32
uint32_t*	uint32
typedef struct {...} AStruct**	Bus: AStruct
typedef enum {..} AnEnum**	Enum: AnEnum
* If the C Caller takes an integer type, for example, int16_t, you can modify it to a fixed-point type with matching base type, for example to fixdt(1, 16, 3). ** The C Caller sync button prompts you to import struct or enum types used by a C function as Simulink bus and enumeration types.	

Size — Specifies the data dimensions in the argument.

C Argument Dimensions	Simulink Port Dimensions
`double u`	scalar (1)

`double u[]`
 	
`double u[][2]`
	inherited (-1) (default)
	
	If the argument is for an output port, the size should be specified. The size of an output port cannot be inherited.

`double *u`	inherited (-1) (default) If the argument is for an `inputoutput` port, the size cannot be inherited even though the size in the `output` port can be inherited. For global variables, size is scalar (1).
`double u[2][3]`	Size is [2, 3].

Create a FunctionPortSpecification Object and Edit C Caller Block Properties

To change **Port Specification** table properties programmatically, you can create a `FunctionPortSpecification` object and modify its properties. To create a `FunctionPortSpecification` object for a selected C Caller block in a model, type in the command line:

```
myCCallerConfigObj = get_param(gcb, 'FunctionPortSpecification')
```

```
myCCallerConfigObj =
```

```
  FunctionPortSpecification with properties:

        CPrototype:  'real_T add(real_T u1, real_T u2);'
     InputArguments:  [1×2 Simulink.CustomCode.FunctionArgument]
     ReturnArgument:  [1×1 Simulink.CustomCode.FunctionArgument]
     GlobalArguments: [1×0 Simulink.CustomCode.FunctionArgument]
```

The `CPrototype` property is read-only, and shows the declaration of C function input variables. The `InputArgument` and `ReturnArgument` properties create a `FunctionArgument` object that you can further edit its properties according to the rules defined for **Port Specification** table above. You can See `FunctionPortSpecification` to learn more.

To modify the global arguments in a C Caller block, create a handle to the `GlobalArguments` object using `getGlobalArg` and modify its properties.

Create a Custom C Caller Library

You can create a library model to group your C Caller blocks and keep your models organized.

1. Open a new library model. On the **Simulation** tab, select **New > Library**.
2. On the **Modeling** tab, under **Design**, click **Simulation Custom Code**.
3. Select `C` or `C++` in the **Language** option, depending on your code, and ensure the **Import custom code** box is selected.
4. Follow the instructions in “Specify Source Code and Dependencies” on page 42-2 to add your source files and their dependencies.
5. Create C Caller blocks to call C functions.
6. To insert a block from your library model to a Simulink model, simply drag the block into your model.

Generate Debug Symbols for Custom Code

To attach an external debugger to the MATLAB process, and debug external C code, generate the debug symbols using:

```
Simulink.CustomCode.debugSymbols('on')
```

After you turn this setting on and update your model, your debug symbols are generated and you can attach an external debugger to your MATLAB process.

Turn this setting off using:

```
Simulink.CustomCode.debugSymbols('off')
```

Limitations

- **Global Variables** — Global variables as function input outputs do not support multi dimensional arrays.
- **Initialization/Termination of Custom Code Settings** — If you need to allocate and deallocate memory for your custom code, insert allocate and deallocate in the **Initialize function** and **Terminate function** fields of custom code settings.
- **Complex Data Support** — The C Caller block does not support complex data types in Simulink.

- **Continuous Sample Time** — The C Caller block does not support continuous sample time.
- **Variable Arguments** — Variable arguments in C are not supported, for example, `int sprintf(char *str, const char *format, ...)`.
- **C++ Syntax** — The C Caller block does not support native C++ syntax directly. You need to write a C function wrapper to interface with C++ code.

To test your models that includes C Caller blocks, see “Test Integrated C Code” (Simulink Test).

See Also

C Caller | `FunctionPortSpecification` | MATLAB Function | MATLAB System | S-Function | S-Function Builder | `getGlobalArg` | legacy_code

More About

- “Integrate C Code Using the MATLAB Function Block” on page 44-122
- “Integrate C Functions Using Legacy Code Tool”
- “Row-Major and Column-Major Array Layouts” (MATLAB Coder)
- “Default function array layout”
- “Specify by function”

Integrate Algorithms Using C Function

Call and Integrate External C Algorithms into Simulink

In this section...

You can call and integrate your external C code into Simulink models using C Function blocks. C Function blocks allow you to call external C code and customize the integration of your code using the **Output Code**, **Start Code** and **Terminate Code** panes in the block parameters dialog. Use C Function block to:

- Call functions from external C code, and customize the code for your Simulink models.
- Preprocess data to call a C function and postprocess data after calling the function.
- Specify different code for simulation and code generation.
- Call multiple functions.
- Initialize and work with persistent data cached in the block.
- Allocate and deallocate memory.

Use the C Function block to call external C algorithms into Simulink that you want to modify. To call a single C function from a Simulink model, use the C Caller block. To integrate dynamic systems that have continuous states or state changes, use the S-Function block.

The following examples use C Function blocks to calculate the sum and mean of inputs.

Write External Source Files

Begin by creating the external source files.

1. Create a header file named `data_array.h`.

```
/* Define a struct called DataArray */
typedef struct DataArray_tag {
    /* Define a pointer called pData */
    double* pData;
    /* Define the variable length */
    int length;
} DataArray;

/* Function declaration */
double data_sum(DataArray data);
```

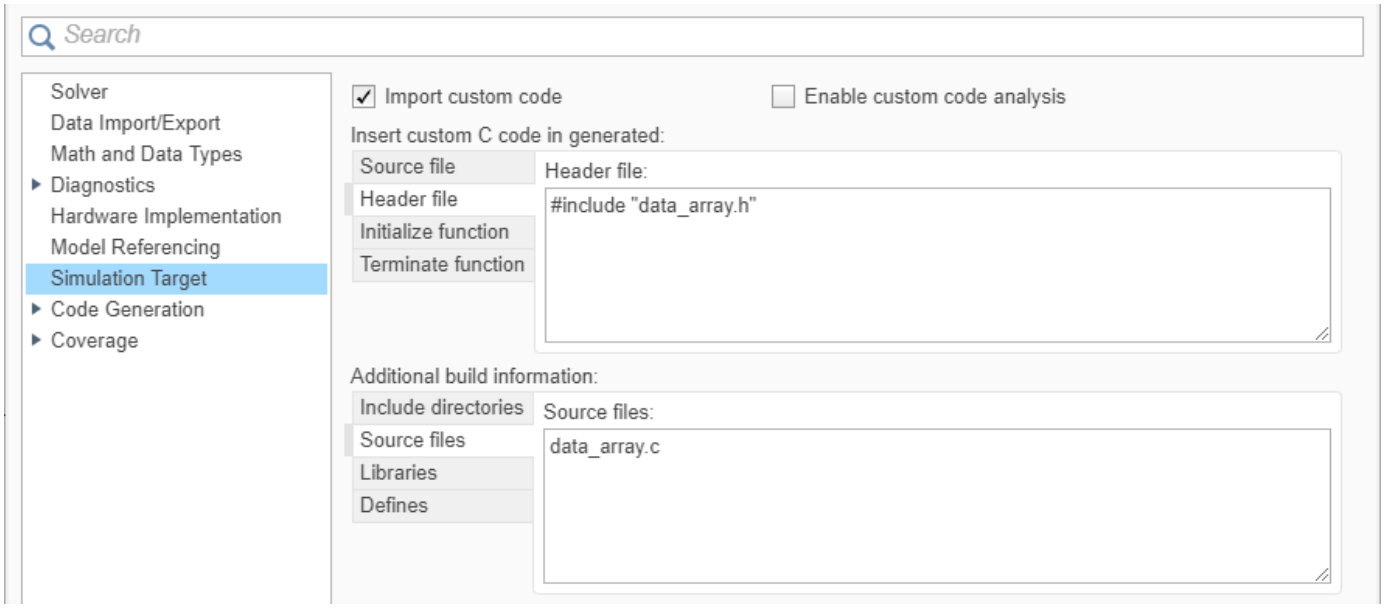
2. In the same folder, create a new file, `data_array.c`. In this file, write a C function that calculates the sum of input numbers.

```
#include "data_array.h"

/* Define a function that takes in a struct */
double data_sum(DataArray data)
{
    /* Define 2 local variables to use in the function */
```

```
    double sum = 0.0;
    int i;
    /* Calculate the sum of values */
    for (i = 0; i < data.length; i++) {
        sum = sum + data.pData[i];
    }
    /* Return the result to the block */
    return sum;
}
```

Enter the External Code Into Simulink

1. In a new, blank model, add a C Function block. The C Function block is in the **User-Defined Functions** library of the Library Browser.
2. Double-click the C Function block to open the block dialog. Click  to open the **Configuration Parameters** dialog. In the **Simulation Target** pane, define your header file under **Insert custom C code in generated: > Header file**.
3. Define the source file under **Additional build information > Source files**.



Click **OK** to close the Configuration Parameters.

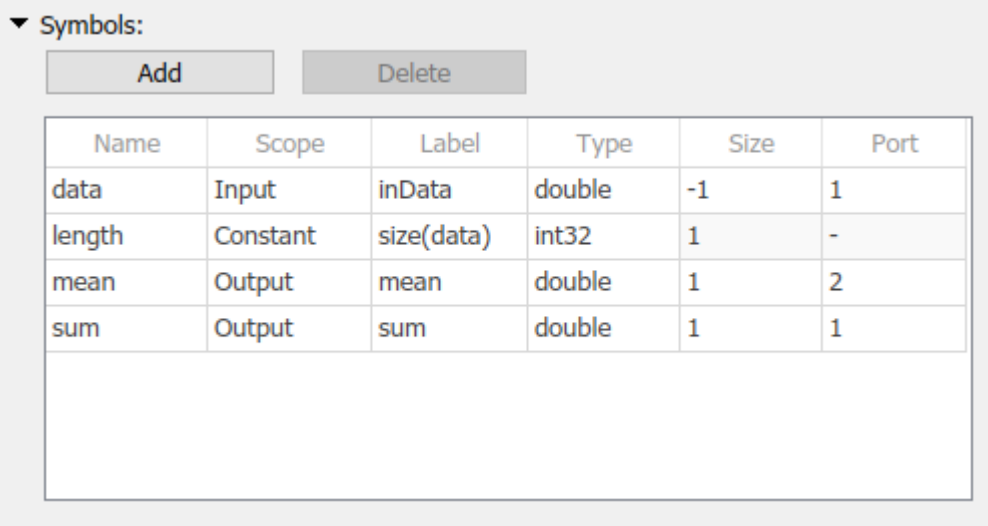
4. In the **Output Code** pane of the C Function block parameters dialog, write the code that the block executes during simulation. In this example, the external C function computes a sum. In the **Output Code** pane, write code that calls the `data_array.c` function to compute the sum, then computes the mean.

```
/* declare the struct dataArr */
DataArray dataArr;
/* store the length and data coming in from the input port */
dataArr.pData = &data;
dataArr.length = length;

/* call the function from the external code to calculate sum */
```

```
sum = data_sum(dataArr);

/* calculate the mean */
mean = sum / length;
```

You can specify code that runs at the start of a simulation and at the end of a simulation in the **Start Code** and **Terminate Code** panes.

5 Use the **Symbols** table to define the symbols used in the external C code. Add or delete a symbol using the **Add** and **Delete** buttons. Define all symbols used in the **Output Code**, **Start Code**, and **Terminate Code** panes to ensure that ports display correctly.



In the **Symbols** table, define the following.

- **Name** — Symbol name in the source code.
- **Scope** — Scope of the symbols and the order in which they appear. You can change the scope of a symbol at any time.
  - `Input` — Input symbol to the C Function block.
  - `Output` — Output symbol to the C Function block.
  - `InputOutput` — Define a symbol as both input and output to the C Function block.

    Use the `InputOutput` scope to map an input passed by a pointer in your C code. Ports created using an `InputOutput` scope have the same name for input and output ports. `InputOutput` scope enables the reuse of buffer for input and output ports. Reusing buffer may optimize memory use and improve code simulation and code generation efficiency, depending on the signal size and the block layout. Limitations include:

    - `InputOutput` symbol cannot be used in `Start` and `Terminate` code.
    - `InputOutput` port does not support `void*` data type.
    - `InputOutput` port does not support `size()` expressions.
  - `Persistent` — Define a symbol as persistent data.

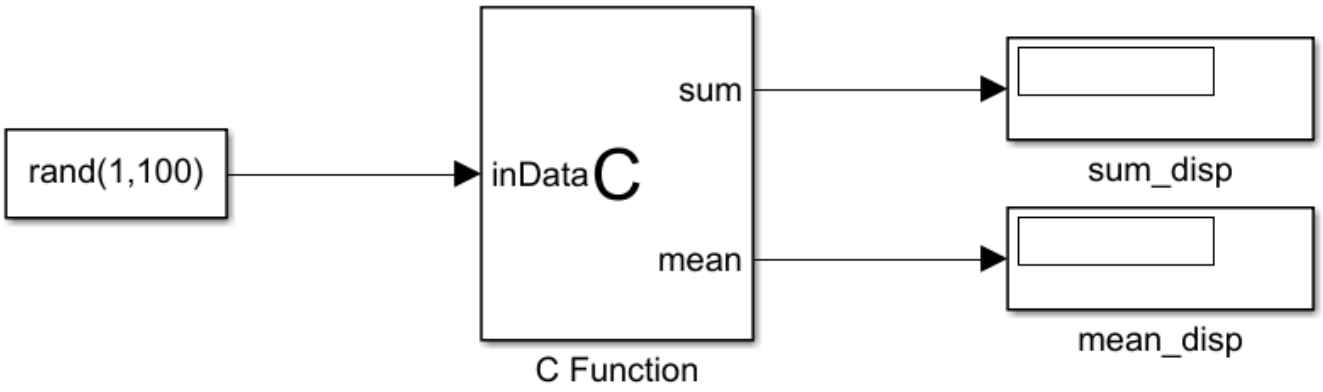
    You can define a void pointer using the `Persistent` scope in the C Function block. A *void pointer* is a pointer that can store any type of data that you created or allocated.

  - `Constant` — Define a symbol as constant using value-size or numeric expressions.
  - `Parameter` — Define a symbol as parameter. The parameter name is defined by the **Label** of the symbol.
- **Label** — Label of the symbol. For symbols with their scope set to `Input` or `Output`, this label appears as the port name on the block. For symbols with their scope set to `Parameter`, this label is the label that appears on the block parameter mask. You cannot define a label for `Persistent` symbols. If the scope is `Constant`, the label is the constant expression.
- **Type** — Data type of the symbol. Select a data type from the drop-down list or specify custom data type.

  To use a custom type such as `Simulink.Bus`, Simulink Enum on page 68-2 or `Simulink.AliasType` that does not have an external header definition associated with a C Function block, set the type correctly on the **Symbol** table.
- **Size** — Size of the symbol data. The C Function block supports only scalars and vectors are supported. Matrices and higher-dimension arrays are not supported. You can use a size expression to define the size of an output. Use `-1` to inherit size.
- **Port** — For input and output symbols, `Port` indicates the port index on the block of the symbol data. For parameter symbols, `Port` indicates the order that the symbol appears in the block parameter mask.

Close the block parameters dialog. After filling in the data in the table, the C Function block now has one input port, and two output ports with the labels specified in the table.

6 Add a Constant block to the Simulink canvas that will be the input to the C Function block. In the Constant block, create a random row array with 100 elements. To display the results, attach display blocks to the outputs of the C Function block.



Call C Library Functions From C Function Block

You can call this subset of the C Math Library functions from the C Function block:

`abs`	`acos`	`asin`	`atan`	`atan2`	`ceil`
`cos`	`cosh`	`exp`	`fabs`	`floor`	`fmod`
`labs`	`ldexp`	`log`	`log10`	`pow`	`sin`
`sinh`	`sqrt`	`tan`	`tanh`		

When you call these functions, double precision applies unless all the input arguments are explicitly single precision. When a type mismatch occurs, a cast of the input arguments to the expected type replaces the original arguments. For example, if you call the `sin` function with an integer argument, a cast of the input argument to a floating-point number of type `double` replaces the original argument.

If you want to call other C library functions that are not listed above, create an external wrapper function that calls the C library function.

Call the abs, fabs, and labs Function

Interpretation of the `abs`, `fabs`, and `labs` functions in C Function block goes beyond the standard C version to include integer and floating-point arguments:

- If `x` is an integer, the standard C function applies to `x`, or `abs(x)`.
- If `x` is a double, the standard C function `labs` applies to `x`, or `labs(x)`.
- If `x` is a single, the standard C function `fabs` applies to `x`, or `fabs(x)`.

Code Replacement Library (CRL) Based on Type

The call to the function should call the correct CRL based on the type of data passed into the function. If no CRL is specified, the call to the function should call to type-specific library. The CRL for C99 generates a type-specific function. For example:

Type passed in	Code generation call
`sin(doubleIn)`	`sin(doubleIn)`
`sin(floatIn)`	`sinf(floatIn)`

Specify Simulation or Code Generation Code

You can specify different output code for simulation and code generation for the C Function block by defining `MATLAB_MEX_FILE`. For example, to specify code that only runs during the model simulation, you use the following.

```
#ifdef MATLAB_MEX_FILE
/* Enter Sim Code */
#else
/* Enter code generation code */
#endif
```

See Also

Functions
`addSymbol` | `deleteSymbol` | `getSymbol`

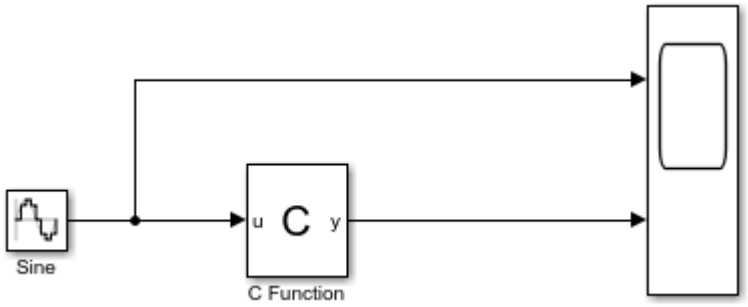
Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

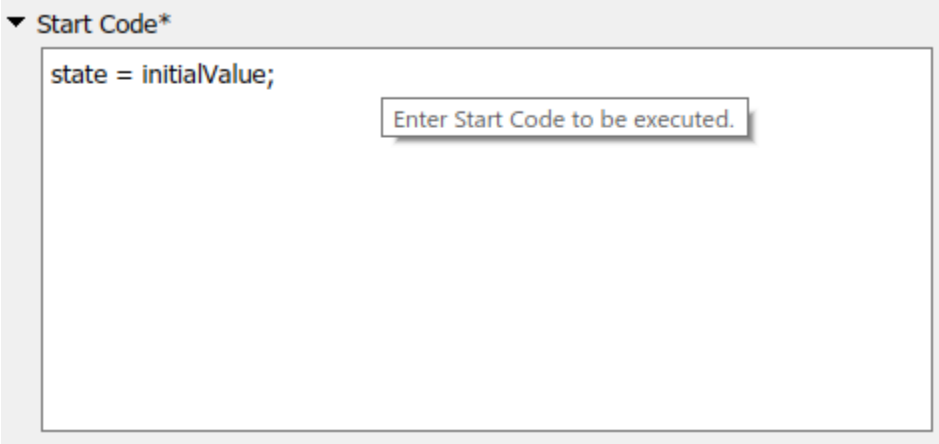
Modify States of a C Function Block Using Persistent Symbols

This example shows a unit delay system where a C Function block takes in a sine wave signal and delays its output by a specified sample period.

```
open_system('CFunctionPersistentExample');
```

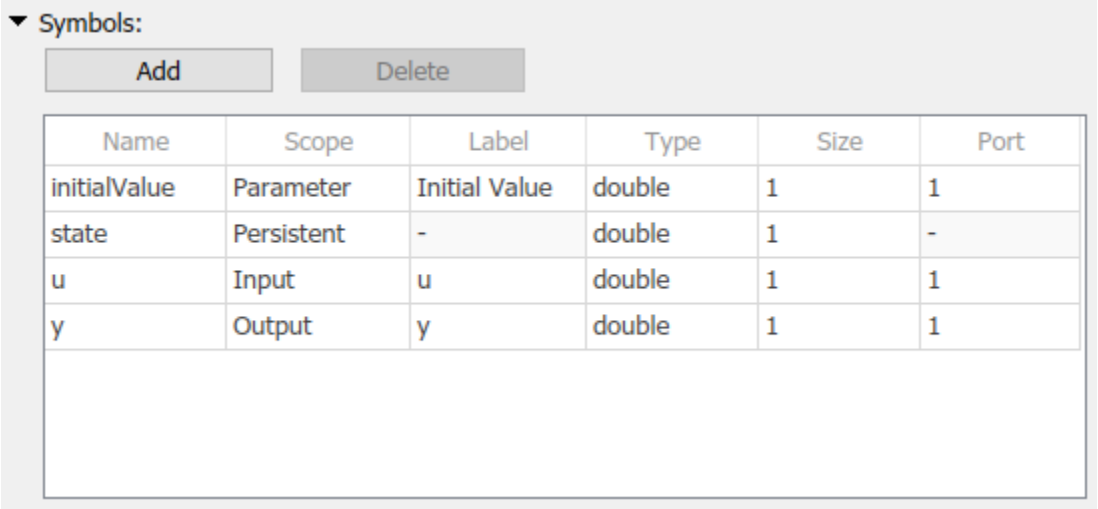


An initial value, specified as a parameter, is cached in the block as persistent data in the **Start Code** pane.



In the **Output Code** pane, a calculation is done on this state. A new input is passed to the block as the next state at the next time step.

The **Symbols** table defines the attributes of the symbols used in the code. Note that the scope of the `initialValue` symbol is specified as `Parameter`, so its value can be changed using the block dialog. The state is defined as having `Persistent` scope. This value changes at each time step.



Name	Scope	Label	Type	Size	Port
initialValue	Parameter	Initial Value	double	1	1
state	Persistent	-	double	1	-
u	Input	u	double	1	1
y	Output	y	double	1	1

See Also

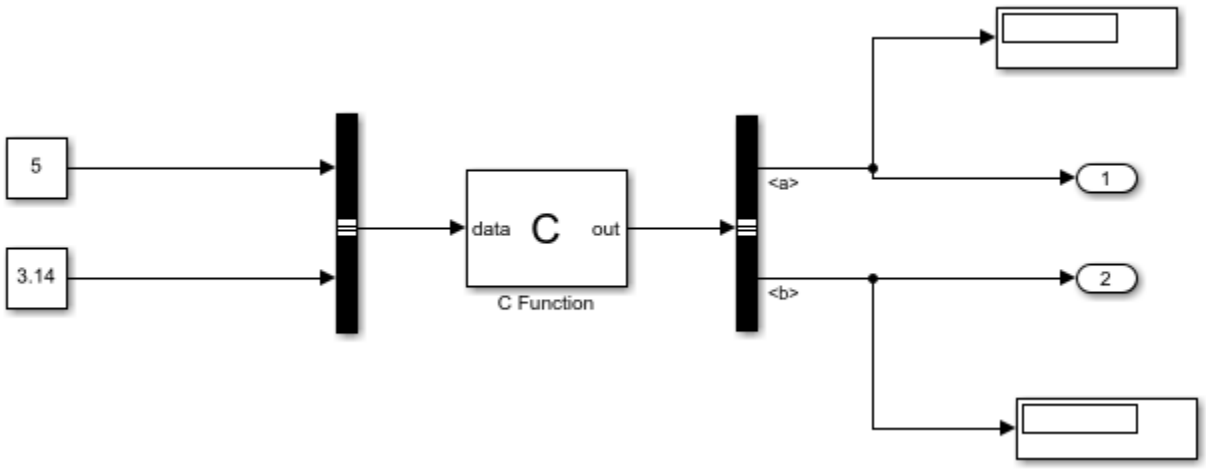
Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Change Values of Signals Using C Function Block and Buses

This example shows how to use buses with a C Function block. In this example two Constant blocks supply values. These signals are combined using a Bus Creator block. Code in the C Function block changes the values of these signals. Using a Bus Selector block, the values are displayed separately in two Display blocks.

```
open_system('mCFunction_BusWithoutCStruct')
```

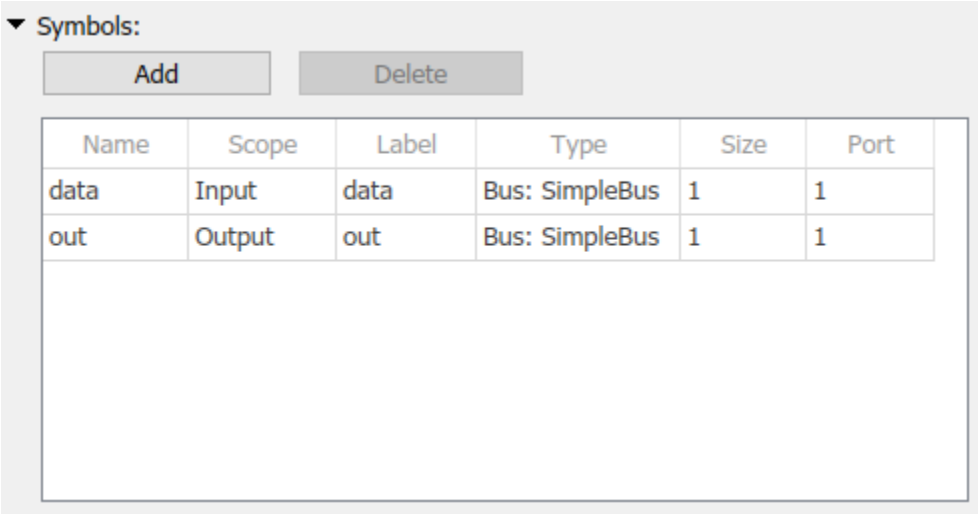


The **Output Code** pane contains code that changes the values of the signals.

Output Code

```
out.a = data.a * 2;
out.b = data.b + 3;
```

Add the symbols used in the code to the **Symbols** table. To use buses with a C Function block, set the **Type** of the symbol to `Bus: SimpleBus`.



See Also

Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Access Elements of a Matrix Using Output Code in a C Function Block

This example shows how to access elements of a matrix from a C Function block using the **Output Code** pane.

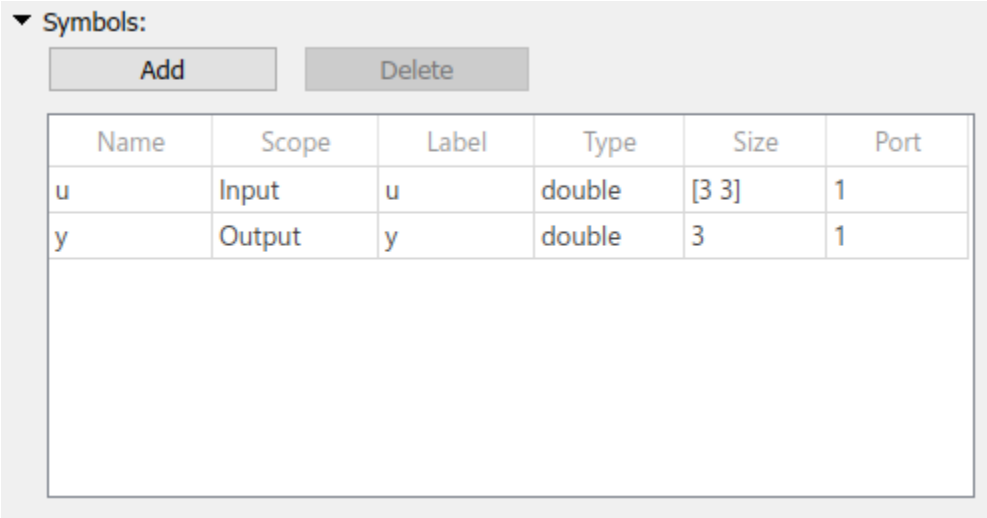
```
open_system('mMatrixColumnSumOutputCode')
```

Elements in each column of the input matrix are accessed and added to calculate a sum for each column.

Output Code

```
int r, c;

for (c = 0; c < 3; c++) {
    y[c] = 0;
    for (r = 0; r < 3; r++) {
        y[c] += u[r][c];
    }
}
```

You must define all symbols used in code on the **Symbols** table of the Block Parameters. To set the size of the input matrix, specify size as `[r c]`. `r` corresponds to the number of rows in the input matrix and `c` corresponds to the number of columns in the input matrix.



See Also

Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Use External Functions with Matrix Input in a C Function Block

This example shows how to pass a matrix input to a C Function block and do row-major operations using external custom code.

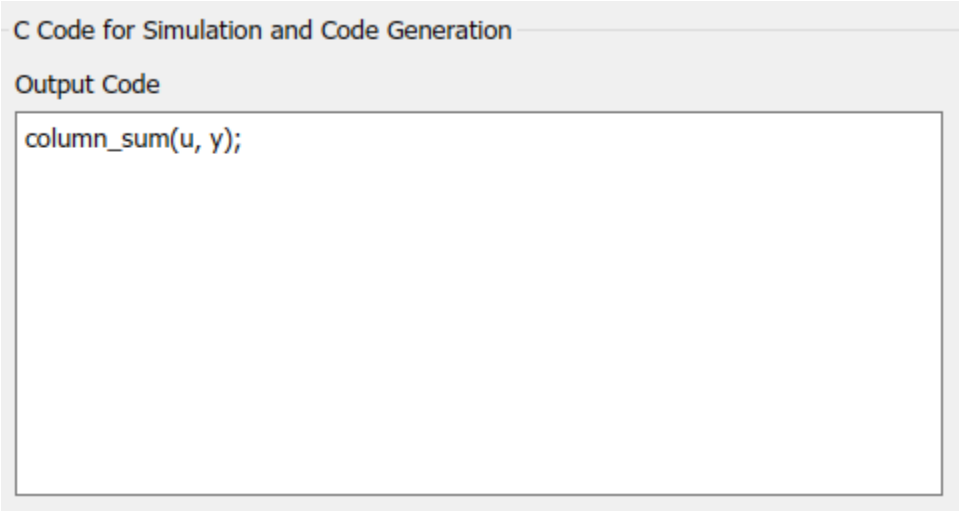
```
open_system('mMatrixColumnSumExternalCode');
```

In this example, a matrix input is used by external custom code to calculate the sum of the each column of the matrix and the results are passed to the output block. The external function accesses input array `arr` as row-major using `arr[r][c]`.

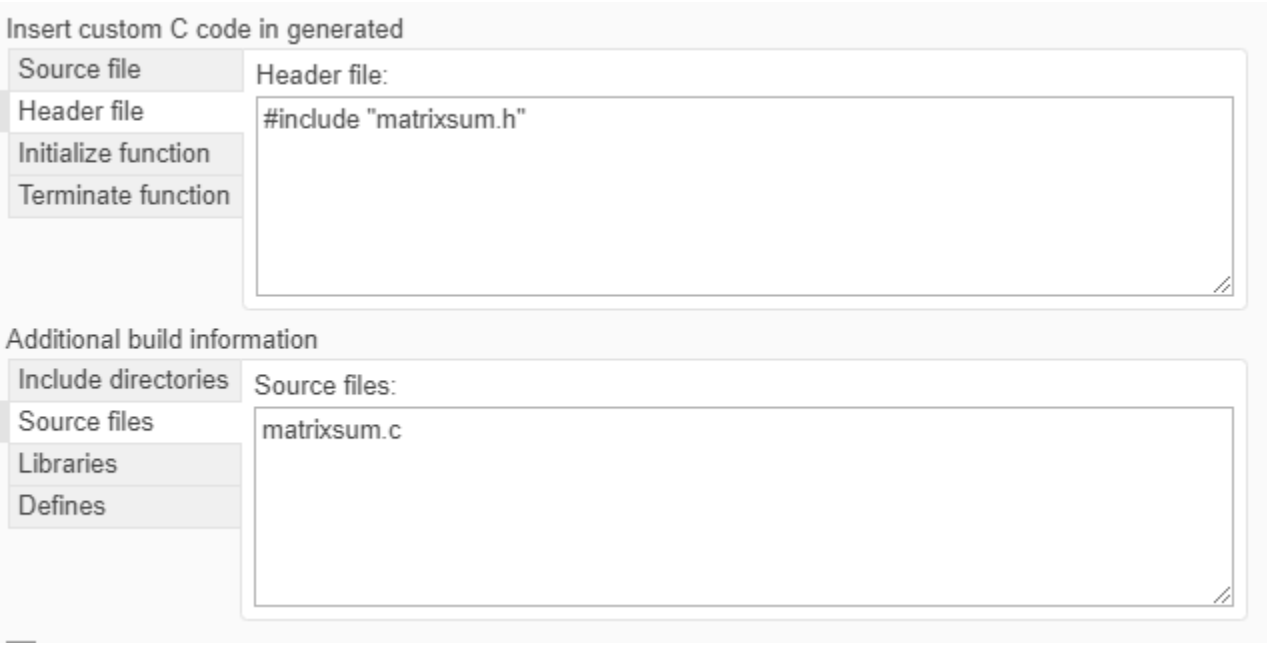
matrixsum.c

```
#include "matrixsum.h"

void column_sum(real_T arr[3][3], real_T sum[3])
{
    int r, c;

    for (c = 0; c < 3; c++) {
        sum[c] = 0;
        for (r = 0; r < 3; r++) {
            sum[c] += arr[r][c];
        }
    }
}
```

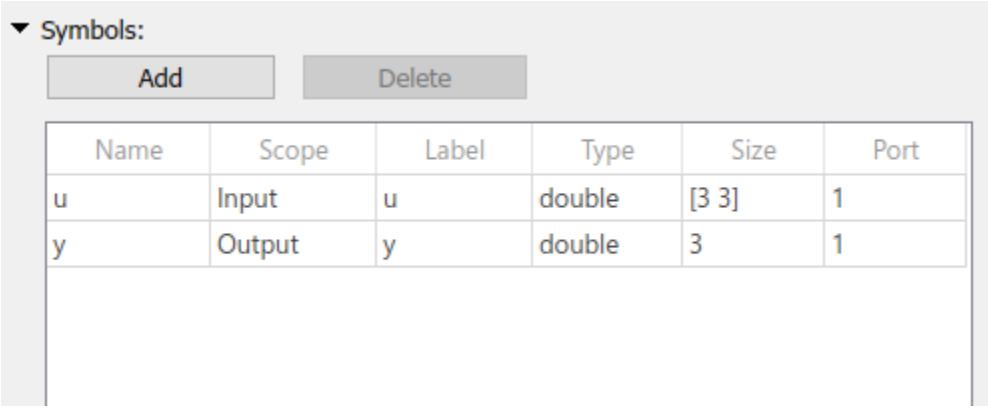
The custom code is called in the **Output Code** pane of the C Function block.



The external source and header files are specified on the **Configuration Parameters** > **Simulation Target** pane.

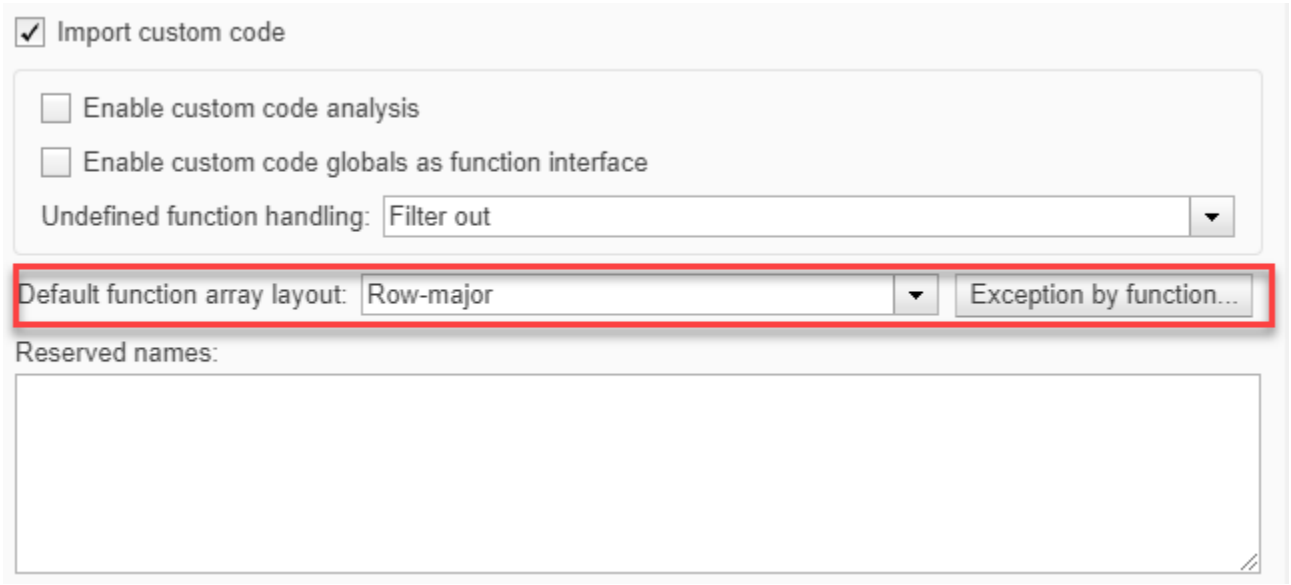


For the matrix input, the size of the input symbol on the matrix table is specified as `[r c]`, where `r` corresponds to the number of rows in the matrix input and `c` corresponds to the number of columns.



Specify the Default Function Array Layout

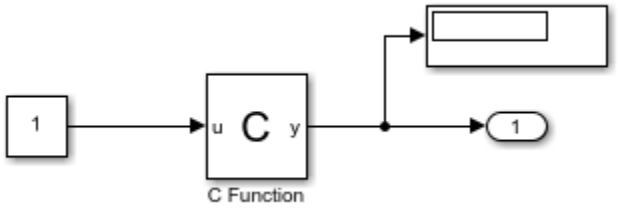
The external C function accesses input array `arr` using row-major layout. To ensure the validity of calculations with MATLAB, go to **Default Function Array Layout** under **Configuration Parameters** > **Simulation Target** and select `Row-major`.



Define an Alias Type in a C Function Block

This example shows how to specify alias data types in a C Function block. The model reads in a source and header file from the **Configuration Parameters** > **Simulation Target** pane. Calculations are performed using the functions in the source file.

```
Simulink.importExternalCTypes('multiply_func.h');
open_system('mCFunction_AliasType');
```

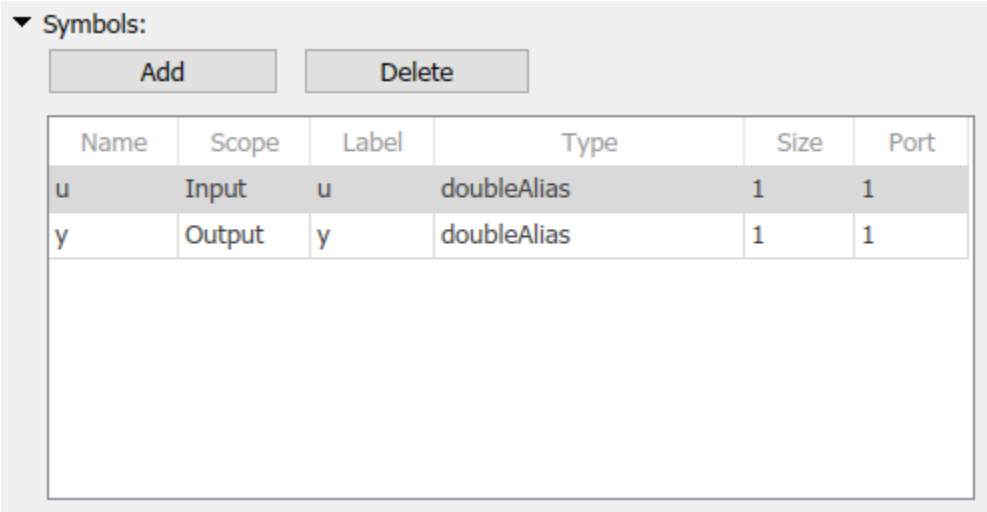


Define an Alias type in the header file

To define an alias type in your C source code to be used in your `C Function` block, use the `typedef` keyword. In this example, a double alias is defined as `typedef double doubleAlias` in the `multiply_func.h` file.

Specify the alias types in the C Function block

You must define your symbols in the **Symbols** table of the C Function block parameters. Define the symbols in the block as alias types by entering the alias type in the **Type** column of the **Symbols** table.



See Also

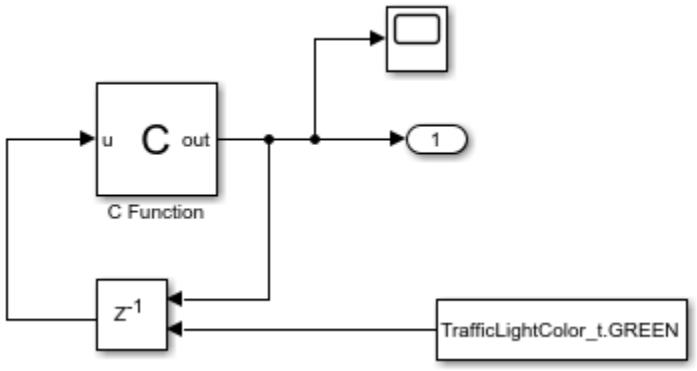
Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Use Enumerated Data in a C Function Block

This example shows how to define and use enumerated data in a C Function block. In this example, a series of traffic lights are defined in an Enumerated Constant block using the Data Type Assistant. The code that controls the traffic lights is written in the **Output Code** pane of the C Function block dialog. In the **Symbols** table, the data types of the block inputs and outputs are defined as `Enum: `*classname*. In this example, the enumerations are defined in the class `TrafficLightColor_t`, so the type is defined as `Enum: TrafficLightColor_t`.

```
open_system('CFunctionEnum')
```

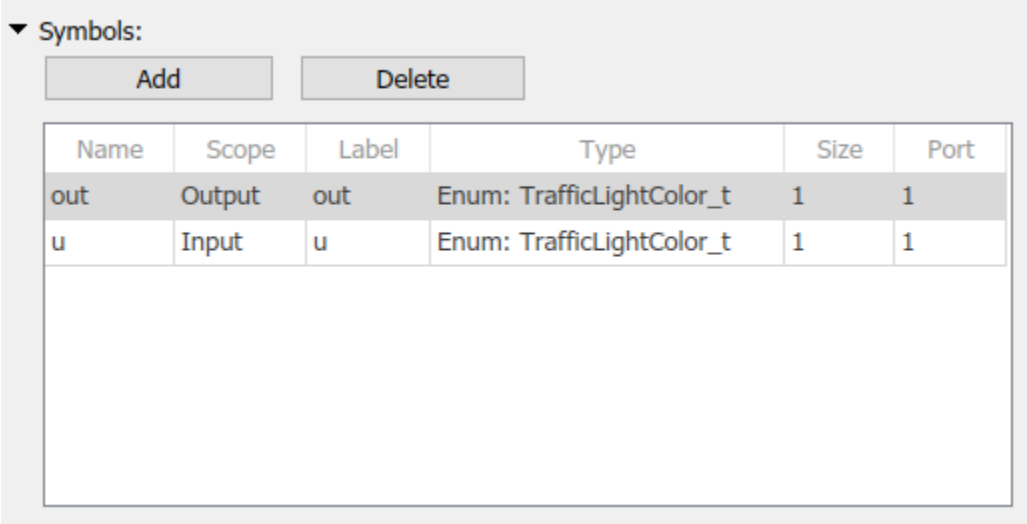


Output Code to Switch Traffic Lights

Output Code

```
switch(u) {
   case RED:
      out = GREEN;
      break;
   case YELLOW:
      out = RED;
      break;
   case GREEN:
      out = YELLOW;
      break;
   default:
      out = UNKNOWN;
}
```

Symbols Table to Define Enumerations



See Also

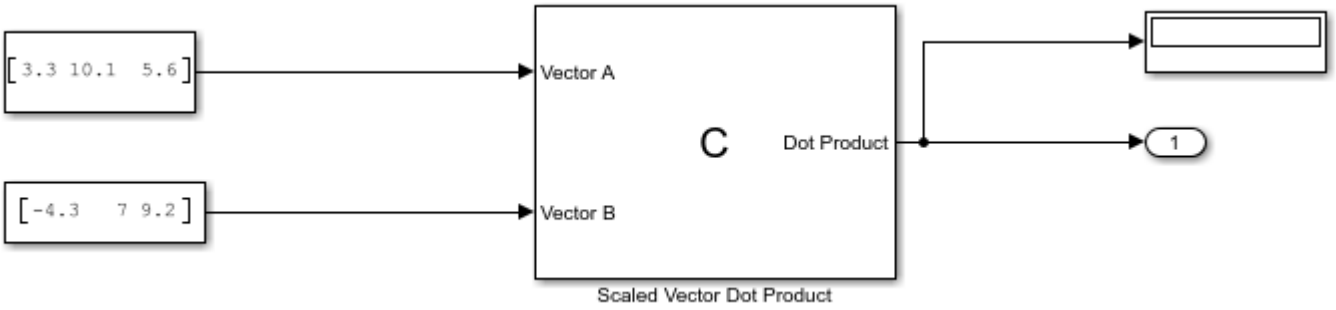
Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Use Inherited Sizes in a C Function Block

This example shows how to use a C Function block to compute a scaled vector dot product from two input vectors. In this example, the C Function block takes two row vectors as inputs with inherited sizes. The block scales the data using parameters. The vector dimension, which is used to define loop indices in the C code, is defined as a constant. The block calculates the dot product, and displays the results.

```
open_system('CFunctionVectorDotProduct');
```



In the C Function Block Parameters dialog, the **Output Code** pane contains the code that performs the vector dot product calculations on the two input vectors.

Output Code

```
int i;
dotProduct = 0;
for(i = 0; i < vectorDimension; ++i){
    dotProduct = dotProduct + (scaleA*vectorA[i] * scaleB*vectorB[i]);
}
```

You must define all symbols used in the code in the **Symbols** table of the Block Parameters. To specify that the sizes of the input vectors are inherited, specify `-1` in the **Size** field of the table. The symbols `scaleA` and `scaleB` are defined as parameters of the block. These parameters appear on the block parameter mask. The definitions of all symbols used in this example are shown in the following table.

Name	Scope	Label	Type	Size	Port
dotProduct	Output	Dot Product	double	1	1
scaleA	Parameter	Scale A by	double	1	1
scaleB	Parameter	Scale B by	double	1	2
vectorA	Input	Vector A	double	-1	1
vectorB	Input	Vector B	double	-1	2
vectorDim...	Constant	size(vecto...	int32	1	-

See Also

Objects
`Symbol` | `SymbolSpec`

Blocks
C Function

Call a Legacy Lookup Table Function Using C Caller block

This example shows how to use the C Caller block to call legacy C functions that implement N-dimensional table lookups.

In this example the legacy C functions are defined in `lookupTable.h`, and implemented in `directLookupTableND.c`. The type definitions used in this example can be found in `your_types.h`.

To enable calling the C functions from the C Function block, in the model, specify the header file and the source file in the **Configuration Parameters** > **Simulation Target** pane. The prototype of the legacy function being called in this example is:

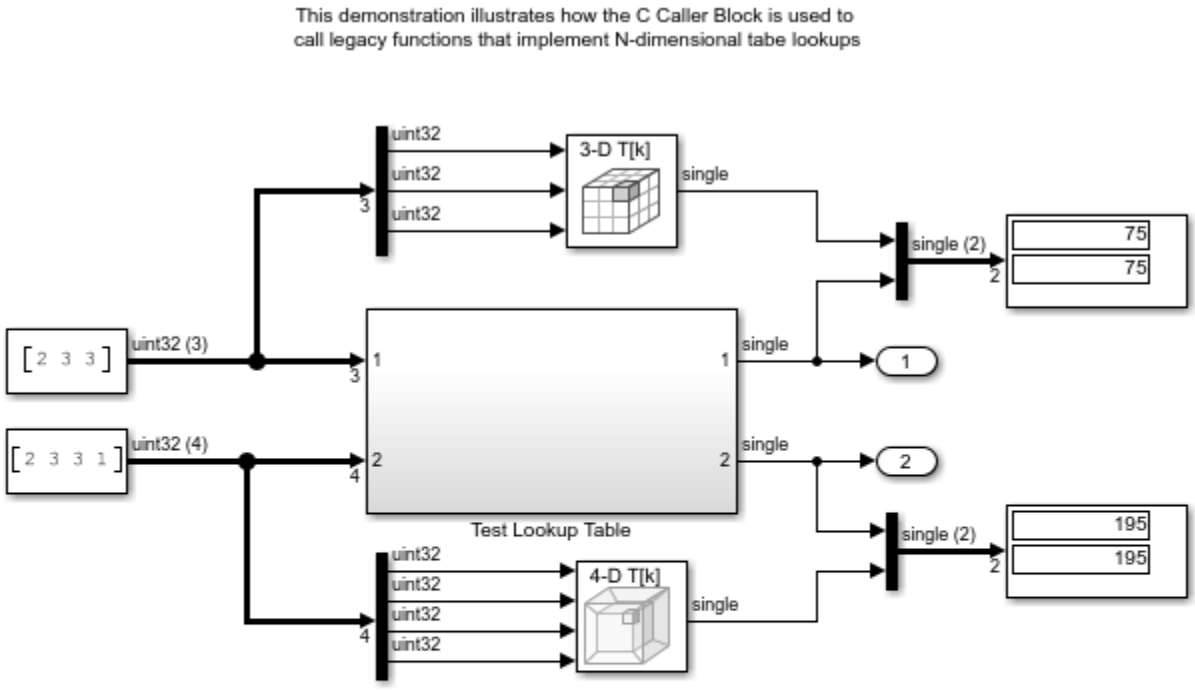
```
FLT directLookupTableND(const FLT *tableND, const UINT32 nbDims, const UINT32
*tableDims, const UINT32 *tableIdx)
```

where `FLT` is a type definition to a floating-point type, and `UINT32` is a type definition to an unsigned 32-bit integer.

- `const FLT *tableND` - Table
- `const UINT32 nbDims` - Dimensionality of the table
- `const UINT32 *tableDims` - Size of the table
- `const UINT32 *tableIdx` - Table index

In the C Caller block, `tableND` and `tableDims` are mapped to C Caller block parameters, `nbDims` is a block constant, and `tableIdx` is the input to the block. The value returned by the legacy C function is the output of the block.

```
model = 'slexCCallerLookupTable';
open_system(model);
sim(model);
slcc('clearCustomCodeModules');
```



Start and Terminate Actions Within a C Function Block

This example shows how to use the C Function block to integrate legacy C functions that have start and terminate actions.

In this example, the legacy C functions are defined in `fault.h`, and implemented in `fault.c`.

To enable calling the C functions from the C Function block, in the model, specify the header file and the source file in the **Configuration Parameters > Simulation Target** pane. This model opens a log file, writes data to it, and then closes the log file. The `openLogFile` function is called in the **Start Code** section and the `closeLogFile` function is called in the **Terminate Code** section of the C Function block. In the **Output Code** section, the `incAndLogFaultCounter` function is called. The prototype of the legacy functions are:

- `void* openLogFile()`
- `closeLogFile(void* fid)`
- `incAndLogFaultCounter(void *fid, unsigned int counter, double time)`

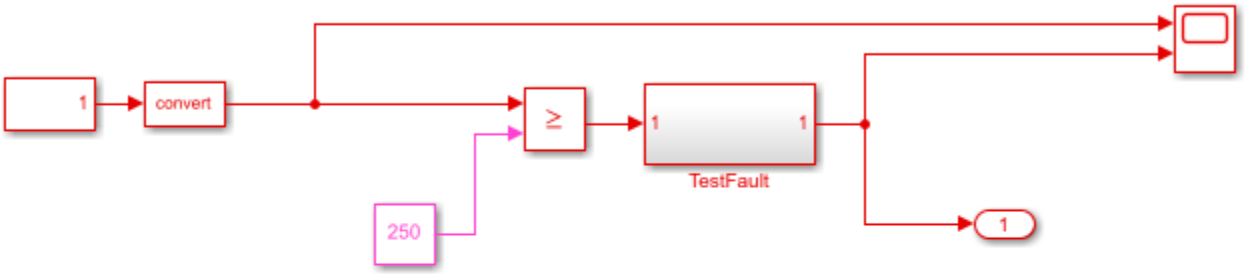
where, `void *fid` is a file pointer, `unsigned int counter` is the fault counter, and `double time` indicates the time.

In this example the ports of the C Function block are configured as follows:

- The file pointer returned by `openLogFile` function and the fault counter are persistent symbols of the C Function block.
- The input of the C Function block is mapped to time.

```
model = 'slexCFunctionStartTerm';
open_system(model);
sim(model);
slcc('clearCustomCodeModules');
```

This demonstration illustrates how the C Function Block is used to
call legacy functions that have start and terminate actions



Start Code → call openLogFile() → open the file 'slex_CFunction_fault.log' for logging faults
Output Code → call incAndLogFaultCounter() → increment the fault counter and append one line in the log file
Terminate Code → call closeLogFile() → close the logging file

- run the model and edit the generated logging file 'slex_CFunction_fault.log'

Call C++ Class Methods Using a C-style Wrapper Function From a C Function Block

This example shows how to use the C Function block to call C++ class methods using a C-style wrapper function. In this example, the C-style wrapper functions are defined in `adder_cpp.h`, and implemented in `adder_cpp.cpp`. To run this example model successfully, you must configure MATLAB® to use a C++ compiler.

This can be done using the `mex -setup` command, and selecting a C++ compiler.

To enable calling of the C functions from the C Function block, in the model, specify the header file and the source file in the **Configuration Parameters > Simulation Target** pane. In this model the custom code header file `adder_cpp.h` declares an `adder` class along with the class member functions. The header file also contains the declaration of the C-style wrapper functions which are called in the C Function block. The prototypes of the C-style wrapper functions are:

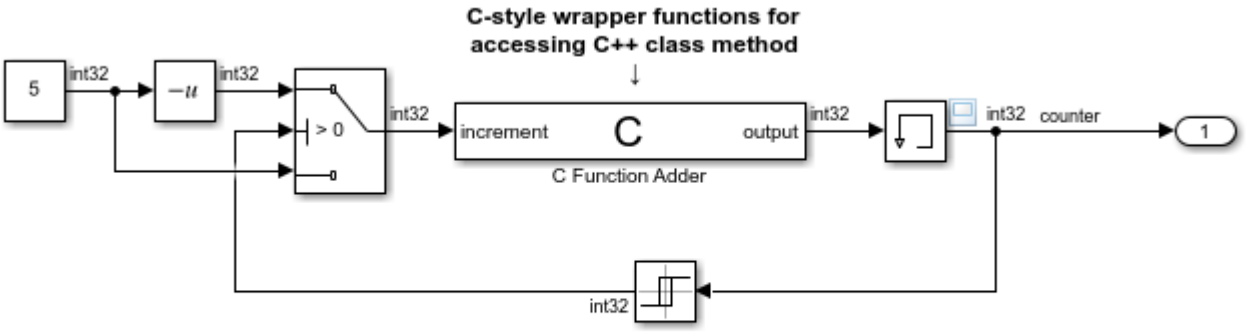
- `void *createAdder()`
- `void deleteAdder(void *obj)`
- `int adderOutput(void *obj, int increment)`

where, `void *obj` is the class object pointer, and `int increment` is an input used by the adder function.

The `createAdder` function is called in the **Start Code** section of the Block Parameters to construct an object of the `Adder` class. The `deleteAdder` function is called in the **Terminate Code** section to destruct the created Adder class object. In the **Output Code** section, the `adderOutput` function is called. In this example the ports of the C Function block are configured as follows:

- The Adder class object, `obj`, return by the `createAdder` function is a persistent symbol of the C Function block.
- The input of the C Function block is mapped to `increment`.
- The output of the C Function block is mapped to the return value of the `adderOutput` function.

```
model = 'slexCFunctionAdder';
open_system(model);
sim(model);
slcc('clearCustomCodeModules');
```



NOTE:
For this demo to run you must go to the MATLAB command prompt and enter: >> *mex -setup*
then select a C++ compiler.

Call Legacy Lookup Table Functions Using C Function Block

This example shows how to use the C Function block to call legacy C functions that implement n-dimensional table lookups.

In this example the legacy C functions are defined in lookupTable.h, and implemented in directLookupTableND.c. The type definitions used in this example can be found in your_types.h.

In the model, the header file and the source file are specified in **Model Configuration Parameters > Simulation Target**. Now the C function can be called in the C Function block. The prototype of the function being called in this example is:

```
FLT directLookupTableND(const FLT *tableND, const UINT32 nbDims, const UINT32 *tableDims, const UINT32 *tableIdx)
```

where `FLT` is a type definition to a floating-point type, and `UINT32` is a type definition to an unsigned 32-bit integer.

This example defines two macro functions that use the function `directLookupTableND`, which are called using the C Function blocks. They are:

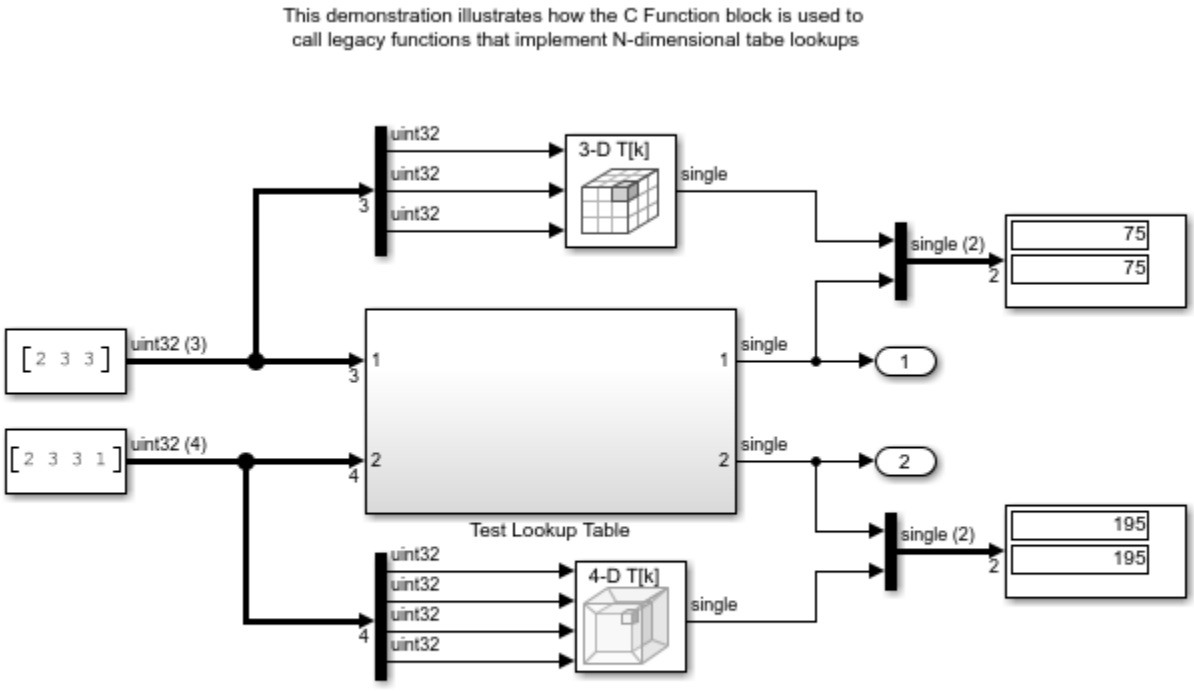
- `DirectLookupTable3D` - 3D lookup table
- `DirectLookupTable4D` - 4D lookup table

Both these functions take the following input arguments:

- `const FLT *tableND` - Table
- `const UINT32 *tableDims` - Size of the table
- `const UINT32 *tableIdx` - Table index

where the `tableND` and `tableDims` are mapped to C Function block parameters and `tableIdx` is the input to the block. The value returned by the legacy C function is the output of the block.

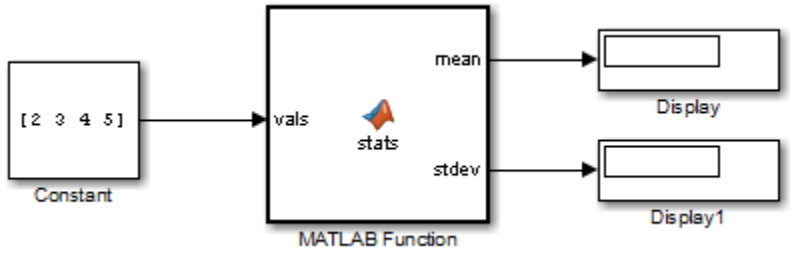
```
model = 'slexCFunctionLookupTable';
open_system(model);
evalc('sim(model)');
slcc('clearCustomCodeModules');
```



Using the MATLAB Function Block

Integrate MATLAB Algorithm in Model

Here is an example of a Simulink model that contains a MATLAB Function block:



The MATLAB Function block contains the following algorithm:

```
function [mean,stdev] = stats(vals)

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

You build this model in “Create Custom Functionality Using MATLAB Function Block” on page 44-6.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

Implementing MATLAB Functions Using Blocks

MATLAB Function blocks enable you to define custom functionality in Simulink models by using the MATLAB language. They are the easiest way to bring MATLAB code into Simulink. MATLAB Function blocks support C/C++ code generation from Simulink Coder and Embedded Coder.

Use these blocks specifically when:

- You have an existing MATLAB function that models custom functionality, or it would be easy for you to create such a function.
- Your model requires custom functionality that is not or cannot be captured in the Simulink graphical language.
- You find it easier to model custom functionality by using a MATLAB function than by using a Simulink block diagram.
- The custom functionality that you want to model does not include continuous or discrete dynamic states. To model dynamic states, use S-functions. See “Create and Configure MATLAB S-Functions”.

How MATLAB Function Blocks Work

When you simulate a model that contains a MATLAB Function block, the software generates binary code or C/C++ MATLAB executable (MEX) code from the block and integrates this code with the model. The MATLAB Function block uses the same infrastructure as MATLAB Coder, which you use to generate C/C++ code from MATLAB code outside of Simulink.

Because the MATLAB Function block relies on code generation technology, it shares much in common with MATLAB Coder. C/C++ code generation limitations for MATLAB Coder also apply to MATLAB Function blocks. However, the MATLAB Function block is self-contained within Simulink, and does not require MATLAB Coder. To generate standalone C/C++ code from a model that contains MATLAB Function blocks, use Simulink Coder.

MATLAB Function Block Capabilities

The following describes *what* you can use a MATLAB Function for in your model. To see *how* to use a MATLAB Function block in an example, see “Create Custom Functionality Using MATLAB Function Block” on page 44-6.

Simulink to MATLAB Interface

MATLAB Function blocks provide an intuitive interface between MATLAB code and a Simulink model. The block input and output variables inherit their properties from Simulink input and output signals.

By default, both the size and type of input and output signals to a MATLAB Function block are inherited from the corresponding Simulink signals. You can also choose to specify the size and type of inputs and outputs explicitly in the Ports and Data Manager or in the Model Explorer. See “Ports and Data Manager” on page 44-29 and **Model Explorer**.

Standalone C/C++ Code Generation

MATLAB Function blocks are supported for C/C++ code generation with Simulink Coder and Embedded Coder. By using code generation on a Simulink model that contains a MATLAB Function block, you can deploy MATLAB functionality outside the MATLAB environment.

For more information on C/C++ code generation from a Simulink model, see “Simulink Coder”.

MATLAB Language and Function Support

In a MATLAB Function block, you can only use the subset of the MATLAB language and language features that are supported for C/C++ code generation. For a list of functions supported for code generation, see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2. For supported language features, see “MATLAB Language Features Supported for C/C++ Code Generation” on page 48-17.

Extrinsic Functions

For simulation, you can call *extrinsic* functions from a MATLAB Function block. Extrinsic functions are functions that are not supported for C/C++ code generation but which can be dispatched to the MATLAB environment for execution during run time. Extrinsic functions execute in the workspace during model simulation.

For code generation, Simulink Coder attempts to compile all functions in a MATLAB Function block unless you explicitly declare them as extrinsic. Extrinsic function calls are elided from generated standalone code, such as standalone C/C++ source code or executable files. See “Resolution of Function Calls for Code Generation” on page 64-2 and “Declaring MATLAB Functions as Extrinsic Functions” on page 64-9.

Simulink Function Block and Stateflow Block Support

From MATLAB Function blocks, you can call functions defined in a Simulink Function block. You can call Stateflow functions when you select the **Export Chart Level Functions (Make Global)** and **Allow exported functions to be called by Simulink** check boxes in the chart Properties dialog box. To learn more about how to call functions defined in Simulink Function and Stateflow blocks, see “Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions” on page 10-121.

See Also

MATLAB Function | `coder.extrinsic`

More About

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134

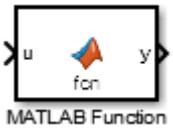
Create Custom Functionality Using MATLAB Function Block

In this section...

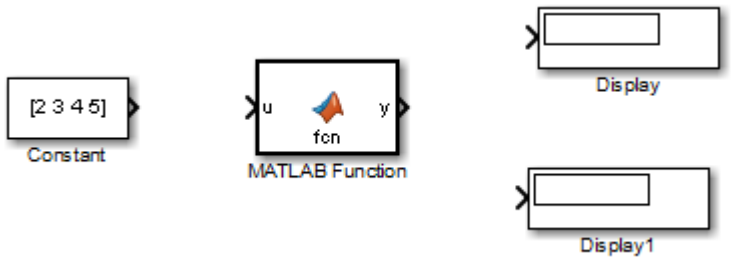
This example shows how to create a model that uses the MATLAB Function block to calculate the mean and standard deviation for a vector of values.

Create Model

1 Create a new Simulink model and insert a MATLAB Function block from the User-Defined Functions library.



2 Add a Constant block and set its value to vector `[2 3 4 5]`. Add two Display blocks to the model. Connect these blocks as shown in the diagram.



3 Save the model as `call_stats_block1`.

Program the MATLAB Function Block

Program the block to calculate the mean and standard deviation for a vector of values:

1 Double-click the MATLAB Function block. A default function signature appears in the MATLAB Function Block Editor. Write any code inside the defined function signatures.

2 Edit the function header line:

```
function [mean,stdev] = stats(vals)
```

From this code, you define a function called `stats`, which calculates a statistical mean and standard deviation for the values in the vector `vals`. The function header declares `vals` as an argument to the `stats` function, with `mean` and `stdev` as return values.

3 In the MATLAB Function Block Editor, enter a line space after the function header and add the following code:

```
% Calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

4 Save the model as `call_stats_block2`.

Build the Function and Check for Errors

After programming the block in a Simulink model, you can build the function and test for errors. Building your MATLAB Function block requires a supported compiler. MATLAB automatically selects one as the default compiler. If you have multiple MATLAB-supported compilers installed on your system, you can change the default compiler using the `mex -setup` command. See “Change Default Compiler”.

Supported Compilers for Simulation and Code Generation Builds

View a list of compilers for building models containing MATLAB Function blocks simulation and code generation.

1 Navigate to the Supported and Compatible Compilers page and select your platform.

2 Scroll to the table under Simulink Product Family.

3 To check the table for models that contain MATLAB Function blocks for simulation, find the compilers checked in the column titled `Simulink For Model Referencing, Accelerator mode, Rapid Accelerator mode, and MATLAB Function blocks`.

To check the table for models that contain MATLAB Function blocks and generate code, find the compilers checked in the column titled `Simulink Coder` .

Supported Compilers for Code Generation

To generate code for models that contain MATLAB Function blocks, you can use any of the C compilers supported by Simulink software for code generation with Simulink Coder. For a list of these compilers:

1 Navigate to the Supported and Compatible Compilers Web page.

2 Select your platform.

3 In the table for Simulink and related products, find the compilers checked in the column titled Simulink Coder.

Locate and Fix Errors

If errors occur during the build process, the **Diagnostics Viewer** window lists the errors with links to the offending code.

The following exercise shows the way to locate and fix an error in a MATLAB Function block.

1. In the `stats` function, change the local function `avg` to a fictitious local function `aug` and then compile again to see the following messages in window. The **Diagnostics Viewer** window displays each detected error with a shaded red line.
2. Investigate the error titled `Undefined function or variable 'aug'`. In the diagnostic message for the selected error, click the blue link after the function name to display the offending code. The offending line appears highlighted in the `MATLAB Function Block Editor`.
3. The message also links to a report about compile-time type information for variables and expressions in your MATLAB functions. This information helps you diagnose error messages and understand type propagation rules. For more information about the report, see “MATLAB Function Reports” on page 44-41. To see the report, click the highlighted blue link in the line called `Launch diagnostic report`
4. Correct the error by changing `aug` back to `avg` and recompile.

Define Inputs and Outputs

By default, function inputs and outputs inherit their data type and size from the signals attached to their ports. Examine input and output data for the MATLAB Function block to verify that it inherits the correct type and size.

1. Double-click the MATLAB Function block `stats`.
2. In the MATLAB Function Block Editor, select **Edit Data**. The `Ports and Data Manager` opens to help you define arguments for MATLAB Function blocks.

   The left pane displays the argument `vals` and the return values `mean` and `stdev` that you have already created for the MATLAB Function block. Observe that `vals` is assigned a **Scope** of `Input`, which is short for **Input from Simulink**. `mean` and `stdev` are assigned the **Scope** of `Output`, which is short for **Output to Simulink**.
3. In the left pane of the Ports and Data Manager, click anywhere in the row for `vals` to highlight it.

   The right pane displays the **Data** properties dialog box for `vals`. By default, the class, size, units, and complexity of input and output arguments are inherited from the signals attached to each input or output port. Inheritance is specified by setting **Size** to `-1`**, Complexity** to `Inherited`, and **Type** to `Inherit: Same as Simulink`.

   The actual inherited values for size and type are set during model compilation, and are reported in the **Compiled Type** and **Compiled Size** columns of the left pane.

   You can specify the type of input or output argument in the **Type** field of the **Data** properties dialog box, for example, `double`. You can also specify the size of an input or output argument by entering an expression in the **Size** field. For example, you can enter `[2 3]` in the **Size** field to specify `vals` as a `2-by-3` matrix. See “Type Function Arguments” on page 44-45 and “Size Function Arguments” on page 44-51 for more information on the expressions that you can enter for type and size.

Note The default first index for any arrays that you add to a MATLAB Function block function is 1, just as it would be in MATLAB.

Create a MATLAB Function Object and Query Properties

You can create an object for the MATLAB Function block in your model, and modify the properties that belong to this model. To query the properties in `call_stats_block2` model you just created, create a configuration object.

```
myconfig = get_param('call_stats_block2/MATLAB Function', 'MATLABFunctionConfiguration')
```

```
myconfig =

  MATLABFunctionConfiguration with properties:

                        Path: 'call_stats_block2/MATLAB Function'
              FunctionScript: 'function [mean,stdev] = stats(vals)↵↵len = length(vals);↵mean = a
                UpdateMethod: Inherited
                  SampleTime: '-1'
                 Description: ''
                DocumentLink: ''
         SupportVariableSizing: 1
        AllowDirectFeedthrough: 1
    SaturateOnIntegerOverflow: 1
                   TreatAsFi: FixedPoint
                  FimathMode: SameAsMATLAB
                      Fimath: 'fimath('RoundingMethod','Nearest','OverflowAction','Saturate','Pr
```

To change any of the properties in your configuration object, use the dot notation with your object name. For example, to change the description to the MATLAB Function block in this model:

```
myconfig.Description = 'This model outputs the mean and standard deviation values of an array'
```

To learn more about the properties you can modify in your MATLAB Function configuration object, see `MATLABFunctionConfiguration`.

Define Local Variables for Code Generation

To generate code from the MATLAB algorithm in a MATLAB Function block, you must explicitly assign the class, size, and complexity of local variables before using them in operations or returning them as outputs (see “Data Definition for Code Generation” on page 52-2). In the example function `stats`, the local variable `len` is defined before being used to calculate mean and standard deviation:

```
len = length(vals);
```

Once you assign properties to a variable, you cannot redefine its class, size, or complexity elsewhere in the function body with some exceptions (see “Reassignment of Variable Properties” on page 51-8).

Generate Code for the MATLAB Function Block

1. Open the `call_stats_block2` model that you saved at the end of “Program the MATLAB Function Block” on page 44-6.
2. Double-click `stats` block.

3 Select **Build Model > Build** to compile and build the example model.

If you get an error related to the `Variable-step` solver, from **Configuration Parameters > Solver**, change the solver type to a `Fixed-step` solver and rerun the build. To learn more about the differences between fixed-step and variable-step solvers, see “Fixed-Step Versus Variable-Step Solvers” on page 3-6.

If no errors occur, the **Diagnostics Viewer** window displays a message indicating success. Otherwise, this window helps you locate errors, as described in “Locate and Fix Errors” on page 44-8.

Add Code to a MATLAB Function Block Programmatically

This example shows how to programmatically add a MATLAB Function block to a model and populate the block with MATLAB code. If you already have MATLAB code and do not want to add it to a MATLAB Function block manually, this workflow can be convenient.

1 Create and save a model called `myModel`.

2 Create a MATLAB function with the following code and save it in `myAdd.m`.

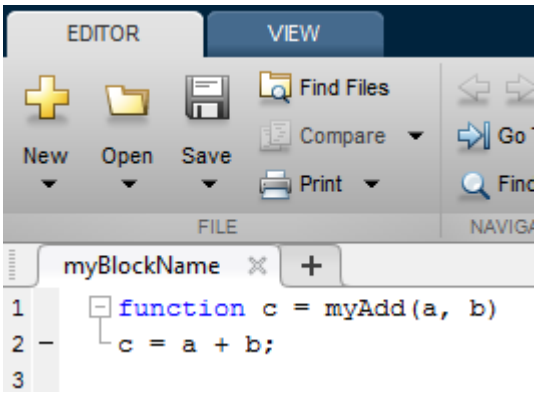
```
function c = myAdd(a, b)
c = a + b;
```

3 Write a MATLAB script that adds a MATLAB Function block to `myModel` and populates it with the contents of `myAdd.m`.

```
% Add a MATLAB Function block to a model and populate the block with MATLAB
% code.
%
% Copyright 2018 The Mathworks, Inc.

open_system('myModel.slx');
libraryBlockPath = 'simulink/User-Defined Functions/MATLAB Function';
newBlockPath = 'myModel/myBlockName';
% Add a MATLAB Function to the model
add_block(libraryBlockPath, newBlockPath);
% In memory, open models and their parts are represented by a hierarchy of
% objects. The root object is slroot. This line of the script returns the
% object that represents the new MATLAB Function block:
blockHandle = find(slroot, '-isa', 'Stateflow.EMChart', 'Path', newBlockPath);
% The Script property of the object contains the contents of the block,
% represented as a character vector. This line of the script loads the
% contents of the file myAdd.m into the Script property:
blockHandle.Script = fileread('myAdd.m');
% Alternatively, you can specify the code directly in a character vector.
% For example:
% blockHandle.Script = 'function c = fcn (a, b)';
```

4 Run the script and observe the new MATLAB Function block in `myModel`.

5 To see the code that you added to the block, double-click the `myBlockName` block.



See Also

`MATLAB Function` | `add_block`

More About

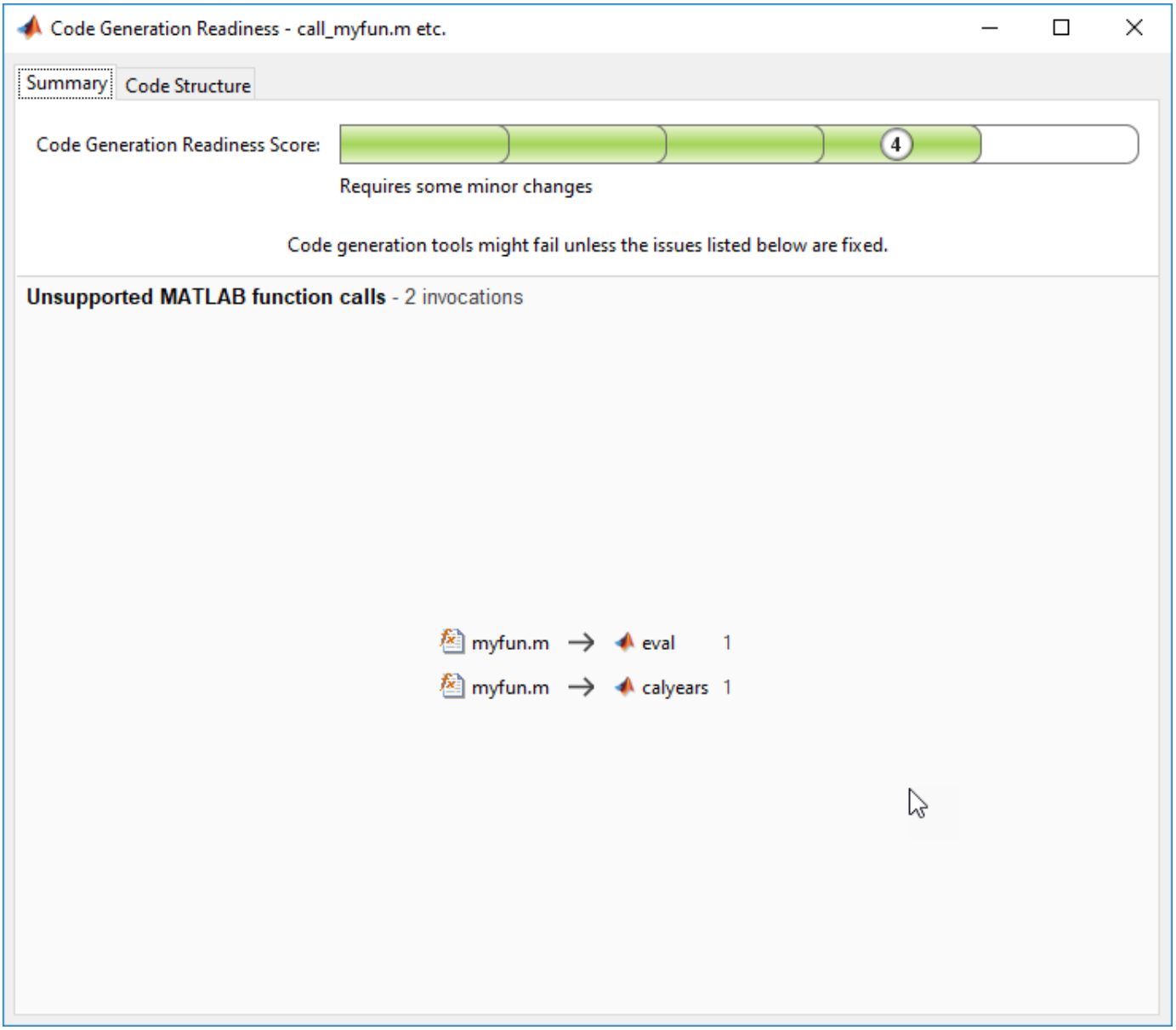
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Ports and Data Manager” on page 44-29
- “Track Object Using MATLAB Code” on page 44-134

Code Generation Readiness Tool

The code generation readiness tool screens MATLAB code for features and functions that code generation does not support. The tool provides a report that lists the source files that contain unsupported features and functions. The report also indicates the amount of work required to make the MATLAB code suitable for code generation. It is possible that the tool does not detect all code generation issues. Under certain circumstances, it is possible that the tool can report false errors. Therefore, before you generate C code, verify that your code is suitable for code generation by generating a MEX function.

The code generation readiness tool does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions” on page 64-8.

Summary Tab



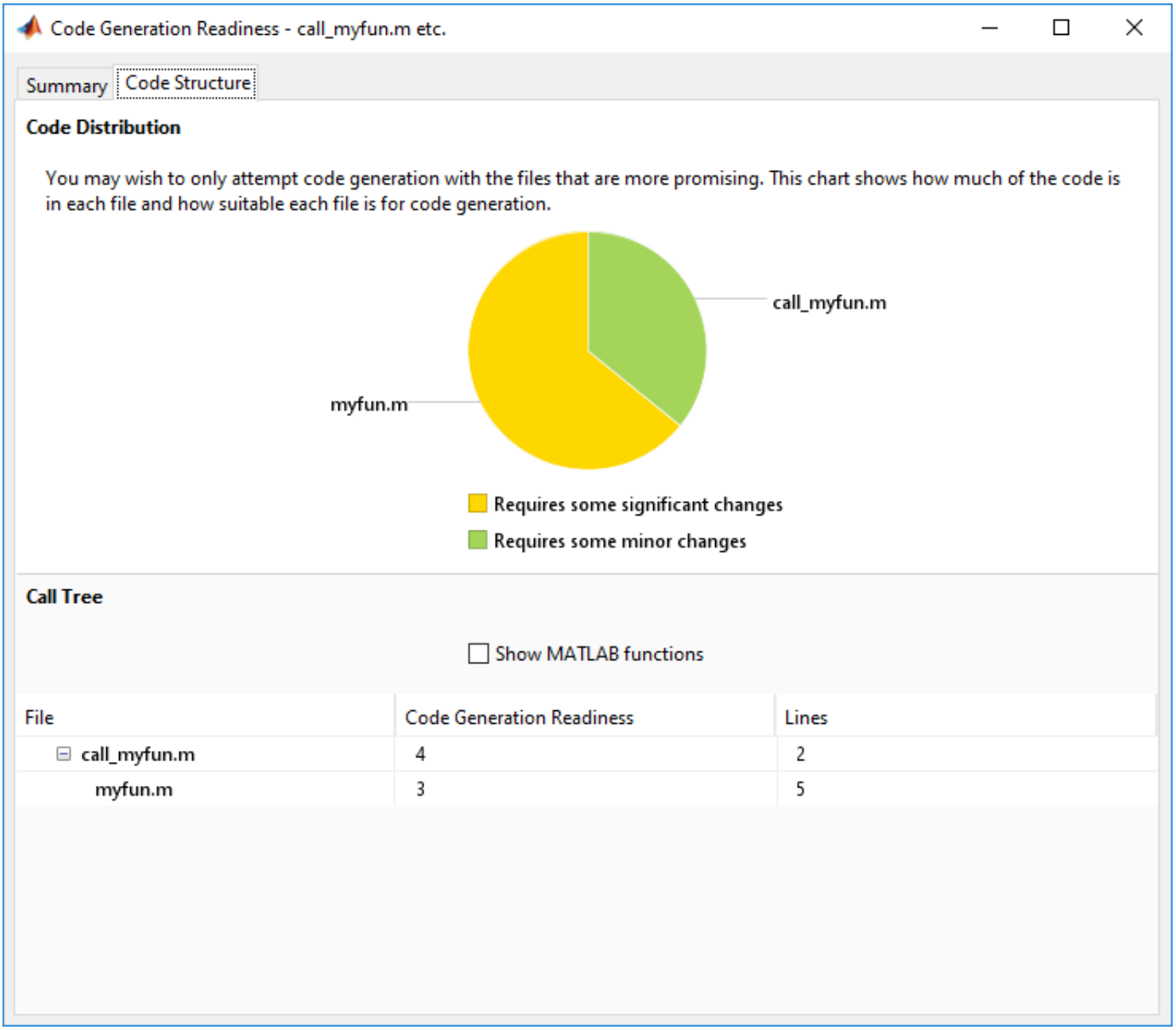
The **Summary** tab provides a **Code Generation Readiness Score**, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues; the code is ready to use with minimal or no changes.

On this tab, the tool also displays information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. To learn more about the issues and how to fix them, use the Code Analyzer.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features.

- Unsupported data types.

Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab displays information about the relative size of each file and how suitable each file is for code generation.

Code Distribution

The **Code Distribution** pane displays a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. During the planning phase of a project, you can use this

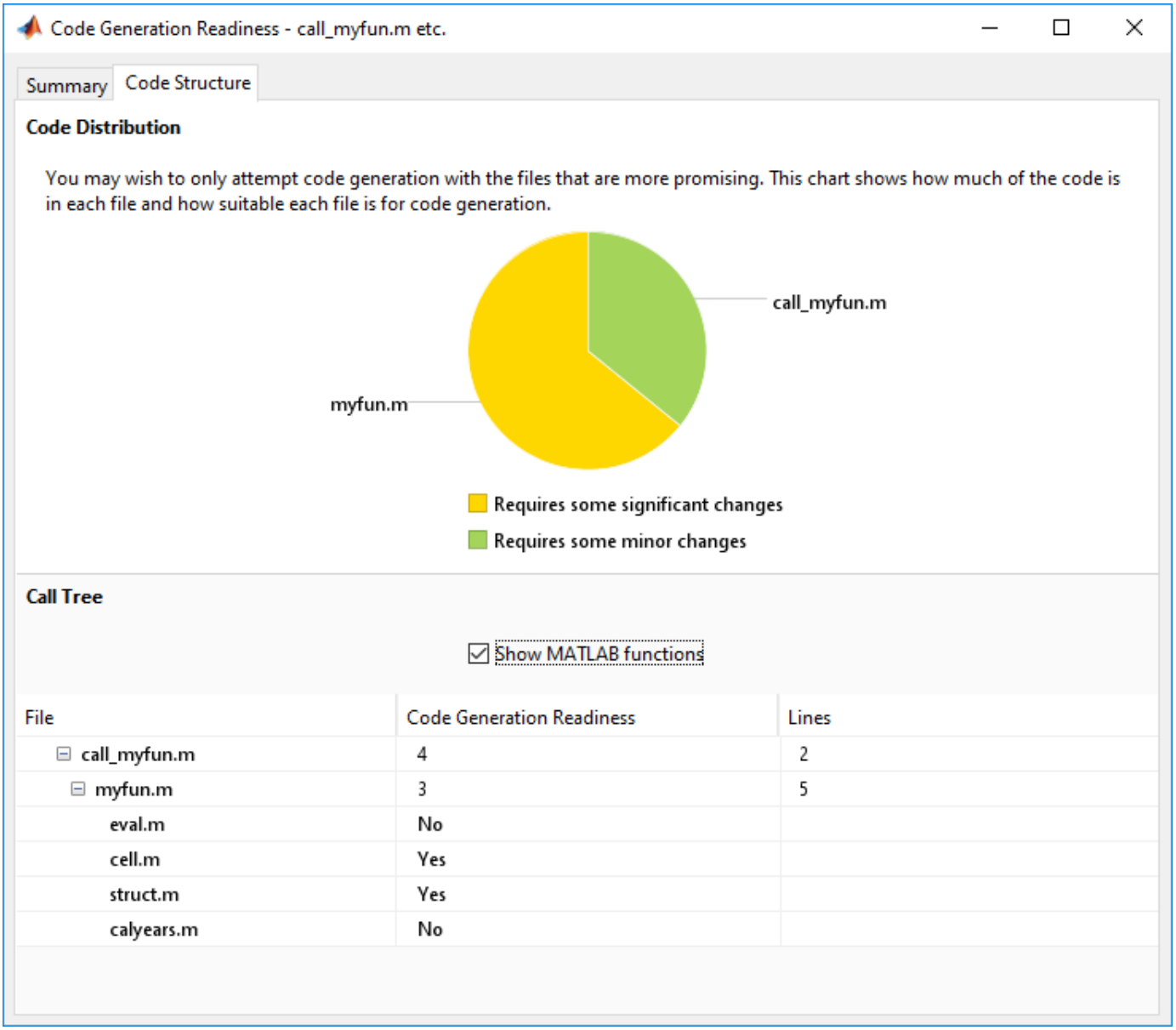
information for estimation and scheduling. If the report indicates that multiple files are not suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane displays information about the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score, which ranges from `1` to `5`. A score of `1` indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of `5` indicates that the tool does not detect code generation issues. The code is ready to use with minimal or no changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions

If you select **Show MATLAB Functions**, the report also lists the MATLAB functions that your function calls. For each of these MATLAB functions, if code generation supports the function, the report sets **Code Generation Readiness** to `Yes`.



See Also

Related Examples

- “Check Code Using the Code Generation Readiness Tool” on page 44-17

Check Code Using the Code Generation Readiness Tool

In this section...

Run Code Generation Readiness Tool at the Command Line

1. Navigate to the folder that contains the file that you want to check for code generation readiness.
2. At the MATLAB command prompt, enter:

   ```
   coder.screener('filename')
   ```

   The **Code Generation Readiness** tool opens for the file named `filename`, provides a code generation readiness score, and lists issues that must be fixed prior to code generation.

Run the Code Generation Readiness Tool From the Current Folder Browser

1. In the current folder browser, right-click the file that you want to check for code generation readiness.
2. From the context menu, select `Check Code Generation Readiness`.

   The **Code Generation Readiness** tool opens for the selected file and provides a code generation readiness score and lists issues that must be fixed prior to code generation.

See Also

More About

- “Code Generation Readiness Tool” on page 44-12

Debugging a MATLAB Function Block

In this section...

Debugging the Function in Simulation

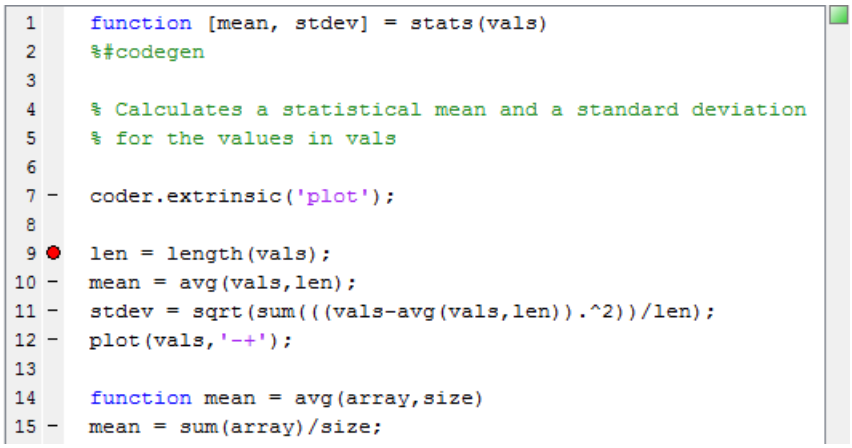
In “Create Custom Functionality Using MATLAB Function Block” on page 44-6, you created an example model with a MATLAB Function block that calculates the mean and standard deviation for a set of input values. The software enables debugging for a MATLAB Function when you set a breakpoint.

To debug the MATLAB Function in this model:

1. Open the `call_stats_block2` model and double-click the MATLAB Function block `stats` to open the editor.
2. In the MATLAB Function Block Editor, click the dash (`-`) in the left margin of the line:

   ```
   len = length(vals);
   ```

   A red dot appears in the line margin, indicating the breakpoint.



3. Simulate the model.

   Simulation pauses when execution reaches the breakpoint. This is indicated by a green arrow in the margin.

```
1    function [mean, stdev] = stats(vals)
2    %#codegen
3
4    % Calculates a statistical mean and a standard deviation
5    % for the values in vals
6
7 -  coder.extrinsic('plot');
8
9    len = length(vals);
10 -  mean = avg(vals,len);
11 -  stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
12 -  plot(vals,'-+');
13
14   function mean = avg(array,size)
15 -  mean = sum(array)/size;
```

4 In the toolbar, click **Step** to advance execution.

The execution arrow advances to the next line of `stats`, which calls the local function `avg`.

5 In the toolbar, click **Step In**.

Execution advances to enter the local function `avg`. Once you are in a local function, you can use the **Step** or **Step In** commands to advance execution. If the local function calls another local function, use **Step In** to enter it. If you want to execute the remaining lines of the local function, use **Step Out**.

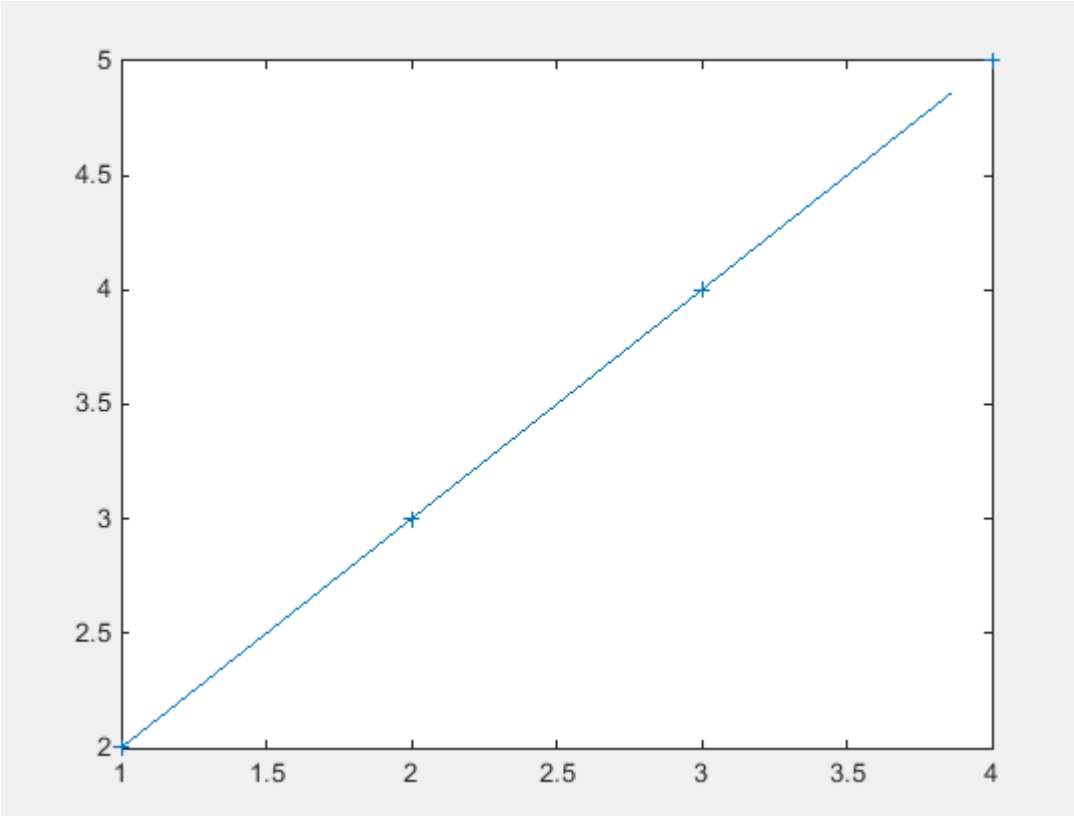
6 Click **Step** to execute the only line in the local function `avg`. When the local function `avg` finishes executing, a green arrow, pointing down, appears under the last line of the function.

7 Click **Step** to return to the function `stats`.

Execution advances to the line after the call to the local function `avg`.

8 Click **Step** twice to calculate the `stdev` and to execute the `plot` function.

The plot function executes in MATLAB:



In the MATLAB Function Block Editor, a green arrow points down under the last line of code, indicating the completion of the function `stats`.

9 Click **Continue** to continue execution of the model.

The computed values of `mean` and `stdev` appear in the Display blocks.

10 In the MATLAB Function Block Editor, click **Quit Debugging** to stop simulation.

Set Conditions on Breakpoints

To help you debug code, you can enter a MATLAB expression as a condition on a breakpoint inside a MATLAB Function block. Simulation then pauses on that breakpoint only when the condition is true. To set a conditional breakpoint, in the MATLAB Function block editor, right-click beside the line of code and select **Set Conditional Breakpoint**. Type the condition in the pop-up window. You can use any valid MATLAB expression as a condition. This condition expression can include numerical values and any data that is in scope at the breakpoint.

To add or modify a condition on an existing breakpoint, right-click the breakpoint and select **Set/Modify Condition**. You can also perform these actions from the **Breakpoints** menu.

Watching Function Variables During Simulation

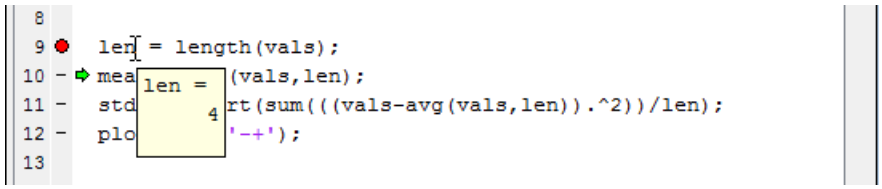
While you simulate a MATLAB Function block, you can use several tools to keep track of variable values in the function.

Watching with the Interactive Display

To display the value of a variable in the function of a MATLAB Function block during simulation:

1. In the MATLAB Function Block Editor, place the mouse cursor over the variable text and observe the pop-up display.

   For example, to watch the variable `len` during simulation, place the mouse cursor over the text `len` in the code. The value of `len` appears adjacent to the cursor, as shown:



Watching with the Command Line Debugger

You can report the values for a function variable with the Command Line Debugger utility in the MATLAB window during simulation. When you reach a breakpoint, the Command Line Debugger prompt, `debug>>`, appears. At this prompt, you can see the value of a variable defined for the MATLAB Function block by entering its name:

```
debug>> stdev

    1.1180

debug>>
```

The Command Line Debugger also provides the following commands during simulation:

Command	Description
`ctrl-c`	Quit debugging and terminate simulation.
`dbcont`	Continue execution to next breakpoint.
`dbquit`	Quit debugging and terminate simulation.
`dbstep [in|out]`	Advance to next program step after a breakpoint is encountered. Step over or step into/out of a MATLAB local function.
`help`	Display help for command line debugging.
`print <var>`	Display the value of the variable `var` in the current scope. If `var` is a vector or matrix, you can also index into `var`. For example, `var(1,2)`.
`save`	Saves all variables in the current scope to the specified file. Follows the syntax of the MATLAB `save` command. To retrieve variables to the MATLAB base workspace, use `load` command after simulation has been ended.
`<var>`	Equivalent to "print <var>" if variable is in the current scope.
`who`	Display the variables in the current scope.
`whos`	Display the size and class (type) of all variables in the current scope.

You can issue any other MATLAB command at the `debug>>` prompt, but the results are executed in the workspace of the MATLAB Function block. To issue a command in the MATLAB base workspace at the `debug>>` prompt, use the `evalin` command with the first argument `'base'` followed by the

second argument command, for example, `evalin('base','whos')`. To return to the MATLAB base workspace, use the `dbquit` command.

Watching with MATLAB

You can display the execution result of a MATLAB Function block line by omitting the terminating semicolon. If you do, execution results for the line are echoed to the MATLAB window during simulation.

Display Size Limits

The MATLAB Function Block Editor does not display the contents of matrices that have more than two dimensions or more than `200` elements. For matrices that exceed these limits, the MATLAB Function Block Editor displays the shape and base type only.

Checking for Data Range Violations

MATLAB Function blocks check inputs and outputs for data range violations when the input or output values enter or leave the blocks. To enable data range violation checking, set **Simulation range checking** in the **Diagnostics: Data Validity** pane of the Configuration Parameters dialog box to `error`.

Specifying a Range

To specify a range for input and output data, follow these steps:

1. In the Ports and Data Manager, select the input or output of interest.

   The data properties dialog box opens.
2. In the data properties dialog box, select the General tab and enter a limit range, as described in “Setting General Properties” on page 44-35.

Debugging Tools

Use the following tools during a MATLAB Function block debugging session:

Tool Button	Description	Shortcut Key
 **Build**	Access this tool from the **Editor** tab by selecting **Build Model > Build**. Check for errors and build a simulation application (if no errors are found) for the model containing this MATLAB Function block.	**Ctrl+B**
 **Update Diagram**	Access this tool from the **Editor** tab by selecting **Build Model > Update Diagram**. Check for errors based on the latest changes you make to the MATLAB Function block.	**Ctrl+D**

Tool Button	Description	Shortcut Key
 **Update Ports**	Access this tool from the **Editor** tab by selecting **Build Model > Update Ports**. Updates the ports of the MATLAB Function block with the latest changes made to the function argument and return values without closing the MATLAB Function Block Editor.	**Ctrl+Shift+A**
 **Run Model**	Start simulation of the model containing the MATLAB Function block. If execution is paused at a breakpoint, continues debugging.	**F5**
 **Stop Model**	Stop simulation of the model containing the MATLAB Function block. Alternatively, from the **Editor** tab, select **Quit Debugging** if execution is paused at a breakpoint.	**Shift+F5**
 **Set/Clear**	Access this tool by selecting **Breakpoints > Set/Clear**. Set a new breakpoint or clear an existing breakpoint for the selected line of code in the MATLAB Function block. The presence of the text cursor or highlighted text selects the line. A breakpoint indicator appears on the selected line. Alternatively, click the hyphen character (-) next to the line number. A breakpoint indicator appears in place of the hyphen. Click the breakpoint indicator to clear the breakpoint.	**F12**
 **Enable/Disable**	Access this tool by selecting **Breakpoints > Enable/ Disable**. Enable or disable an existing breakpoint for the selected line of code in the MATLAB Function block. If the breakpoint is disabled, an indicator appears on the selected line.	**None**
 **Set Condition**	Access this tool by selecting **Breakpoints > Set Condition**. Set a condition on the breakpoint for the selected line of code in the MATLAB Function block. If the breakpoint has a condition associated with it, an indicator appears on the selected line.	
 **Clear All**	Access this tool by selecting **Breakpoints > Clear All**. Clear all existing breakpoints in the MATLAB Function block code.	**None**

Tool Button	Description	Shortcut Key
 **Step**	Step through the execution of the next line of code in the MATLAB Function block. This tool steps past function calls and does not enter called functions for line-by-line execution. You can use this tool only after execution has stopped at a breakpoint.	**F10**
 **Step In**	Step through the execution of the next line of code in the MATLAB Function block. If the line calls a local function, step into the first line of the local function. You can use this tool only after execution has stopped at a breakpoint.	**F11**
 **Step Out**	Step out of line-by-line execution of the current function or local function. If in a local function, the debugger continues to the line following the call to this local function. You can use this tool only after execution has stopped at a breakpoint.	**Shift+F11**
 **Continue**	Continue debugging after a pause, such as stopping at a breakpoint. You can use this tool only after execution has stopped at a breakpoint.	**F5**
 **Quit Debugging**	Exit debug mode. You can use this tool only after execution has stopped at a breakpoint.	**Shift+F5**

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26
- “MATLAB Function Reports” on page 44-41

Prevent Algebraic Loop Errors in MATLAB Function and Stateflow Blocks

You can use Stateflow charts, MATLAB Function blocks, and Stateflow Truth Tables in feedback loops in your model. You can also use these blocks with synchronous subsystems enabled by the State Control block. To prevent algebraic loop or synchronous semantic errors, apply these restrictions.

Simulink Block	Restrictions
Stateflow Chart	Use Moore charts to prevent an algebraic loop. In the Property Inspector, set the **State Machine Type** to `Moore`. Moore charts prevent algebraic loops by ensuring that outputs depend only on current state.
MATLAB Function block	Nondirect feedthrough semantics prevent algebraic loop errors by ensuring that outputs depend only on current state. To enable these semantics, clear the **Allow direct feedthrough** property check box. If your block uses direct feedthrough, do not: • Call imported functions. • Define output function-call events. • Define or use persistent variables. When you apply these restrictions, you allow the Simulink solver to try to solve the algebraic loop.
Truth Table	Do not: • Call imported functions. • Define local or output function-call events. • Define local or data store memory data. • Define or use persistent variables. • Use machine-parented data or events. When you apply these restrictions, you allow the Simulink solver to try to solve the algebraic loop.

See Also

Chart | Truth Table

More About

- “Design Considerations for Moore Charts” (Stateflow)
- “Use Nondirect Feedthrough in a MATLAB Function Block” on page 44-178
- “Algebraic Loop Concepts” on page 3-27
- “Control States in Charts Enabled by Function-Call Input Events” (Stateflow)

MATLAB Function Block Editor

In this section...

Customizing the MATLAB Function Block Editor

Use the toolbar icons to customize the appearance of the MATLAB Function Block Editor in the same manner as the MATLAB editor. See “Basic Settings”.

MATLAB Function Block Editor Tools

Use the following tools to work with the MATLAB Function block:

Tool Button	Description
 **Edit Data**	Opens the Ports and Data Manager dialog to add or modify arguments for the current MATLAB Function block. To learn more, see “Ports and Data Manager” on page 44-29.
 **View Report**	Opens the MATLAB Function report for the MATLAB Function block. For more information, see “MATLAB Function Reports” on page 44-41.
 **Simulation Target**	Opens the **Simulation Target** pane in the **Configuration Parameters** dialog to include custom code.
 **Go To Diagram**	Displays the MATLAB function in its native diagram without closing the editor.

See “Define Inputs and Outputs” on page 44-8 for an example of defining an input argument for a MATLAB Function block.

Editing and Debugging MATLAB Function Block Code

Manual Indenting

To indent a block of code manually:

1. Highlight the text that you would like to indent.
2. Select one of the Indent tools on the Editor tab:

Tool	Description
	Applies smart indenting to selected text.

Tool	Description
	Move selected text right one indent level.
	Move selected text left one indent level.

Opening a Selection

You can open a local function, function, file, or variable from within a file in the MATLAB Function Block Editor.

To open a selection:

1. Position the cursor in the name of the item you would like to open.
2. Right-click and select **Open <selection>** from the context menu.

The Editor chooses the appropriate tool to open the selection. For more information, refer to “Manage Files and Folders”.

Note If you open a MATLAB Function block input or output parameter, the Ports and Data Manager opens with the selected parameter highlighted. You can use the Ports and Data Manager to modify parameter attributes. For more information, refer to “Ports and Data Manager” on page 44-29.

Evaluating a Selection

You can use the **Evaluate a Selection** menu option to report the value for a MATLAB function variable or equation in the MATLAB window during simulation.

To evaluate a selection:

1. Highlight the variable or equation that you would like to evaluate.
2. Hold the mouse over the highlighted text and then right-click and select **Evaluate Selection** from the context menu. (Alternatively, select **Evaluate Selection** from the **Text** menu).

When you reach a breakpoint, the MATLAB command Window displays the value of the variable or equation at the Command Line Debugger prompt.

```
debug>> stdev

    1.1180

debug>>
```

Note You cannot evaluate a selection while MATLAB is busy, for example, running a MATLAB file.

Setting Data Scope

To set the data scope of a MATLAB Function block input parameter:

1. Highlight the input parameter that you would like to modify.
2. Hold the mouse over the highlighted text and then right-click and select **Data Scope for <selection>** from the context menu.

3 Select:

- **Input** if your input data is provided by the Simulink model via an input port to the MATLAB Function block.
- **Parameter** if your input is a variable of the same name in the MATLAB or model workspace or in the workspace of a masked subsystem containing this block.

For more information, refer to “Setting General Properties” on page 44-35.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Ports and Data Manager” on page 44-29

Ports and Data Manager

The Ports and Data Manager provides a convenient method for defining objects and modifying their properties in a MATLAB Function block.

The Ports and Data Manager provides the same data definition capabilities for individual MATLAB Function blocks as the Model Explorer provides across the model hierarchy (see **Model Explorer**).

Ports and Data Manager Dialog Box

The Ports and Data Manager dialog box allows you to add and define data arguments, input triggers, and function call outputs for MATLAB Function blocks. Using this dialog, you can also modify properties for the MATLAB Function block and the objects it contains.

The dialog box consists of two panes:

- The **Contents** (left) pane lists the objects that have been defined for the MATLAB Function block.
- The **Dialog** (right) pane displays fields for modifying the properties of the selected object.

Properties vary according to the scope and type of the object. Therefore, the Ports and Data Manager properties dialogs are dynamic, displaying only the property fields that are relevant for the object you add or modify.

When you first open the dialog box, it displays the properties of the MATLAB Function block.

Opening the Ports and Data Manager

To open the Ports and Data Manager from the MATLAB Function Block Editor, select **Edit Data** on the Editor tab. The Ports and Data Manager appears for the MATLAB Function block that is open and has focus.

Ports and Data Manager Tools

The following tools are specific to the Ports and Data Manager:

Tool Button	Description
 **Go to Block Editor**	Displays the MATLAB function in the MATLAB Function Block Editor.
 **Show Block Dialog**	Displays the default MATLAB function properties. To learn more, see “MATLAB Function Block Properties” on page 44-38. Use this button to return to the settings used by the block after viewing data associated with the block arguments.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “MATLAB Function Block Editor” on page 44-26
- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4

Adding Function Call Outputs to a MATLAB Function Block

A function call output is an event on the output port of a MATLAB Function block that causes a Function-Call Subsystem block in the Simulink model to execute. Another block can invoke a function-call subsystem directly during a simulation. See “Using Function-Call Subsystems” on page 10-34.

Use the Ports and Data Manager to add and modify function call outputs to a MATLAB Function block that is open and has focus. To add a function call output and modify its properties, follow these steps:

1. In the Ports and Data Manager, select **Add** > **Function Call Output**.

   The Ports and Data Manager adds a default definition of the new function call output to the MATLAB Function block and displays the Function Call properties dialog.
2. Modify function call output properties.
3. Return to the MATLAB Function block properties at any time by selecting **Tools** > **Block Dialog**.

Considerations when Supplying Output to the Function-Call Subsystem

If a MATLAB Function block triggers a function-call subsystem, and supplies an output signal to the same function-call subsystem, the signal to the function-call subsystem can effectively be delayed by one time step compared to the function call. At the moment of the function call, the function-call subsystem sees the previous MATLAB Function block output port value even if the output data has been updated within the block MATLAB code.

The Function Call Properties Dialog

The Function Call properties dialog in the Ports and Data Manager allows you to edit the properties of function call outputs in MATLAB Function blocks.

To open the Function Call properties dialog, select a function call output in the Contents pane.

Setting Function Call Output Properties

You can set the following properties in the Function Call properties dialog:

Property	Description
Name	Name of the function call output, following the same naming conventions used in MATLAB.
Port	Index of the port associated with the function call output. Function call output ports are numbered sequentially after input and output ports.
Description	Description of the function call output.
Document link	Link to documentation for the function call output. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click **Document link** displayed at the bottom of the **Function Call** properties dialog, the MATLAB Function block evaluates the link and displays the documentation.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “MATLAB Function Block Editor” on page 44-26
- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4

Adding Input Triggers to a MATLAB Function Block

An input trigger is an event on the input port that causes the MATLAB Function block to execute. See “Using Triggered Subsystems” on page 10-17.

You can define the following types of triggers in MATLAB Function blocks:

- Rising
- Falling
- Either (rising or falling)
- Function call

For a description of each trigger type, see “Setting Input Trigger Properties” on page 44-33.

Use the Ports and Data Manager to add input triggers to a MATLAB Function block that is open and has focus. To add an input trigger and modify its properties, follow these steps:

1. In the Ports and Data Manager, select **Add > Input Trigger**.

   The Ports and Data Manager adds a default definition of the new input trigger to the MATLAB Function block and displays the Trigger properties dialog.
2. Modify trigger properties.
3. Return to the MATLAB Function block properties at any time by selecting **Tools > Block Dialog**.

The Trigger Properties Dialog

The Trigger properties dialog in the Ports and Data Manager allows you to set and modify the properties of input triggers in MATLAB Function blocks.

To open the Trigger properties dialog, select an input trigger in the Contents pane.

Setting Input Trigger Properties

You can set the following properties in the Trigger properties dialog:

Property	Description
Name	Name of the input trigger, following the same naming conventions used in MATLAB.
Port	Index of the port associated with the input trigger. The default value is 1.

Property	Description
Trigger	Type of event that triggers execution of the MATLAB Function block. You can select one of the following types of triggers: • `Rising` (default) — Triggers execution of the MATLAB Function block when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative). • `Falling`— Triggers execution of the MATLAB Function block when the control signal falls from a positive or zero value to a negative value (or zero if the initial value is positive). • `Either`— Triggers execution of the MATLAB Function block when the control signal is either rising or falling. • `Function call`— Triggers execution of the MATLAB Function block from a block that outputs function-call events, or from an S-function
Description	Description of the input trigger.
Document link	Link to documentation for the input trigger. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click the blue text that reads **Document link** displayed at the bottom of the **Trigger** properties dialog, the MATLAB Function block evaluates the link and displays the documentation.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “MATLAB Function Block Editor” on page 44-26
- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4

Adding Data to a MATLAB Function Block

You can define data arguments for MATLAB Function blocks using the following methods:

Method	For Defining	Reference
Define data directly in the MATLAB Function block code	Input and output data	See “Define Inputs and Outputs” on page 44-8.
Use the Ports and Data Manager	Input, output, and parameter data in the MATLAB Function block that is open and has focus	See “Defining Data in the Ports and Data Manager” on page 44-35.
Use the Model Explorer	Input, output, and parameter data in MATLAB Function blocks at all levels of the model hierarchy	See **Model Explorer**

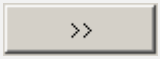
Defining Data in the Ports and Data Manager

To add a data argument, in the Ports and Data Manager, select **Add** > **Data** and modify the data properties.

Setting General Properties

You can set the following properties in the General tab:

Property	Description
Name	Name of the data argument, following the same naming conventions used in MATLAB.
Scope	Where data resides in memory, relative to its parent. Scope determines the range of functionality of the data argument. You can set scope to one of the following values: • `Parameter`— Specifies that the source for this data is a variable of the same name in the MATLAB or model workspace or in the workspace of a masked subsystem containing this block. If a variable of the same name exists in more than one of the workspaces visible to the block, the variable closest to the block in the workspace hierarchy is used (see “Model Workspaces” on page 67-119). • `Input`— Data provided by the model via an input port to the MATLAB Function block. • `Output`— Data provided by the MATLAB Function block via an output port to the model. • `Data Store Memory`— Data provided by a Data Store Memory block in the model (see “Storing Data Using Data Store Memory Blocks” on page 44-92). For more information, see “Define Inputs and Outputs” on page 44-8 and “Add Parameter Arguments” on page 44-54.
Port	Index of the port associated with the data argument. This property applies only to input and output data.

Property	Description
Tunable	Indicates whether the parameter used as the source of this data item is tunable (see "Tunable Parameters" on page 10-69). This property applies only to parameter data. Clear this option if the parameter must be a constant expression, such as for MATLAB toolbox functions supported for code generation (see "Functions and Objects Supported for C/C++ Code Generation" on page 49-2).
Data must resolve to Simulink signal object	Specifies that the data argument must resolve to a Simulink signal object. This property applies only to output data. This property appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`. See "Symbol Resolution" on page 67-127 for more information.
Size	Size of the data argument. Size can be a scalar value or a MATLAB vector of values. Size defaults to –1, which means that it is inherited, as described in "Inheriting Argument Sizes from Simulink" on page 44-51. This property does not apply to Data Store Memory data. For more details, see "Size Function Arguments" on page 44-51.
Variable Size	Indicates whether the size of this data item is variable. This property does not apply to Data Store Memory data.
Complexity	Indicates real or complex data arguments. You can set complexity to one of the following values: • `Off`— Data argument is a real number • `On`— Data argument is a complex number • `Inherited`— Data argument inherits complexity based on its scope. Input and output data inherit complexity from the Simulink signals connected to them; parameter data inherits complexity from the parameter to which it is bound.
Type	• Selecting a built-in type from the **Type** drop down list. • Entering an expression in the **Type** field that evaluates to a data type (see "About Data Types in Simulink" on page 67-2). • Using the Data Type Assistant to specify a data **Mode**, then specifying the data type based on that mode. **Note** To display the Data Type Assistant, click the Show data type assistant button:  For more information, see "Specifying Argument Types" on page 44-45.
Unit (e.g., m, m/s^2, N*m)	Specify physical units for input and output data. By default, the property is set to inherit the unit from the Simulink signal on the corresponding input or output port. See "Units in MATLAB Function Blocks" on page 44-53.

Property	Description
Limit range	Specify the range of acceptable values for input or output data. The MATLAB Function block uses this range to validate the input or output as it enters or leaves the block. You can enter an expression or parameter that evaluates to a numeric scalar value. • **Minimum** — The smallest value allowed for the data item during simulation. The default value is `-inf`. • **Maximum** — The largest value allowed for the data item during simulation. The default value is `inf`.

Setting Description Properties

You can set the following properties on the Description tab:

Property	Description
Save final value to base workspace	The MATLAB Function block assigns the value of the data argument to a variable of the same name in the MATLAB base workspace at the end of simulation.
Description	Description of the data argument.
Document link	Link to documentation for the data argument. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click the blue text, **Document link**, displayed at the bottom of the **Data** properties dialog, the MATLAB Function block evaluates the link and displays the documentation.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “MATLAB Function Block Editor” on page 44-26
- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4

MATLAB Function Block Properties

This section describes each property of a MATLAB Function block.

Name

Name of the MATLAB Function block.

Update method

Method for activating the MATLAB Function block. You can choose from the following update methods:

Update Method	Description
`Inherited` (default)	Input from the Simulink model activates the MATLAB Function block. If you define an input trigger, the MATLAB Function block executes in response to a Simulink signal or function-call event on the trigger port. If you do not define an input trigger, the MATLAB Function block implicitly inherits triggers from the model. These implicit events are the sample times (discrete or continuous) of the signals that provide inputs to the chart. If you define data inputs, the MATLAB Function block samples at the rate of the fastest data input. If you do not define data inputs, the MATLAB Function block samples as defined by its parent subsystem's execution behavior.
`Discrete`	The MATLAB Function block is sampled at the rate you specify as the block's **Sample Time** property. An implicit event is generated at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the model can have different sample times.
`Continuous`	The Simulink software wakes up (samples) the MATLAB Function block at each step in the simulation, as well as at intermediate time points that can be requested by the solver. This method is consistent with the continuous method.

Saturate on integer overflow

Option that determines how the MATLAB Function block handles overflow conditions during integer operations:

Setting	Action When Overflow Occurs
Enabled (default)	Saturates an integer by setting it to the maximum positive or negative value allowed by the word size. Matches MATLAB behavior.
Disabled	In simulation mode, generates a run-time error. For Simulink Coder code generation, the behavior depends on your C language compiler.

Note The **Saturate on integer overflow** option is relevant only for integer arithmetic. It has no effect on fixed-point or double-precision arithmetic.

When you enable **Saturate on integer overflow**, MATLAB adds additional checks during simulation to detect integer overflow or underflow. Therefore, it is more efficient to disable this option if you are sure that integer overflow and underflow will not occur in your MATLAB Function block code.

Note that the code generated by Simulink Coder does *not* check for integer overflow or underflow and, therefore, may produce unpredictable results when **Saturate on integer overflow** is disabled. In this situation, it is recommended that you simulate first to test for overflow and underflow before generating code.

Support variable-size arrays

Specifies that this MATLAB Function block supports input and output data that varies in dimension during simulation. For more information, see “Declare Variable-Size Inputs and Outputs” on page 44-72.

Allow direct feedthrough

Specifies that this MATLAB Function block supports direct feedthrough semantics, so that the output of the block is controlled directly by the value of an input. When you disable **Allow direct feedthrough**, nondirect feedthrough semantics ensure that outputs rely only on the current state of the block. Using nondirect feedthrough enables you to use MATLAB Function blocks in a feedback loop and prevent algebraic loops. For more information, see “Use Nondirect Feedthrough in a MATLAB Function Block” on page 44-178.

Lock Editor

Option for locking the MATLAB Function Block Editor. When enabled, this option prevents users from making changes to the MATLAB Function block.

Treat these inherited Simulink signal types as fi objects

Setting that determines whether to treat inherited fixed-point and integer signals as Fixed-Point Designer `fi` objects (“Ways to Construct fi Objects” (Fixed-Point Designer)).

- When you select `Fixed-point`, the MATLAB Function block treats all fixed-point inputs as Fixed-Point Designer `fi` objects.
- When you select `Fixed-point & Integer`, the MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Designer `fi` objects.

MATLAB Function block fimath

Setting that defines `fimath` properties for the MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the MATLAB Function block.

You can select one of the following options for the **MATLAB Function block fimath**.

Setting	Description
Same as MATLAB	When you select this option, the block uses the same `fimath` properties as the current default fimath. The edit box appears dimmed and displays the current global fimath in read-only form.
Specify other	When you select this option, you can specify your own `fimath` object in the edit box. You can do so in one of two ways: • Constructing the `fimath` object inside the edit box. • Constructing the `fimath` object in the MATLAB or model workspace and then entering its variable name in the edit box. If you use this option and plan to share your model with others, make sure you define the variable in the model workspace. See “Sharing Models with Fixed-Point MATLAB Function Blocks” (Fixed-Point Designer). For more information on `fimath` objects, see “fimath Object Construction” (Fixed-Point Designer).

Description

Description of the MATLAB Function block.

Document link

Link to documentation for the MATLAB Function block. To document a MATLAB Function block, set the **Document link** property to a Web URL address or MATLAB expression that displays documentation in a suitable format (for example, an HTML file or text in the MATLAB Command Window). The MATLAB Function block evaluates the expression when you click the blue **Document link** text.

See Also

MATLAB Function

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “MATLAB Function Block Editor” on page 44-26
- “Debugging a MATLAB Function Block” on page 44-18
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Declare Variable-Size Inputs and Outputs” on page 44-72
- “Use Nondirect Feedthrough in a MATLAB Function Block” on page 44-178

MATLAB Function Reports

In this section...

When you simulate or build a Simulink model that contains MATLAB Function blocks, Simulink generates a report for each MATLAB Function block in your model. Use the report to debug your MATLAB functions and verify that they are suitable for code generation. The report provides type information for the variables and expressions in your functions. This information helps you to find sources of error messages and to understand type propagation rules.

Stateflow produces one report for each Stateflow chart, regardless of the number of MATLAB functions it contains.

If you have identical MATLAB Function blocks in your model, for example, one in a library and one in the model, a single report is generated for the identical blocks.

Opening a MATLAB Function Report

Use one of these methods:

- In the MATLAB Function Block Editor, select **View Report**.
- If compilation errors occur, in the **Diagnostic Viewer** window, select the `report` link.

Error and Warning Messages

View errors and warnings on the **All Messages** tab. To highlight the source code for an error or warning, click the message. It is best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message.

Functions List

In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function. To edit the function, click **Edit in MATLAB**. A function that is in the MATLAB Function block opens in the MATLAB Function Block Editor. Other functions open in the MATLAB Editor.

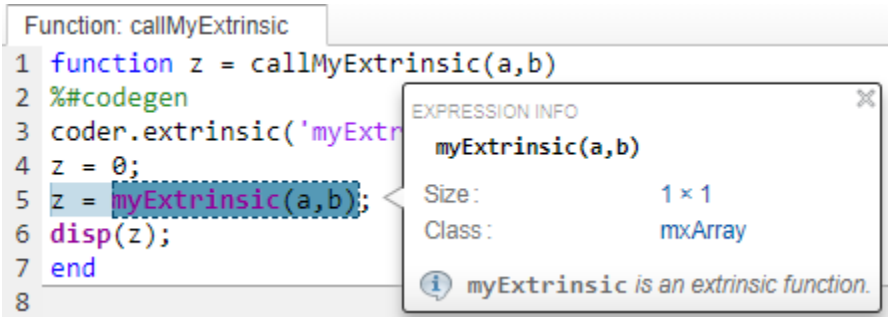
MATLAB Source

To view a MATLAB function in the code pane, click the function in the **MATLAB Source** pane. To see information about the type of a variable or expression, pause over the variable or expression.

In the code pane, syntax highlighting of MATLAB source code helps you to identify MATLAB syntax elements. Syntax highlighting also helps you to identify certain code generation attributes such as whether a function is extrinsic or whether an argument is constant.

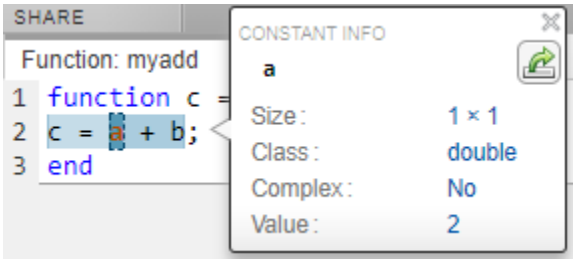
Extrinsic Functions

In the MATLAB code, the report identifies an extrinsic function with purple text. The information window indicates that the function is extrinsic.



Constant Arguments

In the MATLAB code, orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The information window includes the constant value.



Knowing the value of the constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for different constant argument values.

To export the value to a variable in the workspace, click  .

MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types

This information helps you to debug errors, such as type mismatch errors, and to understand type propagation.

Visual Indicators on the Variables Tab

This table describes symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	`{:}`	Heterogeneous cell array (all elements have the same properties)
Name	`{n}`	nth element of a heterogeneous cell array
Class	`v > n`	`v` is reused with a different class, size, and complexity. The number `n` identifies each unique reuse (a reuse with a unique set of properties). When you pause over a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity. See “Reuse the Same Variable with Different Properties” on page 51-9.
Size	`:n`	Variable-size dimension with an upper bound of n
Size	`:?`	Variable-size with no upper bound
Size	italics	Variable-size array whose dimensions do not change size during execution
Class	`sparse` prefix	Sparse array
Class	`complex` prefix	Complex number
Class		Fixed-point type To see the fixed-point properties, click the badge.

Report Limitations

- The variables table does not show individual elements of `varagin` and `vargout`.
- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26
- “Ports and Data Manager” on page 44-29
- “MATLAB Function Block Properties” on page 44-38

Type Function Arguments

In this section...

About Function Arguments

You create function arguments for a MATLAB Function block by entering them in its function header in the MATLAB Function Block Editor. When you define arguments, the Simulink software creates corresponding ports on the MATLAB Function block that you can attach to signals. You can select a data type mode for each argument that you define for a MATLAB Function block. Each data type mode presents its own set of options for selecting a data type.

By default, the data type mode for MATLAB Function block function arguments is **Inherited**. This means that the function argument inherits its data type from the incoming or outgoing signal. To override the default type, you first choose a data type mode and then select a data type based on the mode.

Specifying Argument Types

To specify the type of a MATLAB Function block function argument:

1. From the MATLAB Function Block Editor, select **Edit Data** to open the Ports and Data Manager.
2. In the left pane, select the argument of interest.
3. In the **Data** properties dialog box (right pane), click the Show data type assistant button  to display the Data Type Assistant. Then, choose an option from the **Mode** drop-down menu.

   The **Data** properties dialog box changes dynamically to display additional fields for specifying the data type associated with the mode.
4. Based on the mode you select, specify a desired data type:

Mode	What to Specify
`Inherit` (default)	You cannot specify a value. The data type is inherited from previously-defined data, based on the scope you selected for the MATLAB Function block function argument: • If scope is **Input**, data type is inherited from the input signal on the designated port. • If scope is **Output**, data type is inherited from the output signal on the designated port. • If scope is **Parameter**, data type is inherited from the associated parameter, which can be defined in the Simulink masked subsystem or the MATLAB workspace. See "Inheriting Argument Data Types" on page 44-46.
`Built in`	Select from the drop-down list of supported data types, as described in "Built-In Data Types for Arguments" on page 44-47.
`Fixed point`	Specify the fixed-point data properties as described in "Specifying Fixed-Point Designer Data Properties" on page 44-48.
`Expression`	Enter an expression that evaluates to a data type, as described in "Specifying Argument Types with Expressions" on page 44-47.
`Bus Object`	In the **Bus object** field, enter the name of a `Simulink.Bus` object to define the properties of a MATLAB structure. You must define the bus object in the base workspace. See "How Structure Inputs and Outputs Interface with Bus Signals" on page 44-60. **Note** You can click the **Edit** button to create or modify `Simulink.Bus` objects using the Simulink Bus Editor (see "Attach Bus Signals to MATLAB Function Blocks" on page 44-58.
`Enumerated`	In the Enumerated field, enter the name of a `Simulink.IntEnumType` object that you define in the base workspace. See "Code Generation for Enumerations" on page 44-84.

Inheriting Argument Data Types

MATLAB Function block function arguments can inherit their data types, including fixed point types, from the signals to which they are connected.

1. Select the argument of interest in the Ports and Data Manager
2. In the **Data** properties dialog, select `Inherit: Same as Simulink` from the **Type** drop-down menu.

See "Built-In Data Types for Arguments" on page 44-47 for a list of supported data types.

Note An argument can also inherit its complexity (whether its value is a real or complex number) from the signal that is connected to it. To inherit complexity, set the **Complexity** field on the **Data** properties dialog to **Inherited**.

After you build the model, the **Compiled Type** column of the Ports and Data Manager gives the actual type inherited from Simulink in the compiled simulation application.

The inherited type of output data is inferred from diagram actions that store values in the specified output. In the preceding example, the variables `mean` and `stdev` are computed from operations with double operands, which yield results of type `double`. If the expected type matches the inferred type, inheritance is successful. In all other cases, a mismatch occurs during build time.

Note Library MATLAB Function blocks can have inherited data types, sizes, and complexities like ordinary MATLAB Function blocks. However, all instances of the library block in a given model must have inputs with the same properties.

Built-In Data Types for Arguments

When you select **Built-in** for **Data type mode**, the **Data** properties dialog displays a **Data type** field that provides a drop-down list of supported data types. You can also choose a data type from the **Data Type** column in the Ports and Data Manager. The supported data types are:

Data Type	Description
`double`	64-bit double-precision floating point
`single`	32-bit single-precision floating point
`int32`	32-bit signed integer
`int16`	16-bit signed integer
`int8`	8-bit signed integer
`uint32`	32-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint8`	8-bit unsigned integer
`boolean`	Boolean (1 = `true`; 0 = `false`)

Specifying Argument Types with Expressions

You can specify the types of MATLAB Function block function arguments as expressions in the Ports and Data Manager.

1. Select `<data type expression>` from the **Type** drop-down menu of the Data properties dialog.
2. In the **Type** field, replace "*`<data type expression>`*" with an expression that evaluates to a data type. The following expressions are allowed:
   - Alias type from the MATLAB workspace, as described in `Simulink.AliasType`.
   - `fixdt` function to create a `Simulink.NumericType` object describing a fixed-point or floating-point data type
   - `type` operator, to base the type on previously defined data

Specifying Fixed-Point Designer Data Properties

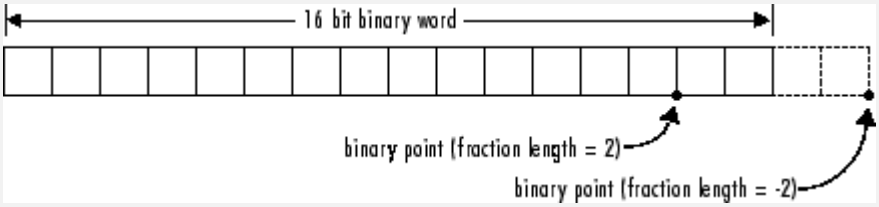
MATLAB Function blocks can represent signals and parameter values as fixed-point numbers. To simulate models that use fixed-point data in MATLAB Function blocks, you must install the Fixed-Point Designer product on your system.

You can set the following fixed-point properties:

Signedness. Select whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative quantities. Unsigned data represents positive values only. The default is `Signed`.

Word length. Specify the size (in bits) of the word that will hold the quantized integer. Large word sizes represent large quantities with greater precision than small word sizes. Word length can be any integer between 0 and 128 bits. The default is 16.

Scaling. Specify the method for scaling your fixed point data to avoid overflow conditions and minimize quantization errors. You can select the following scaling modes:

Scaling Mode	Description
`Binary point` (default)	If you select this mode, the Data Type Assistant displays the **Fraction Length** field, specifying the binary point location. Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:  The default is 0.
`Slope and bias`	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias**. • Slope can be any *positive* real number. The default is 1.0. • Bias can be any real number. The default value is 0.0. You can enter slope and bias as expressions that contain parameters defined in the MATLAB workspace.

Note You should use binary-point scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point scaling are performed with simple bit shifts and eliminate the expensive code implementations required for separate slope and bias values.

Data type override. Specify whether the data type override setting is `Inherit` (default) or `Off`.

Calculate Best-Precision Scaling. The Simulink software can automatically calculate “best-precision” values for both `Binary point` and `Slope and bias` scaling, based on the Limit range properties you specify.

To automatically calculate best precision scaling values:

1. Specify **Minimum**, **Maximum**, or both Limit range properties.
2. Click **Calculate Best-Precision Scaling**.

The Simulink software calculates the scaling values, then displays them in either the **Fraction Length**, or **Slope** and **Bias** fields.

Note The Limit range properties do not apply to `Constant` or `Parameter` scopes. Therefore, Simulink cannot calculate best-precision scaling for these scopes.

Fixed-point Details. You can view the following Fixed-point details:

Fixed-point Detail	Description
Representable maximum	The maximum number that can be represented by the chosen data type, sign, word length and fraction length (or data type, sign, slope and bias).
Maximum	The maximum value specified.
Minimum	The minimum value specified.
Representable minimum	The minimum number that can be represented by the chosen data type, sign, word length and fraction length (or data type, sign, slope and bias).
Precision	The precision for the given word length and fraction length (or slope and bias).

Using Data Type Override with the MATLAB Function Block

If you set the Data Type Override mode to `Double` or `Single` in Simulink, the MATLAB Function block sets the type of all inherited input signals and parameters to `fi double` or `fi single` objects respectively (see “MATLAB Function Block with Data Type Override” (Fixed-Point Designer) for more information). You must check the data types of your inherited input signals and parameters and use the Ports and Data Manager (see “Ports and Data Manager” on page 44-29) to set explicit types for any inputs that should not be fixed-point. Some operations, such as `sin`, are not applicable to fixed-point objects.

Note If you do not set the correct input types explicitly, you may encounter compilation problems after setting Data Type Override.

How Do I Set Data Type Override?

To set Data Type Override, follow these steps:

1. In the Simulink **Apps** tab, select **Fixed-Point Tool**.
2. Set the value of the **Data type override** parameter to `Double` or `Single`.

See Also

Related Examples

- “Add Parameter Arguments” on page 44-54

More About

- “MATLAB Function Block Editor” on page 44-26
- “Size Function Arguments” on page 44-51

Size Function Arguments

In this section...

Specifying Argument Size

To examine or specify the size of an argument, follow these steps:

1. From the MATLAB Function Block Editor, select **Edit Data**.
2. Enter the size of the argument in the **Size** field of the Data properties dialog, located in the **General** pane.

Note The default value is `-1`, indicating that size is inherited, as described in “Inheriting Argument Sizes from Simulink” on page 44-51.

Inheriting Argument Sizes from Simulink

Size defaults to `-1`, which means that the data argument inherits its size from Simulink based on its scope:

For Scope	Inherits Size
`Input`	From the Simulink input signal connected to the argument.
`Output`	From the Simulink output signal connected to the argument.
`Parameter`	From the Simulink or MATLAB parameter to which it is bound. See “Add Parameter Arguments” on page 44-54.

After you compile the model, the **Compiled Size** column in the **Contents** pane displays the actual size used in the compiled simulation application.

The size of an output argument is the size of the value that is assigned to it. If the expected size in the Simulink model does not match, a mismatch error occurs during compilation of the model.

Note No arguments with inherited sizes are allowed for MATLAB Function blocks in a library.

Specifying Argument Sizes with Expressions

The size of a data argument can be a scalar value or a MATLAB vector of values.

To specify size as a scalar, set the **Size** field to 1 or leave it blank. To specify **Size** as a vector, enter an array of up to two dimensions in `[row column]` format where

- Number of dimensions equals the length of the vector.
- Size of each dimension corresponds to the value of each element of the vector.

For example, a value of `[2 4]` defines a 2-by-4 matrix. To define a row vector of size 5, set the **Size** field to `[1 5]`. To define a column vector of size 6, set the **Size** field to `[6 1]` or just `6`. You can enter a MATLAB expression for each `[row column]` element in the **Size** field. Each expression can use one or more of the following elements:

- Numeric constants
- Arithmetic operators, restricted to `+`, `-`, `*`, and `/`
- Parameters
- Calls to the MATLAB functions `min`, `max`, and `size`

The following examples are valid expressions for **Size**:

```
k+1
size(x)
min(size(y),k)
```

In these examples, `k`, `x`, and `y` are variables of scope `Parameter`.

Once you build the model, the **Compiled Size** column displays the actual size used in the compiled simulation application.

See Also

Related Examples

- “Add Parameter Arguments” on page 44-54

More About

- “MATLAB Function Block Editor” on page 44-26
- “Type Function Arguments” on page 44-45

Units in MATLAB Function Blocks

In this section...

Units for Input and Output Data

MATLAB Function blocks support the specification of physical units as properties for data inputs and outputs. Specify units by using the **Unit (e.g., m, m/s^2, N*m)** parameter. When you start typing in the unit field, this parameter provides matching suggestions for units that Simulink supports. By default, the property is set to inherit the unit from the Simulink signal on the corresponding input or output port. If you select the **Data must resolve to Simulink signal object** property for output data, you cannot specify units. In this case, output data is assigned the same unit type as the Simulink signal connected to the output port.

To display the units on the Simulink lines in the model, on the **Debug** tab, select **Information Overlays > Units**.

Consistency Checking

MATLAB Function blocks check the consistency of the signal line unit from Simulink with the unit setting for the corresponding input or output data in the block. If the units do not match, Simulink displays a warning during model update.

Units for Stateflow Limitations

The unit property settings do not affect the execution of the MATLAB Function block. Simulink checks only consistency with the corresponding Simulink signal line connected to the input or output. It does not check consistency of assignments inside the MATLAB Function blocks. For example, Simulink does not warn against an assignment of an input with unit set to `ft` to an output with unit set to `m`. A MATLAB Function block does not perform unit conversions.

See Also

More About

- “Unit Specification in Simulink Models” on page 9-2

Add Parameter Arguments

Parameter arguments for MATLAB Function blocks do not take their values from signals in the Simulink model. Instead, Simulink searches up the workspace hierarchy. Simulink first looks in a masked workspace if the MATLAB Function block or a parent subsystem is masked. If the value is not found, it next looks in the model workspace and then the MATLAB base workspace.

You can provide a custom interface for parameters by masking the MATLAB Function block. Creating a mask for a block allows you to define the access for each parameter.

1. In the MATLAB Function Block Editor, add an argument to the function header of the MATLAB Function block. The name of the argument must match the name of the masked parameter or MATLAB variable that you want to pass to the MATLAB Function block.

   The new argument appears as an input port on the MATLAB Function block in the model.
2. In the MATLAB Function Block Editor, click **Edit Data**.
3. Select the new argument.
4. Set **Scope** to `Parameter` and click **Apply**.

   The input port for the parameter argument no longer appears in the MATLAB Function block.

Note Parameter arguments appear as arguments in the function header of the MATLAB Function block to maintain MATLAB consistency. As a result, you can test functions in a MATLAB Function block by copying and pasting them to MATLAB.

See Also

More About

- “Masking Fundamentals” on page 39-2
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26

Resolve Signal Objects for Output Data

In this section...

Implicit Signal Resolution

MATLAB Function blocks participate in signal resolution with Simulink signal objects. By default, output data from MATLAB Function blocks become associated with Simulink signal objects of the same name during a process called *implicit signal resolution*.

By default, implicit signal resolution generates a warning when you update the chart in the Simulink model. The following sections show you how to manage implicit signal resolution at various levels of the model hierarchy. See “Symbol Resolution” on page 67-127 and “Explicit and Implicit Symbol Resolution” on page 67-129 for more information.

Eliminating Warnings for Implicit Signal Resolution in the Model

To enable implicit signal resolution for all signals in a model, but eliminate the attendant warnings, follow these steps:

1. In the Simulink Editor, in the **Modeling** tab, click **Model Settings**.

   The Configuration Parameters dialog appears.
2. In the left pane of the Configuration Parameters dialog, under Diagnostics, select **Data Validity**.

   Data Validity configuration parameters appear in the right pane.
3. In the Signal resolution field, select **Explicit and implicit**.

Disabling Implicit Signal Resolution for a MATLAB Function Block

To disable implicit signal resolution for a MATLAB Function block in your model, follow these steps:

1. In the Simulink Editor, in the **Modeling** tab, click **Model Settings**.

   The Configuration Parameters dialog appears.
2. In the left pane of the Configuration Parameters dialog, under Diagnostics, select **Data Validity**.

   Data Validity configuration parameters appear in the right pane.
3. In the Signal resolution field, select **Explicit only** or **None**.

Forcing Explicit Signal Resolution for an Output Data Signal

To force signal resolution for an output signal in a MATLAB Function block, follow these steps:

1. In the Simulink model, right-click the signal line connected to the output that you want to resolve and select **Properties** from the context menu.
2. In the Signal Properties dialog, enter a name for the signal that corresponds to the signal object.
3. Select the **Signal name must resolve to Simulink signal object** check box and click **OK**.

See Also

`Simulink.Signal`

More About

- “Symbol Resolution” on page 67-127

Types of Structures in MATLAB Function Blocks

In MATLAB Function blocks, you can define structure data as inputs or outputs that interact with bus signals. MATLAB Function blocks also support arrays of buses. You can also define structures inside MATLAB functions that are not part of MATLAB Function blocks.

The following table summarizes how to create different types of structures in MATLAB Function blocks:

Scope	How to Create	Details
Input	Create structure data with scope of `Input`.	You can create structure data as inputs or outputs in the top-level MATLAB function for interfacing to other environments. See “Create Structures in MATLAB Function Blocks” on page 44-63.
Output	Create structure data with scope of `Output`.	
Local	Create a local variable implicitly in a MATLAB function.	See “Define Scalar Structures for Code Generation” on page 54-4.
Persistent	Declare a variable to be persistent in a MATLAB function.	See `persistent`.
Parameter	Create structure data with scope of `Parameter`.	See “Define and Use Structure Parameters” on page 44-69.

Structures in MATLAB Function blocks can contain fields of any type and size, including mux signals, buses, and arrays of structures.

See Also

Related Examples

- “Combine Buses into an Array of Buses” on page 76-64

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

Attach Bus Signals to MATLAB Function Blocks

For an example of how to use structures in a MATLAB Function block, open the model emldemo_bus_struct.

In this model, a MATLAB Function block receives a bus signal using the structure `inbus` at input port 1 and outputs two bus signals from the structures `outbus` at output port 1 and `outbus1` at output port 2. The input signal comes from the Bus Creator block `MainBusCreator`, which bundles signals `ele1`, `ele2`, and `ele3`. The signal `ele3` is the output of another Bus Creator block `SubBusCreator`, which bundles the signals `a1` and `a2`. The structure `outbus` connects to a Bus Selector block `BusSelector1`; the structure `outbus1` connects to another Bus Selector block `BusSelector3`.

To explore the MATLAB function `fcn`, double-click the MATLAB Function block. Notice that the code implicitly defines a local structure variable `mystruct` using the `struct` function, and uses this local structure variable to initialize the value of the first output `outbus`. It initializes the second output `outbus1` to the value of field `ele3` of structure `inbus`.

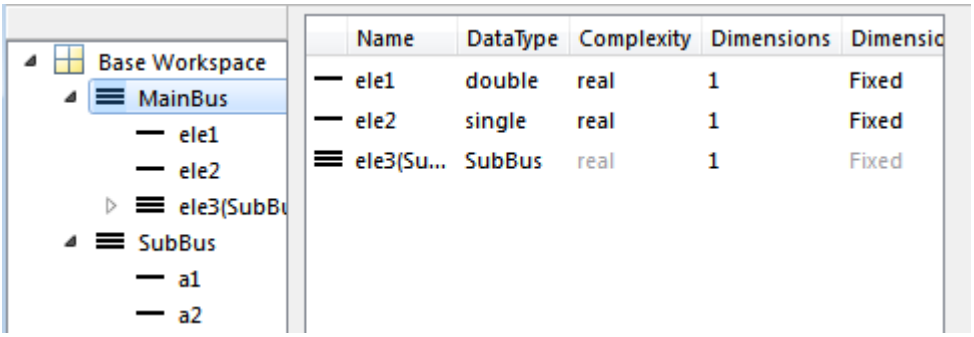
Structure Definitions in Example

Here are the definitions of the structures in the MATLAB Function block in the example, as they appear in the Ports and Data Manager:

Name	Scope	Port	Resolve Signal	DataType	Size
inbus	Input	1		Inherit: Same as Simulink	-1
outbus	Output	1	☐	Bus: MainBus	1
outbus1	Output	2	☐	Bus: SubBus	1

Bus Objects Define Structure Inputs and Outputs

Each structure input and output must be defined by a `Simulink.Bus` object in the base workspace (see “Create Structures in MATLAB Function Blocks” on page 44-63). This means that the structure shares the same properties as the bus object, including number, name, type, and sequence of fields. In this example, the following bus objects define the structure inputs and outputs:



The `Simulink.Bus` object `MainBus` defines structure input `inbus` and structure output `outbus`. The `Simulink.Bus` object `SubBus` defines structure output `outbus1`. Based on these definitions, `inbus` and `outbus` have the same properties as `MainBus` and, therefore, reference their fields by the same names as the fields in `MainBus`, using dot notation (see “Index Substructures and Fields” on

page 44-62). Similarly, `outbus1` references its fields by the same names as the fields in `SubBus`. Here are the field references for each structure in this example:

Structure	First Field	Second Field	Third Field
`inbus`	`inbus.ele1`	`inbus.ele2`	`inbus.ele3`
`outbus`	`outbus.ele1`	`outbus.ele2`	`outbus.ele3`
`outbus1`	`outbus1.a1`	`outbus1.a2`	—

See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

How Structure Inputs and Outputs Interface with Bus Signals

Buses in a Simulink model appear inside the MATLAB Function block as structures; structure outputs from the MATLAB Function block appear as buses in Simulink models. When you create structure inputs, the MATLAB Function block determines the type, size, and complexity of the structure from the input signal. When you create structure outputs, you must define their type, size, and complexity in the MATLAB function.

You connect structure inputs and outputs from MATLAB Function blocks to any bus signal, including:

- Blocks that output bus signals — such as Bus Creator blocks
- Blocks that accept bus signals as input — such as Bus Selector and Gain blocks
- S-Function blocks
- Other MATLAB Function blocks

You can use global bus type data in Data Store Memory blocks with MATLAB Function blocks. For more information on using buses and Data Store Memory, see “Data Stores with Buses and Arrays of Buses” on page 73-23.

Working with Virtual and Nonvirtual Buses

MATLAB Function blocks supports nonvirtual buses only (see “Types of Composite Signals” on page 76-2). For MATLAB Function block bus inputs, incoming virtual bus signals are converted to nonvirtual buses.

See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

Rules for Defining Structures in MATLAB Function Blocks

Follow these rules when defining structures in MATLAB Function blocks:

- For each structure input or output in a MATLAB Function block, you must define a `Simulink.Bus` object in the base workspace to specify its type.
- MATLAB Function blocks support nonvirtual buses only.

See Also

`Simulink.Bus`

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2
- “Working with Virtual and Nonvirtual Buses” on page 44-60

Index Substructures and Fields

As in MATLAB, you index substructures and fields structures in MATLAB Function blocks by using dot notation. However, for code generation from MATLAB, you must reference field values individually (see “Structure Definition for Code Generation” on page 54-2).

For example, in the emldemo_bus_struct model described in “Attach Bus Signals to MATLAB Function Blocks” on page 44-58, the MATLAB function uses dot notation to index fields and substructures:

```
function [outbus, outbus1] = fcn(inbus)
%#codegen
substruct.a1 = inbus.ele3.a1;
substruct.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                  'ele3',substruct);

outbus = mystruct;
outbus.ele3.a2 = 2*(substruct.a2);

outbus1 = inbus.ele3;
```

The following table shows how the code generation software resolves symbols in dot notation for indexing elements of the structures in this example:

Dot Notation	Symbol Resolution
`substruct.a1`	Field `a1` of local structure `substruct`
`inbus.ele3.a1`	Value of field `a1` of field `ele3`, a substructure of structure input`inbus`
`inbus.ele3.a2(1,1)`	Value in row 1, column 1 of field `a2` of field `ele3`, a substructure of structure input `inbus`

See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

Create Structures in MATLAB Function Blocks

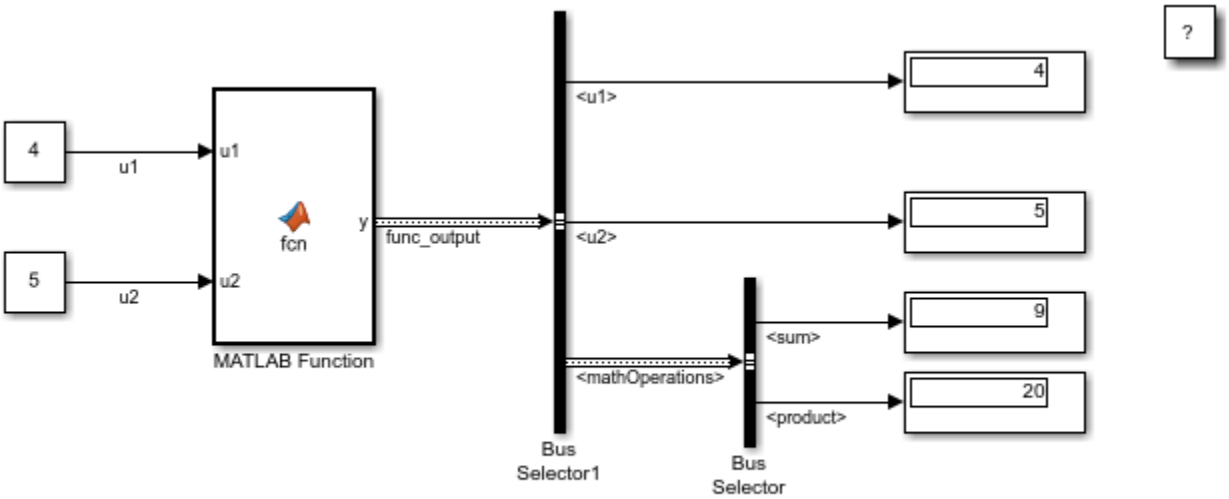
Here is the workflow for creating a structure in a MATLAB Function block:

1. Decide on the type (or scope) of the structure (see "Types of Structures in MATLAB Function Blocks" on page 44-57).
2. Based on the scope, follow these guidelines for creating the structure:

For Structure Scope:	Follow These Steps:
Input	**a** Create a `Simulink.Bus` object in the base workspace to define the structure input. **b** Add data to the MATLAB Function block, as described in "Adding Data to a MATLAB Function Block" on page 44-35. The data should have the following properties • **Scope** = `Input` • **Type** = `Bus: <object name>` For `<object name>`, enter the name of the `Simulink.Bus` object that defines the structure input See "Rules for Defining Structures in MATLAB Function Blocks" on page 44-61.
Output	**a** Create a `Simulink.Bus` object in the base workspace to define the structure output. **b** Add data to the MATLAB Function block with the following properties: • **Scope** = `Output` • **Type** = `Bus: <object name>` For `<object name>`, enter the name of the `Simulink.Bus` object that defines the structure output **c** Define and initialize the output structure implicitly as a variable in the MATLAB function, as described in "Structure Definition for Code Generation" on page 54-2. **d** Make sure the number, type, and size of fields in the output structure variable definition match the properties of the `Simulink.Bus` object.
Local	Define the structure implicitly as a local variable in the MATLAB function, as described in "Structure Definition for Code Generation" on page 54-2. By default, local variables in MATLAB Function blocks are temporary.
Persistent	Define the structure implicitly as a persistent variable in the MATLAB function.
Parameter	**a** Create a structure variable in the base workspace. **b** Add data to the MATLAB Function block with the following properties: • **Name** = same name as the structure variable you created in step 1. • **Scope** = `Parameter` See "Define and Use Structure Parameters" on page 44-69.

Use Nonvirtual Buses with MATLAB Function Blocks

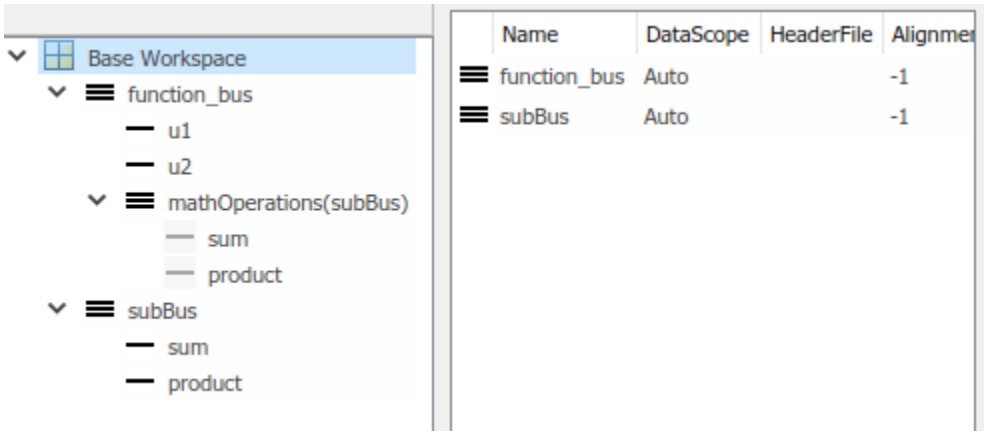
In this example model, the MATLAB Function block includes MATLAB code that creates a structure. If a MATLAB Function block outputs a structure, then you must use a `Simulink.Bus` object to define the bus output.



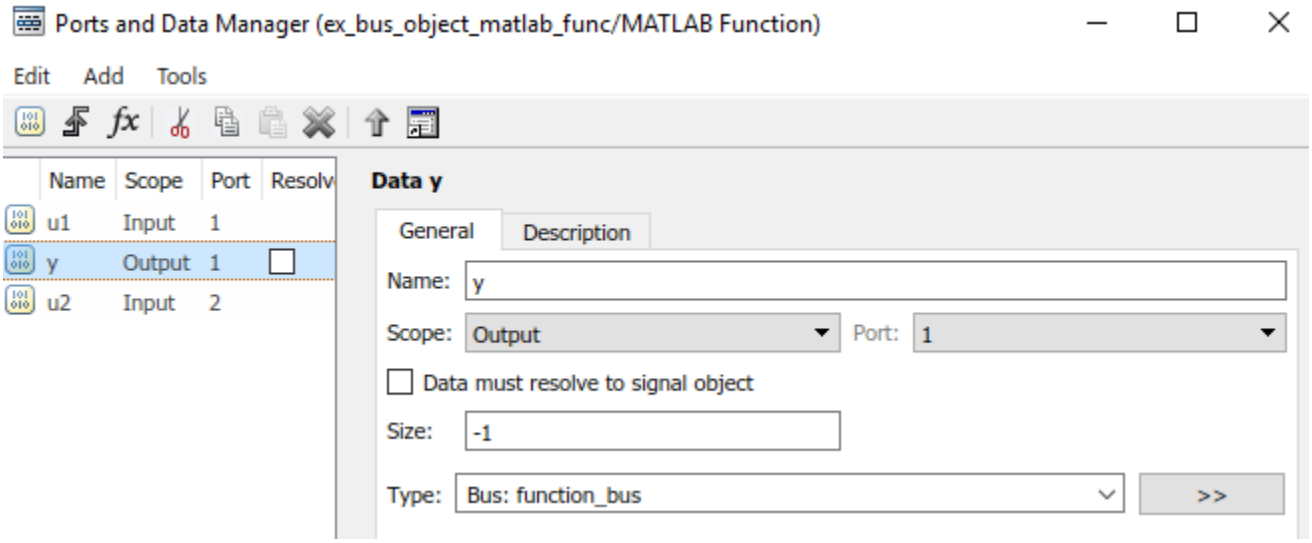
To see the structure definition, double-click the MATLAB Function block.

```
function y = fcn(u1,u2)

y.u1 = u1;
y.u2 = u2;
y.mathOperations.sum     = u1+u2;
y.mathOperations.product = u1*u2;
```

Open the Bus Editor and expand the `Bus` object definition that represents the structure.



To see how the `Bus` object defines the bus output for the MATLAB Function block, in the MATLAB Toolstrip, on the Editor tab, click **Edit Data** and then click `y`. The output **Type** is defined as the `function_bus` `Bus` object.



See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44

Assign Values to Structures and Fields

You can assign values to any structure, substructure, or field in a MATLAB Function block. Here are the guidelines:

Operation	Conditions
Assign one structure to another structure	You must define each structure with the same number, type, and size of fields, either as `Simulink.Bus` objects in the base workspace or locally as implicit structure declarations (see “Create Structures in MATLAB Function Blocks” on page 44-63).
Assign one structure to a substructure of a different structure and vice versa	You must define the structure with the same number, type, and size of fields as the substructure, either as `Simulink.Bus` objects in the base workspace or locally as implicit structure declarations.
Assign an element of one structure to an element of another structure	The elements must have the same type and size.

For example, the following table presents valid and invalid structure assignments based on the specifications for the model described in “Attach Bus Signals to MATLAB Function Blocks” on page 44-58:

Assignment	Valid or Invalid?	Rationale
`outbus = mystruct;`	Valid	Both `outbus` and `mystruct` have the same number, type, and size of fields. The structure `outbus` is defined by the `Simulink.Bus` object `MainBus` and `mystruct` is defined locally to match the field properties of `MainBus`.
`outbus = inbus;`	Valid	Both `outbus` and `inbus` are defined by the same `Simulink.Bus` object, `MainBus`.
`outbus1 = inbus.ele3;`	Valid	Both `outbus1` and `inbus.ele3` have the same type and size because each is defined by the `Simulink.Bus` object `SubBus`.
`outbus1 = inbus;`	Invalid	The structure `outbus1` is defined by a different `Simulink.Bus` object than the structure `inbus`.

See Also

Related Examples

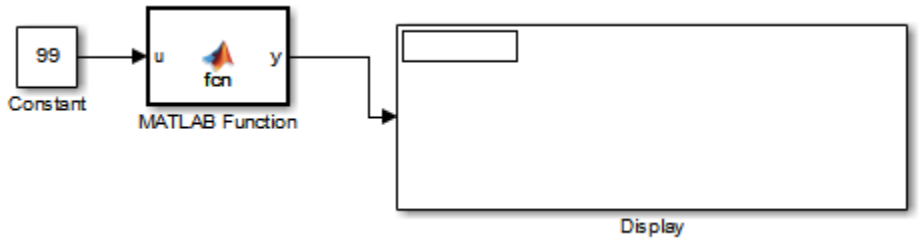
- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

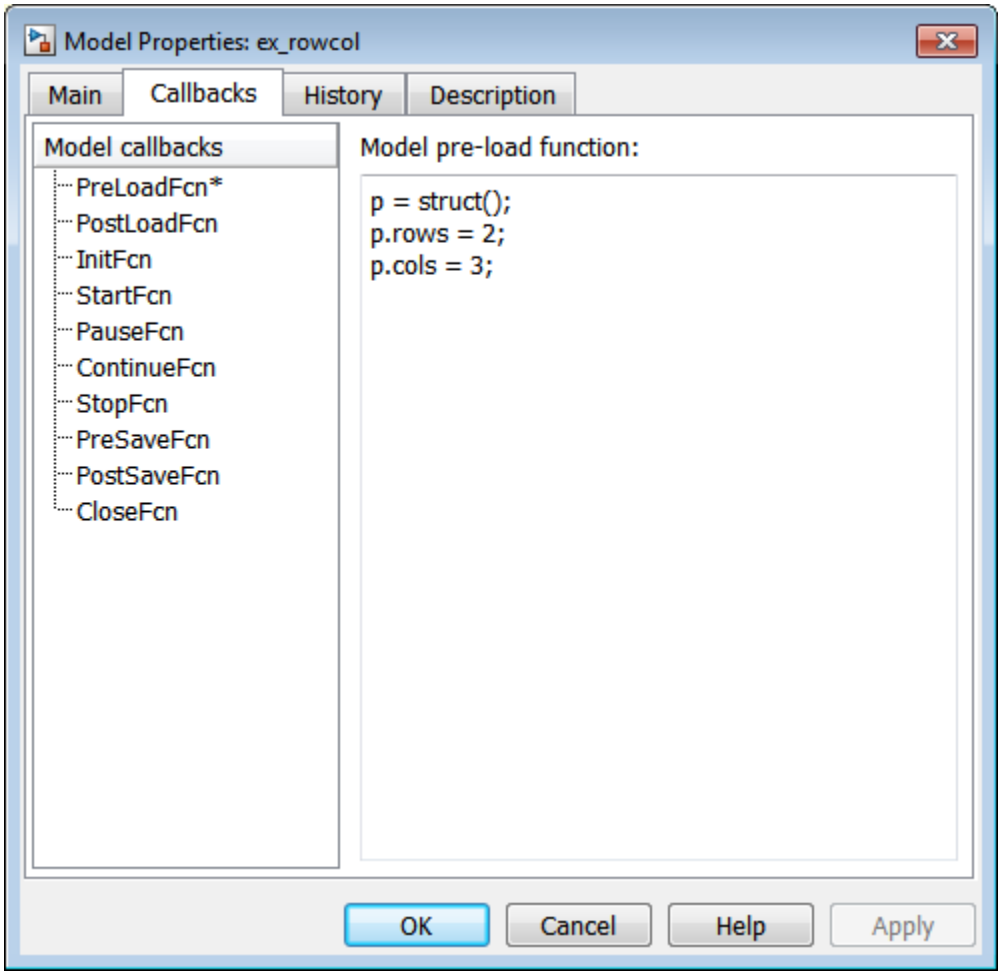
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

Initialize a Matrix Using a Nontunable Structure Parameter

The following simple example uses a nontunable structure parameter input to initialize a matrix output. The model looks like this:



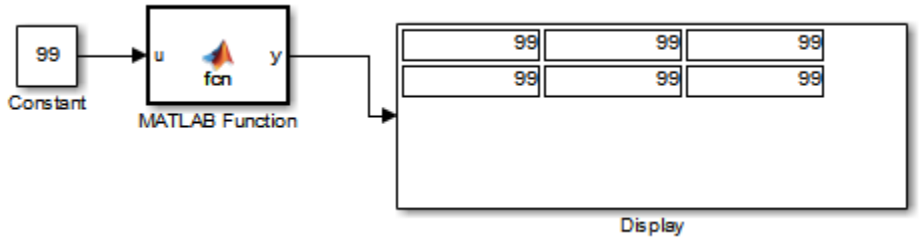
This model defines a structure variable `p` in its pre-load callback function, as follows:



The structure `p` has two fields, `rows` and `cols`, which specify the dimensions of a matrix. The MATLAB Function block uses a constant input `u` to initialize the matrix output `y`. Here is the code:

```
function y = fcn(u, p)
y = zeros(p.rows,p.cols) + u;
```

Running the model initializes each element of the 2-by-3 matrix y to 99, the value of u:



See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2
- “Define and Use Structure Parameters” on page 44-69

Define and Use Structure Parameters

In this section...

Defining Structure Parameters

To define structure parameters in MATLAB Function blocks, follow these steps:

1. Define and initialize a structure variable

   A common method is to create a structure in the base workspace.

2. In the Ports and Data Manager, add data in the MATLAB Function block with the following properties:

Property	What to Specify
Name	Enter same name as the structure variable you defined in the base workspace
Scope	Select `Parameter`
Tunable	Leave checked if you want to change (tune) the value of the parameter during simulation; otherwise, clear to make the parameter nontunable and preserve the initial value during simulation
Type	Select `Inherit: Same as Simulink`

3. Click **Apply**.

FIMATH Properties of Nontunable Structure Parameters

FIMATH properties for nontunable structure parameters containing fixed-point values are based on the initial values of the structure. They do *not* come from the FIMATH properties specified for fixed-point input signals to the parent MATLAB Function block. (These FIMATH properties appear in the properties dialog box for MATLAB Function blocks.)

See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2
- “Organize Related Block Parameter Definitions in Structures” on page 37-19

Limitations of Structures and Buses in MATLAB Function Blocks

- Structures in MATLAB Function blocks support a subset of the operations available for MATLAB structures (see “Structures”).
- You cannot use structures that contain cell arrays or classes for Simulink signals, parameters, or data store memory.
- You cannot use variable-size data with arrays of buses (see “Array of Buses Requirements and Limitations” on page 76-70).

See Also

Related Examples

- “Create Structures in MATLAB Function Blocks” on page 44-63
- “Define and Use Structure Parameters” on page 44-69

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Structure Definition for Code Generation” on page 54-2

Control Support for Variable-Size Arrays in a MATLAB Function Block

By default, support for variable-size arrays is enabled for a MATLAB Function block. To disable this support:

1. In the MATLAB Function Block Editor, select **Edit Data**.
2. Clear the **Support variable-size arrays** check box.

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26
- “Declare Variable-Size Inputs and Outputs” on page 44-72
- “Code Generation for Variable-Size Arrays” on page 53-2

Declare Variable-Size Inputs and Outputs

By default, a MATLAB Function block input signal or output signal is not variable-size. To make the signal variable-size:

1. In the MATLAB Function Block Editor, select **Edit Data**.
2. Select the input or output signal.
3. Select the **Variable size** check box.
4. Enter the size according to this table.

For:	Specify
Input	To inherit the size from Simulink, enter `-1`. Otherwise, specify the explicit size and upper bound. For example, to specify a 2-by-4 matrix, enter `[2 4]`.
Output	Specify the explicit size and upper bound.

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26
- “Control Support for Variable-Size Arrays in a MATLAB Function Block” on page 44-71
- “Code Generation for Variable-Size Arrays” on page 53-2
- “Use a Variable-Size Signal in a Filtering Algorithm” on page 44-73

Use a Variable-Size Signal in a Filtering Algorithm

In this section...

About the Example

This example uses a variable-size vector to store the values of a white noise signal. The size of the vector can vary at run time because the signal values get pruned by functions that:

- Filter out signal values that are not unique within a specified tolerance of each other.
- Average every two signal values and output only the resulting means.

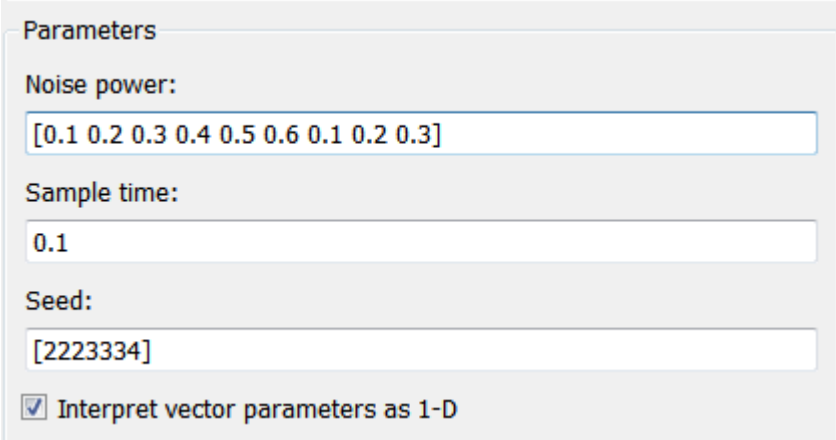
Simulink Model

Open the example model by typing `emldemo_process_signal` at the MATLAB command prompt. The model contains the following blocks:

Simulink Block	Description
Band-Limited White Noise	Generates a set of normally distributed random values as the source of the white noise signal.
MATLAB Function `uniquify`	Filters out signal values that are not unique to within a specified tolerance of each other.
MATLAB Function `avg`	Outputs the average of a specified number of unique signal values.
Unique values	Scope that displays the unique signal values output from the `uniquify` function.
Average values	Scope that displays the average signal values output from the `avg` function.

Source Signal

The band-limited white noise signal has these properties:



The size of the noise power value defines the size of the array that holds the signal values. This array is a 1-by-9 vector of double values.

MATLAB Function Block: uniquify

This block filters out signal values that are not within a tolerance of 0.2 of each other. Here is the code:

```
function y = uniquify(u) %#codegen
y = emldemo_uniquetol(u,0.2);
```

The `uniquify` function calls an external MATLAB function `emldemo_uniquetol` to filter the signal values. `uniquify` passes the 1-by-9 vector of white noise signal values as the first argument and the tolerance value as the second argument. Here is the code for `emldemo_uniquetol`:

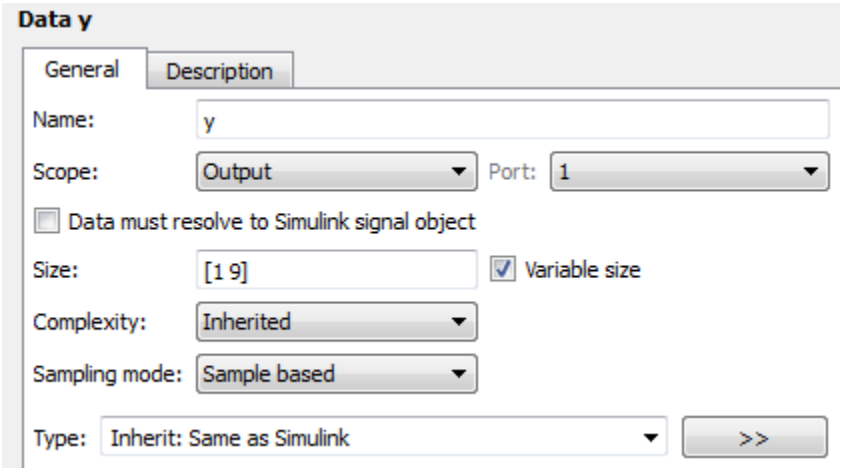
```
function B = emldemo_uniquetol(A,tol) %#codegen

A = sort(A);
coder.varsize('B',[1 100]);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

`emldemo_uniquetol` returns the filtered values of `A` in an output vector `B` so that `abs(B(i) - B(j)) > tol` for all `i` and `j`. Every time Simulink samples the Band-Limited White Noise block, it generates a different set of random values for `A`. As a result, `emldemo_uniquetol` may produce a different number of output signals in `B` each time it is called. To allow `B` to accommodate a variable number of elements, `emldemo_uniquetol` declares it as variable-size data with an explicit upper bound:

```
coder.varsize('B',[1 100]);
```

In this statement, `coder.varsize` declares `B` as a vector whose first dimension is fixed at 1 and whose second dimension can grow to a maximum size of 100. Accordingly, output `y` of the `uniquify`

block must also be variable-size so that it can pass the values returned from `emldemo_uniquetol` to the **Unique values** scope. Here are the properties of `y`:



For variable-size outputs, you must specify an explicit size and upper bound, shown here as [1 9].

MATLAB Function Block: avg

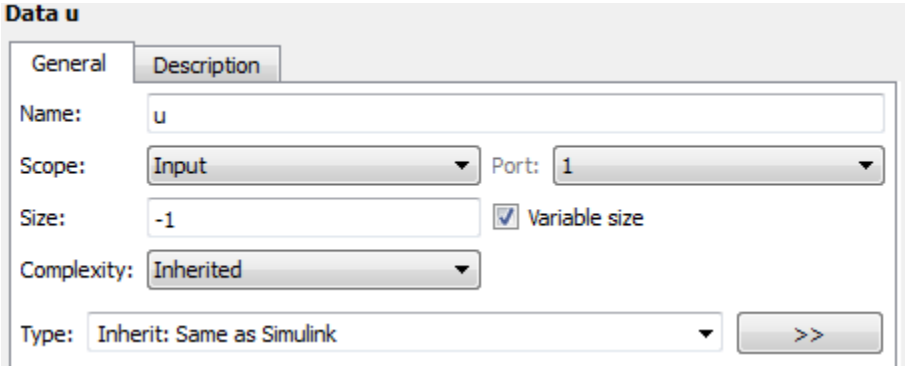
This block averages signal values filtered by the `uniquify` block as follows:

If number of signal values is	The MATLAB Function block
> 1 and divisible by 2	Averages every consecutive pair of values
> 1 but *not* divisible by 2	Drops the first (smallest) value and average the remaining consecutive pairs
= 1	Returns the value unchanged

The `avg` function outputs the results to the **Average values** scope. Here is the code:

```
function y = avg(u) %#codegen

if numel(u) == 1
    y = u;
else
    k = numel(u)/2;
    if k ~= floor(k)
        u = u(2:numel(u));
    end
    y = emldemo_navg(u,2);
end
```

Both input `u` and output `y` of `avg` are declared as variable-size vectors because the number of elements varies depending on how the `uniquify` function block filters the signal values. Input `u` inherits its size from the output of `uniquify`.



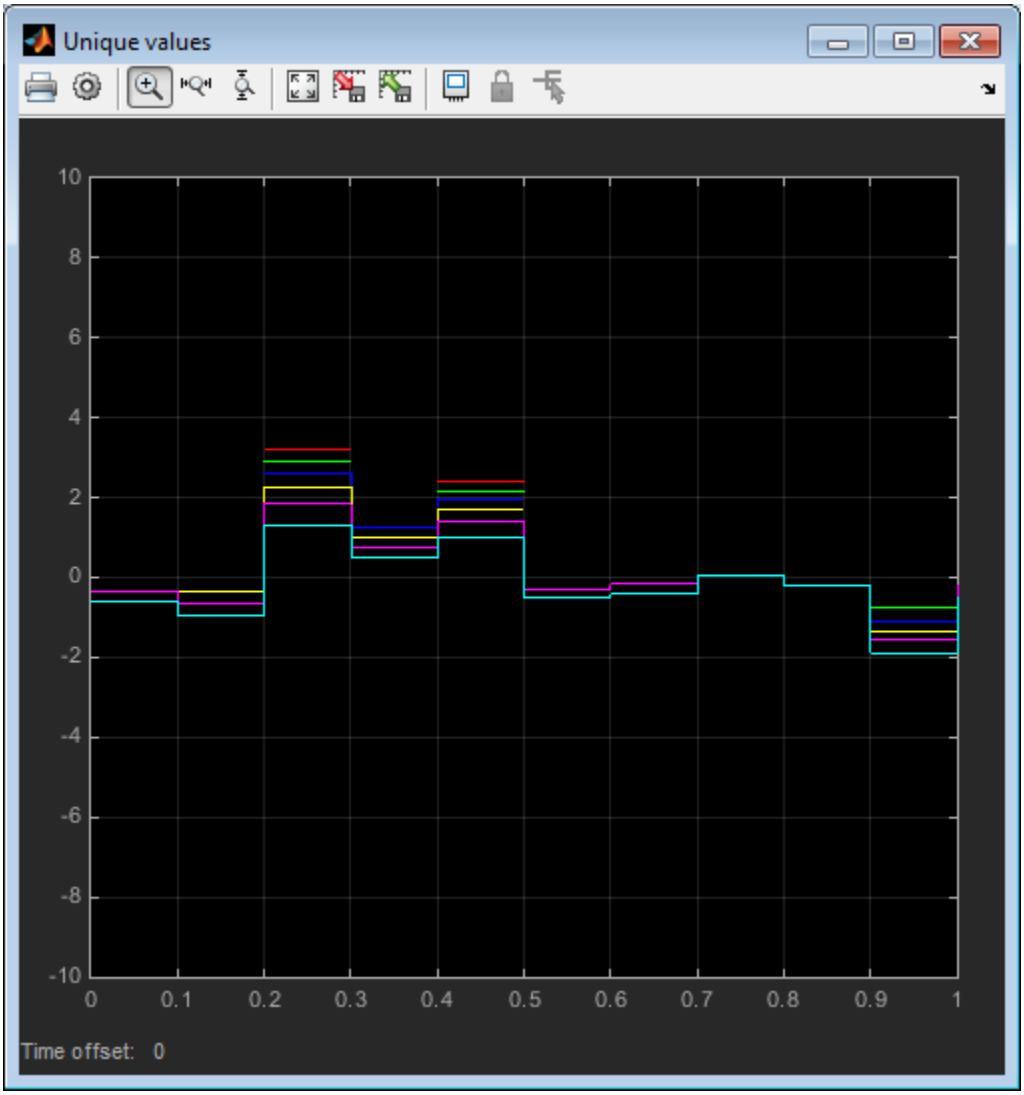
The `avg` function calls an external MATLAB function `emldemo_navg` to calculate the average of every two consecutive signal values. Here is the code for `emldemo_navg`:

```
function B = emldemo_navg(A,n) %#codegen

assert(n>=1 && n<=numel(A));

B = zeros(1,numel(A)/n);
k = 1;
for i = 1 : numel(A)/n
     B(i) = mean(A(k + (0:n-1)));
     k = k + n;
end
```

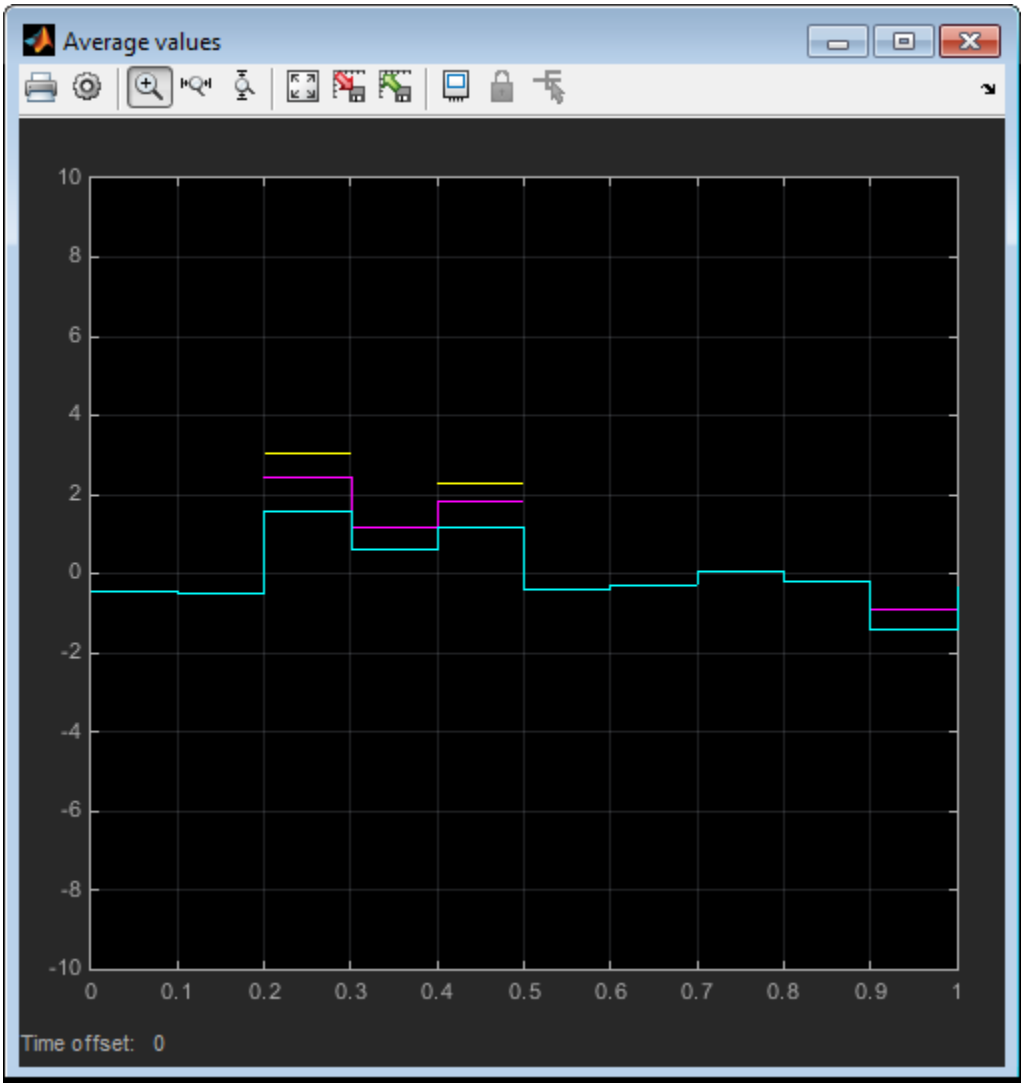
Variable-Size Results

Simulating the model produces the following results:

- The `uniquify` block outputs a variable number of signal values each time it executes:



- The `avg` block outputs a variable number of signal values each time it executes — approximately half the number of the unique values:



See Also

`coder.varsize`

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26

Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block

Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. You can use dynamic memory allocation for arrays inside a MATLAB Function block.

You cannot use dynamic memory allocation for:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.
- Discrete state properties of System objects associated with a MATLAB System block.

Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

Dynamic memory allocation and the freeing of this memory can result in slower execution of the generated code. To control the use of dynamic memory allocation for variable-size arrays in a MATLAB Function block, you can:

- Provide upper bounds for variable-size arrays on page 44-79.
- Disable dynamic memory allocation for MATLAB Function blocks on page 44-79.
- Modify the dynamic memory allocation threshold on page 44-79.

Provide Upper Bounds for Variable-Size Arrays

For an unbounded variable-size array, the code generator allocates memory dynamically on the heap. For a bounded variable-size array, if the size, in bytes, is less than the dynamic memory allocation threshold, the code generator allocates memory statically on the stack. To avoid dynamic memory allocation, provide upper bounds for the array dimensions so that the size of the array, in bytes, is less than the dynamic memory allocation threshold. See “Specify Upper Bounds for Variable-Size Arrays” on page 53-5.

Disable Dynamic Memory Allocation for MATLAB Function Blocks

By default, dynamic memory allocation for MATLAB Function blocks is enabled for GRT-based targets and disabled for ERT-based targets. To change the setting, in the Configuration Parameters dialog box, clear or select **Dynamic memory allocation in MATLAB functions**.

If you disable dynamic memory allocation, you must provide upper bounds for variable-size arrays.

Modify the Dynamic Memory Allocation Threshold

Instead of disabling dynamic memory allocation for all variable-size arrays, you can use the dynamic memory allocation threshold to specify when the code generator uses dynamic memory allocation.

Use the dynamic memory allocation threshold to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. However, static memory allocation can lead to unused storage space. You can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

The default value of the dynamic memory allocation threshold is 64 kilobytes. To change the threshold, in the Configuration Parameters dialog box, set the **Dynamic memory allocation threshold in MATLAB functions** parameter.

To use dynamic memory allocation for all variable-size arrays, set the threshold to `0`.

See Also

More About

- “Code Generation for Variable-Size Arrays” on page 53-2
- “Specify Upper Bounds for Variable-Size Arrays” on page 53-5
- “Use Dynamic Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-81

Use Dynamic Memory Allocation for Variable-Size Arrays in a MATLAB Function Block

This example shows how to use dynamic memory allocation for variable-size arrays in a MATLAB Function block. Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

You can use dynamic memory allocation only for arrays that are local to the MATLAB Function block.

You cannot use dynamic memory allocation for:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.
- Discrete state properties of System objects associated with a MATLAB System block.

Create Model

Create this Simulink model that has a MATLAB Function block with an unbounded variable-size array.

1. Create a Simulink model `mymodel`.
2. Add a MATLAB Function block to the model.
3. In the MATLAB Function block, add this code:

```
function s = myfcn(n)
Z = rand(1,n);
s = sum(Z);
end
```

4. Add a Constant block to the left of the MATLAB Function block.
5. Add an Outport block to the right of the MATLAB Function block.
6. Connect the blocks.

Configure Model for Dynamic Memory Allocation

Make sure that you configure the model to use dynamic memory allocation for variable-size arrays in MATLAB Function blocks. In the Configuration Parameters dialog box, in the **Simulation Target > Advanced parameters** category, make sure that:

- The **Dynamic memory allocation in MATLAB functions** check box is selected.
- The **Dynamic memory allocation threshold in MATLAB functions** parameter has the default value 65536.

Simulate Model Using Dynamic Memory Allocation

1. Simulate the model.
2. In the MATLAB Function Editor, to open the MATLAB Function report, click **View Report**.

   The variables tab shows that Z is a 1-by-:? array. The colon (:) indicates that the second dimension is variable-size. The question mark (?) indicates that the second dimension is unbounded.

Simulation must use dynamic memory allocation for Z because the second dimension of Z does not have an upper bound.

Use Dynamic Memory Allocation for Bounded Arrays

When an array is unbounded, the code generator must use dynamic memory allocation. If an array is bounded, the code generator uses dynamic memory allocation only if the array size, in bytes, is greater than or equal to the dynamic memory allocation threshold. The default value for this threshold is 65536.

Dynamic memory has a run-time performance cost. By controlling its use, you can improve execution speed.

If you make Z a bounded variable-size array with a size that is greater than the threshold, the code generator uses dynamic memory allocation for Z. For example:

1. In `mymodel`, modify `myfcn` so that Z has an upper bound of 500.

   ```
   function s = myfcn(n)
   assert(n < 500);
   Z = rand(1,n);
   s = sum(Z);
   end
   ```

2. Simulate the model.

   In the MATLAB Function report, you see that Z is a 1-by-:500 array. It is variable-size with an upper bound of 500.

3. Lower the dynamic memory allocation to a value less than or equal to `4000`, which is the size, in bytes, of Z. In the Configuration Parameters dialog box, in the **Simulation Target > Advanced parameters** category, set the **Dynamic memory allocation threshold in MATLAB functions** parameter to 4000.
4. Simulate the model.

   The code generator uses dynamic memory allocation because the size of Z is equal to the dynamic memory allocation threshold, 4000.

Generate C Code That Uses Dynamic Memory Allocation

If you have Simulink Coder, you can generate C code for this model. Then, you can see how the code generator represents dynamically allocated arrays.

1. Configure the model to use a fixed-step solver. In the Configuration Parameters dialog box, in the **Solver** pane, under **Solver selection**:
   - For **Type**, select `Fixed-step`.
   - For **Solver**, select `discrete (no continuous states)`.
2. Configure the model to create and use a code generation report. In the Configuration Parameters dialog box, in the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
3. Edit the code in the MATLAB Function block so that it looks like this code:

   ```
   function s = myfcn(n)
   Z = rand(1,n);
   s = sum(Z);
   end
   ```

   Z is an unbounded variable-size array.
4. Make sure that the model is configured for dynamic memory allocation:
   - The **Dynamic memory allocation in MATLAB functions** check box is selected.
   - The **Dynamic memory allocation threshold in MATLAB functions** parameter has the default value 65536.
5. Build the model.
6. In the code generation report, open `mymodel.c`. You can tell that the code generator used dynamic memory allocation for Z because you see the `emxArray` type `emxArray_real_T_mymodel_T` and `emxArray` utility functions, such as `mymodel_emxInit_real_T`. The code generator uses an `emxArray` type for variables whose memory is dynamically allocated. The generated code uses the `emxArray` utility functions to manage the `emxArray`s.

If you have Embedded Coder, you can customize the identifiers for `emxArray` types and the utility functions. See “Identifier Format Control” (Embedded Coder).

See Also

More About

- “Code Generation for Variable-Size Arrays” on page 53-2
- “Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-79
- “Identifier Format Control” (Embedded Coder)

Code Generation for Enumerations

Enumerations represent a fixed set of named values. Enumerations help make your MATLAB code and generated C/C++ code more readable. For example, the generated code can test equality with code such as `if (x == Red)` instead of using `strcmp`. To generate C/C++ code, you must have Simulink Coder.

When you use enumerations in a `MATLAB Function` block, adhere to these restrictions:

- Calls to methods of enumeration classes are not supported.
- Passing strings or character vectors to constructors of enumerations is not supported.
- For a MATLAB Function block, you can import an externally defined type by using `Simulink.defineIntEnumType` or you can define an enumeration class. The enumeration class must derive from one of these base types: `Simulink.IntEnumType`, `int8`, `uint8`, `int16`, `uint16`, or `int32`. See “Define Enumerations for MATLAB Function Blocks” on page 44-84.
- You can use only a limited set of operations on enumerations. See “Allowed Operations on Enumerations” on page 44-85.
- Use enumerations with functions that support enumerated types for code generation. See “MATLAB Toolbox Functions That Support Enumerations” on page 44-86.

Define Enumerations for MATLAB Function Blocks

You can define enumerations for MATLAB Function blocks in two ways:

- To import an externally defined enumeration, use the `Simulink.defineIntEnumType` function. See “Import Enumerations Defined Externally to MATLAB” on page 68-10.
- In a class definition file, define an enumerated type. For example:

```
classdef PrimaryColors < Simulink.IntEnumType
    enumeration
        Red(1),
        Blue(2),
        Yellow(4)
    end
end
```

If you define an enumerated type in a class definition file, the class must derive from one of these base types: `Simulink.IntEnumType`, `int8`, `uint8`, `int16`, `uint16`, or `int32`. Then, you can exchange enumerated data between MATLAB Function blocks and other Simulink blocks in a model.

If you use Simulink Coder to generate C/C++ code, you can use the enumeration class base type to control the size of an enumerated type in generated C/C++ code. You can:

- Represent an enumerated type as a fixed-size integer that is portable to different targets.
- Reduce memory usage.
- Interface with legacy code.
- Match company standards.

The base type determines the representation of the enumerated type in generated C/C++ code.

If the base type is `Simulink.IntEnumType`, the code generator produces a C enumeration type. Consider the following MATLAB enumerated type definition:

```
classdef LEDcolor < Simulink.IntEnumType
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

This enumerated type definition results in the following C code:

```
typedef enum {
  GREEN = 1,
  RED
} LEDcolor;
```

For built-in integer base types, the code generator produces a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. Consider the following MATLAB enumerated type definition:

```
classdef LEDcolor < int16
    enumeration
        GREEN(1),
        RED(2)
    end
 end
```

This enumerated type definition results in the following C code:

```
typedef int16_T LEDcolor;

#define GREEN                          ((LEDcolor)1)
#define RED                            ((LEDcolor)2)
```

To customize the code generated for an enumerated type, see “Customize Simulink Enumeration” on page 68-7.

Allowed Operations on Enumerations

For code generation, you are restricted to the operations on enumerations listed in this table.

Operation	Example	Notes
assignment operator: =		—
relational operators: < > <= >= == ~=	`xon == xoff`	Code generation does not support using == or ~= to test equality between an enumeration member and a string array, a character array, or a cell array of character arrays.
cast operation	`double(LEDcolor.RED)`	—

Operation	Example	Notes

conversion to character array or string	`y = char(LEDcolor.RED);`
	`y1 = cast(LEDcolor.RED,'char');`
	`y2 = string(LEDcolor.RED);`
		• You can convert only compile-time scalar valued enumerations. For example, this code runs in MATLAB, but produces an error in code generation:
		`y2 = string(repmat(LEDcolor.RED,1,2));`
		• The code generator preserves enumeration names when the conversion inputs are constants. For example, consider this enumerated type definition:
		`classdef AnEnum < int32`
		`enumeration`
		`zero(0),`
		`two(2),`
		`otherTwo(2)`
		`end`
		`end`
		Generated code produces `"two"` for
		`y = string(AnEnum.two)`
		and `"otherTwo"` for
		`y = string(AnEnum.two)`

indexing operation	`m = [1 2]`
	`n = LEDcolor(m)`
	`p = n(LEDcolor.GREEN)`
		—

control flow statements: if, switch, while	`if state == sysMode.ON`
	`led = LEDcolor.GREEN;`
	`else`
	`led = LEDcolor.RED;`
	`end`
		—

MATLAB Toolbox Functions That Support Enumerations

For code generation, you can use enumerations with these MATLAB toolbox functions:

- `cast`
- `cat`
- `char`
- `circshift`
- `enumeration`
- `fliplr`
- `flipud`
- `histc`
- `intersect`
- `ipermute`

- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismember`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `setdiff`
- `setxor`
- `shiftdim`
- `sort`
- `sortrows`
- `squeeze`
- `string`
- `union`
- `unique`

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Add Enumerated Inputs, Outputs, and Parameters to a MATLAB Function Block” on page 44-88
- “Use Enumerations to Control an LED Display” on page 44-89

Add Enumerated Inputs, Outputs, and Parameters to a MATLAB Function Block

When you add enumerated inputs, outputs, or parameters to a MATLAB Function block, follow these guidelines:

- For inputs, inherit the type from the enumerated type of the connected Simulink signal or specify the enumeration explicitly.
- For outputs, specify the enumerated type explicitly.
- For tunable parameters, specify the enumerated type explicitly. For nontunable parameters, derive properties from an enumerated parameter in a parent Simulink masked subsystem or enumerated variable defined in the MATLAB base workspace.

To add enumerated data to a MATLAB Function block:

1. In the MATLAB Function Block Editor, select **Edit Data**.
2. In the Ports and Data Manager, select **Add** > **Data**.
3. In the **Name** field, enter a name for the enumerated data.

   For parameters, the name must match the enumerated masked parameter or workspace variable name.
4. In the **Type** field, specify an enumerated type.

   - For an explicit enumerated type, set **Type** to `Enum:<class name>`. Replace `<class name>` with the name of an enumerated data type that you defined in a MATLAB file on the MATLAB path.

     The **Complexity** field is not visible because enumerated data types do not support complex values.
   - To inherit the enumerated type from a connected Simulink signal (for inputs only), set **Type** to `Inherit:Same as Simulink`.

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Code Generation for Enumerations” on page 44-84
- “Use Enumerations to Control an LED Display” on page 44-89

Use Enumerations to Control an LED Display

In this section...

This example shows how to use enumerations in a MATLAB Function block. The example shows how MATLAB Function blocks exchange enumerated data with other Simulink blocks.

The `emldemo_led_switch` model uses enumerations to represent the modes of a device that controls the colors of an LED display. The MATLAB Function block receives an enumerated input signal that represents the mode. The enumerated output signal represents the color that the LED displays.

Simulink Model

To open the model, at the command prompt, enter:

```
emldemo_led_switch
```

The model contains the blocks listed in this table.

Simulink Block	Description
Step	Provides source of the on/off signal. Outputs an initial value of 0 (off) and at 10 seconds steps up to a value of 1 (on).
Data Type Conversion from `double` to `int32`	Converts the `Step` signal of type `double` to type `int32`.
Data Type Conversion from `int32` to enumerated type `switchmode`	Converts the value of type `int32` to the enumerated type `switchmode`. The Data Type Conversion block parameters have these settings: • Output minimum: `[]` • Output maximum: `[]` • Output data type: `Enum:switchmode`
MATLAB Function block `checkState`	Evaluates the enumerated input `state` to determine the value of the enumerated output `ledval`. `state` inherits its enumerated type `switchmode` from the Simulink step signal. `ledval` has the type `Enum:led`.
Display	Displays the value of `ledval`.

Enumeration Class Definitions

The `switchmode` enumeration represents the allowed modes for the input to the `checkstate` block.

```
classdef switchmode < Simulink.IntEnumType
  enumeration
    OFF(0)
    ON(1)
  end
end
```

The `led` enumeration represents the colors that the `checkstate` block can output.

```
classdef led < Simulink.IntEnumType
    enumeration
        GREEN(1),
        RED(8)
    end
end
```

Both enumeration classes inherit from the built-in type `Simulink.IntEnumType` and reside on the MATLAB path.

MATLAB Function Block Function

The function `checkState` uses enumerations to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state. It lights a red LED display to indicate the OFF state.

```
function ledval = checkState(state)
%#codegen

if state == switchmode.ON
    ledval = led.GREEN;
else
    ledval = led.RED;
end
```

Simulation

When you simulate the model, the Display block displays the state of the LED display. If you simulate the model for less than 10 seconds, the state is off. The Display block displays `RED`. If you simulate the model for more than 10 seconds, the state is on. The Display block displays `GREEN`.

See Also

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Code Generation for Enumerations” on page 44-84
- “Add Enumerated Inputs, Outputs, and Parameters to a MATLAB Function Block” on page 44-88
- “Enumerations and Scopes” on page 68-4

Share Data Globally

In this section...

When Do You Need to Use Global Data?

You might need to use global data with a MATLAB Function block if:

- You have multiple MATLAB functions that use global variables and you want to call these functions from MATLAB Function blocks.
- You have an existing model that uses a large amount of global data and you are adding a MATLAB Function block to this model, and you want to avoid cluttering your model with additional inputs and outputs.
- You want to scope the visibility of data to parts of the model.

Using Global Data with the MATLAB Function Block

In Simulink, you store global data using data store memory. You implement data store memory using either Data Store Memory blocks or `Simulink.Signal` objects. How you store global data depends on the number and scope of your global variables. For more information, see "Local and Global Data Stores" on page 73-2 and "Choosing How to Store Global Data" on page 44-92.

How MATLAB Globals Relate to Data Store Memory

In MATLAB functions in Simulink, global declarations are not mapped to the MATLAB global workspace. Instead, you register global data with the MATLAB Function block to map the data to data store memory. This difference allows global data in MATLAB functions to inter-operate with the Simulink solver and to provide diagnostics if they are misused.

A global variable resolves hierarchically to the closest data store memory with the same name in the model. The same global variable occurring in two different MATLAB Function blocks might resolve to different data store memory depending on the hierarchy of your model. You can use this ability to scope the visibility of data to a subsystem.

How to Use Globals with the MATLAB Function Block

To use global data in your MATLAB Function block, or in any code that this block calls, you must:

1. Declare a global variable in your MATLAB Function block, or in any code that is called by the MATLAB Function block.
2. Register a Data Store Memory block or `Simulink.Signal` object that has the same name as the global variable with the MATLAB Function block.

For more information, see “Storing Data Using Data Store Memory Blocks” on page 44-92 and “Storing Data Using Simulink.Signal Objects” on page 44-93.

Choosing How to Store Global Data

The following table summarizes whether to use Data Store Memory blocks or `Simulink.Signal` objects.

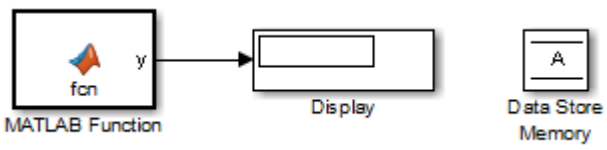
If you want to:	Use:	For more information:
Use a small number of global variables in a single model that does not use model reference.	Data Store Memory blocks. **Note** Using Data Store Memory blocks scopes the data to the model.	“Storing Data Using Data Store Memory Blocks” on page 44-92
Use a large number of global variables in a single model that does not use model reference.	`Simulink.Signal` objects defined in the model workspace. `Simulink.Signal` objects offer these advantages: • You do not have to add numerous Data Store Memory blocks to your model. • You can load the `Simulink.Signal` objects in from a MAT-file.	“Storing Data Using Simulink.Signal Objects” on page 44-93
Share data between multiple models (including referenced models).	`Simulink.Signal` objects defined in the base workspace **Note** If you use Data Store Memory blocks as well as `Simulink.Signal`, note that using Data Store Memory blocks scopes the data to the model.	“Storing Data Using Simulink.Signal Objects” on page 44-93

Storing Data Using Data Store Memory Blocks

This model demonstrates how a MATLAB Function block uses the global data stored in a Data Store Memory block `A`.

1 Open the `dsm_demo` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'dsm_demo.mdl')))
```



2. Double-click the MATLAB Function block to open the MATLAB Function Block Editor.

   The MATLAB Function block code declares a global variable `A`. The block modifies the value of `A` during each execution.

   ```
   function y = fcn
   %#codegen
   global A;
   A = A+1;
   y = A;
   ```

3. Make sure the global variable is registered to the MATLAB Function block. See “Adding Data to a MATLAB Function Block” on page 44-35.

   a. In the MATLAB Function Block Editor, select **Edit Data** to open the Ports and Data Manager dialog box.

   b. In the Ports and Data Manager, select the data **A** in the left pane. This data uses the same name as the global variable.

   c. The **Scope** of the data is set to `Data Store Memory`.

   See also “Ports and Data Manager” on page 44-29.

4. Double-click the Data Store Memory block `A`. In the Block Parameters dialog box, you see that the **Data store name** `A` matches the global variable name. The block has an initial value of `25`.

   When you add a Data Store Memory to your model:

   a. Set the **Data store name** to match the name of the global variable in your MATLAB Function block code.

   b. Set **Data type** to an explicit data type. The data type cannot be `auto`.

   c. Set the **Signal type** and specify an **Initial value**.

5. Simulate the model.

   The MATLAB Function block reads the initial value of global data stored in `A` and updates the value of `A` each time it executes.

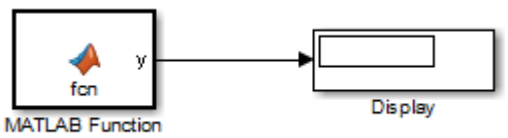
Storing Data Using Simulink.Signal Objects

This model demonstrates how a MATLAB Function block uses the global data stored in a `Simulink.Signal` object `A`.

1. Open the simulink_signal_local model.

   Open the `simulink_signal_local` model. At the MATLAB command line, enter:

   ```
   run(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
                                'examples', 'simulink_signal_local.mdl')))
   ```



The model uses a `Simulink.Signal` object in the model workspace.

Note To use the global data with multiple models, create a `Simulink.Signal` object in the base workspace .

2 Make sure that the `Simulink.Signal` object is added to the Model Explorer.

   a In the **Modeling** tab, click **Model Explorer**.

   b In the left pane of the Model Explorer, select the model workspace for the `simulink_signal_local` model.

      The **Contents** pane displays the data in the model workspace.

   c Click the `Simulink.Signal` object `A`.

      In the right pane, make sure that the Model Explorer displays these attributes for `A`.

Attribute	Value
Data type	`double`
Complexity	`real`
Dimensions	1
Initial value	5

   See also **Model Explorer**.

3 Double-click the MATLAB Function block to open its editor.

   The MATLAB Function block modifies the value of global data `A` each time it executes.

```
function y = fcn
%#codegen
global A;
A = A+1;
y = A;
```

4 Make sure the `Simulink.Signal` object is registered to the MATLAB Function block.

   a In the MATLAB Function Block Editor, select **Edit Data** to open the Ports and Data Manager dialog box.

   b In the Ports and Data Manager, select the data **A** in the left pane. This data uses the same name as the global variable.

   c Set the **Scope** of the data to `Data Store Memory`.

   See also “Ports and Data Manager” on page 44-29.

5 Simulate the model.

   The MATLAB Function block reads the initial value of global data stored in `A` and updates the value of `A` each time it executes.

Using Data Store Diagnostics to Detect Memory Access Issues

You can configure your model to provide run-time and compile-time diagnostics for avoiding problems with data stores. Diagnostics are available in the Configuration Parameters dialog box and the

parameters dialog box for the Data Store Memory block. These diagnostics are available for Data Store Memory blocks only, not for `Simulink.Signal` objects. For more information on using data store diagnostics, see “Data Store Diagnostics” on page 73-3.

Note If you pass data store memory arrays to functions, optimizations such as `A=foo(A)` might result in the code generation software marking the entire contents of the array as read or written even though only some elements were accessed.

Limitations of Using Shared Data in MATLAB Function Blocks

There is no Data Store Memory support for:

- MATLAB structures
- Variable-sized data

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

Initialize Persistent Variables in MATLAB Functions

A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. For code generation, functions must initialize a persistent variable if it is empty. For more information, see `persistent`.

When programming MATLAB functions in these situations:

- MATLAB Function blocks with no direct feedthrough
- MATLAB Function blocks in models that contain State Control blocks in `Synchronous` mode
- MATLAB functions in Stateflow charts that implement Moore machine semantics

The specialized semantics impact how a function initializes its persistent data. Because the initialization must be independent of the input to the function, follow these guidelines:

- The function initializes its persistent variables only by accessing constants.
- The control flow of the function does not depend on whether the initialization occurs.

For example, this function has a persistent variable `n`.

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

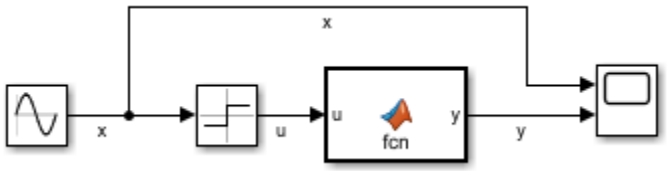
This type of initialization results in an error because the initial value of `n` depends on the input `u` and the `return` statement interrupts the normal control flow of the function.

To correct the error, initialize the persistent variable by setting it to a constant value and remove the `return` statement. For example, this function initializes the persistent variable without producing an error.

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

MATLAB Function Block With No Direct Feedthrough

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.

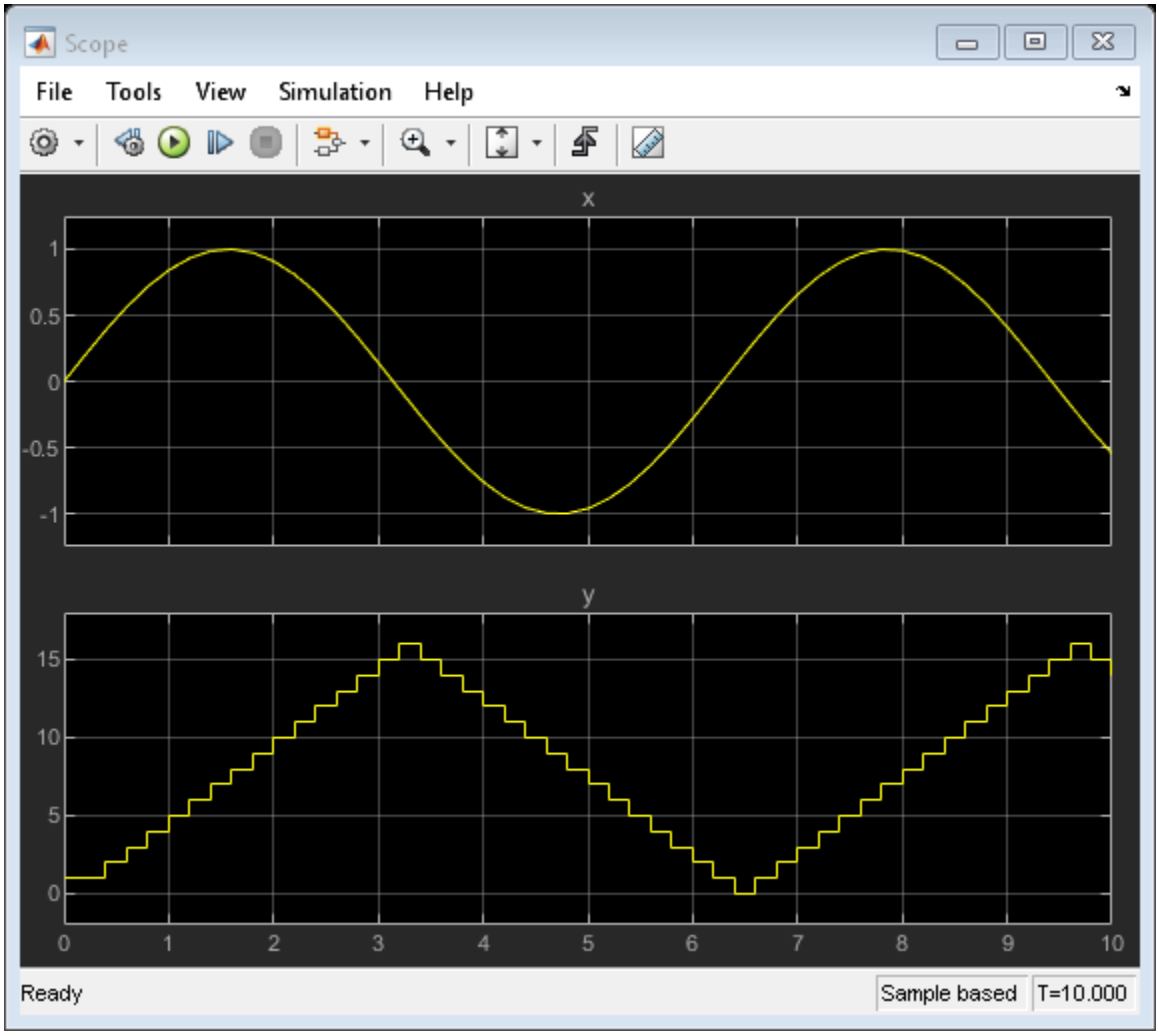


In the MATLAB function block:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

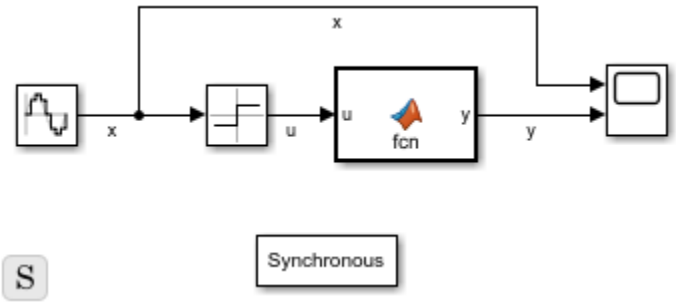
Because the **Allow direct feedthrough** check box is cleared, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



State Control Block in Synchronous Mode

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.

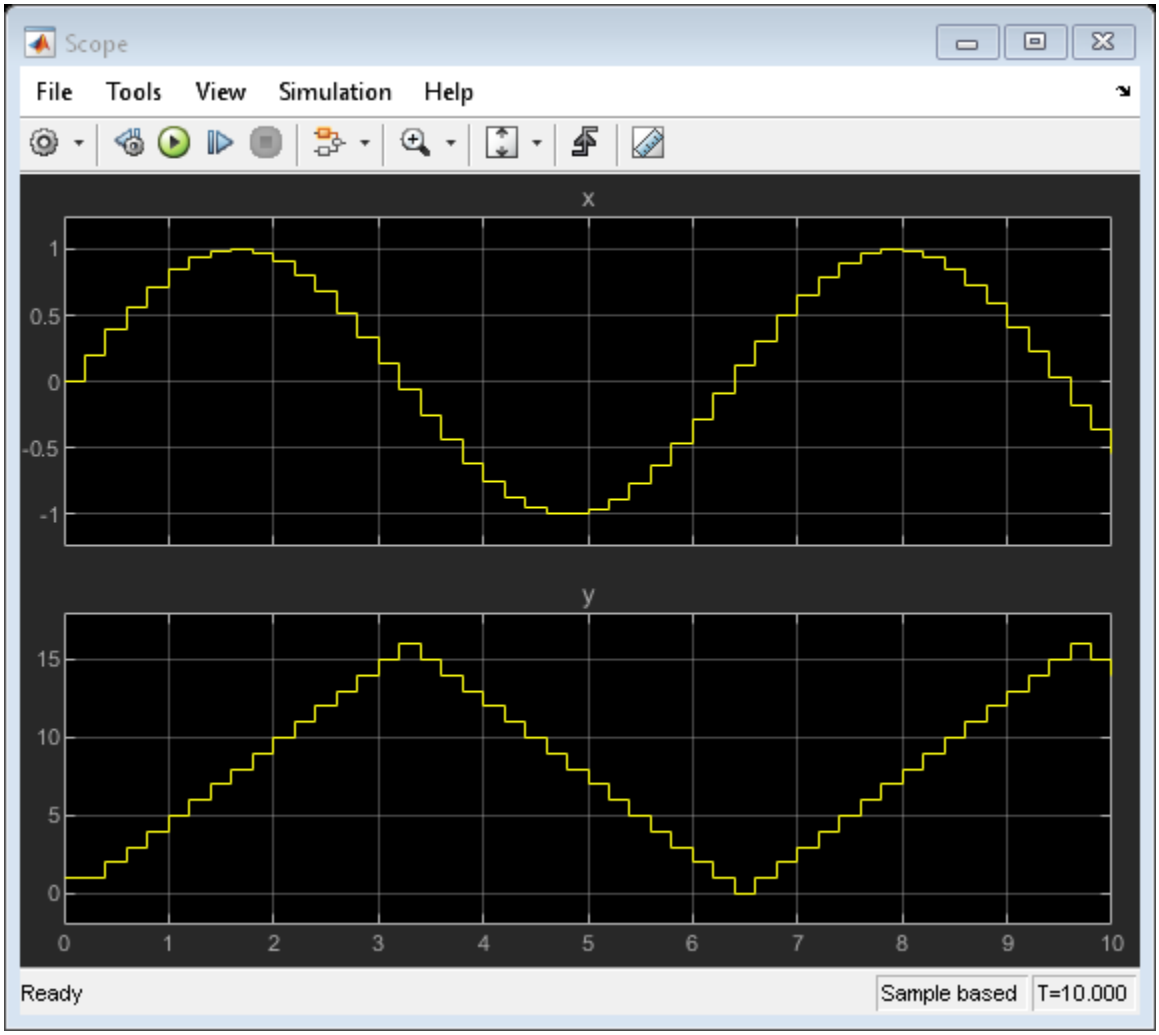


In the MATLAB function block:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

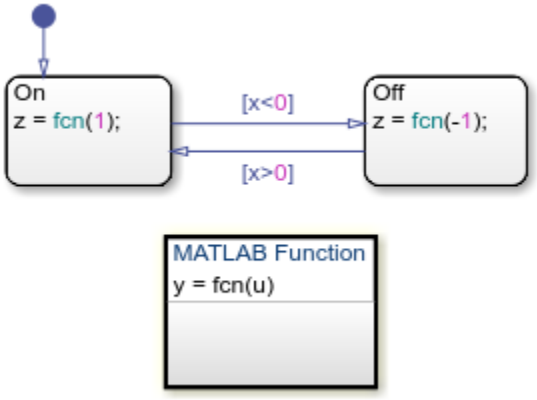
Because the model contains a State Control block in `Synchronous` mode, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



Stateflow Chart Implementing Moore Semantics

This model contains a Stateflow chart that implements Moore machine semantics. The chart contains a MATLAB function that defines the function `fcn`, described previously. The input `u` has values of 1 and -1 that depend on the state of the chart.

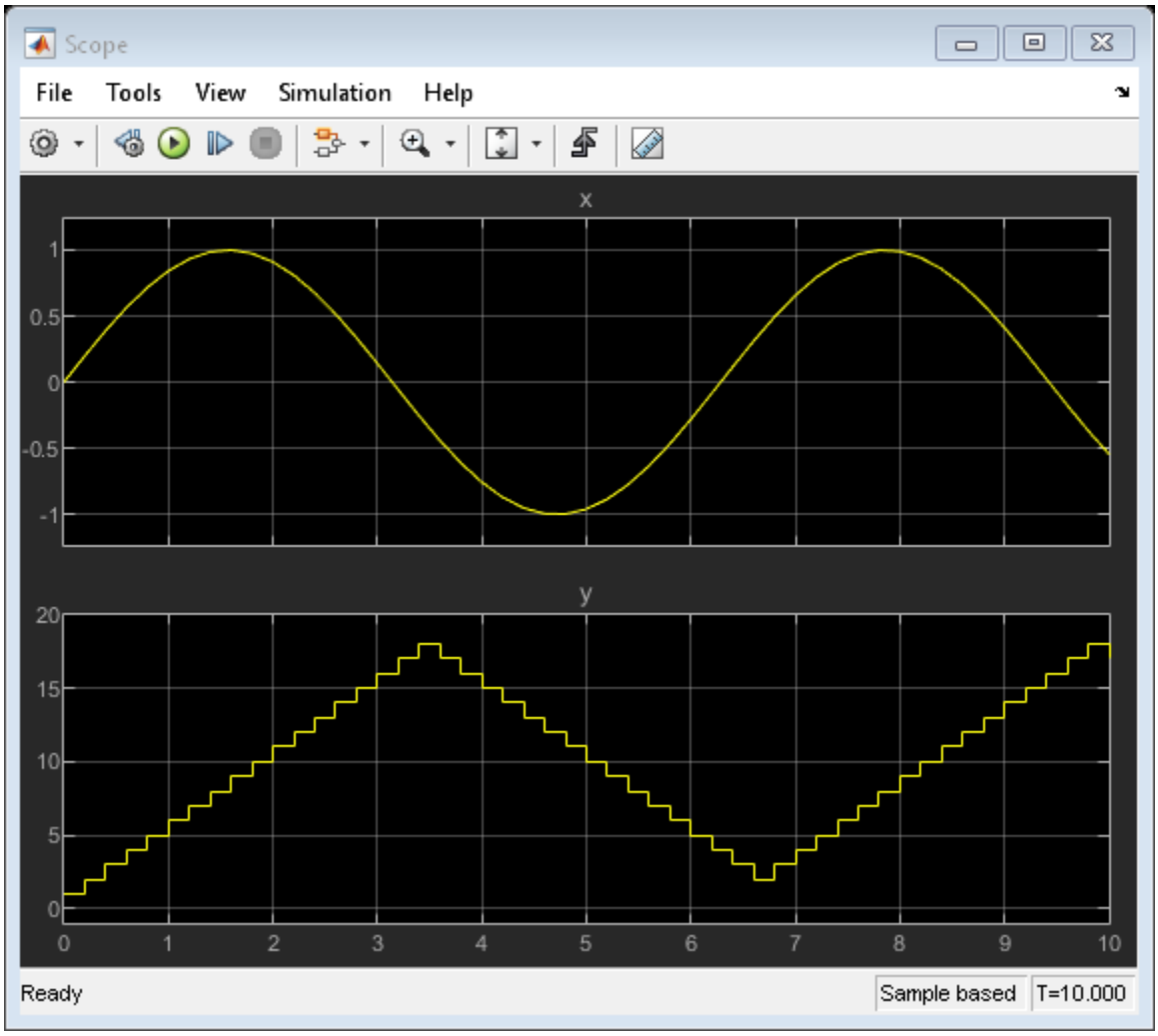


In the MATLAB function:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

Because the chart implements Moore semantics, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



See Also

Chart | MATLAB Function | State Control | `persistent`

More About

- “Use Nondirect Feedthrough in a MATLAB Function Block” on page 44-178
- “Synchronous Subsystem Behavior with the State Control Block” (HDL Coder)
- “Design Considerations for Moore Charts” (Stateflow)

Create Custom Block Libraries

In this section...

When to Use MATLAB Function Block Libraries

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. If you want to reuse a set of MATLAB algorithms in Simulink models, you can encapsulate your MATLAB code in a MATLAB Function block library.

As with other Simulink block libraries, you can specialize each instance of MATLAB Function library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code

How to Create Custom MATLAB Function Block Libraries

Here is a basic workflow for creating custom block libraries with MATLAB Function blocks. To work through these steps with an example, see “Example: Creating a Custom Signal Processing Filter Block Library” on page 44-102.

1. Add polymorphic MATLAB code to MATLAB Function blocks in a Simulink model.

   Polymorphic code is code that can process data with different properties, such as type, size, and complexity.
2. Configure the blocks to inherit the properties you want to specialize.

   For a list of properties you can specialize, see “Properties You Can Specialize Across Instances of Library Blocks” on page 44-114.
3. Optionally, customize your library code using masking.
4. Add instances of MATLAB Function library blocks to a Simulink model.

Note If your MATLAB Function block library is masked, you cannot modify contents of the block with mask initialization code. The **Allow library block to modify its contents** option in the Mask dialog box is not supported for MATLAB Function block libraries.

Example: Creating a Custom Signal Processing Filter Block Library

- “What You Will Learn” on page 44-103
- “About the Filter Algorithms” on page 44-103
- “Step 1: Add the Filter Algorithms to MATLAB Function Library Blocks” on page 44-103

What You Will Learn

This simple example takes you through the workflow described in “How to Create Custom MATLAB Function Block Libraries” on page 44-102 to show you how to:

- Create a library of signal processing filter algorithms using MATLAB Function blocks
- Customize one of the library blocks using mask parameters
- Convert one of the filter algorithms to source-protected P-code that you can call from a MATLAB Function library block

About the Filter Algorithms

The MATLAB filter algorithms are:

my_fft

Performs a discrete Fourier transform on an input signal. The input can be a vector, matrix, or multidimensional array whose length is a power of 2.

my_conv

Convolves two input vector signals. Outputs a subsection of the convolution with a size specified by a mask parameter, `Shape`.

my_sobel

Convolves a 2D input matrix with a Sobel edge detection filter.

Step 1: Add the Filter Algorithms to MATLAB Function Library Blocks

1. In Simulink, create a library model. On the **Simulation** tab, select **New > Library**
2. Drag three MATLAB Function blocks into the model from the User-Defined Functions section of the Simulink Library Browser and name them:
   - `my_fft_filter`
   - `my_conv_filter`
   - `my_sobel_filter`
3. Save the library model as `my_filter_lib`.
4. Open the MATLAB Function block named `my_fft_filter`, replace the template code with the following code, and save the block:

   ```
   function y = my_fft(x)

   y = fft(x);
   ```
5. Replace the template code in `my_conv_filter` block with the following code and save the block:

   ```
   function c = my_conv(a, b)

   c = conv(a, b);
   ```

6 Replace the template code in `my_sobel_filter` block with the following code and save the block:

```
function y = my_sobel(u)

%% "my_sobel_filter" is a MATLAB function
%%  on the  MATLAB path.
y = my_sobel_filter(u);
```

The `my_sobel` function acts as a wrapper that calls a MATLAB function, `my_sobel_filter`, on the code generation path. `my_sobel_filter` implements the algorithm that convolves a 2D input matrix with a Sobel edge detection filter. By calling the function rather than inlining the code directly in the MATLAB Function block, you can reuse the algorithm both as MATLAB code and in a Simulink model. You will create `my_sobel_filter` next.

7 In the same folder where you created `my_filter_lib`, create a new MATLAB function `my_sobel_filter` with the following code:

```
function y = my_sobel_filter(u)

% Sobel edge detection filter
h = [1     2     1;...
     0     0     0;...
    -1    -2    -1];

y = abs(conv2(u, h));
```

Save the file as `my_sobel_filter.m`.

Step 2: Configure Blocks to Inherit Properties You Want to Specialize

In this example, the data in the signal processing filter algorithms must inherit size, type, and complexity from the Simulink model. By default, data in MATLAB Function blocks inherit these properties. To explicitly configure data to inherit properties:

1 Open a MATLAB Function block and select **Edit Data**.

2 In the left pane of the Ports and Data Manager, select the data of interest.

3 In the right pane, configure the data to inherit properties from Simulink:

To Inherit	What to Specify
Size	Enter `-1` in Size field
Complexity	Select **Inherited** from the Complexity menu
Type	Select **Inherit: Same as Simulink** from the Type menu

For example, if you open the MATLAB Function block my_fft_filter and look at the properties of input x in the Ports and Data Manager, you see that size, type, and complexity are inherited by default.

Note If your design has specific requirements or constraints, you can enter values for any of these properties, rather than inherit them from Simulink. For example, if your algorithm is not supposed to work with complex inputs, set **Complexity** to **Off**.

Step 3: Customize Your Library Using Masking

In this exercise you will modify the convolution filter `my_conv` to use a custom parameter `shape` that specifies what subsection of the convolution to output. To customize this algorithm for your library, place the `my_conv_filter` block under a masked subsystem and define `shape` as a mask parameter.

1. Convert the block to a masked subsystem:

   a. Right-click the my_conv_filter block and select **Subsystem & Model Reference > Create Subsystem from Selection**.

      The my_conv_filter block changes to a subsystem block.

   b. Change the name of the subsystem to my_conv_filter.

   c. Right-click the my_conv_filter subsystem and select **Mask > Create Mask** from the context menu.

      The Mask Editor appears with the **Icon & Ports** tab open.

   d. Enter in the **Icon drawing commands** text box:

      ```
      disp('my_conv');
      port_label('output', 1, 'c');
      port_label('input', 1, 'a');
      port_label('input', 2, 'b');
      ```

   e. Select the **Parameters & Dialog** tab.

   f. Highlight the **Parameters** line item in the Dialog box pane.

   g. Add a popup-type parameter by clicking **Popup** under the **Parameter** list in the **Controls** pane.

      A new parameter will appear in the Dialog box pane.

   h. In the **Property editor** pane, set the **Properties**:

Property	Value
Name	`shape`
Value	`full`
Prompt	`shape`
Type	`popup`

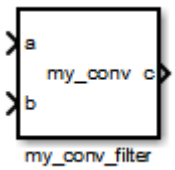
Type options	Open the Type Options Editor and enter:
	
	`full`
	`same`
	`valid`

   i. Set the **Attributes**, **Dialog**, and **Layout** properties in the **Property editor** pane:

Attributes, Dialog, and Layout Items	Value
Attributes	• Evaluate: Checked • Tunable: Cleared • Read only: Cleared • Hidden: Cleared • Never save: Cleared
Dialog	• Enable: Checked • Visible: Checked • Callback: no entry
Layout	• Item location: Grayed out • Prompt location: Left

j Click **OK**.

Your subsystem should now look like this:



2 Set subsystem properties for code reuse:

a Right-click the my_conv_filter subsystem and select **Block Parameters (Subsystem)** from the context menu.

b In the subsystem parameters dialog box, select the **Treat as atomic unit** check box.

The dialog box expands to display new fields.

c To generate a reusable function, select the Code Generation tab and in the **Function packaging** field, select `Reusable function` from the drop-down menu.

Note This is an optional step, required for this example. If you leave the default setting of **Auto**, the code generation software uses an internal rule to determine whether to inline the function or not.

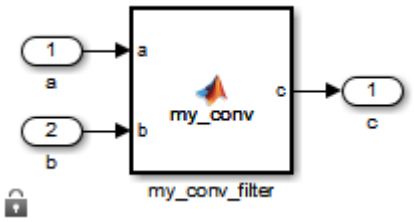
---

d Click **OK**.

3 Define the `shape` parameter in the MATLAB Function `my_conv`:

a Right-click the my_conv_filter subsystem and select **Mask > Look Under Mask** from the context menu.

The block diagram under the masked subsystem opens, containing the my_conv_filter block:



b Change the names of the port blocks to match the data names as follows:

Change:	To:
`In1`	`a`
`In2`	`b`
`Out1`	`c`

c Double-click the my_conv_filter block to open the MATLAB Function Block Editor.

d In the MATLAB Function Block Editor, select **Edit Data**.

e In the Ports and Data Manager, select **Add** > **Data**.

A new data element appears selected, along with its properties dialog.

f Enter the following properties:

Property	What To Specify
Name	Enter `shape`.
Scope	Select **Parameter**.
Tunable	Clear the box.

g Leave **Size**, **Complexity**, and **Type** as inherited (the defaults), as described in “Step 2: Configure Blocks to Inherit Properties You Want to Specialize” on page 44-104.

h Click **Apply**, close the Ports and Data Manager, and return to the MATLAB Function Block Editor.

4 Use the `shape` parameter to determine the size of the convolution to output:

a In the MATLAB Function Block Editor, modify the `my_conv` function to call `conv` with the right shape:

```
function c = my_conv(a, b, shape)
if shape == 1
    c = conv(a, b, 'full');
elseif shape == 2
    c = conv(a, b, 'same');
else
    c = conv(a, b, 'valid');
end
```

b Save your changes and close the MATLAB Function Block Editor.

Step 4: Add Instances of MATLAB Library Blocks to a Simulink Model

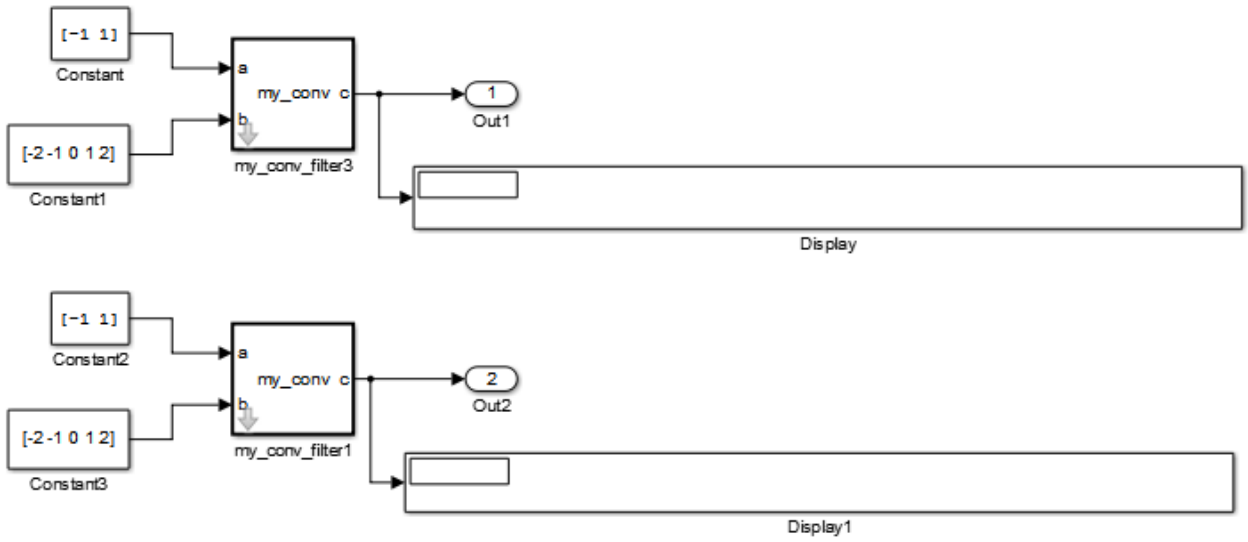
In this exercise, you will add specialized instances of the my_conv_filter library block to a simple test model.

1. Open a new Simulink model.

   For purposes of this exercise, set the following configuration parameters for simulation:

Pane	Section	What to Specify
Solver	**Solver selection**	• Select **Fixed-Step** for **Type** • Select **discrete (no continuous states)** for **Solver** • Enter **1** for **Fixed-step size**
Data Import/Export	**Save options**	**Structure** for **Format**

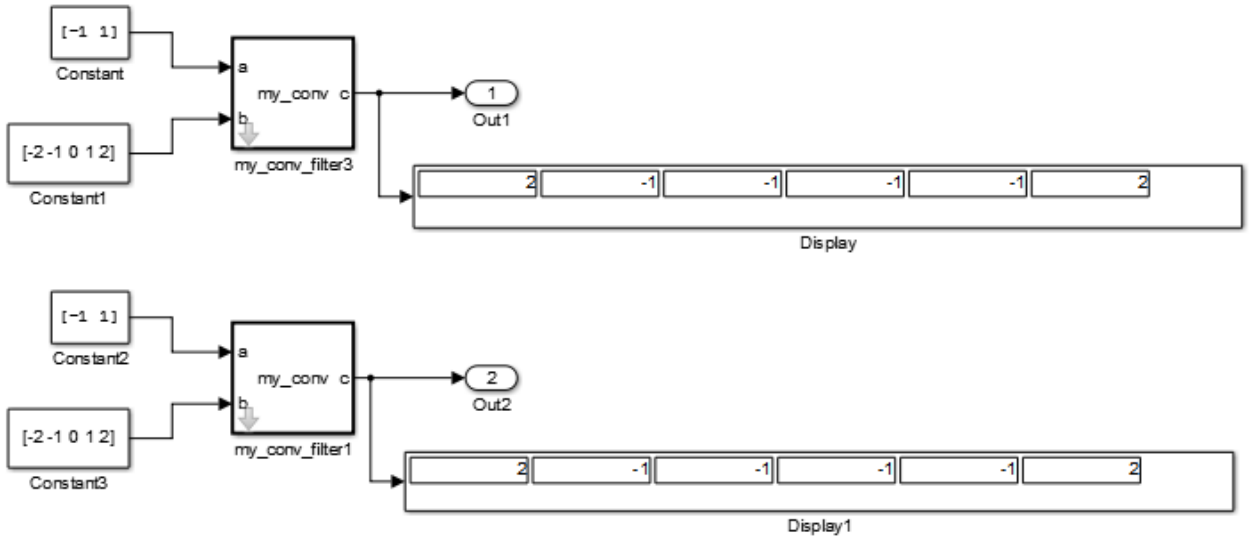
2. Drag two instances of the my_conv_filter block from the `my_filter_lib` library into the model.
3. Add Constant, Outport, and Display blocks. Your model should look something like this:



   Both library instances share the same size, type, and complexity for inputs `a` and `b` respectively.
4. Double-click each library instance.

   The `shape` parameter defaults to **full** for both instances.
5. Simulate the model.

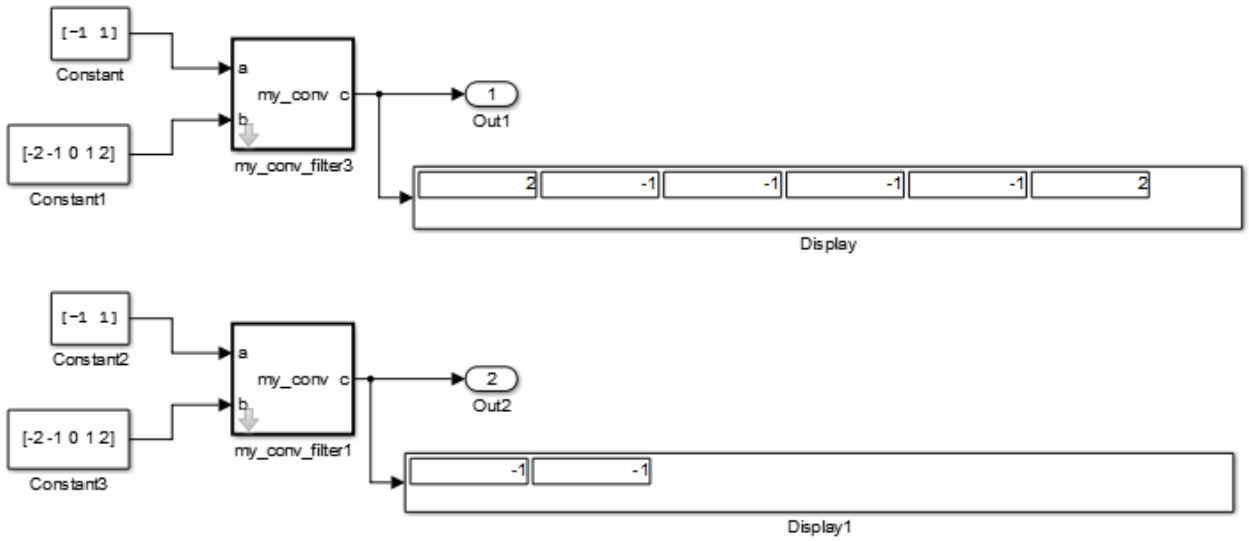
   Each library instance outputs the same result, the full 2D convolution:



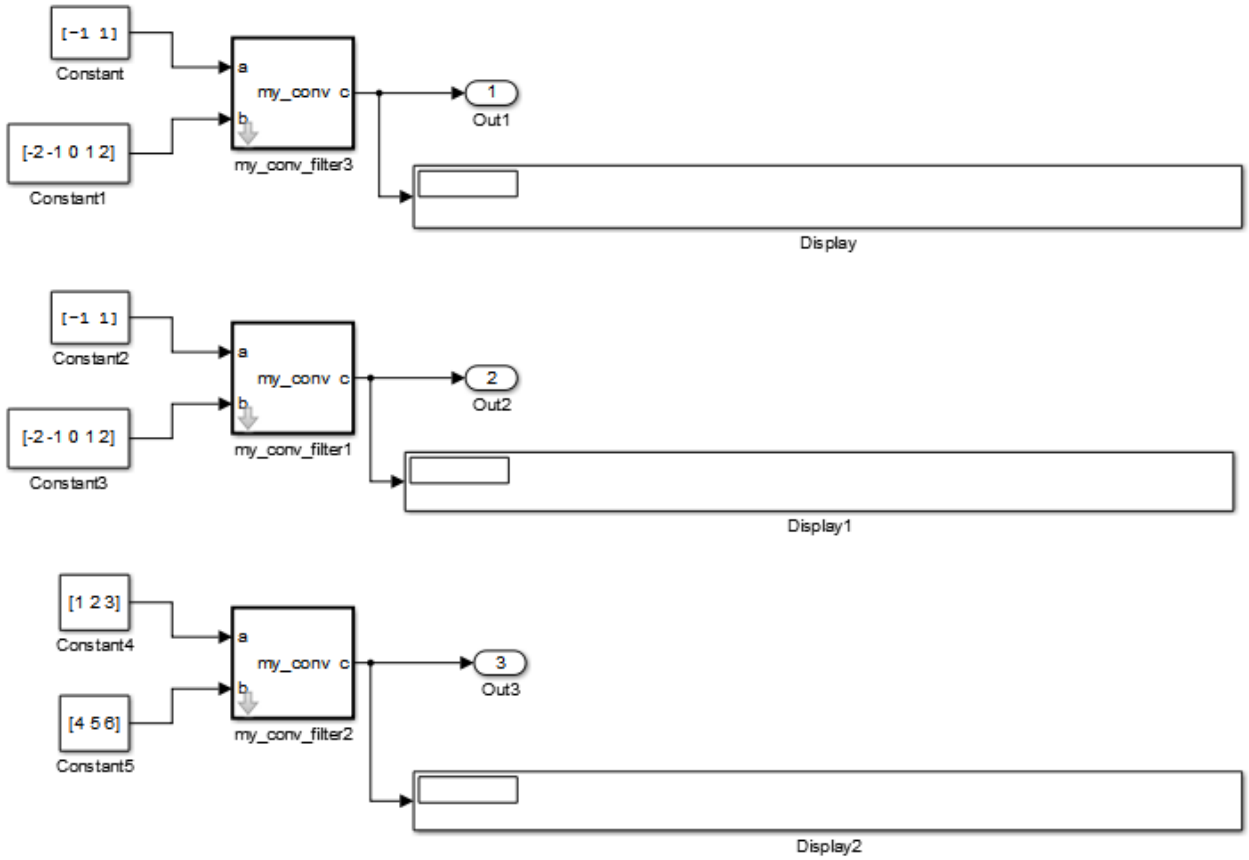
6 Specialize the second instance, my_conv_filter1 by setting the value of its `shape` parameter to **same**.

7 Now simulate the model again.

This time, the outputs have different sizes: my_conv_filter3 outputs the full 2D convolution, while my_conv_filter1 displays the central part of the convolution as a 1-by-2 vector, the same size as `a`:

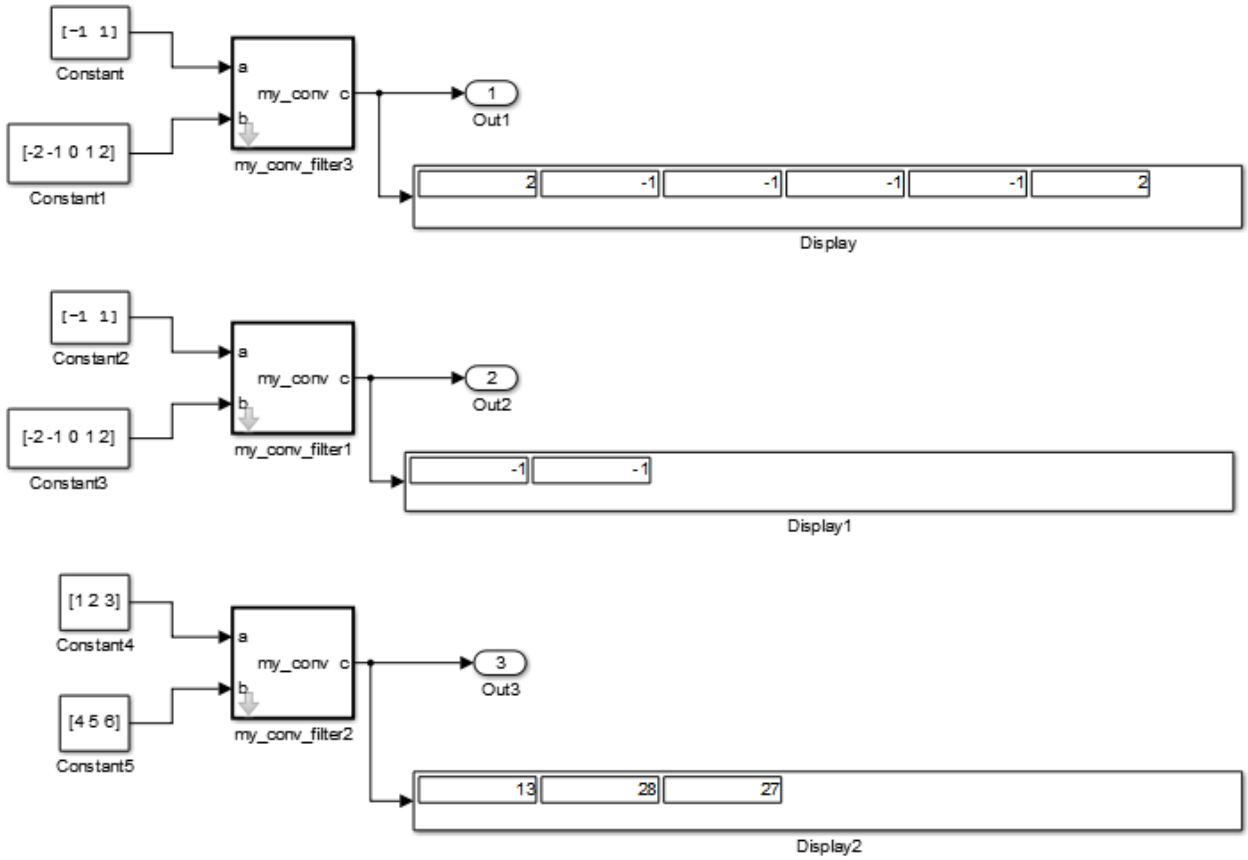


8 Now, add a third instance by copying my_conv_filter1. Specialize the new instance, my_conv_filter2, so that it does not inherit the same size inputs as the first two instances:



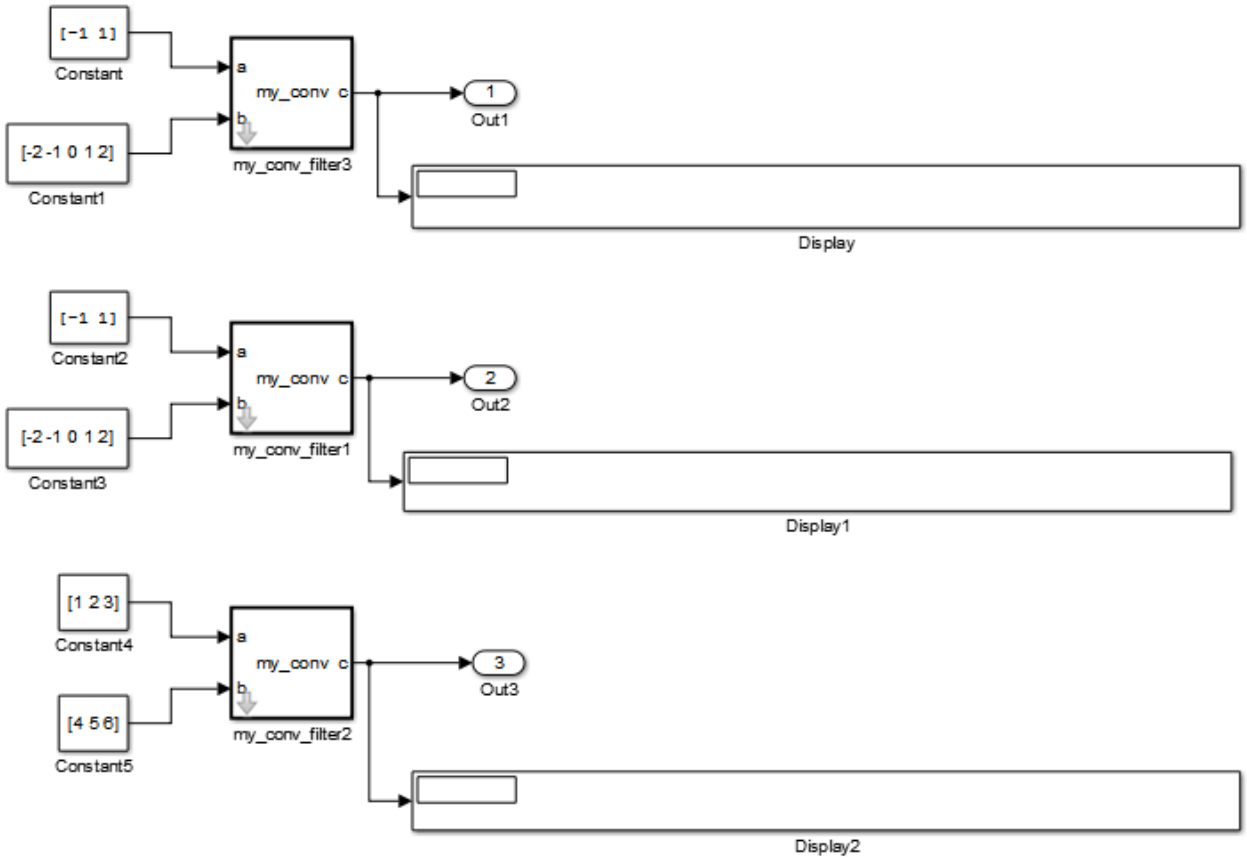
9 Simulate the model again.

This time, my_conv_filter1 and my_conv_filter2 each display the central part of the convolution, but the output sizes are different because each matches a different sized input `a`.



Code Reuse with Library Blocks

When instances of MATLAB Function library blocks inherit the same properties, they can reuse generated code, as illustrated by an example based on “Step 4: Add Instances of MATLAB Library Blocks to a Simulink Model” on page 44-107:



In this model, the library instances my_conv_filter and my_conv_filter1 inherit the same size, type, and complexity for each respective input. For each instance, input `a` is a 1-by-2 vector and input `b` is a 1-by-5 vector. By comparison, the inputs of my_conv_filter2 inherit different respective sizes; both are 1-by-3 vectors.

In addition, each library instance has a mask parameter called shape that determines what subsection of the convolution to output. Assume that the value of shape is the same for each instance.

To generate code for this example, follow these steps:

1. Enable code reuse for the library block:
   a. In the library, right-click the MATLAB Function block my_conv_filter and select **Block Parameters (Subsystem)** from the context menu.
   b. In the Function Block Parameters dialog box, set these parameters:
      - Select the **Treat as atomic unit** check box.
      - In the **Function packaging** field, select `Reusable function` from the drop-down menu.
2. Configure the model for code generation.

   For purposes of this exercise, set the following configuration parameters:

Pane	Section	What to Specify
Code Generation	**Target selection**	Enter **ert.tlc** for `System target file`
Code Generation > **Report**		Select **Create code generation report** check box.

3. Build the model.

   If you build this model, the generated C code reuses logic for the `my_conv_filter` and `my_conv_filter1` library instances because they inherit the same input properties:

```
/*
 * Output and update for atomic system:
 *    '<Root>/my_conv_filter'
 *    '<Root>/my_conv_filter1'
 */
void sp_algorithm_tes_my_conv_filter(const real32_T rtu_a[2], const real32_T
  rtu_b[5], rtB_my_conv_filter_sp_algorithm *localB)
{
  int32_T jA;
  int32_T jA_0;
  real32_T s;
  int32_T jC;

  /* MATLAB Function Block: '<S1>/my_conv_filter' */
  /* MATLAB Function 'my_conv_filter/my_conv_filter': '<S4>:1' */
  /* '<S4>:1:4' */
  for (jC = 0; jC < 6; jC++) {
    if (5 < jC + 2) {
      jA = jC - 4;
    } else {
      jA = 0;
    }

    if (2 < jC + 1) {
      jA_0 = 2;
    } else {
      jA_0 = jC + 1;
    }

    s = 0.0F;
    while (jA + 1 <= jA_0) {
      s += rtu_b[jC - jA] * rtu_a[jA];
      jA++;
    }

    localB->c[jC] = s;
  }

  /* end of MATLAB Function Block: '<S1>/my_conv_filter' */
}
```

   However, a separate function is generated for my_conv_filter2:

```
/* Output and update for atomic system: '<Root>/my_conv_filter2' */
void sp_algorithm_te_my_conv_filter2(const real_T rtu_a[3], const real_T rtu_b[3],
  rtB_my_conv_filter_sp_algorit_h *localB)
{
  int32_T jA;
  int32_T jA_0;
  real_T s;
  int32_T jC;

  /* MATLAB Function Block: '<S3>/my_conv_filter' */
  /* MATLAB Function 'my_conv_filter/my_conv_filter': '<S6>:1' */
  /* '<S6>:1:4' */
  for (jC = 0; jC < 5; jC++) {
    if (3 < jC + 2) {
      jA = jC - 2;
    } else {
      jA = 0;
    }

    if (3 < jC + 1) {
      jA_0 = 3;
    } else {
      jA_0 = jC + 1;
    }

    s = 0.0;
    while (jA + 1 <= jA_0) {
      s += rtu_b[jC - jA] * rtu_a[jA];
      jA++;
    }

    localB->c[jC] = s;
  }

  /* end of MATLAB Function Block: '<S3>/my_conv_filter' */
}
```

Note Generating C code for this model requires a Simulink Coder or Embedded Coder license.

Debugging MATLAB Function Library Blocks

You debug MATLAB Function library blocks the same way you debug any MATLAB Function block. However, when you add a breakpoint in a library block, the breakpoint is shared by all instances. As you continue execution, the debugger stops at the breakpoint in each instance.

Properties You Can Specialize Across Instances of Library Blocks

You can specialize instances of MATLAB Function library blocks by allowing them to inherit any of the following properties from Simulink:

Property	Inherits by Default?	How to Specify Inheritance
Type	Yes	Set data type property to **Inherit: Same as Simulink**.
Size	Yes	Set data size property to **-1**.
Complexity	Yes	Set data complexity property to **Inherited**.
Limit range	No	Specify minimum and maximum values as Simulink parameters. For example, if minimum value = `aParam` and maximum value = `aParam` + 3, different instances of a MATLAB Function library block can resolve to different `aParam` parameters defined in their parent mask subsystems.
Sampling mode (input)	Yes	MATLAB Function block input ports always inherit sampling mode
Data type override mode for fixed-point data	Yes	Set data type override property to **Inherit**.
Sample time (block)	Yes	Set block sample time property to **-1**.

See Also

More About

- “Type Function Arguments” on page 44-45
- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Create a Custom Library” on page 41-2
- “MATLAB Function Block Editor” on page 44-26
- “Masking Fundamentals” on page 39-2
- “Debugging a MATLAB Function Block” on page 44-18

Use Traceability in MATLAB Function Blocks

In this section...

Extent of Traceability in MATLAB Function Blocks

Like other Simulink blocks, MATLAB Function blocks support bidirectional traceability, but extend navigation to lines of source code. That is, you can navigate between a line of generated code and its corresponding line of source code. In other Simulink blocks, you can navigate between a line of generated code and its corresponding object.

In addition, you can select to include the source code as comments in the generated code. When you select **MATLAB source code as comments** parameter, the MATLAB source code appears immediately after the associated traceability tag. For more information, see “How to Include MATLAB Code as Comments in the Generated Code” on page 44-118.

For information about how traceability works in Simulink blocks, see “Verify Generated Code by Using Code Tracing” (Embedded Coder).

Traceability Requirements

To enable traceability comments in your code, you must have a license for Embedded Coder software. These comments appear only in code that you generate for an Embedded Real-Time (ERT) target.

Note Traceability is not supported for MATLAB files that you call from a MATLAB Function block.

Tutorial: Using Traceability in a MATLAB Function Block

This example shows how to trace between source code and generated code in a MATLAB Function block in the `eml_fire` model. Follow these steps:

1. Type `eml_fire` at the MATLAB prompt.
2. In the Simulink model window, in the **Modeling** tab, click **Model Settings**.
3. In the **Code Generation** pane, go to the **Target selection** section and enter `ert.tlc` for the system target file. Then click **Apply**.Traceability comments appear hyperlinked in generated code only for embedded real-time (`ert`) targets.
4. In the **Code Generation > Report** pane, select the **Create code generation report** (Simulink Coder) parameter, if not already selected.

   This action automatically selects the “Open report automatically” (Simulink Coder), “Code-to-model” (Embedded Coder), and “Model-to-code” (Embedded Coder) parameters.
5. Verify that **Code-to-model** and **Model-to-code** parameters are enabled.
6. In the **Code Generation > Comments** pane, select the “MATLAB source code as comments” (Simulink Coder) and “Stateflow object comments” (Simulink Coder) parameters. These

parameters control different parts of the traceability comment. See “Location of Comments in Generated Code” on page 44-118 for more information.

7 Go to the **Code Generation > Interface** pane. In the **Software environment** section, select the **continuous time** parameter. Then click **Apply**. Because this example model contains a block with a continuous sample time, you must perform this step before generating code.

8 In the model window, press **Ctrl+B**.

This action generates source code and header files for the `eml_fire` model that contains the `flame` block. After the code generation process is complete, the code generation report appears automatically.

9 Click the `eml_fire.c` hyperlink in the report.

10 Scroll down through the code to see the traceability comments, which appear as links inside `/*...*/` brackets, as in this example.

```
/* '<S2>:1:19' for x = 1 : WIDTH */
for (x = 0; x < 256; x++) {
  /* '<S2>:1:21' yb = y+2; */
  yb = iU;

  /* '<S2>:1:22' xb1 = x-1; */
  xb1 = x;
```

11 Click the `<S2>:1:19` hyperlink in this traceability comment:

```
/* '<S2>:1:19' */
```

Line 19 of the function in the source code appears highlighted in the MATLAB Function Block Editor.

12 You can trace a line in a MATLAB function to lines of generated code. For example, right-click on line 21 of your function and select **Code Generation > Navigate to Code** from the context menu.

The code location for line 21 appears highlighted in `eml_fire.c`.

13 You can trace a line of generated code to a line of source code in a MATLAB function using the line number hyperlinks in the generated code.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134
- “Include MATLAB Code as Comments in Generated Code” on page 44-118

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Verify Generated Code by Using Code Tracing” (Embedded Coder)

Include MATLAB Code as Comments in Generated Code

If you have a Simulink Coder license, you can include MATLAB source code as comments in the code generated for a MATLAB Function block. Including this information in the generated code enables you to:

- Correlate the generated code with your source code.
- Understand how the generated code implements your algorithm.
- Evaluate the quality of the generated code.

When you select **MATLAB source code as comments** parameter, the generated code includes:

- The source code as a comment immediately after the traceability tag. When you enable traceability and generate code for ERT targets (requires an Embedded Coder license), the traceability tags are hyperlinks to the source code. For more information on traceability for the MATLAB Function block, see “Use Traceability in MATLAB Function Blocks” on page 44-116.

  For examples and information on the location of the comments in the generated code, see “Location of Comments in Generated Code” on page 44-118.
- The function help text in the function body in the generated code. The function help text is the first comment after the MATLAB function signature. It provides information about the capabilities of the function and how to use it.

  Note With an Embedded Coder license, you can also include the function help text in the function banner of the generated code. For more information, see “Including MATLAB user comments in Generated Code” on page 44-120.

How to Include MATLAB Code as Comments in the Generated Code

To include MATLAB source code as comments in the code generated for a MATLAB Function block:

1. In the model, in the **Modeling** tab, click **Model Settings**.
2. In the **Code Generation > Comments** pane, select **MATLAB source code as comments** and click **Apply**.

Location of Comments in Generated Code

The automatically generated comments containing the source code appear after the traceability tag in the generated code as follows.

```
/* '<S2>:1:18' for y = 1 : 2 : (HEIGHT-4) */
```

Selecting the **Stateflow object comments** parameter generates the traceability comment `'<S2>:1:18'`. Selecting the **MATLAB source code as comments** parameter generates the `for y = 1 : 2 : (HEIGHT-4)` comment.

Straight-Line Source Code

The comment containing the source code precedes the generated code that implements the source code statement. This comment appears after any comments that you add that precede the generated

code. The comments are separated from the generated code because the statements are assigned to function outputs.

MATLAB Code

```
function [x y] = straightline(r,theta)
%#codegen
% Convert polar to Cartesian
x = r * cos(theta);
y = r * sin(theta);
```

Commented C Code

```
/* MATLAB Function 'straightline': '<S1>:1' */
        /*  Convert polar to Cartesian */
        /* '<S1>:1:4' x = r * cos(theta); */
        /* '<S1>:1:5' y = r * sin(theta); */
        straightline0_Y.x = straightline0_U.r * cos(straightline0_U.theta);

        /* Outport: '<Root>/y' incorporates:
         *  Inport: '<Root>/r'
         *  Inport: '<Root>/theta'
         *  MATLAB Function Block: '<Root>/straightline'
         */
        straightline0_Y.y = straightline0_U.r * sin(straightline0_U.theta);
```

If Statements

The comment for the `if` statement immediately precedes the code that implements the statement. This comment appears after any comments that you add that precede the generated code. The comments for the `elseif` and `else` clauses appear immediately after the code that implements the clause, and before the code generated for statements in the clause.

MATLAB Code

```
function y = ifstmt(u,v)
%#codegen
if u > v
    y = v + 10;
elseif u == v
    y = u * 2;
else
    y = v - 10;
end
```

Commented C Code

```
/* MATLAB Function 'MLFcn': '<S1>:1' */
        /* '<S1>:1:3' if u > v */
        if (MLFcn_U.u > MLFcn_U.v) {
          /* Outport: '<Root>/y' */
          /* '<S1>:1:4' y = v + 10; */
          MLFcn_Y.y = MLFcn_U.v + 10.0;
        } else if (MLFcn_U.u == MLFcn_U.v) {
          /* Outport: '<Root>/y' */
          /* '<S1>:1:5' elseif u == v */
          /* '<S1>:1:6' y = u * 2; */
          MLFcn_Y.y = MLFcn_U.u * 2.0;
        } else {
```

```
        /* Outport: '<Root>/y' */
        /* '<S1>:1:7' else */
        /* '<S1>:1:8' y = v - 10; */
        MLFcn_Y.y = MLFcn_U.v - 10.0;
```

For Statements

The comment for the `for` statement header immediately precedes the generated code that implements the header. This comment appears after any comments that you add that precede the generated code.

MATLAB Code

```
function y = forstmt(u)
%#codegen
y = 0;
for i=1:u
    y = y + 1;
end
```

Commented C Code

```
/* MATLAB Function 'MLFcn': '<S1>:1' */
        /* '<S1>:1:3' y = 0; */
        rtb_y = 0.0;

        /* '<S1>:1:5' for i=1:u */
        for (i = 1.0; i <= MLFcn_U.u; i++) {
          /* '<S1>:1:6' y = y + 1; */
          rtb_y++;
```

While Statements

The comment for the `while` statement header immediately precedes the generated code that implements the statement header. This comment appears after any comments that you add that precede the generated code.

Switch Statements

The comment for the `switch` statement header immediately precedes the generated code that implements the statement header. This comment appears after any comments that you add that precede the generated code. The comments for the `case` and `otherwise` clauses appear immediately after the generated code that implements the clause, and before the code generated for statements in the clause.

Including MATLAB user comments in Generated Code

MATLAB user comments include the function help text and other comments. The function help text is the first comment after the MATLAB function signature. It provides information about the capabilities of the function and how to use it. You can include the MATLAB user comments in the code generated for a MATLAB Function block.

1. In the model, on the **Modeling** tab, click **Model Settings**.
2. In the **Code Generation > Comments** pane, select “MATLAB user comments” (Embedded Coder) and click **Apply**.

Limitations of MATLAB Source Code as Comments

The MATLAB Function block has the following limitations for including MATLAB source code as comments.

- You cannot include MATLAB source code as comments for:
  - MathWorks toolbox functions
  - P-code
  - Simulation targets
  - Stateflow Truth Table blocks
- The appearance or location of comments can vary depending on the following conditions:
  - Comments might still appear in the generated code even if the implementation code is eliminated, for example, due to constant folding.
  - Comments might be eliminated from the generated code if a complete function or code block is eliminated.
  - For certain optimizations, the comments might be separated from the generated code.
  - The generated code always includes legally required comments from the MATLAB source code, even if you do not choose to include source code comments in the generated code.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134
- “Use Traceability in MATLAB Function Blocks” on page 44-116

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Verify Generated Code by Using Code Tracing” (Embedded Coder)

Integrate C Code Using the MATLAB Function Block

In this section...

Call C Code from a Simulink Model

You can call external C code from a Simulink model using a MATLAB Function block. Follow these high-level steps:

1. Start with existing C code consisting of the source (`.c`) and header (`.h`) files.
2. In the MATLAB Function block, enter the MATLAB code that calls the C code. Use the `coder.ceval` function. To pass data by reference, use `coder.ref`, `coder.rref`, or `coder.wref`.
3. Specify the C source and header files for simulation in the **Simulation Target** pane of the **Model Configuration Parameters** dialog box. Include the header file using double quotations, for example, `#include "program.h"`. If you need to access C source and header files outside your working folder, list the path in the **Simulation Target** pane, in the **Include Directories** text box.

   Alternatively, use the `coder.cinclude` and `coder.updateBuildInfo` functions to specify source and header files within your MATLAB code. To develop an interface to external code, you can use the `coder.ExternalDependency` class. To see which workflow is supported, see “Import custom code”.
4. Test your Simulink model and ensure it functions correctly.
5. If you have a Simulink Coder license, you can generate code for targets using this method. To use the same source and header files for code generation, click **Use the same custom code settings as Simulation Target** in the **Code Generation > Custom Code** pane. You can also specify different source and header files.

   To conditionalize your code to execute different commands for simulation and code generation, you can use the `coder.target` function.

Use coder.ceval in a MATLAB Function Block

This example shows how to call the simple C program `doubleIt` from a MATLAB Function block.

1. Create the source file `doubleIt.c` in your current working folder.

```
#include "doubleIt.h"

double doubleIt(double u)
{
     return(u*2.0);
}
```

2. Create the header file `doubleIt.h` in your current working folder.

```
#ifndef MYFN
#define MYFN
```

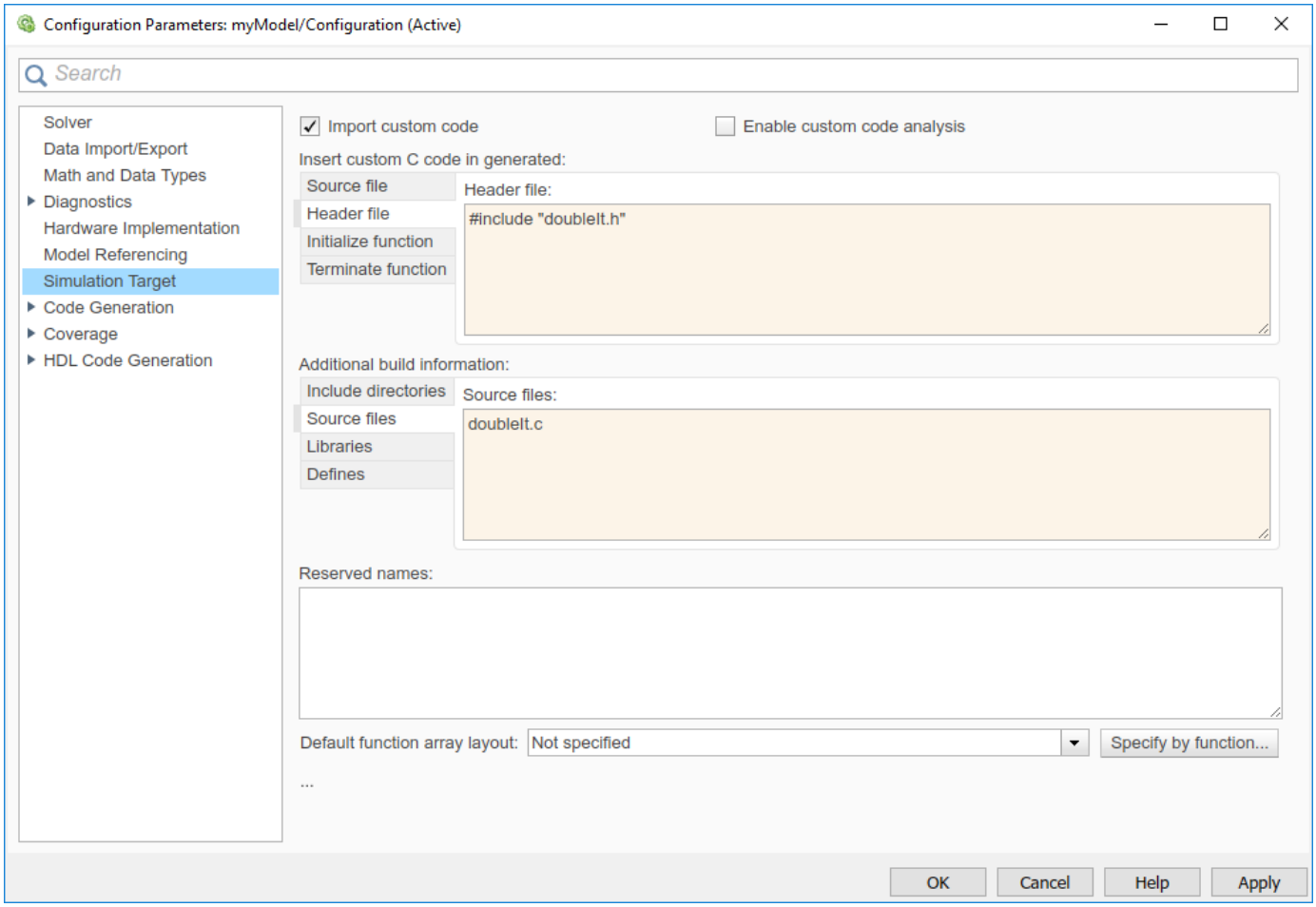
```
double doubleIt(double u);

#endif
```

3. Create a new Simulink model. Save it as `myModel`.
4. In the **Library Browser**, from **User-Defined Functions**, add a MATLAB Function block to the model and double-click the block to open the editor.
5. Enter code that calls the `doubleIt` program:

```
function y = callingDoubleIt(u)

y = 0.0;
y = coder.ceval('doubleIt',u);
```

6. Connect a Constant block having a value of `3.5` to the input port of the MATLAB Function block.
7. Connect a Display block to the output port.

8. In the **Model Configuration Parameters** dialog box, open the **Simulation Target** pane.
9. In the **Insert custom C code in generated** section, select **Header file** from the list, and enter `#include "doubleIt.h"` in the **Header file** text box.
10. In the **Additional build information** section, select **Source files** from the list, enter `doubleIt.c` in the **Source files** text box, and click **OK**.



11 Run the simulation. The value 7 appears in the Display block.

Control Imported Bus and Enumeration Type Definitions

This procedure applies to simulation only.

Simulink generates code for MATLAB Function blocks and Stateflow to simulate the model. When you call external C code using MATLAB Function blocks or Stateflow, you can control the type definitions for imported buses and enumerations in model simulation.

Simulink can generate type definitions, or you can supply a header file containing the type definitions. You control this behavior using the **Generate typedefs for imported bus and enumeration types** check box in the **Model Configuration Parameters** dialog box.

To include a custom header file defining the enumeration and bus types:

1 Clear the **Generate typedefs for imported bus and enumeration types** check box.

2 List the header file in the **Simulation Target** pane, in the **Header file** text box.

To configure Simulink to automatically generate type definitions:

1 Select the **Generate typedefs for imported bus and enumeration types** check box.

2 Do not list a header file that corresponds to the buses or enumerations.

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.cinclude` | `coder.ref` | `coder.rref` | `coder.target` | `coder.updateBuildInfo` | `coder.wref`

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Add Code to a MATLAB Function Block Programmatically” on page 44-10

More About

- “Model Configuration Parameters: Simulation Target”
- “Implementing MATLAB Functions Using Blocks” on page 44-4

Enhance Code Readability for MATLAB Function Blocks

In this section...

Requirements for Using Readability Optimizations

To use readability optimizations in your code, you must have an Embedded Coder license. These optimizations appear only in code that you generate for an embedded real-time (`ert`) target.

Note These optimizations do not apply to MATLAB files that you call from the MATLAB Function block.

For more information, see “Configure a System Target File” (Embedded Coder) and “Model Configuration Parameters: Code Style” (Embedded Coder).

Converting If-Elseif-Else Code to Switch-Case Statements

When you generate code for embedded real-time targets, you can choose to convert `if-elseif-else` decision logic to `switch-case` statements. This conversion can enhance readability of the code.

For example, when a MATLAB Function block contains a long list of conditions, the `switch-case` structure:

- Reduces the use of parentheses and braces
- Minimizes repetition in the generated code

How to Convert If-Elseif-Else Code to Switch-Case Statements

The following procedure describes how to convert generated code for the MATLAB Function block from `if-elseif-else` to `switch-case` statements.

Step	Task	Reference
1	Verify that your block follows the rules for conversion.	“Verifying the Contents of the Block” on page 44-129
2	Enable the conversion.	“Enabling the Conversion” on page 44-130
3	Generate code for your model.	“Generating Code for Your Model” on page 44-130

Rules for Conversion

For the conversion to occur, the following rules must hold. LHS and RHS refer to the left-hand side and right-hand side of a condition, respectively.

Construct	Rules to Follow
MATLAB Function block	Must have two or more *unique* conditions, in addition to a default. For more information, see “How the Conversion Handles Duplicate Conditions” on page 44-127.
Each condition	Must test equality only.
	Must use the same variable or expression for the LHS. **Note** You can reverse the LHS and RHS.
Each LHS	Must be a single variable or expression, not a compound statement.
	Cannot be a constant.
	Must have an integer or enumerated data type.
	Cannot have any side effects on simulation. For example, the LHS can read from but not write to global variables.
Each RHS	Must be a constant.
	Must have an integer or enumerated data type.

How the Conversion Handles Duplicate Conditions

If a MATLAB Function block has duplicate conditions, the conversion preserves only the first condition. The generated code discards all other instances of duplicate conditions.

After removal of duplicates, two or more unique conditions must exist. Otherwise, no conversion occurs and the generated code contains all instances of duplicate conditions.

The following examples show how the conversion handles duplicate conditions.

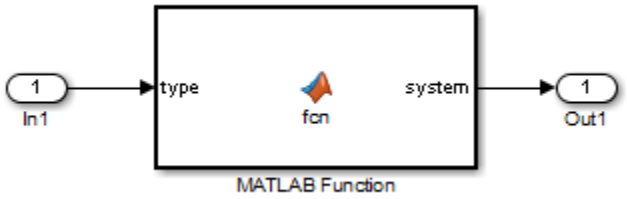
Example of Generated Code	Code After Conversion

`if (x == 1) {`
`  block1`
`} else if (x == 2) {`
`  block2`
`} else if (x == 1) {  // duplicate`
`  block3`
`} else if (x == 3) {`
`  block4`
`} else if (x == 1) {  // duplicate`
`  block5`
`} else {`
`  block6`
`}`
	`switch (x) {`
	`  case 1:`
	`  block1; break;`
	`  case 2:`
	`  block2; break;`
	`  case 3:`
	`  block4; break;`
	`  default:`
	`  block6; break;`
	`}`

`if (x == 1) {`
`  block1`
`} else if (x == 1) {  // duplicate`
`  block2`
`} else {`
`  block3`
`}`
	No change, because only one unique condition exists

Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements

Suppose that you have the following model with a MATLAB Function block. Assume that the output data type is `double` and the input data type is `Controller`, an enumerated type that you define. (For more information, see “Code Generation for Enumerations” on page 44-84.)



The block contains the following code:

```
function system = fcn(type)
%#codegen

if (type == Controller.P)
    system = 0;
elseif (type == Controller.I)
    system = 1;
elseif (type == Controller.PD)
    system = 2;
elseif (type == Controller.PI)
    system = 3;
elseif (type == Controller.PID)
    system = 4;
else
    system = 10;
end
```

The enumerated type definition in `Controller.m` is:

```
classdef Controller < Simulink.IntEnumType
  enumeration
    P(0)
    I(1)
    PD(2)
    PI(3)
    PID(4)
    UNKNOWN(10)
  end
end
```

If you generate code for an embedded real-time target using default settings, you see something like this:

```
if (if_to_switch_eml_blocks_U.In1 == P) {
  /* '<S1>:1:4' */
  /* '<S1>:1:5' */
  if_to_switch_eml_blocks_Y.Out1 = 0.0;
} else if (if_to_switch_eml_blocks_U.In1 == I) {
  /* '<S1>:1:6' */
```

```
  /* '<S1>:1:7' */
  if_to_switch_eml_blocks_Y.Out1 = 1.0;
} else if (if_to_switch_eml_blocks_U.In1 == PD) {
  /* '<S1>:1:8' */
  /* '<S1>:1:9' */
  if_to_switch_eml_blocks_Y.Out1 = 2.0;
} else if (if_to_switch_eml_blocks_U.In1 == PI) {
  /* '<S1>:1:10' */
  /* '<S1>:1:11' */
  if_to_switch_eml_blocks_Y.Out1 = 3.0;
} else if (if_to_switch_eml_blocks_U.In1 == PID) {
  /* '<S1>:1:12' */
  /* '<S1>:1:13' */
  if_to_switch_eml_blocks_Y.Out1 = 4.0;
} else {
  /* '<S1>:1:15' */
  if_to_switch_eml_blocks_Y.Out1 = 10.0;
}
```

The LHS variable `if_to_switch_eml_blocks_U.In1` appears multiple times in the generated code.

Note By default, variables that appear in the block do not retain their names in the generated code. Modified identifiers guarantee that no naming conflicts occur.

Traceability comments appear between each set of `/*` and `*/` markers. To learn more about traceability, see “Use Traceability in MATLAB Function Blocks” on page 44-116.

Verifying the Contents of the Block

Check that the block follows all the rules in “Rules for Conversion” on page 44-126.

Construct	How the Construct Follows the Rules
MATLAB Function block	Five unique conditions exist, in addition to the default: • `(type == Controller.P)` • `(type == Controller.I)` • `(type == Controller.PD)` • `(type == Controller.PI)` • `(type == Controller.PID)`
Each condition	Each condition: • Tests equality • Uses the same input for the LHS

Construct	How the Construct Follows the Rules
Each LHS	Each LHS: • Contains a single variable • Is the input to the block and therefore not a constant • Is of enumerated type `Controller`, which you define in `Controller.m` on the MATLAB path • Has no side effects on simulation
Each RHS	Each RHS: • Is an enumerated value and therefore a constant • Is of enumerated type `Controller`

Enabling the Conversion

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** pane, select `ert.tlc` for the **System target file**.

   This step specifies an embedded real-time target for your model.
3. In the **Code Generation > Code Style** pane, select the **Convert if-elseif-else patterns to switch-case statements** check box.

   Tip This conversion works on a per-model basis. If you select this check box, the conversion applies to:

   - All MATLAB Function blocks in a model
   - MATLAB functions in all Stateflow charts of that model
   - Flow charts in all Stateflow charts of that model

   For more information, see “Enhance Readability of Code for Flow Charts” (Embedded Coder).

Generating Code for Your Model

In the model window, press **Ctrl+B**.

The code for the MATLAB Function block uses `switch-case` statements instead of `if-elseif-else` code:

```
switch (if_to_switch_eml_blocks_U.In1) {
 case P:
  /* '<S1>:1:4' */
  /* '<S1>:1:5' */
  if_to_switch_eml_blocks_Y.Out1 = 0.0;
  break;

 case I:
  /* '<S1>:1:6' */
  /* '<S1>:1:7' */
  if_to_switch_eml_blocks_Y.Out1 = 1.0;
  break;
```

```
 case PD:
  /* '<S1>:1:8' */
  /* '<S1>:1:9' */
  if_to_switch_eml_blocks_Y.Out1 = 2.0;
  break;

 case PI:
  /* '<S1>:1:10' */
  /* '<S1>:1:11' */
  if_to_switch_eml_blocks_Y.Out1 = 3.0;
  break;

 case PID:
  /* '<S1>:1:12' */
  /* '<S1>:1:13' */
  if_to_switch_eml_blocks_Y.Out1 = 4.0;
  break;

 default:
  /* '<S1>:1:15' */
  if_to_switch_eml_blocks_Y.Out1 = 10.0;
  break;
}
```

The `switch-case` statements provide the following benefits to enhance readability:

- The code reduces the use of parentheses and braces.
- The LHS variable `if_to_switch_eml_blocks_U.In1` appears only once, minimizing repetition in the code.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Use Traceability in MATLAB Function Blocks” on page 44-116
- “Code Generation for Enumerations” on page 44-84

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

Control Run-Time Checks

In this section...

Types of Run-Time Checks

In simulation, the code generated for your MATLAB Function block includes the following run-time checks:

- Memory integrity checks

  These checks detect violations of memory integrity in code generated for MATLAB Function blocks and stop execution with a diagnostic message.

  Caution For safety, these checks are enabled by default. Without memory integrity checks, violations result in unpredictable behavior.

- Responsiveness checks in code generated for MATLAB Function blocks

  These checks enable periodic checks for Ctrl+C breaks in the generated code. Enabling responsiveness checks also enables graphics refreshing.

  Caution For safety, these checks are enabled by default. Without these checks, the only way to end a long-running execution might be to terminate MATLAB.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more lines of generated code and slower simulation than generating code with the checks disabled. Disabling run-time checks usually results in streamlined generated code and faster simulation, with these caveats:

Consider disabling:	Only if:
Memory integrity checks	You are sure that your code is safe and that all array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C.

How to Disable Run-Time Checks

MATLAB Function blocks enable run-time checks by default, but you can disable them explicitly for all MATLAB Function blocks in your Simulink model. Follow these steps:

1 Open your MATLAB Function block.

2. In the MATLAB Function Block Editor, select **Simulation Target**.
3. In the Configuration Parameters dialog box, clear the **Ensure memory integrity** or **Ensure responsiveness** check boxes, as applicable, and click **Apply**.

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26

Track Object Using MATLAB Code

In this section...

Learning Objectives

In this tutorial, you will learn how to:

- Use the MATLAB Function block to add MATLAB functions to Simulink models for modeling, simulation, and deployment to embedded processors.

  This capability is useful for coding algorithms that are better stated in the textual language of MATLAB than in the graphical language of Simulink.
- Use `coder.extrinsic` to call MATLAB code from a MATLAB Function block.

  This capability allows you to do rapid prototyping. You can call existing MATLAB code from Simulink without having to make this code suitable for code generation.
- Check that existing MATLAB code is suitable for code generation before generating code.

  You must prepare your code before generating code.
- Specify variable-size inputs when generating code.

Tutorial Prerequisites

- “What You Need to Know” on page 44-134
- “Required Products” on page 44-134

What You Need to Know

To complete this tutorial, you should have basic familiarity with MATLAB software. You should also understand how to create and simulate a basic Simulink model.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- Simulink
- Simulink Coder

- C compiler

For a list of supported compilers, see `https://www.mathworks.com/support/compilers/current_release/`.

You must set up the C compiler before generating C code. See “Setting Up Your C Compiler” on page 44-139.

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

Example: The Kalman Filter

Description

This section describes the example used by the tutorial. You do not have to be familiar with the algorithm to complete the tutorial.

The example for this tutorial uses a Kalman filter to estimate the position of an object moving in a two-dimensional space from a series of noisy inputs based on past positions. The position vector has two components, `x` and `y`, indicating its horizontal and vertical coordinates.

Kalman filters have a wide range of applications, including control, signal and image processing; radar and sonar; and financial modeling. They are recursive filters that estimate the state of a linear dynamic system from a series of incomplete or noisy measurements. The Kalman filter algorithm relies on the state-space representation of filters and uses a set of variables stored in the state vector to characterize completely the behavior of the system. It updates the state vector linearly and recursively using a state transition matrix and a process noise estimate.

Algorithm

This section describes the algorithm of the Kalman filter and is implemented in the MATLAB version of the filter supplied with this tutorial.

The algorithm predicts the position of a moving object based on its past positions using a Kalman filter estimator. It estimates the present position by updating the Kalman state vector, which includes the position (*x* and *y*), velocity (*Vx* and *Vy*), and acceleration (*Ax* and *Ay*) of the moving object. The Kalman state vector, `x_est`, is a persistent variable.

```
% Initial conditions
persistent x_est p_est
if isempty(x_est)
    x_est = zeros(6, 1);
    p_est = zeros(6, 6);
end
```

`x_est` is initialized to an empty `6x1` column vector and updated each time the filter is used.

The Kalman filter uses the laws of motion to estimate the new state:

$$X = X_0 + Vx \,.\, dt$$
$$Y = Y_0 + Vy \,.\, dt$$
$$Vx = Vx_0 + Ax \,.\, dt$$
$$Vy = Vy_0 + Ay \,.\, dt$$

These laws of motion are captured in the state transition matrix `A`, which is a matrix that contains the coefficient values of `x`, `y`, V_x, V_y, A_x, and A_y.

```
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

Filtering Process

The filtering process has two phases:

- Predicted state and covariance

  The Kalman filter uses the previously estimated state, `x_est`, to predict the current state, `x_prd`. The predicted state and covariance are calculated in:

  ```
  % Predicted state and covariance
  x_prd = A * x_est;
  p_prd = A * p_est * A' + Q;
  ```

- Estimation

  The filter also uses the current measurement, `z`, and the predicted state, `x_prd`, to estimate a more accurate approximation of the current state. The estimated state and covariance are calculated in:

  ```
  % Measurement matrix
  H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
  Q = eye(6);
  R = 1000 * eye(2);

  % Estimation
  S = H * p_prd' * H' + R;
  B = H * p_prd';
  klm_gain = (S \ B)';

  % Estimated state and covariance
  x_est = x_prd + klm_gain * (z - H * x_prd);
  p_est = p_prd - klm_gain * H * p_prd;

  % Compute the estimated measurements
  y = H * x_est;
  ```

Reference

Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Files for the Tutorial

About the Tutorial Files

The tutorial uses the following files:

- Simulink model files for each step of the tutorial.
- Example MATLAB code files for each step of the tutorial.

  Throughout this tutorial, you work with Simulink models that call MATLAB files containing a Kalman filter algorithm.
- A MAT-file that contains example input data.
- A MATLAB file for plotting.

Location of Files

The tutorial files are available in the following folder: `docroot\toolbox\simulink\examples\kalman`. To run the tutorial, you must copy these files to a local folder. For instructions, see “Copying Files Locally” on page 44-138.

Names and Descriptions of Files

Type	Name	Description
MATLAB function files	`ex_kalman01`	Baseline MATLAB implementation of a scalar Kalman filter.
	`ex_kalman02`	Version of the original algorithm suitable for code generation.
	`ex_kalman03`	Version of Kalman filter suitable for code generation and for use with frame-based and packet-based inputs.
	`ex_kalman04`	Disabled inlining for code generation.
Simulink model files	`ex_kalman00`	Simulink model without a MATLAB Function block.
	`ex_kalman11`	Complete Simulink model with a MATLAB Function block for scalar Kalman filter.
	`ex_kalman22`	Simulink model with a MATLAB Function block for a Kalman filter that accepts fixed-size (frame-based) inputs.

Type	Name	Description
	ex_kalman33	Simulink model with a MATLAB Function block for a Kalman filter that accepts variable-size (packet-based) inputs.
	ex_kalman44	Simulink model to call ex_kalman04.m, which has inlining disabled.
MATLAB data file	position	Contains the input data used by the algorithm.
Plot files	plot_trajectory	Plots the trajectory of the object and the Kalman filter estimated position.

Tutorial Steps

Copying Files Locally

Copy the tutorial files to a local working folder:

1. Create a local *solutions* folder, for example, c:\simulink\kalman\solutions.
2. Change to the docroot\toolbox\simulink\examples folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'simulink', 'examples'))
   ```
3. Copy the contents of the kalman subfolder to your local *solutions* folder, specifying the full path name of the *solutions* folder:

   ```
   copyfile('kalman', 'solutions')
   ```

   For example:

   ```
   copyfile('kalman', 'c:\simulink\kalman\solutions')
   ```

   Your *solutions* folder now contains a complete set of solutions for the tutorial. If you do not want to perform the steps for each task in the tutorial, you can view the solutions to see how the code should look.
4. Create a local *work* folder, for example, c:\simulink\kalman\work.
5. Copy the following files from your *solutions* folder to your *work* folder.

   - ex_kalman01

- `ex_kalman00`
- `position`
- `plot_trajectory`

Your *`work`* folder now contains all the files that you need to get started with the tutorial.

Setting Up Your C Compiler

Building your MATLAB Function block requires a supported compiler. MATLAB automatically selects one as the default compiler. If you have multiple MATLAB-supported compilers installed on your system, you can change the default using the `mex -setup` command. See “Change Default Compiler”.

About the ex_kalman00 Model

First, examine the `ex_kalman00` model supplied with the tutorial to understand the problem that you are trying to solve using the Kalman filter.

1. Open the `ex_kalman00` model in Simulink:

   a. Set your MATLAB current folder to the folder that contains your working files for this tutorial. At the MATLAB command line, enter:

      ```
      cd work
      ```

      where *`work`* is the full path name of the folder containing your files.

   b. At the MATLAB command line, enter:

      ```
      ex_kalman00
      ```

      This model is an incomplete model to demonstrate how to integrate MATLAB code with Simulink. The complete model is `ex_kalman11`, which is also supplied with this tutorial.

InitFcn Model Callback Function

The model uses this callback function to:

- Load position data from a MAT-file.
- Set up data used by the Index generator block, which provides the second input to the Selector block.

To view this callback:

1. On the **Modeling** tab, select **Model Settings** > **Model Properties**.
2. Select the **Callbacks** tab.
3. Select `InitFcn` in the **Model callbacks** pane.

   The callback appears.

   ```
   load position.mat;
   [R,C]=size(position);
   idx=(1:C)';
   t=idx-1;
   ```

Source Blocks

The model uses two Source blocks to provide position data and a scalar index to a Selector block.

Selector Block

The model uses a Selector block that selects elements of its input signal and generates an output signal based on its index input and its **Index Option** settings. By changing the configuration of this block, you can generate different size signals.

To view the Selector block settings, double-click the Selector block to view the function block parameters.

In this model, the **Index Option** for the first port is `Select all` and for the second port is `Index vector (port)`. Because the input is a `2 x 310` position matrix, and the index data increments from `1` to `310`, the Selector block simply outputs one `2x1` output at each sample time.

MATLAB Function Block

The model uses a MATLAB Function block to plot the trajectory of the object and the Kalman filter estimated position. This function:

- First declares the `figure`, `hold`, and `plot_trajectory` functions as extrinsic because these MATLAB visualization functions are not supported for code generation. When you call an unsupported MATLAB function, you must declare it to be extrinsic so MATLAB can execute it, but does not try to generate code for it.
- Creates a figure window and holds it for the duration of the simulation. Otherwise a new figure window appears for each sample time.
- Calls the `plot_trajectory` function, which plots the trajectory of the object and the Kalman filter estimated position.

Simulation Stop Time

The simulation stop time is `309`, because the input to the filter is a vector containing `310` elements and Simulink uses zero-based indexing.

Adding a MATLAB Function Block to Your Model

To modify the model and code yourself, work through the exercises in this section. Otherwise, open the supplied model `ex_kalman11` in your *`solutions`* subfolder to see the modified model.

For the purposes of this tutorial, you add the MATLAB Function block to the `ex_kalman00.mdl` model supplied with the tutorial. You would have to develop your own test bench starting with an empty Simulink model.

Adding the MATLAB Function Block

To add a MATLAB Function block to the `ex_kalman00` model:

1. Open `ex_kalman00` in Simulink.

   ```
   ex_kalman00
   ```

2. Add a MATLAB Function block to the model:

   a. At the MATLAB command line, type `slLibraryBrowser` to open the Simulink Library Browser.

b From the list of Simulink libraries, select the `User-Defined Functions` library.

c Click the MATLAB Function block and drag it into the `ex_kalman00` model. Place the block just above the red text annotation that reads `Place MATLAB Function Block here.`

d Delete the red text annotations from the model.

e Save the model in the current folder as `ex_kalman11`.

Calling Your MATLAB Code from the MATLAB Function Block

To call your MATLAB code from the MATLAB Function block:

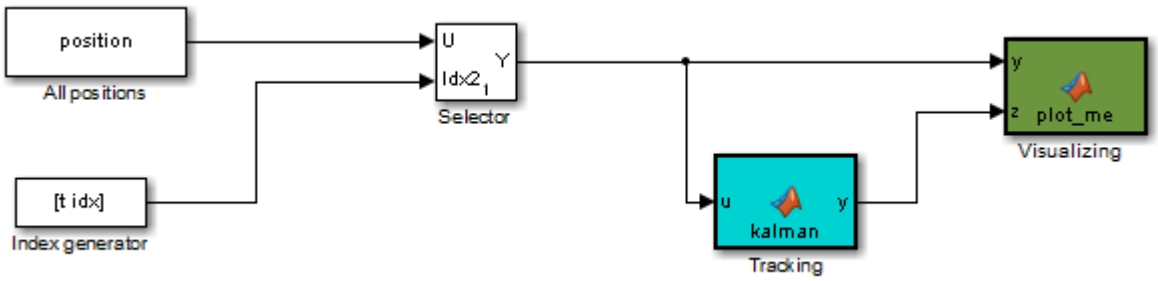
1. Double-click the MATLAB Function block to open the MATLAB Function Block Editor.
2. Delete the default code displayed in the editor.
3. Copy the following code to the MATLAB Function block.

```
function y = kalman(u)
%#codegen

y = ex_kalman01(u);
```

4. Save the model.

Connecting the MATLAB Function Block Input and Output

1. Connect the MATLAB Function block input and output so that your model looks like this.



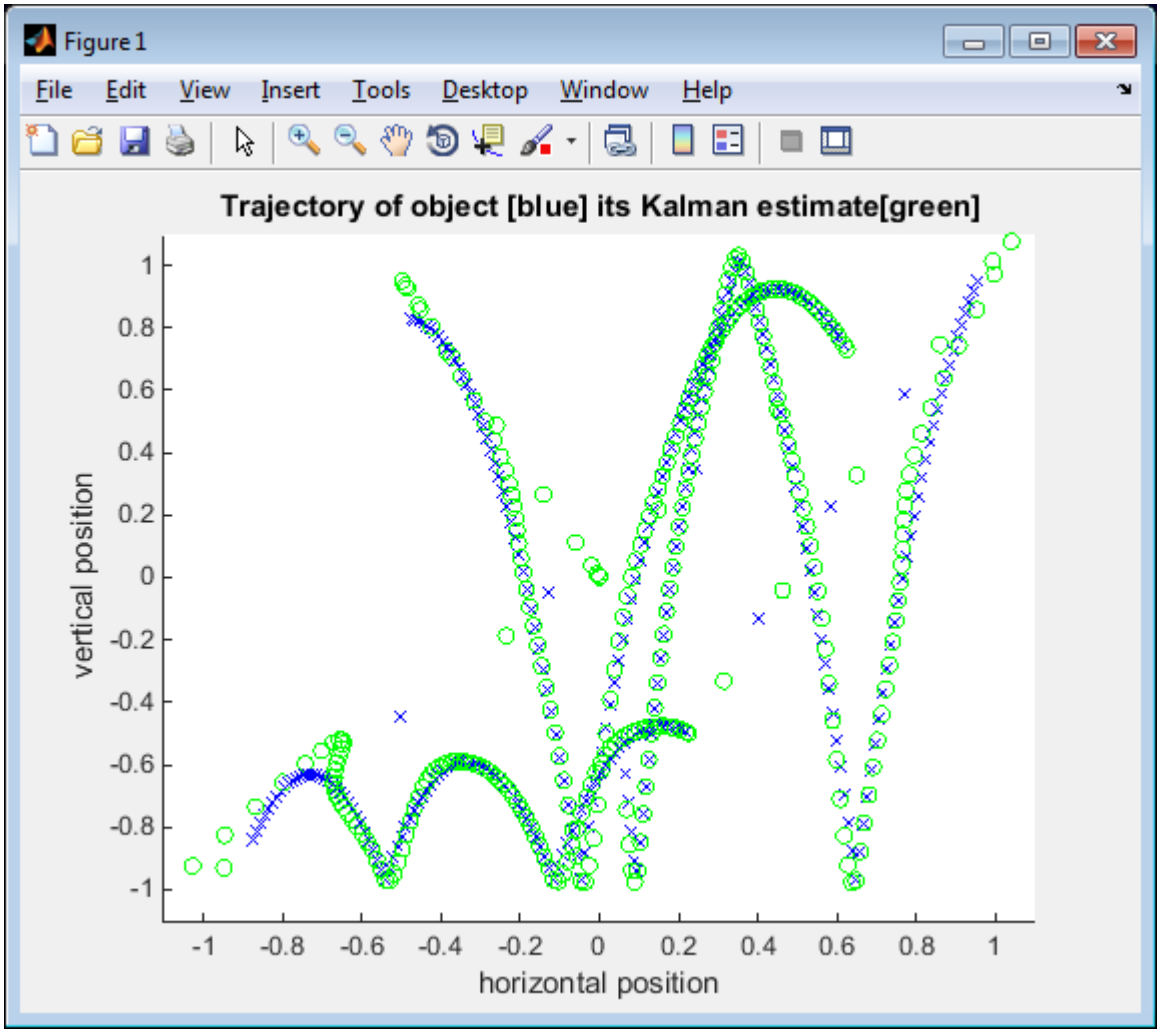
2. Save the model.

Simulating the ex_kalman11 Model

To simulate the model:

1. In the Simulink model window, click **Run**.

   As Simulink runs the model, it plots the trajectory of the object in blue and the Kalman filter estimated position in green. Initially, you see that it takes a short time for the estimated position to converge with the actual position of the object. Then three sudden shifts in position occur—each time the Kalman filter readjusts and tracks the object after a few iterations.



2 The simulation stops.

You have proved that your MATLAB algorithm works in Simulink. You are now ready to modify the filter to accept a fixed-size input, as described in “Modifying the Filter to Accept a Fixed-Size Input” on page 44-142.

Modifying the Filter to Accept a Fixed-Size Input

The filter you have worked on so far in this tutorial uses a simple batch process that accepts one input at a time, so you must call the function repeatedly for each input. In this part of the tutorial, you learn how to modify the algorithm to accept a fixed-sized input, which makes the algorithm suitable for frame-based processing. You then modify the model to provide the input as fixed-size frames of data and call the filter passing in the data one frame at a time.

Modifying Your MATLAB Code

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `ex_kalman03.m` in your *`solutions`* subfolder to see the modified algorithm.

You can now modify the algorithm to process a vector containing more than one input. You need to find the length of the vector and call the filter code for each element in the vector in turn. You do this by calling the filter algorithm in a `for` loop.

1. Open `ex_kalman02.m` in the MATLAB Editor. At the MATLAB command line, enter:

   ```
   edit ex_kalman02.m
   ```

2. Add a `for` loop around the filter code.

   a. Before the comment:

      ```
      % Predicted state and covariance
      ```

      insert:

      ```
      for i=1:size(z,2)
      ```

   b. After:

      ```
      % Compute the estimated measurements
      y = H * x_est;
      ```

      insert:

      ```
      end
      ```

   c. Select the code between the `for` statement and the end statement, right-click to open the context menu and select **Smart Indent** to indent the code.

   Your filter code should now look like this:

   ```
   for i=1:size(z,2)
        % Predicted state and covariance
        x_prd = A * x_est;
        p_prd = A * p_est * A' + Q;

        % Estimation
        S = H * p_prd' * H' + R;
        B = H * p_prd';
        klm_gain = (S \ B)';

        % Estimated state and covariance
        x_est = x_prd + klm_gain * (z - H * x_prd);
        p_est = p_prd - klm_gain * H * p_prd;

        % Compute the estimated measurements
        y = H * x_est;
   end
   ```

3. Modify the line that calculates the estimated state and covariance to use the i^{th} element of input `z`.

   Change:

   ```
   x_est = x_prd + klm_gain * (z - H * x_prd);
   ```

   to:

   ```
   x_est = x_prd + klm_gain * (z(1:2,i) - H * x_prd);
   ```

4. Modify the line that computes the estimated measurements to append the result to the `i`th element of the output `y`.

   Change:

   ```
   y = H * x_est;
   ```

   to:

   ```
   y(:,i) = H * x_est;
   ```

   The code analyzer message indicator in the top right turns orange to indicate that the code analyzer has detected warnings. The code analyzer underlines the offending code in orange and places a orange marker to the right.

5. Move your pointer over the orange marker to view the error information.

   The code analyzer detects that `y` must be fully defined before sub-scripting it and that you cannot grow variables through indexing in generated code.

6. To address this warning, preallocate memory for the output `y`, which is the same size as the input `z`. Add this code before the `for` loop.

   ```
   % Pre-allocate output signal:
   y=zeros(size(z));
   ```

   The orange marker disappears and the code analyzer message indicator in the top right edge of the code turns green, which indicates that you have fixed all the errors and warnings detected by the code analyzer.

   Why Preallocate the Outputs?

   You must preallocate outputs because code generation does not support increasing the size of an array over time.

7. Change the function name to `ex_kalman03` and save the file as `ex_kalman03.m` in the current folder.

You are ready to begin the next task in the tutorial, “Modifying Your Model to Call the Updated Algorithm” on page 44-144.

Modifying Your Model to Call the Updated Algorithm

To modify the model yourself, work through the exercises in this section. Otherwise, open the supplied model `ex_kalman22.mdl` in your *`solutions`* subfolder to see the modified model.

Next, update your model to provide the input as fixed-size frames of data and call `ex_kalman03` passing in the data one frame at a time.

1. Open `ex_kalman11` model in Simulink.

   ```
   ex_kalman11
   ```

2. Double-click the MATLAB Function block to open the MATLAB Function Block Editor.

3. Replace the code that calls `ex_kalman02` with a call to `ex_kalman03`.

   ```
   function y = kalman(u)
   %#codegen

   y = ex_kalman03(u);
   ```

4. Close the editor.
5. Modify the `InitFcn` callback:
   a. On the **Modeling** tab, select **Model Settings** > **Model Properties**.

      The Model Properties dialog box opens.
   b. In this dialog box, select the **Callbacks** tab.
   c. Select `InitFcn` in the **Model callbacks** pane.
   d. Replace the existing callback with:

      ```
      load position.mat;
      [R,C]=size(position);
      FRAME_SIZE=5;
      idx=(1:FRAME_SIZE:C)';
      LEN=length(idx);
      t=(1:LEN)'-1;
      ```

      This callback sets the frame size to `5`, and the index to increment by `5`.
   e. Click **Apply** and close the Model Properties dialog box.
6. Update the Selector block to use the correct indices.
   a. Double-click the Selector block to view the function block parameters.

      The Function Block Parameters dialog box opens.
   b. Set the second **Index Option** to `Starting index (port)`.
   c. Set the **Output Size** for the second input to `FRAME_SIZE`, click **Apply** and close the dialog box.

   Now, the **Index Option** for the first port is `Select all` and for the second port is `Starting index (port)`. Because the index increments by `5` each sample time, and the output size is `5`, the Selector block outputs a `2x5` output at each sample time.
7. Change the model simulation stop time to `61`. Now the frame size is `5`, so the simulation completes in a fifth of the sample times.
   a. In the Simulink model window, on the **Modeling** tab, click **Model Settings**.
   b. In the left pane of the Configuration Parameters dialog box, select **Solver**.
   c. In the right pane, set **Stop time** to `61`.
   d. Click **Apply** and close the dialog box.
8. Save the model as `ex_kalman22.mdl`.

Testing Your Modified Algorithm

To simulate the model:

1. In the Simulink model window, Click **Run**.

   As Simulink runs the model, it plots the trajectory of the object in blue and the Kalman filter estimated position in green as before when you used the batch filter.
2. The simulation stops.

You have proved that your algorithm accepts a fixed-size signal. You are now ready for the next task, “Using the Filter to Accept a Variable-Size Input” on page 44-146.

Using the Filter to Accept a Variable-Size Input

In this part of the tutorial, you learn how to specify variable-size data in your Simulink model. Then you test your Kalman filter algorithm with variable-size inputs and see that the algorithm is suitable for processing packets of data of varying size. For more information on using variable-size data in Simulink, see “Variable-Size Signal Basics” on page 77-2.

Updating the Model to Use Variable-Size Inputs

To modify the model yourself, work through the exercises in this section. Otherwise, open the supplied model `ex_kalman33.mdl` in your *`solutions`* subfolder to see the modified model.

1. Open `ex_kalman22.mdl` in Simulink.

   ```
   ex_kalman22
   ```

2. Modify the `InitFcn` callback:

   a. On the **Modeling** tab, select **Model Settings** > **Model Properties**.

      The Model Properties dialog box opens.

   b. Select the **Callbacks** tab.

   c. Select `InitFcn` in the **Model callbacks** pane.

   d. Replace the existing callback with:

      ```
      load position.mat;
      idx=[ 1 1 ;2 3 ;4 6 ;7 10 ;11 15 ;16 30 ;
           31 70 ;71 100 ;101 200 ;201 250 ;251 310];
      LEN=length(idx);
      t=(0:1:LEN-1)';
      ```

      This callback sets up indexing to generate eleven different size inputs. It specifies the start and end indices for each sample time. The first sample time uses only the first element, the second sample time uses the second and third elements, and so on. The largest sample, `101` to `200`, contains 100 elements.

   e. Click **Apply** and close the **Model Properties** dialog box.

3. Update the Selector block to use the correct indices.

   a. Double-click the Selector block to view the function block parameters.

      The Function Block Parameters dialog box opens.

   b. Set the second **Index Option** to `Starting and ending indices (port)`, then click **Apply** and close the dialog box.

      This setting means that the input to the index port specifies the start and end indices for the input at each sample time. Because the index input specifies different starting and ending indices at each sample time, the Selector block outputs a variable-size signal as the simulation progresses.

4. Use the Ports and Data Manager to set the MATLAB Function input `x` and output `y` as variable-size data.

a. Double-click the MATLAB Function block to open the MATLAB Function Block Editor.
b. From the editor menu, select **Edit Data**.
c. In the Ports and Data Manager left pane, select the input `u`.

   The Ports and Data Manager displays information about `u` in the right pane.
d. On the **General** tab, select the **Variable size** check box and click **Apply**.
e. In the left pane, select the output `y`.
f. On the **General** tab:
   i. Set the **Size** of `y` to `[2 100]` to specify a 2-D matrix where the upper bounds are `2` for the first dimension and `100` for the second, which is the maximum size input specified in the `InitFcn` callback.
   ii. Select the **Variable size** check box.
   iii. Click **Apply**.
g. Close the Ports and Data Manager.

5. Now do the same for the other MATLAB Function block. Use the Ports and Data Manager to set the Visualizing block inputs `y` and `z` as variable-size data.
   a. Double-click the Visualizing block to open the MATLAB Function Block Editor.
   b. From the editor menu, select **Edit Data**.
   c. In the Ports and Data Manager left pane, select the input `y`.
   d. On the **General** tab, select the **Variable size** check box and click **Apply**.
   e. In the left pane, select the input `z`.
   f. On the **General** tab, select the **Variable size** check box and click **Apply**.
   g. Close the Ports and Data Manager.
6. Change the model simulation stop time to `10`. This time, the filter processes one of the eleven different size inputs each sample time.
7. Save the model as `ex_kalman33.mdl`.

Testing Your Modified Model

To simulate the model:

1. In the Simulink model window, click **Run**.

   As Simulink runs the model, it plots the trajectory of the object in blue and the Kalman filter estimated position in green as before.

   Note that the signal lines between the Selector block and the Tracking and Visualization blocks change to show that these signals are variable-size.
2. The simulation stops.

You have successfully created an algorithm that accepts variable-size inputs. Next, you learn how to debug your MATLAB Function block, as described in “Debugging the MATLAB Function Block” on page 44-147.

Debugging the MATLAB Function Block

You can debug your MATLAB Function block just like you can debug a function in MATLAB.

1. Double-click the MATLAB Function block that calls the Kalman filter to open the MATLAB Function Block Editor.
2. In the editor, click the dash (-) character in the left margin of the line:

   ```
   y = kalman03(u);
   ```

   A small red ball appears in the margin of this line, indicating that you have set a breakpoint.
3. In the Simulink model window, click **Run**.

   The simulation pauses when execution reaches the breakpoint and a small green arrow appears in the left margin.
4. Place the pointer over the variable `u`.

   The value of `u` appears adjacent to the pointer.
5. From the MATLAB Function Block Editor menu, select **Step In**.

   The `kalman03.m` file opens in the editor and you can now step through this code using **Step**, **Step In**, and **Step Out**.
6. Select **Step Out**.

   The `kalman03.m` file closes and the MATLAB Function block code reappears in the editor.
7. Place the pointer over the output variable `y`.

   You can now see the value of `y`.
8. Click the red ball to remove the breakpoint.
9. From the MATLAB Function Block Editor menu, select **Quit Debugging**.
10. Close the editor.
11. Close the figure window.

    Now you are ready for the next task, “Generating C Code” on page 44-148.

Generating C Code

You have proved that your algorithm works in Simulink. Next you generate C/C++ code for your model. Code generation requires Simulink Coder.

Note Before generating code, you must check that your MATLAB code is suitable for code generation. If you call your MATLAB code as an extrinsic function, you must remove extrinsic calls before generating code.

---

1. Rename the MATLAB Function block to `Tracking`. To rename the block, double-click the annotation `MATLAB Function` below the MATLAB Function block and replace the text with `Tracking`.

   When you generate code for the MATLAB Function block, Simulink Coder uses the name of the block in the generated code. It is good practice to use a meaningful name.
2. Before generating code, ensure that Simulink Coder creates a code generation report. This HTML report provides easy access to the list of generated files with a summary of the configuration settings used to generate the code.

a. In the Simulink model window, on the **Modeling** tab, click **Model Settings**.

   The Configuration Parameters dialog box opens.

b. In the left pane of the Configuration Parameters dialog box, select **Report** under **Code Generation**.

c. In the right pane, select **Create code generation report** and **Open report automatically**.

d. Click **Apply** and close the Configuration Parameters dialog box.

e. Save your model.

3. To generate code for the Tracking block:

   a. Right-click the Tracking block and select **C/C++ Code** > **Build Selected Subsystem**.

   b. In the **Build code for Subsystem** window, click **Build**. For more information, see "Generate Code and Executables for Individual Subsystems" (Simulink Coder).

4. The Simulink software generates an error informing you that it cannot log variable-size signals as arrays. You need to change the format of data saved to the MATLAB workspace. To change this format:

   - In the Simulink model window, on the **Modeling** tab, click **Model Settings**.

     The Configuration Parameters dialog box opens.

   - In the left pane of the Configuration Parameters dialog box, select **Data Import/Export** and set the **Format** to `Structure with time`.

     The logged data is now a structure that has two fields: a time field and a signals field, enabling Simulink to log variable-size signals.

   - Click **Apply** and close the Configuration Parameters dialog box.
   - Save your model.

5. Repeat step 3 to generate code for the Tracking block.

   The Simulink Coder software generates C code for the block and launches the code generation report.

   For more information on using the code generation report, see "Reports for Code Generation" (Simulink Coder).

6. In the left pane of the code generation report, click the `Tracking.c` link to view the generated C code. Note that in the code generated for the MATLAB Function block, `Tracking`, there might be no separate function code for the `ex_kalman03` function because function inlining is enabled by default.

7. Modify your filter algorithm to disable inlining:

   a. In `ex_kalman03.m`, after the function declaration, add:

   ```
   coder.inline('never');
   ```

   b. Change the function name to `ex_kalman04` and save the file as `ex_kalman04.m` in the current folder.

   c. In your `ex_kalman33` model, double-click the Tracking block.

      The MATLAB Function Block Editor opens.

   d. Modify the call to the filter algorithm to call `ex_kalman04`.

```
function y = kalman(u)
%#codegen

y = ex_kalman04(u);
```

e Save the model as `ex_kalman44.mdl`.

8 Generate and inspect the C code.

a Repeat step 3.

b In the left pane of the code generation report, click the `Tracking.c` link to view the generated C code.

Inspect the generated C code for the `ex_kalman04` function.

```
/* Forward declaration for local functions */
static void Tracking_ex_kalman04(const real_T z_data[620], const int32_T
  z_sizes[2], real_T y_data[620], int32_T y_sizes[2]);

/* Function for MATLAB Function Block: '<Root>/Tracking' */
static void Tracking_ex_kalman04(const real_T z_data[620], const int32_T   48
     z_sizes[2], real_T y_data[620], int32_T y_sizes[2])
```

Best Practices Used in This Tutorial

Best Practice — Saving Incremental Code Updates

Save your code before making modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention. For example, add a two-digit suffix to the file name for each file in a sequence.

Key Points to Remember

- Back up your MATLAB code before you modify it.
- Decide on a naming convention for your files and save interim versions frequently. For example, this tutorial uses a two-digit suffix to differentiate the various versions of the filter algorithm.
- For simulation purposes, before generating code, call your MATLAB code using `coder.extrinsic` to check that your algorithm is suitable for use in Simulink. This practice provides these benefits:
  - You do not have to make the MATLAB code suitable for code generation.
  - You can debug your MATLAB code in MATLAB while calling it from Simulink.
- Create a Simulink Coder code generation report. This HTML report provides easy access to the list of generated files with a summary of the configuration settings used to generate the code.

See Also

`coder.extrinsic`

Related Examples

- “Filter Audio Signal Using MATLAB Code” on page 44-152
- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

- “Code Generation for Variable-Size Arrays” on page 53-2
- “Get Started with Simulink Coder” (Simulink Coder)

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “When to Generate Code from MATLAB Algorithms” on page 48-2
- “Functions and Objects Supported for C/C++ Code Generation” on page 49-2

Filter Audio Signal Using MATLAB Code

In this section...

Learning Objectives

In this tutorial, you will learn how to:

- Use the MATLAB Function block to add MATLAB functions to Simulink models for modeling, simulation, and deployment to embedded processors.

  This capability is useful for coding algorithms that are better stated in the textual language of MATLAB than in the graphical language of Simulink.
- Use `coder.extrinsic` to call MATLAB code from a MATLAB Function block.

  This capability allows you to call existing MATLAB code from Simulink without first having to make this code suitable for code generation, allowing for rapid prototyping.
- Check that existing MATLAB code is suitable for code generation.
- Convert a MATLAB algorithm from batch processing to streaming.
- Use persistent variables in code that is suitable for code generation.

  You need to make the filter weights persistent so that the filter algorithm does not reset their values each time it runs.

Tutorial Prerequisites

- “What You Need to Know” on page 44-152
- “Required Products” on page 44-152

What You Need to Know

To work through this tutorial, you should have basic familiarity with MATLAB software. You should also understand how to create a basic Simulink model and how to simulate that model. For more information, see “Create a Simple Model”.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- Simulink

- Simulink Coder
- DSP System Toolbox
- C compiler

  For a list of supported compilers, see https://www.mathworks.com/support/compilers/current_release/.

For instructions on installing MathWorks products, refer to the installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window. For instructions on installing and setting up a C compiler, see "Setting Up the C or C++ Compiler" (MATLAB Coder).

Example: The LMS Filter

Description

A least mean squares (LMS) filter is an adaptive filter that adjusts its transfer function according to an optimizing algorithm. You provide the filter with an example of the desired signal together with the input signal. The filter then calculates the filter weights, or coefficients, that produce the least mean squares of the error between the output signal and the desired signal.

This example uses an LMS filter to remove the noise in a music recording. There are two inputs. The first input is the distorted signal: the music recording plus the filtered noise. The second input is the desired signal: the unfiltered noise. The filter works to eliminate the difference between the output signal and the desired signal and outputs the difference, which, in this case, is the clean music recording. When you start the simulation, you hear both the noise and the music. Over time, the adaptive filter removes the noise so you hear only the music.

Algorithm

This example uses the least mean squares (LMS) algorithm to remove noise from an input signal. The LMS algorithm computes the filtered output, filter error, and filter weights given the distorted and desired signals.

At the start of the tutorial, the LMS algorithm uses a batch process to filter the audio input. This algorithm is suitable for MATLAB, where you are likely to load in the entire signal and process it all at once. However, a batch process is not suitable for processing a signal in real time. As you work through the tutorial, you refine the design of the filter to convert the algorithm from batch-based to stream-based processing.

The baseline function signature for the algorithm is:

```
function [ signal_out, err, weights ] = ...
    lms_01(signal_in, desired)
```

The filtering is performed in the following loop:

```
for n = 1:SignalLength
  % Compute the output sample using convolution:
```

```
  signal_out(n,ch) = weights' * signal_in(n:n+FilterLength-1,ch);
  % Update the filter coefficients:
  err(n,ch) = desired(n,ch) - signal_out(n,ch) ;
  weights = weights + mu*err(n,ch)*signal_in(n:n+FilterLength-1,ch);
end
```

where `SignalLength` is the length of the input signal, `FilterLength` is the filter length, and `mu` is the adaptation step size.

What Is the Adaptation Step Size?

LMS algorithms have a step size that determines the amount of correction to apply as the filter adapts from one iteration to the next. Choosing the appropriate step size requires experience in adaptive filter design. A step size that is too small increases the time for the filter to converge. Filter convergence is the process where the error signal (the difference between the output signal and the desired signal) approaches an equilibrium state over time. A step size that is too large might cause the adapting filter to overshoot the equilibrium and become unstable. Generally, smaller step sizes improve the stability of the filter at the expense of the time it takes to adapt.

Filtering Process

The filtering process has three phases:

- Convolution

  The convolution for the filter is performed in:

  ```
  signal_out(n,ch) = weights' * signal_in(n:n+FilterLength-1,ch);
  ```

  #### What Is Convolution?

  Convolution is the mathematical foundation of filtering. In signal processing, convolving two vectors or matrices is equivalent to filtering one of the inputs by the other. In this implementation of the LMS filter, the convolution operation is the vector dot product between the filter weights and a subset of the distorted input signal.

- Calculation of error

  The error is the difference between the desired signal and the output signal:

  ```
  err(n,ch) = desired(n,ch) - signal_out(n,ch);
  ```

- Adaptation

  The new value of the filter weights is the old value of the filter weights plus a correction factor that is based on the error signal, the distorted signal, and the adaptation step size:

  ```
  weights = weights + mu*err(n,ch)*signal_in(n:n+FilterLength-1,ch);
  ```

Reference

Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Files for the Tutorial

- “About the Tutorial Files” on page 44-155

About the Tutorial Files

The tutorial uses the following files:

- Simulink model files for each step of the tutorial.
- MATLAB code files for each step of the example.

  Throughout this tutorial, you work with Simulink models that call MATLAB files that contain a simple least mean squares (LMS) filter algorithm.

Location of Files

The tutorial files are available in the following folder: `docroot\toolbox\simulink\examples\lms`. To run the tutorial, you must copy these files to a local folder. For instructions, see "Copying Files Locally" on page 44-156.

Names and Descriptions of Files

Type	Name	Description
MATLAB files	`lms_01`	Baseline MATLAB implementation of batch filter. Not suitable for code generation.
	`lms_02`	Filter modified from batch to streaming.
	`lms_03`	Frame-based streaming filter with Reset and Adapt controls.
	`lms_04`	Frame-based streaming filter with Reset and Adapt controls. Suitable for code generation.
	`lms_05`	Disabled inlining for code generation.
	`lms_06`	Demonstrates use of `coder.nullcopy`.
Simulink model files	`acoustic_environment`	Simulink model that provides an overview of the acoustic environment.
	`noise_cancel_00`	Simulink model without a MATLAB Function block.
	`noise_cancel_01`	Complete `noise_cancel_00` model including a MATLAB Function block.
	`noise_cancel_02`	Simulink model for use with `lms_02.m`.
	`noise_cancel_03`	Simulink model for use with `lms_03.m`.
	`noise_cancel_04`	Simulink model for use with `lms_04.m`.
	`noise_cancel_05`	Simulink model for use with `lms_05.m`.
	`noise_cancel_06`	Simulink model for use with `lms_06.m`.
	`design_templates`	Simulink model containing Adapt and Reset controls.

Tutorial Steps

Copying Files Locally

Copy the tutorial files to a local folder:

1. Create a local *`solutions`* folder, for example, `c:\test\lms\solutions`.
2. Change to the `docroot\toolbox\simulink\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'simulink', 'examples'))
   ```
3. Copy the contents of the `lms` subfolder to your *`solutions`* folder, specifying the full path name of the *`solutions`* folder:

   ```
   copyfile('lms', 'solutions')
   ```

   Your *`solutions`* folder now contains a complete set of solutions for the tutorial. If you do not want to perform the steps for each task, you can view the supplied solution to see how the code should look.
4. Create a local *`work`* folder, for example, `c:\test\lms\work`.
5. Copy the following files from your *`solutions`* folder to your *`work`* folder.

   - `lms_01`
   - `lms_02`
   - `noise_cancel_00`
   - `acoustic_environment`
   - `design_templates`

   Your *`work`* folder now contains all the files that you need to get started.

   You are now ready to set up your C compiler.

Setting Up Your C Compiler

Building your MATLAB Function block requires a supported compiler. MATLAB automatically selects one as the default compiler. If you have multiple MATLAB-supported compilers installed on your system, you can change the default using the `mex -setup` command. See “Change Default Compiler” and the list of .

Running the acoustic_environment Model

Run the `acoustic_environment` model supplied with the tutorial to understand the problem that you are trying to solve using the LMS filter. This model adds band-limited white noise to an audio signal and outputs the resulting signal to a speaker.

To simulate the model:

1. Open the `acoustic_environment` model in Simulink:
   a. Set your MATLAB current folder to the folder that contains your working files for this tutorial. At the MATLAB command line, enter:

      ```
      cd work
      ```

      where *`work`* is the full path name of the folder containing your files. See “Find Files and Folders” for more information.
   b. At the MATLAB command line, enter:

      ```
      acoustic_environment
      ```
2. Ensure that your speakers are on.
3. To simulate the model, from the Simulink model window, click **Run**.

   As Simulink runs the model, you hear the audio signal distorted by noise.
4. While the simulation is running, double-click the Manual Switch to select the audio source.

   Now you hear the desired audio input without any noise.

The goal of this tutorial is to use a MATLAB LMS filter algorithm to remove the noise from the noisy audio signal. You do this by adding a MATLAB Function block to the model and calling the MATLAB code from this block.

Adding a MATLAB Function Block to Your Model

To modify the model and code yourself, work through the exercises in this section. Otherwise, open the supplied model `noise_cancel_01` in your *`solutions`* subfolder to see the modified model.

For the purposes of this tutorial, you add the MATLAB Function block to the `noise_cancel_00` model supplied with the tutorial. In practice, you would have to develop your own test bench starting with an empty Simulink model.

To add a MATLAB Function block to the `noise_cancel_00` model:

1. Open `noise_cancel_00` in Simulink.

   ```
   noise_cancel_00
   ```
2. Add a MATLAB Function block to the model:
   a. At the MATLAB command line, type `slLibraryBrowser` to open the Simulink Library Browser.
   b. From the list of Simulink libraries, select the `User-Defined Functions` library.
   c. Click the MATLAB Function block and drag it into the `noise_cancel_00` model. Place the block just above the red text annotation `Place MATLAB Function Block here`.

d Delete the red text annotations from the model.

e Save the model in the current folder as `noise_cancel_01`.

Calling Your MATLAB Code As an Extrinsic Function for Rapid Prototyping

In this part of the tutorial, you use the `coder.extrinsic` function to call your MATLAB code from the MATLAB Function block for rapid prototyping.

Why Call MATLAB Code As an Extrinsic Function?

Calling MATLAB code as an extrinsic function provides these benefits:

- For rapid prototyping, you do not have to make the MATLAB code suitable for code generation.
- Using `coder.extrinsic` enables you to debug your MATLAB code in MATLAB. You can add one or more breakpoints in the `lms_01.m` file, and then start the simulation in Simulink. When the MATLAB execution engine encounters a breakpoint, it temporarily halts execution so that you can inspect the MATLAB workspace and view the current values of all variables in memory. For more information about debugging MATLAB code, see “Debug a MATLAB Program”.

How to Call MATLAB Code As an Extrinsic Function

To call your MATLAB code from the MATLAB Function block:

1 Double-click the MATLAB Function block to open the MATLAB Function Block Editor.

2 Delete the default code displayed in the MATLAB Function Block Editor.

3 Copy the following code to the MATLAB Function block.

```
function [ Signal_Out, Weights ] = LMS(Noise_In, Signal_In) %#codegen
    % Extrinsic:
    coder.extrinsic('lms_01');

    % Compute LMS:
    [ ~, Signal_Out, Weights ] = lms_01(Noise_In, Signal_In);
end
```

Why Use the Tilde (~) Operator?

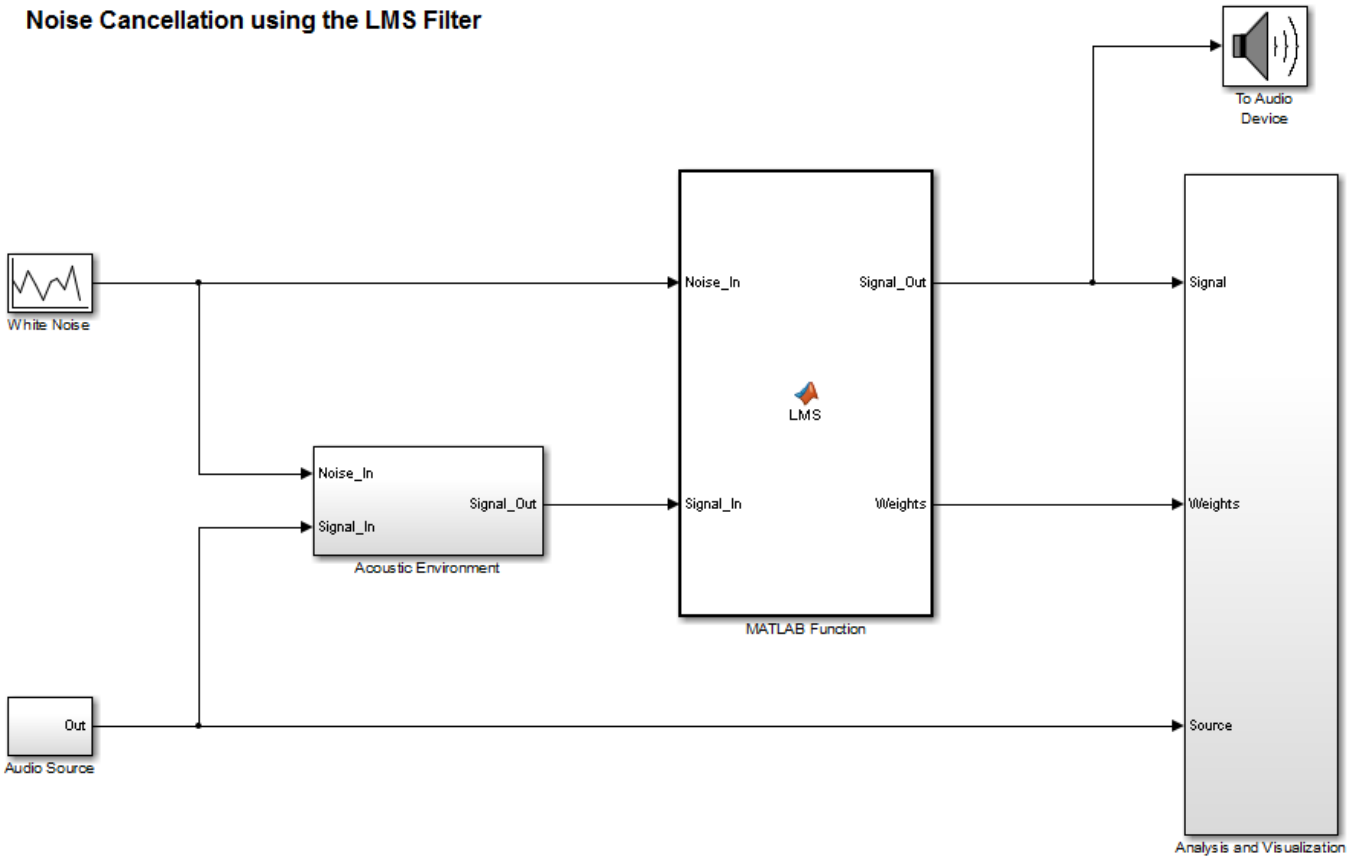
Because the `LMS` function does not use the first output from `lms_01`, replace this output with the MATLAB ~ operator. MATLAB ignores inputs and outputs specified by ~. This syntax helps avoid confusion in your program code and unnecessary clutter in your workspace, and allows you to reuse existing algorithms without modification.

4 Save the model.

The `lms_01` function inputs `Noise_In` and `Signal_In` now appear as input ports to the block and the function outputs `Signal_Out` and `Weights` appear as output ports.

Connecting the MATLAB Function Block Inputs and Outputs

1 Connect the MATLAB Function block inputs and outputs so that your model looks like this.



2 In the MATLAB Function block code, preallocate the outputs by adding the following code after the extrinsic call:

```
% Outputs:
Signal_Out = zeros(size(Signal_In));
Weights = zeros(32,1);
```

The size of `Weights` is set to match the Numerator coefficients of the Digital Filter in the Acoustic Environment subsystem.

Why Preallocate the Outputs?

For code generation, you must assign variables explicitly to have a specific class, size, and complexity before using them in operations or returning them as outputs in MATLAB functions. For more information, see “Differences Between Generated Code and MATLAB Code” on page 48-6.

3 Save the model.

You are now ready to check your model for errors.

Simulating the noise_cancel_01 Model

To simulate the model:

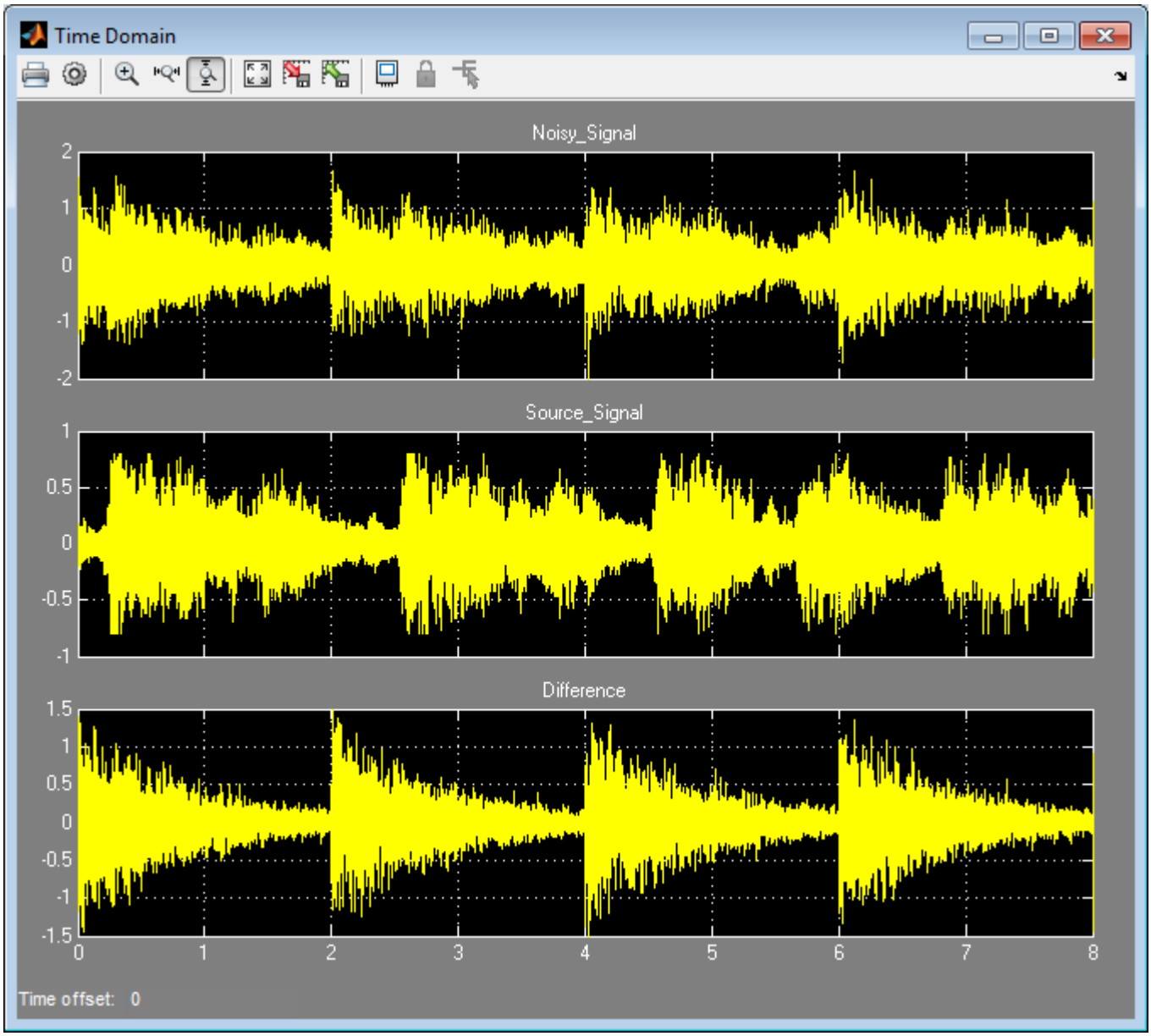
1 Ensure that you can see the **Time Domain** plots.

To view the plots, in the `noise_cancel_01` model, open the Analysis and Visualization block and then open the Time Domain block.

2 In the Simulink model window, click **Run**.

As Simulink runs the model, you see and hear outputs. Initially, you hear the audio signal distorted by noise. Then the filter attenuates the noise gradually, until you hear only the music playing with very little noise remaining. After two seconds, you hear the distorted noisy signal again and the filter attenuates the noise again. This cycle repeats continuously.

MATLAB displays the following plot showing this cycle.



3 Stop the simulation.

Why Does the Filter Reset Every 2 Seconds?

The filter resets every 2 seconds because the model uses `16384` samples per frame and a sampling rate of `8192`, so the `16384` samples represent 2 seconds of audio.

To see the model configuration:

1. Double-click the White Noise subsystem and note that it uses a **Sample time** of `1/Fs` and **Samples per frame** of `FrameSize`. The `music` in the Audio Source subsystem also uses these values.
2. `FrameSize` is set in the model `InitFcn` callback. To view this callback:
   a. Right-click inside the model window and select **Model Properties**.
   b. Select the **Callbacks** tab.
   c. Select `InitFcn` in the **Model callbacks** pane.

   Note that `FrameSize = 16*1024`, which is `16384`.
3. `Fs` is set in the model `PostLoadFcn` callback. To view this callback, select `PostLoadFcn` in the **Model callbacks** pane:

   The following MATLAB commands set up `Fs`:

```
data = load('handel.mat');
music = data.y;
Fs = data.Fs;
```

Modifying the Filter to Use Streaming

What Is Streaming?

A streaming filter is called repeatedly to process fixed-size chunks of input data, or *frames*, until it has processed the entire input signal. The frame size can be as small as a single sample, in which case the filter would be operating in a sample-based mode, or up to a few thousand samples, for frame-based processing.

Why Use Streaming?

The design of the filter algorithm in `lms_01` has the following disadvantages:

- The algorithm does not use memory efficiently.

  Preallocating a fixed amount of memory for each input signal for the lifetime of the program means more memory is allocated than is in use.
- You must know the size of the input signal at the time you call the function.

If the input signal is arriving in real time or as a stream of samples, you would have to wait to accumulate the entire signal before you could pass it, as a batch, to the filter.

- The signal size is limited to a maximum size.

In an embedded application, the filter is likely to be processing a continuous input stream. As a result, the input signal can be substantially longer than the maximum length that a filter working in batch mode could possibly handle. To make the filter work for any signal length, it must run in real time. One solution is to convert the filter from batch-based processing to stream-based processing.

Viewing the Modified MATLAB Code

The conversion to streaming involves:

- Introducing a first-in, first-out (FIFO) queue

  The FIFO queue acts as a temporary storage buffer, which holds a small number of samples from the input data stream. The number of samples held by the FIFO queue must be exactly the same as the number of samples in the filter's impulse response, so that the function can perform the convolution operation between the filter coefficients and the input signal.
- Making the FIFO queue and the filter weights persistent

  The filter is called repeatedly until it has processed the entire input signal. Therefore, the FIFO queue and filter weights need to persist so that the adaptation process does not have to start over again after each subsequent call to the function.

Open the supplied file `lms_02.m` in your *`work`* subfolder to see the modified algorithm.

Summary of Changes to the Filter Algorithm

Note the following important changes to the filter algorithm:

- The filter weights and the FIFO queue are declared as persistent:

```
persistent weights;
persistent fifo;
```

- The FIFO queue is initialized:

```
fifo = zeros(FilterLength,ChannelCount);
```

- The FIFO queue is used in the filter update loop:

```
% For each channel:
for ch = 1:ChannelCount

   % For each sample time:
   for n = 1:FrameSize

     % Update the FIFO shift register:
     fifo(1:FilterLength-1,ch) = fifo(2:FilterLength,ch);
     fifo(FilterLength,ch) = signal_in(n,ch);

     % Compute the output sample using convolution:
     signal_out(n,ch) = weights' * fifo(:,ch);

     % Update the filter coefficients:
     err(n,ch) = desired(n,ch) - signal_out(n,ch) ;
     weights = weights + mu*err(n,ch)*fifo(:,ch);
```

```
   end
end
```

- You cannot output a persistent variable. Therefore, a new variable, `weights_out`, is used to output the filter weights:

```
function [ signal_out, err, weights_out ] = ...
  lms_02(distorted, desired)

weights_out = weights;
```

Modifying Your Model to Call the Updated Algorithm

To modify the model yourself, work through the exercises in this section. Otherwise, open the supplied model `noise_cancel_02` in your *`solutions`* subfolder to see the modified model.

1. In the `noise_cancel_01` model, double-click the MATLAB Function block to open the MATLAB Function Block Editor.
2. Modify the MATLAB Function block code to call `lms_02`.
   a. Modify the extrinsic call.

      ```
      % Extrinsic:
      coder.extrinsic('lms_02');
      ```

   b. Modify the call to the filter algorithm.

      ```
      % Compute LMS:
      [ ~, Signal_Out, Weights ] = lms_02(Noise_In, Signal_In);
      ```

      Modified MATLAB Function Block Code

      Your MATLAB Function block code should now look like this:

      ```
      function [ Signal_Out, Weights ] = LMS(Noise_In, Signal_In)
        % Extrinsic:
        coder.extrinsic('lms_02');
        % Outputs:
        Signal_Out = zeros(size(Signal_In));
        Weights = zeros(32,1);
        % Compute LMS:
        [ ~, Signal_Out, Weights ] = lms_02(Noise_In, Signal_In);
      end
      ```

3. Change the frame size from `16384` to `64`, which represents a more realistic value.
   a. Right-click inside the model window and select **Model Properties**.
   b. Select the **Callbacks** tab.
   c. In the **Model callbacks** list, select `InitFcn`.
   d. Change the value of `FrameSize` to `64`.
   e. Click **Apply** and close the dialog box.
4. Save your model as `noise_cancel_02`.

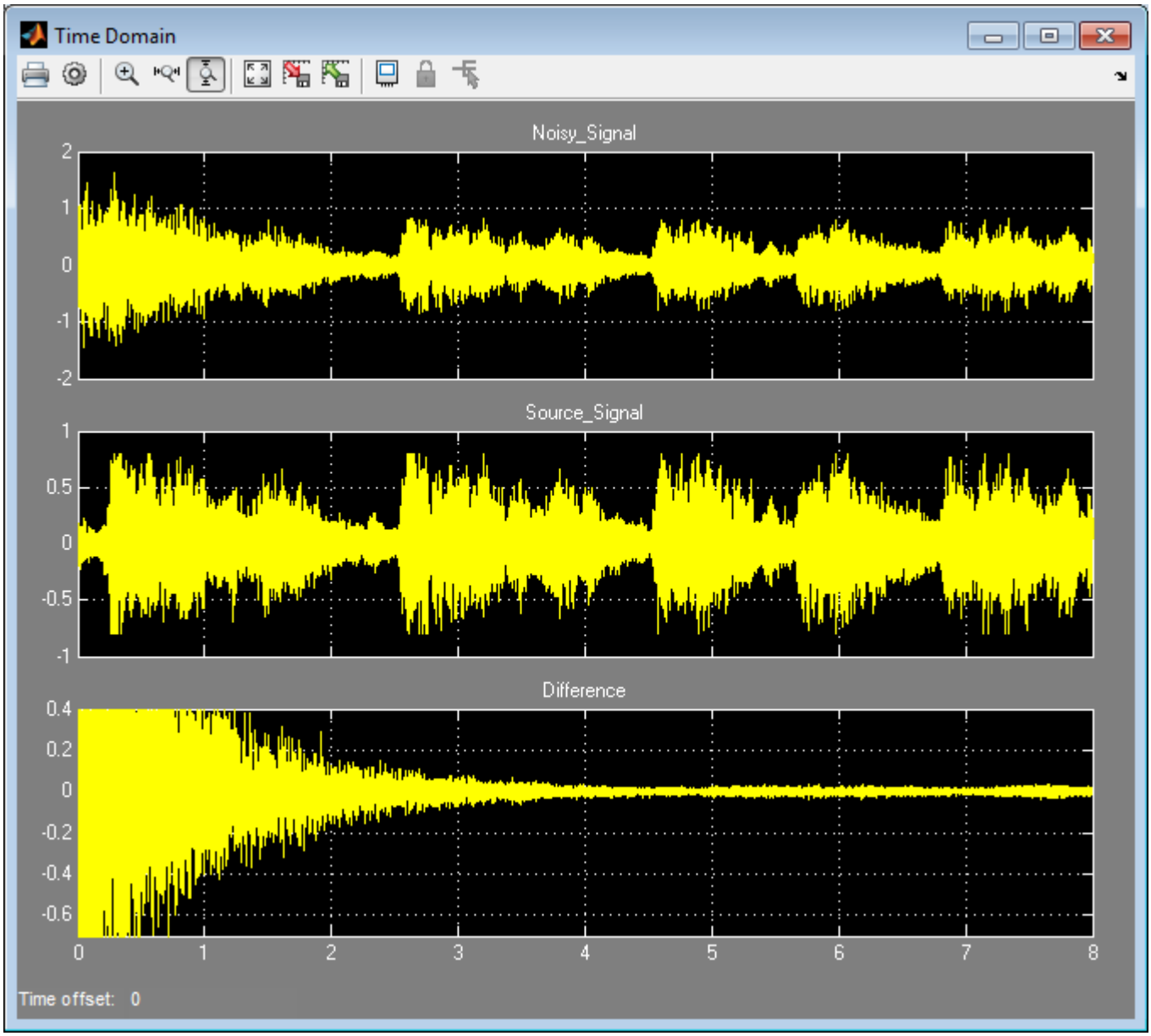
Simulating the Streaming Algorithm

To simulate the model:

1 Ensure that you can see the **Time Domain** plots.

2 Start the simulation.

As Simulink runs the model, you see and hear outputs. Initially, you hear the audio signal distorted by noise. Then, during the first few seconds, the filter attenuates the noise gradually, until you hear only the music playing with very little noise remaining. MATLAB displays the following plot showing filter convergence after only a few seconds.



3 Stop the simulation.

The filter algorithm is now suitable for Simulink. You are ready to elaborate your model to use `Adapt` and `Reset` controls.

Adding Adapt and Reset Controls

Why Add Adapt and Reset Controls?

In this part of the tutorial, you add Adapt and Reset controls to your filter. Using these controls, you can turn the filtering on and off. When `Adapt` is enabled, the filter continuously updates the filter weights. When `Adapt` is disabled, the filter weights remain at their current values. If `Reset` is set, the filter resets the filter weights.

Modifying Your MATLAB Code

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `lms_03.m` in your *`solutions`* subfolder to see the modified algorithm.

To modify your filter code:

1. Open `lms_02.m`.
2. In the `Set up` section, replace

   ```
   if ( isempty(weights) )
   ```

   with

   ```
   if ( reset || isempty(weights) )
   ```
3. In the filter loop, update the filter coefficients only if `Adapt` is `ON`.

   ```
   if adapt
     weights = weights + mu*err(n,ch)*fifo(:,ch);
   end
   ```
4. Change the function signature to use the `Adapt` and `Reset` inputs and change the function name to `lms_03`.

   ```
   function [ signal_out, err, weights_out ] = ...
     lms_03(signal_in, desired, reset, adapt)
   ```
5. Save the file in the current folder as `lms_03.m`:

Summary of Changes to the Filter Algorithm

Note the following important changes to the filter algorithm:

- The new input parameter `reset` is used to determine if it is necessary to reset the filter coefficients:

  ```
  if ( reset || isempty(weights) )
    % Filter coefficients:
    weights = zeros(L,1);
    % FIFO Shift Register:
    fifo = zeros(L,1);
  end
  ```

- The new parameter `adapt` is used to control whether the filter coefficients are updated or not.

```
if adapt
  weights = weights + mu*err(n)*fifo;
end
```

Modifying Your Model to Use Reset and Adapt Controls

To modify the model yourself, work through the exercises in this section. Otherwise, open the supplied model `noise_cancel_03` in your *`solutions`* subfolder to see the modified model.

1. Open the `noise_cancel_02` model.
2. Double-click the MATLAB Function block to open the MATLAB Function Block Editor.
3. Modify the MATLAB Function block code:
   a. Update the function declaration.

      ```
      function [ Signal_Out, Weights ] = ...
         LMS(Adapt, Reset, Noise_In, Signal_In )
      ```

   b. Update the extrinsic call.

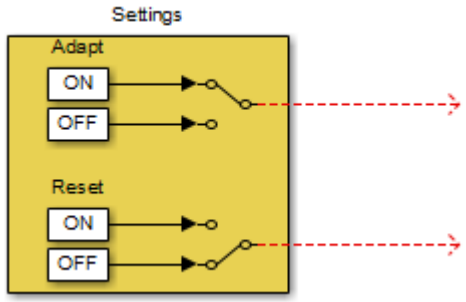
      ```
      coder.extrinsic('lms_03');
      ```

   c. Update the call to the LMS algorithm.

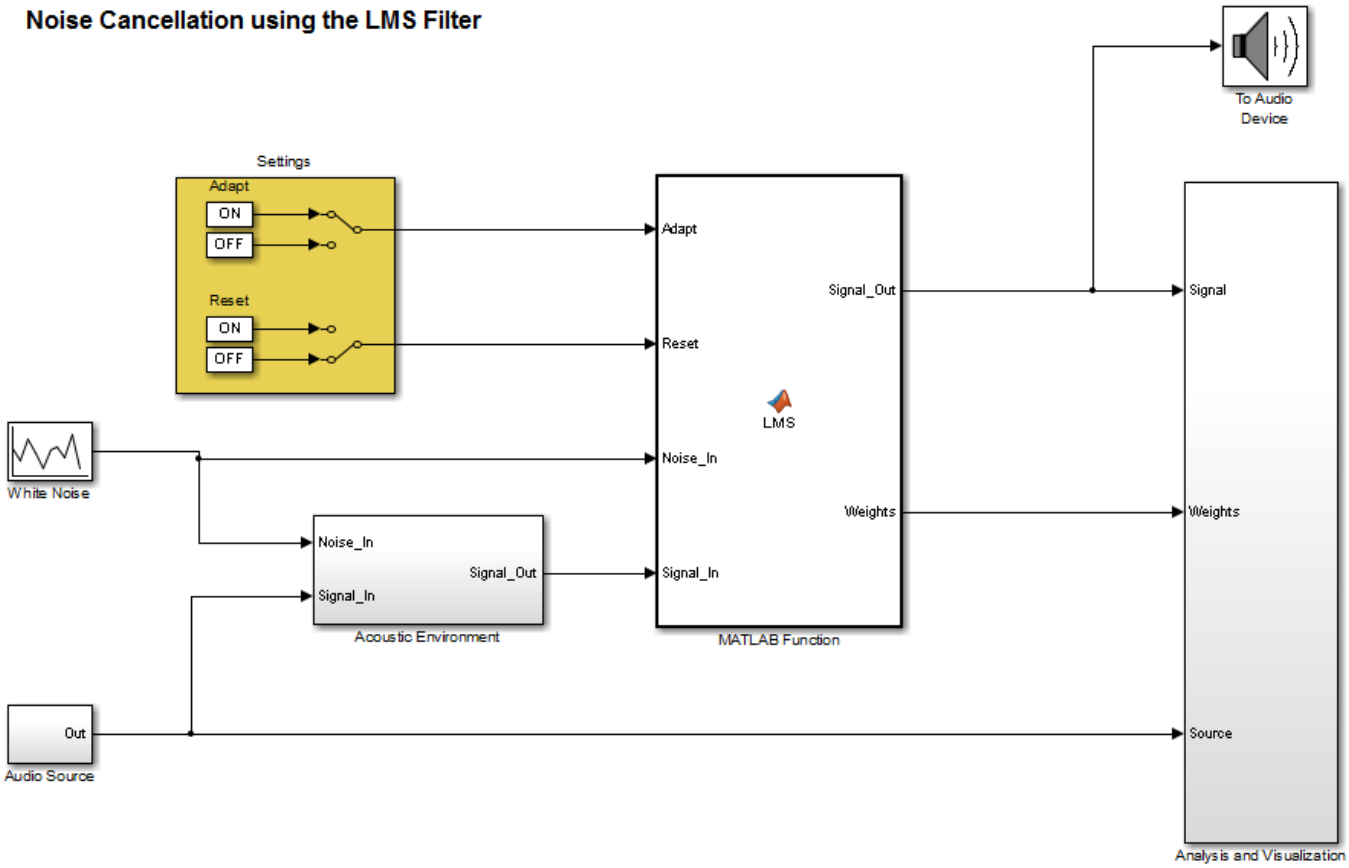
      ```
      % Compute LMS:
      [ ~, Signal_Out, Weights ] = ...
         lms_03(Noise_In, Signal_In, Reset, Adapt);
      ```

   d. Close the MATLAB Function Block Editor.

      The `lms_03` function inputs `Reset` and `Adapt` now appear as input ports to the MATLAB Function block.
4. Open the `design_templates` model.



5. Copy the Settings block from this model to your `noise_cancel_02` model:
   a. From the `design_templates` model menu, select **Edit** > **Select All**.
   b. Select **Edit** > **Copy**.
   c. From the `noise_cancel_02` model menu, select **Edit** > **Paste**.
6. Connect the Adapt and Reset outputs of the Settings subsystem to the corresponding inputs on the MATLAB Function block. Your model should now appear as follows.



7 Save the model as `noise_cancel_03`.

Simulating the Model with Adapt and Reset Controls

To simulate the model and see the effect of the Adapt and Reset controls:

1 In the `noise_cancel_03` model, view the Convergence scope:

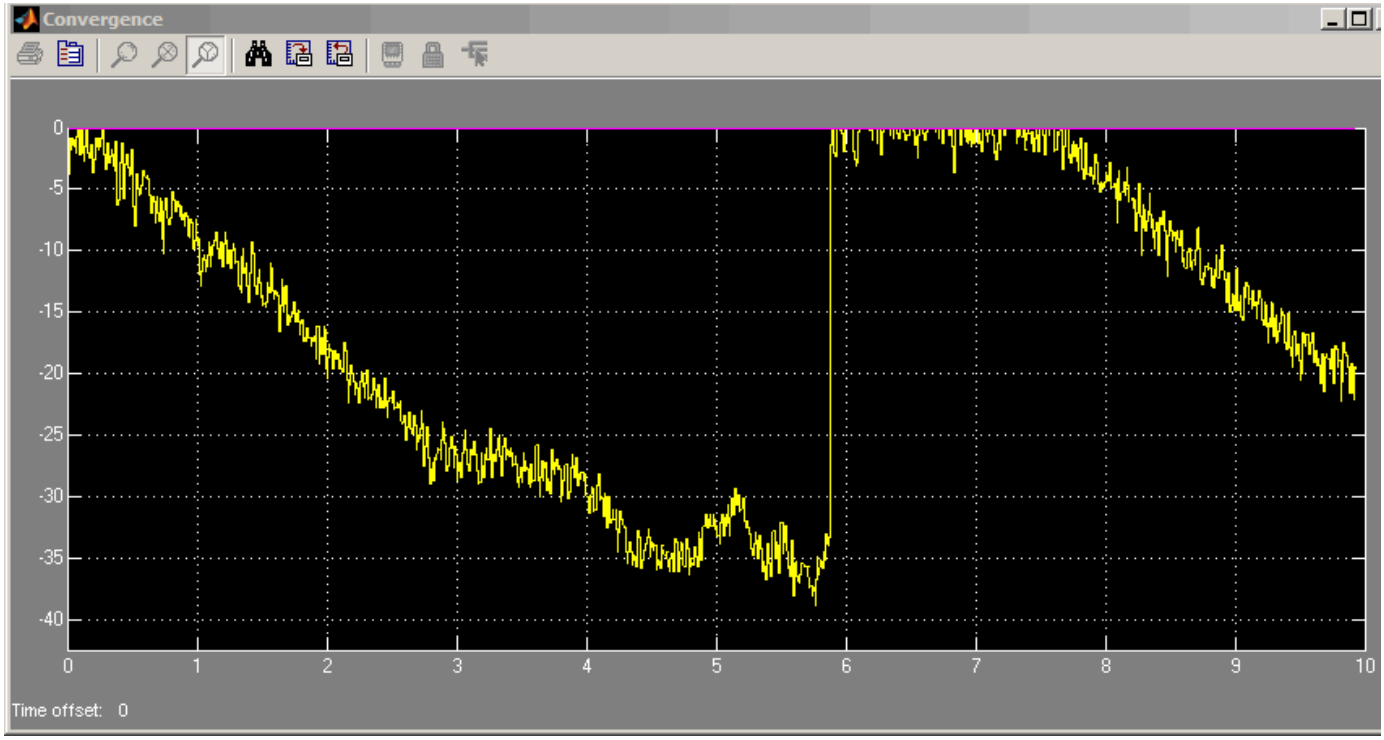
   a Double-click the Analysis and Visualization subsystem.

   b Double-click the Convergence scope.

2 In the Simulink model window, click **Run**.

   Simulink runs the model as before. While the model is running, toggle the Adapt and Reset controls and view the Convergence scope to see their effect on the filter.

   The filter converges when `Adapt` is `ON` and `Reset` is `OFF`, then resets when you toggle`Reset`. The results might look something like this:



3 Stop the simulation.

Generating Code

You have proved that your algorithm works in Simulink. Next you generate code for your model. Before generating code, you must ensure that your MATLAB code is suitable for code generation. For code generation, you must remove the extrinsic call to your code.

Making Your Code Suitable for Code Generation

To modify the model and code yourself, work through the exercises in this section. Otherwise, open the supplied model `noise_cancel_04` and file `lms_04.m` in your *`solutions`* subfolder to see the modifications.

1 Rename the MATLAB Function block to `LMS_Filter`. Select the annotation `MATLAB Function` below the MATLAB Function block and replace the text with `LMS_Filter`.

When you generate code for the MATLAB Function block, Simulink Coder uses the name of the block in the generated code. It is good practice to use a meaningful name.

2 In your `noise_cancel_03` model, double-click the MATLAB Function block.

The MATLAB Function Block Editor opens.

3 Delete the extrinsic declaration.

```
% Extrinsic:
coder.extrinsic('lms_03');
```

4 Delete the preallocation of outputs.

```
% Outputs:
Signal_Out = zeros(size(Signal_In));
Weights = zeros(32,1);
```

5. Modify the call to the filter algorithm.

```
% Compute LMS:
[ ~, Signal_Out, Weights ] = ...
   lms_04(Noise_In, Signal_In, Reset, Adapt);
```

6. Save the model as `noise_cancel_04`.
7. Open `lms_03.m`

   a. Modify the function name to `lms_04`.

   b. Turn on error checking specific to code generation by adding the `%#codegen` compilation directive after the function declaration.

   ```
   function [ signal_out, err, weights_out ] = ...
       lms_04(signal_in, desired, reset, adapt) %#codegen
   ```

   The code analyzer message indicator in the top right turns red to indicate that the code analyzer has detected code generation issues. The code analyzer underlines the offending code in red and places a red marker to the right of it.
8. Move your pointer over the first red marker to view the error information.

   The code analyzer detects that code generation requires `signal_out` to be fully defined before subscripting it and does not support growth of variable size data through indexing.
9. Move your pointer over the second red marker and note that the code analyzer detects the same errors for `err`.
10. To address these errors, preallocate the outputs `signal_out` and `err`. Add this code after the filter setup.

```
% Output Arguments:

% Pre-allocate output and error signals:
signal_out = zeros(FrameSize,ChannelCount);
err = zeros(FrameSize,ChannelCount);
```

Why Preallocate the Outputs?

You must preallocate outputs because code generation does not support increasing the size of an array over time.

The red error markers for the two lines of code disappear. The code analyzer message indicator in the top right edge of the code turns green, which indicates that you have fixed all the errors and warnings detected by the code analyzer.

11. Save the file as `lms_04.m`.

Generating Code for noise_cancel_04

1. Before generating code, ensure that Simulink Coder creates a code generation report. This HTML report provides easy access to the list of generated files with a summary of the configuration settings used to generate the code.

   a. In the Simulink model window, in the **Modeling** tab, click **Model Settings**.

   b. In the left pane of the Configuration Parameters dialog box, select **Code Generation** > **Report**.

c In the right pane, select **Create code generation report** and **Open report automatically**.

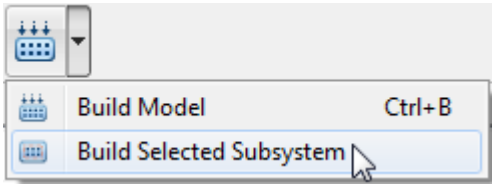
d Click **Apply** and close the Configuration Parameters dialog box.

e Save your model.

2 To generate code for the LMS Filter subsystem:

a In your model, select the LMS Filter subsystem.

b From the Build Model tool menu, select **Build Selected Subsystem**.



The **Build code for subsystem** dialog box appears. Click the **Build** button.

The Simulink Coder software generates C code for the subsystem and opens the code generation report.

For more information on using the code generation report, see “Generate a Code Generation Report” (Simulink Coder).

c In the left pane of the code generation report, click the `LMS_Filter.c` link to view the generated C code. Note that the `lms_04` function has no code because inlining is enabled by default.

3 Modify your filter algorithm to disable inlining:

a In `lms_04.m`, after the function declaration, add:

```
coder.inline('never')
```

b Change the function name to `lms_05` and save the file as `lms_05.m` in the current folder.

c In your `noise_cancel_04` model, double-click the MATLAB Function block.

The MATLAB Function Block Editor opens.

d Modify the call to the filter algorithm to call `lms_05`.

```
% Compute LMS:
[ ~, Signal_Out, Weights ] = ...
   lms_05(Noise_In, Signal_In, Reset, Adapt);
```

e Save the model as `noise_cancel_05`.

4 Generate code for the updated model.

a In the model, select the LMS Filter subsystem.

b From the Build Model tool menu, select **Build Selected Subsystem**.

The **Build code for subsystem** dialog box appears.

c Click the **Build** button.

The Simulink Coder software generates C code for the subsystem and opens the code generation report.

d In the left pane of the code generation report, click the `LMS_Filter.c` link to view the generated C code.

This time the `lms_05` function has code because you disabled inlining.

```
/* Forward declaration for local functions */
   static void LMS_Filter_lms_05 ...
       (const real_T signal_in[64],const real_T ...
       desired[64], real_T reset, real_T adapt, ...
             real_T signal_out[64], ...
      real_T err[64], real_T weights_out[32]);

/* Function for MATLAB Function Block: 'root/LMS_Filter' */
   static void LMS_Filter_lms_05 ...
      (const real_T signal_in[64], const real_T ...
         desired[64], real_T reset, real_T adapt, ...
             real_T signal_out[64], ...
     real_T err[64], real_T weights_out[32])
```

Optimizing the LMS Filter Algorithm

This part of the tutorial demonstrates when and how to preallocate memory for a variable without incurring the overhead of initializing memory in the generated code.

In `lms_05.m`, the MATLAB code not only declares `signal_out` and `err` to be a `FrameSize`-by-`ChannelCount` vector of real doubles, but also initializes each element of `signal_out` and `err` to zero. These signals are initialized to zero in the generated C code.

MATLAB Code	Generated C Code

`% Pre-allocate output and error signals:`
`signal_out = zeros(FrameSize,ChannelCount);`
`err = zeros(FrameSize,ChannelCount);`
	`/* Pre-allocate output and error signals: */`
	`79 for (i = 0; i < 64; i++) {`
	`80 signal_out[i] = 0.0;`
	`81 err[i] = 0.0;`
	`82 }`

This forced initialization is unnecessary because both `signal_out` and `err` are explicitly initialized in the MATLAB code before they are read.

Note You should not use `coder.nullcopy` when declaring the variables `weights` and `fifo` because these variables need to be initialized in the generated code. Neither variable is explicitly initialized in the MATLAB code before they are read.

Use `coder.nullcopy` in the declaration of `signal_out` and `err` to eliminate the unnecessary initialization of memory in the generated code:

1 In `lms_05.m`, preallocate `signal_out` and `err` using `coder.nullcopy`:

```
% Pre-allocate output and error signals:
signal_out = coder.nullcopy(zeros(FrameSize, ChannelCount));
err = coder.nullcopy(zeros(FrameSize, ChannelCount));
```

Caution After declaring a variable with `coder.nullcopy`, you must explicitly initialize the variable in your MATLAB code before reading it. Otherwise, you might get unpredictable results.

2. Change the function name to `lms_06` and save the file as `lms_06.m` in the current folder.
3. In your `noise_cancel_05` model, double-click the MATLAB Function block.

   The MATLAB Function Block Editor opens.
4. Modify the call to the filter algorithm.

```
% Compute LMS:
[ ~, Signal_Out, Weights ] = ...
   lms_06(Noise_In, Signal_In, Reset, Adapt);
```

5. Save the model as `noise_cancel_06`.

Generate code for the updated model.

1. Select the LMS Filter subsystem.
2. From the Build Model tool menu, select **Build Selected Subsystem**.

   The **Build code for subsystem** dialog box appears. Click the **Build** button.

   The Simulink Coder software and generates C code for the subsystem and opens the code generation report.
3. In the left pane of the code generation report, click the `LMS_Filter.c` link to view the generated C code.

   In the generated C code, this time there is no initialization to zero of `signal_out` and `err`.

See Also

`coder.extrinsic`

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Track Object Using MATLAB Code” on page 44-134

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “MATLAB Function Block Editor” on page 44-26
- “MATLAB Function Reports” on page 44-41

Interface with Row-Major Data in MATLAB Function Block

Array layout can be important for integration, usability, and performance. Simulink uses column-major layout by default, as does the MATLAB Function block. However, many devices, sensors, and libraries use row-major array layout for their data. You can apply your model directly to this data by using the `coder.ceval` function with row-major layout in a MATLAB Function block.

Array layout can also affect performance. Many algorithms perform memory access more efficiently for one specific array layout.

Row-Major Layout in Simulation and Code Generation

For the MATLAB Function block, you can specify row-major array layout inside the block. This specification occurs at the function level and does not alter the array layout of the model outside of the function. The array layout that you specify inside the MATLAB Function block applies to both simulation and C/C++ code generation. See “Specify Array Layout in Functions and Classes” on page 52-13.

For C/C++ code generation using Simulink Coder and Embedded Coder software, you can specify array layout at the model level, which is supported by MATLAB Function blocks. For more information on controlling array layout at the model level, see “Code Generation of Matrices and Arrays” (Simulink Coder). The model code generation setting for array layout has no affect for simulation. See “Array layout” (Simulink Coder). For examples of using row-major layout in MATLAB Function block for code generation, see “Generate Row-Major Code for Model That Contains a MATLAB Function Block” (Simulink Coder).

For the MATLAB Function block, the array layout specification at the function level takes precedence over the array layout specification of the model. However, for global and persistent variables, the array layout specification of the model takes precedence.

Array Layout Conversions

MATLAB and Simulink store data in column-major layout by default. The software automatically inserts array layout conversions as needed when you specify different array layouts in different functions and at different boundaries.

For example, when you simulate a model or generate code for a model that uses column-major layout, and the model contains a MATLAB Function block that uses row-major layout, then the software converts the block input data to row-major and the block output data back to column-major, as needed. Array layout conversions can affect performance. For more information on performance considerations for array layout, see “Code Design for Row-Major Array Layout” on page 52-17.

Array Layout and Algorithmic Efficiency

For certain algorithms, row-major layout provides more efficient memory access. Consider this function for adding two matrices. The algorithm performs the addition through explicit row and column traversal.

```
function [S] = addMatrix(A,B)
coder.rowMajor;
S = zeros(size(A));
for row = 1:size(A,1)
```

```
    for col = 1:size(A,2)
        S(row,col) = A(row,col) + B(row,col);
    end
end
```

If you use this code in a MATLAB Function block, code generation results in this C code for the function:

```
...
/* generated code for addMatrix using row-major */
for (row = 0; row < 20; row++) {
  for (col = 0; col < 10; col++) {
      S[col + 10 * row] = A[col + 10 * row] + B[col + 10 * row];
   }
}
...
```

The arrays are indexed by the generated code using the formula:

```
[col + 10 * row]
```

Because the arrays are stored in row-major layout, adjacent memory elements are separated by single column increments. The *stride length* for the algorithm is equal to one. The stride length is the distance in memory elements between consecutive memory accesses. A shorter stride length provides more efficient memory access.

Using column-major layout for the data results in a longer stride length and less efficient memory access. To see this comparison, consider the generated C code for `addMatrix` that uses column-major layout:

```
...
/* generated code for addMatrix using column-major */
for (row = 0; row < 20; row++) {
  for (col = 0; col < 10; col++) {
     S[row + 20 * col] = A[row + 20 * col] + B[row + 20 * col];
  }
}
...
```

In column-major layout, the column elements are contiguous in memory in the generated code. Adjacent memory elements are separated by single row increments and indexed by the formula:

```
[row + 20 * col]
```

However, the algorithm iterates through the columns in the inner for-loop. Therefore, the column-major C code must make a stride of 20 elements for each consecutive memory access.

The array layout that provides the most efficient memory access depends on the algorithm. For this algorithm, row-major layout of the data provides more efficient memory access. The algorithm traverses over the data row by row. Row-major storage is therefore more efficient.

Row-Major Layout for N-Dimensional Arrays

You can use row-major layout for N-dimensional arrays. When an array is stored in row-major layout, the elements from the last (rightmost) dimension or index are contiguous in memory. In column-major layout, the elements from the first (leftmost) dimension or index are contiguous.

Consider the example function `addMatrix3D`, which accepts three-dimensional inputs.

```
function [S] = addMatrix3D(A,B)
coder.rowMajor;
S = zeros(size(A));
for i = 1:size(A,1)
    for j = 1:size(A,2)
        for k = 1:size(A,3)
            S(i,j,k) = A(i,j,k) + B(i,j,k);
        end
    end
end
end
```

The code generator produces this C code:

```
...
/* row-major layout */
for (i = 0; i < 20; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 5; k++) {
            S[(k + 5 * j) + 50 * i] = A[(k + 5 * j) + 50 * i]
                                      + B[(k + 5 * j) + 50 * i];
        }
    }
}
...
```

In row-major layout, adjacent memory elements are separated by single increments of the last index, `k`. The inner for-loop iterates over adjacent elements separated by only one position in memory.

Remove the `coder.rowMajor` call and generate C code that uses column-major layout:

```
...
/* column-major layout */
for (i = 0; i < 20; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 5; k++) {
            S[(i + 20 * j) + 200 * k] = A[(i + 20 * j) + 200 * k]
                                        + B[(i + 20 * j) + 200 * k];
        }
    }
}
...
```

In column-major layout, adjacent elements are separated by single increments of the first index, `i`. The inner for-loop now iterates over adjacent elements separated by 200 positions in memory. The long stride length can cause performance degradation due to cache misses.

Because the algorithm iterates through the last index, `k`, in the inner for-loop, the stride length is much longer for the generated code that uses column-major layout. For this algorithm, row-major layout of the data provides more efficient memory access.

Specify Array Layout in External Function Calls

To call external C/C++ functions that expect data stored with a specific layout, use `coder.ceval` with the `layout` syntax. If you do not use this syntax, the external function inputs and outputs are assumed to use column-major layout by default.

Consider an external C function designed to use row-major layout called `myCFunctionRM`. To integrate this function into your code, call the function using the `'-layout:rowMajor'` or `'-row'` option. This option ensures that the input and output arrays are stored in row-major order. The code generator automatically inserts array layout conversions as needed.

```
coder.ceval('-layout:rowMajor','myCFunctionRM',coder.ref(in),coder.ref(out))
```

Within a MATLAB function that uses row-major layout, you may seek to call an external function designed to use column-major layout. In this case, use the `'-layout:columnMajor'` or `'-col'` option.

```
coder.ceval('-layout:columnMajor','myCFunctionCM',coder.ref(in),coder.ref(out))
```

You can perform row-major and column-major function calls in the same code. Consider the function `myMixedFn1` as an example:

```
function [E] = myMixedFn1(x,y)
%#codegen
coder.rowMajor;
% specify type of return arguments for ceval calls
D = zeros(size(x));
E = zeros(size(x));

% include external C functions that use row-major & column-major
coder.cinclude('addMatrixRM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixRM.c');
coder.cinclude('addMatrixCM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixCM.c');

% call C function that uses row-major order
coder.ceval('-layout:rowMajor','addMatrixRM', ...
    coder.rref(x),coder.rref(y),coder.wref(D));

% call C function that uses column-major order
coder.ceval('-layout:columnMajor','addMatrixCM', ...
    coder.rref(x),coder.rref(D),coder.wref(E));
end
```

The external files are:

addMatrixRM.h

```
extern void addMatrixRM(const double x[200], const double y[200], double z[200]);
```

addMatrixRM.c

```
#include "addMatrixRM.h"

void addMatrixRM(const double x[200], const double y[200], double z[200])
{
  int row;
```

```
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[col + 10 * row] = x[col + 10 * row] + y[col + 10 * row];
    }
  }
}
```

addMatrixCM.h

```
extern void addMatrixCM(const double x[200], const double y[200], double z[200]);
```

addMatrixCM.c

```
#include "addMatrixCM.h"

void addMatrixCM(const double x[200], const double y[200], double z[200])
{
  int row;
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[row + 20 * col] = x[row + 20 * col] + y[row + 20 * col];
    }
  }
}
```

See Also

`coder.ceval` | `coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- “Specify Array Layout in Functions and Classes” on page 52-13
- “Code Design for Row-Major Array Layout” on page 52-17
- “Code Generation of Matrices and Arrays” (Simulink Coder)

Integration Considerations for MATLAB Function Blocks

Use Nondirect Feedthrough in a MATLAB Function Block

In Simulink blocks, *direct feedthrough* means that the output of a block is controlled directly by the value of an input port signal. In nondirect feedthrough, the value of the output signal does not depend on the value of the input signal in at least one function during the simulation.

By default, MATLAB Function blocks have direct feedthrough enabled. If you disable direct feedthrough, the Simulink semantics 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.

To use nondirect feedthrough:

- Enable function inlining of the MATLAB Function block by using `coder.inline` in the top-level function body.
- In the Ports and Data Manager, in the **MATLAB Function Block Editor**, select **Edit Data** on the **Editor** tab and clear the **Allow direct feedthrough** check box. For more information, see “Ports and Data Manager” on page 44-29.

Tip Do not program outputs to rely on inputs or updated persistent variables. For example, do not use this code in a nondirect feedthrough block:

```
counter = counter + 1;      % update state
output = counter;           % compute output based on updated state
```

Instead, use this code:

```
output = counter;           % compute output based on current state
counter = counter + 1;      % update state
```

See Also

MATLAB Function

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4
- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

System Objects in Simulink

MATLAB System Block

In this section...

Why Use the MATLAB System Block?

System objects let you implement algorithms using the MATLAB language. The MATLAB System block enables you to use System objects in Simulink.

The MATLAB System block lets you:

- Share the same System object in MATLAB and Simulink
- Dedicate integration of System objects with Simulink
- Unit test your algorithm in MATLAB before using it in Simulink
- Customize dialog box customization
- Simulate efficiently with better initialization
- Handle states
- Customize block icons with port labels
- Access two simulation modes

Choosing the Right Block Type

There are several mechanisms for including MATLAB algorithms in Simulink, such as:

- MATLAB System block
- MATLAB Function block
- Interpreted MATLAB Function block
- Level-2 MATLAB S-Function block

For help on choosing the right block, see “Comparison of Custom Block Functionality” on page 40-5.

System Objects

Before you use a MATLAB System block, you must have a System object to associate with the block. A System object is a specialized kind of MATLAB class. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time.

For more information on creating System objects, see “Customize System Objects for Simulink”.

Note To use your System object in the Simulink environment, it must have a constructor that you can call with no arguments. By default, the System object constructor has this capability and you do not need to define your own constructor. However, if you create your own System object constructor, you must be able to call it with no arguments.

System objects exist in other MATLAB products. MATLAB System block supports only the System objects written in the MATLAB language. In addition, if a System object has a corresponding Simulink block, you cannot implement a MATLAB System block for it.

Interpreted Execution or Code Generation

You can use MATLAB System blocks in Simulink models for simulation via interpreted execution or code generation.

- With interpreted execution, the model simulates the block using the MATLAB execution engine.
- With code generation, the model simulates the block using code generation (requires the use the subset of MATLAB code supported for code generation). For a list of supported functions, see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2.

Default Input Signal Attributes

If a MATLAB System block has one or more inputs that are unconnected to another block’s output port or connected to a port that has underspecified attributes, the default input signal attributes for the unspecified attributes are:

Data Attribute	Default
Data Type	double
Size	[1 1] scalar
Complexity	real

MATLAB System Block Limitations

These capabilities are currently not supported.

Category	Limitation Description	Workaround
System Objects	Tunable logical and character vector properties of the System object are nontunable parameters in the MATLAB System block.	—
Data Types	• The MATLAB System block does not support virtual buses as input or output. • System objects cannot use user-defined opaque data types.	—
Sample Time	Cannot use MATLAB System blocks to model continuous time or multirate systems.	—

Category	Limitation Description	Workaround
Linearizations	Cannot use Jacobian based linearization.	—
Global Variables	Global variables defined in the model Configuration Parameters **Simulation Target** > **Custom Code** pane and referenced by the System object are not shared with Stateflow and the MATLAB Function block.	Turn on the **Import custom code option** in the **Simulation Target** pane of the Configuration Parameters dialog box.
Debugging	MATLAB debugging for code-generation-based simulation.	Set the MATLAB System block **Simulate using** parameter to `Interpreted execution`, and then debug. When you are done, set **Simulate using** back to `Code generation`.
Fixed-Point Tool	The Fixed-Point Tool does not return design min/max, min/max logging, and autoscaling information for MATLAB System blocks.	—
Model coverage analysis (Simulink Coverage software)	Simulink Coverage cannot perform model analysis for MATLAB System block with **Simulate using** parameter set to `Interpreted execution`.	—
Check model compatibility (Simulink Design Verifier software)	Simulink Design Verifier cannot perform compatibility checks for a model or subsystem that contains a MATLAB System block.	—

MATLAB System and System Objects Examples

For examples of MATLAB System and System objects, see:

Example	Description
System Identification for an FIR System Using MATLAB System Blocks	This example shows how to use the MATLAB System block to implement Simulink blocks using a System object. It highlights two MATLAB System blocks. Access the MATLAB source code for each System object by clicking the `Source code` link from the block dialog box.
Variable-Size Input and Output Signals Using MATLAB System Blocks	This example shows how to use the MATLAB System block to implement Simulink blocks with variable-size input and output signals. Due to the use of variable-size signals, the example uses propagation methods.
Illustration of Law of Large Numbers Using MATLAB System Blocks	This example shows how to use MATLAB System blocks to illustrate the law of large numbers. Due to the use of MATLAB functions not supported for code generation, the example uses propagation methods and interpreted execution.

Example	Description
Using Buses with MATLAB System Blocks	This example shows how to use MATLAB System blocks with nonvirtual buses at input or output. Due to the use Simulink buses, the example uses propagation methods. The example defines the bus types in the MATLAB base workspace using model callbacks.

See Also

MATLAB System

Related Examples

- “Implement a MATLAB System Block” on page 45-6
- “Change Blocks Implemented with System Objects” on page 45-8
- “Change Block Icon and Port Labels” on page 45-14
- “Add and Implement Propagation Methods” on page 45-27
- “Use System Objects in Feedback Loops” on page 45-16
- “Troubleshoot System Objects in Simulink” on page 45-36

More About

- “Customize System Objects for Simulink”
- “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19
- “Simulation Modes” on page 45-17
- “Simulink Engine Interaction with System Object Methods” on page 45-24
- “Nonvirtual Buses and MATLAB System Block” on page 45-15
- “Considerations for Using System Objects in Simulink” on page 45-22
- “Comparison of Custom Block Functionality” on page 40-5

Implement a MATLAB System Block

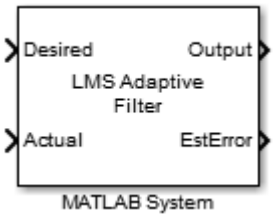
Implement a block and assign a System object to it. You can then explore the block to see the effect.

1 Create a new model and add the MATLAB System block from the User-Defined Functions library.

2 In the block dialog box, from the **New** list, select `Basic`, `Advanced`, or `Simulink Extension` if you want to create a new System object from a template. Modify the template according to your needs and save the System object.

3 Enter the full path name for the System object in the **System object name**. Click the list arrow. If valid System objects exist in the current folder, the names appear in the list.

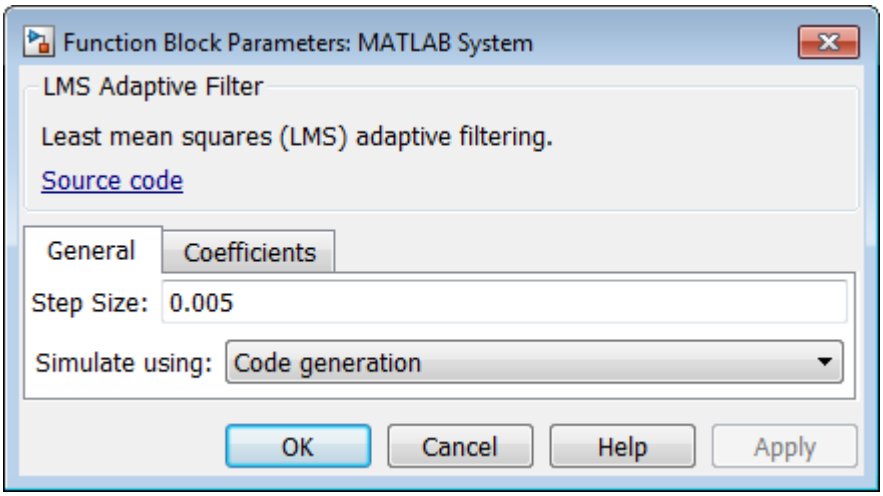
The MATLAB System block icon and port labels update to those of the corresponding System object. For example, suppose you selected a System object named lmsSysObj in your current folder. The block updates as shown in the figure:



Note After you associate the block with a System object class name, you cannot assign a new System object using the same MATLAB System block dialog box. Instead, right-click the MATLAB System block, select **Block Parameters (MATLABSystem)** and enter a new class name in **System object name**.

Understanding the MATLAB System Block

1 Double-click the block. The MATLAB System dialog box reflects the System object parameters. The dialog box usually includes a **Source code** link that leads to the System object class file. For example:



The **Source code** link appears if the System object uses MATLAB language. It does not appear if you have:

- Converted the System object to P-code
- Overridden the default behavior using the `getHeaderImpl` method

2 Click **Source code** and observe that the public and active properties in the System object appear in the MATLAB System block dialog box as block parameters.

3 Select how you want the model to simulate the block using the **Simulate using** parameter. (This parameter appears at the bottom of each MATLAB System block if there is only one tab, or the bottom of the first of multiple tabs.)

See Also

Related Examples

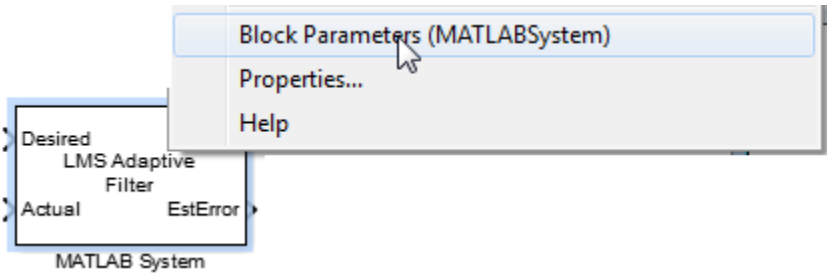
- “Change Blocks Implemented with System Objects” on page 45-8

More About

- “MATLAB System Block” on page 45-2
- “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19
- “Simulation Modes” on page 45-17

Change Blocks Implemented with System Objects

To implement a block with another System object, right-click the MATLAB System block and select Block Parameters (MATLABSystem). Then, use the block dialog box to identify a new class name in **System object name**. For more information, see “Implement a MATLAB System Block” on page 45-6.



See Also

Related Examples

- “System Identification for an FIR System Using MATLAB System Blocks”

Call Simulink Functions from MATLAB System Block

A Simulink Function is a graphical representation of a computational unit in the Simulink environment. Once you create the Simulink function, it can be executed by any computational unit and can be called in multiple places. You can only call a Simulink function inside the `stepImpl`, `outputImpl`, or `updateImpl` method of a System object. See “Simulink Functions Overview” on page 10-113 for more information on Simulink functions.

Note `Interpreted mode` is not supported for a model that contains a MATLAB System block calling a Simulink Function.

Create a Simulink Function Block

Set up a Simulink Function block that implements a simple function such as `y = 2 * u`.

1. Open a new model from **Simulink Editor** and add a Simulink Function block by typing Simulink Function on Simulink canvas.
2. When **Function prototype** window opens, type `y = timestwo_func(u)` as the function definition. This indicates that you define a function called `timestwo_func` that takes `u` as the input argument and produces `y` as the output argument. Alternatively, define the function name from Simulink Function block parameters.
3. Double-click the Simulink Function block and observe that a Trigger Port function block appears as well as an input and an output argument block.
4. Double-click the Trigger Port block in the Simulink Function, and observe that the **Function visibility** is set to `scoped`. To learn more about `scoped` Simulink Functions, see “Scoped and Global Simulink Function Blocks Overview” on page 10-147.
5. Add a Gain block and set its value to 2. Connect it with the input and output argument block. Click **Navigate Up to Parent** to return to the main model.

Note To determine which Simulink Function a MATLAB System block is calling, turn on the function tracing lines. In the **Debug** tab, select **Information Overlays > Function Connectors**.

Create a MATLAB System Block and Define System Object

Drag a MATLAB System block into the model and implement a System object to this block.

1. Add a MATLAB System block to your Simulink model.
2. In the block dialog box, from the `New` list, select **Basic**. This opens a basic System object template for you to type your code.
3. To subclass an object from `matlab.System`, replace `Untitled` with the name of your System object. For this example, name it `SimulinkFcnCaller`.
4. In the `stepImpl` method of your System object, declare the Simulink Function using `getSimulinkFunctionNamesImpl`.

   See an example of the System object code below. To learn more about how to write a System object, see “Define Basic System Objects”.

```
classdef SimulinkFcnCaller < matlab.System
    % Public, tunable properties
```

```
    % SimulinkFcnCaller calls a Simulink Function from a
    % MATLAB System block to multiply the signal's value by 2.

    methods(Access = protected)

        function y = stepImpl(obj,u)
            % Implement algorithm. Calculate y as a function of input u and
            % discrete states.
            y = timestwo_func(u);
        end
        function names = getSimulinkFunctionNamesImpl(obj)
            % Use 'getSimulinkFunctionNamesImpl' method to declare
            % the name of the Simulink Function that will be called
            % from the MATLAB System block's System object code.
            names = {'timestwo_func'};
        end
    end
end
```

5 Save the file and name it `SimulinkFcnCaller.m`.

Call a Simulink Function in a Subsystem from a MATLAB System Block

The hierarchy of the Simulink Function block affects the function calls in the System object. For example, if the Simulink function is defined at a higher hierarchy in the Simulink model, the function is defined for all blocks in that hierarchy. If the Simulink function is defined at a lower hierarchy, you need to qualify the Subsystem and the function name. For example, suppose you have a Subsystem that contains a Simulink Function block at the same level as a MATLAB System block. When a Simulink Function block is placed in a subsystem, the function name is not visible to the outside the subsystem. You can call the Simulink Function block by qualifying the function name using the Subsystem name in your System object. To qualify the Subsystem and the function name follow these steps:

1 In the `stepImpl` method of your System object code, call the Simulink Function using dot notation. For example, in the code `y = Subsystem1.timestwo_func(u)`, `Subsystem1` corresponds to the Subsystem, and `timestwo_func` corresponds to the Simulink Function name.

2 Similarly, declare the Subsystem and the Simulink Function in the `getSimulinkFunctionNamesImpl` method using the dot notation. The System object code shows the `timestwo` example written for a Simulink Function defined at a lower hierarchy than the MATLAB System block.

```
classdef SimulinkFcnCallerQualified < matlab.System

    % SimulinkFcnCallerQualified calls a Simulink Function embedded in a Subsystem
    % from a MATLAB System block, and multiplies the signal's value by 2.

    methods(Access = protected)

        function y = stepImpl(obj,u)
            % Use the '.' notation to call a scoped Simulink Function from
            % a Simulink Function block.
            % Subsystem1 corresponds to  the block name, where
            % timestwo_funct is the Simulink Function name.
            y = Subsystem1.timestwo_func(u);
        end

        function names = getSimulinkFunctionNamesImpl(obj)
            % Use the 'getSimulinkFunctionNamesImpl' method with the '.'
            % notation to declare the name of a Simulink Function in
            % MATLAB System block's System object code.
            names = {'Subsystem1.timestwo_func'};
        end
    end
end
```

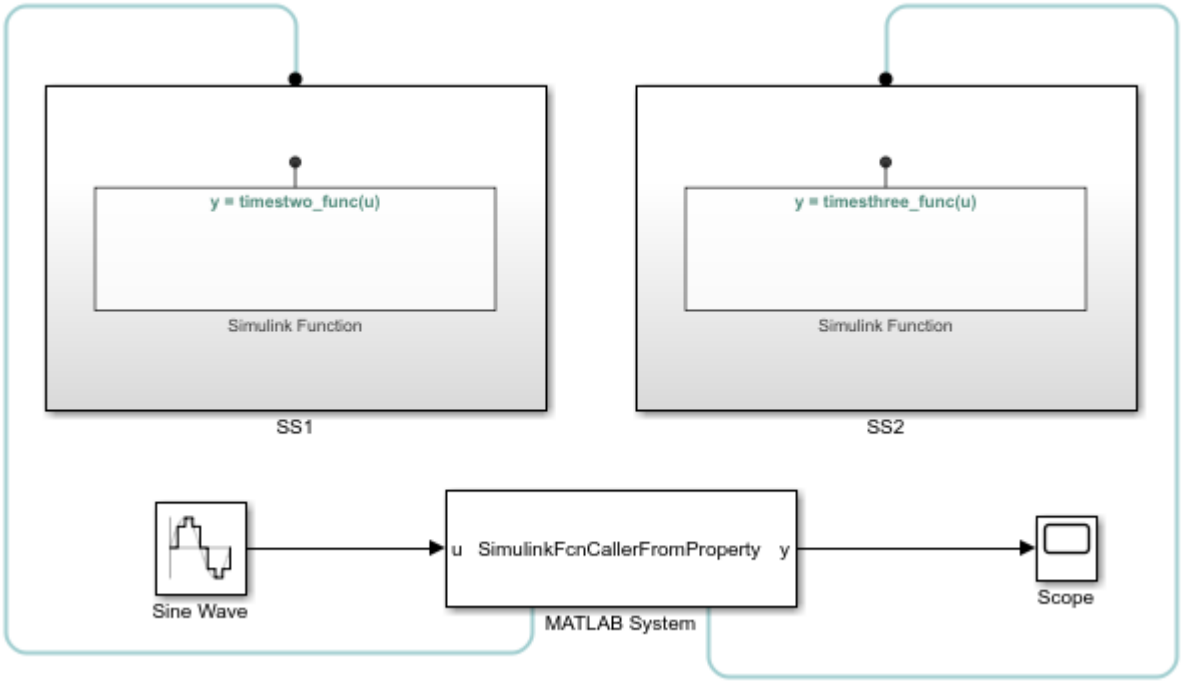
3 Save the System object file and name it `SimulinkFcnCallerQualified.m`.

Call Simulink Functions from a MATLAB System Block

This example shows two `Simulink Functions` conditionally called by a `MATLAB System` block using the nontunable properties of the System object®.

The `MATLAB System` block calls one of the `Simulink Functions` inside two different subsystems, depending on the value of the signal coming from the `Sine Wave` block. If the value of the signal is less than 10, the `MATLAB System` block calls the `timestwo_func` `Simulink Function` inside the `SS1` `Subsystem` block. If the value is larger than 10, it calls the `timesthree_func` in the `SS2` `Subsystem` block.

Function names are defined as nontunable properties, are switched from string to functions using the `str2func` function. Then, these functions are declared as properties in the `getSimulinkFunctionNamesImpl` method.



See Also

`getSimulinkFunctionNamesImpl`

More About

- “Define Basic System Objects”
- “Use a MATLAB Function block to call a Simulink Function block” on page 10-130
- “Use a Function Caller block to call a Simulink Function block” on page 10-128

Specify Sample Time for MATLAB System Block

The sample time of a block is a parameter that indicates when the block produces an output, and eventually updates its internal state. To specify sample time for MATLAB System block, implement the `getSampleTimeImpl` method with a call to `createSampleTime`. To query the MATLAB System block for current sample time and simulation time, use the `getSampleTime` and `getCurrentTime` methods. For more information, see “Specify Sample Time for MATLAB System Block System Objects” on page 45-104.

Types of Sample Time for MATLAB System Block

If you use discrete sample time on page 7-13 in simulation, Simulink only calculates the output of a simulation for each of the fixed time intervals. To specify discrete sample time, in the `createSampleTime`, set `'Type'` to `'Discrete'` and set the `'SampleTime'` property.

When the sample time is inherited on page 7-14, Simulink determines the best sample time for the block based on the block's role within the model. To specify inherited sample time, in the `createSampleTime`, set `'Type'` to `'Inherited'`. When using inherited sample time, you can alter or error out on specific sample times by specifying the `'AlternatePropagation'` or `'ErrorOnPropagation'` Name-Value pair.

In fixed-in-minor-step on page 7-14 sample time, Simulink does not execute the block at the minor time steps, the updates only occur at the major time steps. To specify the fixed-in-minor time step, in the `createSampleTime`, set `'Type'` to `'Fixed In Minor Step'`.

To use controllable on page 7-15 sample time, configure a block to specify controllable sample time with a resolution T_{base}, where T_{base} is the smallest allowable time interval between block executions. A block using controllable sample time can be dynamically set to execute at n multiples of T_{base}, then the block's next execution is

$$T_{next} = n\,T_{base} + T$$

To specify T_{base} in MATLAB System block, in the `createSampleTime`, set `'Type'` to `'Controllable'` and set the `'TickTime'` property to T_{base}. You can set the n in your MATLAB System block using `setNumTicksUntilNextHit`.

For more information on types of sample time, see “Types of Sample Time” on page 7-13. To see an example on how to control the sample time of the MATLAB System block using System object methods, see “Specify Sample Time for MATLAB System Block System Objects” on page 45-104.

See Also

`getSampleTimeImpl` | `getSampleTime` | `getCurrentTime` | `createSampleTime` | `setNumTicksUntilNextHit`

More About

- “What Is Sample Time?” on page 7-2
- “Specify Sample Time for MATLAB System Block System Objects” on page 45-104
- “Types of Sample Time” on page 7-13

See Also

Related Examples

- “Pulse Width Modulation Using MATLAB System Block”

Change Block Icon and Port Labels

To change the icon appearance of your block, you must add specific methods to your System object. For example, to define port labels, implement `getInputNamesImpl` and `getOutputNamesImpl`.

1. To define the icon, implement the `getIconImpl` method.
2. To define the port labels, implement `getInputNamesImpl` to change the input and `getOutputNamesImpl` to change the output port labels.

If you do not implement these methods, by default the System object uses the input and output port names from the `stepImpl` method. If you are using nondirect feedthrough, by default the System object uses the input names from `updateImpl` and the output port names from `outputImpl`.

Modify MATLAB System Block Dialog

To change the MATLAB System block dialog, implement `getPropertyGroupsImpl` and inside the method implement the following classes:

Description	matlab.system.display Methods
Define header text for property group.	`matlab.system.display.Header`
Group properties together.	`matlab.system.display.Section`
Group properties into a separate tab.	`matlab.system.display.SectionGroup`

Change the MATLAB System Block Icon to an Image

You can change the image of MATLAB System block in MATLAB Editor. For a list of accepted image files, see `image`. To use an existing image file for the MATLAB System block:

1. Double-click your MATLAB System block.
2. In the block dialog box, click the **Source code**. The MATLAB Editor that contains the System object code opens.
3. In the MATLAB Editor, from the **System Block** drop-down list, select **Add Image Icon**.
4. In the **Add image icon** dialog window, click **Browse** to select an image of your choice.
5. Click **OK** to insert the corresponding code for the `getIconImpl` method in your System object.

For more information, see “Customize System Block Appearance” on page 45-60.

See Also

MATLAB System | `matlab.system.display.Icon`

Related Examples

- “System Identification for an FIR System Using MATLAB System Blocks”
- “Customize System Objects for Simulink”

Nonvirtual Buses and MATLAB System Block

The MATLAB System block supports nonvirtual buses as input and output signals. The corresponding System object input or output must be a MATLAB structure whose fields match those defined by the nonvirtual bus. If the System object output is a MATLAB structure, it must define propagator methods. In addition, the `getOutputDataTypeImpl` method must return the name of the corresponding bus object. This bus object must exist in the base workspace or a data dictionary linked to the model.

Note If the output is the same bus type as the input, do not use the `propagatedInputDataType` method to obtain the name of the bus object. Instead, you must return the name of the bus object directly.

See Also

Related Examples

- Using Buses with MATLAB System Blocks

More About

- “Customize System Objects for Simulink”

Use System Objects in Feedback Loops

If your algorithm needs to process nondirect feedthrough data through the System object, use the `isInputDirectFeedthroughImpl`, `outputImpl`, and `updateImpl` methods. These methods process nondirect feedthrough data through a System object.

Most System objects use direct feedthrough, where the object's input is needed to generate the output. 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 on internal states and not directly on the inputs. The inputs, or a subset of the inputs, are used to update the object states. For these objects, calculating the output is separated from updating the state values. This enables you to use an object as a feedback element in a feedback loop.

This example shows how to implement a delay object with nondirect feedthrough.

1. In MATLAB, select **New > System object > Basic**.
2. Select **Insert Method** and implement the `outputImpl` and `updateImpl` methods.

   `outputImpl` calculates outputs from states and/or inputs. `updateImpl` updates state values from inputs.

   When implementing the `outputImpl` method, do not access the System object inputs for which the direct feedthrough flag is false.
3. If the System object supports code generation and does not use propagation, Simulink can automatically infer the direct feedthrough settings from the System object MATLAB code. However, if the System object does not support code generation, the default `isInputDirectFeedthroughImpl` method returns false (no direct feedthrough). In this case, override this method to specify nondirect feedthrough behavior.

The processing of the nondirect feedthrough changes the way that the software calls the System object methods within the context of the Simulink engine.

See Also

Related Examples

- "System Identification for an FIR System Using MATLAB System Blocks"

More About

- "Simulink Engine Interaction with System Object Methods" on page 45-24
- "Customize System Objects for Simulink"

Simulation Modes

Interpreted Execution vs. Code Generation

You can use MATLAB System block in Simulink models for simulation via interpreted execution or code generation. Implementing a MATLAB System block with a valid System object class name enables the **Simulate using** parameter. This parameter appears at the bottom of the MATLAB System block dialog if there is only one tab, or the bottom of the first of multiple tabs. Use the **Simulate using** parameter to control how the block simulates. The table describes how to choose the right value for your purpose.

- With interpreted execution, the model simulates the block using the MATLAB execution engine.

  Note With interpreted execution, if you set the **Use division for fixed-point net slope computation** parameter to `On` or `Use division for reciprocals of integers only` in the Configuration Parameters dialog box, you might get unoptimized numeric results. These bad numeric results are because MATLAB code does not support this parameter.

- With code generation, the model simulates the block using code generation, using the subset of MATLAB code supported for code generation.

Action	Select	Pros	Cons
Upon first model run, simulate and generate code for MATLAB System using only the subset of MATLAB functions supported for code generation. Choosing this option causes the simulation to run the generated code.	`Code generation` (default)	Potentially better performance.	System object is limited to the subset of MATLAB functions supported for code generation. Simulation may start more slowly.
Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.	`Interpreted execution`	System object can contain any supported MATLAB function. Faster startup time.	Potentially slower performance. If the MATLAB functions in the System object do not support code generation, the System object must contain propagation methods.

To take advantage of faster performance, consider using propagation methods in your System object. For more information, see “Add and Implement Propagation Methods” on page 45-27.

Simulation Using Code Generation

While simulating and generating code for one or more simulation targets (in this case, System object blocks), the model displays status messages in the bottom left of the Simulink Editor window. A model can have multiple copies of the same MATLAB System block. Blocks are considered the same if they,

- Use the same System object.
- Have inputs and tunable parameters that have identical signals, data types, and complexities.

- Have nontunable parameters that have the same value.

When the model has multiple copies of the same block, the software does not regenerate the code for each block. It reuses the code from the first time that code was generated for one of these blocks. The status messages reflect this and does not show status messages for each of these blocks.

When the code generation process is complete, Simulink creates a MEX-file for the generated code.

See Also

MATLAB System

Related Examples

- “Implement a MATLAB System Block” on page 45-6
- “Change Blocks Implemented with System Objects” on page 45-8
- “Change Block Icon and Port Labels” on page 45-14
- “Add and Implement Propagation Methods” on page 45-27
- “Use System Objects in Feedback Loops” on page 45-16
- “Troubleshoot System Objects in Simulink” on page 45-36

More About

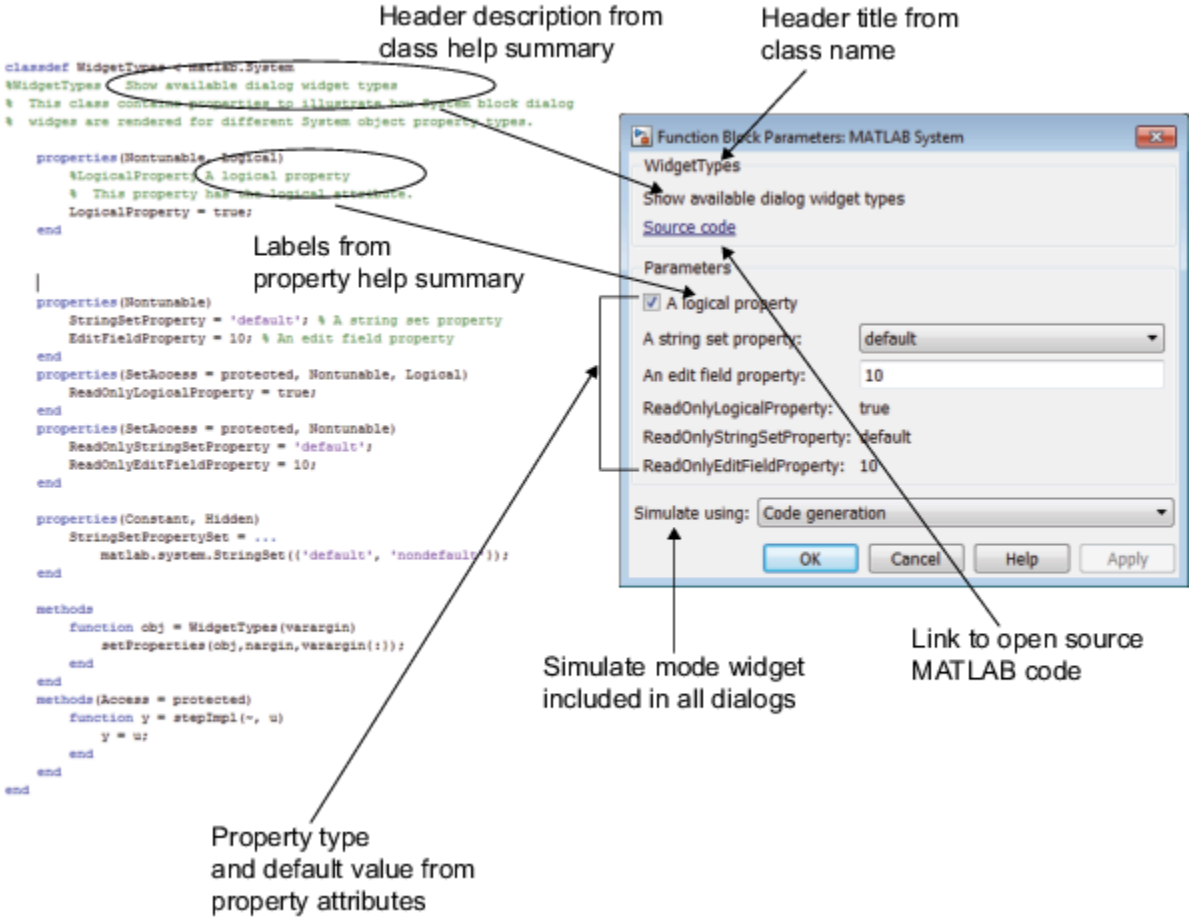
- “Customize System Objects for Simulink”
- “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19
- “Simulink Engine Interaction with System Object Methods” on page 45-24
- “Nonvirtual Buses and MATLAB System Block” on page 45-15
- “Considerations for Using System Objects in Simulink” on page 45-22
- “Comparison of Custom Block Functionality” on page 40-5

Mapping System Object Code to MATLAB System Block Dialog Box

The System object source code controls the appearance of the block dialog box. This section maps the System object code to the block dialog box using the “System Identification for an FIR System Using MATLAB System Blocks” example. This example uses two System objects, one that uses default System object to block dialog box mapping, and one that uses a custom mapping.

System Object to Block Dialog Box Default Mapping

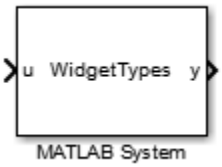
The following figure shows how the source code corresponds to the dialog box elements when you do not customize the dialog using the `getHeaderImpl` or `getPropertyGroupsImpl` methods. (The link to open the source code and the **Simulate using** parameter appear on all MATLAB System block dialog boxes.)



The Delay block from the System Identification for an FIR System Using MATLAB System Blocks is an example of a block that uses a System object that draws the dialog box using the default mapping. This block has one input and one output.

This block uses a System object that has direct feedthrough set to false (nondirect feedthrough). This setting means that the System object does not directly use the input to compute the output, enabling

the model to use this block safely in a feedback system without introducing an algebraic loop. For more information on nondirect feedthrough, see “Use System Objects in Feedback Loops” on page 45-16.

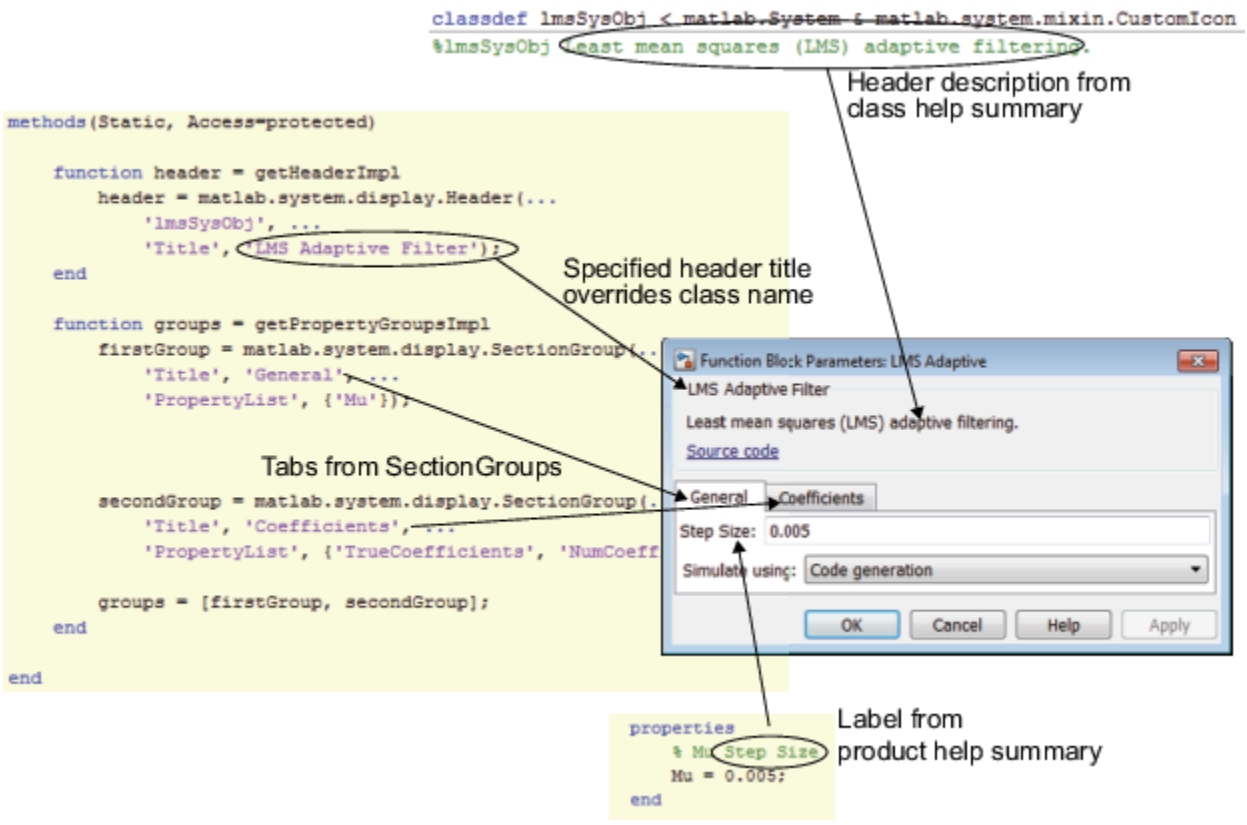


For an example of a custom block dialog box, see “System Object to Block Dialog Box Custom Mapping” on page 45-20.

System Object to Block Dialog Box Custom Mapping

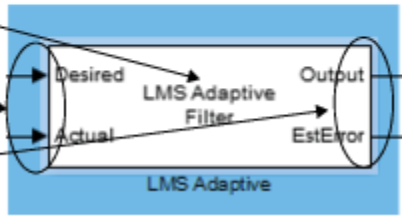
The LMS Adaptive block is an example of a block with a custom header and property groups. The System object code uses the `getHeaderImpl` and `getPropertyGroupsImpl` methods from `matlab.System` to customize these block dialog elements.

The LMS Adaptive Filter block estimates the coefficients of an unknown system (formed by the Unknown System and Delay blocks). Its inputs are the desired signal and the actual signal. Its outputs are the estimated signal and the vector norm of the error in the estimated coefficients. It uses the lmsSysObj System object.



The source code for this System object also defines two input and output ports for the block.

```
function num = getNumInputsImpl(~)
    num = 2;
end

function num = getNumOutputsImpl(~)
    num = 2;
end

function icon = getIconImpl(~)
    icon = sprintf('LMS Adaptive\nFilter');
end

  function [name1, name2] = getInputNamesImpl(~)
      name1 = 'Desired';
      name2 = 'Actual';
  end

  function [name1, name2] = getOutputNamesImpl(~)
      name1 = 'Output';
      name2 = 'EstError';
  end
```



See Also

More About

- “Change Block Icon and Port Labels” on page 45-14
- “Modify MATLAB System Block Dialog” on page 45-14

Considerations for Using System Objects in Simulink

In this section...

There are differences in how you can use System objects in a MATLAB System block in Simulink versus using the same object in MATLAB. You see these differences when working with variable-size signals and tunable parameters and when using System objects as properties.

Variable-Size Signals

To use variable-size signals in a System object, you must implement propagation methods. In particular, use the `isOutputFixedSizeImpl` method to specify if an output is variable-size or fixed-size. This method is needed for interpreted execution and code generation simulation methods.

Tunable Parameters

Simulink registers public tunable properties of a System object as tunable parameters of the corresponding MATLAB System block. If a System object property is tunable, it is also tunable in the MATLAB System block. At runtime, you can change the parameter using one of the following approaches. The change applies at the top of the simulation loop.

- At the MATLAB command line, use the `set_param` to change the parameter value.
- In the Simulink editor, edit the MATLAB System block dialog box to change the parameter value, and then update the block diagram.

You cannot change public tunable properties from System object internal methods such as `stepImpl`.

During simulation, setting an invalid value on a tunable parameter causes an error message and stops simulation.

System Objects as Properties

The MATLAB System block allows a System object to have other System objects as public or private properties. However:

- System objects and other MATLAB objects stored as public properties are read only. As a result, you cannot set the value of the parameter, you can only get the value of a parameter.
- System objects stored as property values appear dimmed in the MATLAB System block dialog box.

Default Property Values

MATLAB does not require that objects assign default values to properties. However, in Simulink, if your System object has properties with no assigned default values, the associated dialog box parameter requires that the value data type be a built-in Simulink data type.

System Objects in For Each Subsystems

To use the MATLAB System block within a For Each Subsystem block, implement the `supportsMultipleInstanceImpl` method. This method should return `true`. The MATLAB System block clones the System object for each For Each Subsystem iteration.

Input Validation

In Simulink, use the `validateInputsImpl` method to validate only attributes (size, data type, and complexity) of the input. Do not use this method to validate the value of the input.

See Also

MATLAB System

Related Examples

- “Implement a MATLAB System Block” on page 45-6
- “Change Blocks Implemented with System Objects” on page 45-8
- “Change Block Icon and Port Labels” on page 45-14
- “Add and Implement Propagation Methods” on page 45-27
- “Use System Objects in Feedback Loops” on page 45-16
- “Troubleshoot System Objects in Simulink” on page 45-36

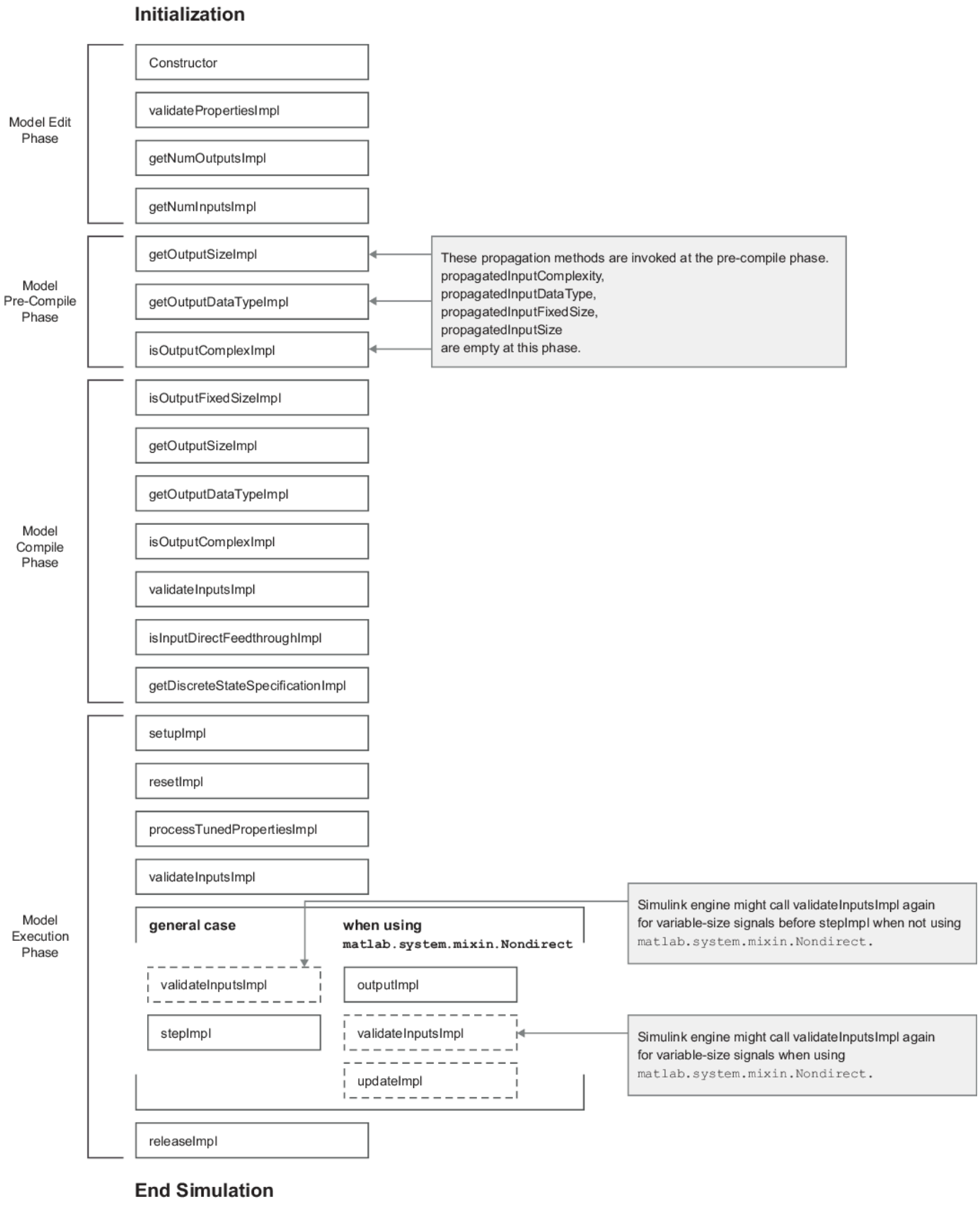
More About

- “Customize System Objects for Simulink”
- “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19
- “Simulink Engine Interaction with System Object Methods” on page 45-24
- “Simulation Modes” on page 45-17
- “Nonvirtual Buses and MATLAB System Block” on page 45-15
- “Comparison of Custom Block Functionality” on page 40-5

Simulink Engine Interaction with System Object Methods

Simulink Engine Phases Mapped to System Object Methods

This diagram shows a process view of the order in which the MATLAB System block invokes System object methods within the context of the Simulink engine.


Initialization
Model Edit Phase
Constructor
validatePropertiesImpl
getNumOutputsImpl
getNumInputsImpl
Model Pre-Compile Phase
getOutputSizeImpl
getOutputDataTypeImpl
isOutputComplexImpl
These propagation methods are invoked at the pre-compile phase.
propagatedInputComplexity,
propagatedInputDataType,
propagatedInputFixedSize,
propagatedInputSize
are empty at this phase.
Model Compile Phase
isOutputFixedSizeImpl
getOutputSizeImpl
getOutputDataTypeImpl
isOutputComplexImpl
validateInputsImpl
isInputDirectFeedthroughImpl
getDiscreteStateSpecificationImpl
Model Execution Phase
setupImpl
resetImpl
processTunedPropertiesImpl
validateInputsImpl
general case
when using
matlab.system.mixin.Nondirect
validateInputsImpl
outputImpl
stepImpl
validateInputsImpl
updateImpl
releaseImpl
Simulink engine might call validateInputsImpl again for variable-size signals before stepImpl when not using matlab.system.mixin.Nondirect.
Simulink engine might call validateInputsImpl again for variable-size signals when using matlab.system.mixin.Nondirect.
End Simulation

Note the following:

- Simulink calls the `stepImpl`, `outputImpl`, and `updateImpl` methods multiple times during simulation at each time step. Simulink typically calls other methods once per simulation.
- The Simulink engine calls the `isOutputFixedSizeImpl`, `getDiscreteStateSpecificationImpl`, `isOutputComplexImpl`, `getOutputDataTypeImpl`, `getOutputSizeImpl` when using propagation methods.
- Simulink calls `saveObjectImpl` and `loadObjectImpl` for saving and restoring SimState, the Simulation Stepper, and Fast Restart.
- Default implementations save and restore all properties with public access, including `DiscreteState`.

See Also

MATLAB System

Related Examples

- “Implement a MATLAB System Block” on page 45-6
- “Change Blocks Implemented with System Objects” on page 45-8
- “Change Block Icon and Port Labels” on page 45-14
- “Add and Implement Propagation Methods” on page 45-27
- “Use System Objects in Feedback Loops” on page 45-16
- “Troubleshoot System Objects in Simulink” on page 45-36

More About

- “Customize System Objects for Simulink”
- “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19
- “Simulation Modes” on page 45-17
- “Nonvirtual Buses and MATLAB System Block” on page 45-15
- “Considerations for Using System Objects in Simulink” on page 45-22
- “Comparison of Custom Block Functionality” on page 40-5

Add and Implement Propagation Methods

In this section...

When to Use Propagation Methods

Propagation methods define output specifications. Use them when the output specifications cannot be inferred directly from the inputs during Simulink model compilation.

Consider using propagation methods in your System object when:

- The System object requires access to all MATLAB functions that do not support code generation, which means that you cannot generate code for simulation. You must use propagation methods in this case. Use these methods to specify information for the outputs.
- You want to use variable-size signals.
- You do not care whether code is generated, but you want to improve startup performance. Use propagation methods to specify information for the inputs and outputs, enabling quicker startup time.

At startup, the Simulink software tries to evaluate the input and output ports of the model blocks for signal attribute propagation. In the case of MATLAB System blocks, if the software cannot perform this evaluation, it displays a message prompting you to add propagation methods to the System object.

Implement Propagation Methods

Simulink evaluates the uses of the propagation methods to evaluate the input and output ports of the MATLAB System block for startup.

Each method has a default implementation, listed in the **Default Implementation Should Suffice if** column. If your System object does not use the default implementation, you must implement a version of the propagation method for your System object.

Description	Propagation Method	Default Implementation Should Suffice if	Example
Gets dimensions of output ports. The associated method is `getOutputSize`.	`getOutputSizeImpl`	• Only one input • Only one output • An input size that is the same as the output size	• FindIfFixedInput or FindIfVarSizeInput block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis and Plot block in Illustration of Law of Large Numbers

Description	Propagation Method	Default Implementation Should Suffice if	Example
Gets data types of output ports. The associated method is `getOutputDataType`.	`getOutputDataTypeImpl`	• Only one input • Only one output • Output data type always the same as the input data type	• FindIfFixedInput or FindIfVarSizeInput block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis and Plot block in Illustration of Law of Large Numbers
Indicates whether output ports are complex or not. The associated method is `isOutputComplex`.	`isOutputComplexImpl`	• Only one input • Only one output • Output complexity always the same as th input complexity	• FindIfFixedInput or FindIfVarSizeInput block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis and Plot block in Illustration of Law of Large Numbers
Whether output ports are fixed size. The associated method is `isOutputFixedSize`.	`isOutputFixedSizeImpl`	• Only one input • Only one output • Output and input are fixed-size	• FindIfFixedInput or FindIfVarSizeInput block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis and Plot block in Illustration of Law of Large Numbers
Gets the size, data type, and complexity of a discrete state property. The associated method is `getDiscreteStateSpecification`.	`getDiscreteStateSpecificationImpl`	No `DiscreteState` properties	N/A

See Also

More About

- “Customize System Objects for Simulink”

Share Data with Other Blocks

In this section...

Share data between MATLAB System and other blocks using the `global` keyword and the Data Store Memory block or `Simulink.Signal` object. You might need to use global data with a MATLAB System block if:

- You have an existing model that uses a large amount of global data, you are adding a MATLAB System block to this model, and you want to avoid cluttering your model with additional inputs and outputs.
- You want to scope the visibility of data to parts of the model.

Data Sharing with the MATLAB System Block

In Simulink, you store global data using data store memory. You implement data store memory using either Data Store Memory blocks or `Simulink.Signal` objects. How you store global data depends on the number and scope of your global variables.

Scoping Rules for Shared Data in MATLAB System Blocks

The MATLAB System block uses these scoping rules:

- If you use the same name for both Data Store Memory block and `Simulink.Signal` object, Data Store Memory block scopes the data to the model.
- A global variable resolves hierarchically to the closest Data Store Memory block with the same name in the model. The same global variable appearing in two different MATLAB System blocks might resolve to different Data Store Memory blocks depending on the hierarchy of the model. You can use this ability to scope the visibility of data to a subsystem.

Using Shared Data in MATLAB System Blocks

MATLAB System blocks support data store memory for:

- MATLAB structures (buses)
- Enumerated data types

How to Use Data Sharing with the MATLAB System Block

To use shared data in your MATLAB System block:

1. Declare a global variable in the System object that you associate with the MATLAB System block.

You can use the `global` keyword in these methods of the System object:

- `stepImpl`
- `outputImpl`
- `updateImpl`

2 Add a Data Store Memory block or `Simulink.Signal` object that has the same name as the global variable in the System object.

To share data between referenced models using the `Simulink.Signal` object, define the `Simulink.Signal` object in the base workspace and use the same global variable name as in the MATLAB System block.

Choose How to Store Shared Data

You can use Data Store Memory blocks or `Simulink.Signal` objects to store shared data.

Type of Data	Global Data Storage Method	Related Links
A small number of global variables in a single model that does not use model reference.	Data Store Memory blocks. **Note** Using Data Store Memory blocks scopes the data to the model.	"How to Use Data Store Memory Blocks for the MATLAB System Block" on page 45-30
A large number of global variables in a single model that does not use model reference.	`Simulink.Signal` objects defined in the model workspace. `Simulink.Signal` objects offer these advantages: • You do not have to add numerous Data Store Memory blocks to your model. • You can load the `Simulink.Signal` objects in from a MAT-file.	"How to Set Up Simulink.Signal Objects" on page 45-32
Data shared between multiple models (including referenced models).	`Simulink.Signal` objects defined in the base workspace **Note** If you use Data Store Memory blocks as well as `Simulink.Signal`, note that using Data Store Memory blocks scopes the data to the model.	"How to Set Up Simulink.Signal Objects" on page 45-32

How to Use Data Store Memory Blocks for the MATLAB System Block

1 Declare a global keyword in the System object methods that support globals. For example:

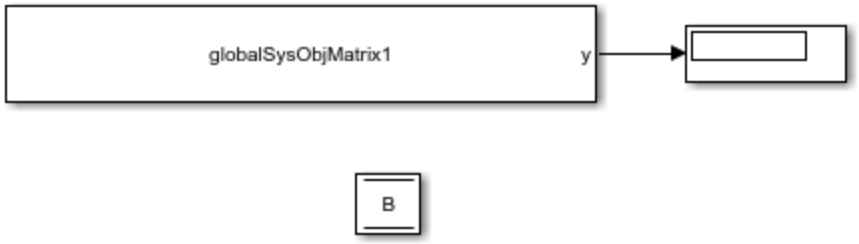
```
global A;
```

2. Add a MATLAB System block to your model.
3. Double-click the MATLAB System block and associate the System object.
4. Add a Data Store Memory block to your model and set:
   a. **Data store name** to match the name of the global variable in your MATLAB System block code.
   b. **Data type** to an explicit data type.

      The data type cannot be `auto`.
   c. **Signal type**.
   d. **Initial value**.

      The initial value of the Data Store Memory block cannot be unspecified.

Use Data Store Memory with the MATLAB System Block



This model demonstrates how a MATLAB System block uses the global data stored in Data Store Memory block `B`. The MATLAB System block is associated with the `globalSysObjMatrix1` System object. To see the completed model, open the `ex_globalsys_objmatrix1` model.

1. Drag these blocks into a new model:
   - MATLAB System
   - Data Store Memory
   - Display
2. Create a System object to associate with the MATLAB System block. To start, from the MATLAB System block, create a Basic System object template file.
3. In MATLAB Editor, create a System object with code like the following. Save the System object as `globalSysObjMatrix1.m`. The System object modifies `B` each time it executes.

```
classdef globalSysObjMatrix1 < matlab.System
    % Global/DSM support scalar example

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
        end

        function y = stepImpl(obj)
            global B;
            B(:,1)= B(:,1)+100;
            y = B;
        end
```

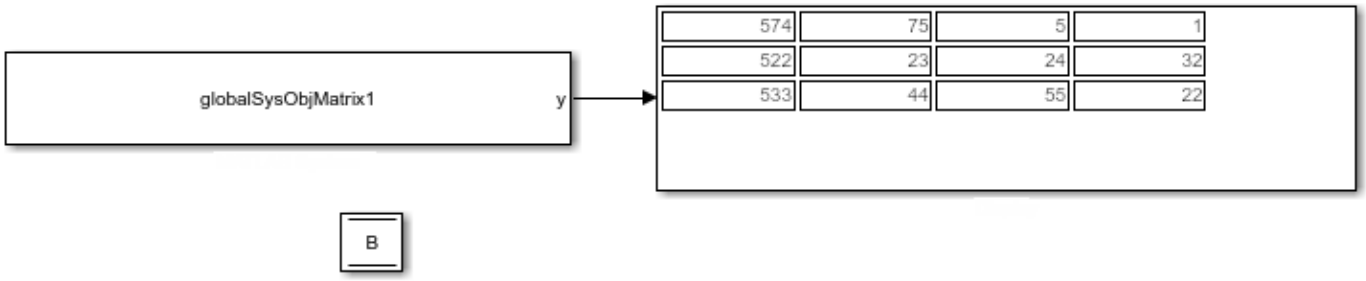
```
    end
end
```

4. Double-click the MATLAB System block and associate the `globalSysObjMatrix1` System object with the block.
5. In the model, double-click the Data Store Memory block `B`.
6. In the Signal Attributes tab, enter an initial value, for example:

   ```
   [74 75 5 1;22 23 24 32;33 44 55 22]
   ```

7. Simulate the model.

   The MATLAB System block reads the initial value of global data stored in `B` and updates the value of `B` each time it executes. This model executes for five time steps.



8. Save and close your model.

How to Set Up Simulink.Signal Objects

Create a `Simulink.Signal` object in the model workspace.

Tip Create a `Simulink.Signal` object in the base workspace to use the global data with multiple models.

---

1. In the Model Explorer, navigate to ***`model_name`*** > **Model Workspace** in the **Model Hierarchy** pane.

   Select **Add** > **Simulink Signal**.
2. Ensure that these settings apply to the `Simulink.Signal` object:
   a. Set **Data type** to an explicit data type.

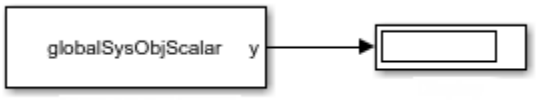
      The data type cannot be `auto`.
   b. Set **Dimensions** to be fully specified.

      The signal dimensions cannot be `-1` or inherited.
   c. Set the **Complexity**.
   d. Specify an **Initial value**.

      The initial value of the signal cannot be unspecified.

e Set **Name** to the name of the global variable.

Use a Simulink.Signal Object with a MATLAB System Block



This simple model demonstrates how a MATLAB System block uses a `Simulink.Signal` with signal B. The MATLAB System block is associated with the `globalSysObjScalar` System object. To see the completed model, open the `ex_globalsys_simulink_signal_share` model.

1. Drag these blocks into a new model:
   - MATLAB System
   - Display
2. Create a System object to associate with the MATLAB System block. To start, from the MATLAB System block, create a Basic System object template file.
3. In MATLAB Editor, create a System object. Save the System object as `globalSysObjScalar.m`. The System object modifies `B` each time it executes.

```
classdef globalSysObjScalar < matlab.System
    % Global/DSM support scalar example

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
        end

        function y = stepImpl(obj)
            global B;
            B= B+100;
            y = B;
        end
    end
end
```

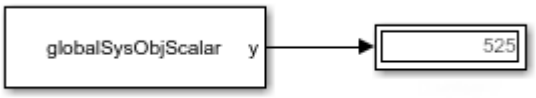
4. Double-click the MATLAB System block and associate the `globalSysObjScalar` System object with the block.
5. From the model, on the **Modeling** tab, click **Model Explorer**.
6. In the left pane of the Model Explorer, select the model workspace for this model.

   The **Contents** pane displays the data in the model workspace.
7. In the Model Explorer, in the **Model Hierarchy** pane, navigate to ***`model_name`*** > **Model Workspace**. In the **Contents** pane, set **Name** to `B`.
8. Navigate back to ***`model_name`*** > **Model Workspace**.
   - Select **Add** > **Simulink Signal**.
   - Make these settings for the `Simulink.Signal` object:

Attribute	Value
Data type	`double`
Complexity	`real`
Dimensions	1
Initial value	25

9 Simulate the model.

The MATLAB System block reads the initial value of global data stored in `B` and updates the value of `B` each time it executes. The model runs for five time steps.



10 Save and close your model.

Using Data Store Diagnostics to Detect Memory Access Issues

You can configure your model to provide run-time and compile-time diagnostics to avoid problems with data stores. Diagnostics are available in the Configuration Parameters dialog box and the Parameters dialog box for the Data Store Memory block. These diagnostics are available for Data Store Memory blocks only, not for `Simulink.Signal` objects. For more information on using data store diagnostics, see “Data Store Diagnostics” on page 73-3.

Limitations of Using Shared Data in MATLAB System Blocks

The MATLAB System block does not support data store memory for variable-sized data

Use Shared Data with P-Coded System Objects

If the System object is P-code, you must implement the `getGlobalNamesImpl` method to provide the global variable names you use in the System object. For example:

```
classdef GlobalSysObjMatrix < matlab.System
    % Matrix DSM support: Increment first row by 1 at each time step
    methods (Access = protected)
        function y = stepImpl(obj)
            global B;
            B(1,:) = B(1,:)+1;
            y = B;
        end

                function globalNames = getGlobalNamesImpl(~)
            globalNames = {'B'};
                end
    end
end
```

See Also

Data Store Memory | MATLAB System | `Simulink.Signal`

More About

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Local and Global Data Stores” on page 73-2

Troubleshoot System Objects in Simulink

In this section...

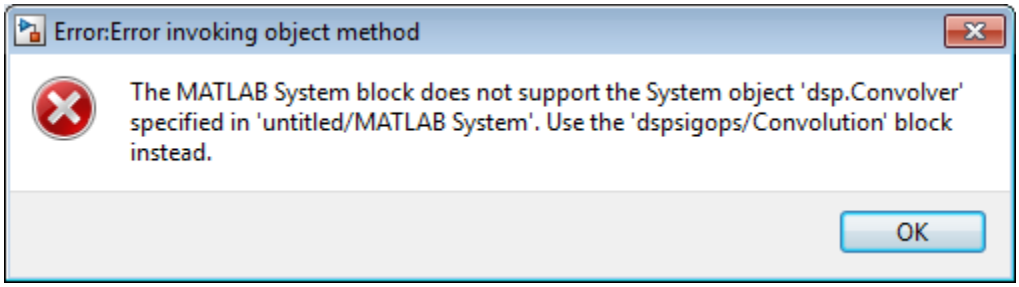
Class Not Found

The MATLAB System block **System object name** parameter requires that you enter the full path to the System object class. In addition:

- Check that the System object class is on your MATLAB path.
- Check capitalization to make sure it matches.
- Check that the class name is a supported System object.
- Do not include the file extension.

Error Invoking Object Method

The MATLAB System block supports only System objects written in the MATLAB language. If the software can identify an alternative block, it suggests that block in the error message, for example:



This message indicates that there is an existing dedicated and optimized block that you should use.

Performance

For fastest performance, set the block **Simulate using** parameter to `Code generation`. This setting allows the MATLAB System block to run as fast as it can. The parameter is set to this value by default.

This setting causes a slower startup time, as the software generates C code and creates a MEX-file from it. However, after code generation, later simulations have better performance. When the block uses generated code to simulate, performance is typically better than simulation without generated code.

In some cases, the implementation of your System object does not allow you to generate code, which requires you to set **Simulate using** to `Interpreted execution`. For example, your System object can require MATLAB functions beyond the subset supported for code generation. In this case, use

propagation methods to specify the block input and output port information. The MATLAB System block then propagates this signal attribution information.

See Also

More About

- “Add and Implement Propagation Methods” on page 45-27

Customize MATLAB System Block Dialog

This example shows you how to customize the block dialog for a MATLAB System block.

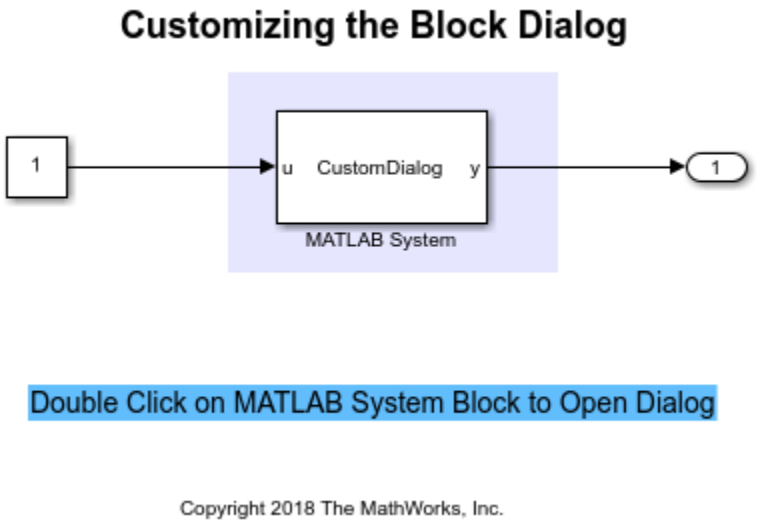
System objects

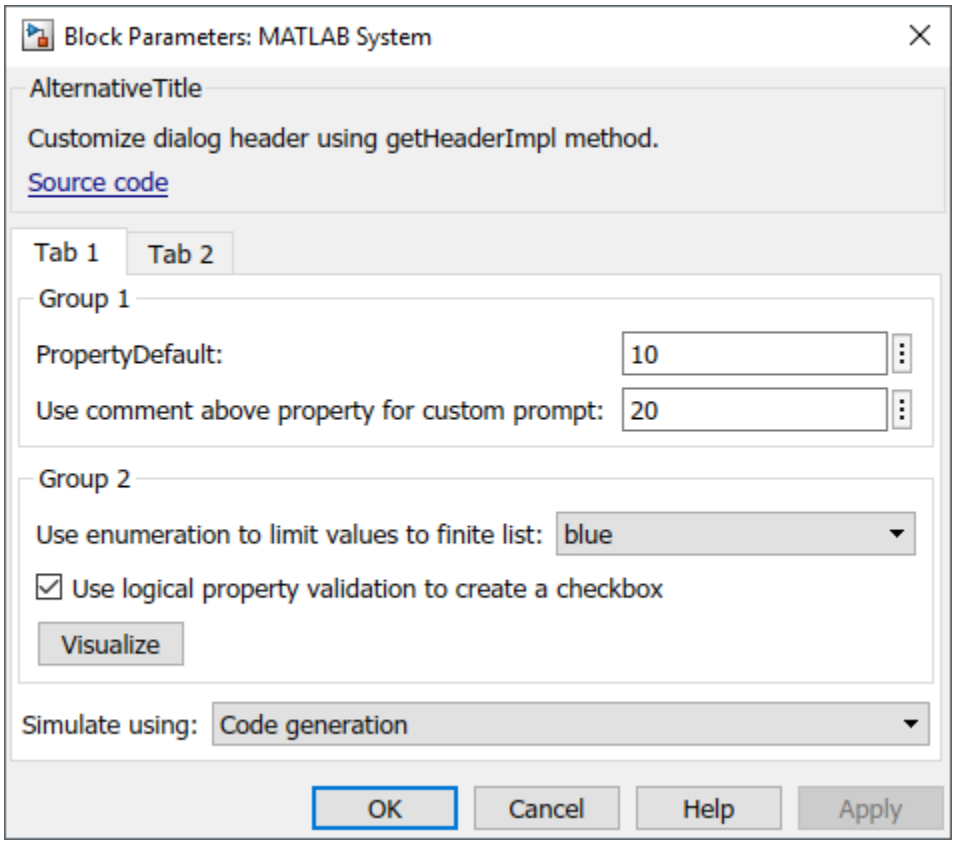
System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

This example contains a MATLAB System block that implements the System object `CustomDialog`. Use the MATLAB System block dialog box to modify properties of the System object. This example shows how to customize the prompt for a property and how to create check boxes, lists, groups, tabs, and buttons in the dialog box.





System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source Code" hyperlink from the block dialog. The System object `CustomDialog` implements the `getPropertyGroupsImpl` and `getHeaderImpl` methods that are used to customize the appearance of the block dialog and organize the System object properties.

1. `PropertyDefault` - Property with no customization
2. `PropertyCustomPrompt` - Property with custom prompt
3. `PropertyEnum` - Enumeration property with a finite list of options
4. `PropertyLogical` - Property validation with `logical` to create check box
5. `PropertyInDifferentTab` - Property shown on a different tab of the dialog box

The `getPropertyGroupsImpl` method uses the matlab.system.display.Section and matlab.system.display.SectionGroup classes to create property sections and tabs in the dialog box. `getPropertyGroupsImpl` also creates a button in the "Group 2" section which calls the `visualize` method of the System object.

```
classdef CustomDialog < matlab.System
% CustomDialog Customize System block dialog

    properties
        PropertyDefault = 10
        % For PropertyDefault, with no comment above, property name is used
```

```
    % as prompt

    % PropertyCustomPrompt Use comment above property for custom prompt
    PropertyCustomPrompt = 20

    % PropertyEnum Use enumeration to limit values to finite list
    PropertyEnum (1,1) ColorValues = ColorValues.blue

    % PropertyInDifferentTab Use getPropertyGroupsImpl to create tabs
    PropertyInDifferentTab = 30
end

properties(Nontunable)
    % PropertyLogical Use logical property validation to create a checkbox
    % Logical properties need to be Nontunable for use in Simulink
    PropertyLogical (1,1) logical = true
end

methods(Access = protected)
    function y = stepImpl(~, u)
        y = u;
    end
end

methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
    % Use getPropertyGroupsImpl to create property sections in the
    % dialog. Create two sections with titles "Group1" and
    % "Group2". "Group1" contains PropertyDefault and
    % PropertyCustomPrompt. "Group2" contains PropertyEnum,
    % PropertyLogical, and a Visualize button.
        group1 = matlab.system.display.Section(...
            'Title','Group 1',...
            'PropertyList',{'PropertyDefault','PropertyCustomPrompt'});

        group2 = matlab.system.display.Section(...
            'Title','Group 2',...
            'PropertyList',{'PropertyEnum','PropertyLogical'});

        % Add a button that calls back into the visualize method
        group2.Actions = matlab.system.display.Action(@(actionData,obj)...
                visualize(obj,actionData),'Label','Visualize');

        tab1 = matlab.system.display.SectionGroup(...
                'Title', 'Tab 1', ...
                'Sections',  [group1, group2]);

        tab2 = matlab.system.display.SectionGroup(...
                'Title', 'Tab 2', ...
                'PropertyList',  {'PropertyInDifferentTab'});

        groups = [tab1, tab2];
    end

    function header = getHeaderImpl
       header = matlab.system.display.Header(mfilename('class'), ...
               'Title','AlternativeTitle',...
               'Text','Customize dialog header using getHeaderImpl method.');
```

```
        end
    end

    methods
        function visualize(obj, actionData)
            % Use actionData to store custom data
            f = actionData.UserData;
            if isempty(f) || ~ishandle(f)
                f = figure;
                actionData.UserData = f;
            else
                figure(f); % Make figure current
            end

            d = 1:obj.PropertyCustomPrompt;
            plot(d);
        end
    end
end
```

See Also

`getPropertyGroupsImpl`

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2
- “Customize System Block Appearance” on page 45-60

Break Algebraic Loops

This example shows how to create a MATLAB System block that can break an algebraic loop in the model.

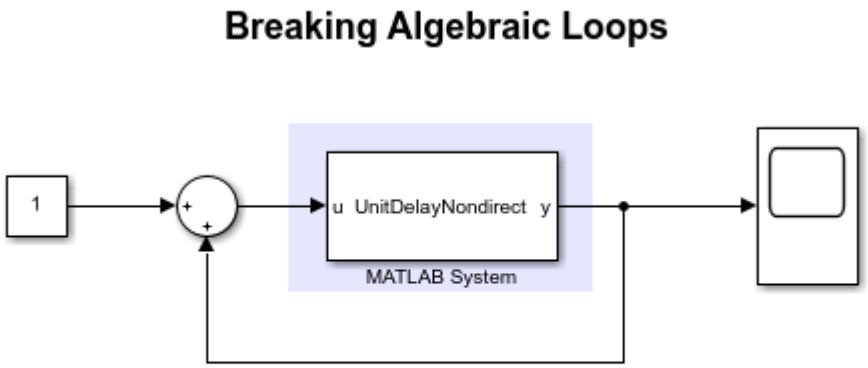
System objects

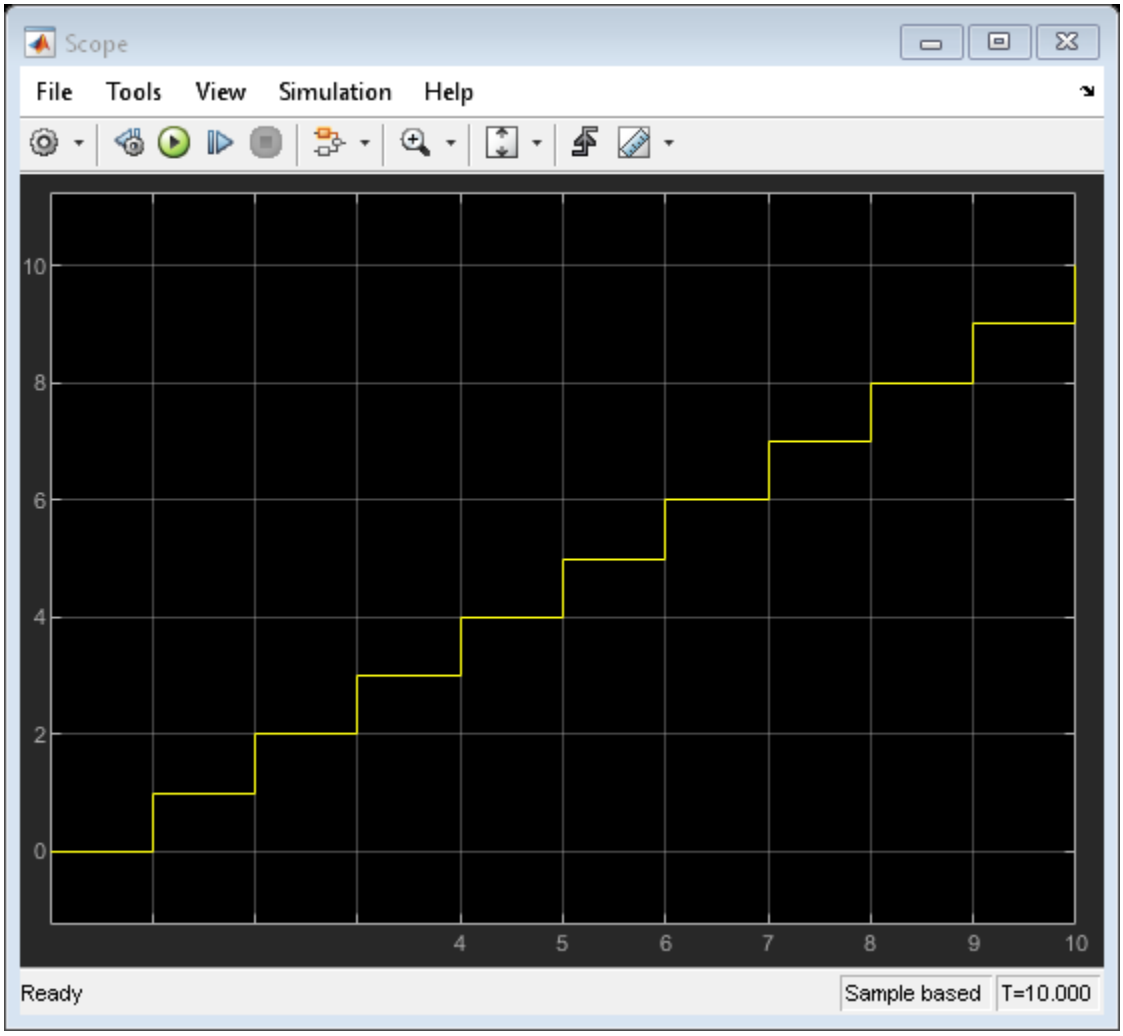
System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

This example has a MATLAB System block that is used in a feedback loop in the model. The feedback loop is used to accumulate input values and the result is displayed in the Scope block. The feedback in the model creates an algebraic loop. To solve the algebraic loop, Simulink needs a block that has nondirect feedthrough. Blocks that have nondirect feedthrough are used in the feedback loops to break algebraic loops. This block can produce an output in the current time step without receiving the input first. In this example, MATLAB System block has nondirect feedthrough.





The MATLAB System block uses the System object `UnitDelayNondirect` that implements a unit delay. The output shows how the feedback loop accumulates the input signal values.

System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source Code" hyperlink from the block dialog. The `UnitDelayNondirect` System object implements the `resetImpl`, `outputImpl`, and `updateImpl` methods. The System object has one property called `State`.

- `resetImpl` initializes the `State` property to 0.
- `outputImpl` returns the value stored in `State`. This System object has nondirect feedthrough because `outputImpl` does not use any inputs.
- `updateImpl` uses the inputs to update `State`.

```
classdef UnitDelayNondirect < matlab.System
% UnitDelayNondirect Delay input by one time step

properties(DiscreteState)
```

```
        State
    end

    methods(Access = protected)
        function resetImpl(obj)
            obj.State = 0; % Initialize states
        end
        function y = outputImpl(obj, ~)
            y = obj.State; % Output current state
            % Input is not used in this method
        end
        function updateImpl(obj,u)
            obj.State = u; % Update state with input
        end
    end
end
```

See Also

`outputImpl` | `updateImpl`

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2
- “Use System Objects in Feedback Loops” on page 45-16
- “Simulink Engine Interaction with System Object Methods” on page 45-24

Customize MATLAB System Block Appearance

This example shows how to customize the appearance of the MATLAB System block.

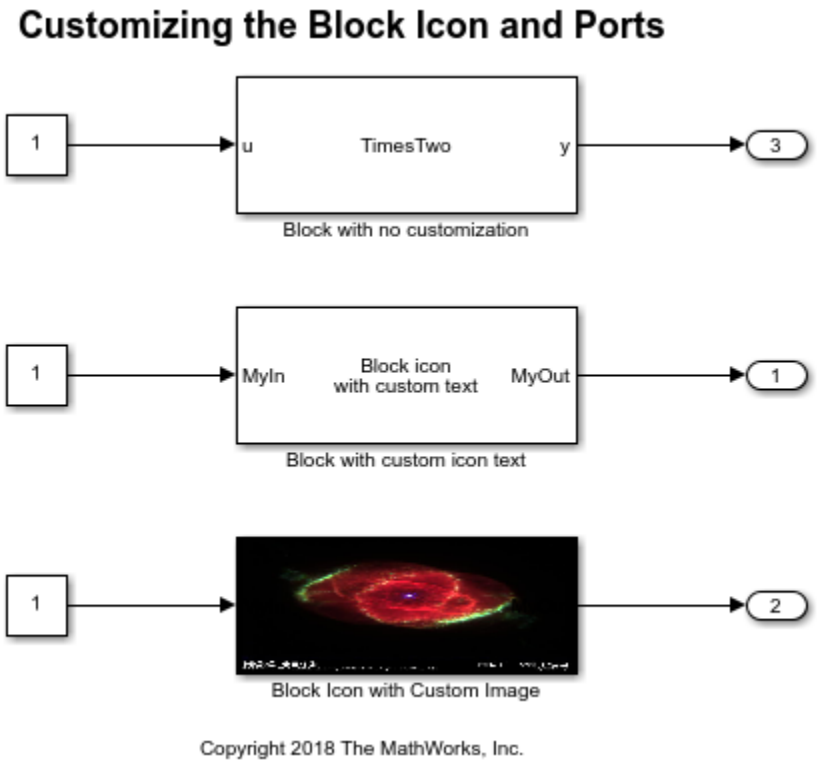
System objects

System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

There are three MATLAB System blocks in this model. The first block does not have any customization for block appearance and by default shows the name of the System object on the block. The port labels for this block are obtained from the name of the arguments in the `stepImpl` method of the System object. Second block shows custom text and custom port labels on the block icon. The third block shows a custom block icon image.



System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source Code" hyperlink from the block dialog. The `TimesTwo` System object used in the first block has no customization and implements only the `stepImpl` method. The `CustomBlockIconExample` System object implements the following methods for customizing block appearance.

- `getInputNamesImpl` - Customize input port labels
- `getOutputNamesImpl` - Customize output port labels
- `getIconImpl` - Display text or an image on the block

The System object has a `DisplayImage` property to choose between text and image for display on the block.

`TimesTwo` System object

```
classdef TimesTwo < matlab.System
%TimesTwo Multiply input by 2
%   obj = TimesTwo returns a System object, obj, that
%   multiples its input by two.

    methods(Access = protected)
        function y = stepImpl(~, u)
            y = 2 * u;
        end
    end
end
```

`CustomBlockIconExample` System object

```
classdef CustomBlockIconExample < matlab.System
% SystemObjectBlockIconExample Customize Block Icon

    properties(Nontunable)
        % DisplayImage Select to display image as block icon
        DisplayImage (1,1) logical = false
    end

    methods(Access = protected)
        function y = stepImpl(~, u)
            y = u;
        end
        function inputName = getInputNamesImpl(~)
            inputName = "MyIn";
        end
        function outputName = getOutputNamesImpl(~)
            outputName = "MyOut";
        end
        function icon = getIconImpl(obj)
            % Return text or image to be displayed on the block icon
            % Use array of strings to display multiple lines of text
            if obj.DisplayImage
                % Display image
                icon = matlab.system.display.Icon('slexngc6543aPix.jpg');
            else
                % Display text
                icon = ["Block icon", "with custom text"];
            end
        end
```

```
    end
end
```

See Also

`getIconImpl` | `matlab.system.display.Icon`

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2
- “Change Block Icon and Port Labels” on page 45-14
- “Customize System Block Appearance” on page 45-60

Implement Algorithm with Tunable Parameters

Introduction

This example shows how to implement an algorithm with tunable parameters by using a MATLAB System block.

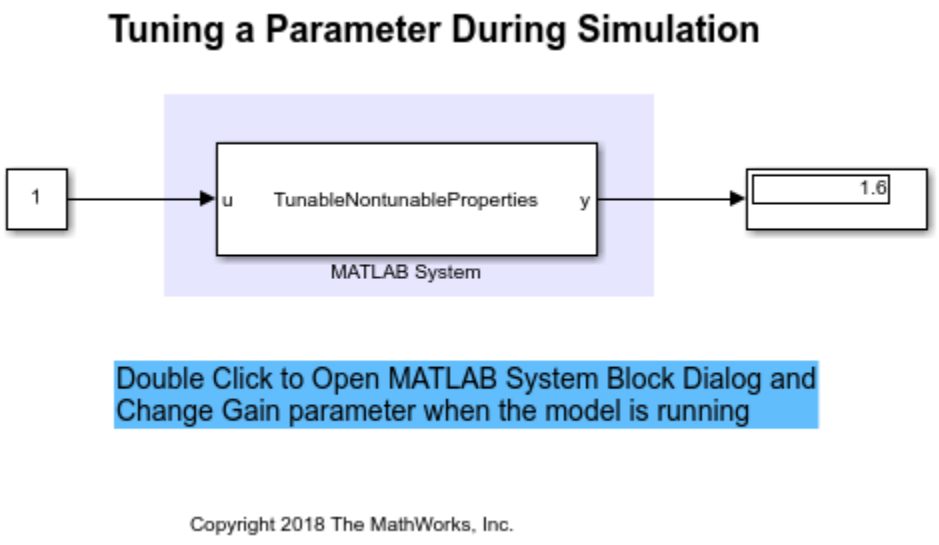
System objects

System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

The MATLAB System block implements the System object `TunableNontunableProperties` that multiples input by a `Gain` parameter and adds a `Bias` parameter to the input. The input to MATLAB System block is provided by a Constant block and the Display block shows the result of applying the gain and bias. The `Gain` parameter is tunable and can be changed when the simulation is running. The `Bias` parameter is nontunable and cannot be changed during the simulation. When the model is running, you can change tunable properties, but you cannot change nontunable properties.



System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source Code" hyperlink from the block dialog. The System object `TunableNontunableProperties` implements the `setupImpl`, `stepImpl` and `processTunedPropertiesImpl` methods. The System object has two public properties: `Gain` and `Bias`. The `Bias` property has the attribute Nontunable, which makes this property read-only during simulation. The `Gain` property has no attributes, so by default, it is tunable and can be modified during the simulation.

The System object has a private property called `pGain`. `pGain` stores the value of the public `Gain` property after the range of `Gain` is restricted between 1 and 2. You can initialize `pGain` by copying the value of `Gain` in the `setupImpl` method. In this example, `pGain` is used in the `stepImpl` method to compute the output. In a System object, whenever public tunable properties are changed, `processTunedPropertiesImpl` is called. In this example, `processTunedPropertiesImpl` updates the private property `pGain` based on the value from the public `Gain` property.

```
classdef TunableNontunableProperties < matlab.System
% TunableNontunableProperties Multiply input by Gain and add Bias

    properties
        Gain = 1.5
    end

    properties(Nontunable)
        Bias = 0.1
    end

    properties(Access = private)
        pGain
    end

    methods(Access = protected)
        function setupImpl(obj, ~)
            % Copy public property value Gain to private property pGain and
            % restrict its range between 1 and 2
            obj.pGain = obj.Gain;
            if obj.pGain < 1
                obj.pGain = 1;
            elseif obj.pGain > 2
                obj.pGain = 2;
            end
        end

        function y = stepImpl(obj, u)
            y = u * obj.pGain + obj.Bias;
        end

        function processTunedPropertiesImpl(obj)
            % Update private property pGain from the public Gain property
            % and restrict its range between 1 and 2.
            obj.pGain = obj.Gain;
            if obj.pGain < 1
                obj.pGain = 1;
            elseif obj.pGain > 2
                obj.pGain = 2;
            end
        end
```

```
    end
end
```

See Also

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2

Implement a Simple Algorithm

Introduction

This example shows how to use a simple System object in Simulink with the MATLAB System block.

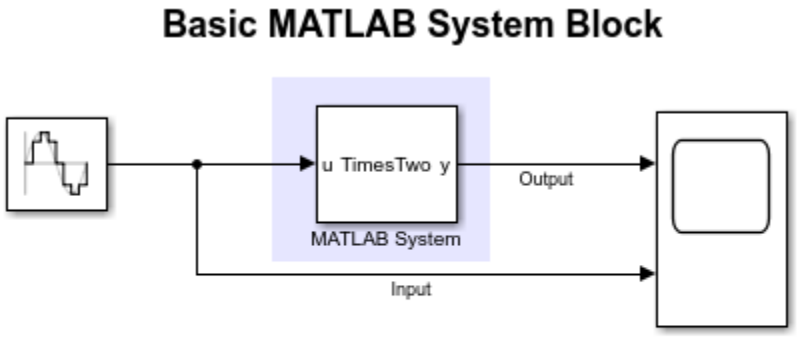
System objects

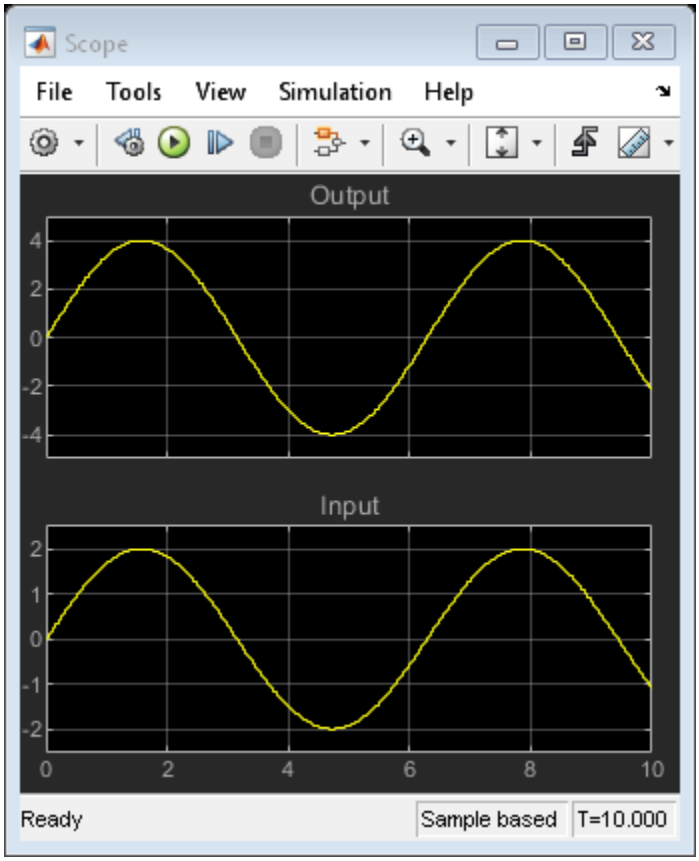
System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

This model has a MATLAB System block using the System object `TimesTwo` that multiples the input by two. The input to the MATLAB System block is provided by the Sine Wave block. The output along with the input is displayed in the Scope block. When you run the model, you can see that the input to MATLAB System block is multiplied by two in the Scope block.





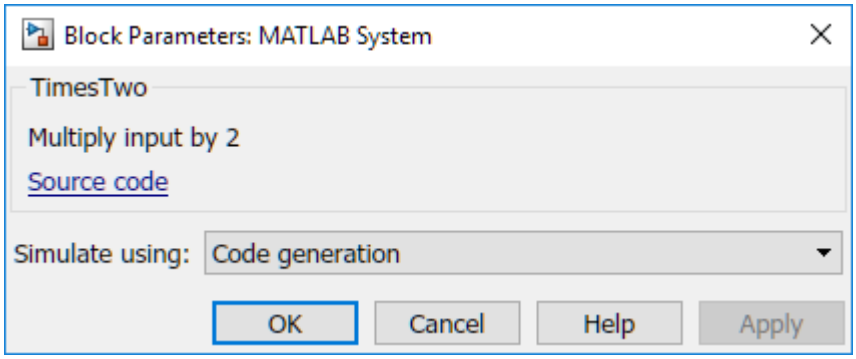
System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source Code" hyperlink from the block dialog. The System object implements only the `stepImpl` method. The algorithm does not need any properties or additional methods.

```
classdef TimesTwo < matlab.System
%TimesTwo Multiply input by 2
%   obj = TimesTwo returns a System object, obj, that
%   multiples its input by two.

    methods(Access = protected)
        function y = stepImpl(~, u)
            y = 2 * u;
        end
    end
end
```

MATLAB System Block Icon and Dialog

The MATLAB System block displays the name of the System object `TimesTwo` on the block and uses the input and output variable names from `stepImpl` method of the `TimesTwo` class as port labels. If you open the MATLAB System block dialog by double clicking on the block, the dialog shows title as `TimesTwo` and a description as "Multiply input by 2" as shown below. The title comes from the name of the System object used and the description is created from the class help summary in the System object.



See Also

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2

Specify Output Characteristics of MATLAB System Block

This example shows how to specify output size, data type and complexity of a MATLAB System block.

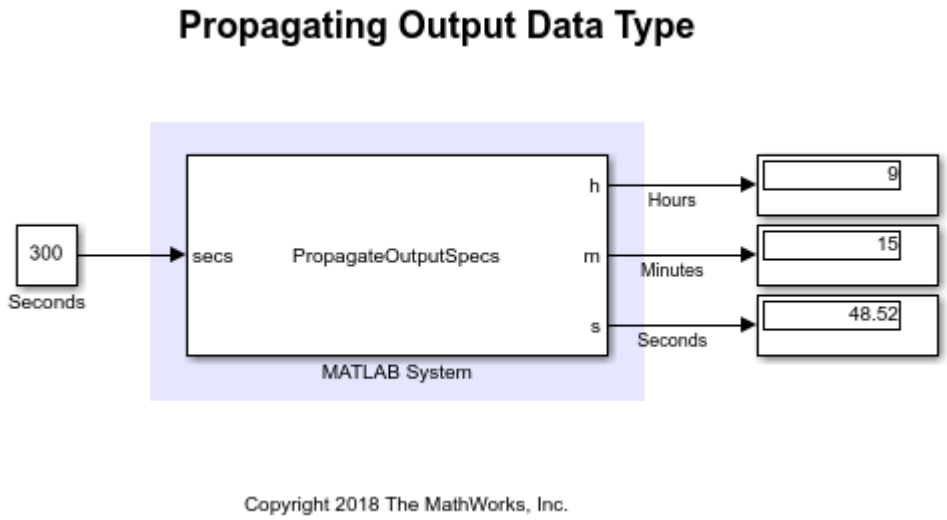
System objects

System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

This example has a MATLAB System block that adds input number of seconds to the current time and produces the resulting hours, minutes and seconds as outputs. The output values from each output port are displayed in the Display blocks.



This example, shows how to specify output size, data type and complexity of a System object. You specify the output properties when MATLAB System block cannot infer them automatically. The MATLAB System block uses the System object `PropagateOutputSpecs` that implements methods to propagate input data type, size and complexity.

System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source code" hyperlink from the block dialog. The System object `PropagateOutputSpecs` implements the `stepImpl` method that adds the input value in seconds to current time and outputs the resulting hours, minutes and seconds. The `stepImpl` method uses `datetime` function to calculate its output. Since `datetime` function is not supported for code generation, MATLAB System block cannot infer the output specifications automatically. The System object implements the following methods to specify output properties:

- `getOutputSizeImpl` - Specify output size

- `getOutputDataTypeImpl` - Specify output data type
- `isOutputComplexImpl` - Specify output complexity
- `isOutputFixedSizeImpl` - Specify whether output can be variable-size

```
classdef PropagateOutputSpecs < matlab.System
% PropagateOutputSpecs Propagation in Simulink

    methods(Access = protected)
        function [h, m, s] = stepImpl(~, secs)
        % Add input hours, minutes and seconds to current time
            d = datetime;
            d = d + seconds(secs);
            h = hour(d);
            m = minute(d);
            s = second(d);
        end

        function [o1, o2, o3] = getOutputSizeImpl(obj)
            % Return size for output port to be same as input port
            inSize = propagatedInputSize(obj, 1);
            o1 = inSize;
            o2 = inSize;
            o3 = inSize;
        end

        function [o1, o2, o3] = getOutputDataTypeImpl(obj)
            % Return data type for output port to be same as input port
            inType = propagatedInputDataType(obj, 1);
            o1 = inType;
            o2 = inType;
            o3 = inType;
        end

        function [o1, o2, o3] = isOutputComplexImpl(~)
            % Return output port complexity to be real
            o1 = false;
            o2 = false;
            o3 = false;
        end

        function [o1, o2, o3] = isOutputFixedSizeImpl(~)
            % Return true for each output port with fixed size
            o1 = true;
            o2 = true;
            o3 = true;
        end
    end
end
```

See Also

`getOutputDataTypeImpl` | `getOutputSizeImpl` | `getOutputSizeImpl` | `isOutputComplexImpl` | `isOutputFixedSizeImpl`

Related Examples

- “Specify Output” on page 45-74
- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2

Implement Algorithm that Calls External C Code

Introduction

This example shows how to use external C code in a System object.

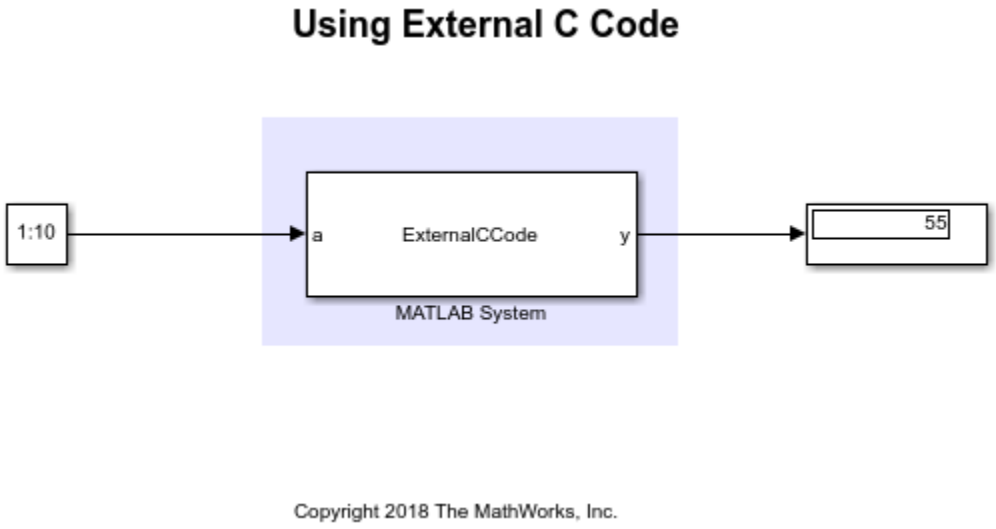
System objects

System objects allow you to implement algorithms using MATLAB. System objects are a specialized kind of MATLAB object, designed specifically for implementing and simulating dynamic systems with inputs that change over time.

After you define a System object, you can include it in a Simulink model using a MATLAB System block.

Model Description

The MATLAB System block uses the System object `ExternalCCode` that calls external C function `extSum` to compute the sum of its input elements. The Display block shows the result of the sum of values from the source block.



System Object Class Definition

You can access MATLAB source code used by the MATLAB System block by clicking the "Source code" hyperlink from the block dialog. The System object `ExternalCCode` implements the `stepImpl` method to compute its output. `stepImpl` calls external C function `extSum` to do the computation. The System object inherits from the `coder.ExternalDependency` class and implements the following methods to use external C code.

- `getDescriptiveName` - Return the name you want to associate with external dependency
- `isSupportedContext` - Return true if external dependency is supported in the current build context
- `updateBuildInfo` - Provide additional information required to link external code

The System object also calls `coder.cinclude` from `stepImpl` method to include external C header file `sum.h`.

```
classdef ExternalCCode < matlab.System & coder.ExternalDependency
% ExternalCCode Compute output by calling into external C Code

    methods(Access = protected)
        function y = stepImpl(~, a)
            % Add header file sum.h to build
            coder.cinclude('sum.h');
            y = 0.0; %#ok<NASGU> % Pre-initialize y since coder cannot
                                 % identify the type of output y from an
                                 % external C function
            % Call external C function to calculate sum of input elements
            y = coder.ceval('extSum', coder.rref(a), int32(numel(a)));
        end
    end

    methods(Static)
        function bName = getDescriptiveName(~)
        % Return a descriptive name for the external dependency. Code
        % generator uses this name for error messages.
            bName = 'SumAPI';
        end

        function tf = isSupportedContext(~)
        % Use this function to determine whether current build context
        % supports external dependency. Build context includes information
        % about target language and code generation target.
            tf = true;
        end

        function updateBuildInfo(buildInfo, ~)
            % Add source file sum.c to build
            buildInfo.addSourceFiles('sum.c');
        end
    end
end
```

External C code

External C function `extSum` is defined in sum.c file.

```
#include "sum.h"

double extSum(const double *a, int numElems)
{
    int ii;
    double sum = 0.0;
    for (ii=0; ii < numElems; ii++) {
        sum += a[ii];
    }
    return sum;
}
```

`extSum` is declared in sum.h.

```
#ifndef SUM_H
#define SUM_H

extern double extSum(const double *a, int numElems);

#endif
```

See Also

`coder.ExternalDependency.updateBuildInfo` | `coder.ceval` | `coder.cinclude`

Related Examples

- “What Are System Objects?”
- “Why Use the MATLAB System Block?” on page 45-2

Customize System Block Appearance

In this section...

Specify Input and Output Names

Specify the names of the input and output ports of a System object–based block implemented using a MATLAB System block.

Use `getInputNamesImpl` and `getOutputNamesImpl` to specify the names of the input port as “source data” and the output port as “count.”

If you do not specify the `getInputNamesImpl` and `getOutputNamesImpl` methods, the object uses the `stepImpl` method input and output variable names for the input and output port names, respectively. If the `stepImpl` method uses *varargin* and *varargout* instead of variable names, the port names default to empty character vectors.

```
methods (Access = protected)
   function inputName = getInputNamesImpl(~)
          inputName = 'source data';
   end

   function outputName = getOutputNamesImpl(~)
          outputName = 'count';
   end
end
```

Complete Class Definition with Named Inputs and Outputs

```
classdef MyCounter < matlab.System

   % MyCounter Count values above a threshold

    properties
       Threshold = 1
    end
    properties (DiscreteState)
       Count
    end

    methods
       function obj = MyCounter(varargin)
          setProperties (obj,nargin,varargin{:});
       end
    end

    methods (Access = protected)
       function setupImpl(obj)
          obj.Count = 0;
       end
       function resetImpl(obj)
```

```
            obj.Count = 0;
        end
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
               obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
        function inputName = getInputNamesImpl(~)
            inputName = 'source data';
        end
        function outputName = getOutputNamesImpl(~)
            outputName = 'count';
        end
    end
end
```

Add Text to Block Icon

Add text to the block icon of a System object–based block implemented using a MATLAB System block.

1. Subclass from custom icon class.

   ```
   classdef MyCounter < matlab.System & matlab.system.mixin.CustomIcon
   ```

2. Use `getIconImpl` to specify the block icon as `New Counter` with a line break between the two words.

   ```
   methods (Access = protected)
       function icon = getIconImpl(~)
           icon = {'New','Counter'};
       end
   end
   ```

   Complete Class Definition File with Defined Icon

   ```
   classdef MyCounter < matlab.System & ...
      matlab.system.mixin.CustomIcon

      % MyCounter Count values above a threshold

       properties
          Threshold = 1
       end
       properties (DiscreteState)
          Count
       end

       methods
          function obj = MyCounter(varargin)
             setProperties(obj,nargin,varargin{:});
          end
       end

       methods (Access = protected)
          function setupImpl(obj)
             obj.Count = 0;
   ```

```
        end
        function resetImpl(obj)
           obj.Count = 0;
        end
        function y = stepImpl(obj,u)
           if (u > obj.Threshold)
              obj.Count = obj.Count + 1;
           end
           y = obj.Count;
        end
        function icon = getIconImpl(~)
           icon = {'New','Counter'};
        end
    end
end
```

Add Image to Block Icon

Define an image on the block icon of a System object–based block implemented using a MATLAB System block.

1. Subclass from custom icon class.

   ```
   classdef MyCounter < matlab.System & matlab.system.mixin.CustomIcon
   ```

2. Use `getIconImpl` method to call the `matlab.system.display.Icon` class and specify the image.

   ```
   methods (Access = protected)
       function icon = getIconImpl(~)
           icon = matlab.system.display.Icon('counter.png');
       end
   end
   ```

   Complete Class Definition File with Icon Image

   ```
   classdef MyCounter < matlab.System & ...
      matlab.system.mixin.CustomIcon

      % MyCounter Count values above a threshold

       properties
          Threshold = 1
       end
       properties (DiscreteState)
          Count
       end

       methods
          function obj = MyCounter(varargin)
             setProperties(obj,nargin,varargin{:});
          end
       end

       methods (Access = protected)
          function setupImpl(obj)
             obj.Count = 0;
          end
   ```

```
        function resetImpl(obj)
           obj.Count = 0;
        end
        function y = stepImpl(obj,u)
           if (u > obj.Threshold)
              obj.Count = obj.Count + 1;
           end
           y = obj.Count;
        end
        function icon = getIconImpl(~)
           icon = matlab.system.display.Icon('counter.png');
        end
     end
end
```

See Also

`getIconImpl` | `getInputNamesImpl` | `getNumInputsImpl` | `getNumOutputsImpl` | `getOutputNamesImpl` | `matlab.system.mixin.CustomIcon`

Related Examples

- “Change the Number of Inputs”
- “Using ~ as an Input Argument in Method Definitions”
- “Subclassing Multiple Classes”

Customize System Block Dialog Box

In this section...

You can customize the dialog box for the MATLAB System block by adding properties and methods in the corresponding System object. You can add tabs, organized properties into groups and sections, add block descriptions, simulation type control, and add custom buttons.

Define Block Dialog Tabs, Sections, and Order of Properties

This example customizes the block dialog box for the `MultipleGroupsWithTabs` MATLAB System block by specifying property display names and modifying the `getPropertyGroupImpl` method.

Change Property Label

To change the property label that appears on the dialog box, add comments before each property in this format `%PropertyName Block Dialog Label` with no space between the comment and the property name. For example, to display the `StartValue` property as **Start Value**, specify:

```
%StartValue Start Value
StartValue = 0
```

The `MultipleGroupsWithTabs` System object in this example relabels each property for display in the MATLAB System block dialog.

Organize Dialog Box

The `MutlitpleGroupsWithTabs` System object class defines a `getPropertyGroupsImpl` method. Inside the `getPropertyGroupsImpl` method, this example defines two tabs (section groups) and three parameter groupings (sections).

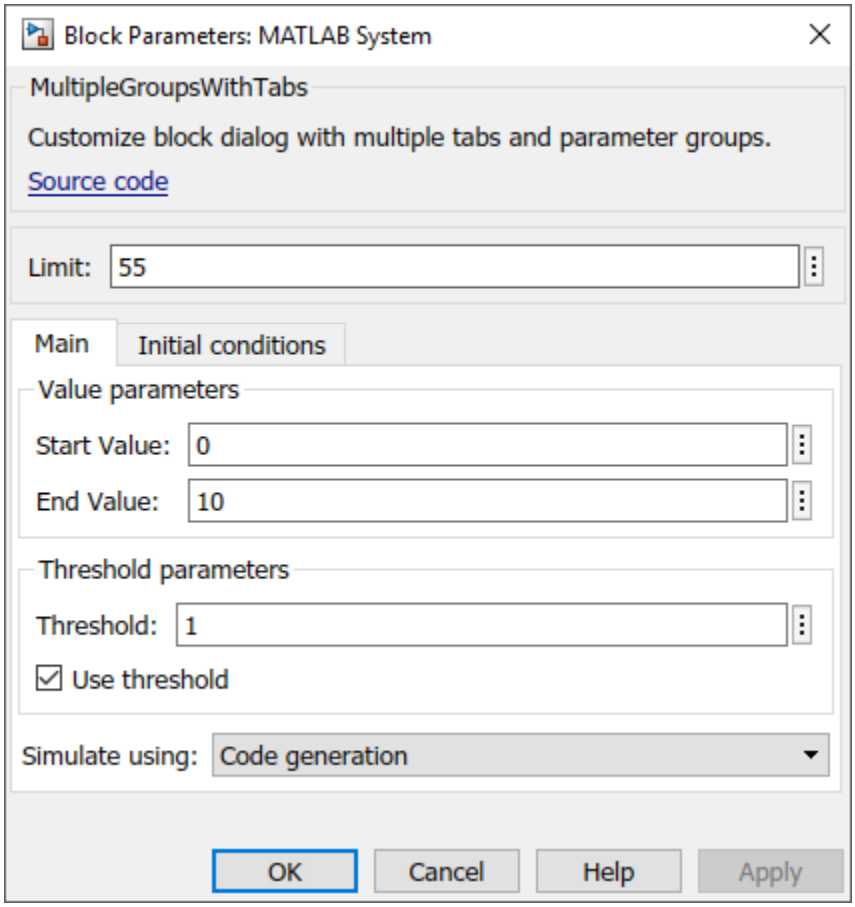
```
classdef MultipleGroupsWithTabs < matlab.System
    % MultipleGroupsWithTabs Customize block dialog with multiple tabs and parameter groups.

    % Public, tunable properties
    properties
        %StartValue Start Value
        StartValue = 0

        %EndValue End Value
        EndValue = 10

        Threshold = 1

        %BlockLimit Limit
        BlockLimit = 55
    end
```

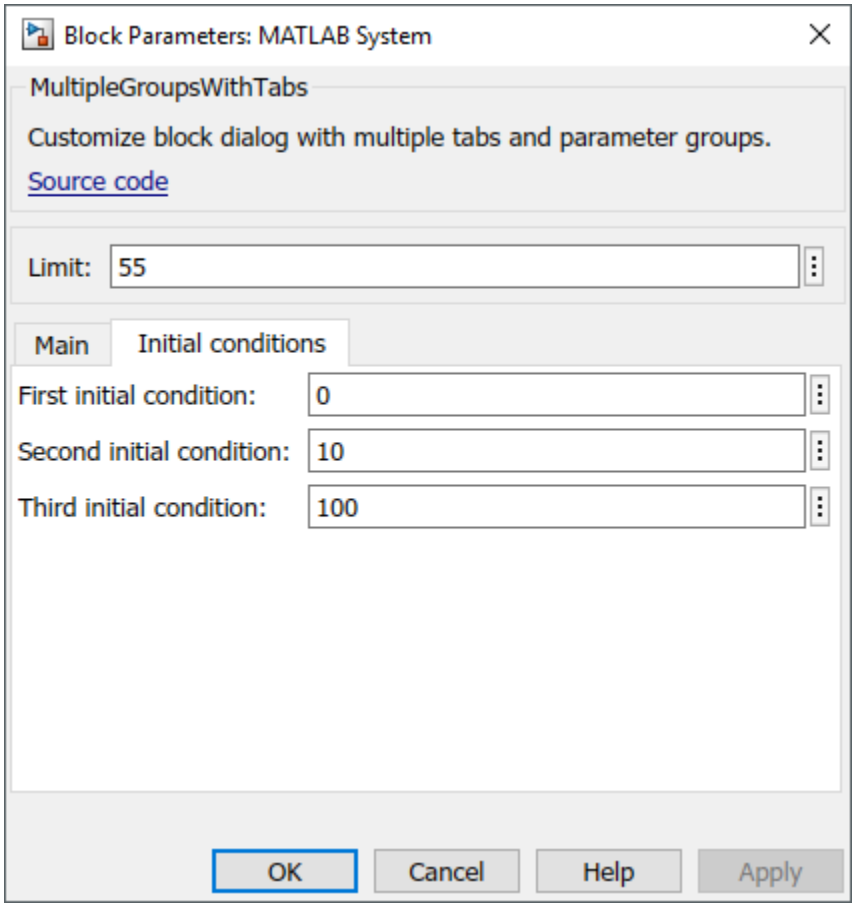
```
    % Public Nontunable
    properties(Nontunable)
        %IC1 First initial condition
        IC1 = 0

        %IC2 Second initial condition
        IC2 = 10

        %IC3 Third initial condition
        IC3 = 100

        %UseThreshold Use threshold
        UseThreshold (1,1) logical = true
    end

    methods (Static, Access = protected)
        function groups = getPropertyGroupsImpl
            % Section to always display above any tabs.
            alwaysSection = matlab.system.display.Section(...
                'Title','','PropertyList',{'BlockLimit'});

            % Group with no sections
            initTab = matlab.system.display.SectionGroup(...
                'Title','Initial conditions', ...
                'PropertyList',{'IC1','IC2','IC3'});

            % Section for the value parameters
            valueSection = matlab.system.display.Section(...
                'Title','Value parameters',...
                'PropertyList',{'StartValue','EndValue'});

            % Section for the threshold parameters
            thresholdSection = matlab.system.display.Section(...
                'Title','Threshold parameters',...
                'PropertyList',{'Threshold','UseThreshold'});

            % Group with two sections: the valueSection and thresholdSection sections
            mainTab = matlab.system.display.SectionGroup(...
                'Title','Main', ...
                'Sections',[valueSection,thresholdSection]);

            % Return an array with the group-less section, the group with
            % two sections, and the group with no sections.
            groups = [alwaysSection,mainTab,initTab];
        end
    end
end
```

Resulting Dialog Box

```
load_system('ShowSystemBlockDialog')
open_system('ShowSystemBlockDialog/MATLAB System')
```


1
MultipleGroupsWithTabs
1


Block Parameters: MATLAB System
MultipleGroupsWithTabs
Customize block dialog with multiple tabs and parameter groups.
Source code
Limit:
55
Main
Initial conditions
Value parameters
Start Value:
0
End Value:
10
Threshold parameters
Threshold:
1
Use threshold
Simulate using:
Code generation
OK
Cancel
Help
Apply



Define Property Sections

This example customizes the block dialog box for a MATLAB System block by specifying property display names and modifying the `getPropertyGroupImpl` method. This customization is demonstrated with the System object `AddPropertySections`.

Change Property Labels

To change the property label that appears on the dialog box, add comments before each property in this format `%PropertyName Block Dialog Label` with no space between the percent sign and the property name. For example, to display the `UseAlpha` property as **Use alpha**, specify:

```
%UseAlpha Use alpha
UseAlpha = 0
```

The `AddPropertySections` System object included with this example relabels properties for display in the MATLAB System block dialog.

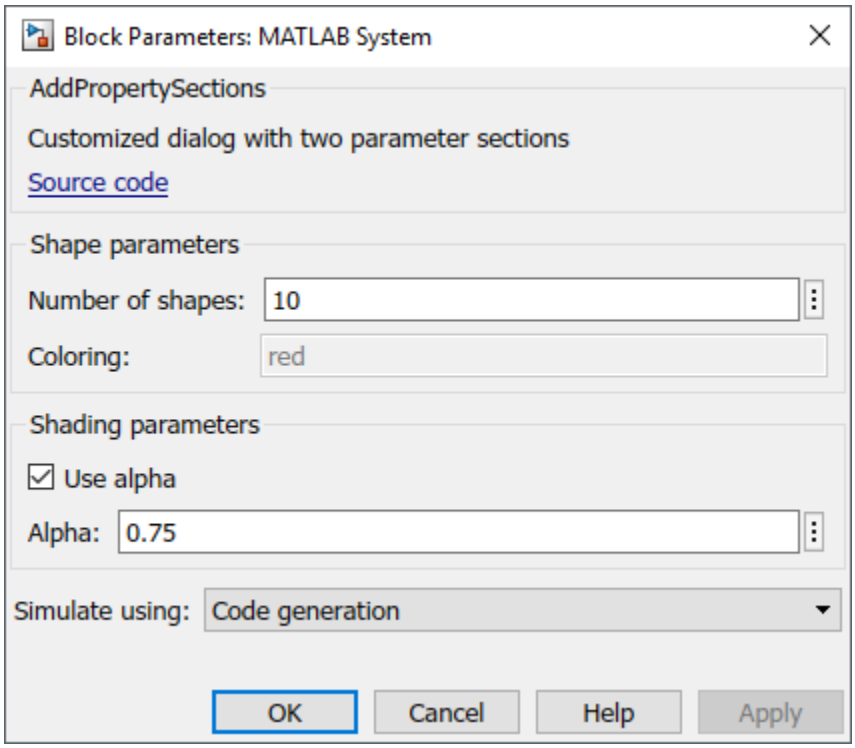
Organize Dialog Box

To organize the properties on the dialog box, the `AddPropertySections` System object class defines a `getPropertyGroupsImpl` method. Inside the `getPropertyGroupsImpl` method, this example defines two sections, each with two properties.

```
classdef AddPropertySections < matlab.System
    % AddPropertySections Customized dialog with two parameter sections

    % Public, tunable properties
    properties

        %NumberOfShapes Number of shapes
        NumberOfShapes = 10

        Alpha = 0.75
    end

    % Public, nontunable properties
    properties(Nontunable)
        Coloring (1, 1) {mustBeMember(Coloring,["red","blue","green"])} = "red"

        %UseAlpha Use alpha
        UseAlpha (1,1) logical = false
    end

    methods (Static, Access = protected)
        function groups = getPropertyGroupsImpl
            % Section for the value parameters
            valueSection = matlab.system.display.Section(...
                'Title','Shape parameters',...
                'PropertyList',{'NumberOfShapes','Coloring'});

            % Section for the threshold parameters
            shadingSection = matlab.system.display.Section(...
                'Title','Shading parameters',...
                'PropertyList',{'UseAlpha','Alpha'});

            % Return an array with the two sections.
            groups = [valueSection, shadingSection];
        end
    end
end
```

Resulting Dialog Box

```
load_system('CustomSystemBlockDialog')
open_system('CustomSystemBlockDialog/MATLAB System')
```



Add Header Description

Add a header panel to a MATLAB System block by adding the `getHeaderImpl` method to your System object.

Use `getHeaderImpl` to specify a panel title and text for the `MyCounter` System object. If you do not specify the `getHeaderImpl`, the block does not display any title or text for the panel.

As for all `Impl` methods, set the `getHeaderImpl` method access to `protected` because the method is only called internally.

```
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header('MyCounter',...
        'Title','My Enhanced Counter');
   end
end
```

Complete Class Definition

```
 classdef MyCounter < matlab.System

   % MyCounter Count values

    properties
       Threshold = 1
    end
    properties (DiscreteState)
       Count
    end
```

```
    methods (Static, Access = protected)
      function header = getHeaderImpl
         header = matlab.system.display.Header('MyCounter',...
           'Title','My Enhanced Counter',...
           'Text', 'This counter is an enhanced version.');
      end
    end

    methods (Access = protected)
       function setupImpl(obj,u)
          obj.Count = 0;
       end
       function y = stepImpl(obj,u)
          if (u > obj.Threshold)
             obj.Count = obj.Count + 1;
          end
          y = obj.Count;
       end
       function resetImpl(obj)
          obj.Count = 0;
       end
    end
end
```

Control Simulation Type in MATLAB System Block

Specify a simulation type and whether the **Simulate using** parameter appears on the Simulink MATLAB System block dialog box. The simulation options are `'Code generation'` and `'Interpreted mode'`.

If you do not include the `getSimulateUsingImpl` method in your class definition file, the System object allows both simulation modes and defaults to `'Code generation'`. If you do not include the `showSimulateUsingImpl` method, the **Simulate using** parameter appears on the block dialog box.

You must set the `getSimulateUsingImpl` and `showSimulateUsingImpl` methods to `static` and the access for these methods to `protected`.

Use `getSimulateUsingImpl` to specify that only interpreted execution is allowed for the System object.

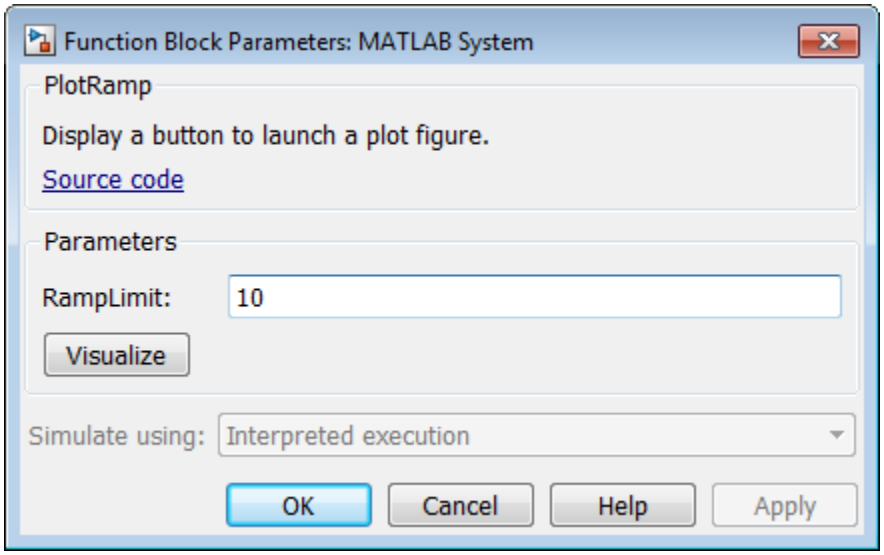
```
methods(Static,Access = protected)
   function simMode = getSimulateUsingImpl
       simMode = 'Interpreted execution';
    end
end
```

Complete Class Definition

```
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  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

    function simMode = getSimulateUsingImpl
      simMode = 'Interpreted execution';
    end
  end

  methods
    function obj = ActionDemo(varargin)
      setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj)
      figure;
      d = 1:obj.RampLimit;
      plot(d);
    end
   methods(Static,Access = protected)
   end
  end
end
```

The resulting dialog box with the **Simulate using** parameter:



Add Custom Button to MATLAB System Block

Add a button to the MATLAB System block dialog box. This button opens a figure that plots a ramp function.

Use `matlab.system.display.Action` to define the MATLAB function or code associated with a button in the MATLAB System block dialog box. The example also shows how to set button options and use an `actionData` object input to store a figure handle. This part of the code example uses the

same figure when the button is clicked multiple times, rather than opening a new figure for each button click.

```
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
```

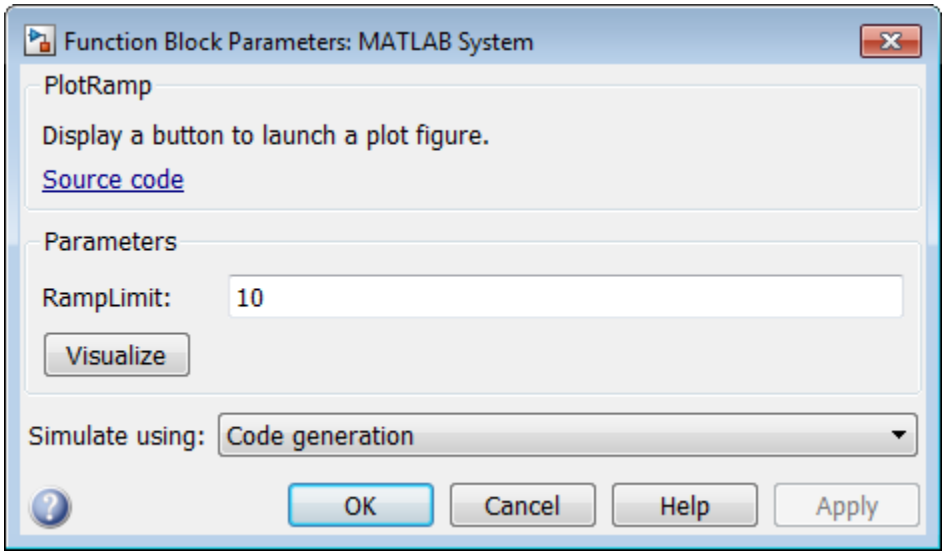
Complete Class Definition File for Dialog Button

Define a property group and a second tab in the class definition file.

```
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  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
end
```

The resulting dialog box with the **Visualize** button:



See Also

`getHeaderImpl` | `getPropertyGroupsImpl` | `getSimulateUsingImpl` | `matlab.system.display.Header` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup` | `showSimulateUsingImpl`

More About

- "Using ~ as an Input Argument in Method Definitions"

Specify Output

In this section...

Sometimes, Simulink cannot infer the output characteristics of your System object during model compilation. To give Simulink more information about the System object output, use these methods.

Set Output Size

Specify the size of a System object output using the `getOutputSizeImpl` method. Use this method when Simulink cannot infer the output size from the inputs during model compilation. For instance, when the System object has multiple inputs or outputs or has variable-sized output.

For variable-size inputs, the propagated input size from `propagatedInputSizeImpl` differs depending on the environment.

- MATLAB — When you first run 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.

Subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
 classdef CounterReset < matlab.System & ...
     matlab.system.mixin.Propagates
```

Use the `getOutputSizeImpl` method to specify the output size.

```
methods (Access = protected)
   function sizeout = getOutputSizeImpl(~)
      sizeout = [1 1];
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
```

```
            obj.Count = 0;
        end

        function y = stepImpl(obj,u1,u2)
            % Add to count if u1 is above threshold
            % Reset if u2 is true
            if (u2)
              obj.Count = 0;
            elseif (any(u1 > obj.Threshold))
              obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function resetImpl(obj)
            obj.Count = 0;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            if strcmp(name,'Count')
                sz = [1 1];
                dt = 'double';
                cp = false;
            else
                error(['Error: Incorrect State Name: ', name.']);
            end
        end
        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end
        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end
        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
        function flag = isInputSizeMutableImpl(~,idx)
            if idx == 1
              flag = true;
            else
              flag = false;
            end
        end
    end
end
```

Set Fixed- or Variable-Size Output

Specify the System object output is fixed-size. Fixed-size output is always the same size, while variable-size output can be different size vectors.

Simulink cannot infer the output size for variable-size output. To avoid errors, implement `isOutputFixedSizeImpl` and `getOutputSizeImpl`.

`isOutputFixedSizeImpl` accepts System object handle and returns an array of flags. Array size is equal to the size of the output ports. The value of the flags and their meanings are:

- `true` — the output size is fixed (output port on MATLAB System block creates variable-size signal)
- `false` — the output size is variable (output port on MATLAB System block creates fixed-size signal)

Subclass from both the `matlab.System` base class and the `matlab.system.mixin.Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `isOutputFixedSizeImpl` method to specify that the output is fixed size.

```
methods (Access = protected)
    function fixedout = isOutputFixedSizeImpl(~)
        fixedout = true;
    end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function y = stepImpl(obj,u1,u2)
         % Add to count if u1 is above threshold
         % Reset if u2 is true
         if (u2)
           obj.Count = 0;
         elseif (any(u1 > obj.Threshold))
           obj.Count = obj.Count + 1;
         end
         y = obj.Count;
      end

      function resetImpl(obj)
         obj.Count = 0;
      end

      function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
         if strcmp(name,'Count')
            sz = [1 1];
            dt = 'double';
```

```
            cp = false;
         else
            error(['Error: Incorrect State Name: ', name.']);
         end
      end
      function dataout = getOutputDataTypeImpl(~)
         dataout = 'double';
      end
      function sizeout = getOutputSizeImpl(~)
         sizeout = [1 1];
      end
      function cplxout = isOutputComplexImpl(~)
         cplxout = false;
      end
      function fixedout = isOutputFixedSizeImpl(~)
         fixedout = true;
      end
      function flag = isInputSizeMutableImpl(~,idx)
         if idx == 1
           flag = true;
         else
           flag = false;
         end
      end
   end
end
```

Set Output Data Type

Specify the data type of a System object output using the `getOutputDataTypeImpl` method. A second example shows how to specify a gain object with bus output. Use this method when Simulink cannot infer the data type from the inputs during model compilation or when you want different input and output data types. If you want bus output, also use the `getOutputDataTypeImpl` method. To use bus output, you must define the bus data type in the base workspace and you must include the `getOutputDataTypeImpl` method in your class definition file.

For both examples, subclass from both the `matlab.System` base class and the `matlab.system.mixin.Propagates` mixin class.

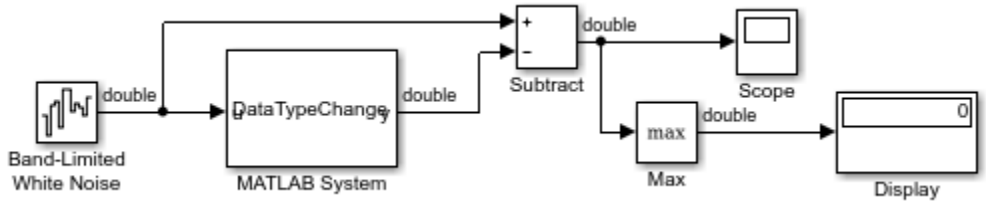
```
  classdef DataTypeChange < matlab.System & ...
    matlab.system.mixin.Propagates
```

Specify, in your class definition file, how to control the output data type from a MATLAB System block. Use the `getOutputDataTypeImpl` method to change the output data type from double to single, or propagate the input as a double. It also shows how to cast the data type to change the output data type in the `stepImpl` method, if necessary.

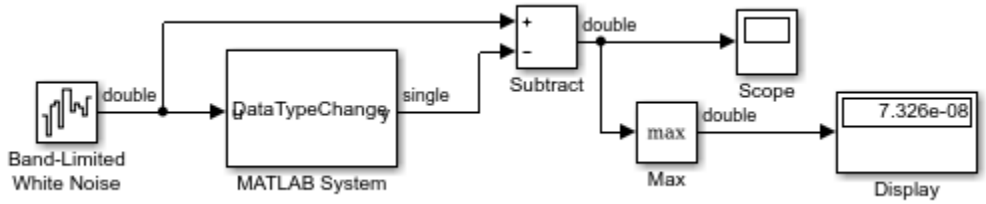
```
methods (Access = protected)
   function out = getOutputDataTypeImpl(obj)
      if obj.Quantize == true
         out = 'single';
      else
         out = propagatedInputDataType(obj,1);
      end
   end
end
```

```
classdef DataTypeChange < matlab.System & ...
  matlab.system.mixin.Propagates

   properties(Nontunable)
      Quantize = false;
   end

   methods(Access = protected)
      function y = stepImpl(obj,u)
         if obj.Quantize == true
            % Cast for output data type to differ from input.
            y = single(u);
         else
            % Propagate output data type.
            y = u;
         end
      end

      function out = getOutputDataTypeImpl(obj)
         if obj.Quantize == true
            out = 'single';
         else
            out = propagatedInputDataType(obj,1);
         end
      end
   end
end
```

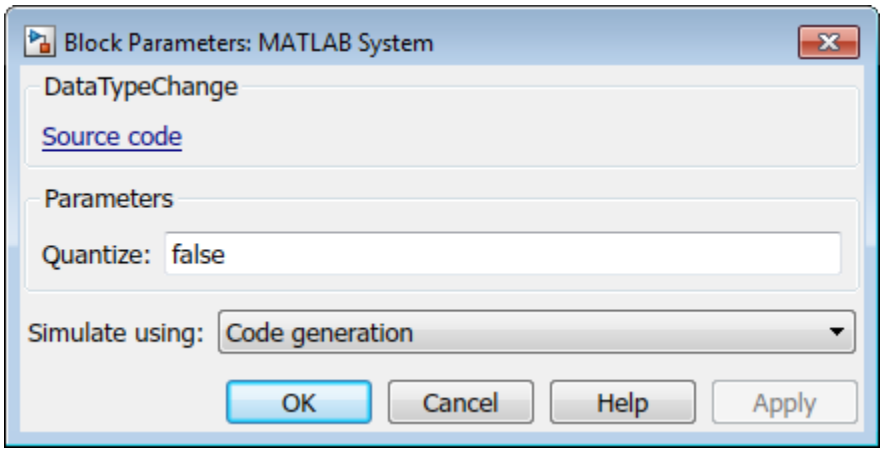
This model shows propagated double data type.



This model shows the result of changing the data type from double to single. The Display block shows the effect of quantizing the data.



The block mask for the MATLAB System block includes an edit field to switch between using propagation (**Quantize** = `false`) and switching from double to single (**Quantize** = `true`).



Use the `getOutputDataTypeImpl` method to specify the output data type as a bus. Specify the bus name in a property.

```
properties(Nontunable)
   OutputBusName = 'bus_name';
end

methods (Access = protected)
   function out = getOutputDataTypeImpl(obj)
      out = obj.OutputBusName;
   end
end
```

View the method in the complete class definition file. This class definition file also includes code to implement a custom icon for this object in the MATLAB System block

```
classdef busGain < matlab.System & matlab.system.mixin.Propagates
% busGain Apply a gain of two to bus input.

   properties
      GainK = 2;
   end

   properties(Nontunable)
      OutputBusName = 'bus_name';
   end

   methods (Access=protected)
      function out = stepImpl(obj,in)
         out.a = obj.GainK * in.a;
         out.b = obj.GainK * in.b;
      end

      function out = getOutputSizeImpl(obj)
         out = propagatedInputSize(obj, 1);
      end

      function out = isOutputComplexImpl(obj)
         out = propagatedInputComplexity(obj, 1);
      end
```

```
      function out = getOutputDataTypeImpl(obj)
         out = obj.OutputBusName;
      end

      function out = isOutputFixedSizeImpl(obj)
         out = propagatedInputFixedSize(obj,1);
      end
   end
end
```

Set Output Complexity

Specify whether a System object output is complex or real using the `isOutputComplexImpl` method. Use this method when Simulink cannot infer the output complexity from the inputs during model compilation.

Subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
 classdef CounterReset < matlab.System & ...
     matlab.system.mixin.Propagates
```

Use the `isOutputComplexImpl` method to specify that the output is real.

```
methods (Access = protected)
   function cplxout = isOutputComplexImpl(~)
      cplxout = false;
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function y = stepImpl(obj,u1,u2)
         % Add to count if u1 is above threshold
         % Reset if u2 is true
         if (u2)
           obj.Count = 0;
         elseif (any(u1 > obj.Threshold))
           obj.Count = obj.Count + 1;
         end
         y = obj.Count;
      end
```

```
      function resetImpl(obj)
         obj.Count = 0;
      end

      function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
         if strcmp(name,'Count')
            sz = [1 1];
            dt = 'double';
            cp = false;
         else
            error(['Error: Incorrect State Name: ', name.']);
         end
      end
      function dataout = getOutputDataTypeImpl(~)
         dataout = 'double';
      end
      function sizeout = getOutputSizeImpl(~)
         sizeout = [1 1];
      end
      function cplxout = isOutputComplexImpl(~)
         cplxout = false;
      end
      function fixedout = isOutputFixedSizeImpl(~)
         fixedout = true;
      end
      function flag = isInputSizeMutableImpl(~,idx)
         if idx == 1
           flag = true;
         else
           flag = false;
         end
      end
   end
end
```

Set Discrete State Output Specification

Specify the size, data type, and complexity of a discrete state property using the `getDiscreteStateSpecificationImpl` method. Use this method when your System object has a property with the `DiscreteState` attribute and Simulink cannot infer the output specifications during model compilation.

Subclass from both the `matlab.System` base class and from the `Propagates` mixin class.

```
 classdef CounterReset < matlab.System & ...
     matlab.system.mixin.Propagates
```

Use the `getDiscreteStateSpecificationImpl` method to specify the size and data type. Also specify the complexity of a discrete state property `Count`, which is used in the counter reset example.

```
function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
    if strcmp(name,'Count')
        sz = [1 1];
        dt = 'double';
        cp = false;
    else
```

```
        error(['Error: Incorrect State Name: ', name.']);
    end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function y = stepImpl(obj,u1,u2)
         % Add to count if u1 is above threshold
         % Reset if u2 is true
         if (u2)
           obj.Count = 0;
         elseif (any(u1 > obj.Threshold))
           obj.Count = obj.Count + 1;
         end
         y = obj.Count;
      end

      function resetImpl(obj)
         obj.Count = 0;
      end

      function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
         if strcmp(name,'Count')
            sz = [1 1];
            dt = 'double';
            cp = false;
         else
            error(['Error: Incorrect State Name: ', name.']);
         end
      end
      function dataout = getOutputDataTypeImpl(~)
         dataout = 'double';
      end
      function sizeout = getOutputSizeImpl(~)
         sizeout = [1 1];
      end
      function cplxout = isOutputComplexImpl(~)
         cplxout = false;
      end
      function fixedout = isOutputFixedSizeImpl(~)
         fixedout = true;
      end
```

```
      function flag = isInputSizeMutableImpl(~,idx)
         if idx == 1
           flag = true;
         else
           flag = false;
         end
      end
   end
end
```

See Also

`getDiscreteStateSpecificationImpl` | `getOutputDataTypeImpl` | `getOutputSizeImpl` | `isOutputComplexImpl` | `isOutputFixedSizeImpl` | `matlab.system.mixin.Propagates`

More About

- “Subclassing Multiple Classes”
- “Using ~ as an Input Argument in Method Definitions”

Set Model Reference Discrete Sample Time Inheritance

Disallow model reference discrete sample time inheritance for a System object. The System object defined in this example has one input, so by default, it allows sample time inheritance. To override the default and disallow inheritance, the class definition file for this example includes the `allowModelReferenceDiscreteSampleTimeInheritanceImpl` method, with its output set to `false`.

```
methods (Access = protected)
   function flag = ...
       allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
     flag = false;
   end
end
```

View the method in the complete class definition file.

```
classdef MyCounter < matlab.System

   % MyCounter Count values

    properties
       Threshold = 1;
    end

    properties (DiscreteState)
       Count
    end

    methods (Static, Access = protected)
      function header = getHeaderImpl
         header = matlab.system.display.Header('MyCounter',...
           'Title','My Enhanced Counter',...
           'Text', 'This counter is an enhanced version.');
      end
    end

    methods (Access = protected)
       function flag = ...
            allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
          flag = false
       end
       function setupImpl(obj,u)
          obj.Count = 0;
       end
       function y = stepImpl(obj,u)
          if (u > obj.Threshold)
             obj.Count = obj.Count + 1;
          end
          y = obj.Count;
       end
       function resetImpl(obj)
          obj.Count = 0;
       end
```

```
    end
end
```

See Also

`allowModelReferenceDiscreteSampleTimeInheritanceImpl` | `matlab.System`

Use Update and Output for Nondirect Feedthrough

Implement nondirect feedthrough for a System object by using the `updateImpl`, `outputImpl`, and `isInputDirectFeedthroughImpl` methods. In nondirect feedthrough, the outputs depend only on the internal states and properties of the object, rather than the input at that instant in time. You use these methods to separate the output calculation from the state updates of a System object. Implementing these two methods overrides the `stepImpl` method. These methods enable you to use the object in a feedback loop and prevent algebraic loops.

Subclass from the Nondirect Mixin Class

To use the `updateImpl`, `outputImpl`, and `isInputDirectFeedthroughImpl` methods, you must subclass from both the `matlab.System` base class and the `Nondirect` mixin class.

```
classdef IntegerDelaySysObj < matlab.System & ...
  matlab.system.mixin.Nondirect
```

Implement Updates to the Object

Implement an `updateImpl` method to update the object with previous inputs.

```
methods (Access = protected)
   function updateImpl(obj,u)
      obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
   end
end
```

Implement Outputs from Object

Implement an `outputImpl` method to output the previous, not the current input.

```
methods (Access = protected)
   function [y] = outputImpl(obj,~)
      y = obj.PreviousInput(end);
   end
end
```

Implement Whether Input Is Direct Feedthrough

Implement an `isInputDirectFeedthroughImpl` method to indicate that the input is nondirect feedthrough.

```
methods (Access = protected)
   function flag = isInputDirectFeedthroughImpl(~,~)
      flag = false;
   end
end
```

Complete Class Definition File with Update and Output

```
classdef intDelaySysObj < matlab.System &...
     matlab.system.mixin.Nondirect
   % intDelaySysObj Delay input by specified number of samples.

   properties
      InitialOutput = 0;
   end
```

```
    properties (Nontunable)
       NumDelays = 1;
    end
    properties (DiscreteState)
       PreviousInput;
    end

    methods (Access = protected)
       function validatePropertiesImpl(obj)
          if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
             error('Number of delays must be > 0 scalar value.');
          end
          if (numel(obj.InitialOutput)>1)
             error('Initial Output must be scalar value.');
          end
       end

       function setupImpl(obj)
          obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
       end

       function resetImpl(obj)
          obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
       end

       function [y] = outputImpl(obj,~)
          y = obj.PreviousInput(end);
       end
       function updateImpl(obj, u)
          obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
       end
       function flag = isInputDirectFeedthroughImpl(~,~)
          flag = false;
       end
    end
end
```

See Also

`isInputDirectFeedthroughImpl` | `matlab.system.mixin.Nondirect` | `outputImpl` | `updateImpl`

More About

- “Subclassing Multiple Classes”
- “Using ~ as an Input Argument in Method Definitions”

Enable For Each Subsystem Support

Enable For Each subsystem support by using a System object in a Simulink For Each subsystem. Include the `supportsMultipleInstanceImpl` method in your class definition file. This method applies only when the System object is used in Simulink via the MATLAB System block.

Use the `supportsMultipleInstanceImpl` method and have it return `true` to indicate that the System object supports multiple calls in a Simulink For Each subsystem.

```
methods (Access = protected)
   function flag = supportsMultipleInstanceImpl(obj)
       flag = true;
   end
end
```

View the method in the complete class definition file.

```
classdef RandSeed < matlab.System
% RANDSEED Random noise with seed for use in For Each subsystem

    properties (DiscreteState)
       count;
    end

    properties (Nontunable)
       seed = 20;
       useSeed (1,1) logical = false;
    end

   methods (Access = protected)
       function y = stepImpl(obj,u1)
         % Initial use after reset/setup
         % and use the seed
         if (obj.useSeed && ~obj.count)
             rng(obj.seed);
         end
         obj.count = obj.count + 1;
         [m,n] = size(u1);
         % Uses default rng seed
         y = rand(m,n) + u1;
       end

      function setupImpl(obj)
          obj.count = 0;
      end
      function resetImpl(obj)
          obj.count = 0;
      end

       function flag = supportsMultipleInstanceImpl(obj)
         flag = obj.useSeed;
       end
```

```
    end
end
```

See Also

`matlab.System` | `supportsMultipleInstanceImpl`

Define System Object for Use in Simulink

In this section...

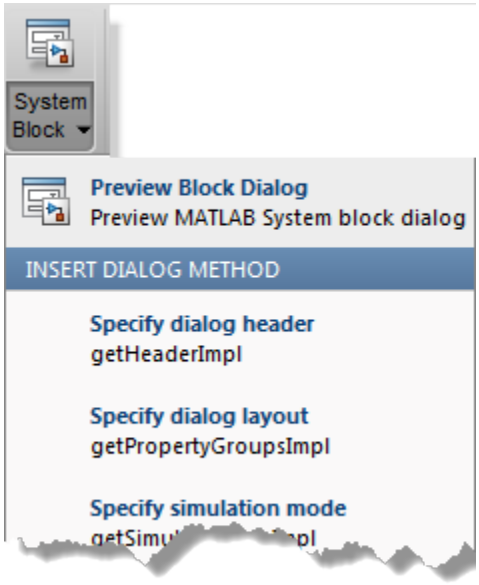
Develop System Object for Use in MATLAB System Block

You can develop a System object for use in a System block and interactively preview the block dialog box. This feature requires Simulink.

With the **System Block** editing options, the MATLAB Editor inserts predefined code into the System object. This coding technique helps you create and modify your System object faster and increases accuracy by reducing typing errors.

Using these options, you can also:

- View and interact with the block dialog box design as you define the System object.
- Add dialog box customization methods. If the block dialog box is open when you make changes, the block dialog design preview updates the display on saving the file.
- Add icon methods. However, these elements display only on the MATLAB System Block in Simulink, not in the **Preview Block Dialog**.



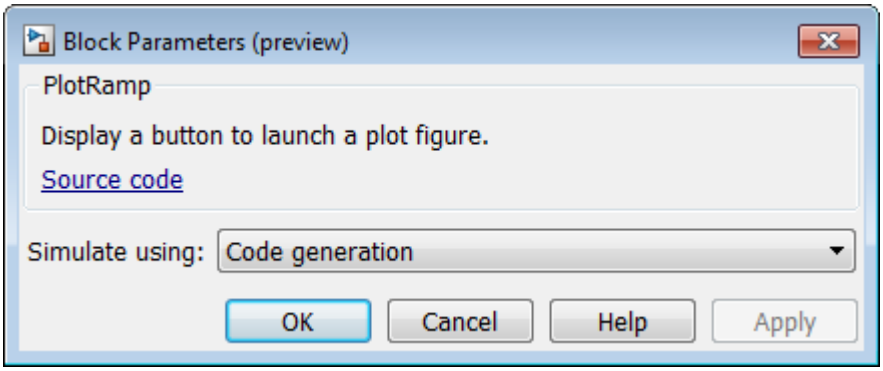
Define Block Dialog Box for Plot Ramp

1. Create a System object using the menu option **New > System Object > Simulink Extension**.
2. Name the System object `PlotRamp` and save the file. This name becomes the block dialog box title.
3. Delete the comment at the beginning of the file and replace it with the block description.

```
% Display a button to launch a plot figure.
```

This comment becomes the block parameters dialog box description, under the block title.

4 Select **System Block** > **Preview Block Dialog**. The block dialog box displays as you develop the System object.



5 Add a ramp limit by selecting **Insert Property** > **Numeric**. Then change the property name and set the value to `10`.

```
properties (Nontunable)
  RampLimit = 10;
end
```

6 Locate the `getPropertyGrouplsImpl` method using the **Analyze** button.

```
function group = getPropertyGroupsImpl
    % Define property section(s) for System block dialog
    group = matlab.system.display.Section(mfilename('class'));
end
```

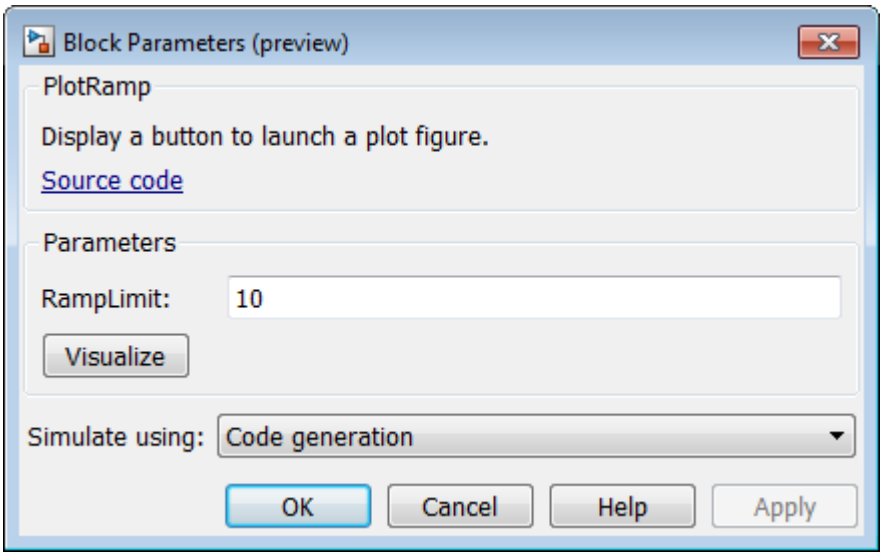
7 Create the group for the **Visualize** action.

```
function group = getPropertyGroupsImpl
    % Define property section(s) for System block dialog
    group = matlab.system.display.Section(mfilename('class'));
    group.Actions = matlab.system.display.Action(@(~,obj)...
      visualize(obj),'Label','Visualize');
end
```

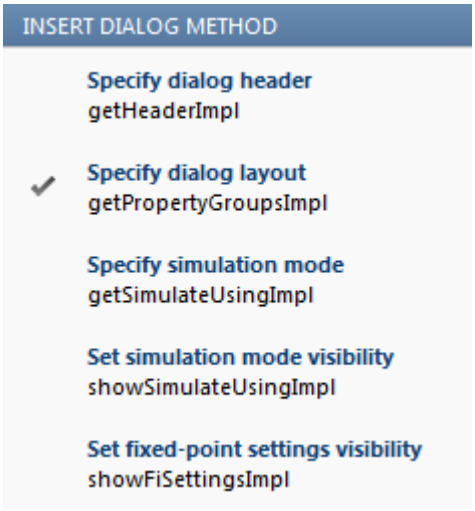
8 Add a function that adds code to display the **Visualize** button on the dialog box.

```
methods
  function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

9 As you add elements to the System block definition, save your file. Observe the effects of your code additions to the System block definition.



The **System Block** menu also displays checks next to the methods included in your file.



10 Delete any unused methods in the template or modify the methods to further customize the System object and System block. The class definition file now has all the code necessary for the `PlotRamp` System object.

```
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  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 visualize(obj)
        figure;
        d = 1:obj.RampLimit;
        plot(d);
      end
    end
  end
```

After you complete your System block definition, save it, and then load it into a MATLAB System block in Simulink.

See Also

Related Examples

- “Insert System Object Code Using MATLAB Editor”
- System Design in Simulink Using System Objects on page 45-98

Use Global Variables in System Objects

Global variables are variables that you can access in other MATLAB functions or Simulink blocks.

System Object Global Variables in MATLAB

For System objects that are used only in MATLAB, you define global variables in the System object class definition files in the same way that you define global variables in other MATLAB code (see “Global Variables”).

System Object Global Variables in Simulink

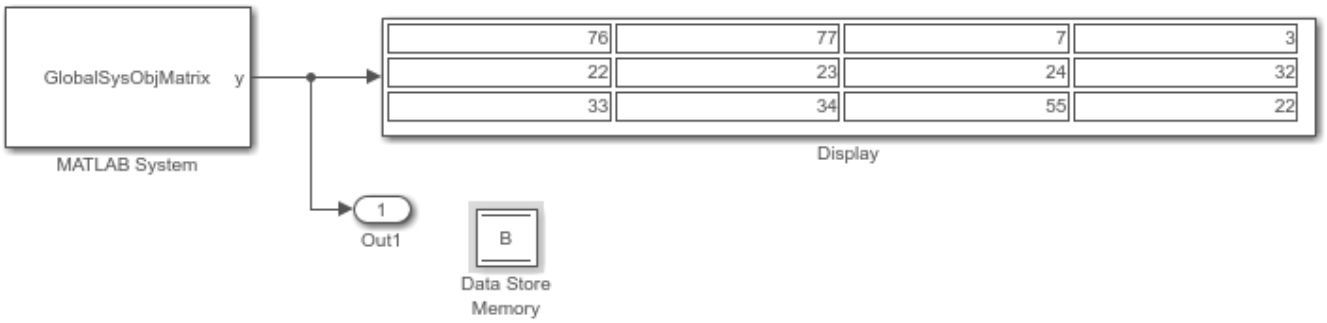
For System objects that are used in the MATLAB System block in Simulink, you also define global variables as you do in MATLAB. However, to use global variables in Simulink, if you have declared global variables in methods called by `stepImpl`, `updateImpl`, or `outputImpl`, you must declare global variables in the `stepImpl`, `updateImpl`, or `outputImpl` method, respectively.

You set up and use global variables for the MATLAB System block in the same way as you do for the MATLAB Function block (see “Data Stores” and “Share Data Globally” on page 44-91). Like the MATLAB Function block, you must also use variable name matching with a Data Store Memory block to use global variables in Simulink.

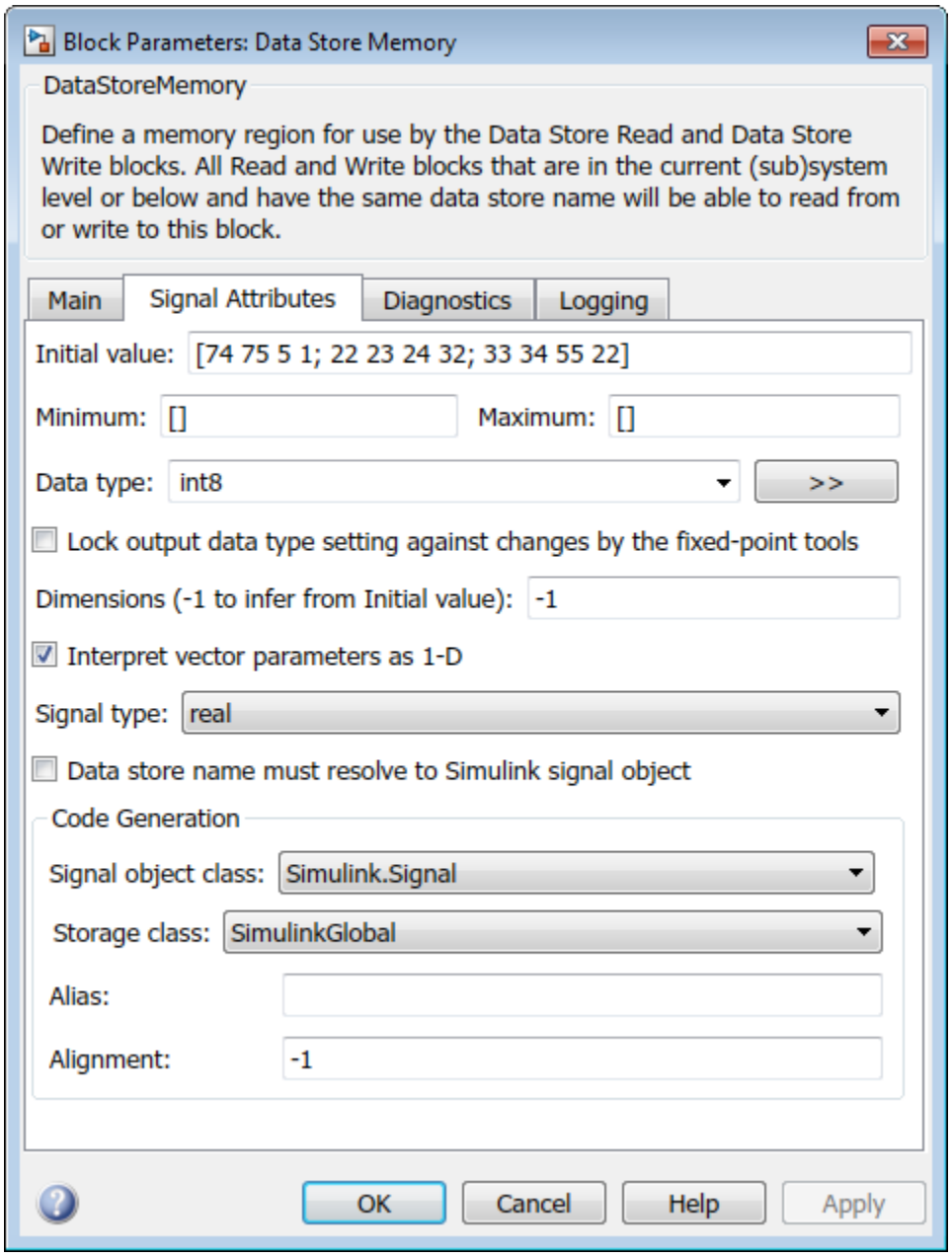
For example, this class definition file defines a System object that increments the first row of a matrix by 1 at each time step. If the file is P-coded, you must include `getGlobalNamesImpl`.

```
classdef GlobalSysObjMatrix < matlab.System
   methods (Access = protected)
      function y = stepImpl(obj)
         global B;
         B(1,:) = B(1,:)+1;
         y = B;
      end

      % Include getGlobalNamesImpl only if the class file is P-coded.
      function globalNames = getGlobalNamesImpl(~)
         globalNames = {'B'};
      end
   end
end
```

This model includes the `GlobalSysObjMatrix` object in a MATLAB System block and the associated Data Store Memory block.


GlobalSysObjMatrix
y
MATLAB System
76
77
7
3
22
23
24
32
33
34
55
22
Display
1
Out1
B
Data Store
Memory


Block Parameters: Data Store Memory
DataStoreMemory
Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.
Main
Signal Attributes
Diagnostics
Logging
Data store name: B
Rename All...
Corresponding Data Store Read/Write blocks:
refresh
globalMatrix/MATLAB System
OK
Cancel
Help
Apply



See Also

`getGlobalNamesImpl`

More About

- “Global Variables”
- “Share Data Globally” on page 44-91

System Design in Simulink Using System Objects

In this section...

System Design and Simulation in Simulink

You can use System objects in your model to simulate in Simulink.

1. Create a System object to be used in your model. See “Define New System Objects for Use in Simulink” on page 45-98 for information.
2. Test your new System object in MATLAB. See “Test New System Objects in MATLAB” on page 45-102
3. Add the System object to your model by using the MATLAB System block. See “Add System Objects to Your Simulink Model” on page 45-103 for information.
4. Add other Simulink blocks as needed and connect the blocks to construct your system.
5. Run the system

Define New System Objects for Use in Simulink

- “Define System Object with Block Customizations” on page 45-98
- “Define System Object with Nondirect Feedthrough” on page 45-101

A System object is a component you can use to create your system in MATLAB. You can write the code in MATLAB and use that code to create a block in Simulink. To define your own System object, you write a class definition file, which is a text-based MATLAB file that contains the code defining your object. See “Integrate System Objects Using MATLAB System Block”.

Define System Object with Block Customizations

Create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter.

Create a class definition text file to define your System object. The code in this example creates a least mean squares (LMS) filter and includes customizations to the block icon and dialog box appearance. It is similar to the “System Identification for an FIR System Using MATLAB System Blocks” Simulink example.

Note Instead of manually creating your class definition file, you can use the **New > System Object > Simulink Extension** menu option to open a template. This template includes customizations of the System object for use in the Simulink MATLAB System block. You edit the template file, using it as guideline, to create your own System object.

On the first line of the class definition file, specify the name of your System object and subclass from both `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class

enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon.

Add the appropriate basic System object methods to set up, reset, set the number of inputs and outputs, and run your algorithm. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `setupImpl` method to perform one-time calculations and initialize variables.
- Use the `stepImpl` method to implement the block's algorithm.
- Use the `resetImpl` method to reset the state properties or `DiscreteState` properties.
- Use the `getNumInputsImpl` and `getNumOutputsImpl` methods to specify the number of inputs and outputs, respectively.

Add the appropriate `CustomIcon` methods to define the appearance of the MATLAB System block in Simulink. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `getHeaderImpl` method to specify the title and description to display on the block dialog box.
- Use the `getPropertyGroupsImpl` method to specify groups of properties to display on the block dialog box.
- Use the `getIconImpl` method to specify the text to display on the block icon.
- Use the `getInputNamesImpl` and `getOutputNamesImpl` methods to specify the labels to display for the block input and output ports.

The full class definition file for the least mean squares filter is:

```
classdef lmsSysObj < matlab.System &...
      matlab.system.mixin.CustomIcon
   % lmsSysObj Least mean squares (LMS) adaptive filtering.
   % #codegen

   properties
      % Mu Step size
      Mu = 0.005;
   end

   properties (Nontunable)
      % Weights  Filter weights
      Weights = 0;
      % N  Number of filter weights
      N = 32;
   end

   properties (DiscreteState)
      X;
      H;
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.X = zeros(obj.N,1);
         obj.H = zeros(obj.N,1);
```

```
    end

    function [y, e_norm] = stepImpl(obj,d,u)
        tmp = obj.X(1:obj.N-1);
        obj.X(2:obj.N,1) = tmp;
        obj.X(1,1) = u;
        y = obj.X'*obj.H;
        e = d-y;
        obj.H = obj.H + obj.Mu*e*obj.X;
        e_norm = norm(obj.Weights'-obj.H);
    end

    function resetImpl(obj)
        obj.X = zeros(obj.N,1);
        obj.H = zeros(obj.N,1);
    end

end

% Block icon and dialog customizations
methods (Static, Access = protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header(...
             'lmsSysObj', ...
             'Title', 'LMS Adaptive Filter');
    end

    function groups = getPropertyGroupsImpl
        upperGroup = matlab.system.display.SectionGroup(...
             'Title','General',...
             'PropertyList',{'Mu'});

        lowerGroup = matlab.system.display.SectionGroup(...
             'Title','Coefficients', ...
             'PropertyList',{'Weights','N'});

        groups = [upperGroup,lowerGroup];
    end
end

methods (Access = protected)
    function icon = getIconImpl(~)
        icon = sprintf('LMS Adaptive\nFilter');
    end
    function [in1name, in2name] = getInputNamesImpl(~)
        in1name = 'Desired';
        in2name = 'Actual';
    end
    function [out1name, out2name] = getOutputNamesImpl(~)
        out1name = 'Output';
        out2name = 'EstError';
    end
```

```
    end
end
```

Define System Object with Nondirect Feedthrough

Create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter and uses feedback loops.

Create a class definition text file to define your System object. The code in this example creates an integer delay and includes feedback loops, and customizations to the block icon. For information on feedback loops, see “Use System Objects in Feedback Loops” on page 45-16. This example implements a System object that you can use for nondirect feedthrough. It is similar to the “System Identification for an FIR System Using MATLAB System Blocks” Simulink example.

On the first line of the class definition file, subclass from `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon. The `Nondirect` mixin enables the methods that let you specify how the block is updated and what it outputs.

Add the appropriate basic System object methods to set up and reset the object and set and validate the properties. Since this object supports nondirect feedthrough, you do not implement the `stepImpl` method. You implement the `updateImpl` and `outputImpl` methods instead. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `setupImpl` method to initialize some of the object’s properties.
- Use the `resetImpl` method to reset the property states.
- Use the `validatePropertiesImpl` method to check that the property values are valid.

Add the following `Nondirect` mixin class methods instead of the `stepImpl` method to specify how the block updates its state and its output. See the reference pages and the full class definition file below for the implementation of each of these methods.

- Use the `outputImpl` method to implement code to calculate the block output.
- Use the `updateImpl` method to implement code to update the block’s internal states.
- Use the `isInputDirectFeedthroughImpl` method to specify that the block is not direct feedthrough. Its inputs do not directly affect its outputs.

Add the `getIconImpl` method to define the block icon when it is used in Simulink via the MATLAB System block. See the reference page and the full class definition file below for the implementation of this method.

The full class definition file for the delay is:

```
classdef intDelaySysObj < matlab.System &...
     matlab.system.mixin.Nondirect &...
     matlab.system.mixin.CustomIcon
   % intDelaySysObj Delay input by specified number of samples.
   % #codegen

   properties
      % InitialOutput Initial output
```

```
        InitialOutput = 0;
    end

    properties (Nontunable)
        % NumDelays Number of delays
        NumDelays = 1;
    end

    properties (DiscreteState)
        PreviousInput;
    end

    methods (Access = protected)
        function setupImpl(obj, ~)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function [y] = outputImpl(obj, ~)
            % Output does not directly depend on input
            y = obj.PreviousInput(end);
        end

        function updateImpl(obj, u)
            obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
        end

        function flag = isInputDirectFeedthroughImpl(~,~)
            flag = false;
        end

        function validatePropertiesImpl(obj)
            if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
                error('Number of delays must be positive non-zero ...
                  scalar value.');
            end
            if (numel(obj.InitialOutput)>1)
                error('Initial output must be scalar value.');
            end
        end

        function resetImpl(obj)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function icon = getIconImpl(~)
            icon = sprintf('Integer\nDelay');
        end
    end
end
```

Test New System Objects in MATLAB

1 Create an instance of your new System object. For example, create an instance of the `lmsSysObj`.

```
s = lmsSysObj;
```

2 Run the object multiple times with different inputs. Doing this step tests for syntax errors and other possible issues before you add it to Simulink. For example,

```
desired = 0;
actual = 0.2;
s(desired,actual);
```

Add System Objects to Your Simulink Model

System Objects in the MATLAB Function Block

You can include System object code in Simulink models with the MATLAB Function block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Objects in the MATLAB System Block

You can include individual System objects that you create with a class definition file into Simulink with the MATLAB System block. This option is one way to add your own algorithm blocks into your Simulink models.

Add your System objects to your Simulink model by using the MATLAB System block as described in “Mapping System Object Code to MATLAB System Block Dialog Box” on page 45-19.

For information, see “Integrate System Objects Using MATLAB System Block”.

Specify Sample Time for MATLAB System Block System Objects

This example shows how to control the sample time of the MATLAB System block using System object™ methods.

Inside the class definition, use the System object sample time methods to configure the sample time and modify the System object behavior based on the current simulation time. If you want to use inherited sample time, you do not need to specify a sample time in your System object definition.

Specify Sample Time

To specify the sample time, implement the `getSampleTimeImpl` method and create a sample time specification object with `createSampleTime`.

In this example, a property `SampleTimeTypeProp` is created to assign the sample time based on different property values. The `getSampleTimeImpl` method creates a sample time specification based on the `SampleTimeTypeProp` property. The `getSampleTimeImpl` method returns the sample time specification object `sts` created by `createSampleTime`.

```
18        methods(Access = protected)
19            function sts = getSampleTimeImpl(obj)
20                switch char(obj.SampleTimeTypeProp)
21                    case 'Inherited'
22                        sts = createSampleTime(obj,'Type','Inherited');
23                    case 'InheritedNotControllable'
24                        sts = createSampleTime(obj,'Type','Inherited',...
25                            'AlternatePropagation','Controllable');
26                    case 'InheritedErrorConstant'
27                        sts = createSampleTime(obj,'Type','Inherited',...
28                            'ErrorOnPropagation','Constant');
29                    case 'FixedInMinorStep'
30                        sts = createSampleTime(obj,'Type','Fixed In Minor Step');
31                    case 'Discrete'
32                        sts = createSampleTime(obj,'Type','Discrete',...
33                          'SampleTime',obj.SampleTime, ...
34                          'OffsetTime',obj.OffsetTime);
35                    case 'Controllable'
36                        sts = createSampleTime(obj,'Type','Controllable',...
37                            'TickTime',obj.TickTime);
38                end
39            end
```

Query Simulation Time and Sample Time

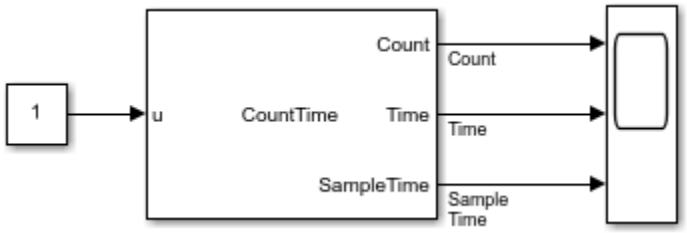
Use the `getSampleTime` and `getCurrentTime` methods to query the MATLAB System block for the current sample time and simulation time, respectively. `getSampleTime` returns a sample time specification object with properties describing the sample time settings.

```
40
41            function [Count, Time, SampleTime] = stepImpl(obj,u)
42                Count = obj.Count + u;
43                obj.Count = Count;
44                Time = getCurrentTime(obj);
45                sts = getSampleTime(obj);
```

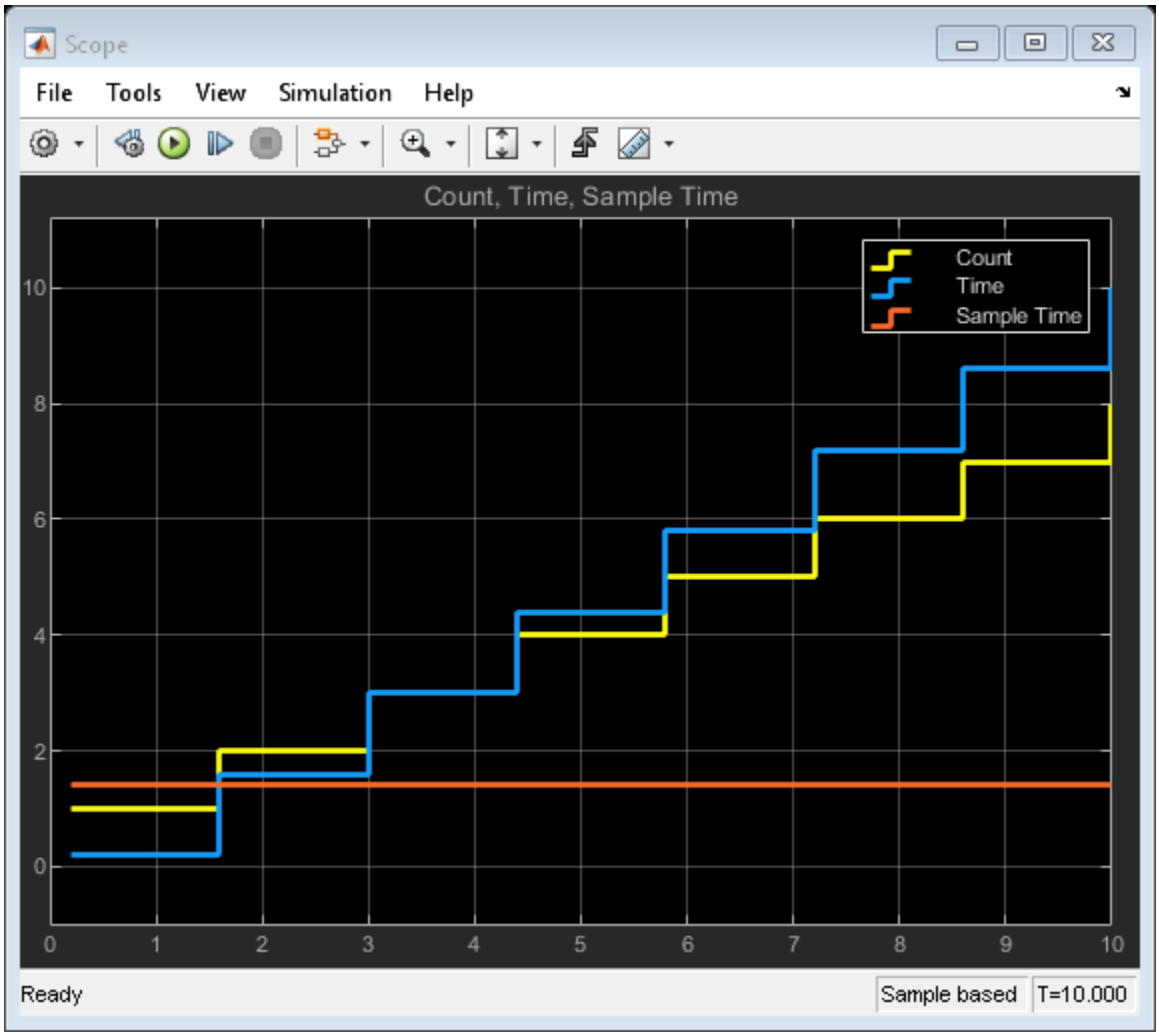
```
46                 if strcmp(sts.Type,'Controllable')
47                    setNumTicksUntilNextHit(obj,obj.Count);
48                 end
49                 SampleTime = sts.SampleTime;
50             end
```

Behavior in Simulink

Include this System object in a MATLAB System block.



In the scope, you can see the effects of the sample time changes to the block.



Full Class Definition

Full class definition of the `CountTime` System object.

```
classdef CountTime < matlab.System
    % Counts Hits and Time

    properties(Nontunable)
        SampleTime = 1.4; % Sample Time
        OffsetTime = 0.2; % Offset Time
        TickTime = 0.1;
        SampleTimeTypeProp (1, 1) {mustBeMember(SampleTimeTypeProp, ...
            ["Discrete","FixedInMinorStep","Controllable",...
            "Inherited","InheritedNotControllable",...
            "InheritedErrorConstant"])} = "Discrete"
    end

    properties(DiscreteState)
        Count
    end

    methods(Access = protected)
        function sts = getSampleTimeImpl(obj)
            switch char(obj.SampleTimeTypeProp)
                case 'Inherited'
                    sts = createSampleTime(obj,'Type','Inherited');
                case 'InheritedNotControllable'
                    sts = createSampleTime(obj,'Type','Inherited',...
                        'AlternatePropagation','Controllable');
                case 'InheritedErrorConstant'
                    sts = createSampleTime(obj,'Type','Inherited',...
                        'ErrorOnPropagation','Constant');
                case 'FixedInMinorStep'
                    sts = createSampleTime(obj,'Type','Fixed In Minor Step');
                case 'Discrete'
                    sts = createSampleTime(obj,'Type','Discrete',...
                      'SampleTime',obj.SampleTime, ...
                      'OffsetTime',obj.OffsetTime);
                case 'Controllable'
                    sts = createSampleTime(obj,'Type','Controllable',...
                        'TickTime',obj.TickTime);
            end
        end

        function [Count, Time, SampleTime] = stepImpl(obj,u)
            Count = obj.Count + u;
            obj.Count = Count;
            Time = getCurrentTime(obj);
            sts = getSampleTime(obj);
            if strcmp(sts.Type,'Controllable')
               setNumTicksUntilNextHit(obj,obj.Count);
            end
            SampleTime = sts.SampleTime;
        end

        function setupImpl(obj)
            obj.Count = 0;
```

```
        end

        function resetImpl(obj)
            % Initialize / reset discrete-state properties
            obj.Count = 0;
        end

        function flag = isInactivePropertyImpl(obj,prop)
            flag = false;
            switch char(obj.SampleTimeTypeProp)
                case {'Inherited', ...
                        'InheritedNotControllable', ...
                        'FixedInMinorStep'}
                    if any(strcmp(prop,{'SampleTime','OffsetTime','TickTime'}))
                        flag = true;
                    end
                case 'Discrete'
                    if any(strcmp(prop,{'TickTime'}))
                        flag = true;
                    end
                case 'Controllable'
                    if any(strcmp(prop,{'SampleTime','OffsetTime'}))
                        flag = true;
                    end
            end
        end
    end
end
```

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `matlab.system.mixin.SampleTime` | `setNumTicksUntilNextHit`

More About

- “What Is Sample Time?” on page 7-2

Create Moving Average Filter Block with System Object

This example shows how to extend the `movingAverageFilter` System object™ for use in Simulink™. To use a System object in Simulink, include the System object in a MATLAB System block.

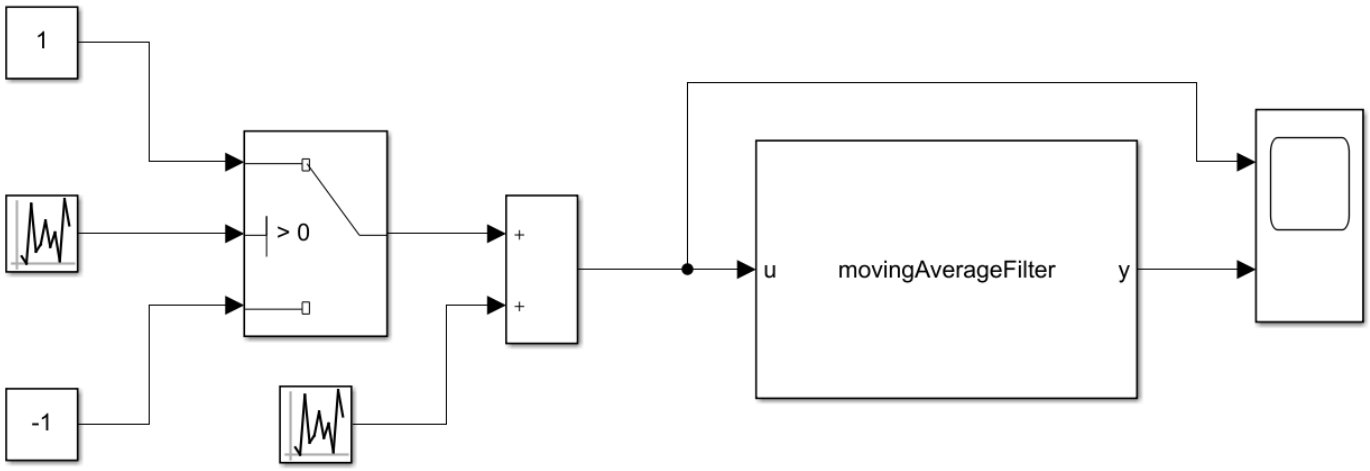
`movingAverageFilter` System Object

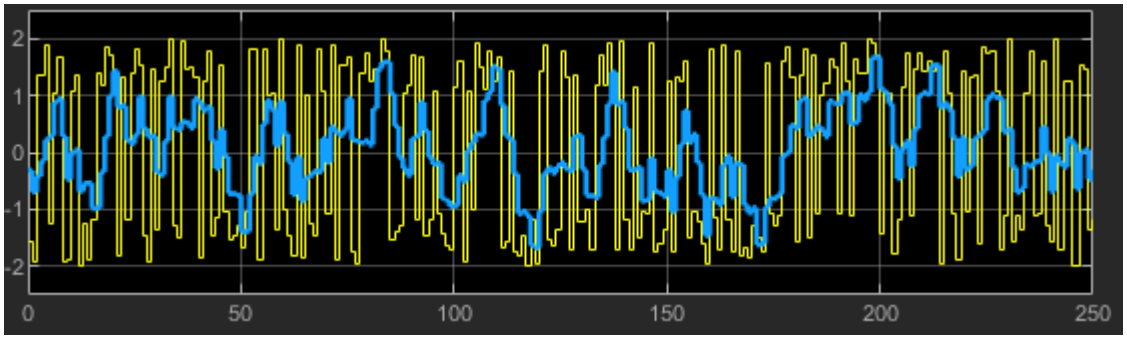
This example extends the `movingAverageFilter` System object built in “Create Moving Average System object”. The `movingAverageFilter` System object computes the unweighted mean of a specified number of previous inputs. Use the `WindowLength` property to specify how many previous samples to use.

Use in Simulink

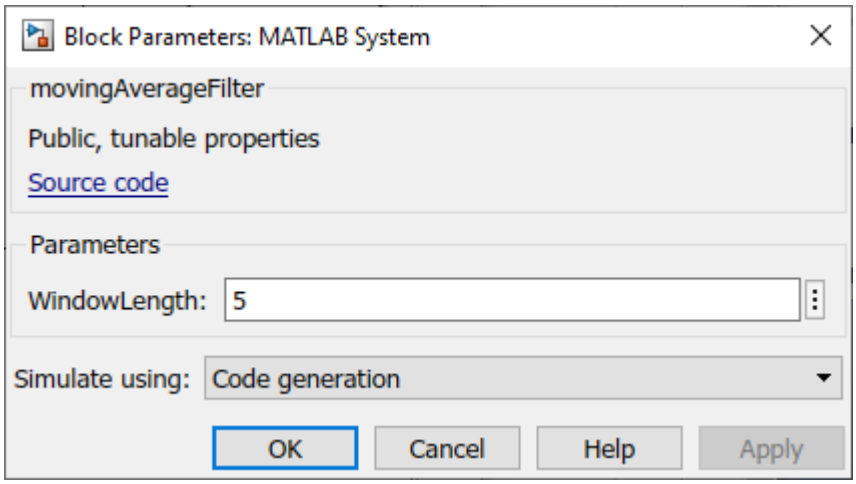
The object is already ready to use in Simulink. Create a Simulink model and add a MATLAB System block. Specify `movingAverageFilter` as the System object name. For example, this model uses the moving average filter to eliminate noise from a signal.

```
model = 'movingaveragefilter_sl';
open_system(model);
```





The block dialog window shows the public, tunable parameters:



Customize MATLAB System Block

Optionally, you can customize the block appearance and block dialog for a MATLAB System block by adding methods to the System object.

Add Simulink Block Icon Customization Method

By default the block icon shows the name of the System object, in this case `movingAverageFilter`. Customize the Moving Average Filter block icon with a cleaner name. In the **Editor** toolstrip, select the **System Block** dropdown button, then select **Add Text Icon**. The `getIconImpl` method is added to `movingAverageFilter`. Inside `getIconImpl`, set `icon` equal to the string array `["Moving","Average","Filter"];`

```
function icon = getIconImpl(~)
    % Define icon for the System block.
    icon = ["Moving","Average","Filter"];
end
```

Customize Block Dialog

You can also customize the block dialog by adding methods and comments to the System object. For details about block dialog customization, see “Customize System Block Appearance” on page 45-60. In this example, rename the `WindowLength` property in the dialog box and add a method to customize the description.

By default, all public properties appear as parameters in the block dialog with their property names. In this example, add comments before the `WindowLength` property so that it appears as **Moving window length** in the dialog. Add a comments above the property in the form: `PropertyName Name in dialog`

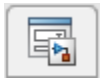
```
 % WindowLength Moving window length
 WindowLength (1,1){mustBeInteger,mustBePositive} = 5
```

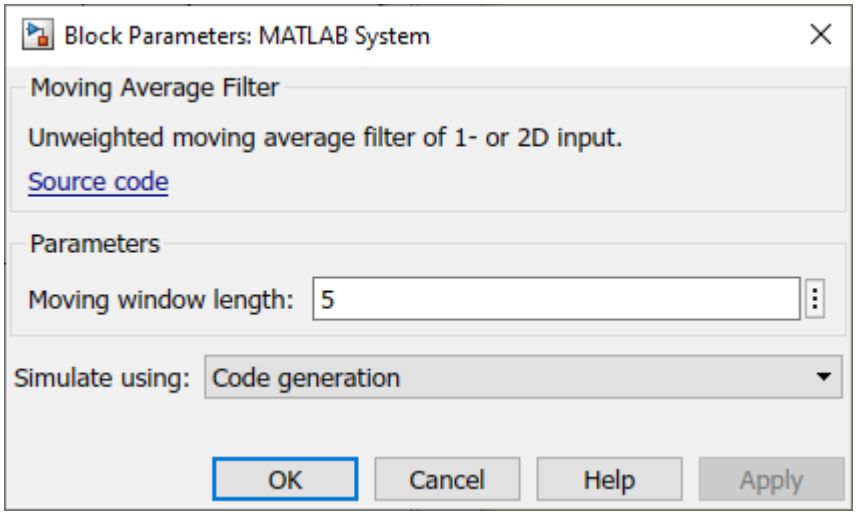
To specify the header and description in the block dialog, in the toolstrip select **System Block > Specify Dialog Header**. This option adds the `getHeaderImpl` method to `movingAverageFilter`. Modify the call to `matlab.system.display.Header` to this:

```
methods(Access = protected, Static)
    function header = getHeaderImpl
```

```
            % Define header panel for System block dialog
            header = matlab.system.display.Header('movingAverageFilter',...
                'Title','Moving Average Filter',...
                'Text', 'Unweighted moving average filter of 1- or 2D input.');
        end
    end
```

You can see a preview of the block dialog by clicking the button in the toolstrip above **System Block**.

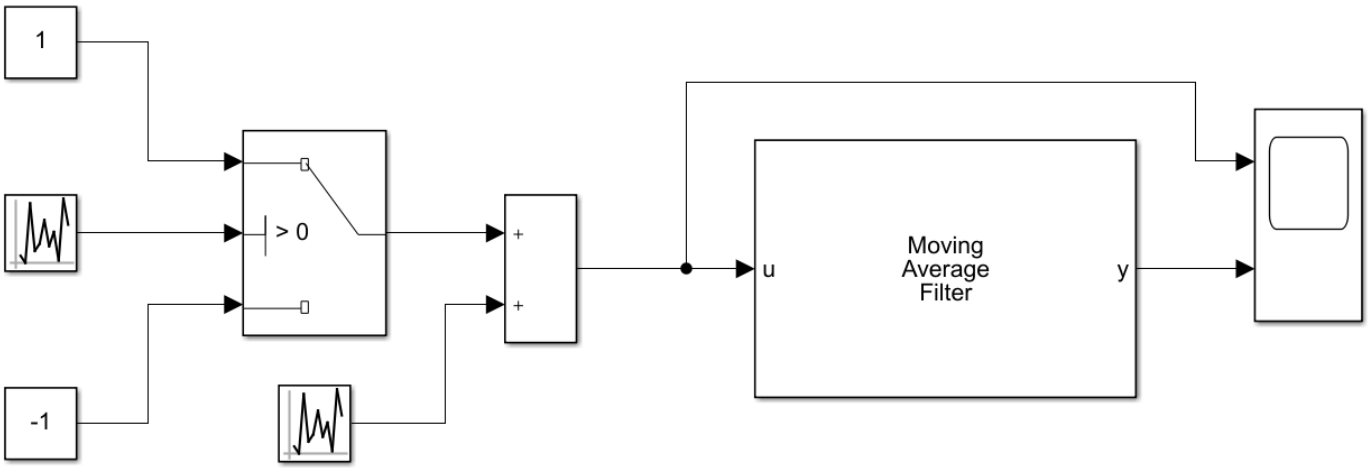




Customized Block in Simulink

This is the block with the added customizations:

```
model = 'movingaveragefilter_sl_extended';
open_system(model);
```



To see the completed System object with the Simulink customization methods, type:

```
edit movingAverageFilter_extended.m
```

See Also

Manage and Create a Blockset Using Blockset Designer

Create a Blockset Project

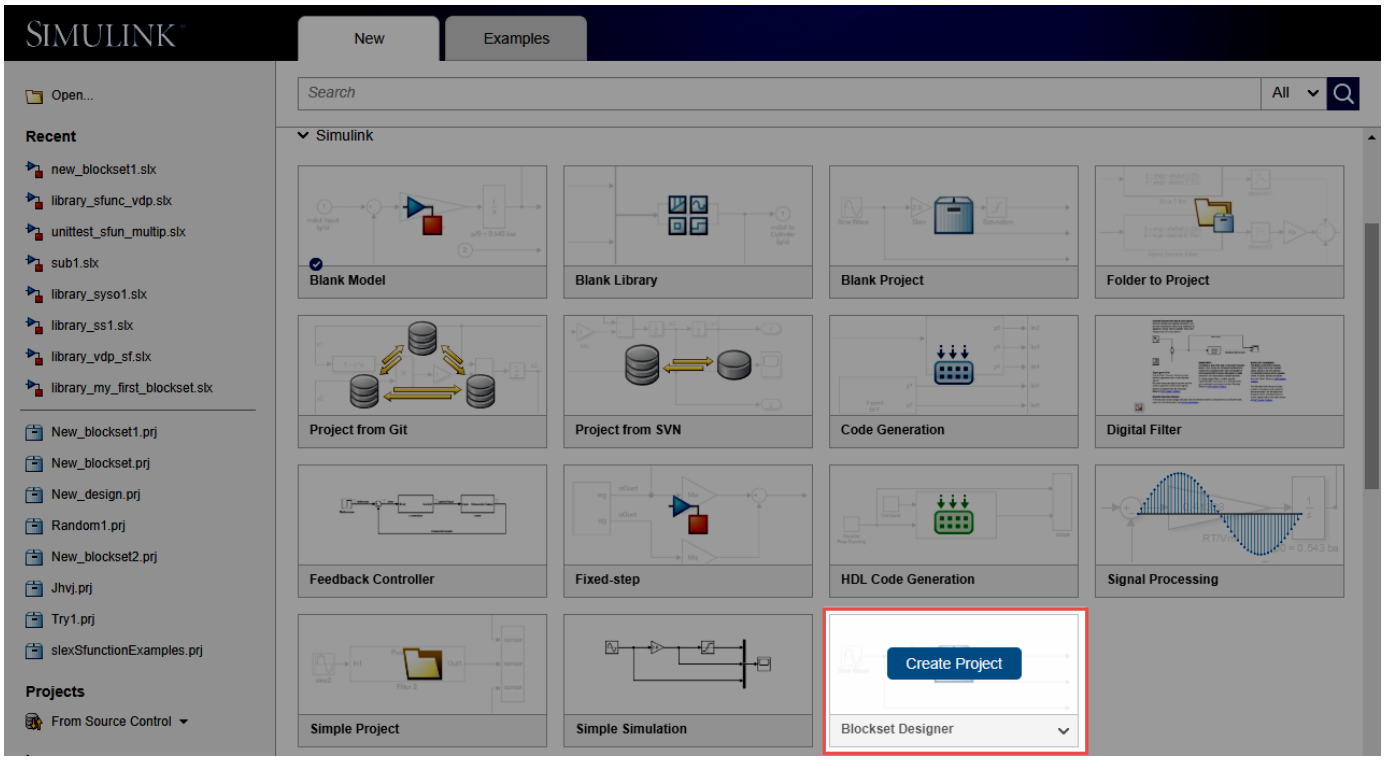
In this section...

A blockset is a collection of blocks organized in Simulink libraries for a purpose. The blocks could be part of a standalone library or may be part of an extensive project.

The Blockset Designer is a Projects-based tool that allows you to create, group and manage custom blocks. After creating different blocks, you can add tests, document your blocks, and run Model Advisor checks. If your model contains any S-function or S-function Builder blocks, you can build them using the interface. In addition, you can import your existing blocksets, and set up a Blockset Designer project. You can create new Subsystem, MATLAB System, S-function, and S-function Builder blocks using the Blockset Designer.

Create a New Blockset Project

1 Open the Simulink start page, and pause on Blockset Designer, and click **Create Project**.

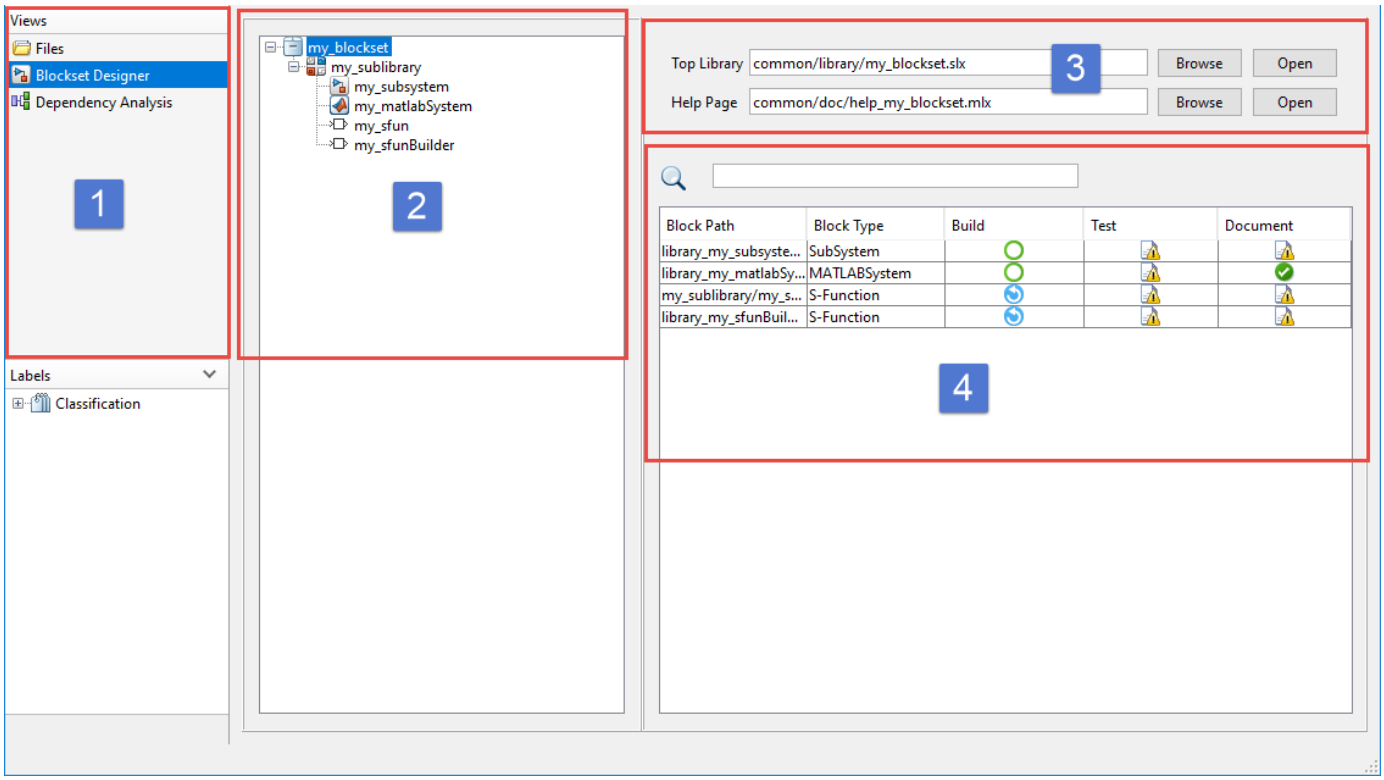


2 Specify a name for your project. Note that when you start typing a name for your project, a new folder is automatically created for you in the directory username\MATLAB\Projects \<foldername>. If you would like to specify an existing project folder, click the **Browse** button.

If you create a new folder, click **OK** to create a new project folder. Confirm this action by clicking **Yes**.

3 Now you are directed to the Blockset Designer project user interface. You are looking at the project at the blockset level. In a Blockset Designer project, the  icon represents the blockset you are working on.

The `Blockset Designer` consists of these components at the blockset level:



1 — The Views panel shows the project file views and the interface of the `Blockset Designer`. To view all files in the project root, click **Files**, and then the **All** tab. To only see the files in the project, click **Project**.

2 — The Blockset Tree panel shows the organization of the blocks and sublibraries in the project. When you add a new block or a new sublibrary to your blockset, it will show up in this tree. You can also use this tree to switch between the sublibrary and block control menus.

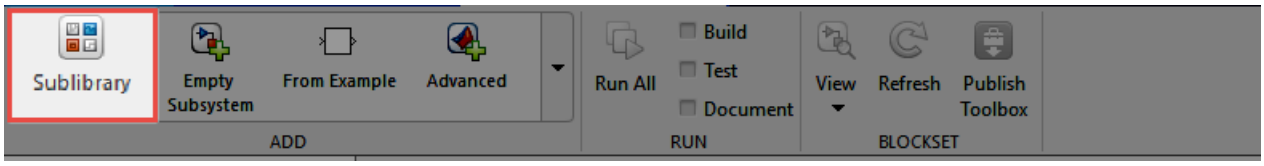
3 — The **Top Library** corresponds to the `Browser.Library` in the library information file `slblocks.m`, which is the entry point to the blockset. Click **Open** to display the location of the top library. Click **Browse** to change the top library for the blockset. Note that if you change the top library, `slblocks.m` updates accordingly, and your blockset project is reloaded based on the new top library. See “Add Libraries to the Library Browser” on page 41-7 to learn more about `slblocks.m`

4 — The status table shows the status of the blocks and their artifacts. For more information on the status table, see “Create and Organize Block Artifacts” on page 46-14.

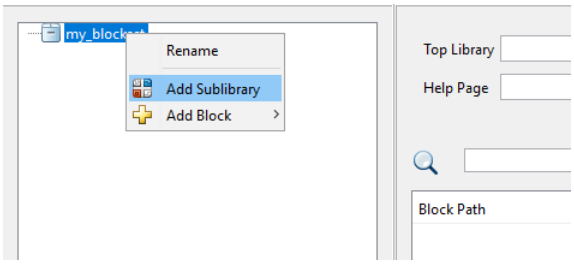
Add a New Sublibrary

Start building your blockset project by adding a sublibrary. Sublibraries enable you to organize and manage your blocks. In your blockset project, the  icon represents the sublibraries. You can add a new sublibrary to your blockset using in these ways:

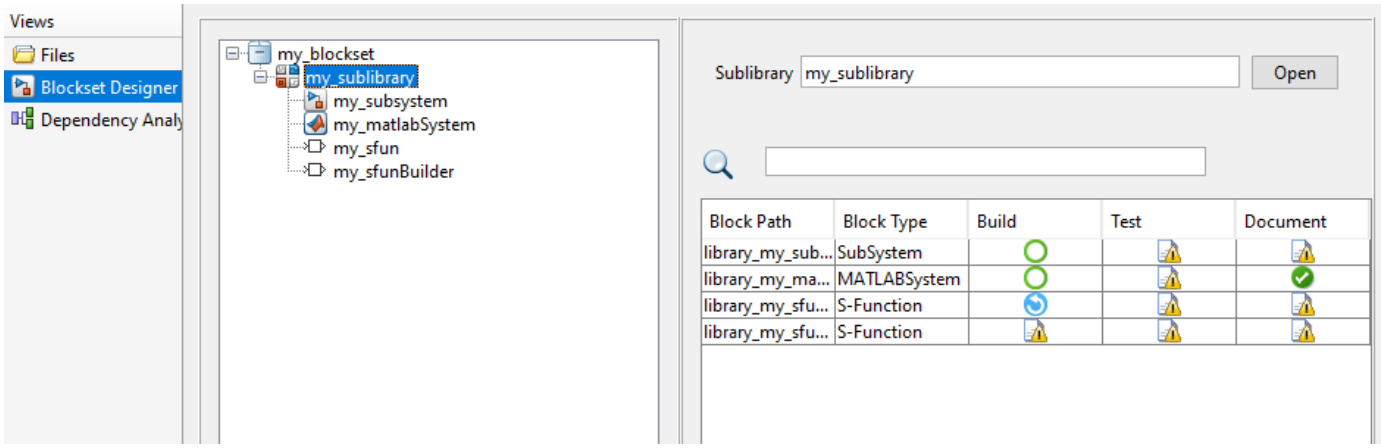
1 From the toolstrip, select the **Sublibrary** button.



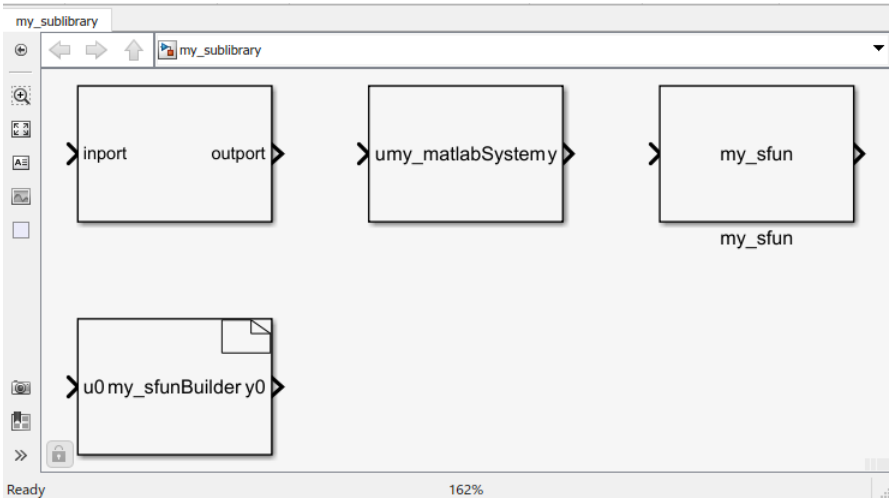
2 Click the blockset from the blockset tree, and from the context menu, select **Sublibrary**.



3 Name your sublibrary and close the window. After creating sublibraries, click one of them and see the changes in the working area. Observe that on the top right of the working area, you see **Sublibrary** instead of the **Top Library** and the **Help Page**.



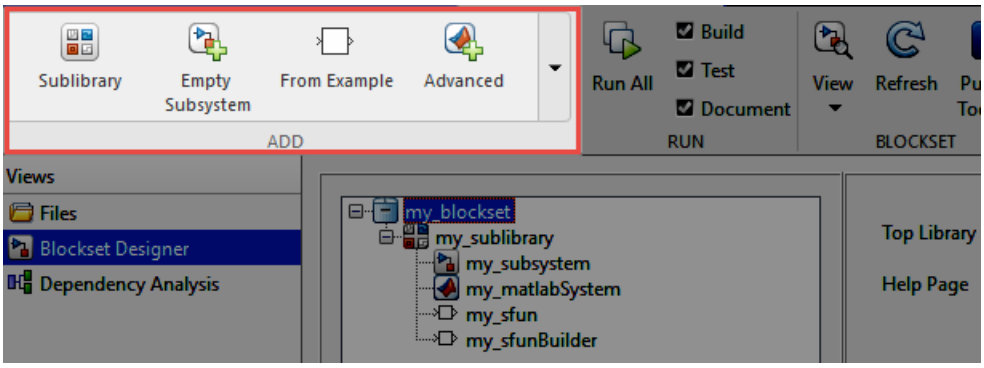
4 Click **Open** on the right to open the library model constructed from your sublibrary. This library model contains all blocks from the selected sublibrary.



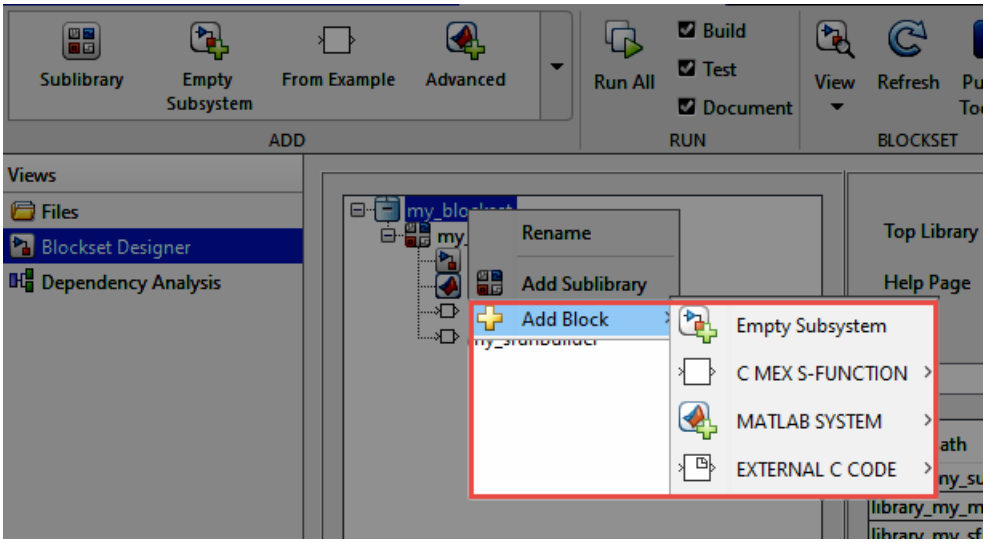
Add a New Block

Blockset Designer supports many custom blocks.

- You can add a new block to your blockset project from the **ADD** menu on the toolstrip



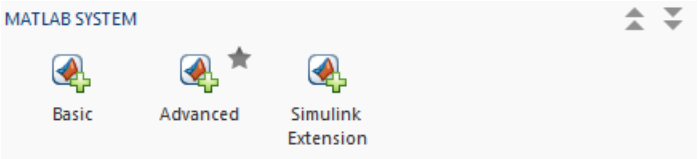
- Alternatively, when you select the blockset or the sublibrary, open the context menu and click **Add Block**.



You can create these blocks in the Blockset Designer:

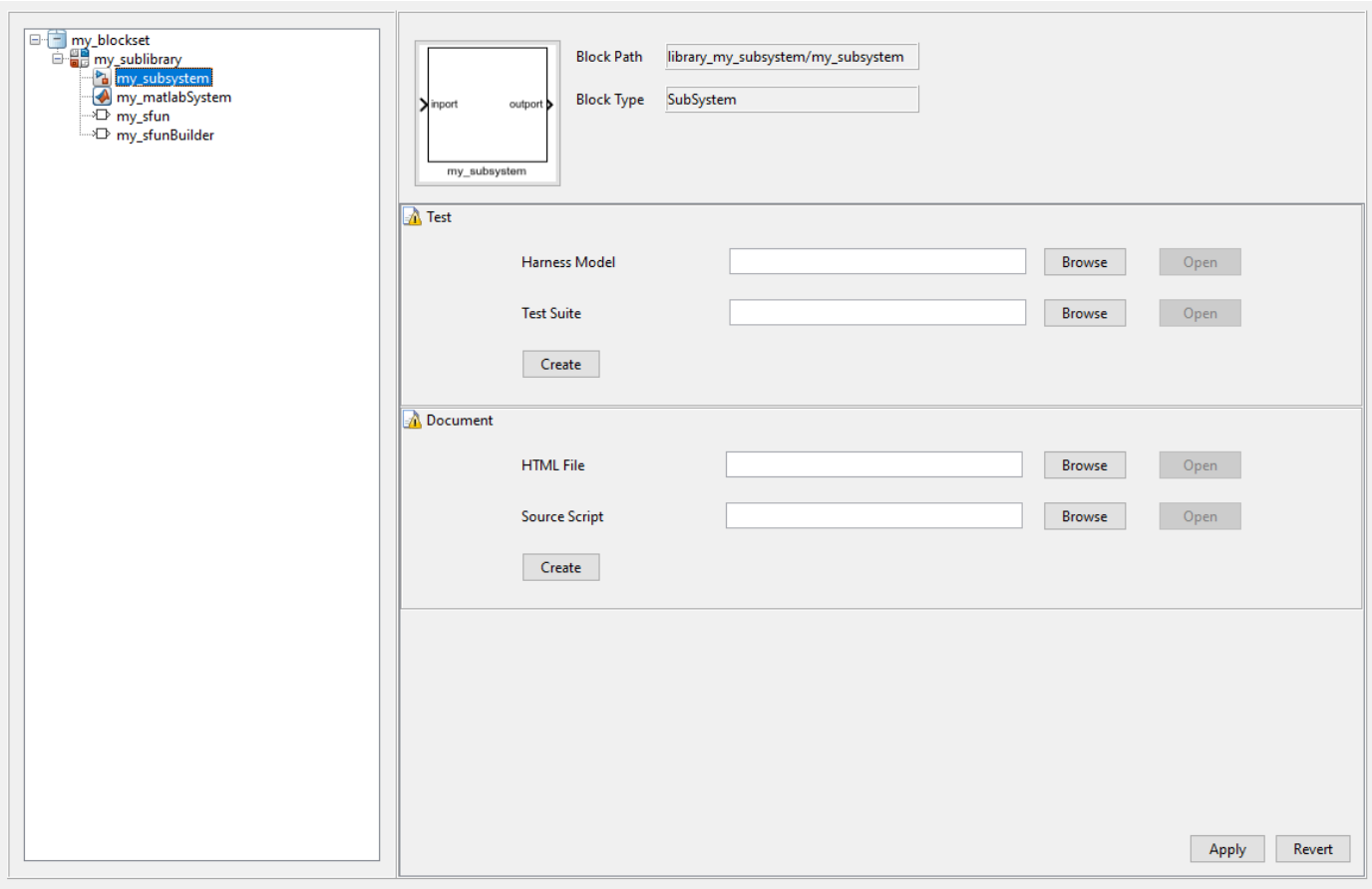
- C MEX S-function blocks in basic, discrete, continuous templates. You can also create a C MEX S-function from an existing example.

- Subsystem block
- MATLAB System block with basic, advanced, and Simulink Extension System object templates.

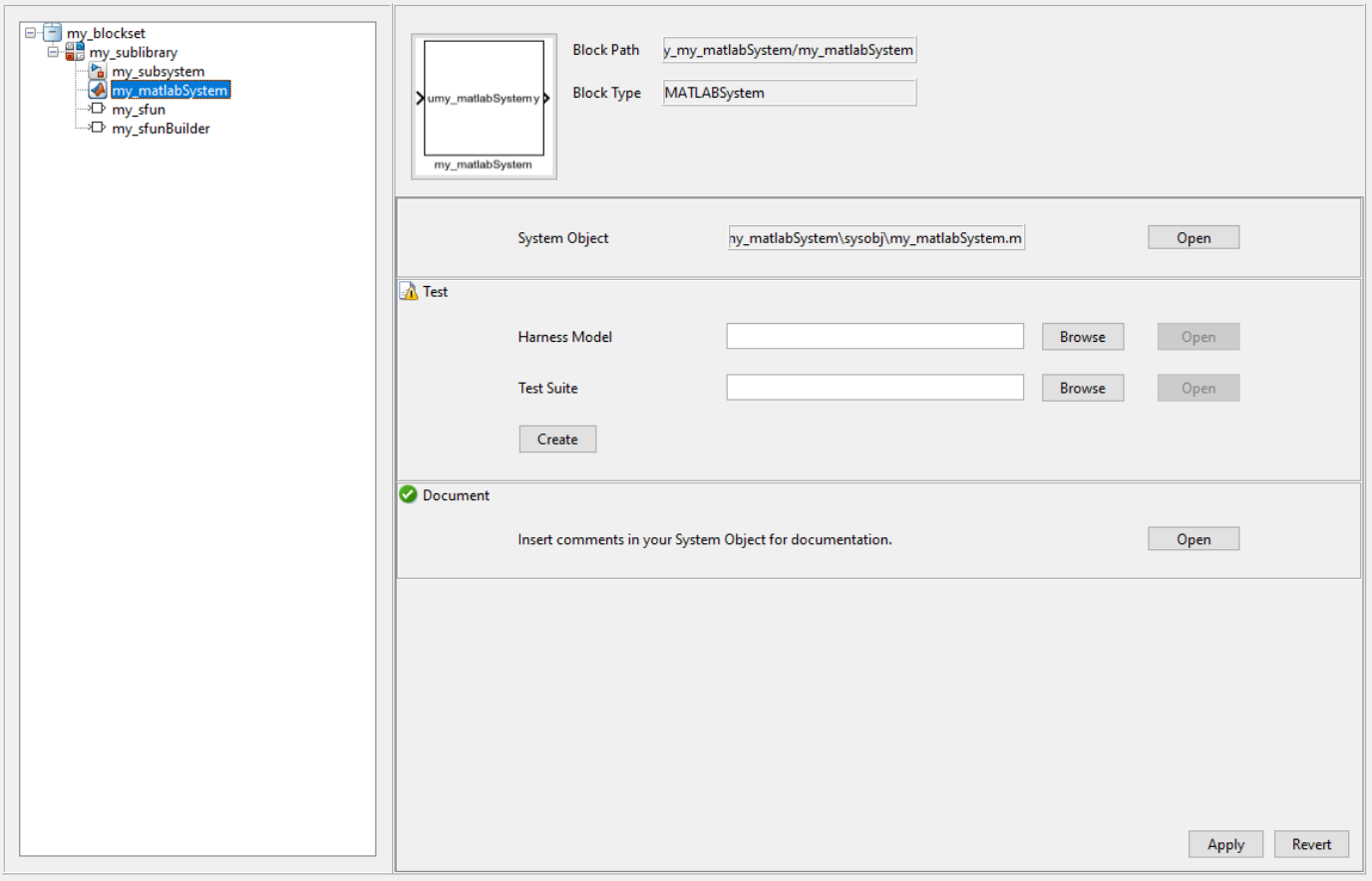


- S-functions created using the S-function Builder.

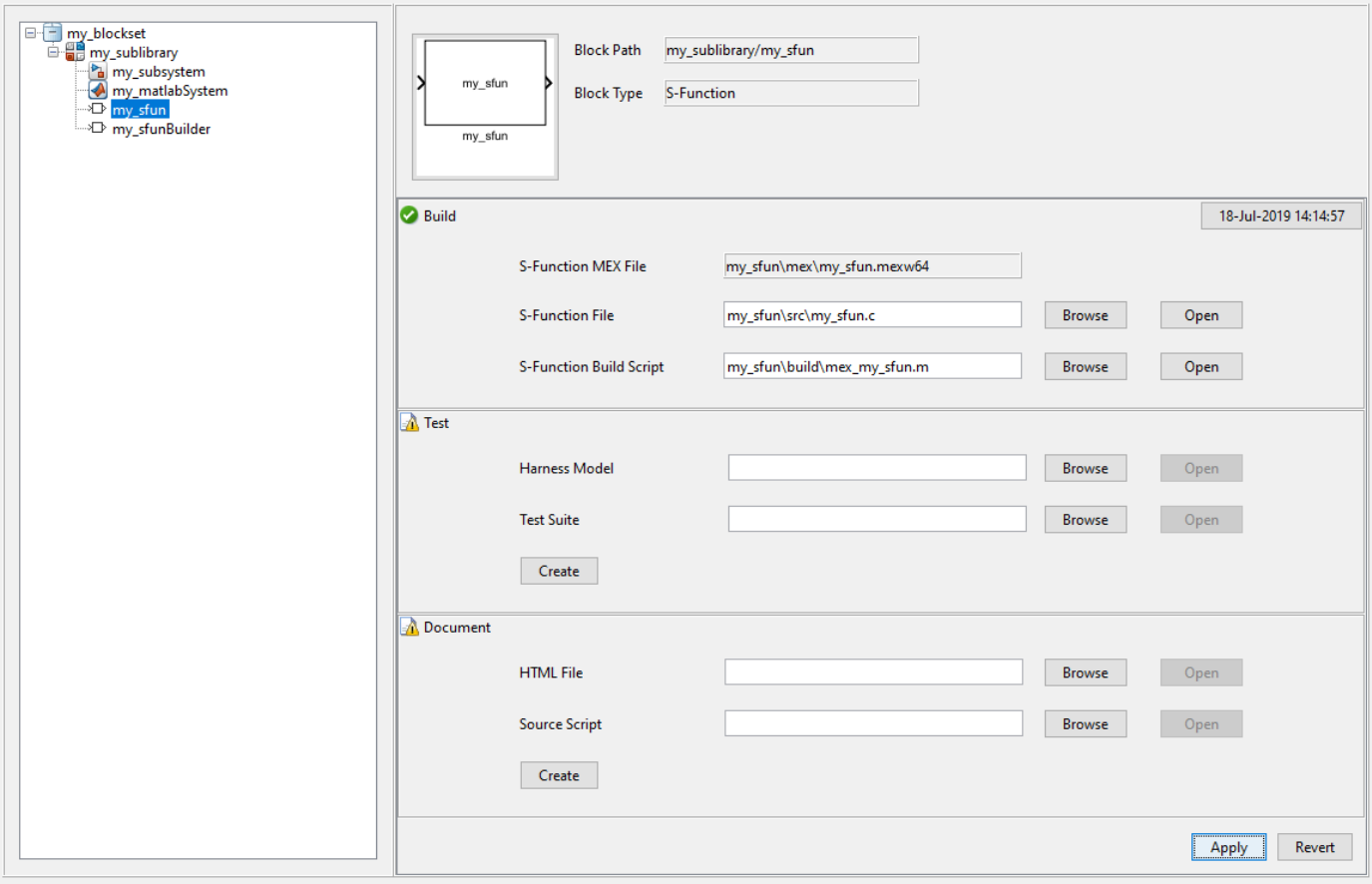
The working area changes according to the blocks added to your project. For a Subsystem block, you see block properties such as the block path and type, **Test**, and **Documentation** in the working area.



When you click a MATLAB System block from the blockset tree, you can edit the System object code and add **Test** using the working area. To add documentation for a MATLAB System block, add comments in the System object code, and they are automatically added to block documentation for you.



If you have an S-function or a S-function Builder block in your blockset project, you can monitor the block properties, build and edit your S-function code, build the script, add test and document to your block.



Note that when you add an S-function to your Blockset Designer project, you need to write the code for your S-function and build it before adding test. To learn more about how to build S-functions using the Blockset Designer, see “S-Function Related Artifacts” on page 46-15.

Besides the blocks mentioned above, you can add and customize other available Simulink library blocks in the project, but you cannot do any operations such as adding test or documentation.

When you are finished creating your project, you can add:

- Test, or a test suite for your blocks.
- Documentation for your sublibraries and/or for your blocks.

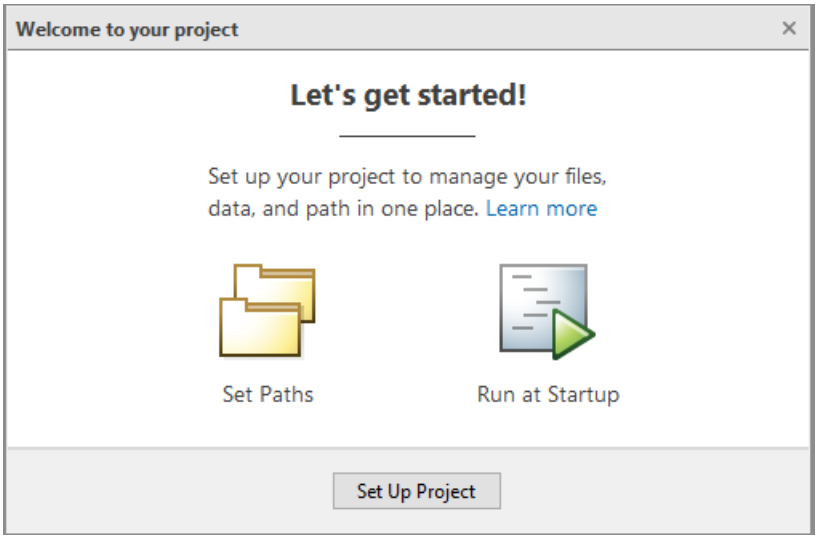
See “Create and Organize Block Artifacts” on page 46-14 for more information.

Create a Project from an Existing Blockset

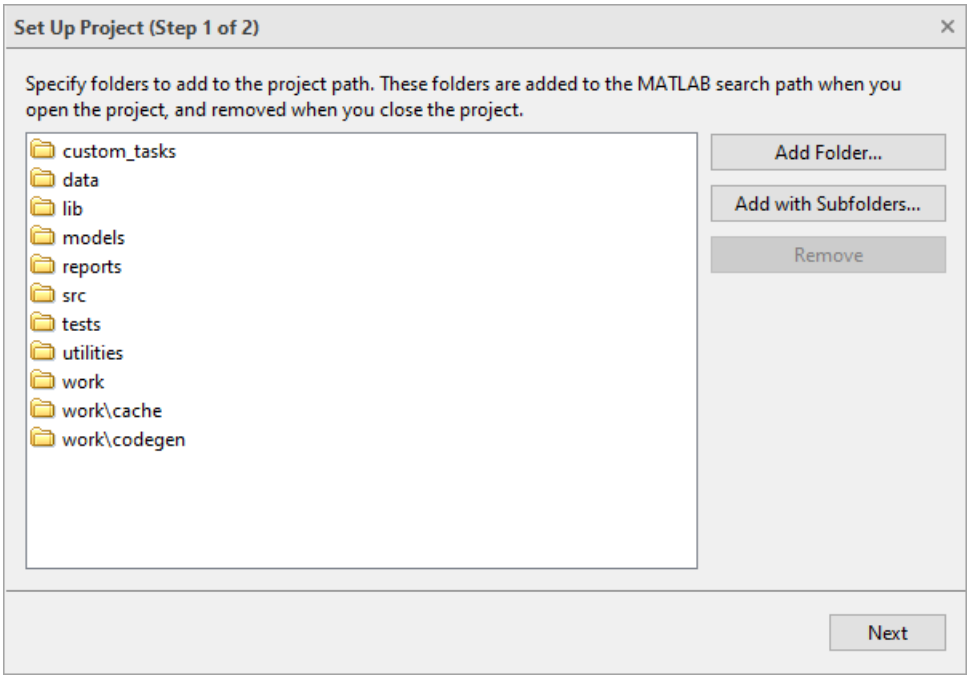
Using the Blockset Designer, you can organize blocks, and add tests and documentation to your blocks and blockset. To import your blockset and create a new Blockset Designer project:

1. Open the Simulink Start Page. Under **Simulink** select Blockset Designer, and click **Create Project**
2. In the New Project dialog box, enter a project name, browse to select the folder containing your files, and click **OK**.

3 In the Welcome to your project dialog box, click **Set Up Project** to continue.



4 In the Set Up Project (Step 1 of 3) dialog box, choose folders to add to the project path. When you open the project, these folders are added to your MATLAB search path, and removed when you close the project. Add project folders and subfolders to the project path to ensure that you have access to `slblocks.m` and related libraries. `slblocks.m` contains all information about the top library in your project. To learn more, see “Add Libraries to the Library Browser” on page 41-7. To add all project folders, select **Add with Subfolders** and then the project folder containing all your subfolders. Click **Next**.

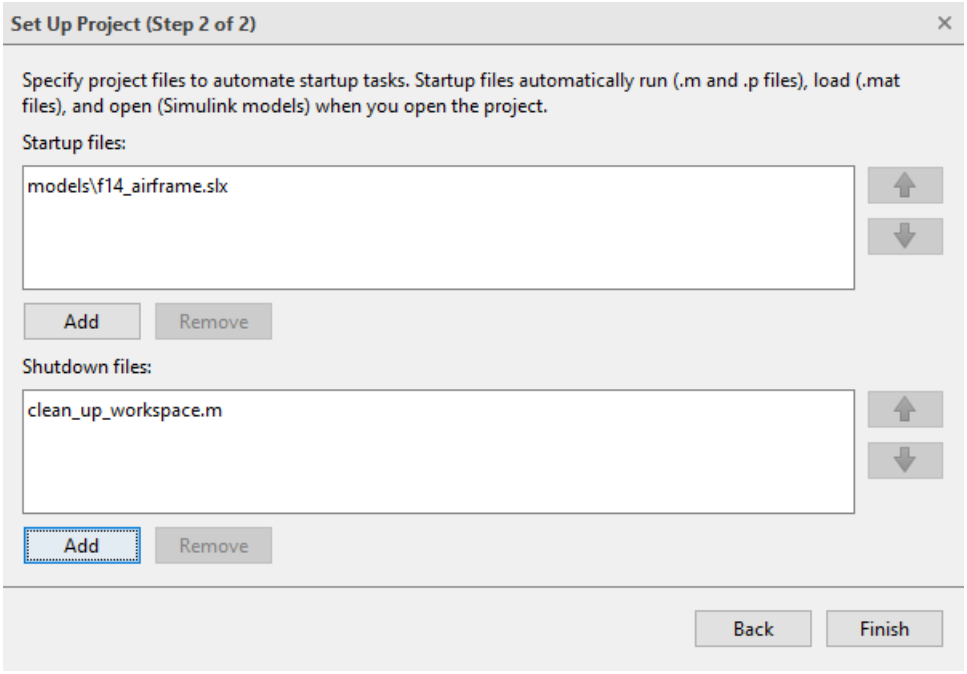


5 In the Set Up Project (Step 2 of 3) dialog box, optionally specify startup and shutdown files.

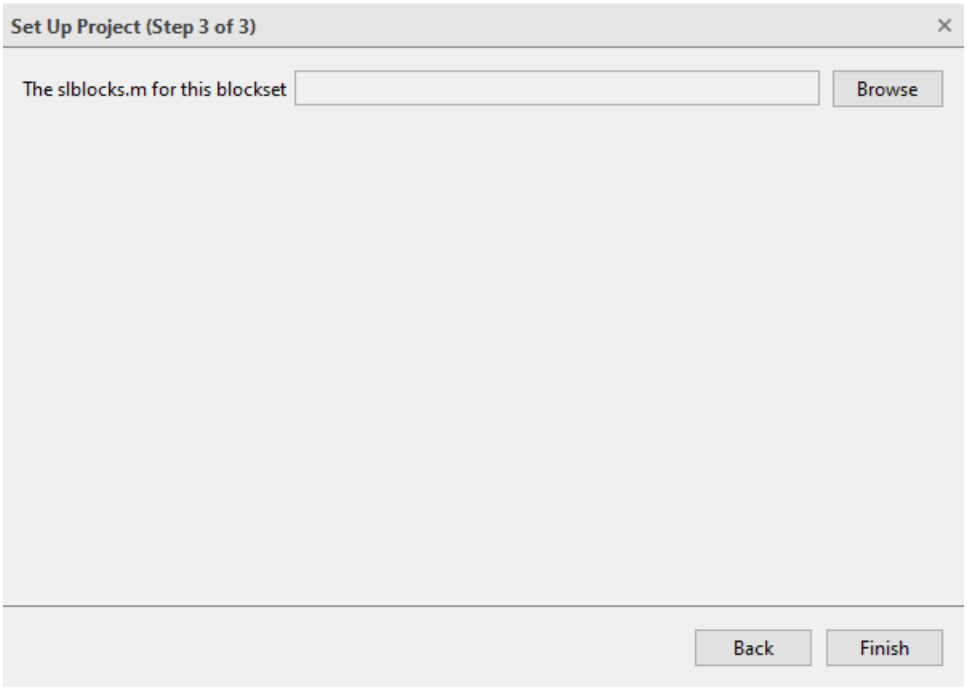
- Use startup files to configure settings when you open the project. Startup files automatically run (`.m` and `.p` files), load (.mat files), or open (Simulink models) when you open the project.

- Use shutdown files to specify MATLAB code to run as the project shuts down. You do not need to use shutdown files to close models when you close a project, because it automatically closes any project models that are open, unless they are dirty. The project prompts you to save or discard changes.

Click **Add** to specify startup or shutdown files.



6 In the Set Up Project (Step 3 of 3) dialog box, specify a `slblocks.m` library information file for this blockset. This file contains information on your blockset settings, and the location of your top library. Without this file, your project cannot be set up properly. Click **Browse** to select the file.



7 Click **Finish** and a new project opens. The Blockset Designer automatically adds all your files to the project. During the import process, based on the top library, the Blockset Designer explores the hierarchy of the blockset, creates dedicated folders for each type of supported blocks, and adds the folders to project and search corresponding artifacts such as S-function MEX file, source files, and System object files for MATLAB System block automatically. See “Blockset Project File Structure” on page 46-12 for more information.

Now you can start adding more blocks to your project, or add test or documentation to your existing blocks in the project. To learn more about how to build, test and document your blocks, see “Create and Organize Block Artifacts” on page 46-14.

Blockset Project File Structure

When you create a blockset project, either by creating a new blockset or importing an existing blockset, the Blockset Designer organizes your project in a certain file organization. This organization changes with the different type of blocks you have in your project. Use this table as a guide to determine which folders are created for each block. To see your blockset folder structure, click **Files** in the **Views** panel.

Block Type	Folder Name	Folder Contains
Subsystem	**doc**	Documentation source and html file for the documentation
	library	Library model for the block
	unittest	Unit test model, test suite, and input file for the test
MATLAB System	**library**	Library model for the block

Block Type	Folder Name	Folder Contains
	sysobj	The System object code for the MATLAB System block
	unittest	Unit test model, test suite, and input file for the test
S-function and S-function Builder	**build**	S-function build script
	doc	Documentation source and HTML file for the documentation
	library	Library model for the block
	mex	S-function MEX and code generation files
	src	S-function source files
	unittest	Unit test model, test suite, and input file for the test
Blockset Project (common)	**doc**	Documentation source and HTML file for the documentation
	library	Library model for the block
	script	`blocksetroot.m` file to return the blockset root folder

See Also

MATLAB System | S-Function | S-Function Builder | Subsystem, Atomic Subsystem, CodeReuse Subsystem

More About

- “Create and Organize Block Artifacts” on page 46-14
- “Publish the Created Blockset” on page 46-18

Create and Organize Block Artifacts

After creating a new blockset or importing your existing blocksets, you can start adding artifacts. In Blockset Designer, there are different artifacts associated with different blocks:

- Subsystem blocks — You can add test and documentation.
- MATLAB System blocks — You can add test using the Blockset Designer working area. To document your blocks, add comments in the System object code.
- S-function and S-function Builder Blocks — You can build S-functions, add test and documentation.

Follow the status of your artifacts by checking the icons in the working area.

Icon	Artifact Status
	No files are specified for the artifact. To resolve, click **Create** or select an artifact from **Browse**.
	The artifact files have been updated but has not been run yet. To resolve, click **Run** in the toolstrip for the desired artifact.
	No action needed for the shown status of this block. For example, you do not need to build a MATLAB System or a Subsystem block, and on the status table, **Build** column shows this icon.
	Build operation of the selected S-function block encountered an error. To resolve, check the S-function build report from the top right of the **Build** working area, or check your S-function code to resolve issues. See “S-Function Related Artifacts” on page 46-15 for more details.

Add Tests to Blocks

To ensure that your custom blocks run seamlessly, you can add a test to your blocks. Blockset Designer gives you the capability to create a test harness automatically, or using you can bring in your own test suite.

To create a test model and a test suite automatically using Blockset Designer, click **Create**. Your test model opens automatically, investigate this model and verify that it matches your testing needs. Note

that you can only create one test using Blockset Designer, but you can add many tests using test suite. To write your test, open the MATLAB unit test file. This file is just a template for you create your test. If you are going to use the test harness the Blockset Designer created for you, make sure to uncomment the model file.

To bring in your own tests, browse and select your test model and click **Apply**. To bring your own test suite, browse to your test suite and click **Apply**.

Check Blocks Using Model Advisor

Use **Model Advisor** functionalities to do further checks on your blocks. To use **Model Advisor**, you must have a test harness model for your block. Access it from **Check**, select **Model Advisor**.You can create a test model automatically using the steps above, or you can browse and select your own harness model.

To learn more about Model Advisor checks, see “Check Your Model Using the Model Advisor” on page 5-2.

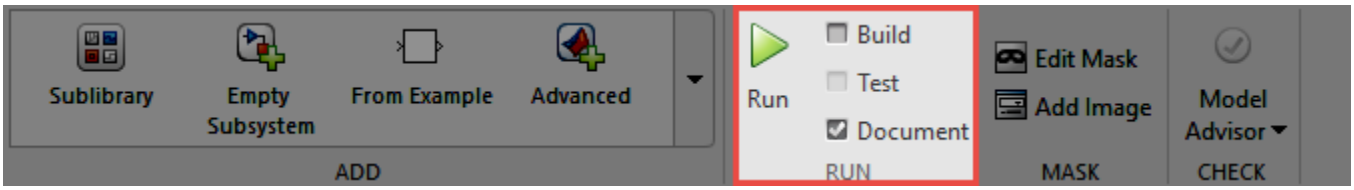
Run S-Function Checks

Use S-function Checks to analyze the quality of your S-functions. To use S-function checks, you must have a test harness model for your block. To run these checks, from **Check**, select **Run S-Function Checks** in Simulink toolstrip. The checks generate an S-function check report for your S-function. To learn more about S-function checks, see “Check S-Functions Using S-Function Analyzer APIs”.

Document the Blocks

The Blockset Designer enables creating documentation for your custom blocks. To create documentation automatically, click **Create** in the **Document** part of the working area.

This creates a Live Editor file for you to write your documentation. After you complete your block documentation, save the Live Editor file. To produce an HTML file for your documentation, from the Simulink toolstrip, confirm that the Document checkbox is selected, and click **Run**.



S-Function Related Artifacts

In the steps above, you already learned how to create a new S-function block. Here you learn how to build an S-function in a Blockset Designer project.

Build S-Function Blocks

If you create your S-function as a new block:

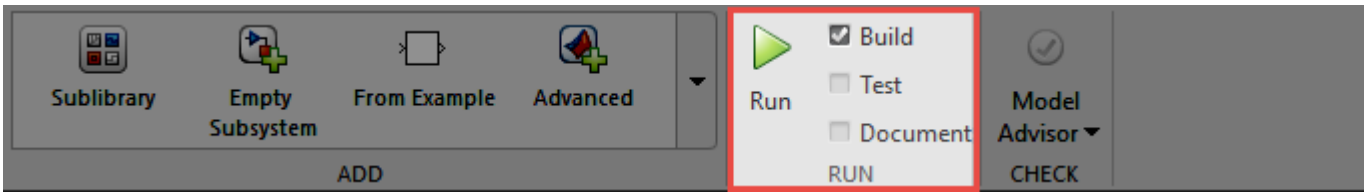
1. Check the working area and observe that **S-Function File** and **S-Function Build Script** fields are already populated for your S-function.

2. To open the S-function code template, click **Open** next to the **S-Function File**. Write the code for your S-function and save your code. Close the MATLAB Editor.

   To learn more about writing S-functions, see “Implement C/C++ S-Functions”. If you prefer to create a S-function automatically, see S-Function Builder.

3. Return to the Blockset Designer working area.

4. From the Blockset Designer toolstrip, confirm the **Build** checkbox is selected and click **Run**  .



5. During the run, your S-function files are added to the project. A build report for your S-function that contains your S-function build status is generated. If you face any issues during build, check this report. You can reopen this report by clicking on the timestamp in the top right corner of **Build** section.

6. Return to the working area and observe that after build operation, an S-function MEX File is generated for this S-function block.

Build S-Function Builder Blocks

1. After creating an S-function Builder block, the S-Function Builder dialog window automatically opens from the library model. Write your code in the panes of the S-function Builder dialog window.

2. Click **Build** on the top-right of the S-function Builder dialog window. This will build your S-function, create all S-function artifacts, and communicate with the Blockset Designer.

3. Click **Close** to return to `Blockset Designer` working area.

When you are finished adding and building Blockset Designer artifacts, you are ready to publish your project as a toolbox and share. For more information, see “Publish the Created Blockset” on page 46-18.

See Also

MATLAB System | S-Function | S-Function Builder | Subsystem, Atomic Subsystem, CodeReuse Subsystem

More About

- “Create a Blockset Project” on page 46-2
- “Publish the Created Blockset” on page 46-18

Publish the Created Blockset

After you have added artifacts for your blocks in a Blockset Designer project, you can publish and share your project as a toolbox. To publish:

1. Click the blockset from the blockset tree, and click **Publish Toolbox** from the Simulink toolstrip.

   The Blockset Designer collects your files in the project, and creates the **publish** folder in **Views > Files**. This folder contains all your files from the separate folders in the blockset project based on the dependency analysis, but now collected together under one folder. See “What Is Dependency Analysis?” on page 18-2 to learn more about dependency analysis. The publish folder contains these folders and the associated content:

Folder Name	Contains
doc	• All block documentation HTML files and blockset-level documentation XML files • `helpdoc.xml` and `info.xml` is generated to be used in toolbox documentation. See“Display Custom Documentation” for more information.
extra	For all blocks: • S-function source files and build scripts • Documentation source files • Test models, suites, and their generated input files • All other necessary files based on the dependency analysis, such as block icons.
library	All library models in the blockset all files under <projectroot>/common/library. This includes library models of sublibraries, as well as the blockset project library model.
mex	All S-function mex files
script	`blocksetroot.m` file to return blockset root folder and all files under <projectroot>/common/script.
sysobj	All System object code for the MATLAB System blocks in the blockset project

   If you have any other files that you would like to include in the publish, manually copy them to the publish folder.

   Since the publish folder is added to project path by default, to avoid shadowing files in project, remove this folder from the project path or delete it after the publish process. If you make any changes to your blockset after publishing, click **Publish** again the move the updates files to the publish folder.

2. The **Toolbox Information** fields are populated with the project name, author, and description. Edit the information if needed.

3. To ensure that MATLAB detects installation components, review the toolbox contents from **Toolbox Files and Folders**. If you want to include files not already included in the project files, edit the **Exclude files and folders**.
4. Click **Package** from the toolstrip to package your toolstrip.

   To save your toolbox and share it on the MATLAB Central File Exchange, select **Package and Share** from the **Package** menu at the top of the Package a Toolbox dialog box. This option generates a `.mltbx` file in your current MATLAB folder and opens a web page for your toolbox submission to the File Exchange. MATLAB populates the File Exchange submission form with information about the toolbox. Review and submit the form to share your toolbox on File Exchange.

   To share your toolbox with others, use the `.mltbx` file. All files you added when you packaged the toolbox are included in the `.mltbx` file. When your toolbox is installed, `.mltbx` file manages the MATLAB path or other installation details.

To learn more about how to create and share toolboxes, and to see the details of the packaging user interface, see “Create and Share Toolboxes”.

See Also

MATLAB System | S-Function | S-Function Builder | Subsystem, Atomic Subsystem, CodeReuse Subsystem

More About

- “Create and Share Toolboxes”
- “Create and Organize Block Artifacts” on page 46-14

FMUs and Co-Simulation in Simulink

Import FMUs

Use the FMU block to import Functional Mockup Units (FMUs) into Simulink.

The FMU block automatically selects the FMU mode based on the existing FMU you want to import:

- `Co-Simulation` — Integrate FMUs that implement an FMI Co-Simulation interface. These FMUs can contain local solvers used for tool coupling.
- `Model Exchange` — Integrate FMUs that implement an FMI model exchange interface. These FMUs do not contain local solvers. Instead, these FMUs inherit solvers from Simulink.

This block supports FMI versions 1.0 and 2.0. For FMI version 2.0, if your FMU contains both Co-Simulation and Model Exchange elements, the block detects this state and prompts you to select the operation mode for the block.

You can use your FMU block as you do other Simulink blocks. The FMU block supports normal, rapid accelerator, and accelerator modes. Rapid accelerator mode is unsupported for FMU blocks with FMU logging.

This topic assumes that you provide a `.fmu` file.

FMU XML File Directives

The default parameter values derive from the corresponding parameter `start` value defined in the FMU `ModelDescription.xml` file. A block parameter value overwrites the initial value of the corresponding parameter defined in the FMU binary implementation.

Simulink interprets these FMU tags accordingly.

FMU Tag	Simulink
`ScalarVariable` has attributes set as follows: • `causality="none"` or `causality="internal"` • `variability="parameter"` • `start` value is defined	Interprets `ScalarVariable` element as block parameter
`Real`	Interprets block parameter as edit field
`Integer`	Interprets block parameter as edit field
`Boolean`	Interprets block parameter as check box
`Enumeration`	Interprets block parameter as drop-down list
`String`	Interprets as UTF-8 encoded string

The FMU block supports the following encoding formats for the model description XML file:

- ISO-8859–1
- UTF-8
- UTF-16

Additional Support and Limitations

Capability	FMI Version 2.0 Support	FMI Version 1.0 Support
Save SimState to base workspace	✓	
Fast restart	✓	
Simulation Stepper	✓	
Solver Jacobian	✓	
Linearize models	✓	
Declare parameter as tunable and tune it during simulation	✓	
For Each subsystem blocks	✓	
Parameters of type string	✓	✓
Rapid accelerator mode	✓	✓
Software-in-the-loop (SIL) and processor-in-the-loop (PIL) modes		
Code generation		Supports code generation target `slrealtime.tlc` in Co-Simulation mode. Does not support FMU blocks for Model Exchange mode. For more information, see “Apply Functional Mockup Units by Using Simulink Real-Time” (Simulink Real-Time).
Model coverage		
Simulink Design Verifier		
Model reference in accelerator mode	✓	✓

Simulink supports stepping back and forth, tuning parameters in between, and saving states for the FMU Import block as long as the FMU itself supports these features. FMU flags that it supports these features by setting `canGetAndSetFMUstate` and `canSerializeFMUstate` fields in its model description XML to `true`.

FMU Import Examples

For examples of importing FMUs into and System objects, see **Integrating FMUs for Simulation** in Simulink Examples:

Example	Description
“Importing a Co-Simulation FMU into Simulink”	This model shows how to use the FMU block to load an FMU file that supports Co-Simulation mode.
“Importing a Model Exchange FMU into Simulink”	This model shows how to use the FMU block to load an FMU file that supports Model Exchange mode.
“Using Bus Signals and Structure Parameters in the FMU Import Block”	This model shows how to use bus signals and structure parameters in an FMU block that supports Model Exchange mode.

See Also

FMU

More About

- “Implement an FMU Block” on page 47-5

External Websites

- FMI Standard

Implement an FMU Block

Implement a block and assign a functional mockup unit (FMU) to it. You can then explore the block to see the FMU. This example uses the FMU block with the vehicle FMU.

1. Create a model and add the FMU block.
2. In the block dialog box, enter the path name for an FMU file in the **FMU name** parameter and click **OK** or **Apply**. The file extension `.fmu` is optional.

   The first time you click **OK** or **Apply**, the block identifies which FMU mode to set your FMU to, co-simulation or model exchange.

   The block also creates a `slprj/_fmu/`*`fmu_name`* folder and unpacks the contents of the FMU file into this folder, which optionally include:

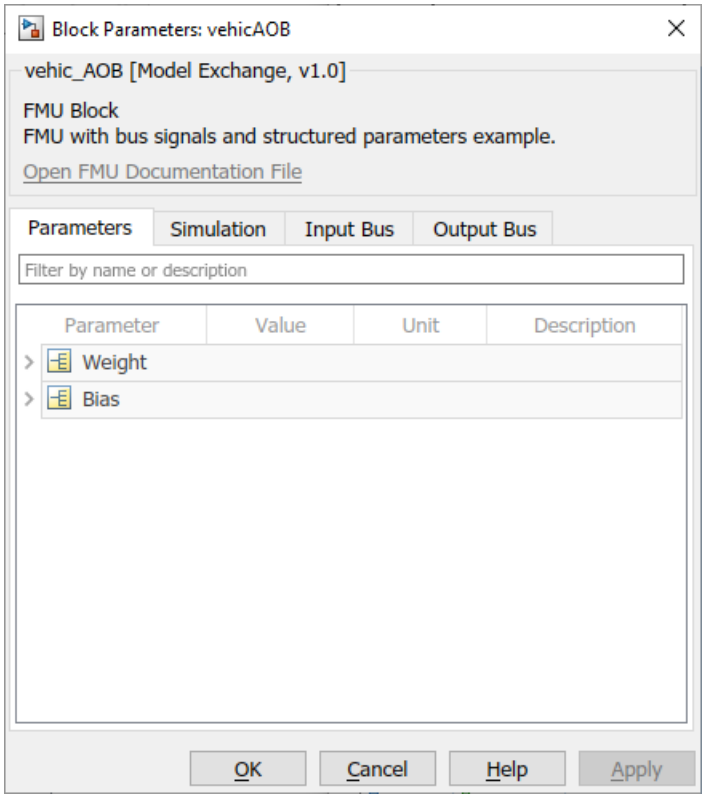
   - `binaries` — FMU binary files
   - `documentation` — FMU documentation HTML files
   - `resources` — FMU source files
   - `sources` — FMU source files
   - Other supporting files, such as block mask and description files

The FMU block icon and port labels update to the labels of the corresponding FMU. After you associate the block with an FMU, if you want to change the FMU, right-click the FMU block, and select **Block Parameters**, and enter a new FMU name in **FMU name**. The section in this topic use the FMU from the “Using Bus Signals and Structure Parameters in the FMU Import Block” example.

Explore the FMU Block

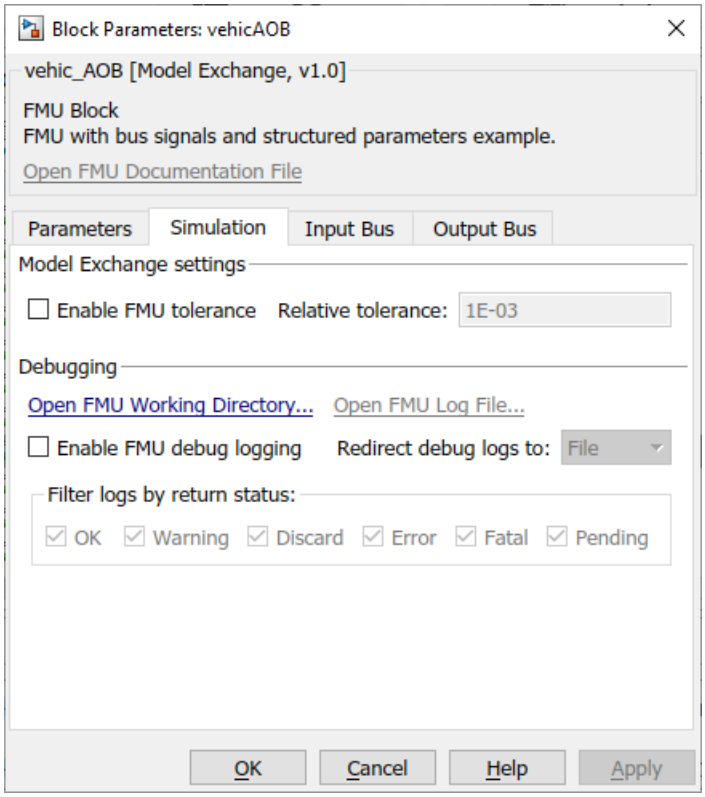
Double-click the block. Suppose that you entered an FMU named `fmuVehicAOB.fmu` from your current folder. The FMU block dialog box reflects the FMU parameters defined in the `fmuVehicAOB.fmu` file.

Parameters Tab



Lists the FMU block parameters. Edit the values as necessary. You can edit the elements of a structure parameter by expanding the tree view.

Simulation Tab

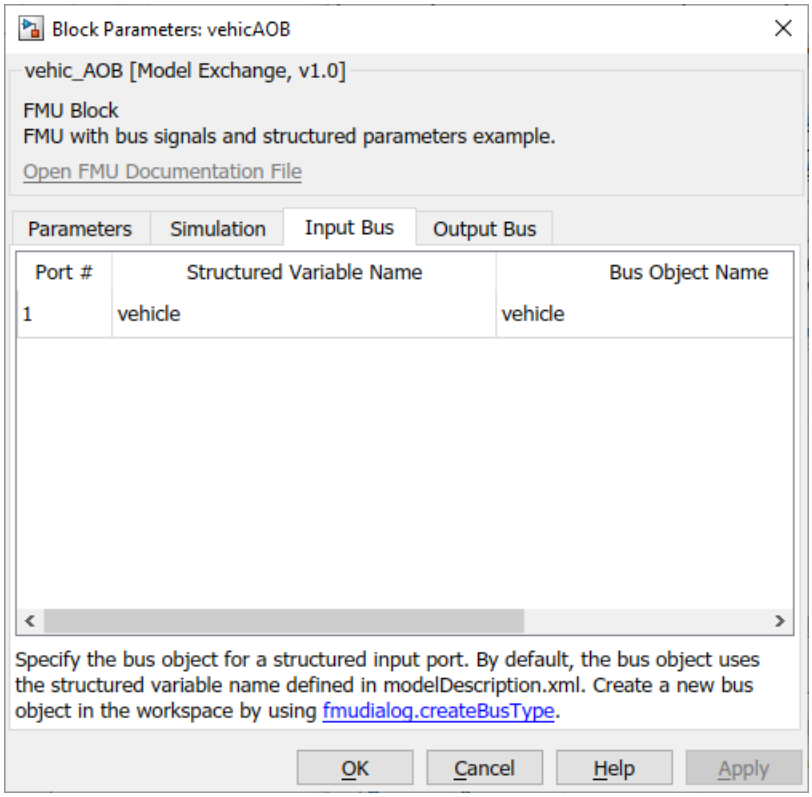


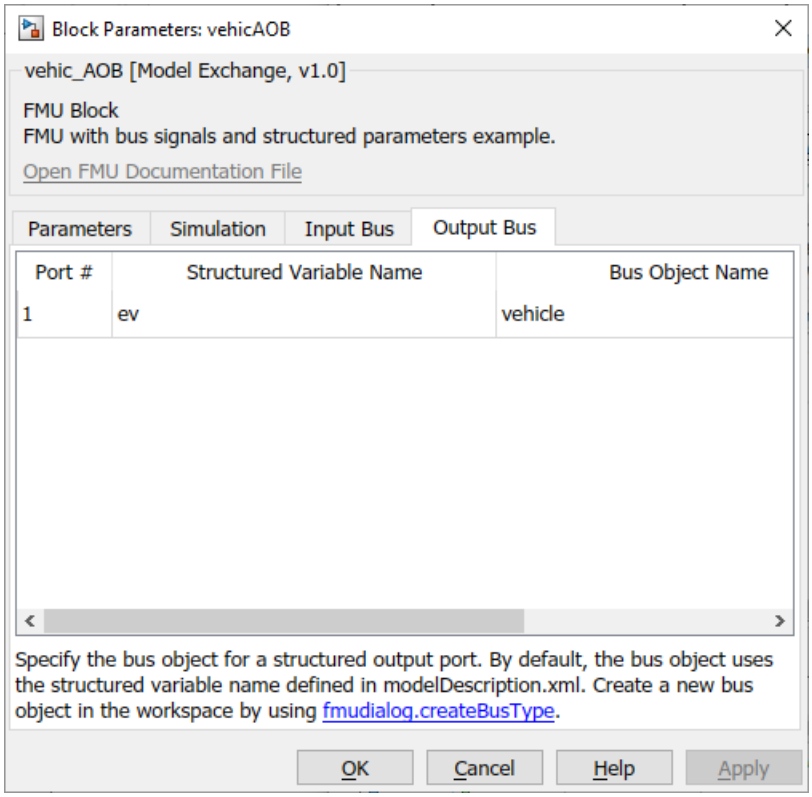
Enables logging and associated customizations.

- To enter a relative tolerance, select **Enable FMU tolerance** and set it.
- To determine the sample time of the block in the model, set **Communication step size**. To inherit the step size from the Simulink solver, set to `-1`. This option is available only if the FMU is a co-simulation FMU.
- To enable logging, select the **Enable FMU Debug Logging**.
- In **Redirect debug logs to**, select the destination for the logs.
  - `File`, saved to `slprj\_fmu\_logs_`*`modelname`*`\`*`modelname_blockname`*`.txt`
  - `Display`, displayed in the MATLAB Command Window.

  If the **Enable FMU Debug Logging** check box is selected and the **Redirect debug logs to** parameter is set to `Display`, you cannot use the FMU block for co-simulation. For more information on co-simulation and multiple cores, see “Run Co-Simulation Components on Multiple Cores” on page 47-24
- In the **Filter logs by return status**, select the check box for the return status you want.

Input and Output Bus Tabs





These two tabs list the input and output bus objects that the block defines.

In the **Bus Object Name** parameter, you can change the bus object names to match the bus objects defined in the workspace.

To create a bus object in the workspace:

```
fmudialog.createBusType(gcb)
```

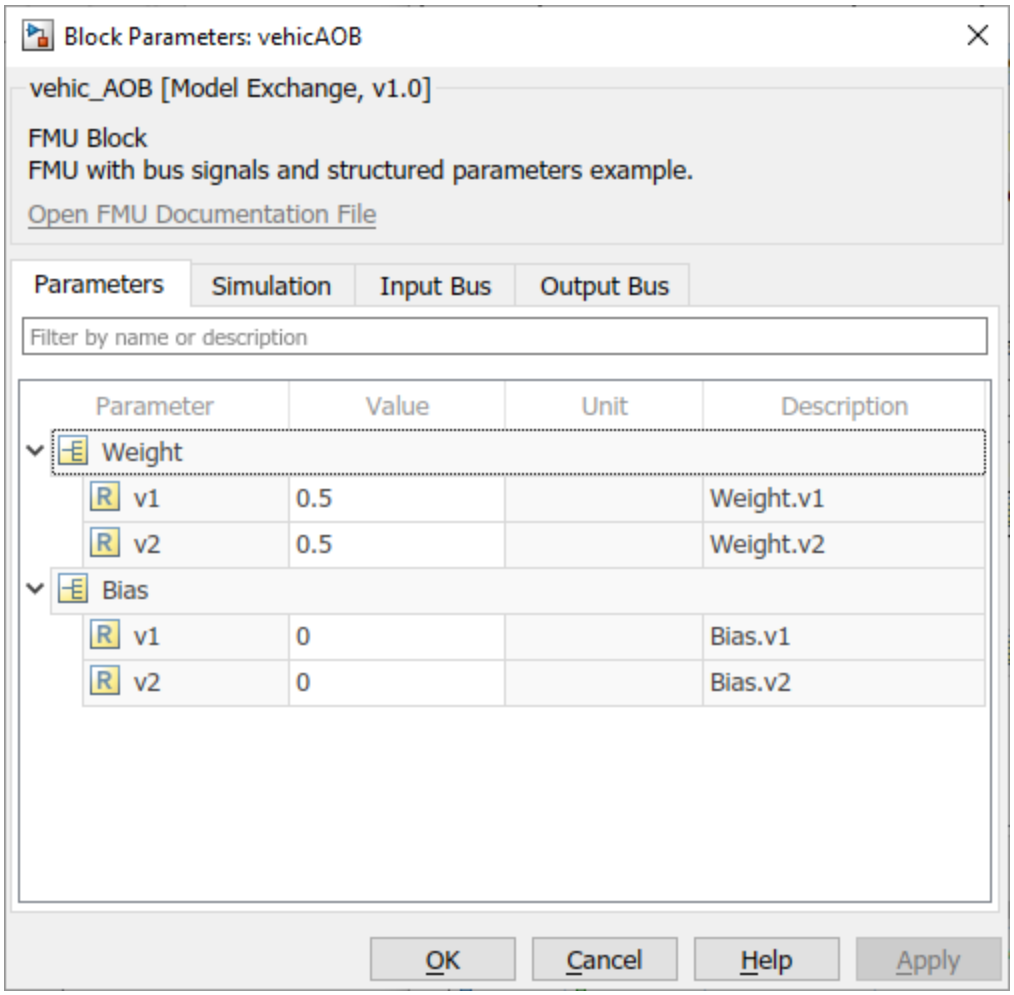
Change Block Input, Output, and Parameter Structures

You can change the layout of FMU block input ports, output ports, and parameters with these parameters:

Parameter	Action	Settings
`FMUInputMapping`	Change hierarchy of input ports.	`'Flat'` — Separates input into individual signals. `'Structured'` — Combines input into a structure of signals (bus).

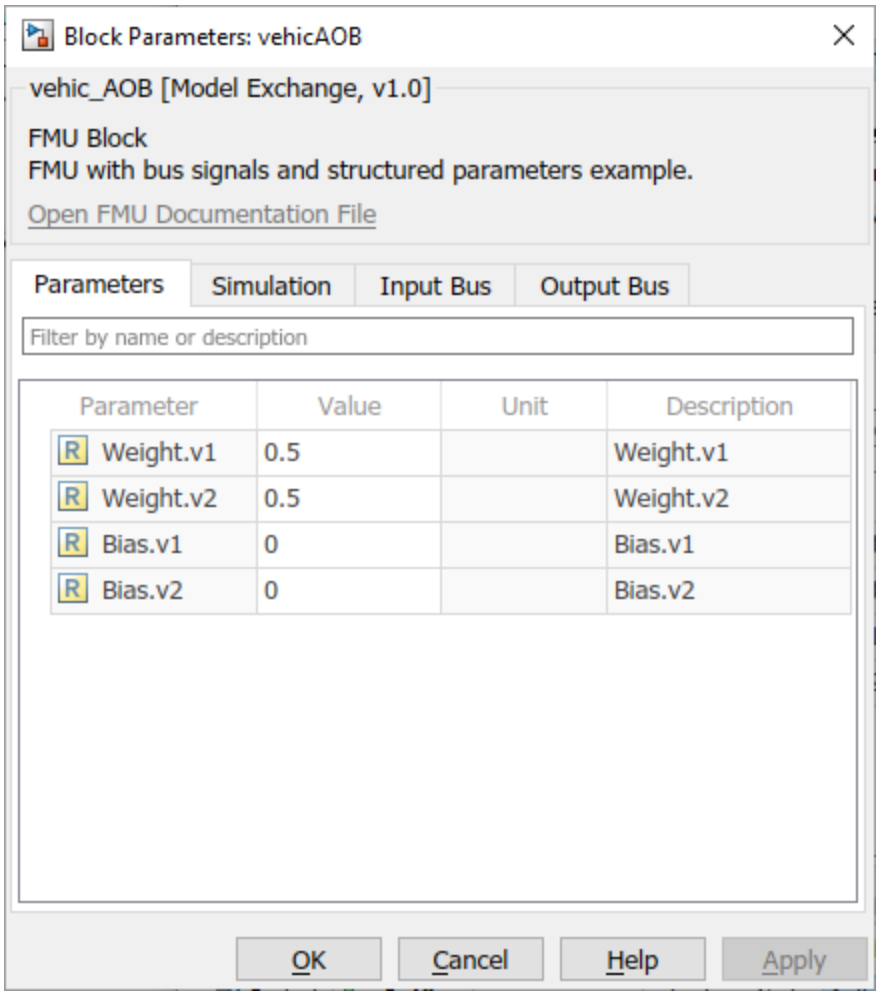
Parameter	Action	Settings
`FMUOutputMapping`	Change hierarchy of output ports.	`'Flat'` — Separates output into individual signals. `'Structured'` — Combines output into a structure of signals (bus).
`FMUParamMapping`	Change hierarchy of parameters.	`'Flat'` — Separates parameters into individual parameters, listed by the parameter name and value. `'Structured'` — Combines parameters into a structure of parameter values (`struct`).

Use the `get_param` and `set_param` functions to set these values. For example, assume a block parameter tab with a structure construct:



The parameters are contained in a `struct`. To list the parameters individually, set the `FMUParamMapping` property to `'Flat'`:

```
set_param(gcb,'FMUParamMapping', 'Flat')
```



Timing Considerations

You can set the sample time for the FMU block with the **Communication step size** parameter. This block sample time setting, t_C, like all Simulink blocks, must be an integer multiple of the model sample time, t_M. Simulink generates an error if the communication step size t_C is not a multiple of the model step size t_M.

The local step size of the FMU t_L, on the other hand, is part of the FMU specification and is known to the FMU only internally. For proper operation, the communication step size, t_C must also be an integer multiple of t_L. If the model sample time t_M or the block sample time t_C is incompatible with the FMU local step size t_L, the FMU may or may not produce an error at run time, depending on its implementation.

Troubleshooting FMUs

If there are problems with using the FMU:

- Check the compliance of the FMU with the FMI standard. Use the FMU compliance checker.
- Select the **Enable FMU Debug Logging** check box on the FMU block Simulation tab.
- Contact the FMU supplier.

See Also

FMU | `fmudialog.createBusType`

More About

- “Import FMUs” on page 47-2
- “Co-Simulation Execution and Numerical Compensation” on page 47-17
- “Run Co-Simulation Components on Multiple Cores” on page 47-24

External Websites

- FMI Standard

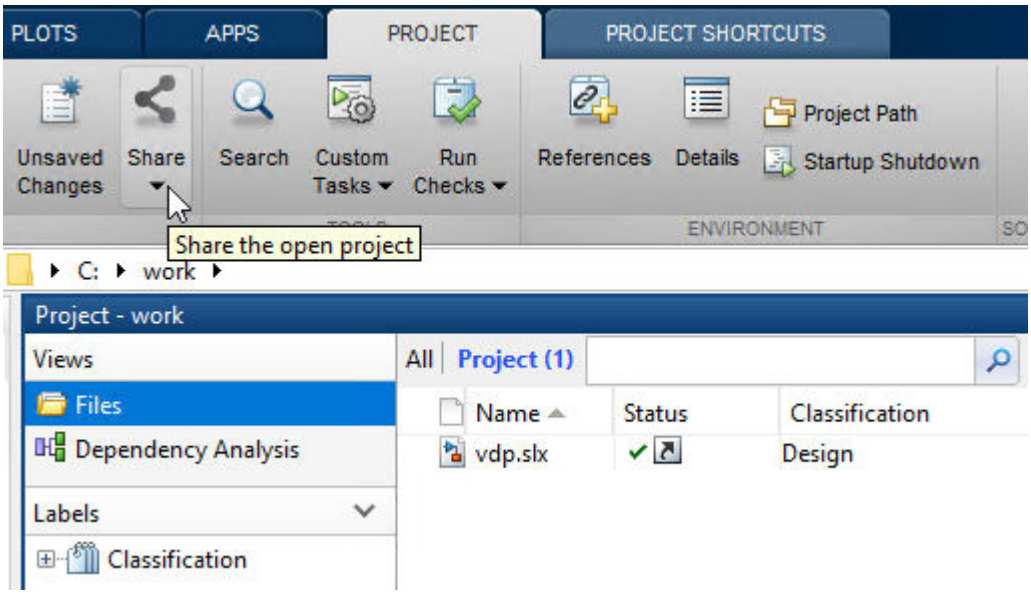
Export a Model as a Tool-Coupling FMU

To integrate Simulink components into third-party software, export a Simulink model as a tool-coupling functional mockup unit (FMU). When a third-party tool runs the FMU, it checks out required licenses and starts a local installation of Simulink to start the model. Tool-coupling FMUs support fixed-step and variable-step solvers.

Use a project to export an FMU. Open the model and select **New > Project > New Project from this Model** to create a project from a model.

You can export a FMU from a project interactively.

1 In the project, select **Share > Tool-Coupling FMU**.



2 Type in the **Copyright**, **Description**, and **FMU Icon** fields. Click **Save Settings and Export as** and provide a name.

The generated FMU includes model implementation, as well as the metadata provided during export.

```
<?xml version="1.0" encoding="utf-8"?>
<fmiModelDescription author="" copyright="" description="" fmiVersion="2.0"
                     generationDateAndTime="2018-08-16T15:51:48Z"
                     generationTool="Simulink (R2018b)"
                     guid="5bd096be-a08d-020e-fc96-847aa21def5b"
                     license=""
                     modelName="vdpSlave"
                     variableNamingConvention="structured"
                     version="1.8">
```

You can also create and export a project to a FMU from the command line as follows:

```
>> p = slproject.create('vdpProject')
>> copyfile(which('vdp'), './vdpSlave.slx')
>> p.addFile('./vdpSlave.slx')
>> Simulink.fmuexport.ExportSimulinkProjectToFMU(p,'vdpSlave.slx','-fmuname','vdpFMU')
```

For more FMU export options from the command line, type:

```
help Simulink.fmuexport.ExportSimulinkProjectToFMU
```

The model must satisfy these conditions for exporting:

- Model must be in Normal or Accelerator simulation mode.
- Root input and output ports must be of numerical data type.

If the co-simulation component is an FMU exported from Simulink. the local sample time for that FMU is the sample time of the original model.

Include Tunable Parameters for Tool-Coupling FMU

To include tunable parameters:

1. Open the model from the associated Simulink project.
2. From the Simulink model, click the **Modeling** tab and start Model Explorer.
3. Select Model Workspace and add a MATLAB variable or Simulink parameter.
4. For each parameter you add and want tunable, in the **Data Properties** or **Simulink.Parameter** pane, select the **Argument** check box.
5. Click **Apply**.
6. Reference the tunable parameters in the model.
7. Export the tool-coupling FMU.

Use the Exported Tool-Coupling FMU

The exported FMU requires a local installation of Simulink to run. The MATLAB version used for co-simulation must be the same as the MATLAB version where the FMU is exported. On Windows, the application that runs the FMU can check out the required licenses automatically. For other operating systems, apply these settings:

- On Linux:

  ```
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:<InstallationFolder>/bin/glnxa64:<InstallationFolder
  export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<InstallationFolder>/bin/glnxa64:<InstallationFolder
  ```

- On Mac OS:

  ```
  setenv DYLD_LIBRARY_PATH ${DYLD_LIBRARY_PATH}:<InstallationFolder>/bin/maci64:<InstallationFol
  export DYLD_LIBRARY_PATH=${DYLD_LIBRARY_PATH}:<InstallationFolder>/bin/maci64:<InstallationFol
  ```

For Mac OS, due to System Integrity Protection (SIP), `setenv` command does not work for applications that starts new processes, such as MATLAB. Follow Append library path to "DYLD_LIBRARY_PATH" in MAC to set `DYLD_LIBRARY_PATH` to `<InstallationFolder>/bin/maci64:<InstallationFolder>/extern/bin/maci64`.

Before you can run the FMU, you must set up a MATLAB session from your operating system console. After you set up this session, start the third-party application and import the tool-coupling FMU. Each FMU instance requires a new MATLAB session.

Start a Dedicated Session from MATLAB

If the application that imports the FMU runs a single instance of the FMU, you can use MATLAB to start a session.

```
>> shareMATLABForFMUCoSim
```

This dedicates the current MATLAB session available for requests from the external tool to co-simulate an imported FMU. When an FMU is connected this session, Simulink editor and Simulink project are loaded, and co-simulation starts automatically. You can use this session to pause, resume co-simulation, as well as tune parameters or plot signals from the command window while co-simulation is running. If co-simulation is finished, stopped by user, or interrupted by a runtime error, MATLAB closes, unloads Simulink editor and Simulink project, and discards changes to the model. If an error occurs, it displays in the simulation tool that imports this FMU. Each session can connect to one FMU instance at the same time.

Start a Dedicated Session from the Operating System

If the application that imports the FMU runs multiple FMU instances, you can use the operating system console to start dedicated MATLAB sessions.

- On Windows:

  ```
  <matlabroot>\toolbox\shared\fmu_share\script\fmu-matlab-setup.cmd
  ```

- On Linux and Mac OS:

  ```
  <matlabroot>/toolbox/shared/fmu_share/script/fmu-matlab-setup
  ```

Run `matlabroot` in MATLAB to find out <*matlabroot*>. The setup program starts and waits for a command. See available commands by typing `help`:

```
> help
Command list:
quit - Close all shared MATLABs and exit.
list - List shared MATLABs.
start NUMBER_OF_MATLABS - Start NUMBER_OF_MATLABS more MATLABs.
stop NUMBER_OF_MATLABS - Stop NUMBER_OF_MATLABS MATLABs.
ignore - Stop asking about the hardware core count when launching MATLABs.
clean MATLAB_NUMBER - Clean up the MATLAB workspace for MATLAB #MATLAB_NUMBER. Use 0 for all MATI
help - Print the command list.
```

Launch one session for each FMU to run concurrently. If there is a single FMU, type:

```
> start 1
```

If there are three FMUs running concurrently, type:

```
> start 3
```

Only one MATLAB management tool should be running on a single machine.

See Also

More About

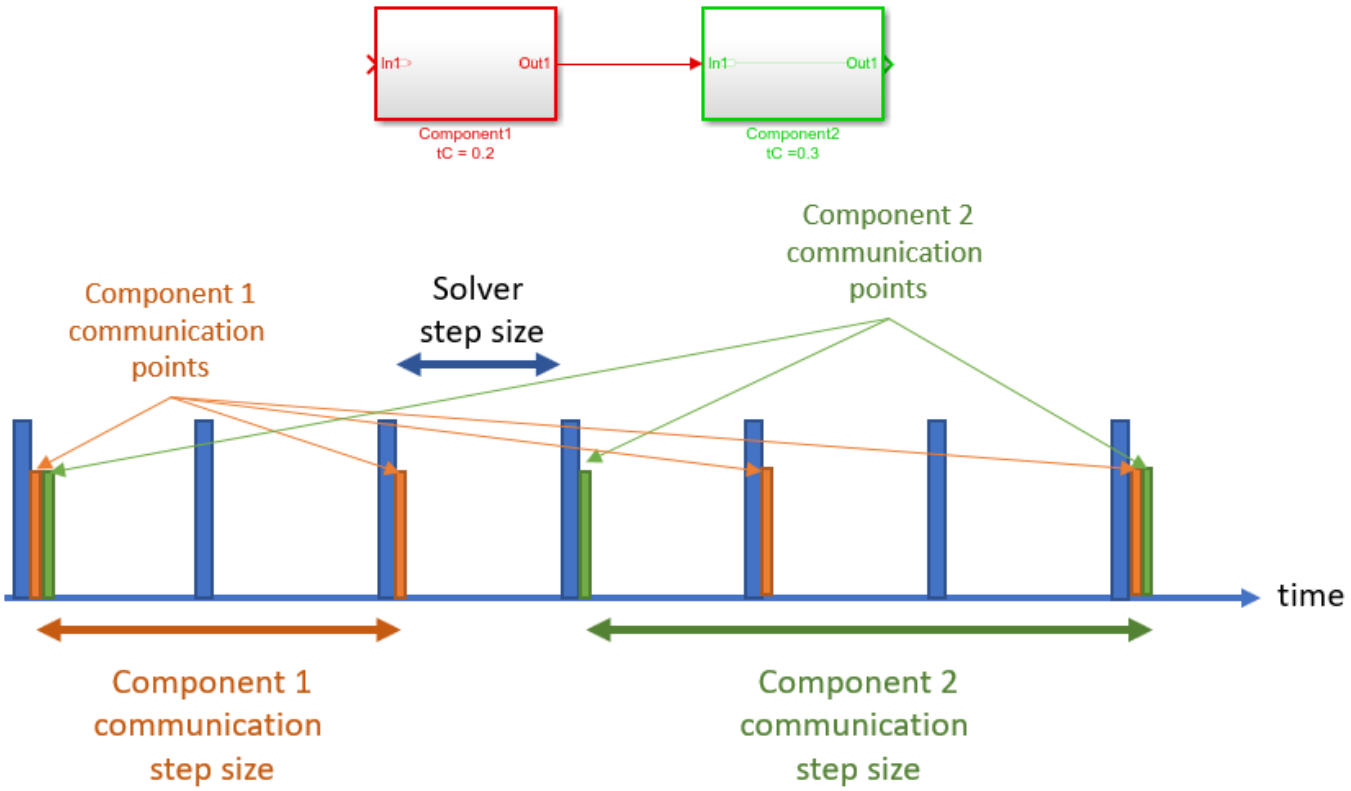
- “Run Co-Simulation Components on Multiple Cores” on page 47-24
- “Identify co-simulation signals for numerical compensation”

Co-Simulation Execution and Numerical Compensation

Simulink supports co-simulation between components with local solvers or involves external simulation tools. For example, co-simulation can involve an S-function implemented as a co-simulation gateway between Simulink and third-party tools or custom code. A co-simulation component can be a Functional Mockup Unit (FMU) in co-simulation mode imported to Simulink.

Execution Timing

In Simulink simulation, the solver step size must be an integer divisor of each periodic, discrete block sample time. In other words, if a co-simulation component defines its own sample time, Simulink must communicate with the component at those time steps. Co-simulation can involve components whose time steps are determined internally and are not known to Simulink. The only information available to Simulink is the block sample time, either through the communication step size parameter of an FMU block, or the sample time definition in an S-function implementation. The block sample time determines the time steps Simulink must communicate with the co-simulation component. If the solver step size is not automatic, the communication step size must be an integer multiple of the solver step size.



If the co-simulation component internally uses a local solver, then this local solver should also be taken into account when determining the communication step size of the block. The step size of the local solver is not exposed to Simulink, and knowledge of the implementation is necessary to set the communication step size correctly. The behavior of co-simulation in case of a potential incompatibility also depends on this internal solver implementation.

Numerical Compensation

Co-simulation signals typically represent continuous physical quantities that are discretized due to co-simulation. Data exchanges between co-simulation components such as C MEX S-functions and Co-Simulation FMU blocks can introduce numerical inaccuracies from signal delays. Use numerical compensation to improve numerical behavior for simulation involving components that use their own solver. Model Advisor includes a check that detects co-simulation components and advises numerical compensation.

Numerical Compensation Prerequisites

Simulink automatically performs numerical compensation for co-simulation signals between co-simulation components. Simulink performs numerical compensation at the input of the destination block. A signal is automatically qualified for numerical compensation if its source port and destination port satisfy these conditions:

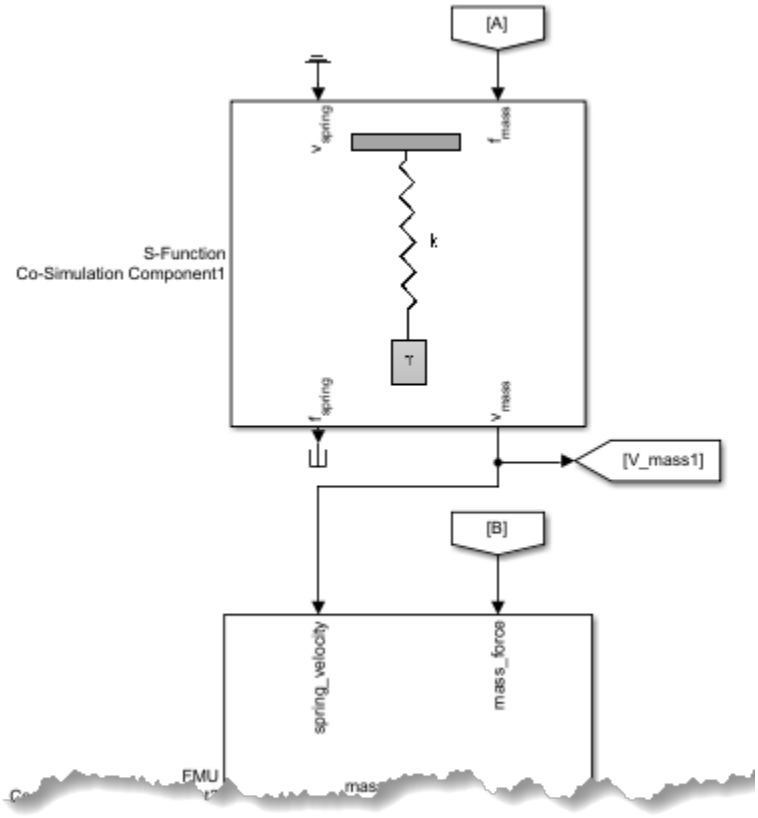
- The source port for the signal must satisfy these requirements:
  - S-Function
    - Output port data type is double
    - Output port sample time is periodic and discrete
    - Output port complexity is real
    - `ssSetOutputPortIsContinuousQuantity()` is set to `true` for the port
  - FMU
    - Output port data type is double
    - FMU is in co-simulation mode
    - Block sample time is periodic and discrete
    - Output port maps to a variable with `variability='continuous'` in `modelDescription.xml`
- The destination port for the signal must satisfy these requirements:
  - S-Function
    - Input port data type is double
    - Input port sample time is periodic and discrete
    - Input port complexity is real
    - `ssSetInputPortIsContinuousQuantity()` is set to `true` for this port
    - `ssSetInputPortDirectFeedThrough()` is set to `false` for this port
  - FMU
    - Input port data type is double
    - FMU is in co-simulation mode
    - Block sample time is periodic and discrete
    - Input port maps to a variable with `variability='continuous'` in `modelDescription.xml`

For an example of identifying co-simulation signals for numerical compensation, see the `slexCoSimTripleMassSpringExample` model.

Manage Numerical Compensation Options Using the Dialog

When Simulink detects the signals that can have numerical compensation, it marks the corresponding input ports with the ⚙ icon.

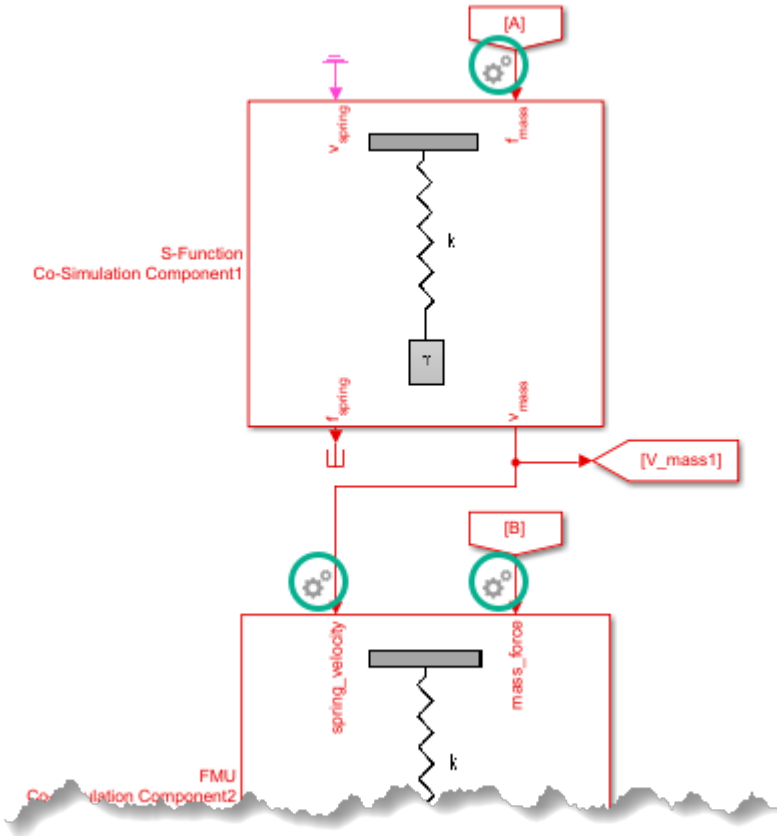
The following model includes co-simulation signals that can have numerical compensation:



1. Open the model.

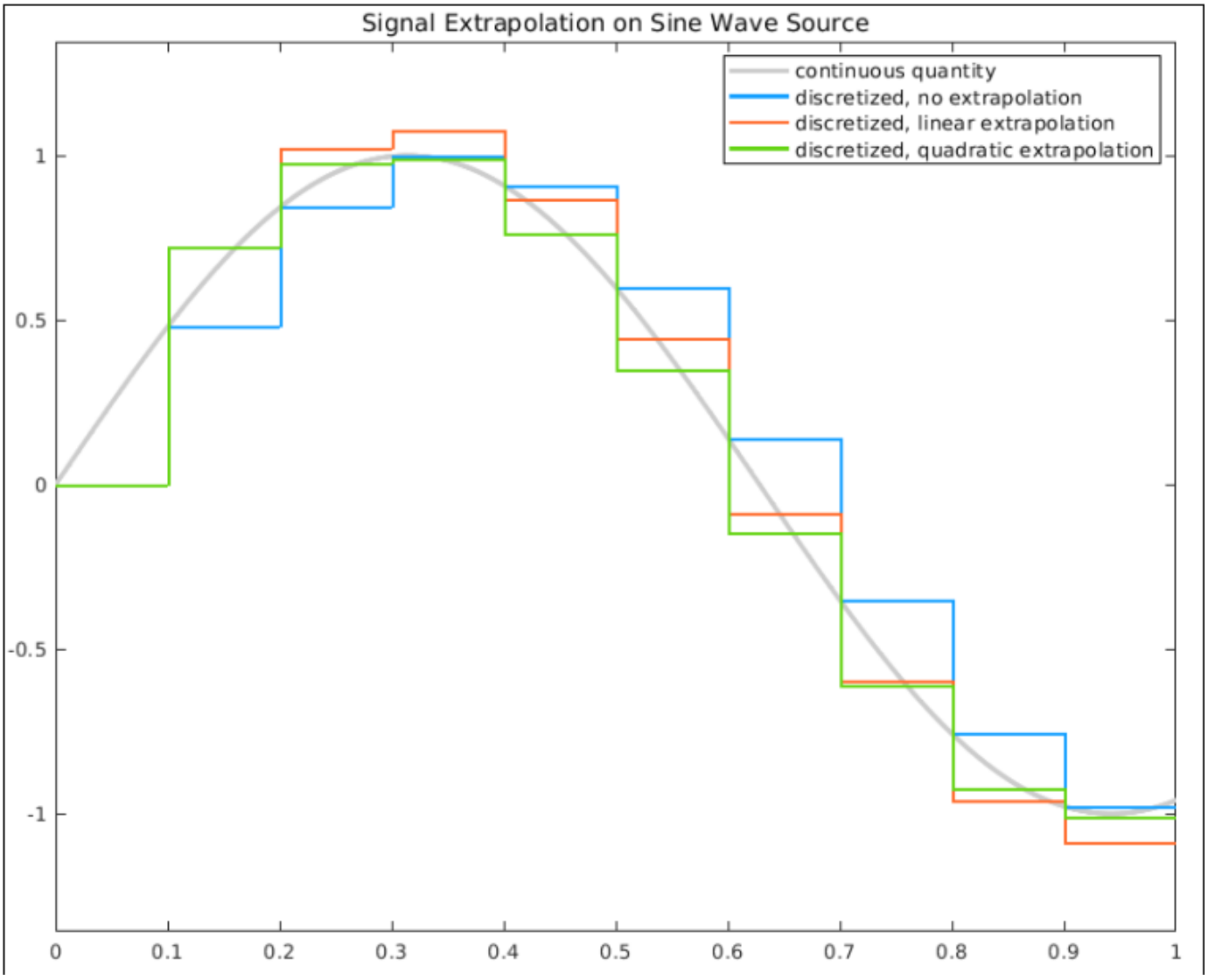
   `slexCoSimTripleMassSpringExample`

2. Update the diagram. Simulink detects the signals that can have numerical compensation, and marks the corresponding ports with the ⚙ icon.



3 Adjust the parameters for compensation accuracy: Right-click the icon and select **Configure Cosimulation Signal Compensation** and adjust the calculation parameters:

- **Extrapolation Method** — This method computes a compensated signal value for the current time step of simulation using extrapolation of simulation signal values generated from previous time steps. Three types of extrapolations are offered for selection.
  - `Linear` is the default, it uses signal values generated from previous two time steps to linearly estimate the signal value for use in the current time step of simulation.
  - `Quadratic` uses signal values of the previous three time steps to fit the data to a quadratic polynomial.
  - `Cubic` uses signal values of the previous four time steps to fit the data to a cubic polynomial.



In the beginning of the simulation, when there is an insufficient number of past signal values, a lower order extrapolation method is used automatically. Higher order extrapolation methods use more past signal values to predict the current signal value and can improve accuracy of the prediction. However, high order extrapolation methods can also be numerically unstable[1]. The best extrapolation method depends on the nature of the signal.

- **Signal correction coefficient** — This method further adjusts the extrapolated signal value based on past simulation results and past estimated signal values. A correction coefficient is offered for selection between `0` and `1`, where `0` means no adjustment to be made to the extrapolated signal value. The default setting for the correction coefficient is `1`. For a given extrapolated signal at a given time step, the larger the signal correction coefficient is, the more adjustment is made to the given extrapolated signal.

If numerical compensation is not beneficial, disable it by left-clicking the icon. When disabled, the icon appears with a red slash.

Manage Numerical Compensation Options from the Command Line

If automatic compensation is not possible, you can manually enable numerical compensation using the `CoSimSignalCompensationMode` property.

The `CoSimSignalCompensationMode` property has these values:

Icon	Setting	Behavior
	`'Auto'`	Enable automatic numerical compensation, allowing Simulink to detect if the port has a signal eligible for numerical compensation.
	`'Auto_Off'`	Disable automatic numerical compensation. Icon appears with a red slash.
	`'Always'`	Force the port to be considered numerical compensation compliant, even if the signal is not eligible for numerical compensation. This setting lets you add compensation without declaring it to be continuous.
	`'Always_Off'`	Disable forcing the port to be considered numerical compensation compliant.

For example, to disable numerical compensation for the first input port from the previous model:

1. Select the block for which you want to select the port for numerical compensation. For example, get all the port handles for the currently selected block, `gcb`.

   ```
   p = get_param(gcb, 'PortHandles')
   ```

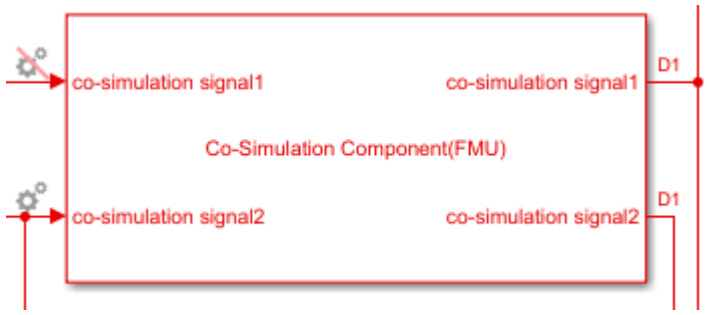
   This function returns all the ports for the currently selected block. For example,

   ```
   p =

     struct with fields:

         Inport: [22.0001 20.0001]
        Outport: [23.0001 25.0001]
         Enable: []
        Trigger: []
          State: []
          LConn: []
          RConn: []
       Ifaction: []
          Reset: []
   ```

2. To disable numerical compensation for the first port:

   ```
   set_param(p.Inport(1), 'CoSimSignalCompensationMode', 'Auto_Off')
   ```

   The associated port appears with a red slash.



You can also set signal compensation parameters from the command line. The first step, again, is to obtain the port handles:

```
p = get_param(block, 'PortHandles')
```

Set the compensation parameters using the `CoSimSignalCompensationConfig` parameter, in this format:

```
set_param(p.Inport,'CoSimSignalCompensationConfig','{<CompensationParam>:<ParamValue>}' )
```

Find compensation parameter names and possible values in this table:

Compensation parameter	Parameter name	Parameter value
`ExtrapolationMethod`	Extrapolation method	`'LinearExtrapolation'`, `'QuadraticExtrapolation'`, or `'CubicExtrapolation'`
`CompensationCoefficient`	Compensation coefficient	Scalar between `0` and `1`

For example, set the extrapolation method for the port:

```
set_param( p.Inport, 'CoSimSignalCompensationConfig', '{"ExtrapolationMethod":"LinearExtrapolati
```

Set both the extrapolation method and the compensation coefficient:

```
set_param(p.Inport,'CoSimSignalCompensationConfig', '{"ExtrapolationMethod":"QuadraticExtrapolation", "CompensationCoefficient":"0.7"}'
```

References

[1] Runge, Carl. "Uber empirische Funktionen und die Interpolation zwischen aquidistanten Ordinaten", *Zeitschrift für Mathematik und Physik*. Vol. 46, 1901, pp. 224–243.

See Also

FMU | S-Function | `ssGetInputPortIsContinuousQuantity` | `ssGetOutputPortIsContinuousQuantity` | `ssSetInputPortIsContinuousQuantity` | `ssSetOutputPortIsContinuousQuantity`

More About

- “Run Co-Simulation Components on Multiple Cores” on page 47-24
- “Identify co-simulation signals for numerical compensation”

Run Co-Simulation Components on Multiple Cores

Simulink supports co-simulation between components using local solvers or that involves simulation tools. For example, co-simulation can involve an S-function implemented as a co-simulation gateway between Simulink and third-party tools or custom code. It can also involve an FMU in co-simulation mode imported to Simulink.

To improve performance, consider running models that contain co-simulation blocks (components) on multiple threads if:

- You are integrating multiple co-simulation components
- Integration at the component level is computationally intense

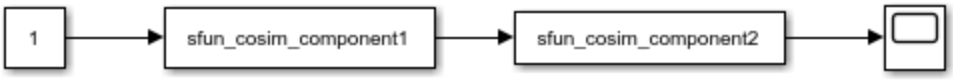
This topic assumes that you are familiar with multithreaded programming and concepts.

Simulink lets you run C MEX S-functions and Co-Simulation FMU blocks on multiple threads if they satisfy these requirements:

- The block is nondirect feedthrough.
- The block is threadsafe, that is, the block can work with multiple threads accessing shared data, resources, and objects without any conflicts.

Note Multithreaded co-simulation for Level-2 MATLAB S-Function blocks is not supported.

By default, Simulink configures all models to run on multiple threads. However, not all models have co-simulation components that can run on multiple threads, and not all models benefit from running on multiple threads. To see if a model has co-simulation components that can benefit from running on multiple threads, follow these steps:



1. Open your model.
2. Start the Performance Advisor tool (on the **Debug** tab, click **Performance Advisor**).
3. Select **Simulation > Checks that Require Simulation to Run > Select multi-thread co-simulation setting on or off**.

   This check verifies that the model or block is optimally configured to take advantage of multithreaded or singlethreaded processing. If the configuration is not optimal, Performance Advisor shows the current setting and warns you that the model is not a good candidate to run on multiple threads.
4. Run the selected check.

Alternatively, you can manually measure the simulation of the model before and after changing the `MultithreadedSim` parameter. For example:

```
tic; sim('sfunction_components'); toc
```

```
Elapsed time is 2.323264 seconds.
```

```
get_param(gcs,'MultithreadedSim')

ans =
 'on'

set_param(gcs,'MultithreadedSim','off')

tic; sim('sfunction_components'); toc

Elapsed time is 4.112674 seconds.
```

For an example of running co-simulation components on multiple cores, see the `slexCoSimPrimeExample` model.

Using the MultithreadedSim Parameter

You can specify that an entire model run on multiple threads, or specify that particular blocks run on multiple threads, using the `MultithreadedSim` parameter. Specify that an entire model run on multiple threads if all the co-simulation blocks in the model are nondirect feedthrough and threadsafe. If some, but not all blocks, are nondirect feedthrough and threadsafe, identify only those blocks to run on multiple threads. The model and blocks use the `MultithreadedSim` parameter as follows.

Setting	Model	Block	Description
`'on'`	✓		(Default) Model can run on multiple threads.
`'off'`	✓	✓	Disable the block or model from running on multiple threads.
`'auto'`		✓	(Default) Let Simulink decide if the block can run on multiple threads.

Enabling the `MultithreadedSim` parameter does not mean that the block or model simulates on multiple threads. Simulation on multiple threads occurs when `MultithreadedSim` is enabled and:

- The block and/or model operate at a single rate.
- The block and/or model are threadsafe. (For example, they do not use static or global data).
- The block and/or model are nondirect feedthrough.
  - For S-function blocks, use the `ssSetInputPortDirectFeedThrough` function.
  - For FMU blocks, leave the `dependencies` attribute in the FMU model description file for the FMU `ModelStructure/Outputs` and `ModelStructure/InitialUnknowns` field as empty. This attribute must be `" "`).
- The block and/or model is exception-free. For S-function blocks, use the `ssSetOptions` function to set `SS_OPTION_EXCEPTION_FREE_CODE`.

Multithreading does not allow solver reset checks, and therefore skips over any use of the `ssSetSolverNeedsReset` and `ssBlockStateForSolverChangedAtMajorStep` functions. Conversely, in accelerator mode, if these functions are used or there are continuous states, multithreading is automatically turned off.

Configuring S-Function Blocks to Run Single or Multithreaded

Whether an S-function block runs single or multithreaded depends on the `MultithreadedSim` parameter value and the `ssSetRuntimeThreadSafetyCompliance` function.

`MultithreadedSim` Setting	`ssSetRuntimeThreadSafetyCompliance` Setting	Single or Multithread
`'auto'`	`RUNTIME_THREAD_SAFETY_COMPLIANCE_UNKNOWN`	Single thread
`'auto'`	`RUNTIME_THREAD_SAFETY_COMPLIANCE_TRUE`	Multithread
`'auto'`	`RUNTIME_THREAD_SAFETY_COMPLIANCE_FALSE`	Single thread
`'off'`	—	The setting is ignored and the S-function block runs singlethreaded

Co-Simulation on Multiple Threads Limitations and Guidelines

- The simulation runs on a single thread for accelerator and rapid accelerator modes. Multithreading is enabled when the simulation mode is normal.

  Set model simulation mode to normal.
- There is no code generation for co-simulation components.
- When the block has these settings, it does not support co-simulation.
- Multithreading is not activated for blocks with constant sample time.
- Multithreading is not enabled when the Simulink debugger is on.

  Turn off Simulink debugger.
- A block that depends on a non-thread-safe block cannot be multithreaded. Consider breaking the dependency, for example, by using a Unit Delay block.

S-Function Block Limitations

- Must have a single rate.

  Consider revising your model to break down multirate components into individual single-rate components.
- Multithreading is not enabled when an S-function has variable sample time.

  Consider using a different sample time (see “Specify Sample Time” on page 7-3).
- Multithreading is not enabled when an S-function has continuous states and solver is fixed-step, which together trigger a continuous states consistency check. To disable continuous states consistency checks, use the `ssSetSkipContStatesConsistencyCheck` function.
- Must have no direct feedthrough ports — In `ssSetInputPortDirectFeedThrough(SimStruct *S,int_T port,int_T dirFeed)`, `dirFeed` must be `0` for each input port.

  Consider breaking the dependency between blocks, for example, by using a Unit Delay block.
- Must be thread safe — In `ssSetRuntimeThreadSafetyCompliance(SimStruct *S,int_T val)`, `val` must be `RUNTIME_THREAD_SAFETY_COMPLIANCE_TRUE`.

For more information, see “Guidelines for Writing Thread-Safe S-Functions”.

- Must be exception-free — In `ssSetOptions(SimStruct *S,uint_T options)`, `options` must include `SS_OPTION_EXCEPTION_FREE_CODE`.

  For more information, see “Guidelines for Writing Thread-Safe S-Functions”.
- Multithreading is not enabled when the S-function Analyzer is on. Try multithreading in normal mode.
- Multithreading is not enabled when S-function has continuous sample time. Consider using a different sample time (see “Specify Sample Time” on page 7-3).
- Multithreading concurrently runs output and update methods. The block must have an output or update method.

FMU Import Block Limitations

- Must be in co-simulation mode.

  Consider switching FMU mode from Model Exchange to Co-Simulation.
- Must be thread-safe, for example, multiple FMUs must not access the same file at the same time.
- Multithreading is not enabled when FMU block logging displays in the MATLAB command window. Redirect FMU block logging to a file using:

  ```
  set_param(blockName,'FMUDebugLoggingRedirect','File')
  ```
- Multithreading is not supported when FMU is running out-of-process. To disable this setting, use:

  ```
  set_param(blockName,'DebugExecutionForFMUViaOutOfProcess','off')
  ```

Model Block Limitations

- Multithreading is not enabled when a Model block has event ports.
- Cannot be inside a For Each Subsystem block.

  Consider moving the Model block out of the For Each Subsystem block.
- Must be in accelerator mode.
- Must have single rate.

  Consider revising your model to break down multirate components into individual single-rate components.
- Cannot use blocks with variable sample time.

  Consider using a different sample time (see “Specify Sample Time” on page 7-3).
- Cannot have continuous states.
- Cannot have direct feedthrough on any input port.

  Consider breaking the dependency between blocks, for example, by using a Unit Delay block.
- Must have a fixed-step solver.
- Cannot access any global data stores.
- Multithreading is not enabled when a model contains a Simulink Function block.
- Cannot use any Simulink functions or caller blocks.

- Cannot contain To File blocks. For more information, see “Export Simulation Data” on page 72-2.
- Cannot contain From File blocks.

  Consider feeding data into the referenced model via an inport from the top-level model.

See Also

FMU | S-Function | `ssGetRuntimeThreadSafetyCompliance` | `ssSetRuntimeThreadSafetyCompliance`

More About

- “Co-Simulation Execution and Numerical Compensation” on page 47-17
- “Multithread Co-Simulation”
- “Select multi-thread co-simulation setting on or off”
- “Guidelines for Writing Thread-Safe S-Functions”

Simulink Community and Connection Partner Program

Simulink supports the integration of multiple third-party functionalities, including apps, models, and toolboxes, from the Simulink community and commercial software tools.

For the integration of third-party functionality, this program includes:

- Simulink community — Provides direct access to all available Simulink apps, models, and toolboxes (Simulink community) using MATLAB Add-Ons. (To open the Add-On Explorer, go to the MATLAB Toolstrip and click **Add-Ons > get Add-Ons**.)
- Third-Party Products — The MathWorks Connections Program includes commercially offered products and services that complement MATLAB and Simulink.

Design Considerations for C/C++ Code Generation

When to Generate Code from MATLAB Algorithms

Generating code from MATLAB algorithms for desktop and embedded systems allows you to perform your software design, implementation, and testing completely within the MATLAB workspace. You can:

- Verify that your algorithms are suitable for code generation
- Generate efficient, readable, and compact C/C++ code automatically, which eliminates the need to manually translate your MATLAB algorithms and minimizes the risk of introducing errors in the code.
- Modify your design in MATLAB code to take into account the specific requirements of desktop and embedded applications, such as data type management, memory use, and speed.
- Test the generated code and easily verify that your modified algorithms are functionally equivalent to your original MATLAB algorithms.
- Generate MEX functions to:
  - Accelerate MATLAB algorithms in certain applications.
  - Speed up fixed-point MATLAB code.
- Generate hardware description language (HDL) from MATLAB code.

When Not to Generate Code from MATLAB Algorithms

Do not generate code from MATLAB algorithms for the following applications. Use the recommended MathWorks product instead.

To:	Use:
Deploy an application that uses handle graphics	MATLAB Compiler™
Use Java	MATLAB Compiler SDK™
Use toolbox functions that do not support code generation	Toolbox functions that you rewrite for desktop and embedded applications
Deploy MATLAB based GUI applications on a supported MATLAB host	MATLAB Compiler
Deploy web-based or Windows applications	MATLAB Compiler SDK
Interface C code with MATLAB	MATLAB `mex` function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	`codegen` function	MATLAB Coder	Try this in “Accelerate MATLAB Algorithm by Generating MEX Function” (MATLAB Coder).
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder app	MATLAB Coder	Try this in “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).
	`codegen` function	MATLAB Coder	Try this in “Generate C Code at the Command Line” (MATLAB Coder).
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder app	MATLAB Coder	See “Accelerate MATLAB Algorithms” (MATLAB Coder).
	`codegen` function	MATLAB Coder	
Integrate MATLAB code into Simulink	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code” on page 44-134.
Speed up fixed point MATLAB code	`fiaccel` function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB” (Fixed-Point Designer).
Integrate custom C code into MATLAB and generate efficient, readable code	`codegen` function	MATLAB Coder	Learn more in “Call C/C++ Code from MATLAB Code” (MATLAB Coder).
Integrate custom C code into code generated from MATLAB	`coder.ceval` function	MATLAB Coder	Learn more in `coder.ceval`.
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder	Learn more at `www.mathworks.com/products/slhdlcoder`.

Prerequisites for C/C++ Code Generation from MATLAB

To generate C/C++ or MEX code from MATLAB algorithms, you must install the following software:

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

  C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

  Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

  You can choose whether the generated code uses static or dynamic memory allocation.

  With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get better speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

  Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

  Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

  To improve the speed of the generated code:

  - Choose a suitable C/C++ compiler. Do not use the default compiler that MathWorks supplies with MATLAB for Windows 64-bit platforms.
  - Consider disabling run-time checks.

    By default, for safety, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower simulation. Disabling run-time checks usually results in streamlined generated code and faster simulation. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

See Also

- “Data Definition Basics”
- “Code Generation for Variable-Size Arrays” on page 53-2

Differences Between Generated Code and MATLAB Code

To convert MATLAB code to efficient C/C++ code, the code generator introduces optimizations that intentionally cause the generated code to behave differently, and sometimes produce different results, than the original source code.

Here are some of the differences:

Functions that have Multiple Possible Outputs

Certain mathematical operations, such as singular value decomposition and eigenvalue decomposition of a matrix, can have multiple answers. Two different algorithms implementing such an operation can return different outputs for identical input values. Two different implementations of the same algorithm can also exhibit the same behavior.

For such mathematical operations, the corresponding functions in the generated code and MATLAB might return different outputs for identical input values. To see if a function has this behavior, in the corresponding function reference page, see the **C/C++ Code Generation** section under **Extended Capabilities**. Examples of such functions include `svd` and `eig`.

Writing to ans Variable

When you run MATLAB code that returns an output without specifying an output argument, MATLAB implicitly writes the output to the `ans` variable. If the variable `ans` already exists in the workspace, MATLAB updates its value to the output returned.

The code generated from such MATLAB code does not implicitly write the output to an `ans` variable.

For example, define the MATLAB function `foo` that explicitly creates an `ans` variable in the first line. The function then implicitly updates the value of `ans` when the second line executes.

```
function foo %#codegen
ans = 1;
2;
disp(ans);
end
```

Run `foo` at the command line. The final value of `ans`, which is `2`, is displayed at the command line.

```
foo
```

```
2
```

Generate a MEX function from `foo`.

```
codegen foo
```

Run the generated MEX function `foo_mex`. This function explicitly creates the `ans` variable and assigns the value `1` to it. But `foo_mex` does not implicitly update the value of `ans` to `2`.

```
foo_mex
```

```
1
```

Logical Short-Circuiting

Suppose that your MATLAB code has the logical operators `&` and `|` placed inside square brackets (`[` and `]`). For such code patterns, the generated code does not employ short-circuiting behavior for these logical operators, but MATLAB execution might employ short-circuiting behavior. See “Logical Short-Circuiting”.

For example, define the MATLAB function `foo` that uses the `&` operator inside square brackets in the conditional expression of an `if...end` block.

```
function foo
if [returnsFalse() & hasSideEffects()]
end
end

function out = returnsFalse
out = false;
end

function out = hasSideEffects
out = true;
disp('This is my string');
end
```

The first argument of the & operator is always `false` and determines the value of the conditional expression. So, in MATLAB execution, short-circuiting is employed and the second argument is not evaluated. So, `foo` does not call the `hasSideEffects` function during execution and does not display anything at the command line.

Generate a MEX function for `foo`. Call the generated MEX function `foo_mex`.

```
foo_mex
```

```
This is my string
```

In the generated code, short-circuiting is not employed. So, the `hasSideEffects` function is called and the string is displayed at the command line.

Loop Index Overflow

Suppose that a `for`-loop end value is equal to or close to the maximum or minimum value for the loop index data type. In the generated code, the last increment or decrement of the loop index might cause the index variable to overflow. The index overflow might result in an infinite loop.

When memory integrity checks are enabled, if the code generator detects that the loop index might overflow, it reports an error. The software error checking is conservative. It might incorrectly report a loop index overflow. By default, memory-integrity checks are enabled for MEX code and disabled for standalone C/C++ code. See “Why Test MEX Functions in MATLAB?” (MATLAB Coder) and “Run-Time Error Detection and Reporting in Standalone C/C++ Code” (MATLAB Coder).

To avoid a loop index overflow, use the workarounds in this table.

Loop Conditions Causing the Potential Overflow	Workaround

• The loop index increments by 1.
• The end value equals the maximum value of the integer type.
	If the loop does not have to cover the full range of the integer type, rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:
	`N=intmax('int16')`
	`for k=N-10:N`
	with:
	`for k=1:10`

• The loop index decrements by 1.
• The end value equals the minimum value of the integer type.
	If the loop does not have to cover the full range of the integer type, rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:
	`N=intmin('int32')`
	`for k=N+10:-1:N`
	with:
	`for k=10:-1:1`

Loop Conditions Causing the Potential Overflow	Workaround

• The loop index increments or decrements by 1.
• The start value equals the minimum or maximum value of the integer type.
• The end value equals the maximum or minimum value of the integer type.
	If the loop must cover the full range of the integer type, cast the type of the loop start, step, and end values to a bigger integer or to double. For example, rewrite:
	`M= intmin('int16');`
	`N= intmax('int16');`
	`for k=M:N`
	`  % Loop body`
	`end`
	as:
	`M= intmin('int16');`
	`N= intmax('int16');`
	`for k=int32(M):int32(N)`
	`  % Loop body`
	`end`

• The loop index increments or decrements by a value not equal to 1. • On the last loop iteration, the loop index is not equal to the end value.	Rewrite the loop so that the loop index in the last loop iteration is equal to the end value.

Index of an Unentered for Loop

In your MATLAB code and generated code, after a `for`-loop execution is complete, the value of the index variable is equal to its value during the final iteration of the `for`-loop.

In MATLAB, if the loop does not execute, the value of the index variable is stored as [ ] (empty matrix). In generated code, if the loop does not execute, the value of the index variable is different than the MATLAB index variable.

- If you provide the `for`-loop start and end variables at run time, the value of the index variable is equal to the start of the range. For example, consider this MATLAB code:

  ```
  function out = indexTest(a,b)
  for i = a:b
  end
  out = i;
  end
  ```

  Suppose that `a` and `b` are passed as `1` and `-1`. The `for`-loop does not execute. In MATLAB, `out` is assigned [ ]. In the generated code, `out` is assigned the value of `a`, which is `1`.

- If you provide the `for`-loop start and end values before compile time, the value of the index variable is equal to `0`. Consider this MATLAB code:

  ```
  function out = indexTest
  for i = 1:-1
  end
  out = i;
  end
  ```

Suppose that you call this function. In MATLAB, `out` is assigned [ ]. In the generated code, `out` is assigned the value `0`.

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Encoding of Characters in Code Generation” on page 52-6.

Order of Evaluation in Expressions

Generated code does not enforce order of evaluation in expressions. For most expressions, order of evaluation is not significant. However, for expressions with side effects, the generated code may produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that:

- Modify persistent or global variables
- Display data to the screen
- Write data to files
- Modify the properties of handle class objects

In addition, the generated code does not enforce order of evaluation of logical operators that do not short circuit.

For more predictable results, it is good coding practice to split expressions that depend on the order of evaluation into multiple statements.

- Rewrite

  ```
  A = f1() + f2();
  ```

  as

  ```
  A = f1();
  A = A + f2();
  ```

  so that the generated code calls `f1` before `f2`.
- Assign the outputs of a multi-output function call to variables that do not depend on one another. For example, rewrite

  ```
  [y, y.f, y.g] = foo;
  ```

  as

  ```
  [y, a, b] = foo;
  y.f = a;
  y.g = b;
  ```
- When you access the contents of multiple cells of a cell array, assign the results to variables that do not depend on one another. For example, rewrite

  ```
  [y, y.f, y.g] = z{:};
  ```

  as

```
[y, a, b] = z{:};
y.f = a;
y.g = b;
```

Name Resolution While Constructing Function Handles

MATLAB and code generation follow different precedence rules for resolving names that follow the symbol `@`. These rules do not apply to anonymous functions. The precedence rules are summarized in this table.

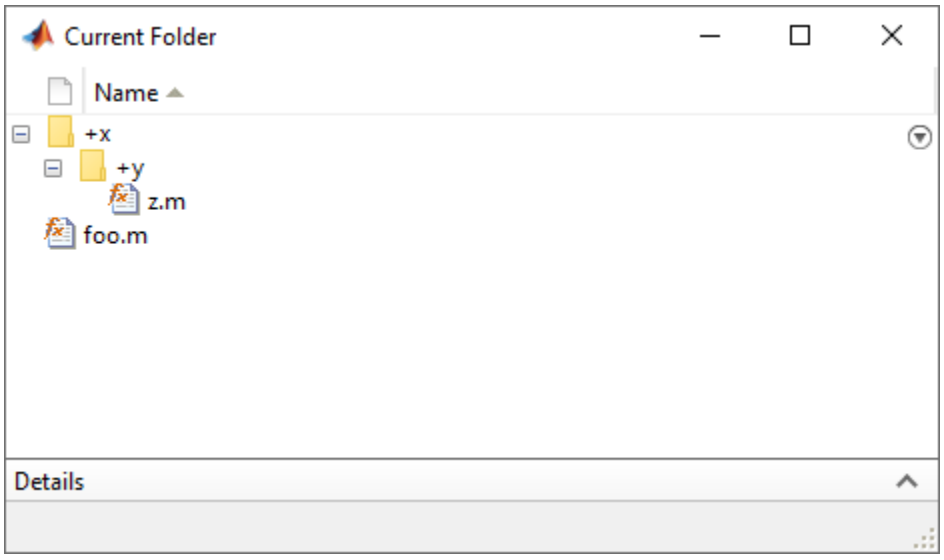
Expression	Precedence Order in MATLAB	Precedence Order in Code Generation
An expression that does not contain periods, for example `@x`	Nested function, local function, private function, path function	Local variable, nested function, local function, private function, path function
An expression that contains exactly one period, for example `@x.y`	Local variable, path function	Local variable, path function (Same as MATLAB)
An expression that contains more than one period, for example `@x.y.z`	Path function	Local variable, path function

If `x` is a local variable that is itself a function handle, generated code and MATLAB interpret the expression `@x` differently:

- MATLAB produces an error.
- Generated code interprets `@x` as the function handle of `x` itself.

Here is an example that shows this difference in behavior for an expression that contains two periods.

Suppose that your current working folder contains a package `x`, which contains another package `y`, which contains the function `z`. The current working folder also contains the entry-point function `foo` for which you want to generate code.



This is the definition for the file `foo`:

```
function out = foo
    x.y.z = @()'x.y.z is an anonymous function';
    out = g(x);
end

function out = g(x)
    f = @x.y.z;
    out = f();
end
```

This is the definition for function `z`:

```
function out = z
    out = 'x.y.z is a package function';
end
```

Generate a MEX function for `foo`. Separately call both the generated MEX function `foo_mex` and the MATLAB function `foo`.

```
codegen foo
foo_mex
foo

ans =

    'x.y.z is an anonymous function'


ans =

    'x.y.z is a package function'
```

The generated code produces the first output. MATLAB produces the second output. Code generation resolves `@x.y.z` to the local variable `x` that is defined in `foo`. MATLAB resolves `@x.y.z` to `z`, which is within the package `x.y`.

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, if infinite loops do not have side effects, optimizations remove them from generated code. As a result, the generated code can possibly terminate even though the corresponding MATLAB code does not.

Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code, but always returns `[4 2]` in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 53-16.

Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. See “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 53-17.

Size of Empty Array That Results from Deleting Elements of an Array

Deleting all elements of an array results in an empty array. The size of this empty array in generated code might differ from its size in MATLAB source code.

Case	Example Code	Size of Empty Array in MATLAB	Size of Empty Array in Generated Code

Delete all elements of an m-by-n array by using the `colon` operator (`:`).	`coder.varsize('X',[4,4],[1,1]);`
	`X = zeros(2);`
	`X(:) = [];`
		`0-by-0`	`1-by-0`

Delete all elements of a row vector by using the `colon` operator (`:`).	`coder.varsize('X',[1,4],[0,1]);`
	`X = zeros(1,4);`
	`X(:) = [];`
		`0-by-0`	`1-by-0`

Delete all elements of a column vector by using the `colon` operator (`:`).	`coder.varsize('X',[4,1],[1,0]);`
	`X = zeros(4,1);`
	`X(:) = [];`
		`0-by-0`	`0-by-1`

Delete all elements of a column vector by deleting one element at a time.	`coder.varsize('X',[4,1],[1,0]);`
	`X = zeros(4,1);`
	`for i = 1:4`
	`  X(1)= [];`
	`end`
		`1-by-0`	`0-by-1`

Binary Element-Wise Operations with Single and Double Operands

If your MATLAB code contains a binary element-wise operation that involves a single type operand and a double type operand, the generated code might not produce the same result as MATLAB.

For such an operation, MATLAB casts both operands to double type and performs the operation with the double types. MATLAB then casts the result to single type and returns it.

The generated code casts the double type operand to single type. It then performs the operation with the two single types and returns the result.

For example, define a MATLAB function `foo` that calls the binary element-wise operation `plus`.

```
function out = foo(a,b)
out = a + b;
end
```

Define a variable `s1` of single type and a variable `v1` of double type. Generate a MEX function for `foo` that accepts a single type input and a double type input.

```
s1 = single(1.4e32);
d1 = -5.305e+32;
codegen foo -args {s1, d1}
```

Call both `foo` and `foo_mex` with inputs `s1` and `d1`. Compare the two results.

```
ml = foo(s1,d1);
mlc = foo_mex(s1,d1);
ml == mlc

ans =

  logical

   0
```

The output of the comparison is a logical `0`, which indicates that the generated code and MATLAB produces different results for these inputs.

Floating-Point Numerical Results

The generated code might not produce the same floating-point numerical results as MATLAB in these:

When computer hardware uses extended precision registers

Results vary depending on how the C/C++ compiler allocates extended precision floating-point registers. Computation results might not match MATLAB calculations because of different compiler optimization settings or different code surrounding the floating-point calculations.

For certain advanced library functions

The generated code might use different algorithms to implement certain advanced library functions, such as `fft`, `svd`, `eig`, `mldivide`, and `mrdivide`.

For example, the generated code uses a simpler algorithm to implement `svd` to accommodate a smaller footprint. Results might also vary according to matrix properties. For example, MATLAB might detect symmetric or Hermitian matrices at run time and switch to specialized algorithms that perform computations faster than implementations in the generated code.

For implementation of BLAS library functions

For implementations of BLAS library functions, generated C/C++ code uses reference implementations of BLAS functions. These reference implementations might produce different results from platform-specific BLAS implementations in MATLAB.

NaN and Infinity

The generated code might not produce exactly the same pattern of `NaN` and `Inf` values as MATLAB code when these values are mathematically meaningless. For example, if MATLAB output contains a `NaN`, output from the generated code should also contain a `NaN`, but not necessarily in the same place.

The bit pattern for `NaN` can differ between MATLAB code output and generated code output because the C99 standard math library that is used to generate code does not specify a unique bit pattern for `NaN` across all implementations. Avoid comparing bit patterns across different implementations, for example, between MATLAB output and SIL or PIL output.

Negative Zero

In a floating-point type, the value `0` has either a positive sign or a negative sign. Arithmetically, `0` is equal to `-0`, but some operations are sensitive to the sign of a 0 input. Examples include `rdivide`, `atan2`, `atan2d`, and `angle`. Division by `0` produces `Inf`, but division by `-0` produces `-Inf`. Similarly, `atan2d(0,-1)` produces `180`, but `atan2d (-0,-1)` produces `-180`.

If the code generator detects that a floating-point variable takes only integer values of a suitable range, then the code generator can use an integer type for the variable in the generated code. If the code generator uses an integer type for the variable, then the variable stores `-0` as `+0` because an integer type does not store a sign for the value `0`. If the generated code casts the variable back to a floating-point type, the sign of `0` is positive. Division by `0` produces `Inf`, not `-Inf`. Similarly, `atan2d(0,-1)` produces `180`, not `-180`.

Code Generation Target

The `coder.target` function returns different values in MATLAB than in the generated code. The intent is to help you determine whether your function is executing in MATLAB or has been compiled for a simulation or code generation target. See `coder.target`.

MATLAB Class Property Initialization

Before code generation, at class loading time, MATLAB computes class default values. The code generator uses the values that MATLAB computes. It does not recompute default values. If the property definition uses a function call to compute the initial value, the code generator does not execute this function. If the function has side effects such as modifying a global variable or a persistent variable, then it is possible that the generated code can produce different results that MATLAB produces. For more information, see “Defining Class Properties for Code Generation” on page 61-4.

MATLAB Classes in Nested Property Assignments That Have Set Methods

When you assign a value to a handle object property, which is itself a property of another object, and so on, then the generated code can call set methods for handle classes that MATLAB does not call.

For example, suppose that you define a set of variables such that `x` is a handle object, `pa` is an object, `pb` is a handle object, and `pc` is a property of `pb`. Then you make a nested property assignment, such as:

```
x.pa.pb.pc = 0;
```

In this case, the generated code calls the set method for the object `pb` and the set method for `x`. MATLAB calls only the set method for `pb`.

MATLAB Handle Class Destructors

The behavior of handle class destructors in the generated code can be different from the behavior in MATLAB in these situations:

- The order of destruction of several independent objects might be different in MATLAB than in the generated code.

- The lifetime of objects in the generated code can be different from their lifetime in MATLAB.
- The generated code does not destroy partially constructed objects. If a handle object is not fully constructed at run time, the generated code produces an error message but does not call the `delete` method for that object. For a System object, if there is a run-time error in `setupImpl`, the generated code does not call `releaseImpl` for that object.

  MATLAB does call the `delete` method to destroy a partially constructed object.

For more information, see “Code Generation for Handle Class Destructors” on page 61-16.

Variable-Size Data

See “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 53-15.

Complex Numbers

See “Code Generation for Complex Data” on page 52-3.

Converting Strings with Consecutive Unary Operators to double

Converting a string that contains multiple, consecutive unary operators to `double` can produce different results between MATLAB and the generated code. Consider this function:

```
function out = foo(op)
out = double(op + 1);
end
```

For an input value `"--"`, the function converts the string `"--1"` to `double`. In MATLAB, the answer is `NaN`. In the generated code, the answer is `1`.

See Also

MATLAB Language Features Supported for C/C++ Code Generation

MATLAB Features That Code Generation Supports

Code generation from MATLAB code supports many major language features including:

- n-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 52-7)
- matrix operations, including deletion of rows and columns
- variable-size data (see “Code Generation for Variable-Size Arrays” on page 53-2)
- subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 53-19)
- complex numbers (see “Code Generation for Complex Data” on page 52-3)
- numeric classes (see “Supported Variable Types” on page 51-11)
- double-precision, single-precision, and integer math
- enumerations (see “Code Generation for Enumerations” on page 44-84)
- fixed-point arithmetic
- program control statements `if`, `switch`, `for`, `while`, and `break`
- arithmetic, relational, and logical operators
- local functions
- persistent variables
- global variables
- structures (see “Structure Definition for Code Generation” on page 54-2)
- cell arrays (see “Cell Arrays”)
- tables (see “Code Generation for Tables” on page 59-2)
- timetables (see “Code Generation for Timetables” on page 60-2)
- characters (see “Encoding of Characters in Code Generation” on page 52-6)
- string scalars (see “Code Generation for Strings” on page 52-10)
- `categorical` arrays (see “Code Generation for Categorical Arrays” on page 56-2)
- `datetime` arrays (see “Code Generation for Datetime Arrays” on page 57-2)
- `duration` arrays (see “Code Generation for Duration Arrays” on page 58-2)
- sparse matrices (see “Code Generation for Sparse Matrices” on page 52-11)
- function handles (see “Function Handle Limitations for Code Generation” on page 62-2)
- anonymous functions (see “Code Generation for Anonymous Functions” on page 63-3)
- recursive functions (see “Code Generation for Recursive Functions” on page 64-16)
- nested functions (see “Code Generation for Nested Functions” on page 63-4)
- variable length input and output argument lists (see “Code Generation for Variable Length Argument Lists” on page 63-2)
- subset of MATLAB toolbox functions (see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2)

- subset of functions and System objects in several toolboxes (see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2)
- MATLAB classes (see “MATLAB Classes Definition for Code Generation” on page 61-2)
- function calls (see “Resolution of Function Calls for Code Generation” on page 64-2)

MATLAB Language Features That Code Generation Does Not Support

Code generation from MATLAB does not support the following frequently used MATLAB features (this list is not exhaustive):

- scripts
- implicit expansion

  Code generation does not support implicit expansion of arrays with compatible sizes during execution of element-wise operations or functions. If your MATLAB code relies on implicit expansion, code generation results in a size-mismatch error. For fixed-size arrays, the error occurs at compile time. For variable-size arrays, the error occurs at run time. For more information about implicit expansion, see “Compatible Array Sizes for Basic Operations”. For code generation, to achieve implicit expansion, use `bsxfun`.
- GPU arrays

  MATLAB Coder does not support GPU arrays. However, if you have GPU Coder™, you can generate CUDA® MEX code that takes GPU array inputs.
- `calendarDuration` arrays
- Java
- Map containers
- time series objects
- `try`/`catch` statements
- `import` statements
- Function argument validation

Functions, Classes, and System Objects Supported for Code Generation

Functions and Objects Supported for C/C++ Code Generation

You can generate efficient C/C++ code for a subset of MATLAB built-in functions and toolbox functions and System objects that you call from MATLAB code.

These functions and System objects are listed in the following tables. In these tables, a  icon before the name of a function or a System object indicates that there are specific usage notes and limitations related to C/C++ code generation for that function or System object. To view these usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **C/C++ Code Generation** section.

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

See Also

More About

- “MATLAB Language Features Supported for C/C++ Code Generation” on page 48-17

System Objects Supported for Code Generation

Code Generation for System Objects

You can generate C and C++ code for a subset of System objects provided by the following toolboxes.

Toolbox Name	See
Communications Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)
Computer Vision Toolbox™	“System Objects in MATLAB Code Generation” (MATLAB Coder)
DSP System Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)
Image Acquisition Toolbox™	• `imaq.VideoDevice` • “Code Generation with VideoDevice System Object” (Image Acquisition Toolbox)
Phased Array System Toolbox™	“Code Generation” (Phased Array System Toolbox)
System Identification Toolbox	“Generate Code for Online Parameter Estimation in MATLAB” (System Identification Toolbox)
WLAN Toolbox™	“System Objects in MATLAB Code Generation” (MATLAB Coder)

To use these System objects, you need to install the requisite toolbox. For a list of System objects supported for C and C++ code generation, see “Functions and Objects Supported for C/C++ Code Generation” on page 49-2.

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. For general information about MATLAB objects, see “Classes”.

Defining MATLAB Variables for C/C++ Code Generation

Variables Definition for Code Generation

In the MATLAB language, variables can change their properties dynamically at run time so you can use the same variable to hold a value of any class, size, or complexity. For example, the following code works in MATLAB:

```
function x = foo(c) %#codegen
if(c>0)
  x = 0;
else
  x = [1 2 3];
end
disp(x);
end
```

However, statically-typed languages like C must be able to determine variable properties at compile time. Therefore, for C/C++ code generation, you must explicitly define the class, size, and complexity of variables in MATLAB source code before using them. For example, rewrite the above source code with a definition for *x*:

```
function x = foo(c) %#codegen
x = zeros(1,3);
if(c>0)
  x = 0;
else
  x = [1 2 3];
end
disp(x);
end
```

For more information, see “Best Practices for Defining Variables for C/C++ Code Generation” on page 51-3.

Best Practices for Defining Variables for C/C++ Code Generation

In this section...

Define Variables By Assignment Before Using Them

For C/C++ code generation, you should explicitly and unambiguously define the class, size, and complexity of variables before using them in operations or returning them as outputs. Define variables by assignment, but note that the assignment copies not only the value, but also the size, class, and complexity represented by that value to the new variable. For example:

Assignment:	Defines:
`a = 14.7;`	`a` as a real double scalar.
`b = a;`	`b` with properties of `a` (real double scalar).
`c = zeros(5,2);`	`c` as a real 5-by-2 array of doubles.
`d = [1 2 3 4 5; 6 7 8 9 0];`	`d` as a real 5-by-2 array of doubles.
`y = int16(3);`	`y` as a real 16-bit integer scalar.

Define properties this way so that the variable is defined on the required execution paths during C/C++ code generation.

The data that you assign to a variable can be a scalar, matrix, or structure. If your variable is a structure, define the properties of each field explicitly.

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in “Eliminate Redundant Copies of Variables in Generated Code” on page 51-6.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see `persistent`.

Example 51.1. Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

```
...
if c > 0
  x = 11;
end
% Later in your code ...
if c > 0
  use(x);
```

```
end
...
```

Here, *x* is assigned only if `c > 0` and used only when `c > 0`. This code works in MATLAB, but generates a compilation error during code generation because it detects that *x* is undefined on some execution paths (when `c <= 0`).

To make this code suitable for code generation, define *x* before using it:

```
x = 0;
...
if c > 0
  x = 11;
end
% Later in your code ...
if c > 0
  use(x);
end
...
```

Example 51.2. Defining Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
  s.a = 11;
  disp(s);
else
  s.a = 12;
  s.b = 12;
end
% Try to use s
use(s);
...
```

Here, the first part of the `if` statement uses only the field *a*, and the `else` clause uses fields *a* and *b*. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation” on page 54-2.

To make this code suitable for C/C++ code generation, define all fields of *s* before using it.

```
...
% Define all fields in structure s
s = struct('a',0, 'b', 0);
if c > 0
  s.a = 11;
  disp(s);
else
  s.a = 12;
  s.b = 12;
end
% Use s
use(s);
...
```

Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 51-8.

Use Type Cast Operators in Variable Definitions

By default, constants are of type `double`. To define variables of other types, you can use type cast operators in variable definitions. For example, the following code defines variable `y` as an integer:

```
...
x = 15; % x is of type double by default.
y = uint8(x); % y has the value of x, but cast to uint8.
...
```

Define Matrices Before Assigning Indexed Variables

When generating C/C++ code from MATLAB, you cannot grow a variable by writing into an element beyond its current size. Such indexing operations produce run-time errors. You must define the matrix first before assigning values to its elements.

For example, the following initial assignment is not allowed for code generation:

```
g(3,2) = 14.6; % Not allowed for creating g
               % OK for assigning value once created
```

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 53-19.

See Also

`coder.nullcopy` | `persistent`

More About

- “Eliminate Redundant Copies of Variables in Generated Code” on page 51-6
- “Structure Definition for Code Generation” on page 54-2
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 53-19
- “Avoid Data Copies of Function Inputs in Generated Code” (MATLAB Coder)

Eliminate Redundant Copies of Variables in Generated Code

In this section...

When Redundant Copies Occur

During C/C++ code generation, the code generator checks for statements that attempt to access uninitialized memory. If it detects execution paths where a variable is used but is potentially not defined, it generates a compile-time error. To prevent these errors, define variables by assignment before using them in operations or returning them as function outputs.

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 51-6.

How to Eliminate Redundant Copies by Defining Uninitialized Variables

1. Define the variable with `coder.nullcopy`.
2. Initialize the variable before reading it.

   When the uninitialized variable is an array, you must initialize all of its elements before passing the array as an input to a function or operator — even if the function or operator does not read from the uninitialized portion of the array.

   What happens if you access uninitialized data?

   Uninitialized memory contains arbitrary values. Therefore, accessing uninitialized data may lead to segmentation violations or nondeterministic program behavior (different runs of the same program may yield inconsistent results).

Defining Uninitialized Variables

In the following code, the assignment statement `X = zeros(1,N)` not only defines `X` to be a 1-by-5 vector of real doubles, but also initializes each element of `X` to zero.

```
function X = withoutNullcopy %#codegen

N = 5;
X = zeros(1,N);
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    elseif mod(i,2) == 1
        X(i) = 0;
```

```
    end
end
```

This forced initialization creates an extra copy in the generated code. To eliminate this overhead, use `coder.nullcopy` in the definition of `X`:

```
function X = withNullcopy %#codegen

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
```

See Also

`coder.nullcopy`

More About

- “Avoid Data Copies of Function Inputs in Generated Code” (MATLAB Coder)

Reassignment of Variable Properties

For C/C++ code generation, there are certain variables that you can reassign after the initial assignment with a value of different class, size, or complexity:

Dynamically sized variables

A variable can hold values that have the same class and complexity but different sizes. If the size of the initial assignment is not constant, the variable is dynamically sized in generated code. For more information, see “Variable-Size Data”.

Variables reused in the code for different purposes

You can reassign the type (class, size, and complexity) of a variable after the initial assignment if each occurrence of the variable can have only one type. In this case, the variable is renamed in the generated code to create multiple independent variables. For more information, see “Reuse the Same Variable with Different Properties” on page 51-9.

Reuse the Same Variable with Different Properties

In this section...

When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if the code generator can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report.

A common example of variable reuse is in `if-elseif-else` or `switch-case` statements. For example, the following function `example1` first uses the variable *t* in an `if` statement, where it holds a scalar double, then reuses *t* outside the `if` statement to hold a vector of doubles.

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
```

When You Cannot Reuse Variables

You cannot reuse (reassign) variables if it is not possible to determine the class, size, and complexity of an occurrence of a variable unambiguously during code generation. In this case, variables cannot be renamed and a compilation error occurs.

For example, the following `example2` function assigns a fixed-point value to *x* in the `if` statement and reuses *x* to store a matrix of doubles in the `else` clause. It then uses *x* after the `if-else` statement. This function generates a compilation error because after the `if-else` statement, variable *x* can have different properties depending on which `if-else` clause executes.

```
function y = example2(use_fixpoint, data) %#codegen
  if use_fixpoint
            % x is fixed-point
      x = fi(data, 1, 12, 3);
  else
             % x is a matrix of doubles
      x = data;
  end
  % When x is reused here, it is not possible to determine its
  % class, size, and complexity
  t = sum(sum(x));
  y = t > 0;
end
```

Example 51.3. Variable Reuse in an if Statement

To see how MATLAB renames a reused variable `t`:

1. Create a MATLAB file `example1.m` containing the following code.

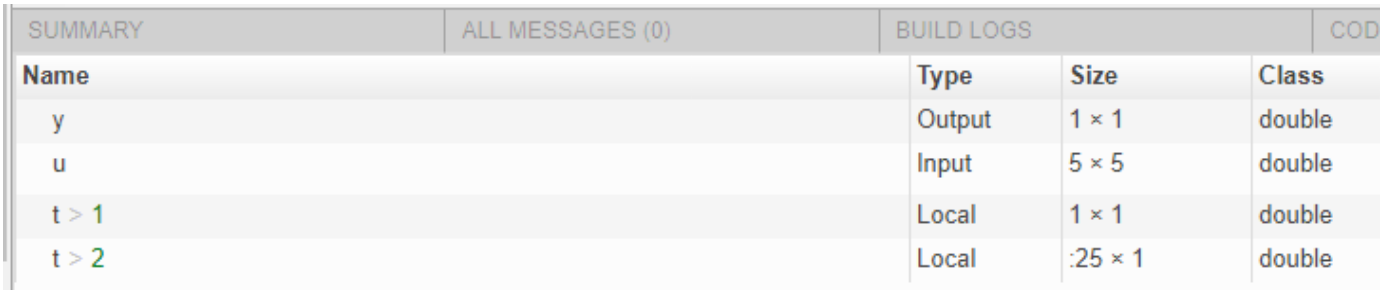
```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
end
```

2. Generate a MEX function for `example1` and produce a code generation report.

```
codegen -o example1x -report example1.m -args {ones(5,5)}
```

3. Open the code generation report.

   On the **Variables** tab, you see two uniquely named local variables `t>1` and `t>2`.



Name	Type	Size	Class
y	Output	1 × 1	double
u	Input	5 × 5	double
t > 1	Local	1 × 1	double
t > 2	Local	:25 × 1	double

4. In the list of variables, click `t>1`. The report highlights the instances of the variable `t` that are inside of the `if` statement. These instances of `t` are scalar double.
5. Click `t>2`. The code generation report highlights the instances of `t` that are outside of the `if` statement. These instances of `t` are variable-size column vectors with an upper bound of 25.

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

- Persistent variables.
- Global variables.
- Variables passed to C code using `coder.ref`, `coder.rref`, `coder.wref`.
- Variables whose size is set using `coder.varsize`.
- The index variable of a `for`-loop when it is used inside the loop body.
- The block outputs of a MATLAB Function block in a Simulink model.
- Chart-owned variables of a MATLAB function in a Stateflow chart.

Supported Variable Types

You can use the following data types for C/C++ code generation from MATLAB:

Type	Description
`char`	Character array
`complex`	Complex data. Cast function takes real and imaginary components
`double`	Double-precision floating point
`int8`, `int16`, `int32`, `int64`	Signed integer
`logical`	Boolean `true` or `false`
`single`	Single-precision floating point
`struct`	Structure
`uint8`, `uint16`, `uint32`, `uint64`	Unsigned integer
Fixed-point	Fixed-point data types

Defining Data for Code Generation

Data Definition for Code Generation

To generate efficient standalone code, you must define the following types and classes of data differently than you normally would when running your code in MATLAB.

Data	What Is Different	More Information
Arrays	Maximum number of elements is restricted	"Array Size Restrictions for Code Generation" on page 52-7
Complex numbers	• Complexity of variables must be set at time of assignment and before first use • Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero **Note** Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic	"Code Generation for Complex Data" on page 52-3
Characters	Restricted to 8 bits of precision	"Encoding of Characters in Code Generation" on page 52-6
Enumerated data	• Supports integer-based enumerated types only • Restricted use in `switch` statements and `for`-loops	"Enumerations"
Function handles	• Using the same bound variable to reference different function handles can cause a compile-time error. • Cannot pass function handles to or from primary or extrinsic functions • Cannot view function handles from the debugger	"Function Handles"

Code Generation for Complex Data

In this section...

Restrictions When Defining Complex Variables

For code generation, you must set the complexity of variables at the time of assignment. Assign a complex constant to the variable or use the `complex` function. For example:

```
x = 5 + 6i; % x is a complex number by assignment.
y = complex(5,6); % y is the complex number 5 + 6i.
```

After assignment, you cannot change the complexity of a variable. Code generation for the following function fails because `x(k) = 3 + 4i` changes the complexity of `x`.

```
function x = test1( )
x = zeros(3,3); % x is real
for k = 1:numel(x)
    x(k) = 3 + 4i;
end
end
```

To resolve this issue, assign a complex constant to `x`.

```
function x = test1( )
x = zeros(3,3)+ 0i; %x is complex
for k = 1:numel(x)
    x(k) = 3 + 4i;
end
end
```

Code Generation for Complex Data with Zero-Valued Imaginary Parts

For code generation, complex data that has all zero-valued imaginary parts remains complex. This data does not become real. This behavior has the following implications:

- In some cases, results from functions that sort complex data by absolute value can differ from the MATLAB results. See “Functions That Sort Complex Values by Absolute Value” on page 52-3.
- For functions that require that complex inputs are sorted by absolute value, complex inputs with zero-valued imaginary parts must be sorted by absolute value. These functions include `ismember`, `union`, `intersect`, `setdiff`, and `setxor`.

Functions That Sort Complex Values by Absolute Value

Functions that sort complex values by absolute value include `sort`, `issorted`, `sortrows`, `median`, `min`, and `max`. These functions sort complex numbers by absolute value even when the imaginary parts are zero. In general, sorting the absolute values produces a different result than sorting the real parts. Therefore, when inputs to these functions are complex with zero-valued imaginary parts in

generated code, but real in MATLAB, the generated code can produce different results than MATLAB. In the following examples, the input to `sort` is real in MATLAB, but complex with zero-valued imaginary parts in the generated code:

- **You Pass Real Inputs to a Function Generated for Complex Inputs**

  1. Write this function:

     ```
     function myout = mysort(A)
     myout = sort(A);
     end
     ```

  2. Call `mysort` in MATLAB.

     ```
     A = -2:2;
     mysort(A)
     ```

     ```
     ans =

         -2    -1     0     1     2
     ```

  3. Generate a MEX function for complex inputs.

     ```
     A = -2:2;
     codegen mysort -args {complex(A)} -report
     ```

  4. Call the MEX Function with real inputs.

     ```
     mysort_mex(A)
     ```

     ```
     ans =

          0     1    -1     2    -2
     ```

     You generated the MEX function for complex inputs, therefore, it treats the real inputs as complex numbers with zero-valued imaginary parts. It sorts the numbers by the absolute values of the complex numbers. Because the imaginary parts are zero, the MEX function returns the results to the MATLAB workspace as real numbers.

- **Input to `sort` Is Output from a Function That Returns Complex in Generated Code**

  1. Write this function:

     ```
     function y = myfun(A)
     x = eig(A);
     y = sort(x,'descend');
     ```

     The output from `eig` is the input to `sort`. In generated code, `eig` returns a complex result. Therefore, in the generated code, `x` is complex.

  2. Call `myfun` in MATLAB.

     ```
     A = [2 3 5;0 5 5;6 7 4];
     myfun(A)
     ```

     ```
     ans =

        12.5777
         2.0000
        -3.5777
     ```

     The result of `eig` is real. Therefore, the inputs to `sort` are real.

3. Generate a MEX function for complex inputs.

   ```
   codegen myfun -args {complex(A)}
   ```

4. Call the MEX function.

   ```
   myfun_mex(A)
   ```

   ```
   ans =

      12.5777
      -3.5777
       2.0000
   ```

   In the MEX function, `eig` returns a complex result. Therefore, the inputs to `sort` are complex. The MEX function sorts the inputs in descending order of the absolute values.

Results of Expressions That Have Complex Operands

In general, expressions that contain one or more complex operands produce a complex result in generated code, even if the value of the result is zero. Consider the following line of code:

```
z = x + y;
```

Suppose that at run time, `x` has the value `2 + 3i` and `y` has the value `2 - 3i`. In MATLAB, this code produces the real result `z = 4`. During code generation, the types for `x` and `y` are known, but their values are not known. Because either or both operands in this expression are complex, `z` is defined as a complex variable requiring storage for a real and an imaginary part. `z` equals the complex result `4 + 0i` in generated code, not `4`, as in MATLAB code.

Exceptions to this behavior are:

- Functions that take complex arguments but produce real results return real values.

  ```
  y = real(x); % y is the real part of the complex number x.
  y = imag(x); % y is the real-valued imaginary part of x.
  y = isreal(x); % y is false (0) for a complex number x.
  ```

- Functions that take real arguments but produce complex results return complex values.

  ```
  z = complex(x,y); % z is a complex number for a real x and y.
  ```

Results of Complex Multiplication with Nonfinite Values

When an operand of a complex multiplication contains a nonfinite value, the generated code might produce a different result than the result that MATLAB produces. The difference is due to the way that code generation defines complex multiplication. For code generation:

- Multiplication of a complex value by a complex value (a + bi) (c + di) is defined as (ac - bd) + (ad + bc)i. The complete calculation is performed, even when a real or an imaginary part is zero.
- Multiplication of a real value by a complex value c(a + bi) is defined as ca + cbi .

Encoding of Characters in Code Generation

MATLAB represents characters in 16-bit Unicode. The code generator represents characters in an 8-bit codeset that the locale setting determines. Differences in character encoding between MATLAB and code generation have these consequences:

- Code generation of characters with numeric values greater than 255 produces an error.
- For some characters in the range 128–255, it might not be possible to represent the character in the codeset of the locale setting or to convert the character to an equivalent 16-bit Unicode character. Passing characters in this range between MATLAB and generated code can result in errors or different answers.
- For code generation, some toolbox functions accept only 7-bit ASCII characters.
- Casting a character that is not in the 7-bit ASCII codeset to a numeric type, such as double, can produce a different result in the generated code than in MATLAB. As a best practice, for code generation, avoid performing arithmetic with characters.

See Also

More About

- “Locale Setting Concepts for Internationalization”

Array Size Restrictions for Code Generation

For code generation, the maximum number of elements of an array is constrained by the code generator and the target hardware.

For fixed-size arrays and variable-size arrays that use static memory allocation, the maximum number of elements is the smaller of:

- `intmax('int32')`.
- The largest integer that fits in the C `int` data type on the target hardware.

For variable-size arrays that use dynamic memory allocation, the maximum number of elements is the smaller of:

- `intmax('int32')`.
- The largest power of 2 that fits in the C `int` data type on the target hardware.

These restrictions apply even on a 64-bit platform.

For a fixed-size array, if the number of elements exceeds the maximum, the code generator reports an error at compile time. For a variable-size array, if the number of elements exceeds the maximum during simulation, the software reports an error. Generated standalone code cannot report array size violations.

See Also

Code Generation for Constants in Structures and Arrays

The code generator does not recognize constant structure fields or array elements in the following cases:

Fields or elements are assigned inside control constructs

In the following code, the code generator recognizes that the structure fields `s.a` and `s.b` are constants.

```
function y = mystruct()
s.a = 3;
s.b = 5;
y = zeros(s.a,s.b);
```

If any structure field is assigned inside a control construct, the code generator does not recognize the constant fields. This limitation also applies to arrays with constant elements. Consider the following code:

```
function y = mystruct(x)
s.a = 3;
if x > 1
    s.b = 4;
else
    s.b = 5;
end
y = zeros(s.a,s.b);
```

The code generator does not recognize that `s.a` and `s.b` are constant. If variable-sizing is enabled, `y` is treated as a variable-size array. If variable-sizing is disabled, the code generator reports an error.

Constants are assigned to array elements using non-scalar indexing

In the following code, the code generator recognizes that `a(1)` is constant.

```
function y = myarray()
a = zeros(1,3);
a(1) = 20;
y = coder.const(a(1));
```

In the following code, because `a(1)` is assigned using non-scalar indexing, the code generator does not recognize that `a(1)` is constant.

```
function y = myarray()
a = zeros(1,3);
a(1:2) = 20;
y = coder.const(a(1));
```

A function returns a structure or array that has constant and nonconstant elements

For an output structure that has both constant and nonconstant fields, the code generator does not recognize the constant fields. This limitation also applies to arrays that have constant and nonconstant elements. Consider the following code:

```
function y = mystruct_out(x)
s = create_structure(x);
y = coder.const(s.a);
```

```
function s = create_structure(x)
s.a = 10;
s.b = x;
```

Because `create_structure` returns a structure `s` that has one constant field and one nonconstant field, the code generator does not recognize that `s.a` is constant. The `coder.const` call fails because `s.a` is not constant.

Code Generation for Strings

Code generation supports 1-by-1 MATLAB string arrays. Code generation does not support string arrays that have more than one element.

A 1-by-1 string array, called a string scalar, contains one piece of text, represented as a 1-by-n character vector. An example of a string scalar is `"Hello, world"`. For more information about strings, see “Text in String and Character Arrays”.

Limitations

For string scalars, code generation does not support:

- Global variables
- Indexing with curly braces `{}`
- Missing values
- Their use as Simulink signals, parameters, or data store memory

For code generation, limitations that apply to classes apply to strings. See “MATLAB Classes Definition for Code Generation” on page 61-2.

Differences Between Generated Code and MATLAB Code

- Converting a string that contains multiple unary operators to `double` can produce different results between MATLAB and the generated code. Consider this function:

  ```
  function out = foo(op)
  out = double(op + 1);
  end
  ```

  For an input value `"--"`, the function converts the string `"--1"` to `double`. In MATLAB, the answer is `NaN`. In the generated code, the answer is `1`.
- Double conversion for a string with misplaced commas (commas that are not used as thousands separators) can produce different results from MATLAB.

See Also

More About

- “Type Function Arguments” on page 44-45

Code Generation for Sparse Matrices

Sparse matrices provide efficient storage in memory for arrays with many zero elements. Sparse matrices can provide improved performance and reduced memory usage for generated code. Computation time on sparse matrices scales only with the number of operations on nonzero elements.

Functions for creating and manipulating sparse matrices are listed in “Sparse Matrices”. To check if a function is supported for code generation, see the function reference page. Code generation does not support sparse matrix inputs for all functions.

Code Generation Guidelines

Initialize matrices by using sparse constructors to maximize your code efficiency. For example, to construct a 3-by-3 identity matrix, use `speye(3,3)` rather than `sparse(eye(3,3))`.

Indexed assignment into sparse matrices incurs an overhead compared to indexed assignment into full matrices. For example:

```
S = speye(10);
S(7,7) = 42;
```

As in MATLAB, sparse matrices are stored in compressed sparse column format. When you insert a new nonzero element into a sparse matrix, all subsequent nonzero elements must be shifted downward, column by column. These extra manipulations can slow performance.

Code Generation Limitations

Code generation does not support sparse matrices for Simulink signals, parameters, or data store memory. Simulation state save and restore is not supported.

To generate code that uses sparse matrices, dynamic memory allocation must be enabled. To store the changing number of nonzero elements, and their values, sparse matrices use variable-size arrays in the generated code. To change dynamic memory allocation settings, see “Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-79. Because sparse matrices use variable-size arrays for dynamic memory allocation, limitations on “Variable-Size Data” also apply to sparse matrices.

You cannot assign sparse data to data that is not sparse. The generated code uses distinct data type representations for sparse and full matrices. To convert to and from sparse data, use the explicit `sparse` and `full` conversion functions.

You cannot define a sparse matrix with competing size specifications. The code generator fixes the size of the sparse matrix when it produces the corresponding data type definition in C/C++. As an example, the function `foo` causes an error in code generation:

```
function y = foo(n)
%#codegen
if n > 0
    y = sparse(3,2);
else
    y = sparse(4,3);
end
```

Logical indexing into sparse matrices is not supported for code generation. For example, this syntax causes an error:

```
S = magic(3);
S(S > 7) = 42;
```

For sparse matrices, you cannot delete array elements by assigning empty arrays:

```
S(:,2) = [];
```

See Also

`full` | `magic` | `sparse` | `speye`

More About

- “Sparse Matrices”
- “Use Dynamically Allocated C++ Arrays in the Generated Function Interfaces” (MATLAB Coder)

Specify Array Layout in Functions and Classes

You can specialize individual MATLAB functions for row-major layout or column-major layout by inserting `coder.rowMajor` or `coder.columnMajor` calls into the function body. Using these function specializations, you can combine row-major data and column-major data in your generated code. You can also specialize classes for one specific array layout. Function and class specializations allow you to:

- Incrementally modify your code for row-major layout or column-major layout.
- Define array layout boundaries for applications that require different layouts in different components.
- Structure the inheritance of array layout between many different functions and classes.

For MATLAB Coder entry-point (top-level) functions, all inputs and outputs must use the same array layout. In the generated C/C++ code, the entry-point function interface accepts and returns data with the same array layout as the function array layout specification.

Note By default, code generation uses column-major array layout.

Specify Array Layout in a Function

For an example of a specialized function, consider `addMatrixRM`:

```
function [S] = addMatrixRM(A,B)
%#codegen
S = zeros(size(A));
coder.rowMajor; % specify row-major code
for row = 1:size(A,1)
   for col = 1:size(A,2)
       S(row,col) = A(row,col) + B(row,col);
   end
end
```

For MATLAB Coder, you can generate code for `addMatrixRM` by using the `codegen` command.

```
codegen addMatrixRM -args {ones(20,10),ones(20,10)} -config:lib -launchreport
```

Because of the `coder.rowMajor` call, the code generator produces code that uses data stored in row-major layout.

Other functions called from a row-major function or column-major function inherit the same array layout. If a called function has its own distinct `coder.rowMajor` or `coder.columnMajor` call, the local call takes precedence.

You can mix column-major and row-major functions in the same code. The code generator inserts transpose or conversion operations when passing data between row-major and column-major functions. These conversion operations ensure that array elements are stored as required by functions with different array layout specifications. For example, the inputs to a column-major function, called from a row-major function, are converted to column-major layout before being passed to the column-major function.

Query Array Layout of a Function

To query the array layout of a function at compile time, use `coder.isRowMajor` or `coder.isColumnMajor`. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```
function [S] = addMatrixRouted(A,B)
 if coder.isRowMajor
     %execute this code if row-major
     S = addMatrixRM(A,B);
 elseif coder.isColumnMajor
     %execute this code if column-major
     S = addMatrix_OptimizedForColumnMajor(A,B);
 end
```

This function behaves differently depending on whether it is row-major or column-major. When `addMatrixRouted` is row-major, it calls the `addMatrixRM` function, which has efficient memory access for row-major data. When the function is column-major, it calls a version of the `addMatrixRM` function optimized for column-major data.

For example, consider this function definition. The algorithm iterates through the columns in the outer loop and the rows in the inner loop, in contrast to the `addMatrixRM` function.

```
function [S] = addMatrix_OptimizedForColumnMajor(A,B)
%#codegen
S = zeros(size(A));
for col = 1:size(A,2)
   for row = 1:size(A,1)
       S(row,col) = A(row,col) + B(row,col);
   end
end
```

Code generation for this function yields:

```
...
/* column-major layout */
for (col = 0; col < 10; col++) {
  for (row = 0; row < 20; row++) {
     S[row + 20 * col] = A[row + 20 * col] + B[row + 20 * col];
  }
}
...
```

The generated code has a stride length of only one element. Due to the specializing queries, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

Specify Array Layout in a Class

You can specify array layout for a class so that object property variables are stored with a specific array layout. To specify the array layout, place a `coder.rowMajor` or `coder.columnMajor` call in the class constructor. If you assign an object with a specified array layout to the property of another object, the array layout of the assigned object takes precedence.

Consider the row-major class `rowMats` as an example. This class contains matrix properties and a method that consists of an element-wise addition algorithm. The algorithm in the method performs

more efficiently for data stored in row-major layout. By specifying `coder.rowMajor` in the class constructor, the generated code uses row-major layout for the property data.

```
classdef rowMats
    properties (Access = public)
        A;
        B;
        C;
    end
    methods
        function obj = rowMats(A,B)
            coder.rowMajor;
            if nargin == 0
                obj.A = 0;
                obj.B = 0;
                obj.C = 0;
            else
                obj.A = A;
                obj.B = B;
                obj.C = zeros(size(A));
            end
        end
        function obj = add(obj)
            for row = 1:size(obj.A,1)
                for col = 1:size(obj.A,2)
                    obj.C(row,col) = obj.A(row,col) + obj.B(row,col);
                end
            end
        end
    end
end
```

Use the class in a simple function `doMath`. The inputs and outputs of the entry-point function must all use the same array layout.

```
function [out] = doMath(in1,in2)
%#codegen
out = zeros(size(in1));
myMats = rowMats(in1,in2);
myMats = myMats.add;
out = myMats.C;
end
```

For MATLAB Coder, you can generate code by entering:

```
A = rand(20,10);
B = rand(20,10);
cfg = coder.config('lib');
codegen -config cfg doMath -args {A,B} -launchreport
```

With default settings, the code generator assumes that the entry-point function inputs and outputs use column-major layout, because you do not specify row-major layout for the function `doMath`. Therefore, before calling the class constructor, the generated code converts `in1` and `in2` to row-major layout. Similarly, it converts the `doMath` function output back to column-major layout.

When designing a class for a specific array layout, consider:

- If you do not specify the array layout in a class constructor, objects inherit their array layout from the function that calls the class constructor, or from code generation configuration settings.
- You cannot specify the array layout in a nonstatic method by using `coder.rowMajor` or `coder.columnMajor`. Methods use the same array layout as the receiving object. Methods do not inherit the array layout of the function that calls them. For static methods, which are used similarly to ordinary functions, you can specify the array layout in the method.
- If you specify the array layout of a superclass, the subclass inherits this array layout specification. You cannot specify conflicting array layouts between superclasses and subclasses.

See Also

`coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- “Interface with Row-Major Data in MATLAB Function Block” on page 44-173
- “Code Design for Row-Major Array Layout” on page 52-17
- “Code Generation of Matrices and Arrays” (Simulink Coder)

Code Design for Row-Major Array Layout

Outside of code generation, MATLAB uses column-major layout by default. Array layout specifications do not affect self-contained MATLAB code. To test the efficiency of your generated code or your MATLAB Function block, create separate versions with row-major layout and column-major layout. Then, compare their performance.

You can design your MATLAB code to avoid potential inefficiencies related to array layout. Inefficiencies can be caused by:

- Conversions between row-major layout and column-major layout.
- One-dimensional or linear indexing of row-major data.
- Reshaping or rearrangement of row-major data.

Array layout conversions are necessary when you mix row-major and column-major specifications in the same code or model, or when you use linear indexing on data that is stored in row-major. When you simulate a model or generate code for a model that uses column-major, and that contains a MATLAB Function block that uses row-major, then the software converts input data to row-major and output data back to column-major as needed, and vice versa.

Inefficiencies can be caused by functions or algorithms that are less optimized for a given choice of array layout. If a function or algorithm is more efficient for a different layout, you can enforce that layout by embedding it in another function with a `coder.rowMajor` or `coder.columnMajor` call.

Linear Indexing Uses Column-Major Array Layout

The code generator follows MATLAB column-major semantics for linear indexing. For more information on linear indexing in MATLAB, see “Array Indexing”.

To use linear indexing on row-major data, the code generator must first recalculate the data representation in column-major layout. This additional processing can slow performance. To improve code efficiency, avoid using linear indexing on row-major data, or use column-major layout for code that uses linear indexing.

For example, consider the function `sumShiftedProducts`, which accepts a matrix as an input and outputs a scalar value. The function uses linear indexing on the input matrix to sum up the product of each matrix element with an adjacent element. The output value of this operation depends on the order in which the input elements are stored.

```
function mySum = sumShiftedProducts(A)
%#codegen
mySum = 0;
% create linear vector of A elements
B = A(:);
% multiply B by B with elements shifted by one, and take sum
mySum = sum( B.*circshift(B,1) );
end
```

For MATLAB Coder, to generate code that uses row-major layout, enter:

```
codegen -config:mex sumShiftedProducts -args {ones(2,3)} -launchreport -rowmajor
```

For an example input, consider the matrix:

```
D = reshape(1:6,3,2)'
```

which yields:

```
D =
     1     2     3
     4     5     6
```

If you pass this matrix as input to the generated code, the elements of `A` are stored in the order:

```
     1     2     3     4     5     6
```

In contrast, because the vector `B` is obtained by linear indexing, it is stored in the order:

```
     1     4     2     5     3     6
```

The code generator must insert a reshaping operation to rearrange the data from row-major layout for `A` to column-major layout for `B`. This additional operation reduces the efficiency of the function for row-major layout. The inefficiency increases with the size of the array. Because linear indexing always uses column-major layout, the generated code for `sumShiftedProducts` produces the same output result whether generated with row-major layout or column-major layout.

In general, functions that compute indices or subscripts also use linear indexing, and produce results corresponding to data stored in column-major layout. These functions include:

- `ind2sub`
- `sub2ind`
- `colon`

See Also

`coder.ceval` | `coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- “Interface with Row-Major Data in MATLAB Function Block” on page 44-173
- “Specify Array Layout in Functions and Classes” on page 52-13
- “Code Generation of Matrices and Arrays” (Simulink Coder)

Code Generation for Variable-Size Data

Code Generation for Variable-Size Arrays

For code generation, an array dimension is fixed-size or variable-size. If the code generator can determine the size of the dimension and that the size of the dimension does not change, then the dimension is fixed-size. When all dimensions of an array are fixed-size, the array is a fixed-size array. In the following example, `Z` is a fixed-size array.

```
function Z = myfcn()
Z = zeros(1,4);
end
```

The size of the first dimension is 1 and the size of the second dimension is 4.

If the code generator cannot determine the size of a dimension or the code generator determines that the size changes, then the dimension is variable-size. When at least one of its dimensions is variable-size, an array is a variable-size array.

A variable-size dimension is either bounded or unbounded. A bounded dimension has a fixed upper size. An unbounded dimension does not have a fixed upper size.

In the following example, the second dimension of `Z` is bounded, variable-size. It has an upper bound of 16.

```
function s = myfcn(n)
if (n > 0)
    Z = zeros(1,4);
else
    Z = zeros(1,16);
end
s = length(Z);
```

In the following example, if the value of `n` is unknown at compile time, then the second dimension of `Z` is unbounded.

```
function s = myfcn(n)
Z = rand(1,n);
s = sum(Z);
end
```

You can define variable-size arrays by:

- Using constructors, such as `zeros`, with a nonconstant dimension
- Assigning multiple, constant sizes to the same variable before using it
- Declaring all instances of a variable to be variable-size by using `coder.varsize`

For more information, see “Define Variable-Size Data for Code Generation” on page 53-7.

You can control whether variable-size arrays are allowed for code generation. See “Enabling and Disabling Support for Variable-Size Arrays” on page 53-3.

Memory Allocation for Variable-Size Arrays

For fixed-size arrays and variable-size arrays whose size is less than a threshold, the code generator allocates memory statically on the stack. For unbounded, variable-size arrays and variable-size arrays

whose size is greater than or equal to a threshold, the code generator allocates memory dynamically on the heap.

For a MATLAB Function block, you cannot use dynamic memory allocation for:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.

You can control whether dynamic memory allocation is allowed or when it is used for code generation. See “Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-79.

The code generator represents dynamically allocated data as a structure type called `emxArray`. The code generator generates utility functions that create and interact with emxArrays. If you use Embedded Coder, you can customize the generated identifiers for the `emxArray` types and utility functions. See “Identifier Format Control” (Embedded Coder).

Enabling and Disabling Support for Variable-Size Arrays

By default, for MATLAB Function blocks, support for variable-size arrays is enabled. To disable this support:

1. In the MATLAB Function Block Editor, select **Edit Data**.
2. Clear the **Support variable-size arrays** check box.

Variable-Size Arrays in a MATLAB Function Report

You can tell whether an array is fixed-size or variable-size by looking at the **Size** column of the **Variables** tab in a MATLAB Function Report.

Name	Type	Size	Class
y	Output	1 × 1	double
n	Input	1 × 1	double
x	Local	1 × :?	double

A colon (:) indicates that a dimension is variable-size. A question mark (?) indicates that the size is unbounded. For example, a size of 1-by-:? indicates that the size of the first dimension is fixed-size 1 and the size of the second dimension is unbounded, variable-size. Italics indicates that your code specifies that an array is variable-size, but the code generator determined that it does not change size.

ALL MESSAGES (0) | VARIABLES

Name	Type	Size	Class
y	Output	1 × 2	double
n	Input	1 × 1	double
Z	Local	*1 × 4*	double

See Also

More About

- “Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-79
- “Specify Upper Bounds for Variable-Size Arrays” on page 53-5
- “Define Variable-Size Data for Code Generation” on page 53-7
- “Use Dynamic Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 44-81

Specify Upper Bounds for Variable-Size Arrays

Specify upper bounds for an array when:

- Dynamic memory allocation is disabled.

  If dynamic memory allocation is disabled, you must specify upper bounds for all arrays.
- You do not want the code generator to use dynamic memory allocation for the array.

  Specify upper bounds that result in an array size (in bytes) that is less than the dynamic memory allocation threshold.

Specify Upper Bounds for MATLAB Function Block Inputs and Outputs

See “Declare Variable-Size Inputs and Outputs” on page 44-72.

Specify Upper Bounds for Local Variables

When using static allocation, the code generator uses a sophisticated analysis to calculate the upper bounds of local data. However, when the analysis fails to detect an upper bound or calculates an upper bound that is not precise enough for your application, you must specify upper bounds explicitly for local variables.

Constrain the Value of Variables That Specify the Dimensions of Variable-Size Arrays

To constrain the value of variables that specify the dimensions of variable-size arrays, use the `assert` function with relational operators. For example:

```
function y = dim_need_bound(n) %#codegen
assert (n <= 5);
L= ones(n,n);
M = zeros(n,n);
M = [L; M];
y = M;
```

This `assert` statement constrains input `n` to a maximum size of 5. `L` is variable-size with upper bounds of 5 in each dimension. `M` is variable-size with an upper bound of 10 in the first dimension and 5 in the second dimension.

Specify the Upper Bounds for All Instances of a Local Variable

To specify the upper bounds for all instances of a local variable in a function, use the `coder.varsize` function. For example:

```
function Y = example_bounds1(u) %#codegen
Y = [1 2 3 4 5];
coder.varsize('Y',[1 10]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

The second argument of `coder.varsize` specifies the upper bound for each instance of the variable specified in the first argument. In this example, the argument `[1 10]` indicates that for every instance of `Y`:

- The first dimension is fixed at size 1.
- The second dimension can grow to an upper bound of 10.

See Also

`coder.varsize`

More About

- “Code Generation for Variable-Size Arrays” on page 53-2
- “Define Variable-Size Data for Code Generation” on page 53-7

Define Variable-Size Data for Code Generation

For code generation, before using variables in operations or returning them as outputs, you must assign them a specific class, size, and complexity. Generally, after the initial assignment, you cannot reassign variable properties. Therefore, after assigning a fixed size to a variable or structure field, attempts to grow the variable or structure field might cause a compilation error. In these cases, you must explicitly define the data as variable-size by using one of these methods.

Method	See
Assign the data from a variable-size matrix constructor such as: • `ones` • `zeros` • `repmat`	"Use a Matrix Constructor with Nonconstant Dimensions" on page 53-7
Assign multiple, constant sizes to the same variable before using (reading) the variable.	"Assign Multiple Sizes to the Same Variable" on page 53-7
Define all instances of a variable to be variable-size.	"Define Variable-Size Data Explicitly by Using coder.varsize" on page 53-8

Use a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For example:

```
function s = var_by_assign(u) %#codegen
y = ones(3,u);
s = numel(y);
```

If you are not using dynamic memory allocation, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function s = var_by_assign(u) %#codegen
assert (u < 20);
y = ones(3,u);
s = numel(y);
```

Assign Multiple Sizes to the Same Variable

Before you use (read) a variable in your code, you can make it variable-size by assigning multiple, constant sizes to it. When the code generator uses static allocation on the stack, it infers the upper bounds from the largest size specified for each dimension. When you assign the same size to a given dimension across all assignments, the code generator assumes that the dimension is fixed at that size. The assignments can specify different shapes and sizes.

When the code generator uses dynamic memory allocation, it does not check for upper bounds. It assumes that the variable-size data is unbounded.

Inferring Upper Bounds from Multiple Definitions with Different Shapes

```
function s = var_by_multiassign(u) %#codegen
if (u > 0)
```

```
    y = ones(3,4,5);
else
    y = zeros(3,1);
end
s = numel(y);
```

When the code generator uses static allocation, it infers that `y` is a matrix with three dimensions:

- The first dimension is fixed at size 3
- The second dimension is variable-size with an upper bound of 4
- The third dimension is variable-size with an upper bound of 5

When the code generator uses dynamic allocation, it analyzes the dimensions of `y` differently:

- The first dimension is fixed at size 3.
- The second and third dimensions are unbounded.

Define Variable-Size Data Explicitly by Using coder.varsize

To explicitly define variable-size data, use the function `coder.varsize`. Optionally, you can also specify which dimensions vary along with their upper bounds. For example:

- Define `B` as a variable-size 2-dimensional array, where each dimension has an upper bound of 64.

  ```
  coder.varsize('B', [64 64]);
  ```

- Define `B` as a variable-size array:

  ```
  coder.varsize('B');
  ```

  When you supply only the first argument, `coder.varsize` assumes that all dimensions of `B` can vary and that the upper bound is `size(B)`.

If a MATLAB Function block input or output signal is variable-size, in the Ports and Data Manager, you must specify that the signal is variable-size. You must also provide the upper bounds. You do not have to use `coder.varsize` with the corresponding input or output variable inside the MATLAB Function block. However, if you specify upper bounds with `coder.varsize`, they must match the upper bounds in the Ports and Data Manager.

Specify Which Dimensions Vary

You can use the function `coder.varsize` to specify which dimensions vary. For example, the following statement defines `B` as an array whose first dimension is fixed at 2, but whose second dimension can grow to a size of 16:

```
coder.varsize('B',[2, 16],[0 1])
```

.

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size. Dimensions that correspond to ones or `true` vary in size. `coder.varsize` usually treats dimensions of size 1 as fixed. See “Define Variable-Size Matrices with Singleton Dimensions” on page 53-9.

For an input or output signal, if you specify the upper bounds with `coder.varsize` inside the MATLAB Function block, they must match the upper bounds in the Ports and Data Manager.

Allow a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix `Y` with fixed 2-by-2 dimensions before the first use (where the statement `Y = Y + u` reads from `Y`). However, `coder.varsize` defines `Y` as a variable-size matrix, allowing it to change size based on decision logic in the `else` clause:

```
function Y = var_by_if(u) %#codegen
if (u > 0)
    Y = zeros(2,2);
    coder.varsize('Y');
    if (u < 10)
        Y = Y + u;
    end
else
    Y = zeros(5,5);
end
```

Without `coder.varsize`, the code generator infers `Y` to be a fixed-size, 2-by-2 matrix. It generates a size mismatch error.

Define Variable-Size Matrices with Singleton Dimensions

A singleton dimension is a dimension for which `size(A,dim) = 1`. Singleton dimensions are fixed in size when:

- You specify a dimension with an upper bound of 1 in `coder.varsize` expressions.

  For example, in this function, `Y` behaves like a vector with one variable-size dimension:

  ```
  function Y = dim_singleton(u) %#codegen
  Y = [1 2];
  coder.varsize('Y', [1 10]);
  if (u > 0)
      Y = [Y 3];
  else
      Y = [Y u];
  end
  ```

- You initialize variable-size data with singleton dimensions by using matrix constructor expressions or matrix functions.

  For example, in this function, `X` and `Y` behave like vectors where only their second dimensions are variable-size.

  ```
  function [X,Y] = dim_singleton_vects(u) %#codegen
  Y = ones(1,3);
  X = [1 4];
  coder.varsize('Y','X');
  if (u > 0)
      Y = [Y u];
  else
      X = [X u];
  end
  ```

You can override this behavior by using `coder.varsize` to specify explicitly that singleton dimensions vary. For example:

```
function Y = dim_singleton_vary(u) %#codegen
Y = [1 2];
coder.varsize('Y', [1 10], [1 1]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

In this example, the third argument of `coder.varsize` is a vector of ones, indicating that each dimension of `Y` varies in size.

Define Variable-Size Structure Fields

To define structure fields as variable-size arrays, use a colon (`:`) as the index expression. The colon (`:`) indicates that all elements of the array are variable-size. For example:

```
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 expression `coder.varsize('data(:).values')` defines the field `values` inside each element of matrix `data` to be variable-size.

Here are other examples:

- `coder.varsize('data.A(:).B')`

  In this example, `data` is a scalar variable that contains matrix `A`. Each element of matrix `A` contains a variable-size field `B`.

- `coder.varsize('data(:).A(:).B')`

  This expression defines field `B` inside each element of matrix `A` inside each element of matrix `data` to be variable-size.

See Also

`coder.varsize`

More About

- “Code Generation for Variable-Size Arrays” on page 53-2
- “Specify Upper Bounds for Variable-Size Arrays” on page 53-5

Diagnose and Fix Variable-Size Data Errors

In this section...

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

You cannot assign variable-size matrices to fixed-size matrices in generated code. Consider this example:

```
function Y = example_mismatch1(n) %#codegen
assert(n < 10);
B = ones(n,n);
A = magic(3);
A(1) = mean(A(:));
if (n == 3)
    A = B;
end
Y = A;
```

Compiling this function produces this error:

```
??? Dimension 1 is fixed on the left-hand side
but varies on the right ...
```

There are several ways to fix this error:

- Allow matrix `A` to grow by adding the `coder.varsize` construct:

  ```
  function Y = example_mismatch1_fix1(n) %#codegen
  coder.varsize('A');
  assert(n < 10);
  B = ones(n,n);
  A = magic(3);
  A(1) = mean(A(:));
  if (n == 3)
      A = B;
  end
  Y = A;
  ```

- Explicitly restrict the size of matrix `B` to 3-by-3 by modifying the `assert` statement:

  ```
  function Y = example_mismatch1_fix2(n) %#codegen
  coder.varsize('A');
  assert(n == 3)
  B = ones(n,n);
  A = magic(3);
  A(1) = mean(A(:));
  if (n == 3)
      A = B;
  ```

```
end
Y = A;
```

- Use explicit indexing to make `B` the same size as `A`:

```
function Y = example_mismatch1_fix3(n) %#codegen
assert(n < 10);
B = ones(n,n);
A = magic(3);
A(1) = mean(A(:));
if (n == 3)
    A = B(1:3, 1:3);
end
Y = A;
```

Empty Matrix Reshaped to Match Variable-Size Specification

If you assign an empty matrix `[]` to variable-size data, MATLAB might silently reshape the data in generated code to match a `coder.varsize` specification. For example:

```
function Y = test(u) %#codegen
Y = [];
coder.varsize('Y', [1 10]);
if u < 0
    Y = [Y u];
end
```

In this example, `coder.varsize` defines `Y` as a column vector of up to 10 elements, so its first dimension is fixed at size 1. The statement `Y = []` designates the first dimension of `Y` as 0, creating a mismatch. The right hand side of the assignment is an empty matrix and the left hand side is a variable-size vector. In this case, MATLAB reshapes the empty matrix `Y = []` in generated code to `Y = zeros(1,0)` so it matches the `coder.varsize` specification.

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

You can define variable-size data by assigning a variable to a matrix with nonconstant dimensions. For example:

```
function y = dims_vary(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

However, compiling this function generates an error because you did not specify an upper bound for `u`.

To fix the problem, add an `assert` statement before the first use of `u`:

```
function y = dims_vary_fix(u) %#codegen
assert (u < 20);
if (u > 0)
```

```
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

Incompatibility with MATLAB for Scalar Expansion

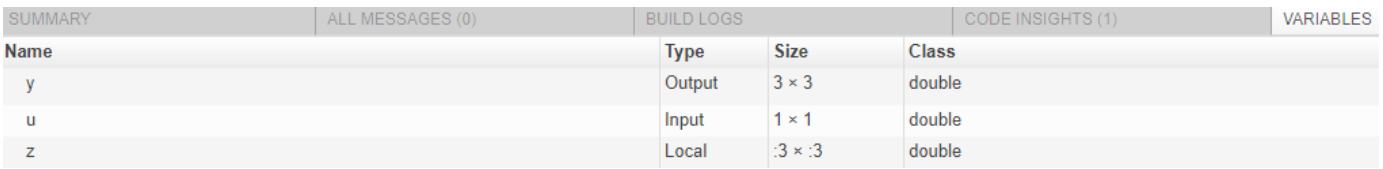
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. If one operand is a scalar and the other is not, scalar expansion applies the scalar to every element of the other operand.

During code generation, scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

Consider this function:

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z;
```

When you generate code for this function, the code generator determines that `z` is variable size with an upper bound of `3`.



If you run the MEX function with `u` equal to 0 or 1, the generated code does not perform scalar expansion, even though `z` is scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

```
scalar_exp_test_err1_mex(0)
Subscripted assignment dimension mismatch: [9] ~= [1].

Error in scalar_exp_test_err1 (line 11)
y(:) = z;
```

To avoid this issue, use indexing to force `z` to be a scalar value.

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z(1);
```

Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function can return a different result in generated code than in MATLAB. In generated code, `size(A)` returns a fixed-length output because it does not drop trailing singleton dimensions of variable-size N-D arrays. By contrast, `size(A)` in MATLAB returns a variable-length output because it drops trailing singleton dimensions.

For example, if the shape of array `A` is `:?x:?x:?` and `size(A,3)==1`, `size(A)` returns:

- Three-element vector in generated code
- Two-element vector in MATLAB code

Workarounds

If your application requires generated code to return the same size of variable-size N-D arrays as MATLAB code, consider one of these workarounds:

- Use the two-argument form of `size`.

  For example, `size(A,n)` returns the same answer in generated code and MATLAB code.
- Rewrite `size(A)`:

  ```
  B = size(A);
  X = B(1:ndims(A));
  ```

  This version returns `X` with a variable-length output. However, you cannot pass a variable-size `X` to matrix constructors such as `zeros` that require a fixed-size argument.

Incompatibility with MATLAB in Determining Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. The size might be `1x0` or `0x1` in generated code, but `0x0` in MATLAB. Therefore, you should not write code that relies on the specific size of empty matrices.

For example, consider the following code:

```
function y = foo(n) %#codegen
x = [];
i = 0;
while (i < 10)
    x = [5 x];
    i = i + 1;
end
if n > 0
    x = [];
end
y = size(x);
end
```

Concatenation requires its operands to match on the size of the dimension that is not being concatenated. In the preceding concatenation, the scalar value has size `1x1` and `x` has size `0x0`. To support this use case, the code generator determines the size for `x` as `[1 x :?]`. Because there is another assignment `x = []` after the concatenation, the size of `x` in the generated code is `1x0` instead of `0x0`.

This behavior persists while determining the size of empty character vectors which are denoted as `''`. For example, consider the following code:

```
function out = string_size
out = size('');
end
```

Here, the value of `out` might be `1x0` or `0x1` in generated code, but `0x0` in MATLAB.

For incompatibilities with MATLAB in determining the size of an empty array that results from deleting elements of an array, see “Size of Empty Array That Results from Deleting Elements of an Array” on page 48-13.

Workaround

If your application checks whether a matrix is empty, use one of these workarounds:

- Rewrite your code to use the `isempty` function instead of the `size` function.
- Instead of using `x=[]` to create empty arrays, create empty arrays of a specific size using `zeros`. For example:

  ```
  function y = test_empty(n) %#codegen
  x = zeros(1,0);
  i=0;
  while (i < 10)
      x = [5 x];
      i = i + 1;
  end
  if n > 0
  ```

```
    x = zeros(1,0);
end
y=size(x);
end
```

Incompatibility with MATLAB in Determining Class of Empty Arrays

The class of an empty array in generated code can be different from its class in MATLAB source code. Therefore, do not write code that relies on the class of empty matrices.

For example, consider the following code:

```
function y = fun(n)
x = [];
if n > 1
    x = ['a' x];
end
y=class(x);
end
```

`fun(0)` returns `double` in MATLAB, but `char` in the generated code. When the statement `n > 1` is false, MATLAB does not execute `x = ['a' x]`. The class of `x` is `double`, the class of the empty array. However, the code generator considers all execution paths. It determines that based on the statement `x = ['a' x]`, the class of `x` is `char`.

Workaround

Instead of using `x=[]` to create an empty array, create an empty array of a specific class. For example, use `blanks(0)` to create an empty array of characters.

```
function y = fun(n)
x = blanks(0);
if n > 1
    x = ['a' x];
end
y=class(x);
end
```

Incompatibility with MATLAB in Matrix-Matrix Indexing

In matrix-matrix indexing, you use one matrix to index into another matrix. In MATLAB, the general rule for matrix-matrix indexing is that the size and orientation of the result match the size and orientation of the index matrix. For example, if `A` and `B` are matrices, `size(A(B))` equals `size(B)`. When `A` and `B` are vectors, MATLAB applies a special rule. The special vector-vector indexing rule is that the orientation of the result is the orientation of the data matrix. For example, i`A` is 1-by-5 and `B` is 3-by-1, then `A(B)` is 1-by-3.

The code generator applies the same matrix-matrix indexing rules as MATLAB. If `A` and `B` are variable-size matrices, to apply the matrix-matrix indexing rules, the code generator assumes that the `size(A(B))` equals `size(B)`. If, at run time, `A` and `B` become vectors and have different orientations, then the assumption is incorrect. Therefore, when run-time error checks are enabled, an error can occur.

To avoid this issue, force your data to be a vector by using the colon operator for indexing. For example, suppose that your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing.

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

The indexing in the first branch specifies that `C` and `B(:)` are compile-time vectors. Therefore, the code generator applies the indexing rule for indexing one vector with another vector. The orientation of the result is the orientation of the data vector, `C`.

Incompatibility with MATLAB in Vector-Vector Indexing

In MATLAB, the special rule for vector-vector indexing is that the orientation of the result is the orientation of the data vector. For example, if `A` is 1-by-5 and `B` is 3-by-1, then `A(B)` is 1-by-3. If, however, the data vector `A` is a scalar, then the orientation of `A(B)` is the orientation of the index vector `B`.

The code generator applies the same vector-vector indexing rules as MATLAB. If `A` and `B` are variable-size vectors, to apply the indexing rules, the code generator assumes that the orientation of `B` matches the orientation of `A`. At run time, if `A` is scalar and the orientation of `A` and `B` do not match, then the assumption is incorrect. Therefore, when run-time error checks are enabled, a run-time error can occur.

To avoid this issue, make the orientations of the vectors match. Alternatively, index single elements by specifying the row and column. For example, `A(row, column)`.

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

The following limitation applies to matrix indexing operations for code generation:

- Initialization of the following style:

  ```
  for i = 1:10
      M(i) = 5;
  end
  ```

  In this case, the size of `M` changes as the loop is executed. Code generation does not support increasing the size of an array over time.

  For code generation, preallocate `M`.

  ```
  M = zeros(1,10);
  for i = 1:10
      M(i) = 5;
  end
  ```

The following limitation applies to matrix indexing operations for code generation when dynamic memory allocation is disabled:

- `M(i:j)` where `i` and `j` change in a loop

  During code generation, memory is not dynamically allocated for the size of the expressions that change as the program executes. To implement this behavior, use `for`-loops as shown:

```
...
M = ones(10,10);
for i=1:10
    for j = i:10
        M(i,j) = 2*M(i,j);
    end
end
...
```

  Note The matrix `M` must be defined before entering the loop.

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

For code generation, when you concatenate variable-size arrays, the dimensions that are not being concatenated must match exactly.

Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements

Suppose that:

- `c` is a variable-size cell array.
- You access the contents of `c` by using curly braces. For example, `c{2:4}`.
- You include the results in concatenation. For example, `[a c{2:4} b]`.
- `c{I}` returns no elements. Either `c` is empty or the indexing inside the curly braces produces an empty result.

For these conditions, MATLAB omits `c{I}` from the concatenation. For example, `[a c{I} b]` becomes `[a b]`. The code generator treats `c{I}` as the empty array `[c{I}]`. The concatenation becomes `[...[c{i}]...]`. This concatenation then omits the array `[c{I}]`. So that the properties of `[c{I}]` are compatible with the concatenation `[...[c{i}]...]`, the code generator assigns the class, size, and complexity of `[c{I}]` according to these rules:

- The class and complexity are the same as the base type of the cell array.
- The size of the second dimension is always 0.
- For the rest of the dimensions, the size of `Ni` depends on whether the corresponding dimension in the base type is fixed or variable size.
  - If the corresponding dimension in the base type is variable size, the dimension has size 0 in the result.
  - If the corresponding dimension in the base type is fixed size, the dimension has that size in the result.

Suppose that `c` has a base type with class `int8` and size `:10x7x8x:?`. In the generated code, the class of `[c{I}]` is `int8`. The size of `[c{I}]` is `0x0x8x0`. The second dimension is 0. The first and last dimensions are 0 because those dimensions are variable size in the base type. The third dimension is 8 because the size of the third dimension of the base type is a fixed size 8.

Inside concatenation, if curly-brace indexing of a variable-size cell array returns no elements, the generated code can have the following differences from MATLAB:

- The class of `[...c{i}...]` in the generated code can differ from the class in MATLAB.

  When `c{I}` returns no elements, MATLAB removes `c{I}` from the concatenation. Therefore, `c{I}` does not affect the class of the result. MATLAB determines the class of the result based on the classes of the remaining arrays, according to a precedence of classes. See “Valid Combinations of Unlike Classes”. In the generated code, the class of `[c{I}]` affects the class of the result of the overall concatenation `[...[c{I}]...]` because the code generator treats `c{I}` as `[c{I}]`. The previously described rules determine the class of `[c{I}]`.
- In the generated code, the size of `[c{I}]` can differ from the size in MATLAB.

  In MATLAB, the concatenation `[c{I}]` is a 0x0 double. In the generated code, the previously described rules determine the size of `[c{I}]`.

Variable-Sizing Restrictions for Code Generation of Toolbox Functions

In this section...

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Restrictions for Variable-Size Data” on page 53-22.

Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape `1x:n` or `:nx1` (one dimension is variable sized and the other is fixed at size 1). Other shapes are not permitted, even if they are vectors at run time.

Automatic dimension restriction

This restriction applies to functions that take the working dimension (the dimension along which to operate) as input. In MATLAB and in code generation, if you do not supply the working dimension, the function selects it. In MATLAB, the function selects the first dimension whose size does not equal 1. For code generation, the function selects the first dimension that has a variable size or that has a fixed size that does not equal 1. If the working dimension has a variable size and it becomes 1 at run time, then the working dimension is different from the working dimension in MATLAB. Therefore, when run-time error checks are enabled, an error can occur.

For example, suppose that `X` is a variable-size matrix with dimensions `1x:3x:5`. In the generated code, `sum(X)` behaves like `sum(X,2)`. In MATLAB, `sum(X)` behaves like `sum(X,2)` unless `size(X,2)` is 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`.

To avoid this issue, specify the intended working dimension explicitly as a constant value. For example, `sum(X,2)`.

Array-to-vector restriction

The function issues an error when a variable-size array that is not a variable-length vector assumes the shape of a vector at run time. To avoid the issue, specify the input explicitly as a variable-length vector instead of a variable-size array.

Array-to-scalar restriction

The function issues an error if a variable-size array assumes a scalar value at run time. To avoid this issue, specify scalars as fixed size.

Toolbox Functions with Restrictions for Variable-Size Data

The following table list functions that have code generation restrictions for variable-size data. For additional restrictions for these functions, and restrictions for all functions and objects supported for

code generation, see “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder).

Function	Restrictions for Variable-Size Data
`all`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`any`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`cat`	• Dimension argument must be a constant.
`conv`	• See “Variable-length vector restriction” on page 53-22. • Input vectors must have the same orientation, either both row vectors or both column vectors.
`cov`	• For `cov(X)`, see “Array-to-vector restriction” on page 53-22.
`cross`	• Variable-size array inputs that become vectors at run time must have the same orientation.
`deconv`	• For both arguments, see “Variable-length vector restriction” on page 53-22.
`detrend`	• For first argument for row vectors only, see “Array-to-vector restriction” on page 53-22.
`diag`	• See “Array-to-vector restriction” on page 53-22.
`diff`	• See “Automatic dimension restriction” on page 53-22. • Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, `diff(x,2,1)` works but `diff(x,5,1)` generates a run-time error.
`fft`	• See “Automatic dimension restriction” on page 53-22.
`filter`	• For first and second arguments, see “Variable-length vector restriction” on page 53-22. • See “Automatic dimension restriction” on page 53-22.
`hist`	• For second argument, see “Variable-length vector restriction” on page 53-22. • For second input argument, see “Array-to-scalar restriction” on page 53-22.
`histc`	• See “Automatic dimension restriction” on page 53-22.
`ifft`	• See “Automatic dimension restriction” on page 53-22.
`ind2sub`	• First input (the size vector input) must be fixed size.
`interp1`	• For the `xq` input, see “Array-to-vector restriction” on page 53-22. • If `v` becomes a row vector at run time, the array to vector restriction on page 53-22 applies. If `v` becomes a column vector at run time, this restriction does not apply.

Function	Restrictions for Variable-Size Data
`ipermute`	• Order input must be fixed size.
`issorted`	• See “Automatic dimension restriction” on page 53-22.
`magic`	• Argument must be a constant. • Output can be fixed-size matrices only.
`max`	• See “Automatic dimension restriction” on page 53-22.
`maxk`	• See “Automatic dimension restriction” on page 53-22.
`mean`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`median`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`min`	• See “Automatic dimension restriction” on page 53-22.
`mink`	• See “Automatic dimension restriction” on page 53-22.
`mode`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`mtimes`	Consider the multiplication `A*B`. If the code generator is aware that `A` is scalar and `B` is a matrix, the code generator produces code for scalar-matrix multiplication. However, if the code generator is aware that `A` and `B` are variable-size matrices, it produces code for a general matrix multiplication. At run time, if `A` turns out to be scalar, the generated code does not change its behavior. Therefore, when run-time error checks are enabled, a size mismatch error can occur.
`nchoosek`	• The second input, `k`, must be a fixed-size scalar. • The second input, `k`, must be a constant for static allocation.. • You cannot create a variable-size array by passing in a variable, `k`, .
`permute`	• Order input must be fixed-size.
`planerot`	• Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.
`poly`	• See “Variable-length vector restriction” on page 53-22.
`polyfit`	• For first and second arguments, see “Variable-length vector restriction” on page 53-22.
`prod`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions for Variable-Size Data
`rand`	• For an upper-bounded variable `N`, `rand(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. • For an upper-bounded variable `N`, `rand([1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`randi`	• For an upper-bounded variable `N`, `randi(imax,1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. • For an upper-bounded variable `N`, `randi(imax,[1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`randn`	• For an upper-bounded variable `N`, `randn(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. • For an upper-bounded variable `N`, `randn([1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`reshape`	• If the input is a variable-size array and the output array has at least one fixed-length dimension, do not specify the output dimension sizes in a size vector `sz`. Instead, specify the output dimension sizes as scalar values, `sz1,...,szN`. Specify fixed-size dimensions as constants. • When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.
`roots`	• See “Variable-length vector restriction” on page 53-22.
`shiftdim`	• If you do not supply the second argument, the number of shifts is determined at compilation time by the upper bounds of the dimension sizes. Therefore, at run time the number of shifts is constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must have the same number of dimensions when you supply a positive number of shifts.
`sort`	• See “Automatic dimension restriction” on page 53-22.
`std`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
`sub2ind`	• First input (the size vector input) must be fixed size.
`sum`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions for Variable-Size Data
`trapz`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`typecast`	• See “Variable-length vector restriction” on page 53-22 on first argument.
`var`	• See “Automatic dimension restriction” on page 53-22. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
`vecnorm`	• See “Automatic dimension restriction” on page 53-22.

Code Generation for MATLAB Structures

Structure Definition for Code Generation

To generate efficient standalone code for structures, you must define and use structures differently than you normally would when running your code in the MATLAB environment:

What's Different	More Information
Use a restricted set of operations.	“Structure Operations Allowed for Code Generation” on page 54-3
Observe restrictions on properties and values of scalar structures.	“Define Scalar Structures for Code Generation” on page 54-4
Make structures uniform in arrays.	“Define Arrays of Structures for Code Generation” on page 54-6
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 44-62
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 44-66

Structure Operations Allowed for Code Generation

To generate efficient standalone code for MATLAB structures, you are restricted to the following operations:

- Index structure fields using dot notation
- Define primary function inputs as structures
- Pass structures to local functions

Define Scalar Structures for Code Generation

In this section...

Restrictions When Defining Scalar Structures by Assignment

When you define a scalar structure by assigning a variable to a preexisting structure, you do not need to define the variable before the assignment. However, if you already defined that variable, it must have the same class, size, and complexity as the structure you assign to it. In the following example, `p` is defined as a structure that has the same properties as the predefined structure `S`:

```
...
S = struct('a',  0, 'b',  1, 'c',  2);
p = S;
...
```

Adding Fields in Consistent Order on Each Control Flow Path

When you create a structure, you must add fields in the same order on each control flow path. For example, the following code generates a compiler error because it adds the fields of structure `x` in a different order in each `if` statement clause:

```
function y = fcn(u) %#codegen
if u > 0
   x.a = 10;
   x.b = 20;
else
   x.b = 30;  % Generates an error (on variable x)
   x.a = 40;
end
y = x.a + x.b;
```

In this example, the assignment to `x.a` comes before `x.b` in the first `if` statement clause, but the assignments appear in reverse order in the `else` clause. Here is the corrected code:

```
function y = fcn(u) %#codegen
if u > 0
   x.a = 10;
   x.b = 20;
else
   x.a = 40;
   x.b = 30;
end
y = x.a + x.b;
```

Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform the following operations on the structure:

- Reading from the structure
- Indexing into the structure array
- Passing the structure to a function

For example, consider this code:

```
...
x.c = 10; % Defines structure and creates field c
y = x; % Reads from structure
x.d = 20; % Generates an error
...
```

In this example, the attempt to add a new field `d` after reading from structure `x` generates an error.

This restriction extends across the structure hierarchy. For example, you cannot add a field to a structure after operating on one of its fields or nested structures, as in this example:

```
function y = fcn(u) %#codegen

x.c = 10;
y = x.c;
x.d = 20; % Generates an error
```

In this example, the attempt to add a new field `d` to structure `x` after reading from the structure's field `c` generates an error.

Define Arrays of Structures for Code Generation

In this section...

Ensuring Consistency of Fields

For code generation, when you create an array of MATLAB structures, corresponding fields in the array elements must have the same size, type, and complexity.

Once you have created the array of structures, you can make the structure fields variable-size by using `coder.varsize`. See “Declare Variable-Size Structure Fields” (MATLAB Coder).

Using repmat to Define an Array of Structures with Consistent Field Properties

You can create an array of structures from a scalar structure by using the MATLAB `repmat` function, which replicates and tiles an existing scalar structure:

1. Create a scalar structure, as described in “Define Scalar Structures for Code Generation” on page 54-4.
2. Call `repmat`, passing the scalar structure and the dimensions of the array.
3. Assign values to each structure using standard array indexing and structure dot notation.

For example, the following code creates `X`, a 1-by-3 array of scalar structures. Each element of the array is defined by the structure `s`, which has two fields, `a` and `b`:

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,3);
X(1).a = 1;
X(2).a = 2;
X(3).a = 3;
X(1).b = 4;
X(2).b = 5;
X(3).b = 6;
...
```

Defining an Array of Structures by Using struct

To create an array of structures using the `struct` function, specify the field value arguments as cell arrays. Each cell array element is the value of the field in the corresponding structure array element. For code generation, corresponding fields in the structures must have the same type. Therefore, the elements in a cell array of field values must have the same type.

For example, the following code creates a 1-by-3 structure array. For each structure in the array of structures, `a` has type `double` and `b` has type `char`.

```
s = struct('a', {1 2 3}, 'b', {'a' 'b' 'c'});
```

Defining an Array of Structures Using Concatenation

To create a small array of structures, you can use the concatenation operator, square brackets (`[ ]`), to join one or more structures into an array. See “Creating, Concatenating, and Expanding Matrices”. For code generation, the structures that you concatenate must have the same size, class, and complexity.

For example, the following code uses concatenation and a local function to create the elements of a 1-by-3 structure array:

```
...
W = [ sab(1,2) sab(2,3) sab(4,5) ];

function s = sab(a,b)
  s.a = a;
  s.b = b;
...
```

Index Substructures and Fields

Use these guidelines when indexing substructures and fields for code generation:

Reference substructure field values individually using dot notation

For example, the following MATLAB code uses dot notation to index fields and substructures:

```
...
substruct1.a1 = 15.2;
substruct1.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                  'ele3',substruct1);

substruct2 = mystruct;
substruct2.ele3.a2 = 2*(substruct1.a2);
...
```

The generated code indexes elements of the structures in this example by resolving symbols as follows:

Dot Notation	Symbol Resolution
`substruct1.a1`	Field `a1` of local structure `substruct1`
`substruct2.ele3.a1`	Value of field `a1` of field `ele3`, a substructure of local structure `substruct2`
`substruct2.ele3.a2(1,1)`	Value in row 1, column 1 of field `a2` of field `ele3`, a substructure of local structure `substruct2`

Reference field values individually in structure arrays

To reference the value of a field in a structure array, you must index into the array to the structure of interest and then reference that structure's field individually using dot notation, as in this example:

```
...
y = X(1).a % Extracts the value of field a
           % of the first structure in array X
...
```

To reference all the values of a particular field for each structure in an array, use this notation in a `for` loop, as in this example:

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,5);
for i = 1:5
    X(i).a = i;
    X(i).b = i+1;
end
```

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 54-6 for more information.

Do not reference fields dynamically

You cannot reference fields in a structure by using dynamic names, which express the field as a variable expression that MATLAB evaluates at run time (see “Generate Field Names from Variables”).

Assign Values to Structures and Fields

When assigning values to a structure, substructure, or field for code generation, use these guidelines:

Field properties must be consistent across structure-to-structure assignments

If:	Then:
Assigning one structure to another structure.	Define each structure with the same number, type, and size of fields.
Assigning one structure to a substructure of a different structure and vice versa.	Define the structure with the same number, type, and size of fields as the substructure.
Assigning an element of one structure to an element of another structure.	The elements must have the same type and size.

For structures with constant fields, do not assign field values inside control flow constructs

In the following code, the code generator recognizes that the structure fields `s.a` and `s.b` are constants.

```
function y = mystruct()
s.a = 3;
s.b = 5;
y = zeros(s.a,s.b);
```

If a field of a structure is assigned inside a control flow construct, the code generator does not recognize that `s.a` and `s.b` are constant. Consider the following code:

```
function y = mystruct(x)
s.a = 3;
if x > 1
    s.b = 4;
else
    s.b = 5;
end
y = zeros(s.a,s.b);
```

If variable-sizing is enabled, `y` is treated as a variable-size array. If variable-sizing is disabled, `y`, the code generator reports an error.

Do not assign mxArrays to structures

You cannot assign `mxArrays` to structure elements; convert `mxArrays` to known types before code generation (see “Working with mxArrays” on page 64-13).

Do not assign handle classes or sparse arrays to global structure variables

Global structure variables cannot contain handle objects or sparse arrays.

Pass Large Structures as Input Parameters

If you generate a MEX function for a MATLAB function that takes a large structure as an input parameter, for example, a structure containing fields that are matrices, the MEX function might fail to load. This load failure occurs because, when you generate a MEX function from a MATLAB function that has input parameters, the code generator allocates memory for these input parameters on the stack. To avoid this issue, pass the structure by reference to the MATLAB function. For example, if the original function signature is:

```
y = foo(a, S)
```

where `S` is the structure input, rewrite the function to:

```
[y, S] = foo(a, S)
```

Code Generation for Cell Arrays

Code Generation for Cell Arrays

When you generate code from MATLAB code that contains cell arrays, the code generator classifies the cell arrays as homogeneous or heterogeneous. This classification determines how a cell array is represented in the generated code. It also determines how you can use the cell array in MATLAB code from which you generate code.

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Cell Array Limitations for Code Generation” on page 55-6.

Homogeneous vs. Heterogeneous Cell Arrays

A homogeneous cell array has these characteristics:

- The cell array is represented as an array in the generated code.
- All elements have the same properties. The type associated with the cell array specifies the properties of all elements rather than the properties of individual elements.
- The cell array can be variable-size.
- You can index into the cell array with an index whose value is determined at run time.

A heterogeneous cell array has these characteristics:

- The cell array is represented as a structure in the generated code. Each element is represented as a field of the structure.
- The elements can have different properties. The type associated with the cell array specifies the properties of each element individually.
- The cell array cannot be variable-size.
- You must index into the cell array with a constant index or with `for`-loops that have constant bounds.

The code generator uses heuristics to determine the classification of a cell array as homogeneous or heterogeneous. It considers the properties (class, size, complexity) of the elements and other factors, such as how you use the cell array in your program. Depending on how you use a cell array, the code generator can classify a cell array as homogeneous in one case and heterogeneous in another case. For example, consider the cell array `{1 [2 3]}`. The code generator can classify this cell array as a heterogeneous 1-by-2 cell array. The first element is double scalar. The second element is a 1-by-2 array of doubles. However, if you index into this cell array with an index whose value is determined at run time, the code generator classifies it as a homogeneous cell array. The elements are variable-size arrays of doubles with an upper bound of 2.

Controlling Whether a Cell Array Is Homogeneous or Heterogeneous

For cell arrays with certain characteristics, you cannot control the classification as homogeneous or heterogeneous:

- If the elements have different classes, the cell array must be heterogeneous.
- If the cell array is variable-size, it must be homogeneous.
- If you index into the cell array with an index whose value is determined at run time, the cell array must be homogeneous.

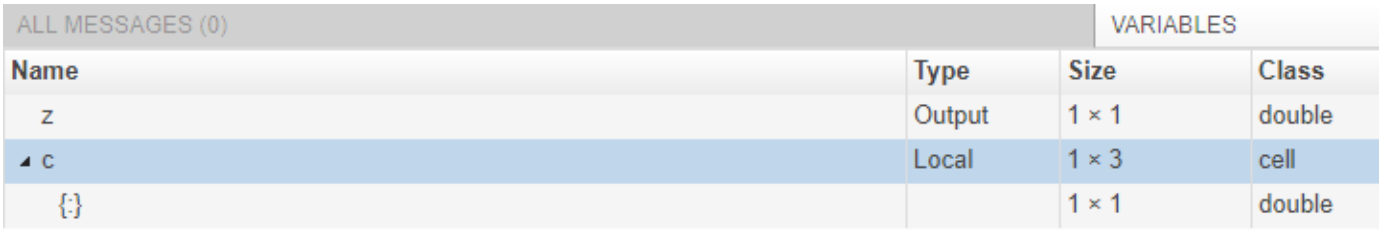
For other cell arrays, you can control the classification as homogeneous or heterogeneous.

- If the cell array elements have the same class, you can force a cell array to be homogeneous by using `coder.varsize`. See “Control Whether a Cell Array Is Variable-Size” on page 55-4.

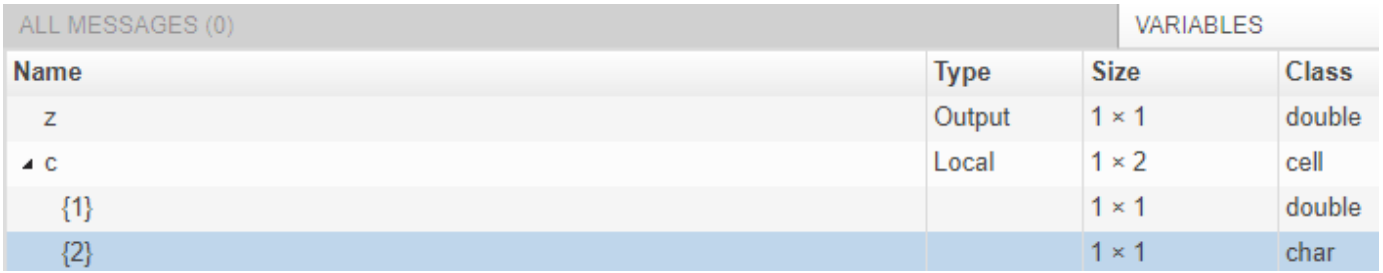
Cell Arrays in Reports

To see whether a cell array is homogeneous or heterogeneous, view the variable in the MATLAB Function report.

For a homogeneous cell array, the report has one entry that specifies the properties of all elements. The notation `{:}` indicates that all elements of the cell array have the same properties.



For a heterogeneous cell array, the report has an entry for each element. For example, for a heterogeneous cell array `c` with two elements, the entry for `c{1}` shows the properties for the first element. The entry for `c{2}` shows the properties for the second element.



See Also

`coder.cstructname` | `coder.varsize`

More About

- “Control Whether a Cell Array Is Variable-Size” on page 55-4
- “Cell Array Limitations for Code Generation” on page 55-6
- “MATLAB Function Reports” on page 44-41

Control Whether a Cell Array Is Variable-Size

The code generator classifies a variable-size cell array as homogeneous. The cell array elements must have the same class. In the generated code, the cell array is represented as an array.

To make a cell array variable-size:

- Create the cell array by using the `cell` function. For example:

```
function z = mycell(n, j)
%#codegen
assert (n < 100);
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

  For code generation, when you create a variable-size cell array by using `cell`, you must adhere to certain restrictions. See “Definition of Variable-Size Cell Array by Using cell” on page 55-7.

- Grow the cell array. For example:

```
function z = mycell(n)
%#codegen
c = {1 2 3};
if n > 3
    c = {1 2 3 4};
end
z = c{n};
end
```

- Force the cell array to be variable-size by using `coder.varsize`. Consider this code:

```
function y =  mycellfun()
%#codegen
c = {1 2 3};
coder.varsize('c', [1 10]);
y = c{1};
end
```

  Without `coder.varsize`, `c` is fixed-size with dimensions 1-by-3. With `coder.varsize`, `c` is variable-size with an upper bound of 10.

  Sometimes, using `coder.varsize` changes the classification of a cell array from heterogeneous to homogeneous. Consider this code:

```
function y =  mycell()
%#codegen
c = {1 [2 3]};
y = c{2};
end
```

  The code generator classifies `c` as heterogeneous because the elements have different sizes. `c` is fixed-size with dimensions 1-by-2. If you use `coder.varsize` with `c`, it becomes homogeneous. For example:

```
function y =  mycell()
%#codegen
c = {1 [2 3]};
coder.varsize('c', [1 10], [0 1]);
y = c{2};
end
```

`c` becomes a variable-size homogeneous cell array with dimensions 1-by-:10.

To force `c` to be homogeneous, but not variable-size, specify that none of the dimensions vary. For example:

```
function y =  mycell()
%#codegen
c = {1 [2 3]};
coder.varsize('c', [1 2], [0 0]);
y = c{2};
end
```

See Also

`coder.varsize`

More About

- “Code Generation for Cell Arrays” on page 55-2
- “Cell Array Limitations for Code Generation” on page 55-6
- “Code Generation for Variable-Size Arrays” on page 53-2

Cell Array Limitations for Code Generation

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to these restrictions:

Cell Array Element Assignment

You must assign a cell array element on all execution paths before you use it. For example:

```
function z = foo(n)
%#codegen
c = cell(1,3);
if n < 1
    c{2} = 1;

else
    c{2} = n;
end
z = c{2};
end
```

The code generator considers passing a cell array to a function or returning it from a function as a use of all elements of the cell array. Therefore, before you pass a cell array to a function or return it from a function, you must assign all of its elements. For example, the following code is not allowed because it does not assign a value to `c{2}` and `c` is a function output.

```
function c = foo()
%#codegen
c = cell(1,3);
c{1} = 1;
c{3} = 3;
end
```

The assignment of values to elements must be consistent on all execution paths. The following code is not allowed because `y{2}` is double on one execution path and char on the other execution path.

```
function y = foo(n)
y = cell(1,3)
if n > 1;
    y{1} = 1
    y{2} = 2;
    y{3} = 3;
else
    y{1} = 10;
```

```
    y{2} = 'a';
    y{3} = 30;
end
```

Variable-Size Cell Arrays

- `coder.varsize` is not supported for heterogeneous cell arrays.
- If you use the `cell` function to define a fixed-size cell array, you cannot use `coder.varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because `x = cell(1,3)` makes `x` a fixed-size,1-by-3 cell array.

  ```
  ...
  x = cell(1,3);
  coder.varsize('x',[1 5])
  ...
  ```

  You can use `coder.varsize` with a cell array that you define by using curly braces. For example:

  ```
  ...
  x = {1 2 3};
  coder.varsize('x',[1 5])
  ...
  ```
- To create a variable-size cell array by using the `cell` function, use this code pattern:

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  end
  ```

  See “Definition of Variable-Size Cell Array by Using cell” on page 55-7.

  To specify upper bounds for the cell array, use `coder.varsize`.

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  coder.varsize('x',[1,20]);
  end
  end
  ```

Definition of Variable-Size Cell Array by Using cell

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array. For example:

```
function z = mycell(n, j)
%#codegen
```

```
assert(n < 100);
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. If the code generator detects that some elements are not assigned, code generation fails with an error message. For example, modify the upper bound of the `for`-loop to `j`.

```
function z = mycell(n, j)
%#codegen
assert(n < 100);
x = cell(1,n);
for i = 1:j %<- Modified here
    x{i} = i;
end
z = x{j};
end
```

With this modification and with inputs `j` less than `n`, the function does not assign values to all of the cell array elements. Code generation produces the error:

```
Unable to determine that every element of 'x{:}' is assigned
before this line.
```

Sometimes, even though your code assigns all elements of the cell array, the code generator reports this message because the analysis does not detect that all elements are assigned. See “Unable to Determine That Every Element of Cell Array Is Assigned” on page 66-8.

To avoid this error, follow these guidelines:

- When you use `cell` to define a variable-size cell array, write code that follows this pattern:

  ```
  function z = mycell(n, j)
  %#codegen
  assert(n < 100);
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  z = x{j};
  end
  ```

  Here is the pattern for a multidimensional cell array:

  ```
  function z = mycell(m,n,p)
  %#codegen
  assert(m < 100);
  assert(n < 100);
  assert(p < 100);
  x = cell(m,n,p);
  for i = 1:m
      for j =1:n
          for k = 1:p
              x{i,j,k} = i+j+k;
  ```

```
        end
    end
end
z = x{m,n,p};
end
```

- Increment or decrement the loop counter by 1.
- Define the cell array within one loop or one set of nested loops. For example, this code is not allowed:

```
function z = mycell(n, j)
assert(n < 50);
assert(j < 50);
x = cell(1,n);
for i = 1:5
    x{i} = 5;
end
for i = 6:n
    x{i} = 5;
end
z = x{j};
end
```

- Use the same variables for the cell dimensions and loop initial and end values. For example, code generation fails for the following code because the cell creation uses `n` and the loop end value uses `m`:

```
function z = mycell(n, j)
assert(n < 50);
assert(j < 50);
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
z = x{j};
end
```

  Rewrite the code to use `n` for the cell creation and the loop end value:

```
function z = mycell(n, j)
assert(n < 50);
assert(j < 50);
x = cell(1,n);
for i = 1:n
    x{i} = 2;
end
z = x{j};
end
```

- Create the cell array with this pattern:

```
x = cell(1,n)
```

  Do not assign the cell array to a field of a structure or a property of an object. For example, this code is not allowed:

```
myobj.prop = cell(1,n)
for i = 1:n
```

```
...
end
```

Do not use the `cell` function inside the cell array constructor `{}`. For example, this code is not allowed:

```
x = {cell(1,n)};
```

- The cell array creation and the loop that assigns values to the cell array elements must be together in a unique execution path. For example, the following code is not allowed.

```
function z = mycell(n)
assert(n < 100);
if n > 3
    c = cell(1,n);
else
    c = cell(n,1);
end
for i = 1:n
    c{i} = i;
end
z = c{n};
end
```

To fix this code, move the assignment loop inside the code block that creates the cell array.

```
function z = cellerr(n)
assert(n < 100);
if n > 3
    c = cell( 1,n);
    for i = 1:n
        c{i} = i;
    end
else
    c = cell(n,1);
    for i = 1:n
        c{i} = i;
    end
end
z = c{n};
end
```

Cell Array Indexing

- You cannot index cell arrays by using smooth parentheses`()`. Consider indexing cell arrays by using curly braces`{}` to access the contents of the cell.
- You must index into heterogeneous cell arrays by using constant indices or by using `for`-loops with constant bounds.

  For example, the following code is not allowed.

```
x = {1, 'mytext'};
disp(x{randi});
```

  You can index into a heterogeneous cell array in a `for`-loop with constant bounds because the code generator unrolls the loop. Unrolling creates a separate copy of the loop body for each loop iteration, which makes the index in each loop iteration constant. However, if the `for`-loop has a

large body or it has many iterations, the unrolling can increase compile time and generate inefficient code.

If `A` and `B` are constant, the following code shows indexing into a heterogeneous cell array in a `for`-loop with constant bounds.

```
x = {1, 'mytext'};
for i = A:B
     disp(x{i});
end
```

Growing a Cell Array by Using {end + 1}

To grow a cell array `X`, you can use `X{end + 1}`. For example:

```
...
X = {1 2};
X{end + 1} = 'a';
...
```

When you use `{end + 1}` to grow a cell array, follow these restrictions:

- In a MATLAB Function block, do not use `{end + 1}` in a `for`-loop.
- Use only `{end + 1}`. Do not use `{end + 2}`, `{end + 3}`, and so on.
- Use `{end + 1}` with vectors only. For example, the following code is not allowed because `X` is a matrix, not a vector:

  ```
  ...
  X = {1 2; 3 4};
  X{end + 1} = 5;

  ...
  ```

- Use `{end + 1}` only with a variable. In the following code, `{end + 1}` does not cause `{1 2 3}` to grow. In this case, the code generator treats `{end + 1}` as an out-of-bounds index into `X{2}`.

  ```
  ...
  X = {'a' { 1 2 3 }};
  X{2}{end + 1} = 4;
  ...
  ```

- When `{end + 1}` grows a cell array in a loop, the cell array must be variable-size. Therefore, the cell array must be homogeneous on page 55-2.

  This code is allowed because `X` is homogeneous.

  ```
  ...
  X = {1  2};
  for i=1:n
      X{end + 1} = 3;
  end
  ...
  ```

  This code is not allowed because `X` is heterogeneous.

  ```
  ...
  X = {1 'a' 2 'b'};
  ```

```
for i=1:n
    X{end + 1} = 3;
end
...
```

Cell Array Contents

Cell arrays cannot contain `mxarrays`. In a cell array, you cannot store a value that an extrinsic function returns.

Passing Cell Arrays to External C/C++ Functions

You cannot pass a cell array to `coder.ceval`. If a variable is an input argument to `coder.ceval`, define the variable as an array or structure instead of as a cell array.

Use in MATLAB Function Block

You cannot use cell arrays for Simulink signals, parameters, or data store memory.

See Also

More About

- “Code Generation for Cell Arrays” on page 55-2
- “Differences Between Generated Code and MATLAB Code” on page 48-6

Code Generation for Categorical Arrays

Code Generation for Categorical Arrays

In this section...

Categorical arrays store data with values from a finite set of discrete categories. You can specify an order for the categories, but it is not required. A categorical array provides efficient storage and manipulation of nonnumeric data, while also maintaining meaningful names for the values.

When you use categorical arrays with code generation, adhere to these restrictions:

Define Categorical Arrays for Code Generation

For code generation, use the `categorical` function to create categorical arrays. For example, suppose the input argument to your MATLAB function is a numeric array of arbitrary size whose elements have values of either `1`, `2`, or `3`. You can convert these values to the categories `small`, `medium`, and `large` and turn the input array into a categorical array, as shown in this code.

```
function c = foo(x) %#codegen
    c = categorical(x,1:3,{'small','medium','large'});
end
```

Allowed Operations on Categorical Arrays

For code generation, you are restricted to the operations on categorical arrays listed in this table.

Operation	Example	Notes

assignment operator: `=`	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`c(1) = 'large';`
	
	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`c(1) = 'large';`
		Code generation does not support using the assignment operator `=` to:
		• Delete an element.
		• Expand the size of a categorical array.
		• Add a new category, even when the array is not protected.

relational operators: `< > <= >= == ~=`	`c = categorical(1:3,'Ordinal',true);`
	`tf = c(1) < c(2);`
	
	`c = categorical(1:3,'Ordinal',true);`
	`tf = c(1) < c(2);`
		Code generation supports all relational operators.

cast to numeric type	`c = categorical(1:3);`
	`double(c(1));`
	
	`c = categorical(1:3);`
	`double(c(1));`
		Code generation supports casting categorical arrays to arrays of double- or single-precision floating-point numbers, or to integers.

Operation	Example	Notes

conversion to text	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`c1 = cellstr(c(1)); % One element`
	`c2 = cellstr(c);  % Entire array`
		Code generation does not support using the `char` or `string` functions to convert categorical values to text.
		
		To convert one or more elements of a categorical array to text, use the `cellstr` function.

indexing operation	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`idx = [1 2];`
	`c(idx);`
	`idx = logical([1 1 0]);`
	`c(idx);`
		Code generation supports indexing by position, linear indexing, and logical indexing.

concatenation	`c1 = categorical(1:3,1:3,{'small','medium','large'});`
	`c2 = categorical(4:6,[2 1 4],{'medium','small','extra-large'});`
	`c = [c1 c2];`
		Code generation supports concatenation of categorical arrays along any dimension.

MATLAB Toolbox Functions That Support Categorical Arrays

For code generation, you can use categorical arrays with these MATLAB toolbox functions:

- `addcats`
- `cat`
- `categorical`
- `categories`
- `cellstr`
- `countcats`
- `ctranspose`
- `double`
- `eq`
- `ge`
- `gt`
- `histcounts`
- `horzcat`
- `int8`
- `int16`
- `int32`

- int64
- intersect
- iscategory
- iscolumn
- isempty
- isequal
- isequaln
- ismatrix
- ismember
- isordinal
- isprotected
- isrow
- isscalar
- isundefined
- isvector
- le
- length
- lt
- max
- mergecats
- min
- ndims
- ne
- numel
- permute
- removecats
- renamecats
- reordercats
- reshape
- setcats
- setdiff
- setxor
- single
- size
- transpose
- uint8
- uint16
- uint32
- uint64

- `union`
- `unique`
- `vertcat`

See Also

More About

- “Categorical Array Limitations for Code Generation” on page 56-7

Define Categorical Array Inputs

Define Categorical Array Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Categorical Array Input” on page 56-6
- “Provide a Categorical Array Type” on page 56-6
- “Provide a Constant Categorical Array Input” on page 56-6

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Categorical Array Input

Provide a Categorical Array Type

To provide a type for a categorical array to :

1. Define a categorical array. For example:

   ```
   C = categorical({'r','g','b'});
   ```

2. Create a type from `C`.

   ```
   t = coder.typeof(C);
   ```

Provide a Constant Categorical Array Input

To specify that a categorical array input is constant, use `coder.Constant` with the `-args` option:

See Also

`categorical` | `coder.Constant` | `coder.typeof`

More About

- “Code Generation for Categorical Arrays” on page 56-2
- “Categorical Array Limitations for Code Generation” on page 56-7

Categorical Array Limitations for Code Generation

When you create categorical arrays in MATLAB code that you intend for code generation, you must specify the categories and elements of each categorical array by using the `categorical` function. See “Categorical Arrays”.

For categorical arrays, code generation does not support the following inputs and operations:

- Arrays of MATLAB objects.
- Sparse matrices.
- Duplicate category names when you specify them using the `categoryNames` input argument of the `categorical` function.
- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

  ```
  function c = foo() %#codegen
      c = categorical(1:3,1:3,{'small','medium','large'});
      c(4) = 'medium';
  end
  ```
- Adding a category. For example, specifying a new category by using the = operator produces an error, even when the categorical array is unprotected.

  ```
  function c = foo() %#codegen
      c = categorical(1:3,1:3,{'small','medium','large'});
      c(1) = 'extra-large';
  end
  ```
- Deleting an element. For example, assigning an empty array to an element produces an error.

  ```
  function c = foo() %#codegen
      c = categorical(1:3,1:3,{'small','medium','large'});
      c(1) = [];
  end
  ```
- Converting categorical values to text by using the `char` or `string` functions. To convert elements of a categorical array to text, use the `cellstr` function.

Limitations that apply to classes also apply to categorical arrays. For more information, see “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`categorical` | `cellstr`

More About

- “Code Generation for Categorical Arrays” on page 56-2

Code Generation for Datetime Arrays

Code Generation for Datetime Arrays

In this section...

The values in a `datetime` array represent points in time using the proleptic ISO calendar.

When you use `datetime` arrays with code generation, adhere to these restrictions.

Define Datetime Arrays for Code Generation

For code generation, use the `datetime` function to create `datetime` arrays. For example, suppose the input arguments to your MATLAB function are numeric arrays whose values indicate the year, month, day, hour, minute, and second components for a point in time. You can create a `datetime` array from these input arrays.

```
function d = foo(y,mo,d,h,mi,s) %#codegen
    d = datetime(y,mo,d,h,mi,s);
end
```

Allowed Operations on Datetime Arrays

For code generation, you are restricted to the operations on `datetime` arrays listed in this table.

Operation	Example	Notes

Assignment operator: `=`	`d = datetime(2019,1:12,1,12,0,0);`
	`d(1) = datetime(2019,1,31);`
	
	`d = datetime(2019,1:12,1,12,0,0);`
	`d(1) = datetime(2019,1,31);`
		Code generation does not support using the assignment operator `=` to:
		• Delete an element.
		• Expand the size of a `datetime` array.

Relational operators: `< > <= >= == ~=`	`d = datetime(2019,1:12,1,12,0,0);`
	`tf = d(1) < d(2);`
	
	`d = datetime(2019,1:12,1,12,0,0);`
	`tf = d(1) < d(2);`
		Code generation supports relational operators.

Indexing operation	`d = datetime(2019,1:12,1,12,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
	
	`d = datetime(2019,1:12,1,12,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
		Code generation supports indexing by position, linear indexing, and logical indexing.

Operation	Example	Notes

Concatenation	`d1 = datetime(2019,1:6,1,12,0,0);`
	`d2 = datetime(2019,7:12,1,12,0,0);`
	`d = [d1 d2];`
	
	`d1 = datetime(2019,1:6,1,12,0,0);`
	`d2 = datetime(2019,7:12,1,12,0,0);`
	`d = [d1 d2];`
		Code generation supports concatenation of `datetime` arrays.

MATLAB Toolbox Functions That Support Datetime Arrays

For code generation, you can use `datetime` arrays with these MATLAB toolbox functions:

- `cat`
- `colon`
- `ctranspose`
- `datetime`
- `datevec`
- `diff`
- `eq`
- `ge`
- `gt`
- `hms`
- `horzcat`
- `hour`
- `interp1`
- `intersect`
- `iscolumn`
- `isempty`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismatrix`
- `ismember`
- `isnat`
- `isreal`
- `isrow`
- `isscalar`
- `issorted`
- `issortedrows`
- `isvector`
- `le`

- `length`
- `linspace`
- `lt`
- `max`
- `mean`
- `min`
- `minus`
- `minute`
- `NaT`
- `ndims`
- `ne`
- `numel`
- `permute`
- `plus`
- `posixtime`
- `repmat`
- `reshape`
- `setdiff`
- `setxor`
- `size`
- `sort`
- `sortrows`
- `topkrows`
- `transpose`
- `union`
- `unique`
- `vertcat`
- `ymd`

See Also

More About

- “Datetime Array Limitations for Code Generation” on page 57-6

Define Datetime Array Inputs

Define Datetime Array Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Datetime Array Input” on page 57-5
- “Provide a Datetime Array Type” on page 57-5
- “Provide a Constant Datetime Array Input” on page 57-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Datetime Array Input

Provide a Datetime Array Type

To provide a type for a `datetime` array to :

1. Define a `datetime` array. For example:

   ```
   D = datetime(2019,1:12,1,12,0,0);
   ```

2. Create a type from `D`.

   ```
   t = coder.typeof(D);
   ```

Provide a Constant Datetime Array Input

To specify that a `datetime` array input is constant, use `coder.Constant` with the `-args` option:

See Also

`NaT` | `coder.Constant` | `coder.typeof` | `datetime`

More About

- “Code Generation for Datetime Arrays” on page 57-2
- “Datetime Array Limitations for Code Generation” on page 57-6

Datetime Array Limitations for Code Generation

When you create `datetime` arrays in MATLAB code that you intend for code generation, you must specify the values by using the `datetime` function. See “Dates and Time”.

For `datetime` arrays, code generation does not support the following inputs and operations:

- Text inputs. For example, specifying a character vector as the input argument produces an error.

  ```
  function d = foo() %#codegen
      d = datetime('2019-12-01');
  end
  ```

- The `'Format'` name-value pair argument. You cannot specify the display format by using the `datetime` function, or by setting the `Format` property of a `datetime` array. To use a specific display format, create a `datetime` array in MATLAB, then pass it as an input argument to a function that is intended for code generation.
- The `'TimeZone'` name-value pair argument and the `TimeZone` property. When you use `datetime` arrays in code that is intended for code generation, they must be unzoned.
- Setting time component properties. For example, setting the `Hour` property in the following code produces an error:

  ```
  d = datetime;
  d.Hour = 2;
  ```

- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

  ```
  function d = foo() %#codegen
      d = datetime(2019,1:12,1,12,0,0);
      d(13) = datetime(2020,1,1,12,0,0);
  end
  ```

- Deleting an element. For example, assigning an empty array to an element produces an error.

  ```
  function d = foo() %#codegen
      d = datetime(2019,1:12,1,12,0,0);
      d(1) = [];
  end
  ```

- Converting `datetime` values to text by using the `char`, `cellstr`, or `string` functions.

Limitations that apply to classes also apply to `datetime` arrays. For more information, see “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`NaT` | `datetime`

More About

- “Code Generation for Datetime Arrays” on page 57-2

Code Generation for Duration Arrays

Code Generation for Duration Arrays

In this section...

The values in a duration array represent elapsed times in units of fixed length, such as hours, minutes, and seconds. You can create elapsed times in terms of fixed-length (24-hour) days and fixed-length (365.2425-day) years.

You can add, subtract, sort, compare, concatenate, and plot duration arrays.

When you use duration arrays with code generation, adhere to these restrictions.

Define Duration Arrays for Code Generation

For code generation, use the `duration` function to create duration arrays. For example, suppose the input arguments to your MATLAB function are three numeric arrays of arbitrary size whose elements specify lengths of time as hours, minutes, and seconds. You can create a duration array from these three input arrays.

```
function d = foo(h,m,s) %#codegen
    d = duration(h,m,s);
end
```

You can use the `years`, `days`, `hours`, `minutes`, `seconds`, and `milliseconds` functions to create duration arrays in units of years, days, hours, minutes, or seconds. For example, you can create an array of hours from an input numeric array.

```
function d = foo(h) %#codegen
    d = hours(h);
end
```

Allowed Operations on Duration Arrays

For code generation, you are restricted to the operations on duration arrays listed in this table.

Operation	Example	Notes

assignment operator: `=`	`d = duration(1:3,0,0);`
	`d(1) = hours(5);`
	
	`d = duration(1:3,0,0);`
	`d(1) = hours(5);`
		Code generation does not support using the assignment operator `=` to:
		• Delete an element.
		• Expand the size of a duration array.

relational operators: `< > <= >= == ~=`	`d = duration(1:3,0,0);`
	`tf = d(1) < d(2);`
	
	`d = duration(1:3,0,0);`
	`tf = d(1) < d(2);`
		Code generation supports relational operators.

Operation	Example	Notes

indexing operation	`d = duration(1:3,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
	
	`d = duration(1:3,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
		Code generation supports indexing by position, linear indexing, and logical indexing.

concatenation	`d1 = duration(1:3,0,0);`
	`d2 = duration(4,30,0);`
	`d = [d1 d2];`
	
	`d1 = duration(1:3,0,0);`
	`d2 = duration(4,30,0);`
	`d = [d1 d2];`
		Code generation supports concatenation of duration arrays.

MATLAB Toolbox Functions That Support Duration Arrays

For code generation, you can use duration arrays with these MATLAB toolbox functions:

- `abs`
- `cat`
- `ceil`
- `colon`
- `cummax`
- `cummin`
- `cumsum`
- `ctranspose`
- `datevec`
- `days`
- `diff`
- `duration`
- `eps`
- `eq`
- `floor`
- `ge`
- `gt`
- `hms`
- `horzcat`
- `hours`
- `interp1`
- `intersect`
- `iscolumn`

- isempty
- isequal
- isequaln
- isfinite
- isinf
- ismatrix
- ismember
- isnan
- isreal
- isrow
- isscalar
- issorted
- issortedrows
- isvector
- ldivide
- le
- length
- linspace
- lt
- max
- mean
- median
- milliseconds
- min
- minus
- minutes
- mldivide
- mode
- mrdivide
- mod
- mtimes
- ndims
- ne
- nnz
- numel
- permute
- plus
- repmat
- rdivide

- `rem`
- `reshape`
- `seconds`
- `setdiff`
- `setxor`
- `sign`
- `size`
- `sort`
- `sortrows`
- `std`
- `sum`
- `times`
- `transpose`
- `uminus`
- `union`
- `unique`
- `uplus`
- `vertcat`
- `years`

See Also

More About

- “Duration Array Limitations for Code Generation” on page 58-7

Define Duration Array Inputs

Define Duration Array Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Duration Array Input” on page 58-6
- “Provide a Duration Array Type” on page 58-6
- “Provide a Constant Duration Array Input” on page 58-6

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Duration Array Input

Provide a Duration Array Type

To provide a type for a duration array to :

1. Define a duration array. For example:

   ```
   D = duration(1:3,0,0);
   ```

2. Create a type from `D`.

   ```
   t = coder.typeof(D);
   ```

Provide a Constant Duration Array Input

To specify that a duration array input is constant, use `coder.Constant` with the `-args` option:

See Also

`coder.Constant` | `coder.typeof` | `duration`

More About

- “Code Generation for Duration Arrays” on page 58-2
- “Duration Array Limitations for Code Generation” on page 58-7

Duration Array Limitations for Code Generation

When you create duration arrays in MATLAB code that you intend for code generation, you must specify the durations by using the `duration`, `years`, `days`, `hours`, `minutes`, `seconds`, or `milliseconds` functions. See “Dates and Time”.

For duration arrays, code generation does not support the following inputs and operations:

- Text inputs. For example, specifying a character vector as the input argument produces an error.

```
function d = foo() %#codegen
    d = duration('01:30:00');
end
```

- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

```
function d = foo() %#codegen
    d = duration(1:3,0,0);
    d(4) = hours(4);
end
```

- Deleting an element. For example, assigning an empty array to an element produces an error.

```
function d = foo() %#codegen
    d = duration(1:3,0,0);
    d(1) = [];
end
```

- Converting duration values to text by using the `char`, `cellstr`, or `string` functions.

Limitations that apply to classes also apply to duration arrays. For more information, see “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`days` | `duration` | `hours` | `milliseconds` | `minutes` | `seconds` | `years`

More About

- “Code Generation for Duration Arrays” on page 58-2

Code Generation for Tables

Code Generation for Tables

In this section...

The `table` data type is a data type suitable for column-oriented or tabular data that is often stored as columns in a text file or in a spreadsheet. Tables consist of rows and column-oriented variables. Each variable in a table can have a different data type and a different size with one restriction: each variable must have the same number of rows. For more information, see “Tables”.

When you use tables with code generation, adhere to these restrictions.

Define Tables for Code Generation

For code generation, use the `table` function. For example, suppose the input arguments to your MATLAB function are three arrays that have the same number of rows and a cell array that has variable names. You can create a table that contains these arrays as table variables.

```
function T = foo(A,B,C,vnames) %#codegen
    T = table(A,B,C,'VariableNames',vnames);
end
```

You can use the `array2table`, `cell2table`, and `struct2table` functions to convert arrays, cell arrays, and structures to tables. For example, you can convert an input cell array to a table.

```
function T = foo(C,vnames) %#codegen
    T = cell2table(C,'VariableNames',vnames);
end
```

For code generation, you must supply table variable names when you create a table. Table variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).

Allowed Operations on Tables

For code generation, you are restricted to the operations on tables listed below.

Operation	Example	Notes

assignment operator: =	`T = table(A,B,C,'VariableNames',vnames);`
	`T{:,1} = D;`
	
	`T = table(A,B,C,'VariableNames',vnames);`
	`T{:,1} = D;`
		Code generation does not support using the assignment operator = to:
		• Delete a variable or a row.
		• Add a variable or a row.

Operation	Example	Notes

indexing operation	`T = table(A,B,C,'VariableNames',vnames);`
	`T(1:5,1:3);`
	
	`T = table(A,B,C,'VariableNames',vnames);`
	`T(1:5,1:3);`
		Code generation supports indexing by position, variable or row name, and logical indexing.
		
		Code generation supports:
		• Table indexing with smooth parentheses, ().
		• Content indexing with curly braces, {}.
		• Dot notation to access a table variable.

concatenation	`T1 = table(A,B,C,'VariableNames',vnames);`
	`T2 = table(D,E,F,'VariableNames',vnames);`
	`T = [T1 ; T2];`
	
	`T1 = table(A,B,C,'VariableNames',vnames);`
	`T2 = table(D,E,F,'VariableNames',vnames);`
	`T = [T1 ; T2];`
		Code generation supports table concatenation.
		
		• For vertical concatenation, tables must have variables that have the same names in the same order.
		• For horizontal concatenation, tables must have the same number of rows. If the tables have row names, then they must have the same row names in the same order.

MATLAB Toolbox Functions That Support Tables

For code generation, you can use tables with these MATLAB toolbox functions:

- `addvars`
- `array2table`
- `cat`
- `cell2table`
- `convertvars`
- `height`
- `horzcat`
- `intersect`
- `isempty`
- `ismember`
- `movevars`
- `ndims`
- `numel`
- `removevars`

- `setdiff`
- `setxor`
- `size`
- `struct2table`
- `table`
- `table2array`
- `table2cell`
- `table2struct`
- `union`
- `unique`
- `vertcat`
- `width`

See Also

More About

- “Table Limitations for Code Generation” on page 59-6

Define Table Inputs

Define Table Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Table Input” on page 59-5
- “Provide a Table Type” on page 59-5
- “Provide a Constant Table Input” on page 59-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Table Input

Provide a Table Type

To provide a type for a table to :

1. Define a table. For example:

   ```
   T = table(A,B,C,'VariableNames',vnames);
   ```

2. Create a type from `T`.

   ```
   t = coder.typeof(T);
   ```

Provide a Constant Table Input

To specify that a table input is constant, use `coder.Constant` with the `-args` option:

See Also

`coder.Constant` | `coder.typeof` | `table`

More About

- “Code Generation for Tables” on page 59-2
- “Table Limitations for Code Generation” on page 59-6

Table Limitations for Code Generation

When you create tables in MATLAB code that you intend for code generation, you must create them by using the `array2table`, `cell2table`, `struct2table`, or `table` functions. For more information, see “Tables”.

For tables, code generation has these limitations:

- You must specify variables names using the `'VariableNames'` name-value pair argument when creating tables from input arrays by using the `table`, `array2table`, or `cell2table` functions.

  You do not have to specify the `'VariableNames'` argument when you preallocate a table by using the `table` function and the `'Size'` name-value pair argument.
- Table variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).
- You cannot change the `VariableNames`, `RowNames`, `DimensionNames`, or `UserData` properties of a table after you create it.

  You can specify the `'VariableNames'` and `'RowNames'` input arguments when you create a table. These input arguments specify the properties.
- To pass table indices into generated code as input arguments, first make the indices constant by using the `coder.Constant` function. If table indices are not constant, then indexing into variables produces an error.
- You cannot add custom metadata to a table. The `addprop` and `rmprop` functions are not supported.
- You cannot change the size of a table by assignments. For example, adding a new row produces an error.

  ```
  function T = foo() %#codegen
      T = table((1:3)',(1:3)','VariableNames',{'Var1','Var2'});
      T(4,2) = 5;
  end
  ```

  Deleting a row or a variable also produces an error.
- When you preallocate a table, you can specify only the following data types by using the `'VariableTypes'` name-value pair argument.

Data Type Name	Initial Value in Each Element
`'double'`, `'single'`	Double- or single-precision `0`
`'doublenan'`, `'doubleNaN'`, `'singlenan'`, `'singleNaN'`	Double- or single-precision `NaN`
`'int8'`, `'int16'`, `'int32'`, `'int64'`	Signed 8-, 16-, 32-, or 64-bit integer `0`
`'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`	Unsigned 8-, 16-, 32-, or 64-bit integer `0`
`'logical'`	`0` (`false`)
`'duration'`	`0` seconds, as a duration value
`'cellstr'`	`{''}` (cell with 0-by-0 character array)

If you specify `'char'` as a data type, then `table` preallocates the corresponding variable as a cell array of character vectors, not as a character array. Best practice is to avoid creating table variables that are character arrays.

- When you vertically concatenate tables, they must have the same variable names in the same order. In MATLAB, the variable names must be the same but can be in different orders.
- When you horizontally concatenate tables, and the tables have row names, they must have the same row names in the same order. In MATLAB, the row names must be the same but can be in different orders.
- If two tables have variables that are N-D cell arrays, then the tables cannot be vertically concatenated.
- You cannot use curly braces to extract data from multiple table variables that are N-D cell arrays, since this operation is horizontal concatenation.
- The set membership functions `intersect`, `setdiff`, `setxor`, and `union` support unsorted tables in all cases. You do not have to specify the `'stable'` option.
- When using the `movevars` function, the input argument `vars` cannot contain duplicate variable names.
- When using the `convertvars` function:
  - Function handles are not supported.
  - The second and third input arguments (`vars` and `dataType`) must be constant.
  - You cannot specify `dataType` as `'char'`.

Limitations that apply to classes also apply to tables. For more information, see “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`array2table` | `cell2table` | `struct2table` | `table`

More About

- “Code Generation for Tables” on page 59-2

Code Generation for Timetables

Code Generation for Timetables

In this section...

The `timetable` data type is a data type suitable for tabular data with time-stamped rows. Like tables, timetables consist of rows and column-oriented variables. Each variable in a timetable can have a different data type and a different size with one restriction: each variable must have the same number of rows.

The *row times* of a timetable are time values that label the rows. You can index into a timetable by row time and variable. To index into a timetable, use smooth parentheses `()` to return a subtable or curly braces `{}` to extract the contents. You can refer to variables and to the vector of row times by their names. For more information, see “Timetables”.

When you use timetables with code generation, adhere to these restrictions.

Define Timetables for Code Generation

For code generation, use the `timetable` function. For example, suppose the input arguments to your MATLAB function are three arrays that have the same number of rows (`A`, `B`, and `C`), a `datetime` or `duration` vector containing row times (`D`), and a cell array that has variable names (`vnames`). You can create a timetable that contains these arrays as timetable variables.

```
function TT = foo(A,B,C,D,vnames) %#codegen
    TT = table(A,B,C,'RowTimes',D,'VariableNames',vnames);
end
```

To convert arrays and tables to timetables, use the `array2timetable` and `table2timetable` functions. For example, you can convert an input `M`-by-`N` matrix to a timetable, where each column of the matrix becomes a variable in the timetable. Assign row times by using a `duration` vector.

```
function TT = foo(A,D,vnames) %#codegen
    TT = array2timetable(A,'RowTimes',D,'VariableNames',vnames);
end
```

For code generation, you must supply timetable variable names when you create a timetable. Timetable variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).

The row times can have either the `datetime` or `duration` data type.

Allowed Operations on Timetables

For code generation, you are restricted to the operations on timetables listed in this table.

Operation	Example	Notes

Assignment operator: =	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);`
	`TT{:,1} = X;`
	
	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);`
	`TT{:,1} = X;`
		Code generation does not support using the assignment operator = to:
		• Delete a variable or a row.
		• Add a variable or a row.

Indexing operation	`D = seconds(1:10);`
	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);`
	`TT(seconds(3:7),1:3);`
	
	`D = seconds(1:10);`
	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);`
	`TT(seconds(3:7),1:3);`
		Code generation supports indexing by position, variable or row time, and logical indexing. Also, you can index using objects created by using the `timerange` or `withtol` functions.
		
		Code generation supports:
		• Timetable indexing with smooth parentheses, ().
		• Content indexing with curly braces, {}.
		• Dot notation to access a timetable variable.

Concatenation	`TT1 = timetable(A,B,C,'RowTimes',D1,'VariableNames',vnames);`
	`TT2 = timetable(D,E,F,'RowTimes',D2,'VariableNames',vnames);`
	`TT = [TT1 ; TT2];`
	
	`TT1 = timetable(A,B,C,'RowTimes',D1,'VariableNames',vnames);`
	`TT2 = timetable(D,E,F,'RowTimes',D2,'VariableNames',vnames);`
	`TT = [TT1 ; TT2];`
		Code generation supports timetable concatenation.
		• For vertical concatenation, timetables must have variables that have the same names in the same order.
		• For horizontal concatenation, timetables must have the same number of rows. They also must have the same row times in the same order.

MATLAB Toolbox Functions That Support Timetables

For code generation, you can use timetables with these MATLAB toolbox functions:

- `addvars`
- `array2timetable`
- `cat`
- `convertvars`
- `height`
- `horzcat`
- `intersect`

- `isempty`
- `ismember`
- `isregular`
- `movevars`
- `ndims`
- `numel`
- `removevars`
- `retime`
- `setdiff`
- `setxor`
- `size`
- `synchronize`
- `table2timetable`
- `timerange`
- `timetable`
- `timetable2table`
- `union`
- `unique`
- `vertcat`
- `width`
- `withtol`

See Also

More About

- “Timetable Limitations for Code Generation” on page 60-6

Define Timetable Inputs

Define Timetable Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Timetable Input” on page 60-5
- “Provide a Timetable Type” on page 60-5
- “Provide a Constant Timetable Input” on page 60-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Timetable Input

Provide a Timetable Type

To provide a type for a timetable to :

1. Define a timetable. For example:

   ```
   TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);
   ```

2. Create a type from `T`.

   ```
   t = coder.typeof(TT);
   ```

Provide a Constant Timetable Input

To specify that a timetable input is constant, use `coder.Constant` with the `-args` option:

See Also

`coder.Constant` | `coder.typeof` | `timetable`

More About

- “Code Generation for Timetables” on page 60-2
- “Timetable Limitations for Code Generation” on page 60-6

Timetable Limitations for Code Generation

When you create timetables in MATLAB code that you intend for code generation, you must create them by using the `array2timetable`, `table2timetable`, or `timetable` functions. For more information, see “Timetables”.

For timetables, code generation has these limitations:

- The name of the first dimension of a timetable is `'Time'` and cannot be changed. The name of the first dimension is also the name of the vector of row times, which you can refer to using dot notation.
- You must specify variables names by using the `'VariableNames'` name-value pair argument when creating timetables from input arrays by using the `timetable` or `array2timetable` functions.

  You do not have to specify the `'VariableNames'` argument when you preallocate a timetable by using the `timetable` function and the `'Size'` name-value pair argument.
- Timetable variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).
- After you create a timetable, you cannot change the `VariableNames`, `DimensionNames`, or `UserData` properties.

  When you create a timetable, you can specify the `'VariableNames'` and `'RowTimes'` input arguments to set the properties having those names.
- To create a regular timetable when specifying the `'SampleRate'`, `'StartTime'`, or `'TimeStep'` name-value pair arguments, first use the `coder.Constant` function to make the values constant. If you do not make them constant, then the row times are considered to be irregular.

  Also, if you create an irregular timetable, then it remains irregular even if you set its sample rate or time step.
- If you create a regular timetable, and you attempt to set irregular row times, then an error is produced.
- To pass timetable indices into generated code as input arguments, first use the `coder.Constant` function to make the indices into the second dimension of the timetable constant. If indices into the second dimension are not constant, then indexing into variables produces an error.
- If you index into a timetable by using `duration` values, or an object produced by the `timerange` or `withtol` functions, then the output is nonconstant with a variable number of rows.
- If you index into a regular timetable by using `duration` values, or an object produced by the `timerange` or `withtol` functions, then the output is considered to be irregular.
- You cannot add custom metadata to a timetable. The `addprop` and `rmprop` functions are not supported.
- You cannot change the size of a timetable by assignments. For example, this call to add a new row produces an error.

```
function TT = foo() %#codegen
    TT = timetable((1:3)',(1:3)','RowTimes',seconds([0,5,10]),...
                   'VariableNames',{'Var1','Var2'});
    TT{4,:} = [5,5];
end
```

Deleting a row or a variable by assignment also produces an error.

- You cannot add a new row by using a new row time in an assignment. For example, this call to add a new row by using a new row time instead of a numeric index does not produce an error, but also does not add the new row.

```
function TT = foo() %#codegen
    TT = timetable((1:3)',(1:3)','RowTimes',seconds([0,5,10]),...
                   'VariableNames',{'Var1','Var2'});
    TT{seconds(15),:} = [5,5];
end
```

- When you preallocate a timetable, you can specify only the following data types by using the `'VariableTypes'` name-value pair argument.

Data Type Name	Initial Value in Each Element
`'double'`, `'single'`	Double- or single-precision `0`
`'doublenan'`, `'doubleNaN'`, `'singlenan'`, `'singleNaN'`	Double- or single-precision `NaN`
`'int8'`, `'int16'`, `'int32'`, `'int64'`	Signed 8-, 16-, 32-, or 64-bit integer `0`
`'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`	Unsigned 8-, 16-, 32-, or 64-bit integer `0`
`'logical'`	`0` (`false`)
`'datetime'`	`NaT` datetime value
`'duration'`	`0` seconds, as a duration value
`'cellstr'`	`{''}` (cell with 0-by-0 character array)

  If you specify `'char'` as a data type, then `timetable` preallocates the corresponding variable as a cell array of character vectors, not as a character array. The best practice is to avoid creating timetable variables that are character arrays.
- When you vertically concatenate timetables, they must have the same variable names in the same order. In MATLAB, the variable names must be the same but can be in different orders in the timetables.
- When you horizontally concatenate timetables, they must have the same row times in the same order. In MATLAB, the row times must be the same but can be in different orders in the timetables.
- If two timetables have variables that are N-D cell arrays, then you cannot vertically concatenate the timetables.
- You cannot use curly braces to extract data from multiple timetable variables that are N-D cell arrays because this operation is horizontal concatenation.
- The set membership functions `intersect`, `setdiff`, `setxor`, and `union` support unsorted timetables in all cases. You do not have to specify the `'stable'` option.
- When using the `convertvars` function:
  - Function handles are not supported.
  - The second and third input arguments (`vars` and `dataType`) must be constant.
  - You cannot specify `dataType` as `'char'`.
- When using the `movevars` function, the input argument `vars` cannot contain duplicate variable names.

- When using the `isregular` function:
  - Use `coder.Constant` to make the input argument `timeComponent` constant.
  - The input argument `timeComponent` cannot be a calendar unit. If you specify it, then its value must be `'time'`.
- When using the `retime` or `synchronize` functions:
  - The row times of the output timetable are always considered to be irregular, even when synchronized to row times that have a regular time step.
  - The `'makima'` interpolation method is not supported.
  - If the `VariableContinuity` properties of the input timetables are not constant, then this function ignores them.
  - The `'weekly'`, `'monthly'`, and `'quarterly'` time steps are not supported.
    - If the input timetables have row times that are `datetime` values, then the `'daily'` and `'yearly'` time steps also are not supported.
- When using the `timerange` function, the input argument `unitOfTime` is not supported.

Limitations that apply to classes also apply to timetables. For more information, see “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`array2timetable` | `table2timetable` | `timetable`

More About

- “Code Generation for Timetables” on page 60-2

Code Generation for MATLAB Classes

MATLAB Classes Definition for Code Generation

To generate efficient standalone code for MATLAB classes, you must use classes differently than when running your code in the MATLAB environment.

What's Different	More Information
Restricted set of language features.	"Language Limitations" on page 61-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 61-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 61-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 61-13 "Code Generation for Handle Class Destructors" on page 61-16 "Handle Object Limitations for Code Generation" on page 61-22
Calls to base class constructor.	"Calls to Base Class Constructor" on page 61-6
Global variables containing MATLAB handle objects are not supported for code generation.	N/A
Inheritance from built-in MATLAB classes is not supported.	"Inheritance from Built-In MATLAB Classes Not Supported" on page 61-7

Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events
- Listeners
- Arrays of objects
- Recursive data structures
  - Linked lists
  - Trees
  - Graphs
- Nested functions in constructors
- Overloadable operators `subsref`, `subsassign`, and `subsindex`

  In MATLAB, classes can define their own versions of the `subsref`, `subsassign`, and `subsindex` methods. Code generation does not support classes that have their own definitions of these methods.
- The `empty` method

  In MATLAB, classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.

- The following MATLAB handle class methods:
  - `addlistener`
  - `eq`
  - `findobj`
  - `findpro`
- The `AbortSet` property attribute

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

  For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

  ```
  codegen ClassNameA
  ```
- A handle class object cannot be an entry-point function input or output.
- A value class object can be an entry-point function input or output. However, if a value class object contains a handle class object, then the value class object cannot be an entry-point function input or output. A handle class object cannot be an entry-point function input or output.
- Code generation does not support global variables that are handle classes.
- You cannot use classes for Simulink signals, parameters, or data store memory.
- Code generation does not support assigning an object of a value class into a nontunable property. For example, `obj.prop=v;` is invalid when `prop` is a nontunable property and `v` is an object based on a value class.
- You cannot use `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to `coder.ceval`. You can pass class properties to `coder.ceval`.
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties” on page 61-21.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- If an object has duplicate property names and the code generator tries to constant-fold the object, code generation can fail. The code generator constant-folds an object when it is used with `coder.const`, or when it is an input to or output from a constant-folded extrinsic function.

  Duplicate property names occur in an object of a subclass in these situations:

  - The subclass has a property with the same name as a property of the superclass.
  - The subclass derives from multiple superclasses that use the same name for a property.

  For information about when MATLAB allows duplicate property names, see “Subclassing Multiple Classes”.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you do when running your code in the MATLAB environment:

- A property validation error ends a simulation with an error message. To test property validation, it is a best practice to run a simulation over the full range of input values. C/C++ code generated by Simulink Coderdoes not detect or report property validation errors.
- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

  When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of varying class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generator requires a complete assignment to each variable. Similarly, before using properties, you must explicitly define their class, size, and complexity.
- Initial values:
  - If the property does not have an explicit initial value, the code generator assumes that it is undefined at the beginning of the constructor. The code generator does not assign an empty matrix as the default.
  - If the property does not have an initial value and the code generator cannot determine that the property is assigned prior to first use, the software generates a compilation error.
  - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

    For example, for a nontunable property, you can use the following assignment:

    ```
    mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
    ```

    You cannot use the following partial assignments:

    ```
    mySystemObject.nonTunableProperty.fieldA = 'a';
    mySystemObject.nonTunableProperty.fieldB = 'b';
    ```
  - `coder.varsize` is not supported for class properties.
  - If the initial value of a property is an object, then the property must be constant. To make a property constant, declare the `Constant` attribute in the property block. For example:

    ```
    classdef MyClass
        properties (Constant)
            p1 = MyClass2;
        end
    end
    ```
  - MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns `true (1)`.
- Variable-size properties:
  - Code generation supports upper-bounded and unbounded variable-size properties for both value and handle classes.

- To generate unbounded variable-size class properties, enable dynamic memory allocation.
- To make a variable-size class property, make two sequential assignments of a class property, one to a scalar and the next to an array.

```
classdef varSizeProp1 < handle
    properties
        prop
        varProp
    end
end

function extFunc(n)
    obj = varSizeProp1;
    % Assign a scalar value to the property.
    obj.prop = 1;
    obj.varProp = 1;
    % Assign an array to the same property to make it variable-sized.
    obj.prop = 1:98;
    obj.varProp = 1:n;
end
```

  In the preceding code, the first assignment to `prop` and `varProp` is scalar, and their second assignment is to an array with the same base type. The size of `prop` has an upper bound of 98, making it an upper-bounded, variable-size property.

  If `n` is unknown at compile time, `obj.varProp` is an unbounded variable-size property. If it is known, it is an upper-bounded, variable-size class property.

- If the class property is initialized with a variable-size array, the property is variable-size.

```
classdef varSizeProp2
    properties
        prop
    end
    methods
        function obj = varSizeProp2(inVar)
            % Assign incoming value to local variable
            locVar = inVar;

            % Declare the local variable to be a variable-sized column
            % vector with no size limit
            coder.varsize('locVar',[inf 1],[1 0]);

            % Assign value
            obj.prop = locVar;
        end
    end
end
```

  In the preceding code, `inVar` is passed to the class constructor and stored in `locVar`. `locVar` is modified to be variable-size by `coder.varsize` and assigned to the class property `obj.prop`, which makes the property variable-size.

  - If the input to the function call `varSizeProp2` is variable-size, `coder.varsize` is not required.

    ```
    function z = constructCall(n)
        z = varSizeProp2(1:n);
    end
    ```

  - If the value is of `n` is unknown at compile-time and has no specified bounds, `z.prop` is an unbounded variable-size class property.
  - If the value is of `n` is unknown at compile-time and has specified bounds, `z.prop` is an upper-bounded variable-size class property.

- If a property is constant and its value is an object, you cannot change the value of a property of that object. For example, suppose that:

- `obj` is an object of `myClass1`.
- `myClass1` has a constant property `p1` that is an object of `myClass2`.
- `myClass2` has a property `p2`.

Code generation does not support the following code:

```
obj.p1.p2 = 1;
```

Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must come before `for`, `if`, `return`, `switch` or `while` statements.

For example, if you define a class `B` based on class `A`:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
            obj = obj@A(a,b);
        end

    end
end
```

Because the class definition for `B` uses an `if` statement before calling the base class constructor for `A`, you cannot generate code for function `callB`:

```
function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
end
```

However, you can generate code for `callB` if you define class `B` as:

```
classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end

    end
end

function [a,b] = getaandb(varargin)
```

```
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end
```

Inheritance from Built-In MATLAB Classes Not Supported

You cannot generate code for classes that inherit from built-in MATLAB classes. For example, you cannot generate code for the following class:

```
classdef myclass < double
```

Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 61-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 61-13

For more information, see:

- “Role of Classes in MATLAB”
- “MATLAB Classes Definition for Code Generation” on page 61-2

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

1 In a writable folder, create a MATLAB value class, `Shape`. Save the code as `Shape.m`.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out =  obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    methods(Abstract = true)
        getarea(obj);
    end
    methods(Static)
        function d = distanceBetweenShapes(shape1,shape2)
            xDist = abs(shape1.centerX - shape2.centerX);
            yDist = abs(shape1.centerY - shape2.centerY);
            d = sqrt(xDist^2 + yDist^2);
        end
    end
end
```

2 In the same folder, create a class, `Square`, that is a subclass of `Shape`. Save the code as `Square.m`.

```
classdef Square < Shape
% Create a Square at coordinates center X and center Y
% with sides of length of side
    properties
        side;
    end
    methods
        function obj = Square(side,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.side = side;
        end
        function Area = getarea(obj)
            Area = obj.side^2;
        end
    end
end
```

3. In the same folder, create a class, `Rhombus`, that is a subclass of `Shape`. Save the code as `Rhombus.m`.

```
classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)
            Area = 0.5*obj.diag1*obj.diag2;
        end
    end
end
```

4. Write a function that uses this class.

```
function [TotalArea, Distance] =   use_shape
%#codegen
s = Square(2,1,2);
r = Rhombus(3,4,7,10);
TotalArea  = s.area + r.area;
Distance = Shape.distanceBetweenShapes(s,r);
```

5. Generate a static library for `use_shape` and generate a code generation report.

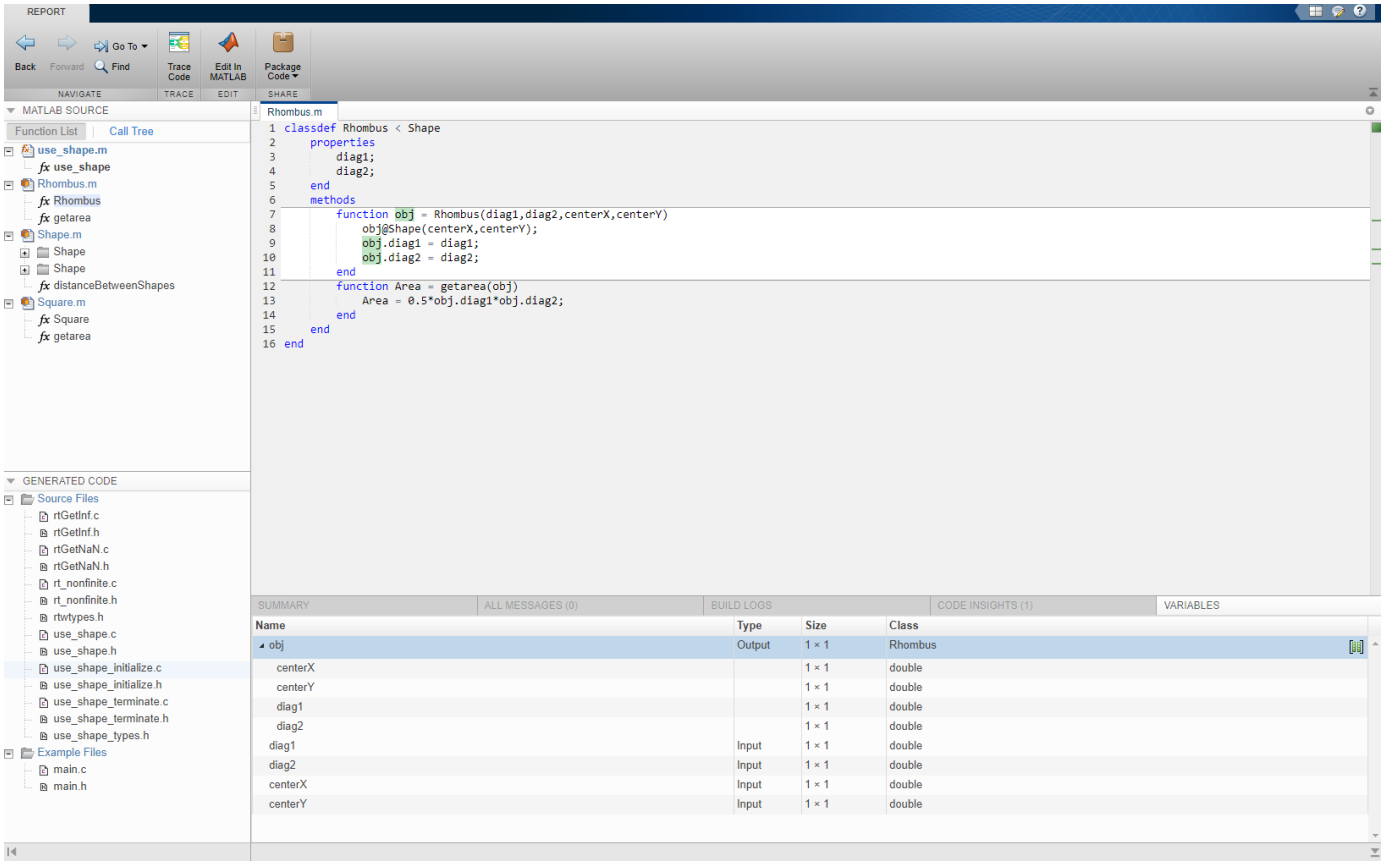
```
codegen -config:lib -report use_shape
```

`codegen` generates a C static library with the default name, `use_shape`, and supporting files in the default folder, `codegen/lib/use_shape`.

6. Click the **View report** link.

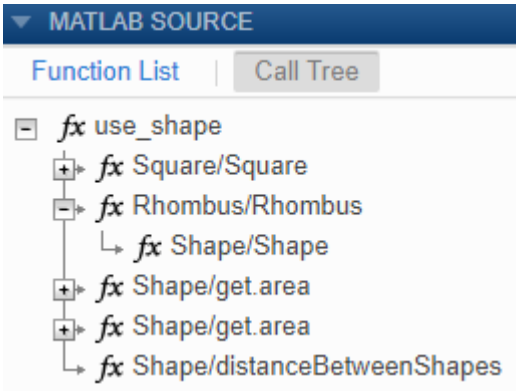
7. To see the `Rhombus` class definition, on the **MATLAB Source** pane, under `Rhombus.m`, click `Rhombus`. The `Rhombus` class constructor is highlighted.

8. Click the **Variables** tab. You see that the variable `obj` is an object of the `Rhombus` class. To see its properties, expand `obj`.



9 In the **MATLAB Source** pane, click **Call Tree**.

The **Call Tree** view shows that `use_shape` calls the `Rhombus` constructor and that the `Rhombus` constructor calls the `Shape` constructor.



10 In the code pane, in the `Rhombus` class constructor, move your pointer to this line:

```
obj@Shape(centerX,centerY)
```

The `Rhombus` class constructor calls the `Shape` method of the base `Shape` class. To view the `Shape` class definition, in `obj@Shape`, double-click `Shape`.

Shape.m

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out =  obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    methods(Abstract = true)
        getarea(obj);
    end
    methods(Static)
        function d = distanceBetweenShapes(shape1,shape2)
            xDist = abs(shape1.centerX - shape2.centerX);
            yDist = abs(shape1.centerY - shape2.centerY);
            d = sqrt(xDist^2 + yDist^2);
        end
    end
end
```

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

1. In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `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
```

2. Write a function that uses this System object.

```
function y = testAddOne(x)
%#codegen
  p = AddOne();
  y = p.step(x);
end
```

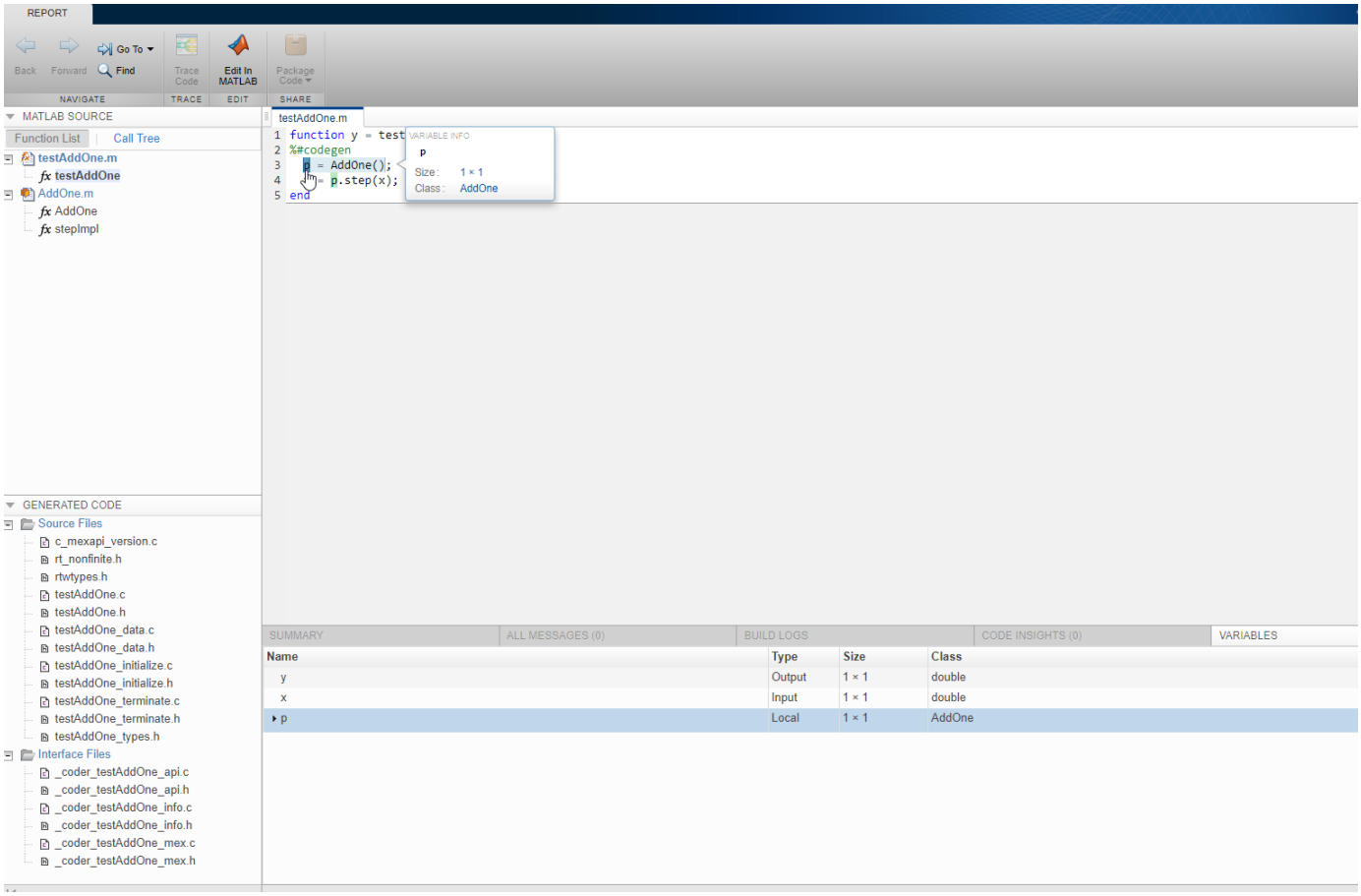
3. Generate a MEX function for this code.

```
codegen -report testAddOne -args {0}
```

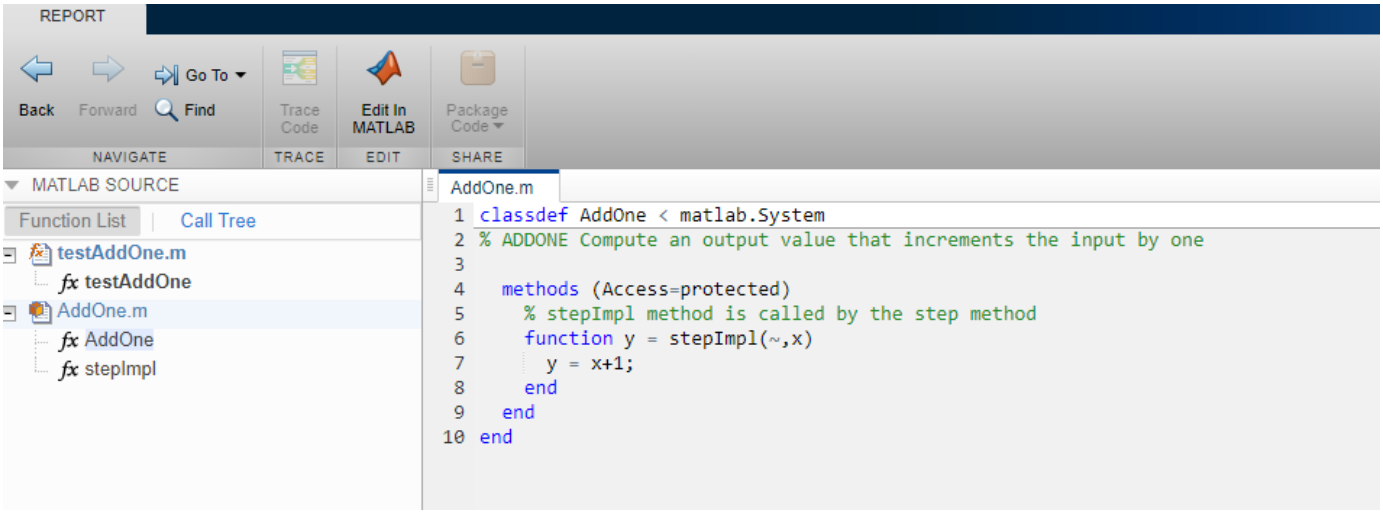
   The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur. The `-args` option specifies that the `testAddOne` function takes one scalar double input.

4. Click the **View report** link.

5. In the **MATLAB Source** pane, click `testAddOne`. To see information about the variables in `testAddOne`, click the **Variables** tab.



6 To view the class definition for `addOne`, in the **MATLAB Source** pane, click `AddOne`.



See Also

More About

- “Code Generation for Handle Class Destructors” on page 61-16

Code Generation for Handle Class Destructors

You can generate code for MATLAB code that uses `delete` methods (destructors) for handle classes. To perform clean-up operations, such as closing a previously opened file before an object is destroyed, use a `delete` method. The generated code calls the `delete` method at the end of an object's lifetime, even if execution is interrupted by a run-time error. When System objects are destroyed, `delete` calls the `release` method, which in turn calls the user-defined `releaseImpl`. For more information on when to define a `delete` method in a MATLAB code, see “Handle Class Destructor”.

Guidelines and Restrictions

When you write the MATLAB code, adhere to these guidelines and restrictions:

- Code generation does not support recursive calls of the `delete` method. Do not create an object of a certain class inside the `delete` method for the same class. This usage might cause a recursive call of `delete` and result in an error message.
- The generated code always calls the `delete` method, when an object goes out of scope. Code generation does not support explicit calls of the `delete` method.
- Initialize all properties of `MyClass` that the `delete` method of `MyClass` uses either in the constructor or as the default property value. If `delete` tries to access a property that has not been initialized in one of these two ways, the code generator produces an error message.
- Suppose a property `prop1` of `MyClass1` is itself an object (an instance of another class `MyClass2`). Initialize all properties of `MyClass2` that the `delete` method of `MyClass1` uses. Perform this initialization either in the constructor of `MyClass2` or as the default property value. If `delete` tries to access a property of `MyClass2` that has not been initialized in one of these two ways, the code generator produces an error message. For example, define the two classes `MyClass1` and `MyClass2`:

```
classdef MyClass1 < handle
    properties
        prop1
    end
    methods
        function h = MyClass1(index)
            h.prop1 = index;
        end
        function delete(h)
            fprintf('h.prop1.prop2 is: %1.0f\n',h.prop1.prop2);
        end
    end
end

classdef MyClass2 < handle
    properties
        prop2
    end
end
```

Suppose you try to generate code for this function:

```
function MyFunction
obj2 = MyClass2;
```

```
obj1 = MyClass1(obj2); % Assign obj1.prop1 to the input (obj2)
end
```

The code generator produces an error message because you have not initialized the property `obj2.prop2` that the `delete` method displays.

Behavioral Differences of Objects in Generated Code and in MATLAB

The behavior of objects in the generated code can be different from their behavior in MATLAB in these situations:

- The order of destruction of several independent objects might be different in MATLAB than in the generated code.
- The lifetime of objects in the generated code can be different from their lifetime in MATLAB. MATLAB calls the `delete` method when an object can no longer be reached from any live variable. The generated code calls the `delete` method when an object goes out of scope. In some situations, this difference causes `delete` to be called later on in the generated code than in MATLAB. For example, define the class:

  ```
  classdef MyClass < handle
      methods
          function delete(h)
              global g
              % Destructor displays current value of global variable g
              fprintf('The global variable is: %1.0f\n',g);
          end
      end
  end
  ```

  Run the function:

  ```
  function MyFunction
  global g
  g = 1;
  obj = MyClass;
  obj = MyClass;
  % MATLAB destroys the first object here
  g = 2;
  % MATLAB destroys the second object here
  % Generated code destroys both objects here
  end
  ```

  The first object can no longer be reached from any live variable after the second instance of `obj = MyClass` in `MyFunction`. MATLAB calls the `delete` method for the first object after the second instance of `obj = MyClass` in `MyFunction` and for the second object at the end of the function. The output is:

  ```
  The global variable is: 1
  The global variable is: 2
  ```

  In the generated code, both `delete` method calls happen at the end of the function when the two objects go out of scope. Running `MyFunction_mex` results in a different output:

  ```
  The global variable is: 2
  The global variable is: 2
  ```

- In MATLAB, `persistent` objects are automatically destroyed when they cannot be reached from any live variable. In the generated code, you have to call the `terminate` function explicitly to destroy the `persistent` objects.
- The generated code does not destroy partially constructed objects. If a handle object is not fully constructed at run time, the generated code produces an error message but does not call the `delete` method for that object. For a System object, if there is a run-time error in `setupImpl`, the generated code does not call `releaseImpl` for that object.

  MATLAB does call the `delete` method to destroy a partially constructed object.

See Also

More About

- “Generate Code for MATLAB Handle Classes and System Objects” on page 61-13
- “System Objects in MATLAB Code Generation” on page 61-25

Class Does Not Have Property

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
  properties
    myprop
  end
  methods
    function this = MyClass
      this.myprop = MyClass2;
    end
    function y = mymethod(this)
      y = this.myprop;
    end
  end
end

classdef MyClass2 < handle
  properties
    aa
  end
end
```

You cannot generate code for function `foo`.

```
function foo

h = MyClass;

h.mymethod().aa = 12;
```

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.

Solution

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo

h = MyClass;

b=h.mymethod();
b.aa=12;
```

See Also

More About

- “MATLAB Classes Definition for Code Generation” on page 61-2

Passing By Reference Not Supported for Some Properties

The code generator does not support passing a property by reference to an external function for these types of properties:

- A property with a get method or a set method.
- A property that uses validation functions.
- A System object property with an attribute, such as `Logical` or `PositiveInteger`, that constrains or modifies the property value.

Instead of passing a property by reference, save the property value in a temporary variable. Then, pass the temporary variable by reference to the external function. After the external function call, assign the temporary variable to the property. For example:

```
tmp = myObj.prop;
coder.ceval('myFcn', coder.ref(tmp));
myObj.prop = tmp;
```

The assignment after the `coder.ceval` call validates or modifies the property value according to the property access methods, validation functions, or attributes.

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

More About

- “MATLAB Classes Definition for Code Generation” on page 61-2

Handle Object Limitations for Code Generation

The code generator statically determines the lifetime of a handle object. When you use handle objects, this static analysis has certain restrictions.

With static analysis the generated code can reuse memory rather than rely on a dynamic memory management scheme, such as reference counting or garbage collection. The code generator can avoid dynamic memory allocation and run-time automatic memory management. These generated code characteristics are important for some safety-critical and real-time applications.

For limitations, see:

- “A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop” on page 61-22
- “A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object” on page 61-22

The code generator analyzes whether all variables are defined prior to use. Undefined variables or data types cause an error during code generation. In certain circumstances, the code generator cannot determine if references to handle objects are defined. See “References to Handle Objects Can Appear Undefined” on page 61-24.

A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop

Consider the handle class `mycls` and the function `usehandle1`. The code generator reports an error because `p`, which is outside the loop, has a property that refers to a `mycls` object created inside the loop.

```
classdef mycls < handle
   properties
       prop
   end
end

function usehandle1
p = mycls;
for i = 1:10
    p.prop = mycls;
end
```

A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object

If a persistent variable refers to a handle object, the code generator allows only one instance of the object during the program’s lifetime. The object must be a singleton object. To create a singleton handle object, enclose statements that create the object in the `if isempty()` guard for the persistent variable.

For example, consider the class `mycls` and the function `usehandle2`. The code generator reports an error for `usehandle2` because `p.prop` refers to the `mycls` object that the statement `inner = mycls` creates. This statement creates a `mycls` object for each invocation of `usehandle2`.

```
classdef mycls < handle
   properties
```

```
       prop
   end
end

function usehandle2(x)
assert(isa(x, 'double'));
persistent p;
inner = mycls;
inner.prop = x;
if isempty(p)
    p = mycls;
    p.prop = inner;
end
```

If you move the statements `inner = mycls` and `inner.prop = x` inside the `if isempty()` guard, code generation succeeds. The statement `inner = mycls` executes only once during the program's lifetime.

```
function usehandle2(x)
assert(isa(x, 'double'));
persistent p;
if isempty(p)
    inner = mycls;
    inner.prop = x;
    p = mycls;
    p.prop = inner;
end
```

Consider the function `usehandle3`. The code generator reports an error for `usehandle3` because the persistent variable `p` refers to the `mycls` object that the statement `myobj = mycls` creates. This statement creates a `mycls` object for each invocation of `usehandle3`.

```
function usehandle3(x)
assert(isa(x, 'double'));
myobj = mycls;
myobj.prop = x;
doinit(myobj);
disp(myobj.prop);
function doinit(obj)
persistent p;
if isempty(p)
    p = obj;
end
```

If you make `myobj` persistent and enclose the statement `myobj = mycls` inside an `if isempty()` guard, code generation succeeds. The statement `myobj = mycls` executes only once during the program's lifetime.

```
function usehandle3(x)
assert(isa(x, 'double'));
persistent myobj;
if isempty(myobj)
  myobj = mycls;
end

doinit(myobj);

function doinit(obj)
```

```
persistent p;
if isempty(p)
    p = obj;
end
```

References to Handle Objects Can Appear Undefined

Consider the function `refHandle` that copies a handle object property to another object. The function uses a simple handle class and value class. In MATLAB, the function runs without error.

```
function [out1, out2, out3] = refHandle()
  x = myHandleClass;
  y = x;
  v = myValueClass();
  v.prop = x;
  x.prop = 42;
  out1 = x.prop;
  out2 = y.prop;
  out3 = v.prop.prop;
end

classdef myHandleClass < handle
    properties
        prop
    end
end

classdef myValueClass
    properties
        prop
    end
end
```

During code generation, an error occurs:

```
Property 'v.prop.prop' is undefined on some execution paths.
```

Three variables reference the same memory location: `x`, `y`, and `v.prop`. The code generator determines that `x.prop` and `y.prop` share the same value. The code generator cannot determine that the handle object property `v.prop.prop` shares its definition with `x.prop` and `y.prop`. To avoid the error, define `v.prop.prop` directly.

System Objects in MATLAB Code Generation

In this section...

You can generate C/C++ code in MATLAB from your system that contains System objects by using MATLAB Coder. You can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms.

Usage Rules and Limitations for System Objects for Generating Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty()`.
- Set arguments to System object constructors as compile-time constants.
- Initialize all System objects properties that `releaseImpl` uses before the end of `setupImpl`.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- System objects accept a maximum of 1024 inputs. A maximum of eight dimensions per input is supported.
- The data type of the inputs should not change.
- The complexity of the inputs should not change.
- If you want the size of inputs to change, verify that support for variable-size is enabled. Code generation support for variable-size data also requires that variable-size support is enabled. By default in MATLAB, support for variable-size data is enabled.
- System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.
- Do not set System objects to become outputs from the MATLAB Function block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a MATLAB Function block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function. But, these functions do not generate code.

Properties

- In MATLAB System blocks, you cannot use variable-size for discrete state properties of System objects. Private properties can be variable-size.
- Objects cannot be used as default values for properties.
- You can only assign values to nontunable properties once, including the assignment in the constructor.
- Nontunable property values must be constant.
- For fixed-point inputs, if a tunable property has dependent data type properties, you can set tunable properties only at construction time or after the object is locked.
- For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

Global Variables

- Global variables are allowed in a System object, unless you are using that System object in Simulink via the MATLAB System block. See “Generate Code for Global Data” (MATLAB Coder).

Methods

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `isLocked`
  - `release`
  - `reset`
  - `set` (for tunable properties)
  - `step`
- For System objects that you define, code generation support is available only for these methods:
  - `getDiscreteStateImpl`
  - `getNumInputsImpl`
  - `getNumOutputsImpl`
  - `infoImpl`
  - `isDoneImpl`
  - `isInputDirectFeedthroughImpl`
  - `outputImpl`
  - `processTunedPropertiesImpl`
  - `releaseImpl` — Code is not generated automatically for this method. To release an object, you must explicitly call the `release` method in your code.
  - `resetImpl`
  - `setupImpl`

- `stepImpl`
- `updateImpl`
- `validateInputsImpl`
- `validatePropertiesImpl`

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Get Started with MATLAB Coder” (MATLAB Coder) and “MATLAB Classes” (MATLAB Coder) for more information.

Note Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “Implementing MATLAB Functions Using Blocks” on page 44-4.

System Objects in the MATLAB System Block

Using the MATLAB System block, you can include in a Simulink model individual System objects that you create with a class definition file. The model can then generate embeddable code. For more information, see “MATLAB System Block” on page 45-2.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

See Also

More About

- “Generate Code That Uses Row-Major Array Layout” (MATLAB Coder)

Code Generation for Function Handles

Function Handle Limitations for Code Generation

When you use function handles in MATLAB code intended for code generation, adhere to the following restrictions:

Do not use the same bound variable to reference different function handles

In some cases, using the same bound variable to reference different function handles causes a compile-time error. For example, this code does not compile:

```
function y = foo(p)
x = @plus;
if p
  x = @minus;
end
y = x(1, 2);
```

Do not pass function handles to or from `coder.ceval`

You cannot pass function handles as inputs to or outputs from `coder.ceval`. For example, suppose that `f` and `str.f` are function handles:

```
f = @sin;
str.x = pi;
str.f = f;
```

The following statements result in compilation errors:

```
coder.ceval('foo', @sin);
coder.ceval('foo', f);
coder.ceval('foo', str);
```

Do not associate a function handle with an extrinsic function

You cannot create a function handle that references an extrinsic MATLAB function.

Do not pass function handles to or from extrinsic functions

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions.

Do not pass function handles to or from entry-point functions

You cannot pass function handles as inputs to or outputs from entry-point functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as inputs. `plotFcn` attempts to call the function referenced by the `fhandle` with the input `data` and plot the results. However, this code generates a compilation error. The error indicates that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of inputs.

Do not try to view function handles from the debugger

You cannot display or watch function handles from the debugger. The function handles appear as empty matrices.

Do not use function handles for Simulink signals, parameters, or data store memory

You can use function handles in a MATLAB Function block. You cannot use function handles for Simulink signals, parameters, or data store memory.

See Also

More About

- “Declaring MATLAB Functions as Extrinsic Functions” on page 64-9

Defining Functions for Code Generation

Code Generation for Variable Length Argument Lists

When you use `varargin` and `varargout` for code generation, there are these restrictions:

- You cannot use `varargin` or `varargout` in the function definition for a top-level function in a MATLAB Function block or in a Stateflow chart that uses MATLAB as the action language.
- You cannot write to `varargin`. If you want to write to input arguments, copy the values into a local variable.
- To index into `varargin` and `varargout`, use curly braces `{}`, not parentheses `()`.
- The code generator must be able to determine the value of the index into `varargin` or `varargout`.

See Also

More About

- “Nonconstant Index into varargin or varargout in a for-Loop” on page 66-12

Code Generation for Anonymous Functions

You can use anonymous functions in MATLAB code intended for code generation. For example, you can generate code for the following MATLAB code that defines an anonymous function that finds the square of a number.

```
sqr = @(x) x.^2;
a = sqr(5);
```

Anonymous functions are useful for creating a function handle to pass to a MATLAB function that evaluates an expression over a range of values. For example, this MATLAB code uses an anonymous function to create the input to the `fzero` function:

```
b = 2;
c = 3.5;
x = fzero(@(x) x^3 + b*x + c,0);
```

Anonymous Function Limitations for Code Generation

Anonymous functions have the code generation limitations of value classes and cell arrays.

You can use anonymous functions in a MATLAB Function block. You cannot use anonymous functions for Simulink signals, parameters, or data store memory.

See Also

More About

- “MATLAB Classes Definition for Code Generation” on page 61-2
- “Cell Array Limitations for Code Generation” on page 55-6
- “Parameterizing Functions”

Code Generation for Nested Functions

You can generate code for MATLAB functions that contain nested functions. For example, you can generate code for the function `parent_fun`, which contains the nested function `child_fun`.

```
function parent_fun
x = 5;
child_fun

    function child_fun
        x = x + 1;
    end

end
```

Nested Function Limitations for Code Generation

When you generate code for nested functions, you must adhere to the code generation restrictions for value classes, cell arrays, and handle classes. You must also adhere to these restrictions:

- If the parent function declares a persistent variable, it must assign the persistent variable before it calls a nested function that uses the persistent variable.
- A nested recursive function cannot refer to a variable that the parent function uses.
- If a nested function refers to a structure variable, you must define the structure by using `struct`.
- If a nested function uses a variable defined by the parent function, you cannot use `coder.varsize` with the variable in either the parent or the nested function.

See Also

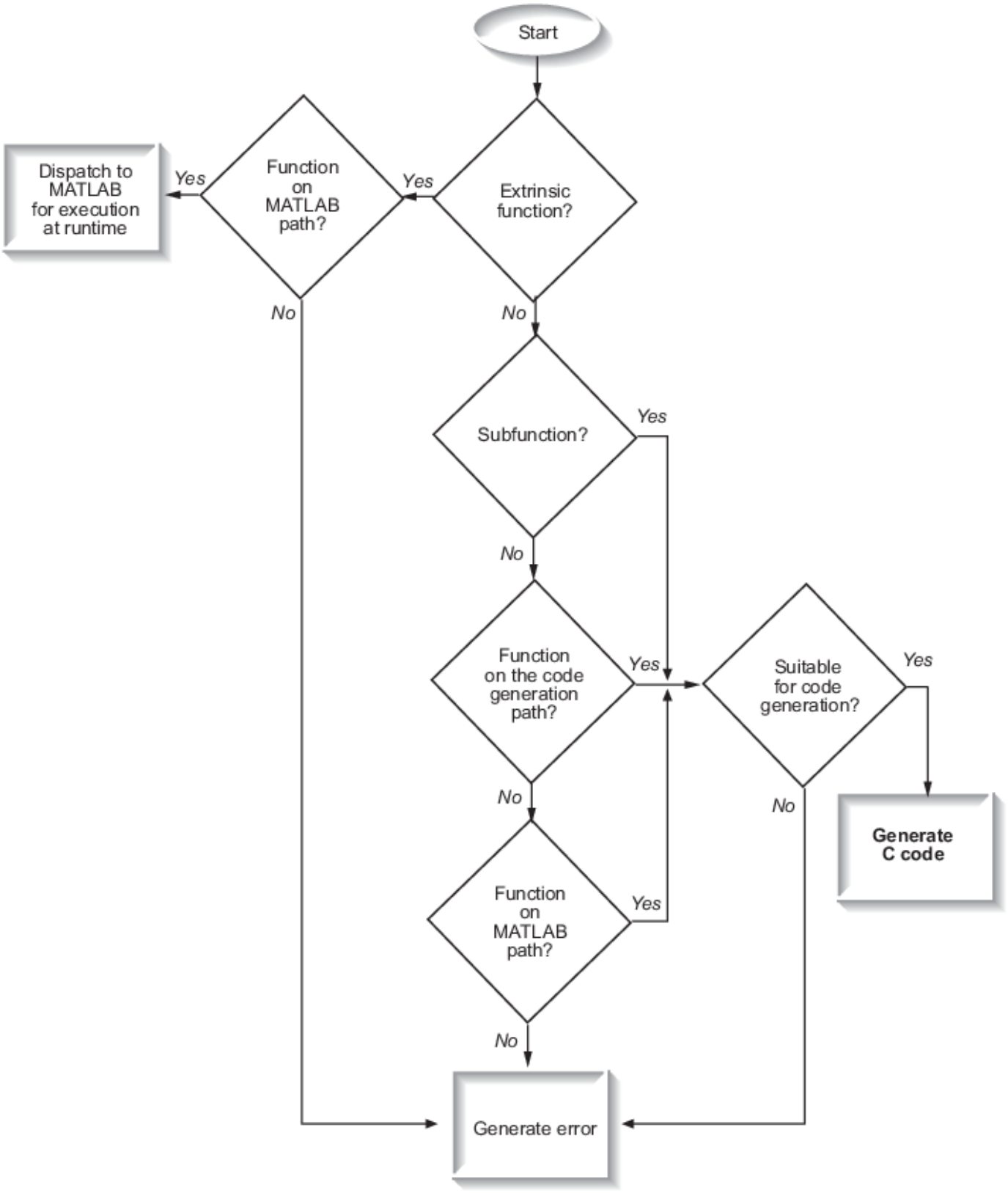
More About

- “MATLAB Classes Definition for Code Generation” on page 61-2
- “Handle Object Limitations for Code Generation” on page 61-22
- “Cell Array Limitations for Code Generation” on page 55-6
- “Code Generation for Recursive Functions” on page 64-16

Calling Functions for Code Generation

Resolution of Function Calls for Code Generation

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:


Start
Extrinsic function?
Yes
Function on MATLAB path?
Yes
Dispatch to MATLAB for execution at runtime
No
No
Subfunction?
Yes
No
Function on the code generation path?
Yes
Suitable for code generation?
Yes
No
Generate C code
No
Function on MATLAB path?
Yes
No
Generate error

Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path

  See “Compile Path Search Order” on page 64-4.
- Attempts to compile functions unless the code generator determines that it should not compile them or you explicitly declare them to be extrinsic.

  If a MATLAB function is not supported for code generation, you can declare it to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions” on page 64-9. During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, the code generator 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. If the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, compilation errors occur.

  The code generator detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.
- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 64-5

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1. Code generation path

   MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.
2. MATLAB path

   If the function is not on the code generation path, MATLAB searches this path.

MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

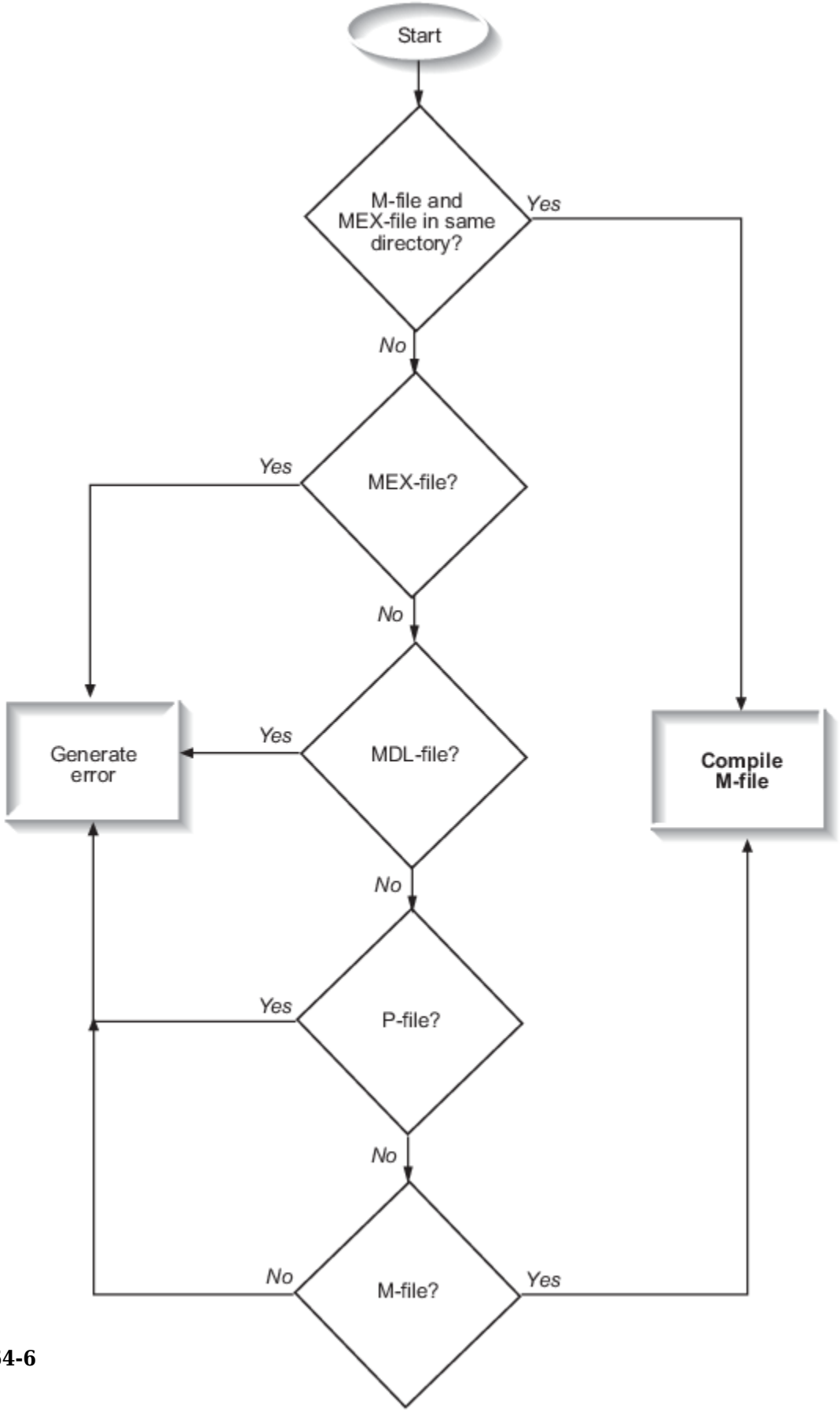
When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

For more information on how to add additional folders to the code generation path, see “Paths and File Infrastructure Setup” (MATLAB Coder).

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:


Start
M-file and MEX-file in same directory?
Yes
No
MEX-file?
Yes
No
MDL-file?
Yes
Generate error
No
P-file?
Yes
No
M-file?
No
Yes
Compile M-file

Compilation Directive %#codegen

Add the `%#codegen` directive (or pragma) to your function after the function signature to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during code generation.

```
function y = my_fcn(x) %#codegen

....
```

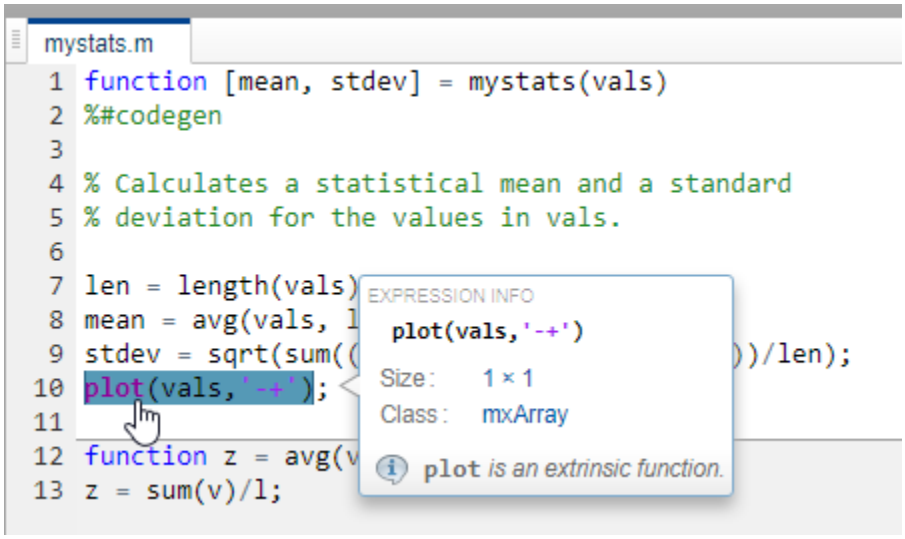
Note The `%#codegen` directive is not necessary for MATLAB Function blocks. Code inside a MATLAB Function block is always intended for code generation. The `%#codegen` directive, or the absence of it, does not change the error checking behavior.

Extrinsic Functions

When processing a call to a function `foo` in your MATLAB code, the code generator finds the definition of `foo` and generates code for its body. In some cases, you might want to bypass code generation and instead use the MATLAB engine to execute the call. Use `coder.extrinsic('foo')` to declare that calls to `foo` do not generate code and instead use the MATLAB engine for execution. In this context, `foo` is referred to as an extrinsic function. This functionality is available only when the MATLAB engine is available in MEX functions or during `coder.const` calls at compile time.

If you generate standalone code for a function that calls `foo` and includes `coder.extrinsic('foo')`, the code generator attempts to determine whether `foo` affects the output. If `foo` does not affect the output, the code generator proceeds with code generation, but excludes `foo` from the generated code. Otherwise, the code generator produces a compilation error.

The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`. For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot`, and then run the generated MEX function, the code generator dispatches calls to the `plot` function to the MATLAB engine. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.



For unsupported functions other than common visualization functions, you must declare the functions to be extrinsic (see “Resolution of Function Calls for Code Generation” on page 64-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “Resolution of Extrinsic Functions During Simulation” on page 64-12).

There are two ways to declare a function to be extrinsic:

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 64-9).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using feval” on page 64-11).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

```
coder.extrinsic('function_name_1', ... , 'function_name_n');
```

Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`. You do not have to declare `axis` as extrinsic because `axis` is one of the common visualization functions that the code generator automatically treats as extrinsic.

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
%  and displays the triangle.

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');
```

The code generator does not produce code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

To test the function, follow these steps:

1. Convert `pythagoras` to a MEX function by executing this command at the MATLAB prompt:

   ```
   codegen -report pythagoras -args {1, 1, [.3 .3 .3]}
   ```

2. Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

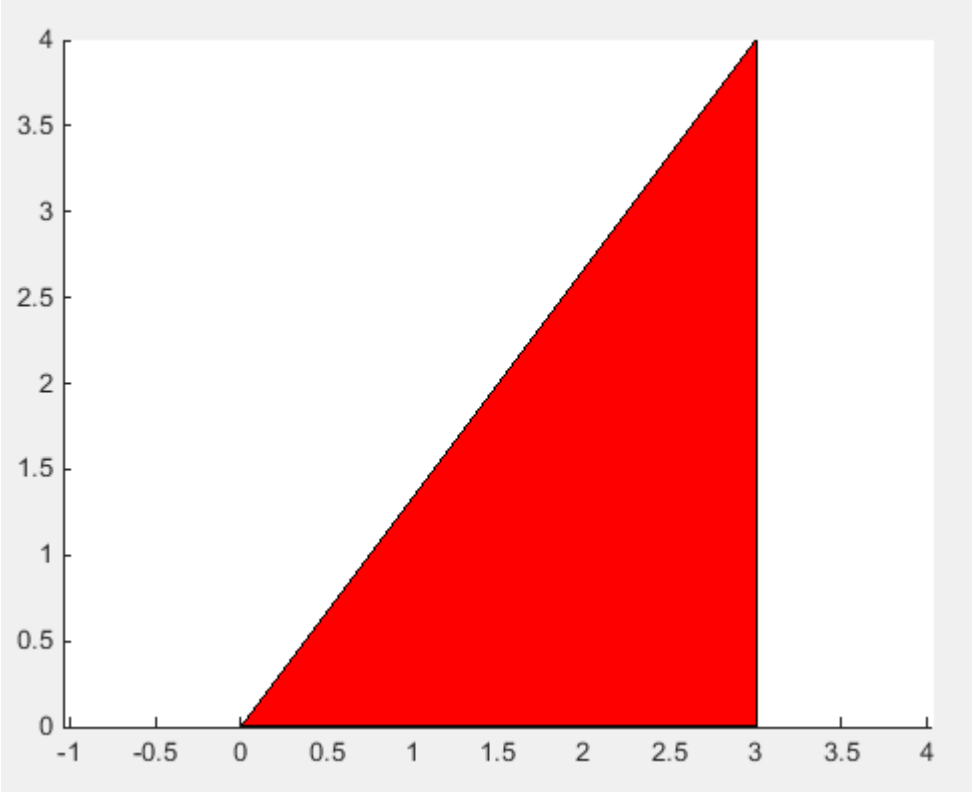
   The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.

   ```
    7 function create_plot(a, b, color)
    8 coder.extrinsic('patch');
    9 x = [0;a;a];
   10 y = [0;0;b];
   11 patch(x,y,color);
   12 axis('equal');
   13 end
   ```

3. Run the MEX function by executing this command:

   ```
   pythagoras_mex(3, 4, [1.0 0.0 0.0]);
   ```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the coder.extrinsic Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output during simulation, without generating unnecessary code (see “Resolution of Extrinsic Functions During Simulation” on page 64-12).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with mxArrays” on page 64-13).
- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 64-11).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 64-11). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using feval” on page 64-11).

Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` as treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

  ```
  function y = foo %#codegen
  coder.extrinsic('rat');
  [N D] = rat(pi);
  y = 0;
  y = mymin(N, D);

  function y = mymin(a,b)
  coder.extrinsic('min');
  y = min(a,b);
  ```

  Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the local function `mymin`.
- Call the MATLAB function using `feval`, as described in “Calling MATLAB Functions Using feval” on page 64-11.

Calling MATLAB Functions Using feval

The function `feval` is automatically interpreted as an extrinsic function during code generation. Therefore, you can use `feval` to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not compiled — which has the same result as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

The code generator does not support the use of `feval` to call local functions or functions that are located in a private folder.

Extrinsic Declaration for Nonstatic Methods

Suppose that you define a class `myClass` that has a nonstatic method `foo`, and then create an instance `obj` of this class. If you want to declare the method `obj.foo` as extrinsic in your MATLAB code that you intend for code generation, follow these rules:

- Write the call to `foo` as a function call. Do not write the call by using the dot notation.
- Declare `foo` to be extrinsic by using the syntax `coder.extrinsic('foo')`.

For example, define `myClass` as:

```
classdef myClass
    properties
        prop = 1
    end
    methods
        function y = foo(obj,x)
            y = obj.prop + x;
        end
    end
end
```

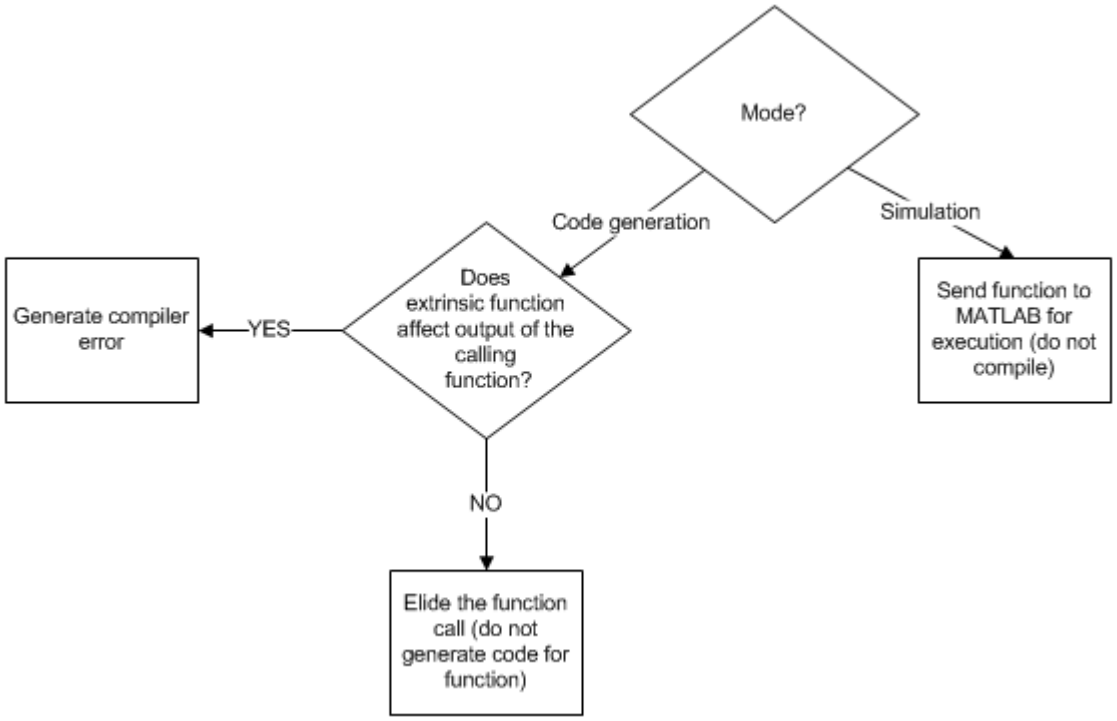
Here is an example MATLAB function that declares `foo` as extrinsic.

```
function y = myFunction(x) %#codegen
coder.extrinsic('foo');
obj = myClass;
y = foo(obj,x);
end
```

Nonstatic methods are also known as ordinary methods. See “Define Class Methods and Functions”.

Resolution of Extrinsic Functions During Simulation

The code generator resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, the code generator 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 (see “Working with mxArrays” on page 64-13). Provided that the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, the code generator issues a compiler error.

Working with mxArrays

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables
- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting mxArrays to Known Types” on page 64-13.

Converting mxArrays to Known Types

To convert an `mxArray` to a known type, assign the `mxArray` to a variable whose type is defined. At run time, the `mxArray` is converted to the type of the variable assigned to it. However, if the data in the `mxArray` is not consistent with the type of the variable, you get a run-time error.

For example, consider this code:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level function `foo` calls the extrinsic function `rat`, which returns two `mxArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxArrays` to another MATLAB function — in this case, `min` — you cannot assign the `mxArray` returned by `min` to the output `y`.

If you run this function `foo` in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

```
Function output 'y' cannot be of MATLAB type.
```

To fix this problem, define `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller, or read or write to the caller workspace do not work during code generation. Such functions include:
  - `dbstack`
  - `evalin`
  - `assignin`
  - `save`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code can produce unpredictable results if your extrinsic function performs the following actions at run time:
  - Change folders
  - Change the MATLAB path
  - Delete or add MATLAB files
  - Change warning states
  - Change MATLAB preferences
  - Change Simulink parameters
- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Code Generation for Recursive Functions

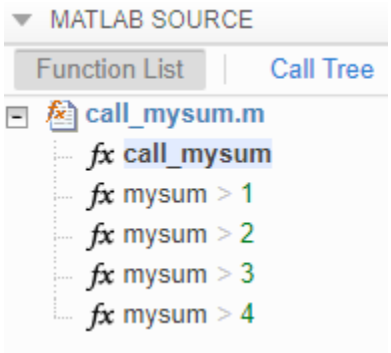
To generate code for recursive MATLAB functions, the code generator uses compile-time recursion on page 64-16 or run-time recursion on page 64-16. You can influence whether the code generator uses compile-time or run-time recursion by modifying your MATLAB code. See “Force Code Generator to Use Run-Time Recursion” on page 64-18.

You can disallow recursion on page 64-17 or disable run-time recursion on page 64-17 by modifying configuration parameters.

When you use recursive functions in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Recursive Function Limitations for Code Generation” on page 64-17.

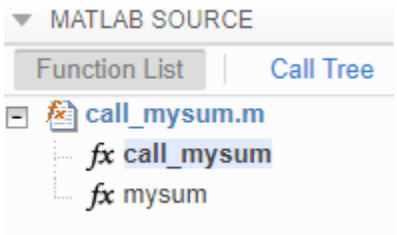
Compile-Time Recursion

With compile-time recursion, the code generator creates multiple versions of a recursive function in the generated code. The inputs to each version have values or sizes that are customized for that version. These versions are known as function specializations. You can tell that the code generator used compile-time recursion by looking at the MATLAB Function report or the generated C code. Here is an example of compile-time recursion in the report.



Run-Time Recursion

With run-time recursion, the code generator produces a recursive function in the generated code. You can tell that the code generator used run-time recursion by looking at the MATLAB Function report or the generated C code. Here is an example of run-time recursion in the report.



Disallow Recursion

In the model configuration parameters, set **Compile-time recursion limit for MATLAB functions** to 0.

Disable Run-Time Recursion

Some coding standards, such as MISRA®, do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C®, disable run-time recursion.

In the model configuration parameters, clear the **Enable run-time recursion for MATLAB functions** check box.

If your code requires run-time recursion and run-time recursion is disabled, you must rewrite your code so that it uses compile-time recursion or does not use recursion.

Recursive Function Limitations for Code Generation

When you use recursion in MATLAB code that is intended for code generation, follow these restrictions:

- The top-level function in a MATLAB Function block cannot be a recursive function, but it can call a recursive function.
- Assign all outputs of a run-time recursive function before the first recursive call in the function.
- Assign all elements of cell array outputs of a run-time recursive function.
- Inputs and outputs of run-time recursive functions cannot be classes.
- The **Maximum stack size** parameter is ignored for run-time recursion.

See Also

More About

- “Force Code Generator to Use Run-Time Recursion” on page 64-18
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 66-5
- “Compile-Time Recursion Limit Reached” on page 66-2
- “Compile-time recursion limit for MATLAB functions”
- “MATLAB Function Reports” on page 44-41

Force Code Generator to Use Run-Time Recursion

When your MATLAB code includes recursive function calls, the code generator uses compile-time or run-time recursion. With compile-time recursion on page 64-16, the code generator creates multiple versions of the recursive function in the generated code. These versions are known as function specializations. With run-time recursion on page 64-16, the code generator produces a recursive function. If compile-time recursion results in too many function specializations or if you prefer run-time recursion, you can try to force the code generator to use run-time recursion. Try one of these approaches:

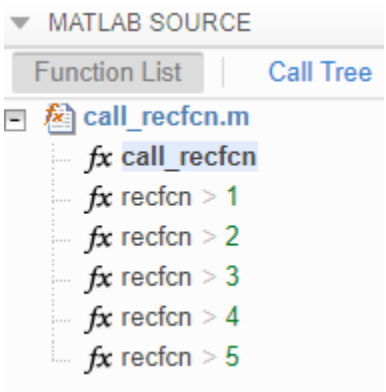
- "Treat the Input to the Recursive Function as a Nonconstant" on page 64-18
- "Make the Input to the Recursive Function Variable-Size" on page 64-19
- "Assign Output Variable Before the Recursive Call" on page 64-20

Treat the Input to the Recursive Function as a Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 5;
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

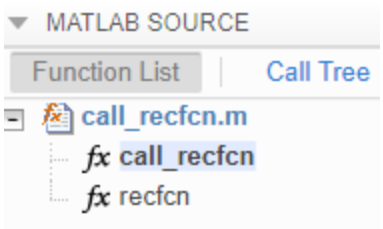
`call_recfcn` calls `recfcn` with the value 5 for the second argument. `recfcn` calls itself recursively until `x` is 1. For each `recfcn` call, the input argument `x` has a different value. The code generator produces five specializations of `recfcn`, one for each call. You can see the specializations in the MATLAB Function report.



To force run-time recursion, in `call_recfcn`, in the call to `recfcn`, instruct the code generator to treat the value of the input argument `x` as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
A = ones(1,n);
x = coder.ignoreConst(5);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

In the MATLAB Function report, you see only one specialization.



Make the Input to the Recursive Function Variable-Size

Consider this code:

```
function z = call_mysum(A)
%#codegen
z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. To force the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```
function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
```

```
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Assign Output Variable Before the Recursive Call

The code generator uses compile-time recursion for this code:

```
function y = callrecursive(n)
x = 10;
y = myrecursive(x,n);
end

function y = myrecursive(x,n)
coder.inline('never')
if x > 1
    y = n + myrecursive(x-1,n-1);

else
    y = n;
end
end
```

To force the code generator to use run-time recursion, modify `myrecursive` so that the output `y` is assigned before the recursive call. Place the assignment `y = n` in the `if` block and the recursive call in the `else` block.

```
function y = callrecursive(n)
x = 10;
y = myrecursive(x,n);
end

function y = myrecursive(x,n)
coder.inline('never')
if x == 1
    y = n;
else
    y = n + myrecursive(x-1,n-1);
end
end
```

See Also

More About

- “Code Generation for Recursive Functions” on page 64-16
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 66-5
- “Compile-Time Recursion Limit Reached” on page 66-2

Avoid Duplicate Functions in Generated Code

Issue

You generate code and it contains multiple, duplicate copies of the same functions, with only slight differences, such as modifications to the function signature. For example, your generated code might contain functions called `foo` and `b_foo`. Duplicate functions can make the generated code more difficult to analyze and manage.

Cause

Duplicate functions in the generated code are the result of function specializations. The code generator specializes functions when it detects that they differ at different call sites by:

- Number of input or output variables.
- Type of input or output variables.
- Size of input or output variables.
- Values of input variables.

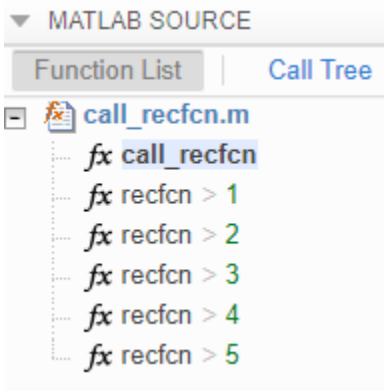
In some cases, these specializations are necessary for the generated C/C++ code because C/C++ functions do not have the same flexibility as MATLAB functions. In other cases, the code generator specializes functions to optimize the generated code or because of a lack of information.

Solution

In certain cases, you can alter your MATLAB code to avoid the generation of duplicate functions.

Identify Duplicate Functions by Using Code Generation Report

You can determine whether the code generator created duplicate functions by inspecting the code generation report or in Simulink, the MATLAB Function report. The report shows a list of the duplicate functions underneath the entry-point function. For example:



Duplicate Functions Generated for Multiple Input Sizes

If your MATLAB code calls a function multiple times and passes inputs of different sizes, the code generator can create specializations of the function for each size. To avoid this issue, use

`coder.ignoreSize` on the function input. For example, this code uses `coder.ignoreSize` to avoid creating multiple copies of the function `indexOf`:

```
function [out1, out2] = test1(in)
  a = 1:10;
  b = 2:40;
  % Without coder.ignoreSize duplicate functions are generated
  out1 = indexOf(coder.ignoreSize(a), in);
  out2 = indexOf(coder.ignoreSize(b), in);
end

function index = indexOf(array, value)
  coder.inline('never');
  for i = 1:numel(array)
      if array(i) == value
          index = i;
          return
      end
  end
  index = -1;
  return
end
```

To generate code, enter:

```
codegen test1 -config:lib -report -args {1}
```

Duplicate Functions Generated for Different Input Values

If your MATLAB code calls a function and passes multiple different constant inputs, the code generator can create specializations of the function for each different constant. In this case, use `coder.ignoreConst` to indicate to the code generator not to treat the value as an immutable constant. For example:

```
function [out3, out4] = test2(in)
  c = ['a', 'b', 'c'];
  if in > 0
      c(2)='d';
  end
  out3 = indexOf(c, coder.ignoreConst('a'));
  out4 = indexOf(c, coder.ignoreConst('b'));
end


function index = indexOf(array, value)
  coder.inline('never');
  for i = 1:numel(array)
      if array(i) == value
          index = i;
          return
      end
  end
  index = -1;
  return
end
```

To generate code, enter:

```
codegen test2 -config:lib -report -args {1}
```

Duplicate Functions Generated for Different Number of Outputs

If your MATLAB code calls a function and accepts a different number of outputs at different call sites, the code generator can produce specializations for each call. For example:

```
[a b] = foo();
c = foo();
```

To make each call return the same number of outputs and avoid duplicate functions, use the ~ symbol:

```
[a b] = foo();
[c, ~] = foo();
```

See Also

`coder.ignoreConst` | `coder.ignoreSize` | `coder.varsize`

More About

- “MATLAB Function Reports” on page 44-41
- “Force Code Generator to Use Run-Time Recursion” on page 64-18

Improve Run-Time Performance of MATLAB Function Block

Avoid Data Copies of Function Inputs in Generated Code

You can reduce the number of copies in your generated code by writing functions that use the same variable as both an input and an output. For example:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, the generated code passes the variable by reference instead of redundantly copying the input to a temporary variable. In the preceding example, input `A` is passed by reference in the generated code because it also acts as an output for function `foo`:

```
...
/* Function Definitions */
void foo(double *A, double B)
{
    *A *= B;
}
...
```

The reference parameter optimization reduces memory usage and execution time, especially when the variable passed by reference is a large data structure. To achieve these benefits at the call site, call the function with the same variable as both input and output.

By contrast, suppose that you rewrite function `foo` without the optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

The generated code passes the inputs by value and returns the value of the output:

```
...
/* Function Definitions */
double foo2(double A, double B)
{
   return A * B;
}
...
```

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
  x1=u1+1;
  y1=bar(x1);
end

function y2=bar(u2)
  % Since foo does not use x1 later in the function,
  % it would be optimal to do this operation in place
  x2=u2.*2;
  % The change in dimensions in the following code
```

```
  % means that it cannot be done in place
  y2=[x2,x2];
end
```

You can modify the code to eliminate redundant copies.

```
function y1=foo(u1) %#codegen
  u1=u1+1;
  [y1, u1]=bar(u1);
end

function [y2, u2]=bar(u2)
    u2=u2.*2;
  % The change in dimensions in the following code
  % still means that it cannot be done in place
  y2=[u2,u2];
end
```

The reference parameter optimization does not apply to constant inputs. If the same variable is an input and an output, and the input is constant, the code generator treats the output as a separate variable. For example, consider the function `foo`:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

Generate code in which `A` has a constant value `2`.

```
codegen -config:lib foo -args {coder.Constant(2) 3} -report
```

The generated code defines the constant `A` and returns the value of the output.

```
...
#define A                              (2.0)
...
double foo(double B)
{
  return A * B;
}
...
```

See Also

Inline Code

Inlining is a 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. The code generator uses internal heuristics to determine whether to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

See Also

Unroll for-Loops

When the code generator unrolls a `for`-loop, instead of producing a `for`-loop in the generated code, it produces a copy of the loop body for each iteration. For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can significantly increase code generation time and generate inefficient code.

Force Loop Unrolling by Using coder.unroll

The code generator uses heuristics to determine when to unroll a `for`-loop. To force loop unrolling, use `coder.unroll`. This affects only the `for` loop that is immediately after `coder.unroll`. For example:

```
function z = call_myloop()
%#codegen
z = myloop(5);
end

function b = myloop(n)
b = zeros(1,n);
coder.unroll();
for i = 1:n
    b(i)=i+n;
end
end
```

Here is the generated code for the for-loop:

```
  z[0] = 6.0;
  z[1] = 7.0;
  z[2] = 8.0;
  z[3] = 9.0;
  z[4] = 10.0;
```

To control when a `for`-loop is unrolled, use the `coder.unroll flag` argument. For example, unroll the loop only when the number of iterations is less than 10.

```
function z = call_myloop()
%#codegen
z = myloop(5);
end

function b = myloop(n)
unroll_flag = n < 10;
b = zeros(1,n);
coder.unroll(unroll_flag);
for i = 1:n
    b(i)=i+n;
end
end
```

To unroll a `for`-loop, the code generator must be able to determine the bounds of the `for`-loop. For example, code generation fails for the following code because the value of `n` is not known at code generation time.

```
function b = myloop(n)
b = zeros(1,n);
```

```
coder.unroll();
for i = 1:n
    b(i)=i+n;
end
end
```

See Also

`coder.unroll`

More About

- “Nonconstant Index into varargin or varargout in a for-Loop” on page 66-12

Generate Reusable Code

With MATLAB, you can generate reusable code in the following ways:

- Write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or MATLAB function library.
- Compile external functions on the MATLAB path and integrate them into generated C code for embedded targets.

See “Resolution of Function Calls for Code Generation” (MATLAB Coder).

Common applications include:

- Overriding generated library function with a custom implementation.
- Implementing a reusable library on top of standard library functions that can be used with Simulink.
- Swapping between different implementations of the same function.

LAPACK Calls for Linear Algebra in a MATLAB Function Block

To improve the simulation speed of MATLAB Function block algorithms that call certain linear algebra functions, Simulink can call LAPACK functions. LAPACK is a software library for numerical linear algebra. If the input arrays for the linear algebra functions meet certain criteria, the simulation calls LAPACK functions in the LAPACK library that is included with MATLAB. MATLAB uses LAPACK in some linear algebra functions such as `eig` and `svd`.

If you use Simulink Coder to generate code for these algorithms, you can specify that the code generator produce LAPACK function calls. The code generator uses the LAPACKE C interface to LAPACK. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet the criteria, the code generator produces LAPACK calls. The build process links to the LAPACK library that you specify. See “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” (Simulink Coder).

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

External Websites

- www.netlib.org/lapack

BLAS Calls for Matrix Operations in a MATLAB Function Block

To improve the simulation speed of MATLAB Function block algorithms that call certain low-level vector and matrix functions (such as matrix multiplication), Simulink can call BLAS functions. BLAS is a software library for low-level vector and matrix computations that has several highly optimized machine-specific implementations. If the input arrays for the matrix functions meet certain criteria, the simulation calls BLAS functions in the BLAS library that is included with MATLAB.

If you use Simulink Coder to generate code for these algorithms, you can specify that the code generator produce BLAS function calls. The code generator uses the CBLAS C interface to BLAS. If you specify that you want to generate BLAS calls, and the input arrays for the matrix functions meet the criteria, the code generator produces BLAS calls. The build process links to the BLAS library that you specify. See “Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” (Simulink Coder).

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

External Websites

- https://www.netlib.org/blas/

FFTW calls for fast Fourier transform functions in a MATLAB Function Block

To execute MATLAB Function block algorithms that call MATLAB fast Fourier transform (FFT) functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, or `ifftn`), Simulink uses the library that MATLAB uses for FFT algorithms.

If you use Simulink Coder to generate code for these algorithms, by default, the code generator produces code for the FFT algorithms instead of producing FFT library calls. To increase the speed of fast Fourier transforms in generated code, you can specify that the code generator produce calls to a specific installed FFTW library. See “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder).

See Also

Related Examples

- “Create Custom Functionality Using MATLAB Function Block” on page 44-6
- “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” (Simulink Coder)

More About

- “Implementing MATLAB Functions Using Blocks” on page 44-4

External Websites

- http://www.fftw.org/

Troubleshooting MATLAB Code in MATLAB Function Blocks

Compile-Time Recursion Limit Reached

Issue

You see a message such as:

```
Compile-time recursion limit reached. Size or type of
input #1 of function 'foo' may change at every call.
```

```
Compile-time recursion limit reached. Value of input #1
of function 'foo' may change at every call.
```

Cause

With compile-time recursion, the code generator produces multiple versions of the recursive function instead of producing a recursive function in the generated code. These versions are known as function specializations. The code generator is unable to use compile-time recursion for a recursive function in your MATLAB code because the number of function specializations exceeds the limit.

Solutions

To address the issue, try one of these solutions:

- “Force Run-Time Recursion” on page 66-2
- “Increase the Compile-Time Recursion Limit” on page 66-4

Force Run-Time Recursion

- For this message:

  ```
  Compile-time recursion limit reached. Value of input #1
  of function 'foo' may change at every call.
  ```

  Use this solution:

  “Force Run-Time Recursion by Treating the Input Value as Nonconstant” on page 66-2.
- For this message:

  ```
  Compile-time recursion limit reached. Size or type of
  input #1 of function 'foo' may change at every call.
  ```

  In the MATLAB Function report, look at the function specializations. If you can see that the size of an argument is changing for each function specialization, then try this solution:

  “Force Run-Time Recursion by Making the Input Variable-Size” on page 66-3.

Force Run-Time Recursion by Treating the Input Value as Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 100;
```

```
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

The second input to `recfcn` has the constant value 100. The code generator determines that the number of recursive calls is finite and tries to produce 100 copies of `recfcn`. This number of specializations exceeds the compile-time recursion limit. To force run-time recursion, instruct the code generator to treat the second input as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
A = ones(1,n);
x = coder.ignoreConst(100);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

If the code generator cannot determine that the number of recursive calls is finite, it produces a run-time recursive function.

Force Run-Time Recursion by Making the Input Variable-Size

Consider this function:

```
function z = call_mysum(A)
%#codegen
z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. If `A` is large enough, the number of function specializations exceeds the compile-time limit. To cause the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```
function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Increase the Compile-Time Recursion Limit

The default compile-time recursion limit of 50 is large enough for most recursive functions that require compile-time recursion. Usually, increasing the limit does not fix the issue. However, if you can determine the number of recursive calls and you want compile-time recursion, increase the limit. For example, consider this function:

```
function z = call_mysum()
%#codegen
B = 1:125;
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

You can determine that the code generator produces 125 copies of the `mysum` function. In this case, if you want compile-time recursion, increase the compile-time recursion limit to 125.

To increase the limit, increase the value of the **Compile-time recursion limit for MATLAB functions** configuration parameter.

See Also

More About

- “Code Generation for Recursive Functions” on page 64-16
- “Compile-time recursion limit for MATLAB functions”

Output Variable Must Be Assigned Before Run-Time Recursive Call

Issue

You see one of these messages:

```
All outputs must be assigned before any run-time
recursive call. Output 'y' is not assigned here.

Simulink does not have enough information to determine output
sizes for this block
```

.

Cause

Run-time recursion produces a recursive function in the generated code. The code generator is unable to use run-time recursion for a recursive function in your MATLAB code because an output is not assigned before the first recursive call.

Solution

Rewrite the code so that it assigns the output before the recursive call.

Direct Recursion Example

In the following code, the statement `y = A(1)` assigns a value to the output `y`. This statement occurs after the recursive call `y = A(1)+ mysum(A(2:end))`.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) > 1
    y = A(1)+ mysum(A(2:end));

else
    y = A(1);
end
end
```

Rewrite the code so that assignment `y = A(1)` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end
```

```
function y = mysum(A)
coder.inline('never');

if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Alternatively, before the `if` block, add an assignment, for example, `y = 0`.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
y = 0;
if size(A,2) > 1
    y = A(1)+ mysum(A(2:end));

else
    y = A(1);
end
end
```

Indirect Recursion Example

In the following code, `rec1` calls `rec2` before the assignment `y = 0`.

```
function z = callrec(n)
z = rec1(n);
end

function y = rec1(x)
%#codegen

if x >= 0
    y = rec2(x-1)+1;
else
    y = 0;
end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

Rewrite this code so that in `rec1`, the assignment `y = 0` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function z = callrec(n)
z = rec1(n);
end
```

```
function y = rec1(x)
%#codegen

if x < 0
    y = 0;
else
    y = rec2(x-1)+1;
end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

See Also

More About

- “Code Generation for Recursive Functions” on page 64-16

Unable to Determine That Every Element of Cell Array Is Assigned

Issue

You see one of these messages:

```
Unable to determine that every element of 'y' is
assigned before this line.
```

```
Unable to determine that every element of 'y' is
assigned before exiting the function.
```

```
Unable to determine that every element of 'y' is
assigned before exiting the recursively called function.
```

Cause

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array.

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. The code generator detects that all elements are assigned when the code follows this pattern:

```
function z = CellVarSize1D(n, j)
%#codegen
assert(n < 100);
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

Here is the pattern for a multidimensional cell array:

```
function z = CellAssign3D(m,n,p)
%#codegen
assert(m < 100);
assert(n < 100);
assert(p < 100);
x = cell(m,n,p);
for i = 1:m
    for j =1:n
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

If the code generator detects that some elements are not assigned, code generation fails. Sometimes, even though your code assigns all elements of the cell array, code generation fails because the analysis does not detect that all elements are assigned.

Here are examples where the code generator is unable to detect that elements are assigned:

- Elements are assigned in different loops

```
...
x = cell(1,n)
for i = 1:5
    x{i} = 5;
end
for i = 6:n
    x{i} = 7;
end
...
```

- The variable that defines the loop end value is not the same as the variable that defines the cell dimension.

```
...
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
...
```

For more information, see “Definition of Variable-Size Cell Array by Using cell” on page 55-7.

Solution

Try one of these solutions:

- “Use recognized pattern for assigning elements” on page 66-9
- “Use repmat” on page 66-9
- “Use coder.nullcopy” on page 66-10

Use recognized pattern for assigning elements

If possible, rewrite your code to follow this pattern:

```
...
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
...
```

Use repmat

Sometimes, you can use `repmat` to define the variable-size cell array.

Consider this code that defines a variable-size cell array. It assigns the value 1 to odd elements and the value 2 to even elements.

```
function z = repDefine(n, j)
%#codegen
assert(n < 100);
c =cell(1,n);
for i = 1:2:n-1
    c{i} = 1;
end
for i = 2:2:n
    c{i} = 2;
end
z = c{j};
```

Code generation does not allow this code because:

- More than one loop assigns the elements.
- The loop counter does not increment by 1.

Rewrite the code to first use `cell` to create a 1-by-2 cell array whose first element is 1 and whose second element is 2. Then, use `repmat` to create a variable-size cell array whose element values alternate between 1 and 2.

```
function z = repVarSize(n, j)
%#codegen
assert(n < 100);
c = cell(1,2);
c{1} = 1;
c{2} = 2;
c1= repmat(c,1,n);
z = c1{j};
end
```

You can pass an initially empty, variable-size cell array into or out of a function by using `repmat`. Use the following pattern:

```
function x = emptyVarSizeCellArray
x = repmat({'abc'},0,0);
coder.varsize('x');
end
```

This code indicates that `x` is an empty, variable-size cell array of `1x3` characters that can be passed into or out of functions.

Use coder.nullcopy

As a last resort, you can use `coder.nullcopy` to indicate that the code generator can allocate the memory for your cell array without initializing the memory. For example:

```
function z = nulcpyCell(n, j)
%#codegen
assert(n < 100);
c =cell(1,n);
c1 = coder.nullcopy(c);
for i = 1:4
    c1{i} = 1;
end
for i = 5:n
    c1{i} = 2;
```

```
end
z = c1{j};
end
```

Use `coder.nullcopy` with caution. If you access uninitialized memory, results are unpredictable.

See Also

`cell` | `coder.nullcopy` | `repmat`

More About

- “Cell Array Limitations for Code Generation” on page 55-6

Nonconstant Index into varargin or varargout in a for-Loop

Issue

Your MATLAB code contains a `for`-loop that indexes into `varargin` or `varargout`. When you generate code, you see this error message:

```
Non-constant expression or empty matrix. This expression
must be constant because its value determines the size
or class of some expression.
```

Cause

At code generation time, the code generator must be able to determine the value of an index into `varargin` or `varagout`. When `varargin` or `varagout` are indexed in a `for`-loop, the code generator determines the index value for each loop iteration by unrolling the loop. Loop unrolling makes a copy of the loop body for each loop iteration. In each iteration, the code generator determines the value of the index from the loop counter.

The code generator is unable to determine the value of an index into `varargin` or `varagout` when:

- The number of copies of the loop body exceeds the limit for loop unrolling.
- Heuristics fail to identify that loop unrolling is warranted for a particular `for`-loop. For example, consider the following function:

  ```
  function [x,y,z] = fcn(a,b,c)
  %#codegen

  [x,y,z] = subfcn(a,b,c);

  function varargout = subfcn(varargin)
  j = 0;
  for i = 1:nargin
      j = j+1;
      varargout{j} = varargin{j};
  end
  ```

  The heuristics do not detect the relationship between the index `j` and the loop counter `i`. Therefore, the code generator does not unroll the `for`-loop.

Solution

Use one of these solutions:

- “Force Loop Unrolling” on page 66-12
- “Rewrite the Code” on page 66-13

Force Loop Unrolling

Force loop unrolling by using `coder.unroll`. For example:

```
function [x,y,z] = fcn(a,b,c)
%#codegen
```

```
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;

coder.unroll();
for i = 1:nargin
    j = j + 1;
    varargout{j} = varargin{j};
end
```

Rewrite the Code

Rewrite the code so that the code generator can detect the relationship between the index and the loop counter. For example:

```
function [x,y,z] = fcn(a,b,c)
%#codegen
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:nargin
    varargout{i} = varargin{i};
end
```

See Also

`coder.unroll`

More About

- “Code Generation for Variable Length Argument Lists” on page 63-2
- “Unroll for-Loops” on page 65-5

Unknown Output Type for coder.ceval

Issue

You see this error message:

```
Output of 'coder.ceval' has unknown type. The enclosing
expression cannot be evaluated.
Specify the output type by assigning the output of
'coder.ceval' to a variable with a known type.
```

Cause

This error message occurs when the code generator cannot determine the output type of a `coder.ceval` call.

Solution

Initialize a temporary variable with the expected output type. Assign the output of `coder.ceval` to this variable.

Example

Assume that you have a C function called `cFunctionThatReturnsDouble`. You want to generate C library code for a function `foo`. The code generator returns the error message because it cannot determine the return type of `coder.ceval`.

```
function foo
%#codegen
callFunction(coder.ceval('cFunctionThatReturnsDouble'));
end

function callFunction(~)
end
```

To fix the error, define the type of the C function output by using a temporary variable.

```
function foo
%#codegen
temp = 0;
temp = coder.ceval('cFunctionThatReturnsDouble');
callFunction(temp);
end

function callFunction(~)
end
```

You can also use `coder.opaque` to initialize the temporary variable.

Example Using Classes

Assume that you have a class with a custom `set` method. This class uses the `set` method to ensure that the object property value falls within a certain range.

```
classdef classWithSetter
   properties
      expectedResult = []
   end
   properties(Constant)
      scalingFactor = 0.001
   end
   methods
      function obj = set.expectedResult(obj,erIn)
         if erIn >= 0 && erIn <= 100
            erIn = erIn.*obj.scalingFactor;
            obj.expectedResult = erIn;
         else
            obj.expectedResult = NaN;
         end
      end
   end
end
```

When generating C library code for the function `foo`, the code generator produces the error message. The input type into the `set` method cannot be determined.

```
function foo
%#codegen
obj = classWithSetter;
obj.expectedResult = coder.ceval('cFunctionThatReturnsDouble');
end
```

To fix the error, initialize a temporary variable with a known type. For this example, use a type of scalar double.

```
function foo
%#codegen
obj = classWithSetter;
temp = 0;
temp = coder.ceval('cFunctionThatReturnsDouble');
obj.expectedResult = temp;
end
```

See Also

`coder.ceval` | `coder.opaque`

Managing Data

Working with Data

About Data Types in Simulink

In this section...

About Data Types

The term *data type* refers to the way in which a computer represents numbers or text in memory. A data type determines the amount of storage allocated to a number or letter, the method used to encode the number's value as a pattern of binary digits, and the operations available for manipulating the type. Most computers provide a choice of data types for representing numbers, each with specific advantages in the areas of precision, dynamic range, performance, and memory usage. To optimize performance, you can specify the data types of variables used in the MATLAB technical computing environment. Simulink builds on this capability by allowing you to specify the data types of Simulink signals and block parameters.

The ability to specify the data types of a model's signals and block parameters is particularly useful in real-time control applications. For example, it allows a Simulink model to specify the optimal data types to use to represent signals and block parameters in code generated from a model by automatic code-generation tools, such as the Simulink Coder product. By choosing the most appropriate data types for your model's signals and parameters, you can dramatically increase performance and decrease the size of the code generated from the model.

Simulink performs extensive checking before and during a simulation to ensure that your model is *typesafe*, that is, that code generated from the model will not overflow or underflow and thus produce incorrect results. Simulink models that use the default data type (`double`) are inherently typesafe. Thus, if you never plan to generate code from your model or use a nondefault data type in your models, you can skip the remainder of this section.

On the other hand, if you plan to generate code from your models and use nondefault data types, read the remainder of this section carefully, especially the section on data type rules (see “Data Typing Guidelines” on page 67-2). In that way, you can avoid introducing data type errors that prevent your model from running to completion or simulating at all.

Data Typing Guidelines

Observing the following rules can help you to create models that are typesafe and, therefore, execute without error:

- Signal data types generally do not affect parameter data types, and vice versa.

  A significant exception to this rule is the Constant block, whose output data type is determined by the data type of its parameter.
- If the output of a block is a function of an input and a parameter, and the input and parameter differ in type, Simulink converts the parameter to the input type before computing the output.
- In general, a block outputs the data type that appears at its inputs.

Significant exceptions include Constant blocks and Data Type Conversion blocks, whose output data types are determined by block parameters.

- Virtual blocks accept signals of any type on their inputs.

  Examples of virtual blocks include Mux and Demux blocks and unconditionally executed subsystems.
- The elements of a signal array connected to a port of a nonvirtual block must be of the same data type.
- The signals connected to the input data ports of a nonvirtual block cannot differ in type.
- Control ports (for example, Enable and Trigger ports) accept any data type.
- Solver blocks accept only `double` signals.
- Connecting a non-`double` signal to a block disables zero-crossing detection for that block.

Data Type Propagation

Whenever you start a simulation, enable display of port data types, or refresh the port data type display, Simulink performs a processing step called data type propagation. This step involves determining the types of signals whose type is not otherwise specified and checking the types of signals and input ports to ensure that they do not conflict. If type conflicts arise, an error dialog is displayed that specifies the signal and port whose data types conflict. The signal path that creates the type conflict is also highlighted.

Note You can insert typecasting (data type conversion) blocks in your model to resolve type conflicts. For more information, see Data Type Conversion.

See Also

`Simulink.AliasType` | `Simulink.NumericType`

Related Examples

- “Control Signal Data Types” on page 67-6
- “Control Block Parameter Data Types” on page 37-44
- “Validate a Floating-Point Embedded Model” on page 67-12
- “Specify Fixed-Point Data Types” on page 67-28
- “Data Types Supported by Simulink” on page 67-4
- “Simulink Strings” on page 67-40

Data Types Supported by Simulink

Simulink supports all built-in numeric MATLAB data types. The term *built-in data type* refers to data types defined by MATLAB itself as opposed to data types defined by MATLAB users. Unless otherwise specified, the term data type in the Simulink documentation refers to built-in data types.

The following table lists the built-in MATLAB data types supported by Simulink.

Name	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer
`int64`	Signed 64-bit integer
`uint64`	Unsigned 64-bit integer
`half`	Half-precision floating point (requires Fixed-Point Designer license)
`string`	Text

Besides these built-in types, Simulink defines a `boolean` (`true` or `false`) type. The values `1` and `0` represent `true` and `false` respectively. For this data type, Simulink represents real, nonzero numeric values (including `Inf`) as `true` (`1`).

Block Support for Data and Signal Types

All Simulink blocks accept signals of type `double` by default. Some blocks prefer `boolean` input and others support multiple data types on their inputs. For more information on the data types supported by a specific block for parameter and input and output values, see the reference page for that block. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

Several blocks support bus objects (`Simulink.Bus`) as data types. See “Data Types for Bus Signals” on page 67-39.

Many Simulink blocks also support fixed-point data types. For more information about fixed-point data, see “Specify Fixed-Point Data Types” on page 67-28. For more information on the data types supported by a specific block for parameter and input and output values, in the Simulink documentation see the Data Type Support section of the reference page for that block. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

To view a table that summarizes the data types supported by the blocks in the Simulink block libraries, execute the following command at the MATLAB command line:

```
showblockdatatypetable
```

See Also

`Simulink.AliasType` | `Simulink.NumericType`

Related Examples

- “Data Typing in Simulink”
- “Control Signal Data Types” on page 67-6
- “Specify Fixed-Point Data Types” on page 67-28
- “Define Simulink Enumerations” on page 68-6
- “Specify Data Types Using Data Type Assistant” on page 67-30
- “About Data Types in Simulink” on page 67-2
- “Simulink Strings” on page 67-40

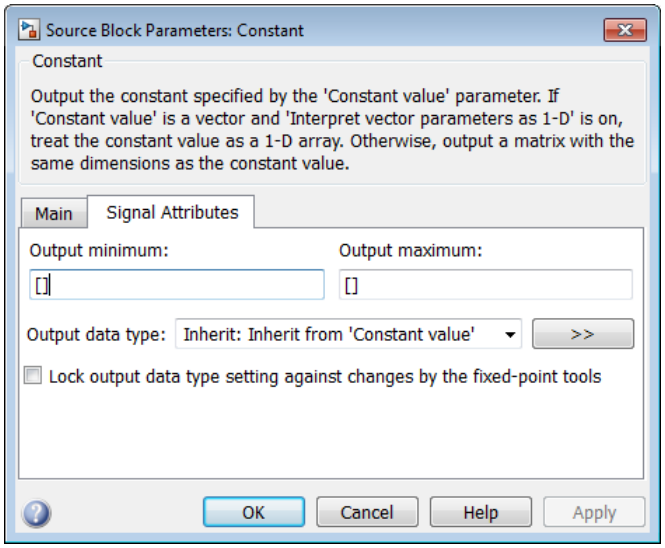
Control Signal Data Types

To control the data type of a signal in a Simulink model, you specify a data type for the corresponding block output.

You can also introduce a new signal of a specific data type into a model in any of the following ways:

- Load signal data of the desired type from the MATLAB workspace into your model via a root-level Inport block or a From Workspace block.
- Create a Constant block in your model and set its parameter to the desired type.
- Use a Data Type Conversion block to convert a signal to the desired data type.

Simulink blocks determine the data type of their outputs by default. Many blocks allow you to override the default type and explicitly specify an output data type, using a block parameter that is typically named **Output data type**. For example, the **Output data type** parameter appears on the **Signal Attributes** pane of the Constant block dialog box.



See the following topics for more information:

For Information About...	See...
Valid data type values that you can specify	"Entering Valid Data Type Values" on page 67-6
An assistant that helps you specify valid data type values	"Specify Data Types Using Data Type Assistant" on page 67-30
Specifying valid data type values for multiple blocks simultaneously	"Use the Model Data Editor for Batch Editing" on page 67-8

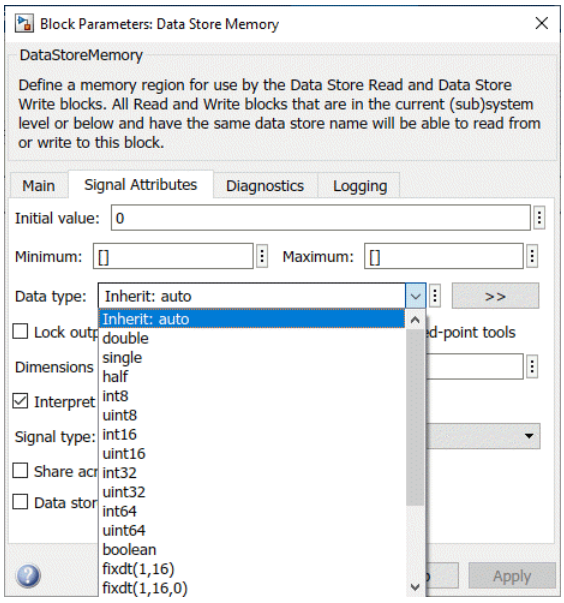
Entering Valid Data Type Values

In general, you can specify the output data type as any of the following:

- A rule that inherits a data type (see "Data Type Inheritance Rules" on page 67-7)

- The name of a built-in data type (see “Built-In Data Types” on page 67-8)
- An expression that evaluates to a data type (see “Data Type Expressions” on page 67-8)

Valid data type values vary among blocks. You can use the pull-down menu associated with a block data type parameter to view the data types that a particular block supports. For example, the **Data type** pull-down menu on the Data Store Memory block dialog box lists the data types that it supports, as shown here.



For more information about the data types that a specific block supports, see the documentation for the block in the Simulink documentation.

Data Type Inheritance Rules

Blocks can inherit data types from a variety of sources, including signals to which they are connected and particular block parameters. You can specify the value of a data type parameter as a rule that determines how the output signal inherits its data type. To view the inheritance rules that a block supports, use the data type pull-down menu on the block dialog box. The following table lists typical rules that you can select.

Inheritance Rule	Description
`Inherit: Inherit via back propagation`	Simulink automatically determines the output data type of the block during data type propagation (see “Data Type Propagation” on page 67-3). In this case, the block uses the data type of a downstream block or signal object.
`Inherit: Same as input`	The block uses the data type of its sole input signal for its output signal.
`Inherit: Same as first input`	The block uses the data type of its first input signal for its output signal.
`Inherit: Same as second input`	The block uses the data type of its second input signal for its output signal.

Inheritance Rule	Description
`Inherit: Inherit via internal rule`	The block uses an internal rule to determine its output data type. The internal rule chooses a data type that optimizes numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. It is not always possible for the software to optimize efficiency and numerical accuracy at the same time.

When you apply inherited data types to a signal, Simulink determines the specific data type of the signal only after you update the block diagram.

- To display this specific data type on the block diagram, see “Port Data Types” on page 75-46.
- To inspect this specific data type for multiple signals in a searchable, sortable table, use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). The right side of the **Data Type** column shows the specific data type for each signal. For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Built-In Data Types

You can specify the value of a data type parameter as the name of a built-in data type, for example, `single` or `boolean`. To view the built-in data types that a block supports, use the data type pull-down menu on the block dialog box. See “Data Types Supported by Simulink” on page 67-4 for a list of all built-in data types that are supported.

Data Type Expressions

You can specify the value of a data type parameter as an expression that evaluates to a numeric data type object. Simply enter the expression in the data type field on the block dialog box. In general, enter one of the following expressions:

- **`fixdt` Command**

  Specify the value of a data type parameter as a command that invokes the `fixdt` function. This function allows you to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type. See the documentation for the `fixdt` function for more information.

- **Data Type Object Name**

  Specify the value of a data type parameter as the name of a data object that represents a data type. Simulink data objects that you instantiate from classes, such as `Simulink.NumericType` and `Simulink.AliasType`, simplify the task of making model-wide changes to output data types and allow you to use custom aliases for data types. See “Data Objects” on page 67-58 for more information about Simulink data objects.

Use the Model Data Editor for Batch Editing

Using the Model Data Editor (see “Configure Data Properties by Using the Model Data Editor” on page 67-131), you can assign the same data type to multiple signals simultaneously. You can use this technique to design the interface of your model by configuring data types and other attributes of multiple Inport and Outport blocks at once (see “Configure Data Interface for Component” on page 22-18). You can also finely control the data types of arbitrary signals in your block algorithm.

For example, the `slexAircraftExample` model that comes with the Simulink product contains numerous Gain blocks. Suppose you want to specify the output data type of the three Gain blocks at the root level of the model as `single`. You can achieve this task as follows:

1. In the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), inspect the **Signals** tab.
2. Next to the **Filter contents** box, toggle the **Filter using selection** button.
3. At the top level of the model, select the signal lines that represent the outputs of the three Gain blocks (labeled `Zw`, `Mw`, and `Mq`). The Model Data Editor shows three rows that correspond to the three signals.
4. In the Model Data Editor, select all three signals (rows). For example, you can press **Ctrl+A** or hold **Shift** while clicking the top and bottom rows in the **Source** column.
5. For any of the three signals, click the cell in the **Data Type** column. From the drop-down list, select `single`. The Model Data Editor applies this selection to all of the selected rows.

To convert a model to a strict single precision design, see “Validate a Floating-Point Embedded Model” on page 67-12.

Share a Data Type Between Separate Algorithms, Data Paths, Models, and Bus Elements

In some cases, you cannot rely on data type inheritance (see “Data Type Inheritance Rules” on page 67-7) to establish equivalence between the data types of different data items (such as signal lines in parallel data paths or bus elements in a `Simulink.Bus` object). Instead, you can create a `Simulink.NumericType` or `Simulink.AliasType` object in a workspace or data dictionary.

Create a `Simulink.NumericType` object if you do not want to rename the shared data type by creating an alias. Set the `IsAlias` property to `false` (the default).

This example shows how to use a `Simulink.NumericType` object to share an output data type between two lookup table blocks in the same model.

1. Open the example model `sldemo_fuelsys`.

   ```
   sldemo_fuelsys
   ```

   The model creates `Simulink.NumericType` objects in the base workspace. One of the objects is named `s16En15`.
2. At the command prompt, inspect the properties of `s16En15`.

   ```
   s16En15
   ```

   ```
   s16En15 =

     NumericType with properties:

       DataTypeMode: 'Single'
            IsAlias: 0
          DataScope: 'Auto'
         HeaderFile: ''
        Description: ''
   ```

   This object represents the built-in Simulink data type `single`.

3. In the model, navigate into the `fuel_rate_control/airflow_calc` subsystem.
4. On the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, inspect the **Signals** tab.
5. In the model, click the output signal of the Pumping Constant block. The Model Data Editor **Data Type** column shows that the signal data type is set to `s16En15`.
6. Click the output signal of the Ramp Rate Ki block. The output data type of this block is also set to `s16En15`.
7. Update the block diagram and, if necessary, expand the width of the **Data Type** column. The right side of the column shows that the two lookup table blocks use the data type `single`.
8. At the command prompt, configure `s16En15` to represent the data type `double`.

   ```
   s16En15.DataTypeMode = 'Double';
   ```

9. Update the block diagram.

   The output signals of the two lookup table blocks now use the data type `double`. Due to data type inheritance, other signals, such as `e0` and `e1`, acquire the same data type.

Alternatively, to establish data type equivalence between algorithms or data paths in the same model, you can use blocks such as Data Type Propagation and Data Type Conversion Inherited. When you use these blocks, you do not need to create and permanently store a data type object. However, you cannot use the blocks to share a data type between signals in different models unless the models are in the same model reference hierarchy.

Reuse Custom C Data Types for Signal Data

In a model, you can create signals that conform to custom C data types, such as structures, that your existing C code defines. Use these signals to:

- Replace existing C code with a Simulink model.
- Integrate C code for simulation in Simulink (for example, by using the Legacy Code Tool).
- Prepare to generate code (Simulink Coder) that you can integrate with existing code.

Use these techniques to match your custom data types:

- For a structure type, create a `Simulink.Bus` object. Use the object as the data type for bus signals. See “Data Types for Bus Signals” on page 67-39.
- For an enumeration, create an enumeration class and use it as the data type for signals. See “Use Enumerated Data in Simulink Models” on page 68-6.
- To match a `typedef` statement that represents an alias of a primitive, numeric data type, use a `Simulink.AliasType` object as the data type for signals. See `Simulink.AliasType`.

To create these classes and objects, you can use the function `Simulink.importExternalCTypes`.

If a MATLAB Function block or Stateflow chart in your model uses an imported enumeration or structure type, configure the model configuration parameters to include (`#include`) the type definition from your external header file. See “Control Imported Bus and Enumeration Type Definitions” on page 44-124 (for a MATLAB Function block) and “Access Custom Code Variables and Functions in Stateflow Charts” (Stateflow) and “Integrate Custom Structures in Stateflow Charts” (Stateflow) (for a chart).

Determine Data Type of Signal That Uses Inherited Setting

When a signal uses an inherited data type setting such as `Inherit: Inherit via internal rule` (the default setting for most blocks), to determine the meaningful data type that the signal uses for simulation, update the block diagram and then use one or both of these techniques:

- In the Simulink Editor, on the **Debug** tab, select **Information Overlays** > **Port Data Type**. The data types appear on the block diagram next to each signal. For more information, see “Port Data Types” on page 75-46.
- Inspect the right side of the **Data Type** column in the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Using these techniques to inspect data types helps you to:

- Design the data type strategy for a model on a high level.
- Debug numerical issues due to quantization and overflows.
- Make a model more easily understood when sharing it.

For more information, see “Port Data Types” on page 75-46.

Data Types Remain double Despite Changing Settings

If many of the data items (signals, parameters, and states) in your model continue to use the data type `double` after you configure block parameters such as **Output data type**, confirm that the model is not configured to override data types. See “Control Data Type Override” on page 67-26.

See Also

`Simulink.AliasType` | `Simulink.Bus` | `Simulink.NumericType`

Related Examples

- “Data Typing Filter”
- “Validate a Floating-Point Embedded Model” on page 67-12
- “Specify Fixed-Point Data Types” on page 67-28
- “Specify Data Types Using Data Type Assistant” on page 67-30
- “About Data Types in Simulink” on page 67-2
- “Data Types Supported by Simulink” on page 67-4
- “Data Types for Bus Signals” on page 67-39

Validate a Floating-Point Embedded Model

You can use data type override mode to temporarily switch the data types in your model. This capability allows you to maintain one model but simulate your model using multiple data types, and validate the numerical behavior for each type. For example, if you implement an algorithm using double-precision data types and want to check whether the algorithm is also suitable for single-precision use, you can apply a data type override to floating-point data types to replace all doubles with singles without permanently affecting any other data types in your model.

Apply a Data Type Override to Floating-Point Data Types

To apply data type override, you must specify the data type that you want to apply and the data type that you want to replace.

You can set data type override using the following method. This example changes all floating-point data types to single.

For example:

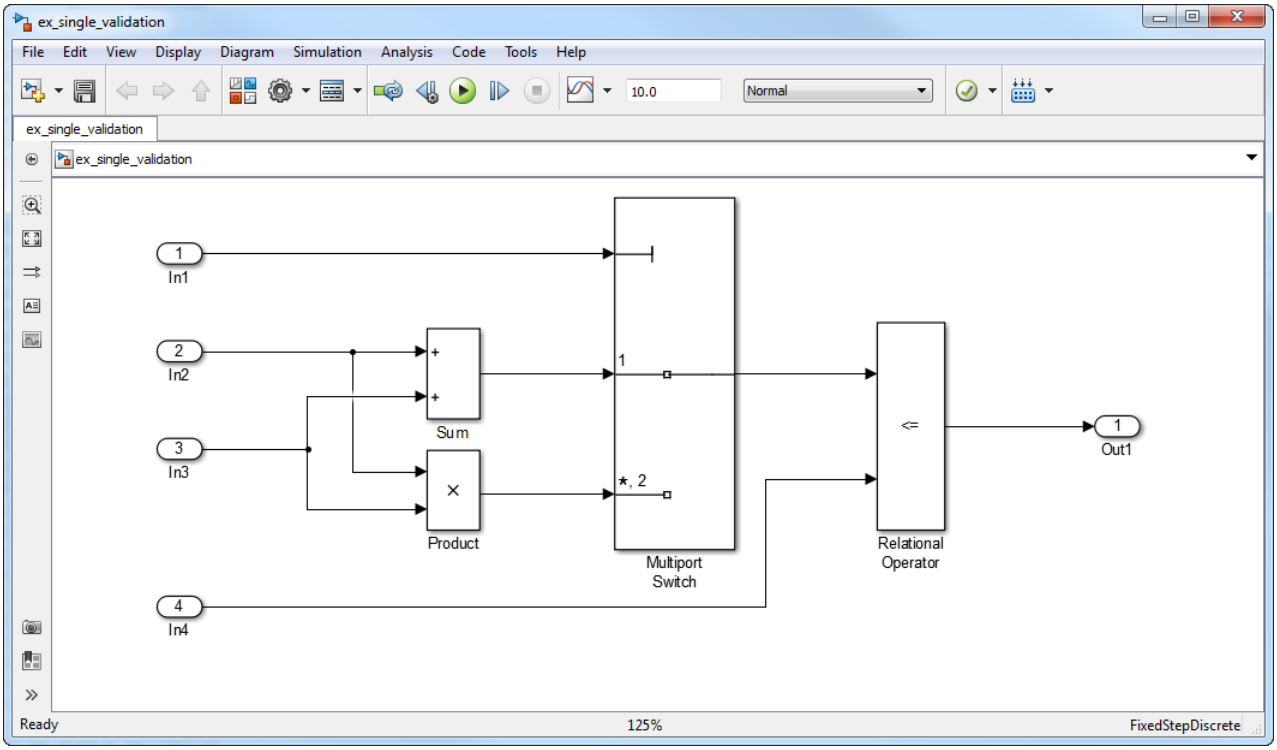
```
set_param(gcs, 'DataTypeOverride', 'Single',...
 'DataTypeOverrideAppliesTo','Floating-point');
```

For more information on data type override settings, see “Control Data Type Override” on page 67-26.

Validate a Single-Precision Model

This example uses the `ex_single_validation` model to show how you can use data type override. It proves that an algorithm, which implements double-precision data types, is also suitable for single-precision embedded use.

About the Model



- The inputs `In2` and `In3` are double-precision inputs to the Sum and Product blocks.
- The outputs of the Sum and Product blocks are data inputs to the Multiport Switch block.
- The input `In1` is the control input to the Multiport Switch block. The value of this control input determines which of its other inputs, the sum of `In2` and `In3` or the product of `In2` and `In3`, passes to the output port. Because `In1` is a control input, its data type is `int8`.
- The Relational Operator block compares the output of the Multiport Switch block to `In4`, and outputs a Boolean signal.

Run the Example

Open the Model

1 Open the `ex_single_validation` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_single_validation
```

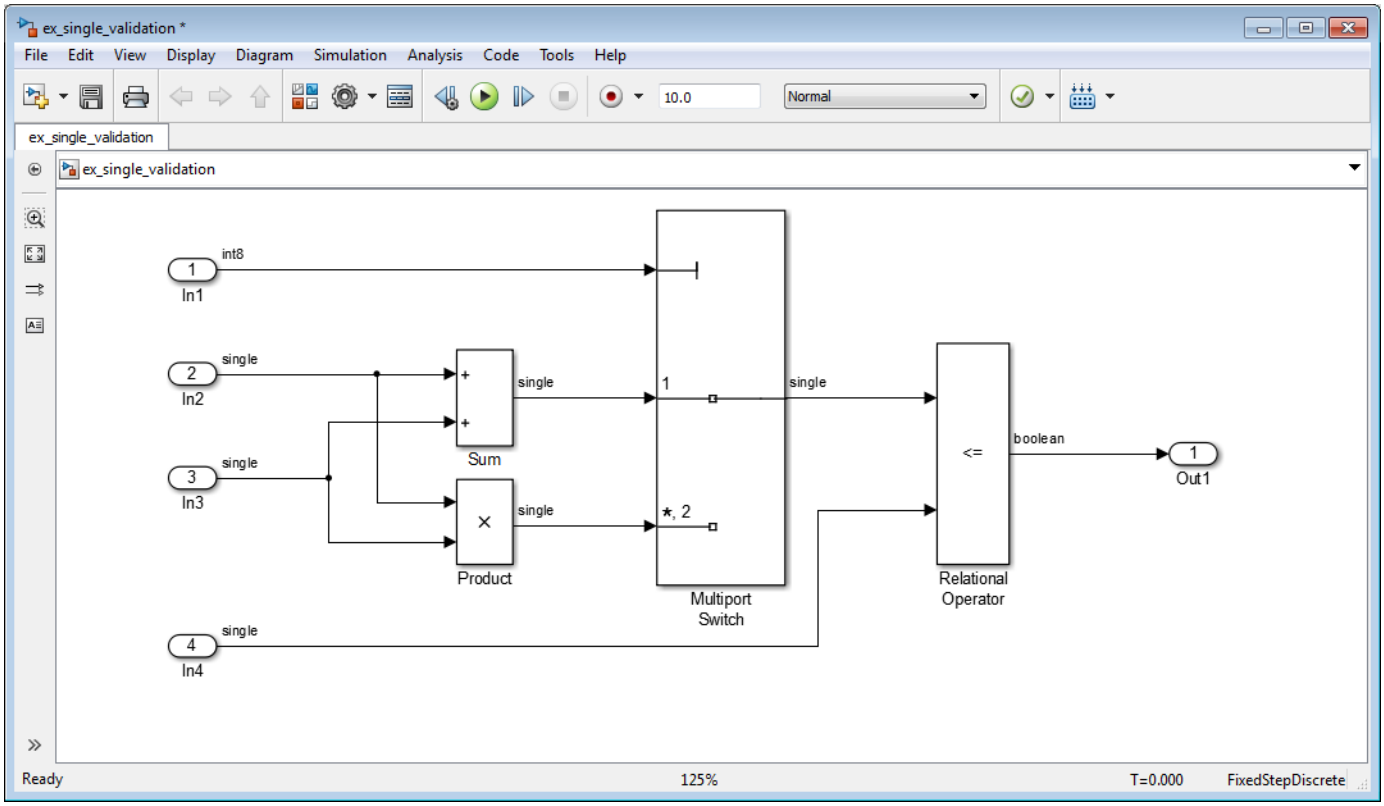
Override Floating-Point Data Types With Singles

1 At the command line, override the floating-point data types in the model with singles

```
set_param(gcs, 'DataTypeOverride', 'Single',...
 'DataTypeOverrideAppliesTo','Floating-point');
```

2 In the model, on the **Modeling** tab, click **Update Model**.

The data type override replaces all the floating-point (`double`) data types in the model with `single` data types, but does not affect the integer or Boolean data types.



Run Model Advisor Check

1. From the model, on the **Modeling** tab, click **Model Advisor**.
2. In the System Selector dialog box, click **OK**.

   The Model Advisor opens.
3. In the Model Advisor, expand the **By Task** node and, under **Modeling Single-Precision Systems**, select the **Identify questionable operations for strict single-precision design** check.
4. In the right pane, click **Run This Check**.

   The check passes indicating that this algorithm is suitable for single-precision use. To ensure that no double-precision data types remain in the generated code, use the Single-Precision Converter before generating code for single-precision embedded use. For more information, see “Getting Started with Single Precision Converter” (Fixed-Point Designer).

Blocks That Support Single Precision

To identify Simulink blocks that support single precision, at the command prompt, enter `showblockdatatypetable`. In a model, to find blocks that do not support single precision, use the Model Advisor check “Identify questionable operations for strict single-precision design”.

See Also

`Simulink.AliasType` | `Simulink.NumericType`

Related Examples

- “Single-Precision Design for Simulink Models” (Fixed-Point Designer)
- “Specify Single-Precision Data Type for Embedded Application” (Simulink Coder)
- “Control Signal Data Types” on page 67-6
- “Default for underspecified data type”
- “Identify questionable operations for strict single-precision design”
- “Inf or NaN block output”
- “About Data Types in Simulink” on page 67-2

Fixed-Point Numbers

In this section...

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type. There are several distinct differences between fixed-point data types and the built-in integer types in MATLAB®. The most notable difference, is that the built-in integer data types can only represent whole numbers, while the fixed-point data types also contain information on the position of the binary point, or the scaling of the number.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted. With Fixed-Point Designer, fixed-point data types can be integers, fractionals, or generalized fixed-point numbers. The main difference between these data types is their default binary point. For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:

where

- b_i is the i^{th} binary digit.
- *wl* is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Binary Point Interpretation

The binary point is the means by which fixed-point numbers are scaled. It is usually the software that determines the binary point. When performing basic math functions such as addition or subtraction, the hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a scale factor. They are performing signed or unsigned fixed-point binary algebra as if the binary point is to the right of b_0.

Fixed-Point Designer supports the general binary point scaling $V=Q*2\hat{}E$. `V` is the real-world value, `Q` is the stored integer value, and `E` is equal to `-FractionLength`. In other words, `RealWorldValue = StoredInteger * 2 ^ -FractionLength`.

`FractionLength` defines the scaling of the stored integer value. The word length limits the values that the stored integer can take, but it does not limit the values `FractionLength` can take. The

software does not restrict the value of exponent `E` based on the word length of the stored integer `Q`. Because `E` is equal to `-FractionLength`, restricting the binary point to being contiguous with the fraction is unnecessary; the fraction length can be negative or greater than the word length.

For example, a word consisting of three unsigned bits is usually represented in scientific notation in one of the following ways.

$$bbb. = bbb. \times 2^{0}$$
$$bb.b = bbb. \times 2^{-1}$$
$$b.bb = bbb. \times 2^{-2}$$
$$.bbb = bbb. \times 2^{-3}$$

If the exponent were greater than 0 or less than -3, then the representation would involve lots of zeros.

$$bbb00000. = bbb. \times 2^{5}$$
$$bbb00. = bbb. \times 2^{2}$$
$$.00bbb = bbb. \times 2^{-5}$$
$$.00000bbb = bbb. \times 2^{-8}$$

These extra zeros never change to ones, however, so they don't show up in the hardware. Furthermore, unlike floating-point exponents, a fixed-point exponent never shows up in the hardware, so fixed-point exponents are not limited by a finite number of bits.

Consider a signed value with a word length of 8, a fraction length of 10, and a stored integer value of 5 (binary value `00000101`). The real-word value is calculated using the formula `RealWorldValue = StoredInteger * 2 ^ -FractionLength`. In this case, `RealWorldValue = 5 * 2 ^ -10 = 0.0048828125`. Because the fraction length is 2 bits longer than the word length, the binary value of the stored integer is `x.xx00000101` , where `x` is a placeholder for implicit zeros. `0.0000000101` (binary) is equivalent to `0.0048828125` (decimal). For an example using a `fi` object, see “Fraction Length Greater Than Word Length” (Fixed-Point Designer).

Signed Fixed-Point Numbers

Computer hardware typically represents the negation of a binary fixed-point number in three different ways: sign/magnitude, one's complement, and two's complement. Two's complement is the preferred representation of signed fixed-point numbers and is the only representation used by Fixed-Point Designer.

Negation using two's complement consists of a bit inversion (translation into one's complement) followed by the addition of a one. For example, the two's complement of 000101 is 111011.

Whether a fixed-point value is signed or unsigned is usually not encoded explicitly within the binary word; that is, there is no sign bit. Instead, the sign information is implicitly defined within the computer architecture.

See Also

More About

- “Scaling, Precision, and Range” on page 67-20

Benefits of Using Fixed-Point Hardware

Digital hardware is becoming the primary means by which control systems and signal processing filters are implemented. Digital hardware can be classified as either off-the-shelf hardware (for example, microcontrollers, microprocessors, general-purpose processors, and digital signal processors) or custom hardware. Within these two types of hardware, there are many architecture designs. These designs range from systems with a single instruction, single data stream processing unit to systems with multiple instruction, multiple data stream processing units.

Within digital hardware, numbers are represented as either fixed-point or floating-point data types. For both of these data types, word sizes are fixed at a set number of bits. However, the dynamic range of fixed-point values is much less than floating-point values with equivalent word sizes. Therefore, in order to avoid overflow or unreasonable quantization errors, fixed-point values must be scaled. Since floating-point processors can greatly simplify the real-time implementation of a control law or digital filter, and floating-point numbers can effectively approximate real-world numbers, then why use a microcontroller or processor with fixed-point hardware support?

- Size and Power Consumption — The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. For example, consider a portable telephone where one of the product design goals is to make it as portable (small and light) as possible. If one of today's high-end floating-point, general-purpose processors is used, a large heat sink and battery would also be needed, resulting in a costly, large, and heavy portable phone.
- Memory Usage and Speed — In general fixed-point calculations require less memory and less processor time to perform.
- Cost — Fixed-point hardware is more cost effective where price/cost is an important consideration. When digital hardware is used in a product, especially mass-produced products, fixed-point hardware costs much less than floating-point hardware and can result in significant savings.

After making the decision to use fixed-point hardware, the next step is to choose a method for implementing the dynamic system (for example, control system or digital filter). Floating-point software emulation libraries are generally ruled out because of timing or memory size constraints. Therefore, you are left with fixed-point math where binary integer values are scaled.

See Also

More About

- “Fixed-Point Numbers” on page 67-16

Scaling, Precision, and Range

In this section...

The dynamic range of fixed-point values is less than floating-point values with equivalent word sizes. To avoid overflow and minimize quantization errors, fixed-point numbers must be scaled.

Scaling

With Fixed-Point Designer, you can select a fixed-point data type whose scaling is defined by its binary point, or you can select an arbitrary linear scaling that suits your needs.

Slope and Bias Scaling

You can represent a fixed-point number by a general slope and bias encoding scheme. The real world value of a slope bias scaled number can be represented by:

$$\text{real-world value} = (\text{slope} \times \text{integer}) + \text{bias}$$

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in slope bias representation that has a bias equal to zero and a slope adjustment factor equal to one. This is referred to as binary point-only scaling or power-of-two scaling.

Binary-Point-Only Scaling

Binary-point-only or power-of-two scaling involves moving the binary point within the fixed-point word. The advantage of this scaling mode is to minimize the number of processor arithmetic operations. The real world value of a binary-point only scaled number can be represented by:

$$\text{real world value} = 2^{-\text{fraction length}} \times \text{integer}$$

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision.

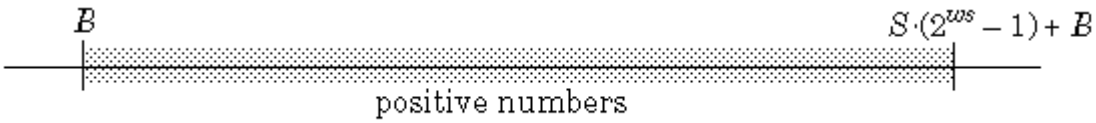
Rounding Methods

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and

scaling, a rounding method is used to cast the value to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding method itself. For more information on the rounding methods available with Fixed-Point Designer, see “Rounding Methods” (Fixed-Point Designer)

Range

Range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for an unsigned two’s complement fixed-point number of word length *ws*, scaling *S*, and bias *B* is illustrated below:



The following figure illustrates the range of representable numbers for a two’s complement signed fixed-point number:

For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two’s complement, negative numbers must be represented as well as zero, so the maximum value is 2^{wl-1}-1. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for —2^{wl-1}, but not for 2^{wl-1}.

See Also

More About

- “Fixed-Point Numbers” on page 67-16
- “Benefits of Using Fixed-Point Hardware” on page 67-19

Fixed-Point Data in MATLAB and Simulink

In this section...

Fixed-Point Data in Simulink

You can use the `fixdt` function in Simulink to specify a fixed-point data type. The `fixdt` function creates a `Simulink.NumericType` object.

Fixed-Point Data Type and Scaling Notation

Simulink data type names must be valid MATLAB identifiers with less than 128 characters. The data type name provides information about container type, number encoding, and scaling.

The following table provides a key for various symbols that appear in Simulink products to indicate the data type and scaling of a fixed-point value.

Symbol	Description	Example
Container Type		
`ufix`	Unsigned fixed-point data type	`ufix8` is an 8-bit unsigned fixed-point data type
`sfix`	Signed fixed-point data type	`sfix128` is a 128-bit signed fixed-point data type
`fltu`	Scaled double override of an unsigned fixed-point data type (`ufix`)	`fltu32` is a scaled doubles override of `ufix32`
`flts`	Scaled double override of a signed fixed-point data type (`sfix`)	`flts64` is a scaled doubles override of `sfix64`
Number Encoding		
`e`	10^	`125e8` equals `125*(10^(8))`
`n`	Negative	`n31` equals `-31`
`p`	Decimal point	`1p5` equals `1.5` `p2` equals `0.2`
Scaling Encoding		
`S`	Slope	`ufix16_S5_B7` is a 16-bit unsigned fixed-point data type with `Slope` of `5` and `Bias` of `7`
`B`	Bias	`ufix16_S5_B7` is a 16-bit unsigned fixed-point data type with `Slope` of `5` and `Bias` of `7`

Symbol	Description	Example
E	Fixed exponent (2^) A negative fixed exponent describes the fraction length	`sfix32_En31` is a 32-bit signed fixed-point data type with a fraction length of `31`
F	Slope adjustment factor	`ufix16_F1p5_En50` is a 16-bit unsigned fixed-point data type with a `SlopeAdjustmentFactor` of `1.5` and a `FixedExponent` of `-50`
C,c,D, or d	Compressed encoding for Bias **Note** If you pass this character vector to the `slDataTypeAndScale` function, it returns a valid `fixdt` data type.	No example available. For backwards compatibility only. To identify and replace calls to `slDataTypeAndScale`, use the "Check for calls to slDataTypeAndScale" Model Advisor check.
T or t	Compressed encoding for Slope **Note** If you pass this character vector to the `slDataTypeAndScale`, it returns a valid `fixdt` data type.	No example available. For backwards compatibility only. To identify and replace calls to `slDataTypeAndScale`, use the "Check for calls to slDataTypeAndScale" Model Advisor check.

Fixed-Point Data in MATLAB

To assign a fixed-point data type to a number or variable in MATLAB, use the `fi` constructor. The resulting fixed-point value is called a `fi` object. For example, the following creates `fi` objects *a* and *b* with attributes shown in the display, all of which we can specify when the variables are constructed. Note that when the `FractionLength` property is not specified, it is set automatically to "best precision" for the given word length, keeping the most-significant bits of the value. When the `WordLength` property is not specified it defaults to 16 bits.

```
a = fi(pi)

a =

              3.1416015625

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

b = fi(0.1)

b =

        0.0999984741210938

          DataTypeMode: Fixed-point: binary point scaling
```

```
    Signedness: Signed
    WordLength: 16
FractionLength: 18
```

Read Fixed-Point Data from the Workspace

Use the From Workspace block to read fixed-point data from the MATLAB workspace into a Simulink model. To do this, the data must be in structure format with a `fi` object in the `values` field. In array format, the From Workspace block only accepts real, double-precision data.

Write Fixed-Point Data to the Workspace

You can write fixed-point output from a model to the MATLAB workspace via the To Workspace block in either array or structure format. Fixed-point data written by a To Workspace block to the workspace in structure format can be read back into a Simulink model in structure format by a From Workspace block.

Scaled Doubles

Scaled doubles are a hybrid between floating-point and fixed-point numbers. Fixed-Point Designer stores them as doubles with the scaling, sign, and word length information retained. For example, the storage container for a fixed-point data type `sfix16_En14` is `int16`. The storage container of the equivalent scaled doubles data type, `flts16_En14` is floating-point `double`. Fixed-Point Designer applies the scaling information to the stored floating-point double to obtain the real-world value. Storing the value in a double almost always eliminates overflow and precision issues.

See Also

Functions

`Simulink.NumericType` | `fi` | `fimath` | `fixdt`

Share Fixed-Point Models

You can edit a model containing fixed-point blocks without having Fixed-Point Designer. However, you must have Fixed-Point Designer to:

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types
- Run a model containing fixed-point data types
- Generate code from a model containing fixed-point data types
- Log the minimum and maximum values produced by a simulation
- Automatically scale the output of a model

If you do not have Fixed-Point Designer, you can work with a model containing Simulink blocks with fixed-point settings as follows:

1.
```
set_param(gcs, 'DataTypeOverride', 'Double',...
 'DataTypeOverrideAppliesTo','AllNumericTypes',...
 'MinMaxOverflowLogging','ForceOff')
```

2. If you use `fi` objects or embedded numeric data types in your model, set the `fipref` `DataTypeOverride` property to `TrueDoubles` or `TrueSingles` (to be consistent with the model-wide data type override setting) and the `DataTypeOverrideAppliesTo` property to `All numeric types`.

   For example, at the MATLAB command line, enter:

   ```
   p = fipref('DataTypeOverride', 'TrueDoubles', ...
          'DataTypeOverrideAppliesTo', 'AllNumericTypes');
   ```

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.

See Also

More About

- “Control Fixed-Point Instrumentation and Data Type Override” on page 67-26
- “Fixed-Point Data in MATLAB and Simulink” on page 67-22

Control Fixed-Point Instrumentation and Data Type Override

In this section...

The conversion of a model from floating point to fixed point requires configuring fixed-point instrumentation and data type overrides. However, leaving these settings on after the conversion can lead to unexpected results. If you do not have Fixed-Point Designer, you can work with a model containing Simulink blocks with fixed-point settings by turning off fixed-point instrumentation and setting data type override to scaled doubles.

Control Instrumentation Settings

The fixed-point instrumentation mode controls which objects log minimum, maximum, and overflow data during simulation. Instrumentation is required to collect simulation ranges using the Fixed-Point Tool. These ranges are used to propose data types for the model. When you are not actively converting your model to fixed point, disable the fixed-point instrumentation to restore the maximum simulation speed to your model.

To enable instrumentation outside of the Fixed-Point Tool, at the command line set the `MinMaxOverflowLogging` parameter to `MinMaxAndOverflow` or `OverflowOnly`.

```
set_param('MyModel', 'MinMaxOverflowLogging', 'MinMaxAndOverflow')
```

Instrumentation requires a Fixed-Point Designer license. To disable instrumentation on a model, set the parameter to `ForceOff` or `UseLocalSettings`.

```
set_param('MyModel', 'MinMaxOverflowLogging', 'UseLocalSettings')
```

Control Data Type Override

Use data type override to simulate your model using double, single, or scaled double data types. If you do not have Fixed-Point Designer software, you can still configure data type override settings to simulate a model that specifies fixed-point data types. Using this setting, the software temporarily overrides data types with floating-point data types during simulation.

```
set_param('MyModel', 'DataTypeOverride', 'Double')
```

To observe the true behavior of your model, set the data type override parameter to `UseLocalSettings` or `Off`.

```
set_param('MyModel', 'DataTypeOverride', 'Off')
```

Instrumentation Settings and Data Type Override for a Model Reference Hierarchy

When you simulate a model that contains referenced models, the data type override and fixed-point instrumentation settings for the top-level model do not control the settings for the referenced models.

You must specify these settings separately for the referenced model. If the settings are inconsistent, for example, if you set the top-level model data type override setting to double and the referenced model to use local settings and the referenced model uses fixed-point data types, data type propagation issues might occur.

When you change the fixed-point instrumentation and data type override settings for any instance of a referenced model, the settings change on all instances of the model and on the referenced model itself.

See Also

More About

- “Share Fixed-Point Models” on page 67-25

Specify Fixed-Point Data Types

Simulink allows you to create models that use fixed-point numbers to represent signals and parameter values. Use of fixed-point data can reduce the memory requirements and increase the speed of code generated from a model.

To execute a model that uses fixed-point numbers, you must have the Fixed-Point Designer product installed on your system. Specifically, you must have the product to:

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types
- Run a model containing fixed-point data types
- Generate code from a model containing fixed-point data types
- Log the minimum and maximum values produced by a simulation
- Automatically scale the output of a model using the autoscaling tool

If the Fixed-Point Designer product is not installed on your system, you can execute a fixed-point model as a floating-point model by enabling automatic conversion of fixed-point data to floating-point data during simulation. See “Overriding Fixed-Point Specifications” on page 67-28 for details.

If you do not have the Fixed-Point Designer product installed and do not enable automatic conversion of fixed-point to floating-point data, an error occurs if you try to execute a fixed-point model.

Note You do not need the Fixed-Point Designer product to edit a model containing fixed-point blocks, or to use the Data Type Assistant to specify fixed-point data types, as described in “Specifying a Fixed-Point Data Type” on page 67-32.

Fixed-point data types that resolve to a base integer type do not require a Fixed-Point Designer license. For example, a block or signal that specifies a data type of `fixdt(1,8,0)`, which is equivalent to the `int8` built-in type will not check out a Fixed-Point Designer license.

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with Fixed-Point Designer. However, even if you do not have Fixed-Point Designer, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, Simulink temporarily overrides fixed-point data types with floating-point data types when simulating the model.

Note If you use fi (Fixed-Point Designer) 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 (Fixed-Point Designer) to prevent the checkout of a Fixed-Point Designer license.

To simulate a model without using Fixed-Point Designer, enter the following at the command line.

```
set_param(gcs, 'DataTypeOverride', 'Double', ...
'DataTypeOverrideAppliesTo', 'AllNumericTypes')
```

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

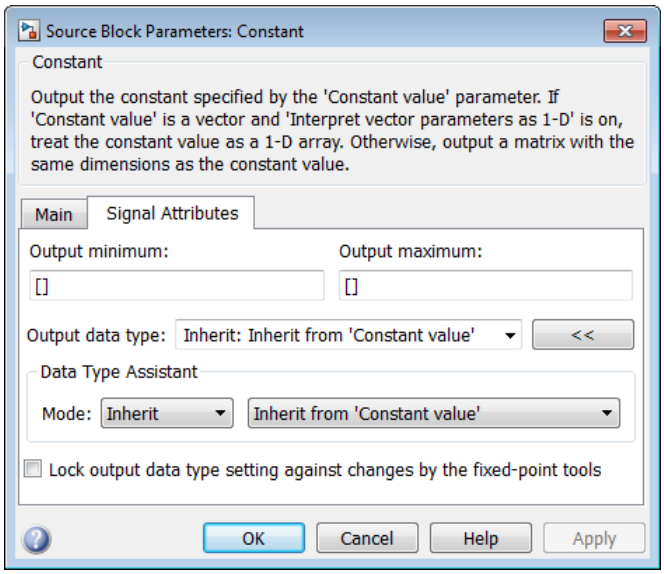
`Simulink.NumericType` | `fixdt`

Related Examples

- “Control Signal Data Types” on page 67-6
- “Specify Data Types Using Data Type Assistant” on page 67-30
- “About Data Types in Simulink” on page 67-2
- “Data Types Supported by Simulink” on page 67-4

Specify Data Types Using Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool that simplifies the task of specifying data types for blocks and data objects. The assistant appears on block and object dialog boxes, adjacent to parameters that provide data type control, such as the **Output data type** parameter. For example, it appears on the **Signal Attributes** pane of the Constant block dialog box shown here.



You can selectively show or hide the **Data Type Assistant** by clicking the applicable button:

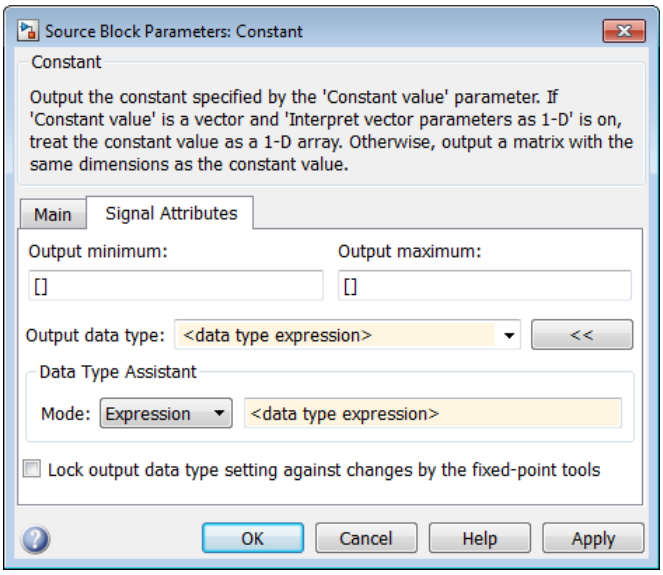
- Click the **Show data type assistant** button  to display the assistant.
- Click the **Hide data type assistant** button  to hide a visible assistant.

Use the **Data Type Assistant** to specify a data type as follows:

1. In the **Mode** field, select the category of data type that you want to specify. In general, the options include the following:

Mode	Description
`Inherit`	Inheritance rules for data types
`Built in`	Built-in data types
`Fixed point`	Fixed-point data types
`Enumerated`	Enumerated data types
`Bus object`	Bus object data types
`Expression`	Expressions that evaluate to data types

   The assistant changes dynamically to display different options that correspond to the selected mode. For example, setting **Mode** to `Expression` causes the Constant block dialog box to appear as follows.

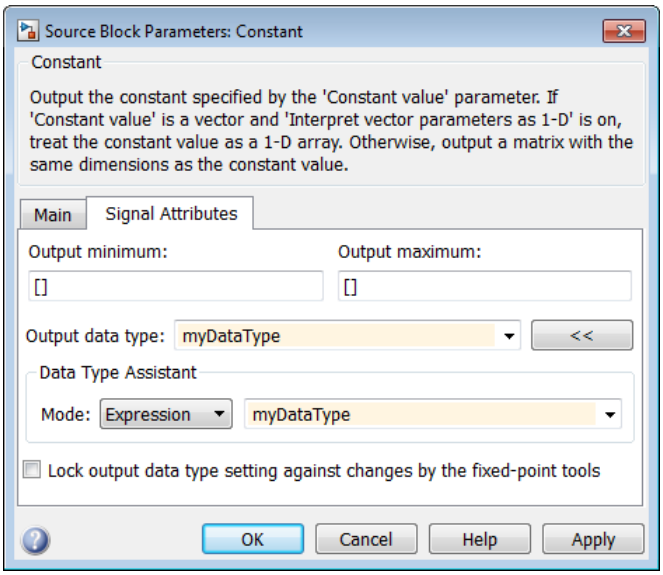


2 In the field that is to the right of the **Mode** field, select or enter a data type.

For example, suppose that you designate the variable `myDataType` as an alias for a `single` data type. You create an instance of the `Simulink.AliasType` class and set its `BaseType` property by entering the following commands:

```
myDataType = Simulink.AliasType
myDataType.BaseType = 'single'
```

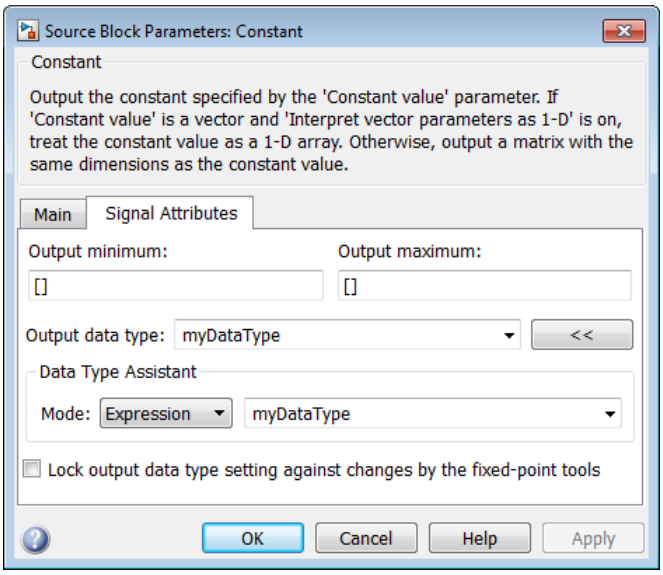
You can use this data type object to specify the output data type of a Constant block. Enter the data type alias name, `myDataType`, as the value of the expression in the assistant.



3 Click the **OK** or **Apply** button to apply your changes.

The assistant uses the data type that you specified to populate the associated data type parameter in the block or object dialog box. In the following example, the **Output data type**

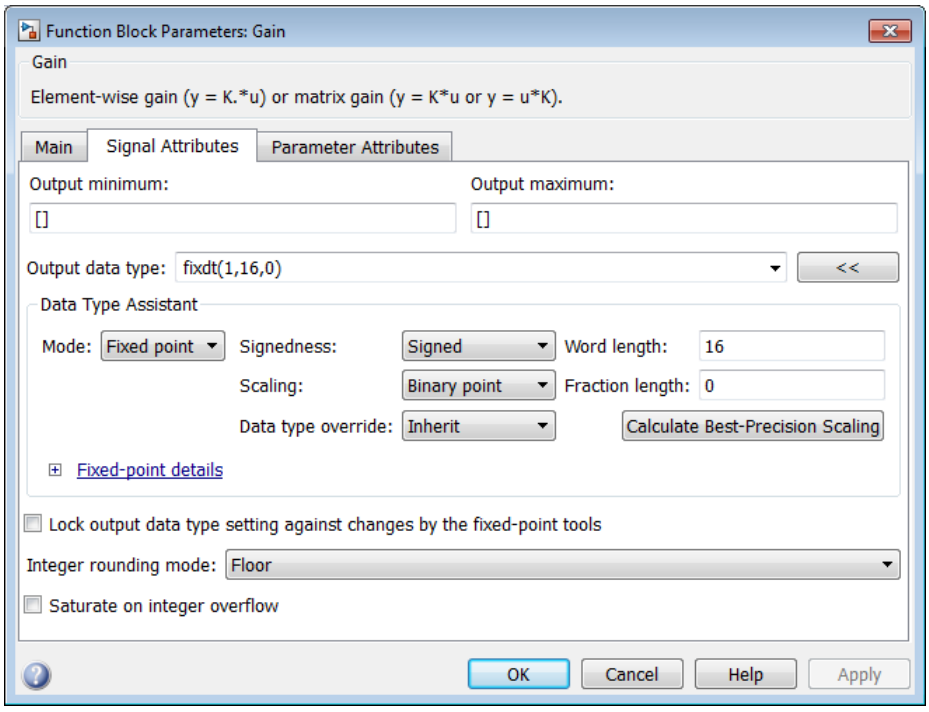
parameter of the Constant block specifies the same expression that you entered using the assistant.



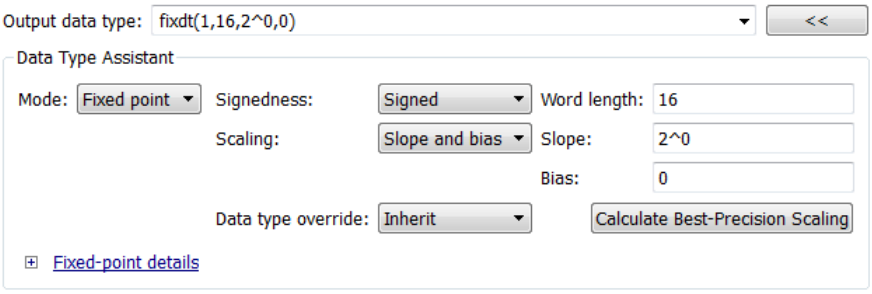
For more information about the data types that you can specify using the **Data Type Assistant**, see “Entering Valid Data Type Values” on page 67-6. For details about specifying fixed-point data types, see “Specify Fixed-Point Data Types with the Data Type Assistant” (Fixed-Point Designer).

Specifying a Fixed-Point Data Type

When the Data Type Assistant **Mode** is `Fixed point`, the Data Type Assistant displays fields for specifying information about your fixed-point data type. For example, the next figure shows the Block Parameters dialog box for a Gain block, with the **Signal Attributes** tab selected and a fixed-point data type specified.



If the **Scaling** is `Slope and bias` rather than `Binary point`, the Data Type Assistant displays a **Slope** field and a **Bias** field rather than a **Fraction length** field:



You can use the Data Type Assistant to set these fixed-point properties:

Signedness

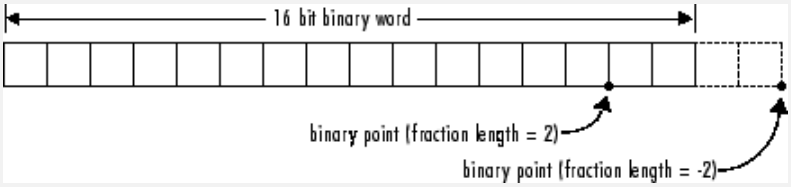
Specify whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative values, but unsigned data represents positive values only. The default setting is `Signed`.

Word length

Specify the bit size of the word that will hold the quantized integer. Large word sizes represent large values with greater precision than small word sizes. Word length can be any integer between 0 and 128. The default bit size is 16.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point` scaling. You can select one of two scaling modes:

Scaling Mode	Description
`Binary point`	If you select this mode, the Data Type Assistant displays the **Fraction Length** field, which specifies the binary point location. Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:  The default binary point is 0.
`Slope and bias`	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias**. Slope can be any positive real number, and the default slope is 1.0. Bias can be any real number, and the default bias is 0.0. You can enter slope and bias as expressions that contain parameters you define in the MATLAB workspace.

Note Use binary-point scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point scaling are performed with simple bit shifts and eliminate expensive code implementations, which are required for separate slope and bias values.

For more information about fixed-point scaling, see “Scaling” (Fixed-Point Designer).

Data type override

When the **Mode** is `Built in` or `Fixed point`, you can use the **Data type override** option to specify whether you want this data type to inherit or ignore the data type override setting specified for its context, that is, for the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal. The default behavior is `Inherit`.

Data Type Override Mode	Description
`Inherit` (default)	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.

Data Type Override Mode	Description
`Off`	Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

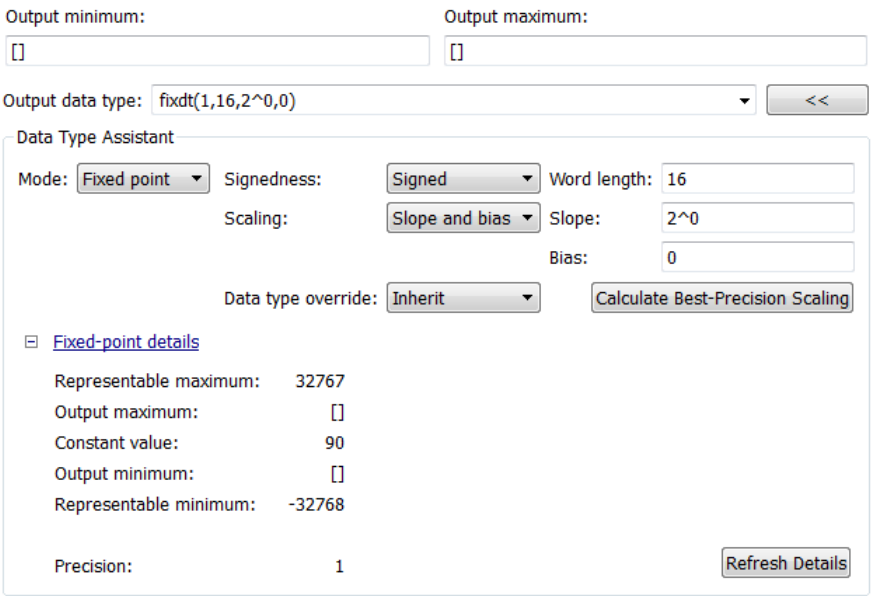
The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override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

Calculate Best-Precision Scaling

Click this button to calculate best-precision values for both `Binary point` and `Slope and bias` scaling, based on the specified minimum and maximum values. Simulink displays the scaling values in the **Fraction Length** field or the **Slope** and **Bias** fields. For more information, see "Constant Scaling for Best Precision" (Fixed-Point Designer).

Showing Fixed-Point Details

When you specify a fixed-point data type, you can use the **Fixed-point details** subpane to see information about the fixed-point data type that is currently displayed in the Data Type Assistant. To see the subpane, click the expander next to **Fixed-point details** in the Data Type Assistant. The **Fixed-point details** subpane appears at the bottom of the Data Type Assistant:



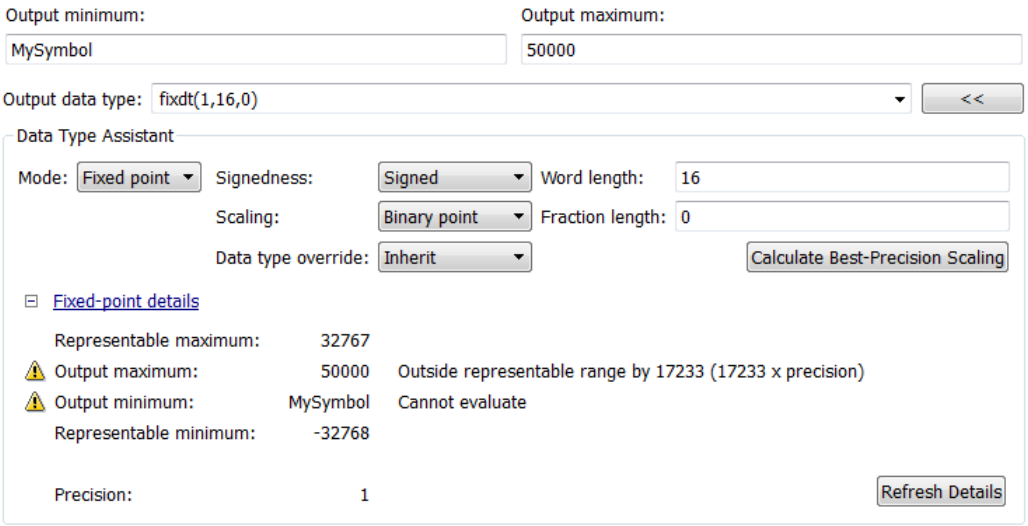
The rows labeled `Output minimum` and `Output maximum` show the same values that appear in the corresponding **Output minimum** and **Output maximum** fields above the Data Type Assistant. The names of these fields may differ from those shown. For example, a fixed-point block parameter would show **Parameter minimum** and **Parameter maximum**, and the corresponding **Fixed-point details** rows would be labeled accordingly. See "Specify Signal Ranges" on page 75-31 and "Specify Minimum and Maximum Values for Block Parameters" on page 37-52 for more information.

The rows labeled `Representable minimum`, `Representable maximum`, and `Precision` always appear. These rows show the minimum value, maximum value, and precision that can be represented by the fixed-point data type currently displayed in the Data Type Assistant.

The values displayed by the **Fixed-point details** subpane *do not* automatically update if you click **Calculate Best-Precision Scaling**, or change the range limits, the values that define the fixed-point data type, or anything elsewhere in the model. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**. The Data Type Assistant then updates or recalculates all values and displays the results.

Clicking **Refresh Details** does not change anything in the model, it only changes the display. Click **OK** or **Apply** to put the displayed values into effect. If the value of a field cannot be known without first compiling the model, the **Fixed-point details** subpane shows the value as `Unknown`.

If any errors occur when you click **Refresh Details**, the **Fixed-point details** subpane shows an error flag on the left of the applicable row, and a description of the error on the right. For example, the next figure shows two errors:

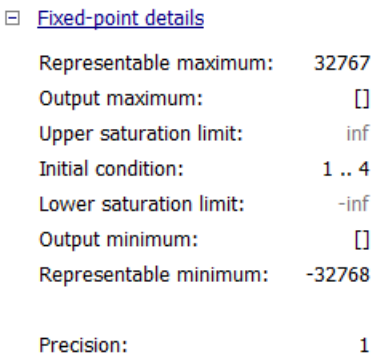


The row labeled `Output minimum` shows the error `Cannot evaluate` because evaluating the expression `MySymbol`, specified in the **Output minimum** field, did not return an appropriate numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane displays the unevaluated expression (truncating to 10 characters if necessary to save space) in place of the unavailable value.

To correct the error in this case, you would need to define `MySymbol` in an accessible workspace to provide an appropriate numeric value. After you clicked **Refresh Details**, the value of `MySymbol` would appear in place of its unevaluated text, and the error indicator and error description would disappear.

To correct the error shown for `Output maximum`, you would need to decrease **Output maximum**, increase **Word length**, or decrease **Fraction length** (or some combination of these changes) sufficiently to allow the fixed-point data type to represent the maximum value that it could have.

Other values relevant to a particular block can also appear in the **Fixed-point details** subpane. For example, on a Discrete-Time Integrator block **Signal Attributes** tab, the subpane can look like this:



The values displayed for **Upper saturation limit** and **Lower saturation limit** are greyed out. This appearance indicates that the corresponding parameters are not currently used by the block. The greyed-out values can be ignored.

To conserve space, **Initial condition** displays the smallest value and the largest value in the vector or matrix, using ellipsis to represent the other values. The underlying definition of the vector or matrix is unaffected.

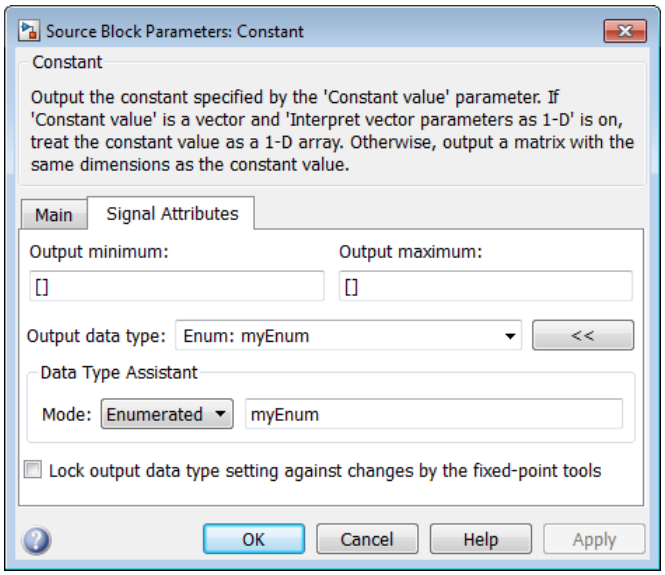
Lock output data type setting against changes by the fixed-point tools

Select this check box to prevent replacement of the current data type with a type that the Fixed-Point Tool or Fixed-Point Advisor chooses. For instructions on autoscaling fixed-point data, see “Scaling” (Fixed-Point Designer).

Specify an Enumerated Data Type

You can specify an enumerated data type by selecting the `Enum: <class name>` option and specify an enumerated object.

In the **Data Type Assistant**, you can use the **Mode** parameter to specify a bus as a data object for a block. Select the `Enumerated` option and specify an enumerated object.

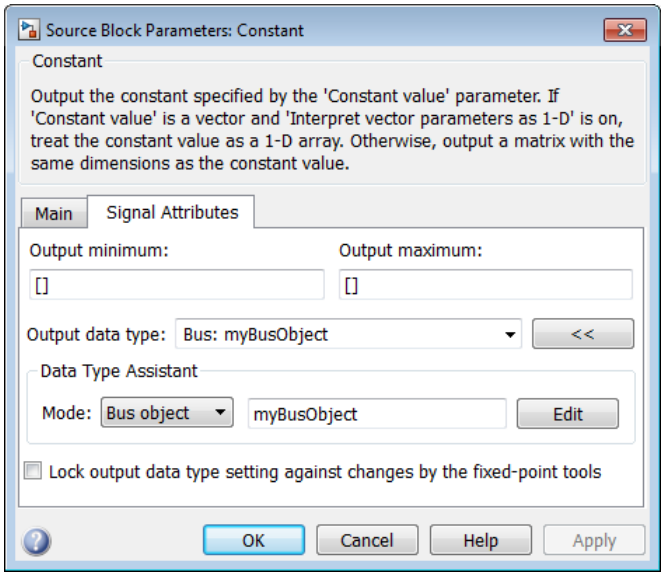


For details about enumerated data types, see “Data Types”.

Specify a Bus Object Data Type

The blocks listed in the section called “Data Types for Bus Signals” on page 67-39 support your specifying a bus object as a data type. For those blocks, in the **Data type** parameter, select the `Bus: <object name>` option and specify a bus object. You cannot use the `Expression` option to specify a bus object as a data type for a block.

In the **Data Type Assistant**, you can use the **Mode** parameter to specify a bus as a data object for a block. Select the `Bus` option and specify a bus object.



You can specify a bus object as the data type for data objects such as `Simulink.Signal`, `Simulink.Parameter`, and `Simulink.BusElement`. In the Model Explorer, in Properties dialog box for a data object, in the **Data type** parameter, select the `Bus: <object name>` option and specify a bus object. You can also use the `Expression` option to specify a bus object.

For more information on specifying a bus object data type, see “Specify Bus Properties with Simulink.Bus Objects” on page 76-44.

See Also

`Simulink.NumericType` | `fixdt`

Related Examples

- “Control Signal Data Types” on page 67-6
- “Specify Fixed-Point Data Types” on page 67-28
- “Define Simulink Enumerations” on page 68-6
- “About Data Types in Simulink” on page 67-2
- “Data Types Supported by Simulink” on page 67-4
- “Data Types for Bus Signals” on page 67-39

Data Types for Bus Signals

A bus object (`Simulink.Bus`) specifies the architectural properties of a bus, as distinct from the values of the signals it contains. For example, a bus object can specify the number of elements in a bus, the order of those elements, whether and how elements are nested, and the data types of constituent signals; but not the signal values.

You can specify a bus object as a data type for the following blocks:

- Bus Creator
- Constant
- Data Store Memory
- Inport
- Outport
- Signal Specification

You can specify a bus object as a data type for the following classes:

- `Simulink.BusElement`
- `Simulink.Parameter`
- `Simulink.Signal`

See “Specify a Bus Object Data Type” on page 67-38 for information about how to specify a bus object as a data type for blocks and classes.

See Also

`Simulink.Bus`

Related Examples

- “Control Signal Data Types” on page 67-6
- “Specify Data Types Using Data Type Assistant” on page 67-30
- “About Data Types in Simulink” on page 67-2
- “Data Types Supported by Simulink” on page 67-4

Simulink Strings

Use strings in multiple applications when you want to pass and manipulate text. For example, when modeling a fuel control system, instead of using enumerated data to model the fuel levels, you can use strings like `"LOW"` or `"EMPTY"`. Simulink strings are compatible with MATLAB strings.

Simulink strings are a built-in signal data type. They appear in the Simulink Editor as `"str`*N*`"` (for example, string with maximum length of *N* characters) or "string" for strings without maximum length (dynamic strings). String lengths can range from 1 to 32,766 characters.

Simulink string signals are inherently discrete. If your string signal has a continuous sample time, the model generates an error at compilation time.

Simulink treats string variables and expressions, such as "a" + "b", the same way it treats numeric variables. The value of a string variable can be both a character vector and a MATLAB string. String variables can exist in base, model, and mask workspaces.

String literals are specified with double quotes (`"Hello"`) or single quotes (`'Hello'`). To be consistent with MATLAB strings, use double quotes. Strings appear on ports and in the Display block with double quotes.

Simulink strings support 256 characters of the ISO/IEC 8859-1 character set. These characters are the first 256 code points of Unicode. Simulink does not support the first character `char(0)` ("NULL") and returns an error if the string contains this character.

When a character cannot be displayed, the block stores the actual information and outputs an escape character with the associated octal value for the character. For example, the decimal value control character for `BREAK PERMITTED HERE` is 130. The block displays this control character as the escaped octal `\202`. The ASCII to String block returns as escaped octals characters in the Unicode set range 0000 to 001F and 007F-009F.

This topic describes how to use strings in Simulink, including:

- A list of available string blocks and blocks particular to string conversions
- Passing string constants into other string blocks
- Null characters in strings
- Strings with no maximum length
- Interactions with Stateflow
- Generated code
- Limitations

This topic also includes simple examples illustrating how to use string blocks. Examples in this topic enable the display of block names. To control the display of block names, on the **Format** tab, select **Auto > Hide Automatic Block Names**. For example, you can use string blocks to display and extract coordinate data and find patterns in strings.

To work with strings in your model, use this table:

Action	Block
Convert a uint8 vector to a string signal.	ASCII to String

Action	Block
Compose an output string signal based on the **Format** parameter and input signals.	Compose String
Scan an input string and convert it to signals per the format specified by **Format** parameter.	Scan String
Compare two input strings.	String Compare
Concatenate input strings to form one output string.	String Concatenate
Output the string specified by the **String** parameter.	String Constant
Return the index of the first occurrence of the pattern string **sub** in the text string **str**.	String Find
Output the number of characters in the input string.	String Length
Convert a string signal to a uint8 vector.	String to ASCII
Convert string signal to double signal.	String to Double
Convert string signal to single signal.	String to Single
Convert an input string to an enumerated signal.	String to Enum
Extract a substring from a string signal.	Substring
Convert the input signal to a string signal.	To String

These Simulink blocks support strings.

Block	Notes
Bus Assignment	Nonvirtual and virtual.
Bus Creator	Nonvirtual and virtual.
Bus Selector	Nonvirtual and virtual.
Data Store Memory	—
Data Store Read	—
Data Store Write	—
Data Type Duplicate	—
Display	Display strings with double quotes.
From	—
From Workspace	Interpolation of data is not supported.
Goto	—
Ground	Ground value is "" (empty string).
Inport	Including root level.
Manual Variant Sink	—
Manual Variant Source	—

Block	Notes
Manual Switch	No mixed numeric and string types. If there are multiple strings, the block uses string with the largest size or the dynamic string.
MATLAB Function	—
Merge	—
Model	—
Multiport Switch	No mixed numeric and string types. If there are multiple strings, the block uses string with the largest size or the dynamic string.
Outport	Including root level.
Probe	—
Signal Editor	Interpolation of data is not supported.
Signal Specification	—
Subsystem (all variations)	—
Switch	No mixed numeric and string types. If there are multiple strings, the block uses string with the largest size or the dynamic.
Terminate Function	—
Terminator	—
Variant Source	—
Variant Model	—
Variant Subsystem	—
Width	—

Simulink Strings and Stateflow

To use textual data to control chart behavior and manipulate text to create natural language output in Stateflow, use strings in C action language charts. Stateflow provides operators to manipulate strings. For more information, see “Manage Textual Information by Using Strings” (Stateflow).

String Constants

To specify string constants, use the String Constant block. Do not use the Constant block, which does not accept strings. In the String Constant block, enter the string with double quotes, such as `"Hello!"` in the **String** parameter. It is the same as adding a numeric constant in the **Constant value** parameter of the Constant block.

Simulink Strings and Null Characters

Simulink strings automatically deal with string termination. Do not use a null terminator at the end of a string. If Simulink detects a null character anywhere in a string, it generates an error. Simulink handles strings during simulation and code generation as follows:

- During simulation, the model simulates. You do not need to do anything else.
- During C code generation, the software adds a null terminator to the end of the string. For example, if the string buffer size is 10 and the real string value is `"AB"`, the third character in the generated code is a null terminator.

String Data Type

As necessary, string blocks create and use string data types, for example, when a block outputs a string type. You can create strings without specifying a maximum length of characters. We refer to strings without a maximum length of characters as dynamic strings.

To create string data types for blocks that support strings, you can:

- Use the **Output data type** or **Data type** parameter on the **Signal Attributes** tab of a Simulink block.
  - To create a string data type with no maximum length of characters, specify `string`. This action creates a dynamic string.
  - To create a string data type with maximum length of characters, such as `100`, enter `stringtype(100)`. You can also use the `stringtype` function on the MATLAB command line to create a string data type.
- Use the `set_param` function with the `OutDataTypeStr` parameter, for example:

  ```
  set_param(gcb,'OutDataTypeStr','stringtype(100)')
  ```

  This function creates a string data type object with a maximum length of 100 characters. String type lengths can range from 1 to 32,766 characters, inclusive.

For example, using the String Constant block to create a string with the **Output data type** parameter set to `stringtype(31)` creates a string data type of `str31`. 31 is the maximum number of characters the string can have.

Strings in Bus Objects

To configure an element of a bus object to accept strings using the Bus Editor, in the **DataType** parameter of the bus element, enter a string type. For example, to specify a string data type whose maximum length of characters is 10, enter `stringtype(10)`. The **Mode** parameter updates accordingly. To specify a dynamic signal with a variable length, enter `string` in this field. You can create mixed use numeric and string bus objects for bus elements. For more information on creating bus objects and elements with the Bus Editor, see “Create and Specify Simulink.Bus Objects” on page 76-46.

Strings and Generated Code

Consider these notes for strings and generated `C` and `C++` code.

Differences Between Simulation and Generated C Code for the Dynamic String Data Type

Simulation of blocks that use a dynamic string data type differ from generated `C` code in these ways:

- Signals with `string` data type have dynamic sizes during simulation. In other words, the size of a string can vary between 0 and 32766. However, in generated `C` code, the coder stores strings as

fixed-size `char_T` arrays. Therefore, the size of a string is limited by a fixed buffer size in generated `C` code. The software truncates extra characters and inserts a null terminator at the end (for example, at the output of a String Concatenate block).

- The coder generates string invariants (parameters or signals) as `C`-style double-quoted strings.
- The default buffer size for a signal with string data type is 256 bytes (contains at most 255 8-bit ASCII characters). To change this buffer size, use the “Buffer size of dynamically-sized string (bytes)” (Simulink Coder) configuration parameter.
- To override a model-wide setting for an individual string signal, use `stringtype(`*`N`*`)` as the signal data type. Signals with this data type are allocated a buffer of *`N`*`+1` bytes in generated `C` code (*N* characters plus one null terminator).
- There is no difference between simulation and generated `C++` code when strings are stored as `std::string` objects that have the same dynamic behavior as simulation.

C++ Code Generation String Library

For `C++` code generation, `std::string` library is available for an ERT-based target with an Embedded Coder license. For more information, see “Generate Code for String Blocks by using the Standard C++ String Library” (Embedded Coder).

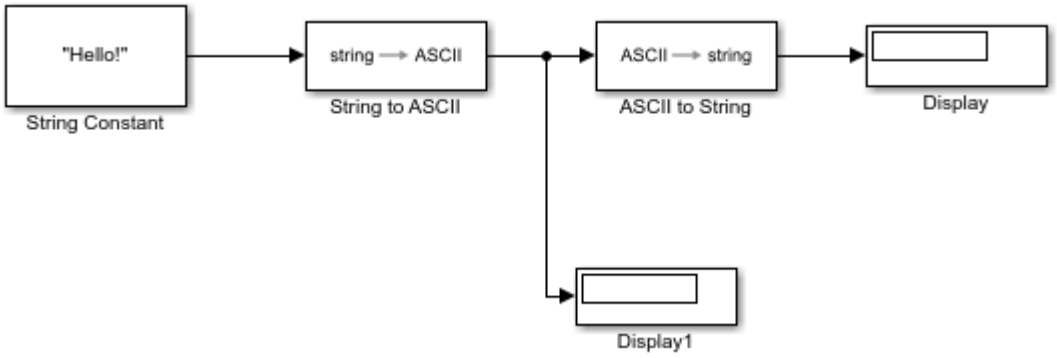
String Data Type Conversions

You cannot use the Data Type Conversion block to convert string data types to other data types and conversely. Instead, use these string conversion blocks.

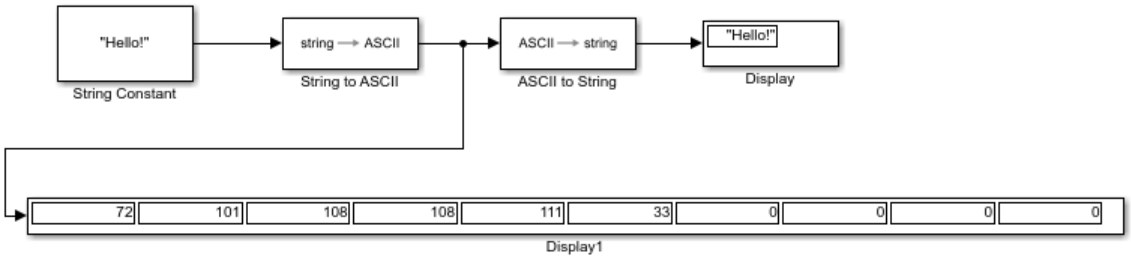
To Convert	Block
String signal to uint8 vector signal while preserving ASCII characters	String To ASCII
Uint8 vector signal to string signal	ASCII to String
String signal to numerical signal double data type	String to Double
String signal to numerical signal single data type	String to Single
String signal to enumerated signal data type	String To Enum
Input signal to string signal	To String

Convert String to ASCII and Back to String

1. Add these blocks to a model:
   - String Constant
   - String to ASCII
   - ASCII to String
   - Two Display blocks
2. Connect the blocks as shown.

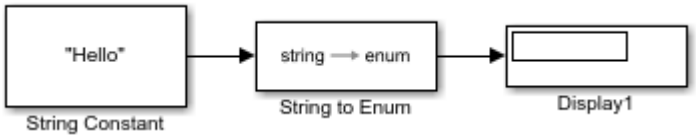


3. In the String Constant block, enter a string, such as `"Hello!"`.
4. In the String to ASCII block, change the maximum string size to 10.
5. Simulate the model and observe the contents of the Display blocks.
   - Display1 shows `Hello!` converted to its ASCII equivalent. For example, `72` is the ASCII equivalent of `H` and `33` is the ASCII equivalent of `!`.
   - Display1 has filled the remaining space to the maximum string length of 10 with zeros (null characters).
   - Display shows `Hello!` after the ASCII to String block reconverts the ASCII code to a string.



Convert String to Enumerated Data Type

1. Add these blocks to a model:
   - String Constant
   - String to Enum
   - Display
2. Connect the blocks as shown.



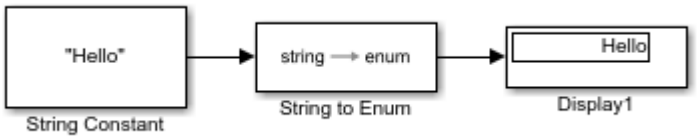
3. In the String Constant block, enter a string, such as `"Hello!"`
4. Create a Simulink enumeration class named `BasicStrings` and store it in the current folder with the file name `BasicStrings.m`, for example:

```
classdef BasicStrings < Simulink.IntEnumType
enumeration
```

```
    Hello(0)
    Red(1)
    Blue(2)
    end
end
```

5 In the String to Enum block, enter the enumeration class as `Enum: BasicStrings`.

6 Simulate the model and observe the contents of the Display block.



Display and Extract Coordinate Data

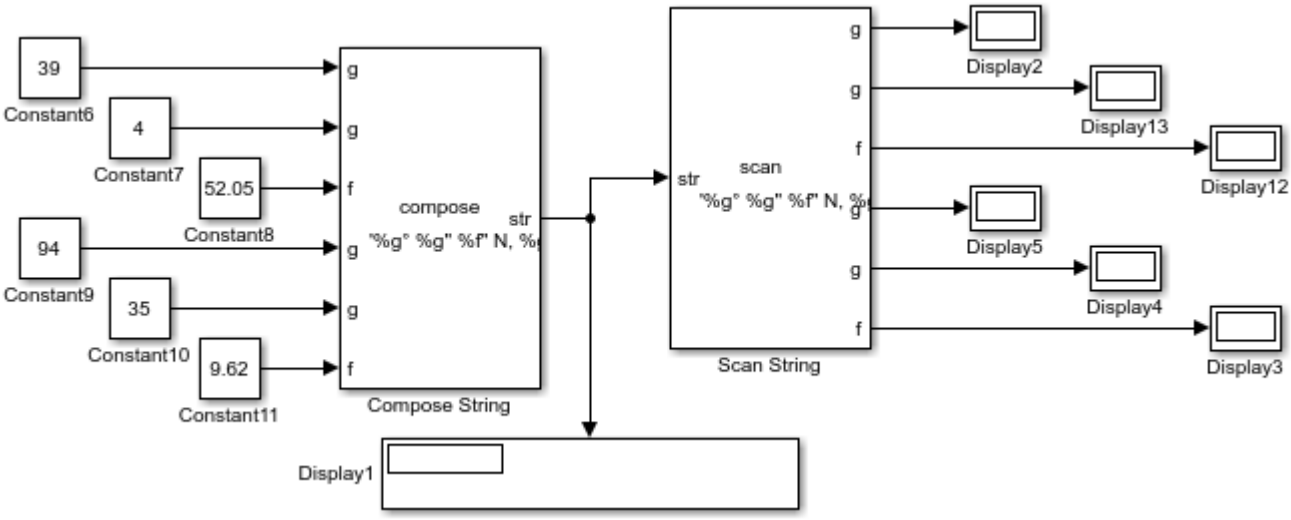
This example shows how you can format and output a set of data as geographic coordinates using the Compose String and Scan String blocks. Based on the C `scanf` and `printf` functions, the Compose String and Scan String blocks are similar in concept to the `sprintf` and `sscanf` functions, with the primary exception being that those functions work with arrays, which the blocks do not. For more information on string block formatted characters, see Compose String and Scan String.

The Compose String block constructs a string from multiple string and numerical inputs combined. Use the **Format** parameter to format the output of each input, one format operator for each input. Each format operator starts with a percent sign, `%`, followed by the conversion character, for example, `%f` generates fixed point output. To supplement the string output, you can also add extra characters to the format specification to appear in the output.

1 Add these blocks to a model:

- Six Constant
- One Compose String
- One Scan String
- Seven Display

2 Change the Constant block constant values to those shown and connect the blocks.



3 In the **Format** parameter for the Compose String blocks, enter these format specifications:

```
'%g° %g'' %f" N, %g° %g'' %f" W'
```

- The `%g` and `%f` formatting operators convert numeric inputs to floating point values. `%g` is a more compact version of `%f`.
- The degree symbol (°), N, W, and `'` are supplemental strings to display in the output string.

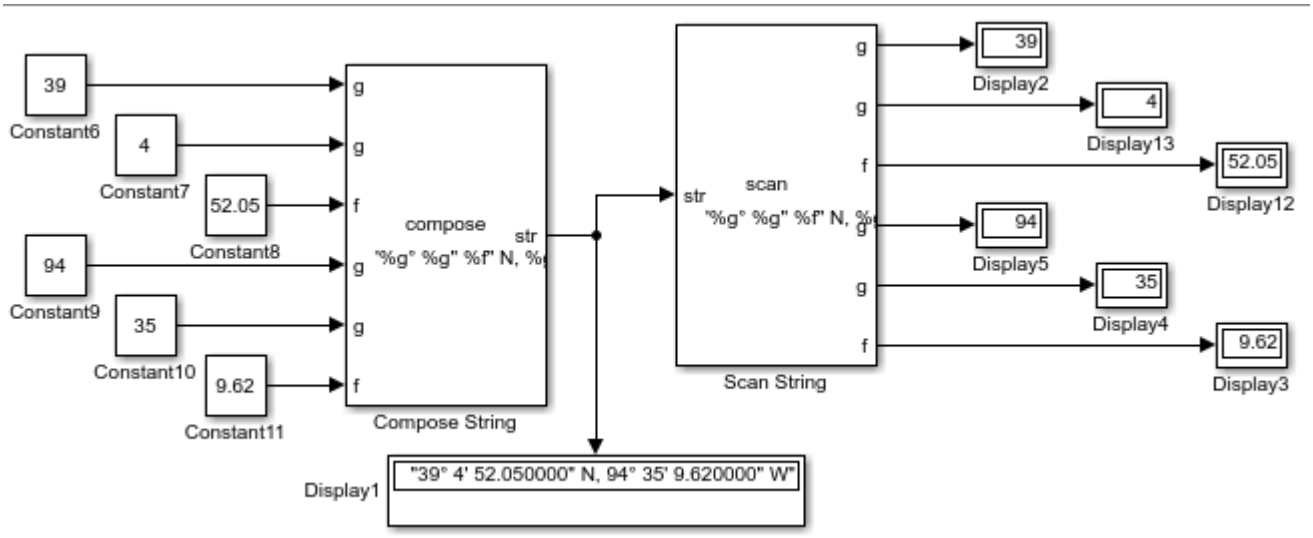
The Compose String block combines the output in the input order, formats each input according to its format operator, adds the extra strings, and outputs the string, formatted as directed and surrounded by double quotes (").

4 In the **Format** parameter for the Scan String block, enter these format specifications:

```
'%g° %g'' %f" N, %g° %g'' %f" W'
```

- The `%g` and `%f` formatting operators convert numeric inputs to floating point values. `%g` is a more compact version of `%f`.
- The degree symbol (°), N, W, and `'` are supplemental strings to display in the output string.

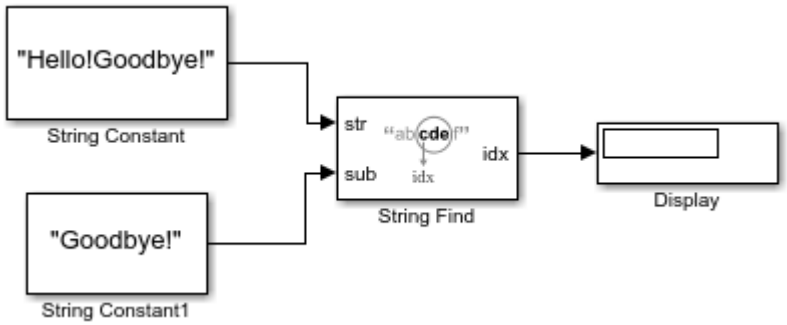
The Scan String block reads its input, converts it according to the format specified by a conversion specification, and returns the results as scalars. The block formats each output according to its conversion specification. It ignores specified text immediately before or after a conversion specifier.



Find Patterns in Strings

To find a pattern in a string, use the String Find block.

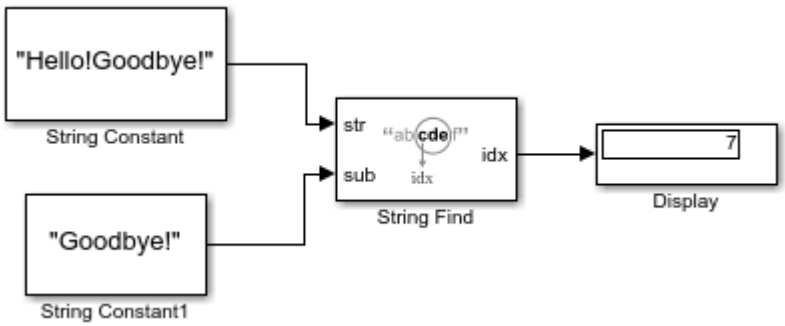
1. Add these blocks to a model:
   - Two String Constant
   - String Find
   - One Display
2. Connect the blocks as shown.



3. In the first String Constant block, enter a string, such as `"Hello!Goodbye!"`.

   Connecting this block to the **str** input port of the String Find block causes the String Find block to look for the pattern in this string.
4. In the second String Constant block, enter a string (or pattern) to look for in the first String Constant string, such as `"Goodbye!"`.

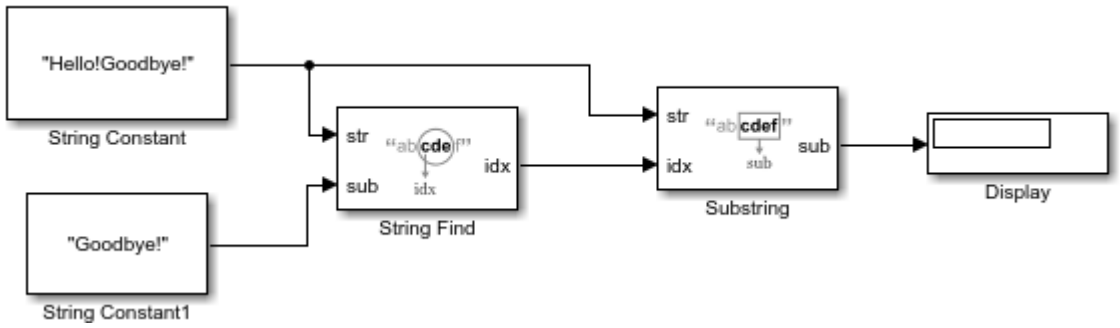
   Connecting this block to the **sub** input port of the block means that the String Find looks for this pattern from the **str** input.
5. Simulate the model and observe the contents of the Display block. For this example, the block displays `7`, which is the location of the letter `G`.



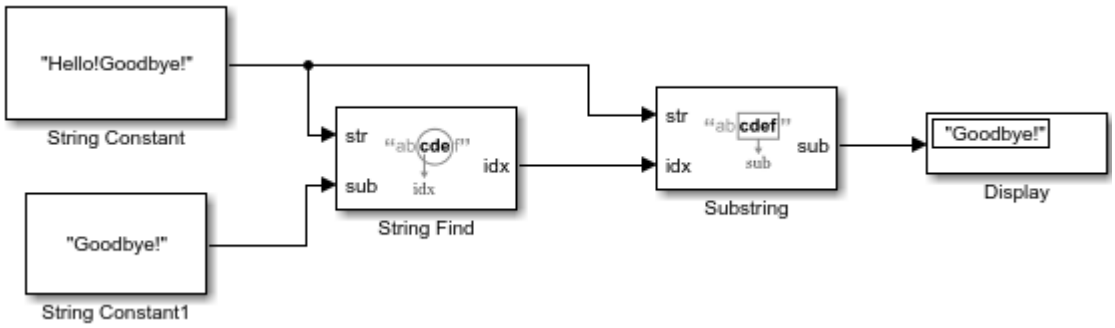
Extract a String

To extract a string from a source string, use the Substring block. This example uses the model described in “Find Patterns in Strings” on page 67-48.

1. Add a Substring block to the model.
2. In the Substring block, select the **Output string from 'idx' to end** parameter. Setting this parameter extracts the string from the location input at the idx port to the end of the string.
3. Connect the new block as shown.



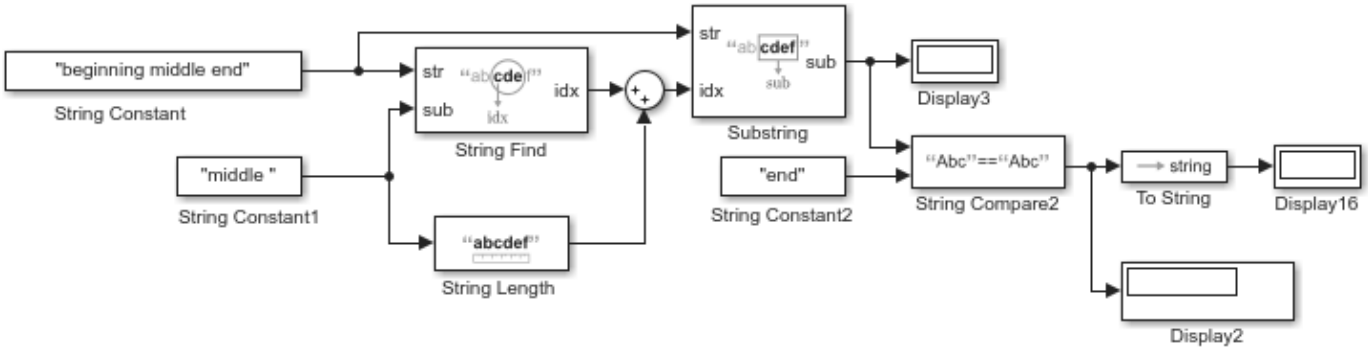
4. Simulate the model and observe the contents of the Display block. For this example, the block displays `"Goodbye!"`, which is the substring extracted starting **idx** to the end of the string.



Get Text Following a Keyword

This example shows basic string manipulation using the Simulink string blocks.

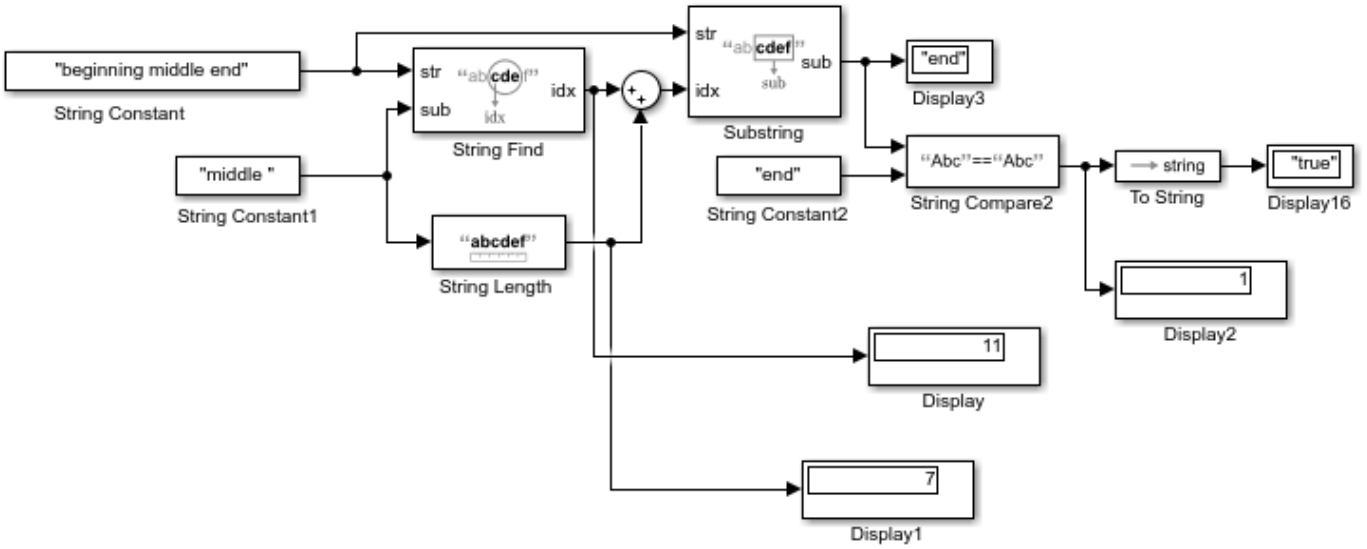
For a model that looks like the following, simulate it.



Observe that the model:

- Creates two strings, `"beginning middle end"` and `"middle"`, using String Constant blocks.
- Looks for the first occurrence of `"middle"` (**idx**) and adds the location of the first letter (`11`) to the length of `"middle"` (7). It uses the String Find and String Length blocks.
- Extracts from `"beginning middle end"` the substring that starts from the end of `"middle"` (`idx+string length = 18`), which is the string `"end"`. It uses the Substring block.
- Compares the calculated value of `"end"` with the actual string `"end"` , which returns the Boolean value `"1"`. It uses the String Constant and String Compare blocks.
- Converts the Boolean value `"1"` to its string equivalent, `"true"`. It uses the To String block.

To see the locations of the characters throughout the model, add one Display block each to the output of the String Find and String Length blocks and simulate it.

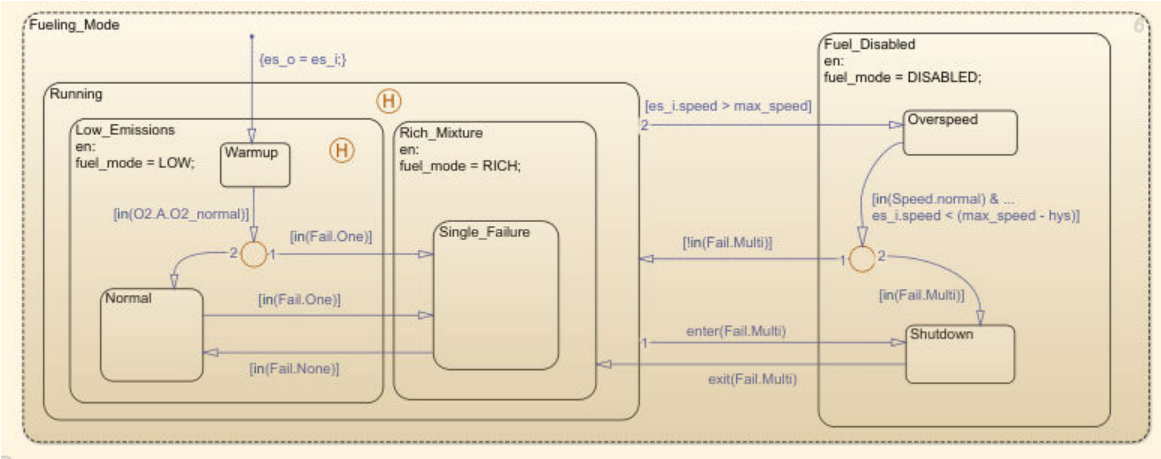


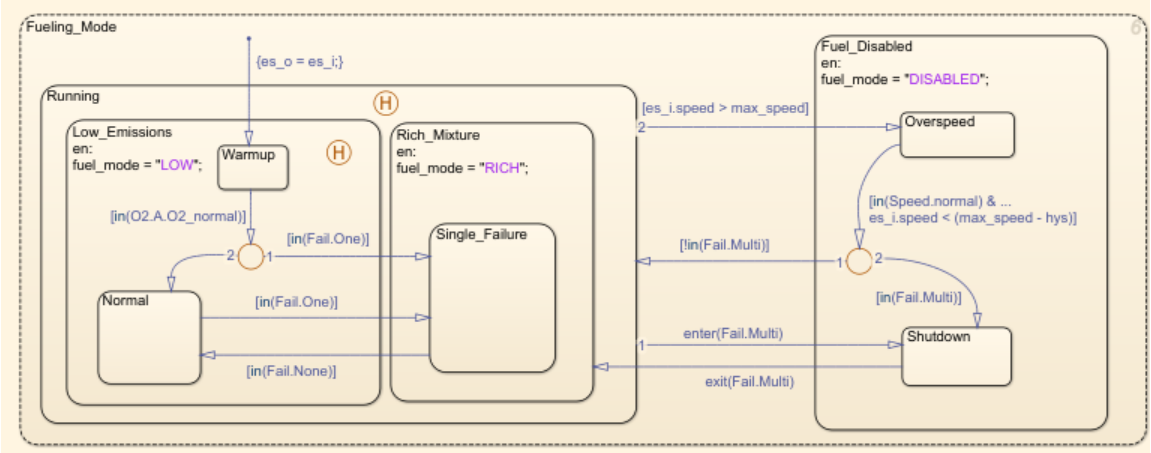
Change Existing Models to Use Strings

In addition to using strings in new models, you can update existing models to use strings. Using strings can simplify the model and make it easier to understand.

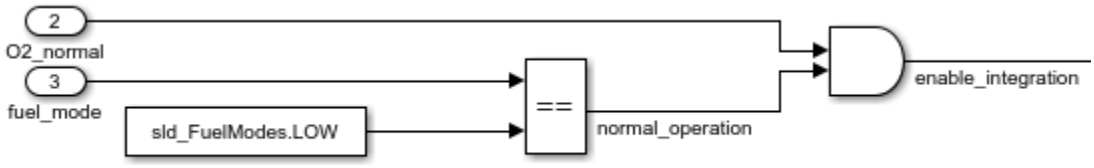
For example, in older models, you may have used enumerated data types to represent or pass text in your model. The `sldemo_fuelsys` example enumerated constants in multiple areas to work with textual data.

In `sldemo_fuelsys/fuel_rate_control/control_logic`, the Stateflow chart uses enumerated data to indicate fuel levels.





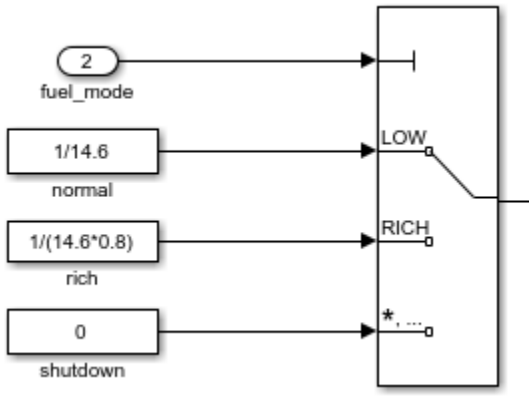
In `sldemo_fuelsys/fuel_rate_control/airflow_calc`, to detect if the fuel is low, the model uses Enumerated Constant and Relational Operator blocks.



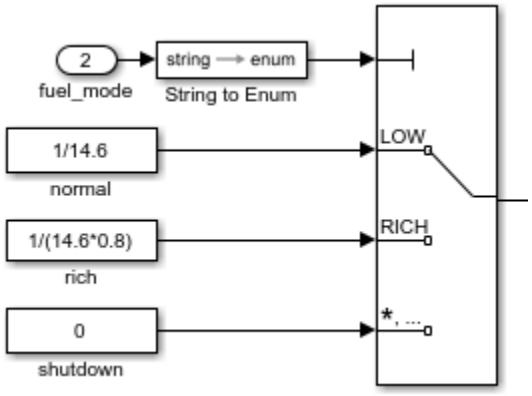
Instead, you can use the String Constant and String Compare blocks by setting:

- The String Constant **String** parameter to `LOW`.
- Using the String Compare block to detect if `LOW` is coming from another part of the model.

Instead of removing all instances of enumerated constants, you can use strings in conjunction with enumerated constants. Doing so allows you to incrementally migrate your model to use strings. In `sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate`, the Multiport Switch block accepts four enumerated data inputs.



If the **fuel_mode** port is outputting a string, you can convert that string to an enumerated data type to work with the output from the Constant blocks in this model.

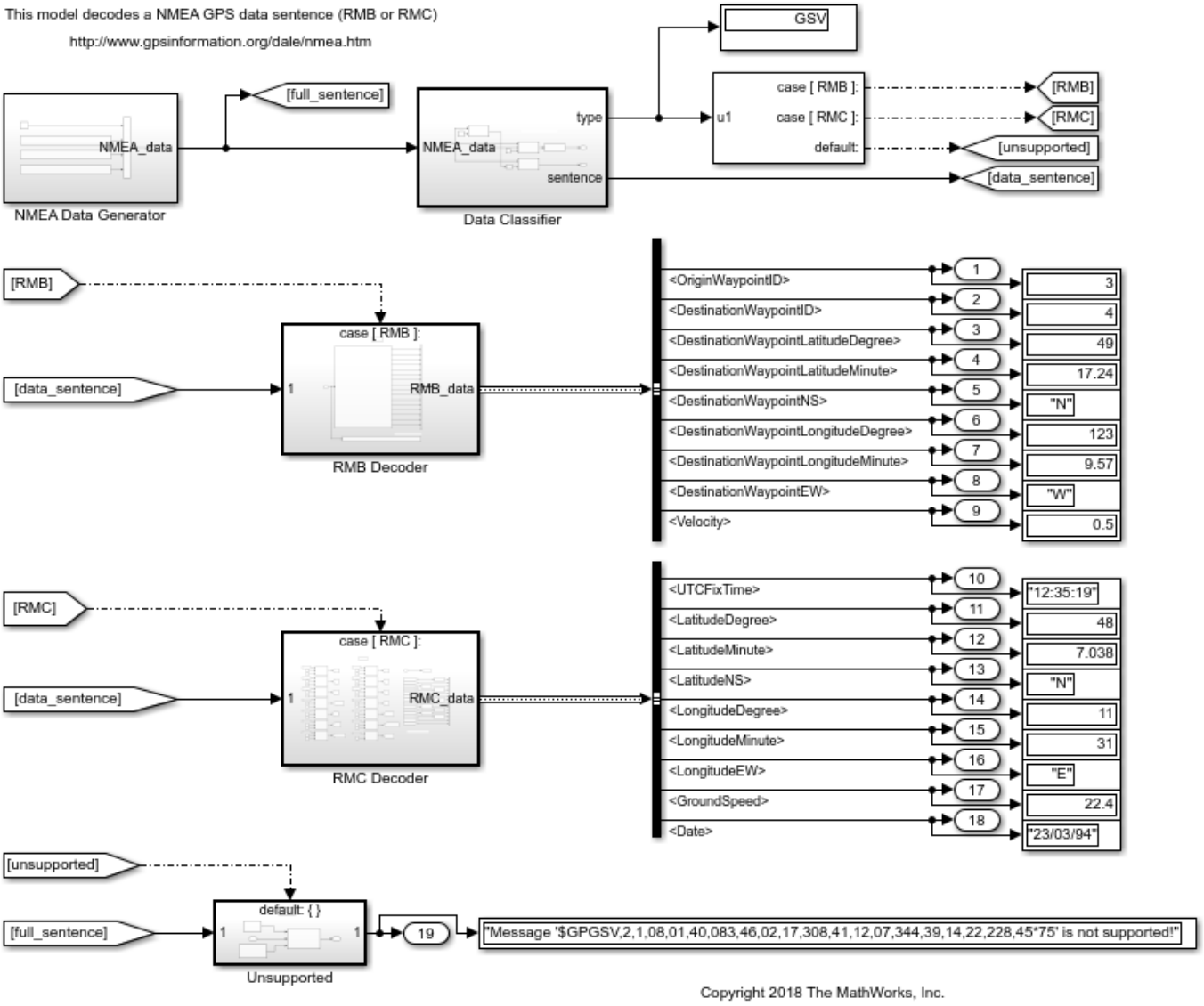


Parse NMEA GPS Text Message

This example shows how to parse text messages in NMEA GPS format using Simulink® string blocks.

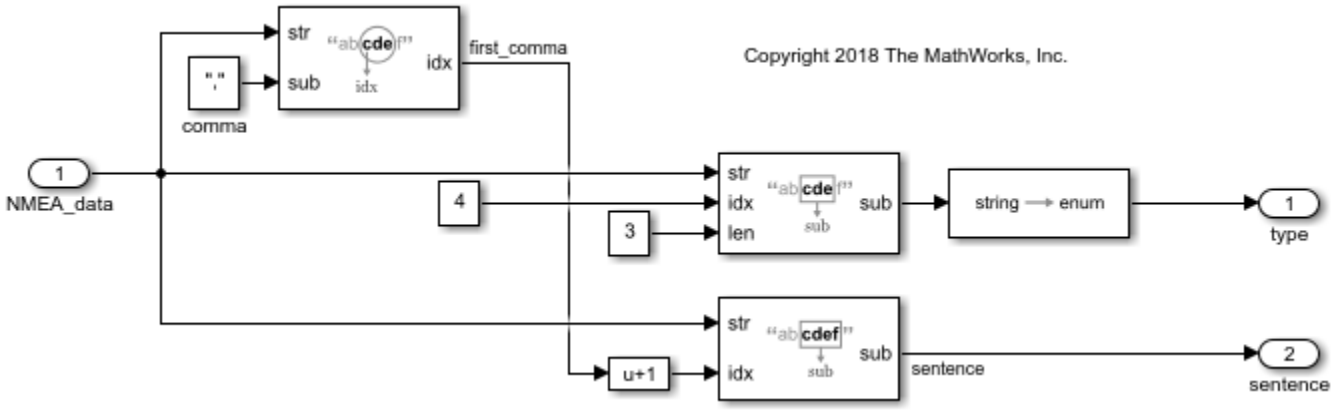
Overview

This model shows how to use string data type and blocks provided by Simulink® to read input text messages and extract numeric and text data.



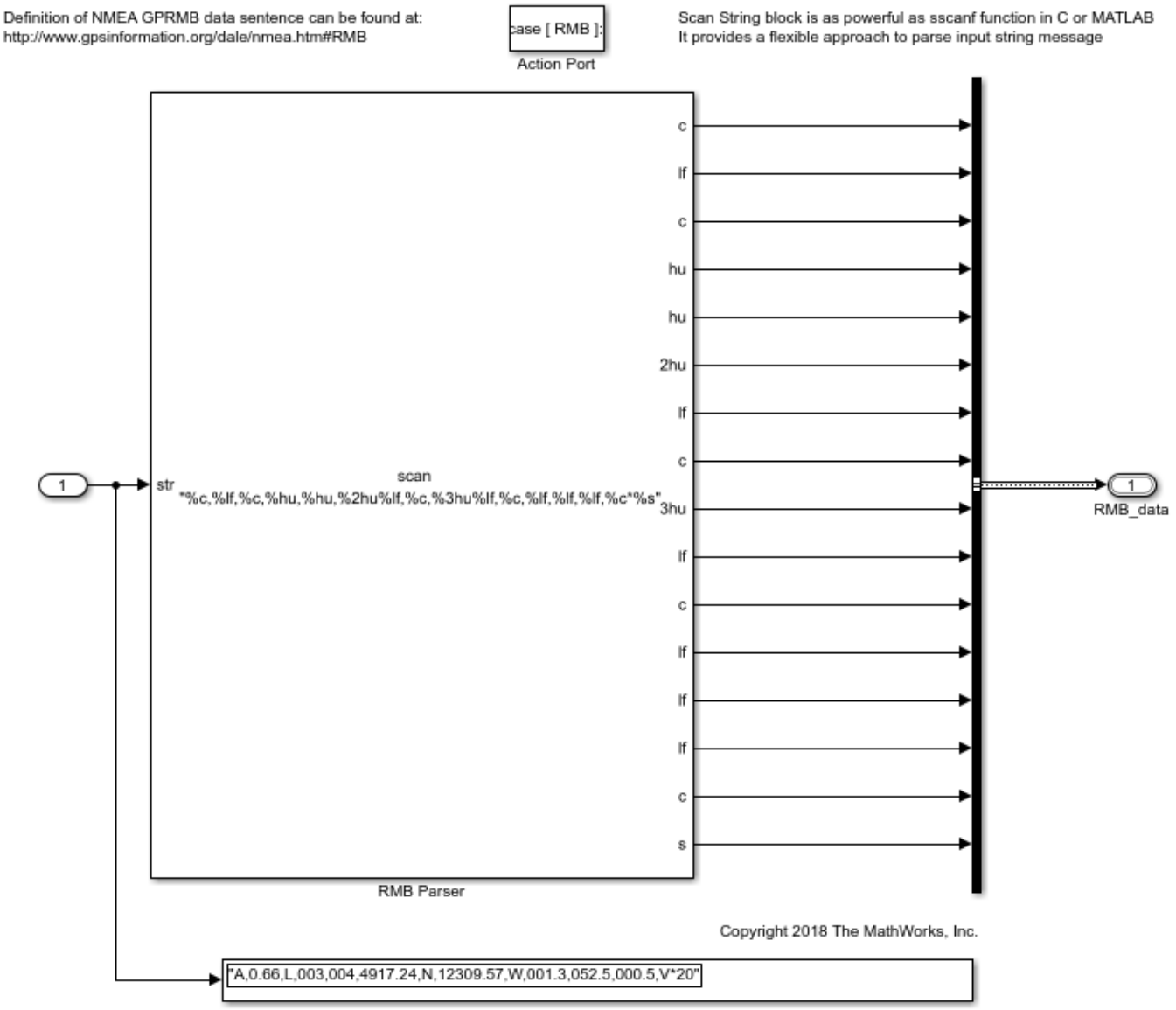
Read message header and convert to enumeration

Simulink® provides the String To Enum block to convert a string to the corresponding enumeration value. In this model, header string "RMB" is converted to NMEASentence.RMB.



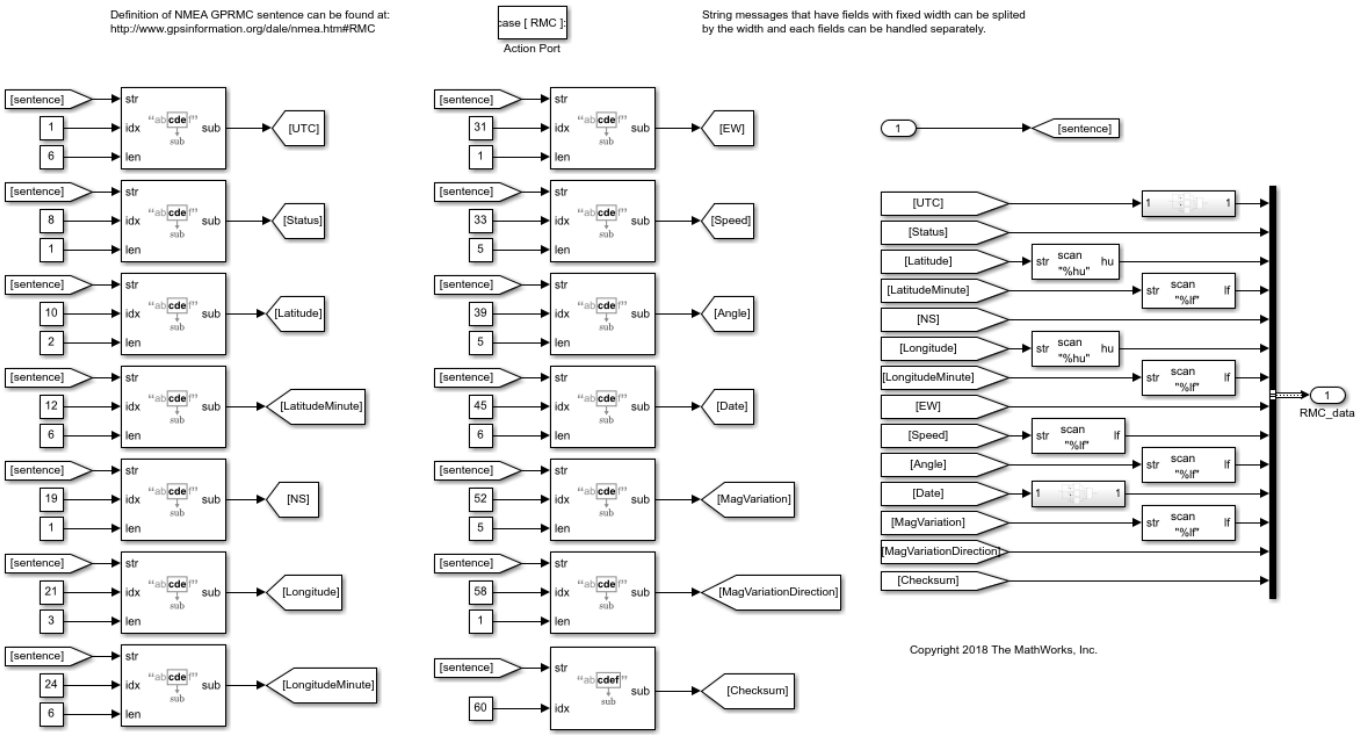
Parse text message using Scan String block

One way to parse a text message is to use the Scan String block. It works like sscanf function in C and MATLAB®.



Parse text message with fixed field width

When a text message has fixed width for each data field, Simulink® provides blocks to split the string by index. After the strings have been split, each field can be handled separately.



Simulink String Limitations

These capabilities are currently not supported:

Category	Limitation Description	Workaround
String array	String arrays are not supported.	Use strings only as scalars.
Unicode characters	Simulink strings do not support the entire Unicode set.	Simulink strings support 256 characters of the ISO/IEC 8859-1 character set (Basic Latin and Latin-1 supplement). These characters are the first 256 code points of Unicode.
MATLAB System, MATLAB S-Function, Stateflow MATLAB chart	Custom blocks created with these blocks do not support strings.	—
Constant, Initial Condition	These blocks do not work with strings.	Use String Constant block.
Control input of the Switch, Multiport Switch, Switch Case, and If blocks	The control input of the Switch block does not accept strings.	Use the data inputs of these blocks.
Simulink.Signal, Simulink.Parameter	The Simulink.Signal and Simulink.Parameter blocks do not support strings.	—

Category	Limitation Description	Workaround
Data Type Conversion block	Do not use the Data Type Conversion block to convert to or from strings.	Use the string conversion blocks
Logging of nonvirtual buses that contain string elements	If a nonvirtual bus contains a string element, the entire virtual bus cannot be logged. You can still log the nonstring elements.	
To Workspace block	Load strings using To Workspace.	Log using output port.
To File block	Load strings using To File.	—
Scope and Spectrum Analyzer blocks, Logic Analyzer app	The Scope and Spectrum Analyzer blocks and the Logic Analyzer app do not display strings.	Use the Display, the Simulation Data Inspector, or the Sequence Viewer in Stateflow.
S-functions	S-functions do not support the string data type.	—
Simulink Real-Time	Applications created by Simulink Real-Time using C++ as the target language errors out.	Use C as the target language.

See Also

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

More About

- “About Data Types in Simulink” on page 67-2
- “Data Types Supported by Simulink” on page 67-4
- “Manage Textual Information by Using Strings” (Stateflow)

Data Objects

In this section...

You can create data objects to specify values, value ranges, data types, tunability, and other characteristics of signals, states, and block parameters. You use the object names in Simulink dialog boxes to specify signal, state, and parameter characteristics. The objects exist in a workspace such as the base workspace, a model workspace, or a Simulink data dictionary. Data objects allow you to make model-wide changes to signal, state, and parameter characteristics by changing only the values of workspace objects.

You create data objects as instances of data classes. Memory structures called data class packages contain the data class definitions. The built-in package `Simulink` defines two data classes, `Simulink.Signal` and `Simulink.Parameter`, that you can use to create data objects. To store lookup table data for sharing between lookup table blocks (such as n-D Lookup Table), you can use the `Simulink.LookupTable` and `Simulink.Breakpoint` classes.

To decide whether to use data objects to configure signals, including Inport and Outport blocks, see “Store Design Attributes of Signals and States” on page 75-5.

You can customize data object properties and methods by defining subclasses of the built-in data classes. For more information about creating a data class package, see “Define Data Classes” on page 67-96.

Data Class Naming Conventions

Simulink uses dot notation to name data classes:

`package.class`

- *package* is the name of the package that contains the class definition.
- *class* is the name of the class.

This notation allows you to create and reference identically named classes that belong to different packages. In this notation, the name of the package qualifies the name of the class.

Class and package names are case sensitive. For example, you cannot use `MYPACKAGE.MYCLASS` and `mypackage.myclass` interchangeably to refer to the same class.

Use Data Objects in Simulink Models

To specify simulation and code generation options for signals, block parameters, and states by modifying variables in a workspace or data dictionary, use data objects. Associate the objects with signals, parameters, and states in a model diagram.

Use Parameter Objects

You can use parameter objects, instead of numeric MATLAB variables, to specify values for block parameters. For example, to create and use a `Simulink.Parameter` object named `myParam` to specify the **Gain** parameter of a Gain block:

1. In the model, on the **Modeling** tab, under **Design**, click **Property Inspector**.
2. In the model, click the target Gain block. The Property Inspector shows the properties and parameters of the block.
3. Set the value of the **Gain** parameter to `myParam`.
4. Next to the parameter value, click the action button ⋮ and select **Create**.
5. In the **Create New Data** dialog box, set **Value** to `Simulink.Parameter(15.23)` and click **Create**.

   The `Simulink.Parameter` object, `myParam`, appears in the base workspace. The property dialog box shows that the object stores the parameter value `15.23` in the **Value** property.
6. Use the property dialog box to specify other characteristics for the block parameter by adjusting the object properties. For example, to specify the minimum and maximum values the parameter can take, use the **Minimum** and **Maximum** properties.

During simulation, the **Gain** parameter now uses the value `15.23`.

To share lookup table data by using `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Package Shared Breakpoint and Table Data for Lookup Tables” on page 37-29.

Use Signal Objects

You can associate a signal line or block state, such as the state of a Unit Delay block, with a signal object.

For Signals

To use a signal object to control the characteristics of a signal in a model, create the object in a workspace by using the same name as the signal.

1. In the model, on the **Modeling** tab, click **Model Data Editor**.
2. In the Model Data Editor, select the **Signals** tab.
3. In the model, select the target signal. The Model Data Editor highlights the row that corresponds to the signal.
4. In the Model Data Editor, in the **Name** column, give the signal a name such as `mySig`.
5. Click the button ⋮ next to the signal name. Select **Create and Resolve**.

6 In the Create New Data dialog box, set **Value** to `Simulink.Signal`. Use the **Location** drop-down list to select a workspace to store the object (the default value is `Base Workspace`). Click **Create**.

   The `Simulink.Signal` object `mySig` appears in the target workspace. Simulink selects the signal property **Signal name must resolve to Simulink signal object**, which forces the signal in the model to use the properties that the signal object stores. To learn how to control the way that signal names resolve to signal objects, see “Symbol Resolution” on page 67-127.

   The property dialog box of the new object opens.

7 Use the property dialog box to specify the signal characteristics. Click **OK**.

To configure the signal programmatically:

```
% Create the signal object.
mySig = Simulink.Signal;
mySig.DataType = 'boolean';

% Get a handle to the block port that creates the
% target signal.
portHandles = get_param('myModel/myBlock','portHandles');
outportHandle = portHandles.Outport;

% Specify the programmatic port parameter 'Name'.
set_param(outportHandle,'Name','mySig')

% Set the port parameter 'MustResolveToSignalObject'.
set_param(outportHandle,'MustResolveToSignalObject','on')
```

To configure a root-level Outport block programmatically, you must use a slightly different technique:

```
% Create the signal object.
mySig = Simulink.Signal;
mySig.DataType = 'boolean';

% Specify the programmatic block parameter 'SignalName'.
set_param('myModel/myOutport','SignalName','mySig')

% Set the block parameter 'MustResolveToSignalObject'.
set_param('myModel/myOutport','MustResolveToSignalObject','on')
```

For States

You can use a signal object to control the characteristics of a block state, such as that of the Discrete-Time Integrator block.

1 In the model, on the **Modeling** tab, click **Model Data Editor**.

2 In the Model Data Editor, select the **States** tab.

3 In the model, select the block that harbors the target state. The Model Data Editor highlights the row that corresponds to the state.

4 In the Model Data Editor, in the **Name** column, give the state a name such as `myState`.

5 Click the button ⋮ next to the state name. Select **Create and Resolve**.

6 In the Create New Data dialog box, set **Value** to `Simulink.Signal`. Use the **Location** drop-down list to select a workspace to store the object (the default value is `Base Workspace`). Click **Create**.

   The `Simulink.Signal` object `myState` appears in the target workspace. Simulink selects the block parameter **State name must resolve to Simulink signal object**, which forces the state in the model to use the properties that the signal object stores. To learn how to control the way that state names resolve to signal objects, see “Symbol Resolution” on page 67-127.

   The property dialog box of the new object opens.

7 Use the property dialog box to specify the state characteristics. Click **OK**.

To configure the state programmatically:

```
% Create the signal object.
myState = Simulink.Signal;
myState.DataType = 'int16';

% Set the state name in the block.
set_param('myModel/myBlock','StateName','myState')

% Set the port parameter 'StateMustResolveToSignalObject'.
set_param('myModel/myBlock','StateMustResolveToSignalObject','on')
```

Data Object Properties

To control parameter and signal characteristics using data objects, you specify values for the data object properties. For example, parameter and signal data objects have a `DataType` property that determines the data type of the target block parameter or signal. Data class definitions determine the names, value types, default values, and valid value ranges of data object properties.

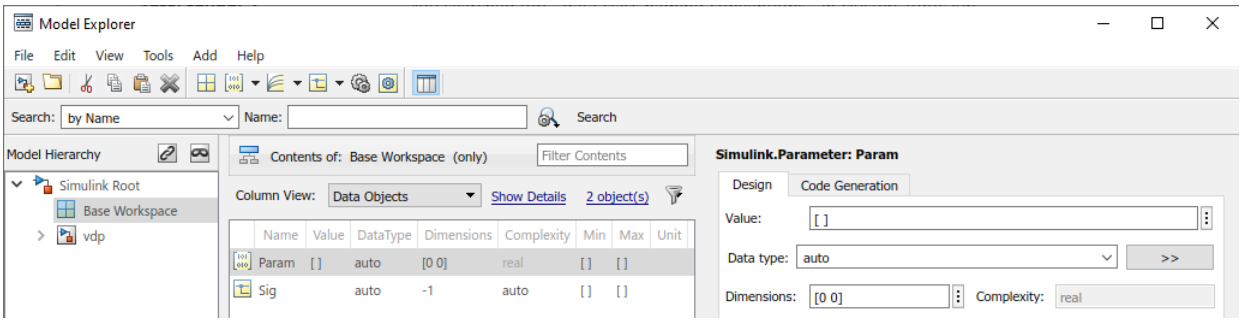
You can use either the Model Explorer or MATLAB commands to change a data object's properties.

For a list of signal object properties, see `Simulink.Signal`. For a list of parameter object properties, see `Simulink.Parameter`.

Use the Model Explorer to Change an Object's Properties

To use the Model Explorer to change an object's properties, select the workspace that contains the object in the Model Explorer's **Model Hierarchy** pane. Then select the object in the Model Explorer's **Contents** pane.

The Model Explorer displays the object's property dialog box in its **Dialog** pane (if the pane is visible).



You can configure the Model Explorer to display some or all of the properties of an object in the **Contents** pane (see **Model Explorer**). To edit a property, click its value in the **Contents** or **Dialog** pane. The value is replaced by a control that allows you to change the value.

Use MATLAB Commands to Change an Object's Properties

You can also use MATLAB commands to get and set data object properties. Use the following dot notation in MATLAB commands and programs to get and set a data object's properties:

```
value = obj.property;
obj.property = value;
```

where *obj* is a variable that references either the object if it is an instance of a value class or a handle to the object if the object is an instance of a handle class (see “Handle Versus Value Classes” on page 67-70), PROPERTY is the property's name, and VALUE is the property's value. For example, the following MATLAB code creates a data type alias object (i.e., an instance of `Simulink.AliasType`) and sets its base type to `uint8`:

```
gain = Simulink.AliasType;
gain.BaseType = 'uint8';
```

You can use dot notation recursively to get and set the properties of objects that are values of other object's properties, e.g.,

```
gain.CoderInfo.StorageClass = 'ExportedGlobal';
```

Create Data Objects from Built-In Data Class Package Simulink

The built-in package `Simulink` defines two data object classes `Simulink.Parameter` and `Simulink.Signal`. You can create these data objects using the user interface or programmatically.

Create Data Objects

1. In the Model Explorer **Model Hierarchy** pane, select a workspace to contain the data objects. For example, click `Base Workspace`.
2. On the toolbar, click the arrow next to **Add Parameter** or **Add Signal** . From the drop-down list, select **Simulink Parameter** or **Simulink Signal**.

   A parameter or signal object appears in the base workspace. The default name for new parameter objects is `Param`. The default name for new signal objects is `Sig`.
3. To create more objects, click **Add Parameter** or **Add Signal**.

To create `Simulink.LookupTable` and `Simulink.Breakpoint` objects, on the Model Explorer toolbar, use the button.

Programmatically Create Data Objects

```
% Create a Simulink.Parameter object named myParam whose value is 15.23.
myParam = Simulink.Parameter(15.23);

% Create a Simulink.Signal object named mySig.
mySig = Simulink.Signal;
```

Convert Numeric Variable into Parameter Object

You can convert a numeric variable into a `Simulink.Parameter` object as follows.

```
/* Define numeric variable in base workspace
myVar = 5;
/* Create data object and assign variable value
myObject = Simulink.Parameter(myVar);
```

Create Data Objects from Another Data Class Package

You can create your own package to define custom data object classes that subclass `Simulink.Parameter` and `Simulink.Signal`. You can use this technique to add your own properties and methods to data objects. If you have an Embedded Coder license, you can define storage classes and memory sections in the package. For more information about creating a data class package, see “Define Data Classes” on page 67-96.

Create Data Objects from Another Package

Suppose that you define a data class package called `myPackage`. Before you can create data objects from the package, you must include the folder containing the package folder on your MATLAB path.

1. In the Model Explorer **Model Hierarchy** pane, select a workspace to contain the data objects. For example, click `Base Workspace`.
2. Click the arrow next to **Add Parameter**  or **Add Signal**  and select **Customize class lists**.
3. In the dialog box, select the check box next to the class that you want. For example, select the check boxes next to `myPackage.Parameter` and `myPackage.Signal`. Click **OK**.
4. Click the arrow next to **Add Parameter** or **Add Signal**. Select the class for the data object that you want to create. For example, select **myPackage Parameter** or **myPackage Signal**.

   A parameter or signal object appears in the base workspace. The default name for new parameter objects is `Param`. The default name for new signal objects is `Sig`.
5. To create more data objects from the package `myPackage`, click **Add Parameter** or **Add Signal** again.

Programmatically Create Data Objects from Another Package

Suppose that you define a data class package called `myPackage`. Before you can create data objects from the package, you must include the folder containing the package folder on your MATLAB path.

```
% Create a myPackage.Parameter object named
% myParam whose value is 15.23.
myParam = myPackage.Parameter(15.23);

% Create a myPackage.Signal object named mySig.
mySig = myPackage.Signal;
```

Create Data Objects Directly from Dialog Boxes

When you open a Signal Properties dialog box, a block dialog box, or the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**), you can efficiently create a signal or parameter data object in a workspace or data dictionary.

Create Parameter Object from Block Dialog Box

1. In a numeric block parameter in the dialog box, specify the name that you want for the data object. For example, specify the name `myParam`.
2. Click the button ⋮ next to the value of the block parameter. Select **Create**.
3. In the **Create New Data** dialog box, specify **Value** as `Simulink.Parameter`.

   Alternatively, you can specify the name of a data class that you created, such as `myPackage.Parameter`. You can also use the drop-down list to select from a list of available data object classes.
4. Specify **Location** as `Base Workspace` and click **Create**.

   You can use the **Location** option to select a workspace to contain the new data object. If a model is linked to a data dictionary, you can choose to create a data object in the dictionary.
5. In the dialog box that opens, configure the data object properties. Specify a numeric value for the parameter in the **Value** box. Click **OK**.

   The parameter object `myParam` appears in the base workspace.
6. In the block parameter dialog box, click **OK**.

Create Signal Object from Signal Properties Dialog Box

1. In the **Signal name** box, specify a signal name such as `mySig`. Click **Apply**.
2. Click the button ⋮ next to the value of **Signal name**. Select **Create and Resolve**.
3. In the **Create New Data** dialog box, specify **Value** as `Simulink.Signal`.

   Alternatively, you can specify the name of a data class that you created, such as `myPackage.Signal`. Also, from the drop-down list, you can select a data object class that exists on the MATLAB path.
4. Specify **Location** as `Base Workspace` and click **Create**.

   You can use the **Location** option to select a workspace to contain the new data object. If a model is linked to a data dictionary, you can choose to create a data object in the dictionary.
5. In the dialog box that opens, configure the data object properties and click **OK**.

   The signal object `mySig` appears in the base workspace. In the Signal Properties dialog box, the **Signal name must resolve to Simulink signal object** property is selected.

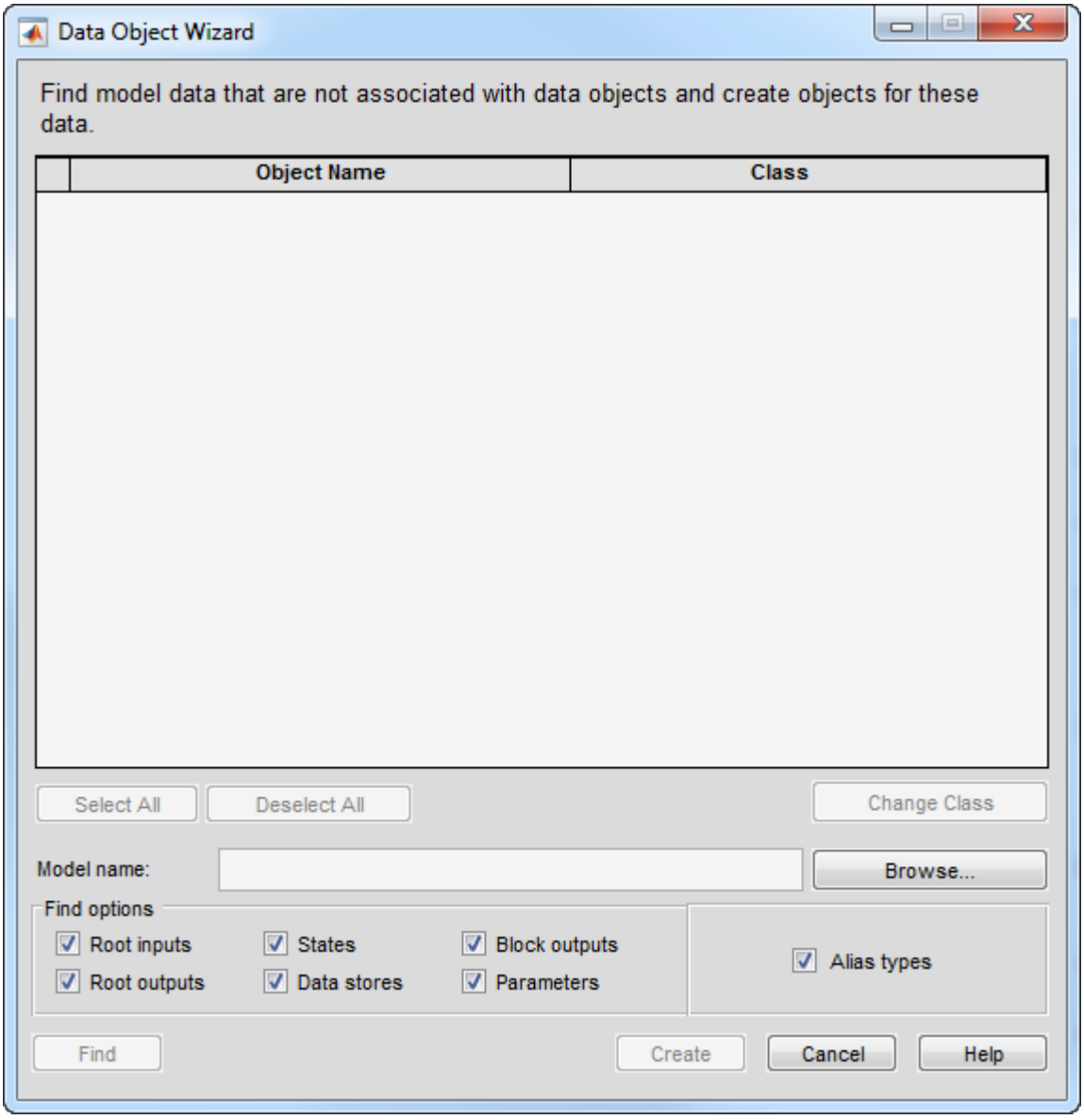
Create Data Objects for a Model Using Data Object Wizard

To create data objects that represent signals, parameters, and states in a model, you can use the Data Object Wizard. The wizard finds data in the model that do not have corresponding data objects.

Based on specifications in the model, the wizard creates the objects and assigns these characteristics:

- Signal, parameter, or state name.
- Numeric value for parameter objects.
- Data type. For signal objects, includes alias types such as `Sumlink.AliasType` and `Simulink.NumericType`.

1 In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Data Object Wizard**.



2 In the **Model name** box, enter the name of the model that you want to search.

By default, the box contains the name of the model from which you opened the wizard.

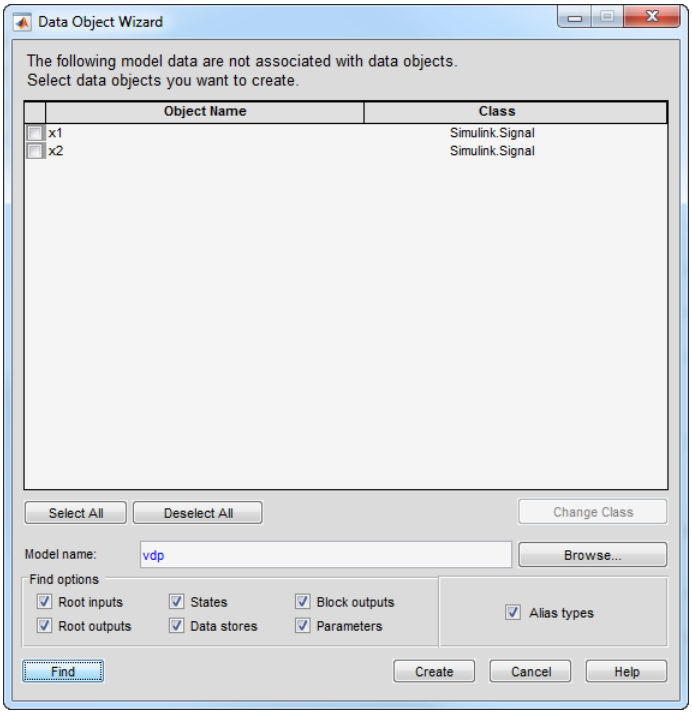
3 Under **Find options**, select the check boxes next to the data object types that you want to create. The table describes the options.

Option	Description
Root inputs	Named signals from root-level Inport blocks.
Root outputs	Named signals from root-level Outport blocks.

Option	Description
States	States associated with these discrete blocks: Discrete Filter Discrete State-Space Discrete-Time Integrator Discrete Transfer Fcn Discrete Zero-Pole Memory Discrete-Time PID Controller Discrete-Time PID Controller (2DOF) Unit Delay
Data stores	Data stores. For more information about data stores, see “Local and Global Data Stores” on page 73-2 .
Block outputs	Named signals whose sources are non-root-level blocks.
Parameters	• Numeric parameters, for example the parameters in these blocks: Constant Gain Relay • Stateflow data with **Scope** set to `Parameter`.
Alias types	Data type replacement names that you specify in **Configuration Parameters > Code Generation > Data Type Replacement**. If you have an Embedded Coder license, the Data Object Wizard creates `Simulink.AliasType` objects for these data type replacement names. For more information about data type replacement, see “Model Configuration Parameters: Code Generation Data Type Replacement” (Embedded Coder)

4 Click **Find**.

The data object table displays the proposed objects.



5 (Optional) To create objects from a data class other than the default classes, select the check box next to the objects whose class you want to change. To select all of the objects, click **Select All**. Click **Change Class**. In the dialog box that opens, select classes by using the drop-down lists next to **Parameter** and **Signal**.

If the classes that you want do not appear in the drop-down list, select `Customize class lists`. In the dialog box that opens, select the check box next to the classes that you want, and click **OK**.

To change the default parameter and signal classes that the wizard uses to propose objects:

- On the Model Explorer **Model Hierarchy** pane, select a workspace. For example, select **Base Workspace**.
- On the toolbar, click the arrow next to **Add Parameter** or **Add Signal** .
- From the drop-down list, select the class that you want the wizard to use. For example, select **myPackage Parameter** or **myPackage Signal**.

  A parameter or signal object appears in the selected workspace. The default name for new parameter objects is `Param`. The default name for new signal objects is `Sig`.

  The next time that you use the Data Object Wizard, the wizard proposes objects using the parameter or signal class that you selected in Model Explorer.

6 Select the check box next to the proposed objects that you want to create. To select all of the proposed objects, click **Select All**.

7 Click **Create**.

The data objects appear in the base workspace. If the target model is linked to a data dictionary, the objects appear in the dictionary.

The wizard changes settings in your model depending on the configuration parameter **Configuration Parameters > Diagnostics > Data Validity > Signal resolution**.

- If you set the parameter to `Explicit only`, the wizard forces the corresponding signals and states in your model to resolve to the new signal objects. The wizard selects the option **Signal name must resolve to Simulink signal object** in each Signal Properties dialog box and **State name must resolve to Simulink signal object** in each block dialog box.
- If you set the parameter to `Explicit and implicit` or `Explicit and warn implicit`, the wizard does not change the setting of **Signal name must resolve to Simulink signal object** or **State name must resolve to Simulink signal object** for any signals or states.

Consider turning off implicit signal object resolution for your model by using the function `disableimplicitsignalresolution`. For more information, see “Explicit and Implicit Symbol Resolution” on page 67-129.

Data Object Wizard Troubleshooting

- The Data Object Wizard compiles models for code generation in order to propose creation of signal objects. Because of this, the wizard cannot be used for models that are not valid for code generation.
- The Data Object Wizard does not propose creation of data objects for these entities in a model:
  - Multiple separate signals that have the same name.
  - A signal with the same name as a variable used in a block parameter.
  - A signal that lacks a single contiguous source block.
  - A signal whose source block is commented out or commented through.
  - Data items that are rendered inactive by Variant Source and Variant Sink blocks. The wizard proposes objects only for data items that are associated with active blocks.
  - Signals and states when you set the model configuration parameter **Signal resolution** to `None`.

Create Data Objects from External Data Source Programmatically

This example shows how to create data objects based on an external data source (such as a Microsoft Excel file) by using a script.

1. Create a new MATLAB script file.
2. Place information in the file that describes the data in the external file that you want to convert to data objects. For example, the following information creates two `Simulink` data objects with the indicated properties. The first is for a parameter and the second is for a signal:

```
% Parameters
ParCon = Simulink.Parameter;
ParCon.CoderInfo.StorageClass = 'Custom'
ParCon.CoderInfo.CustomStorageClass ='Const';
ParCon.Value = 3;
% Signals
SigGlb = Simulink.Signal;
SigGlb.DataType = 'int8';
```

3. Run the script file. The data objects appear in the MATLAB workspace.

If you want to import the target data from the external source, you can write MATLAB functions and scripts that read the information, convert the information to data objects, and load the objects into the base workspace.

You can use these functions to interact with files that are external to MATLAB:

- `xmlread`
- `xmlwrite`
- `xlsread`
- `xlswrite`
- `csvread`
- `csvwrite`
- `dlmread`
- `dlmwrite`

Data Object Methods

Data classes define functions, called methods, for creating and manipulating the objects that they define. A class may define any of the following kinds of methods.

Dynamic Methods

A dynamic method is a method whose identity depends on its name and the class of an object specified implicitly or explicitly as its first argument. You can use either function or dot notation to specify this object, which must be an instance of the class that defines the method or an instance of a subclass of the class that defines the method. For example, suppose class `A` defines a method called `setName` that assigns a name to an instance of `A`. Further, suppose the MATLAB workspace contains an instance of A assigned to the variable `obj`. Then, you can use either of the following statements to assign the name `'foo'` to `obj`:

```
obj.setName('foo');
setName(obj, 'foo');
```

A class may define a set of methods having the same name as a method defined by one of its super classes. In this case, the method defined by the subclass overrides the behavior of the method defined by the parent class. Simulink determines which method to invoke at runtime from the class of the object that you specify as its first or implicit argument. Hence, the term dynamic method.

Note Most Simulink data object methods are dynamic methods. Unless the documentation for a method specifies otherwise, you can assume that a method is a dynamic method.

Static Methods

A static method is a method whose identity depends only on its name and hence cannot change at runtime. To invoke a static method, use its fully qualified name, which includes the name of the class that defines it followed by the name of the method itself. For example, `Simulink.ModelAdvisor` class defines a static method named `getModelAdvisor`. The fully qualified name of this static method is `Simulink.ModelAdvisor.getModelAdvisor`. The following example illustrates invocation of a static method.

```
ma = Simulink.ModelAdvisor.getModelAdvisor('vdp');
```

Constructors

Every data class defines a method for creating instances of that class. The name of the method is the same as the name of the class. For example, the name of the `Simulink.Parameter` class's constructor is `Simulink.Parameter`. The constructors defined by Simulink data classes take no arguments.

The value returned by a constructor depends on whether its class is a handle class or a value class. The constructor for a handle class returns a handle to the instance that it creates if the class of the instance is a handle class; otherwise, it returns the instance itself (see “Handle Versus Value Classes” on page 67-70).

Handle Versus Value Classes

Simulink classes, including data object classes, fall into two categories: value classes and handle classes.

About Value Classes

The constructor for a *value* class (see “Constructors” on page 67-70) returns an instance of the class and the instance is permanently associated with the MATLAB variable to which it is initially assigned. Reassigning or passing the variable to a function causes MATLAB to create and assign or pass a copy of the original object.

For example, `Simulink.NumericType` is a value class. Executing the following statements

```
x = Simulink.NumericType;
y = x;
```

creates two instances of class `Simulink.NumericType` in the workspace, one assigned to the variable `x` and the other to `y`.

About Handle Classes

The constructor for a *handle* class returns a handle object. The handle can be assigned to multiple variables or passed to functions without causing a copy of the original object to be created. For example, `Simulink.Parameter` class is a handle class. Executing

```
x = Simulink.Parameter;
y = x;
```

creates only one instance of `Simulink.Parameter` class in the MATLAB workspace. Variables x and y both refer to the instance via its handle.

A program can modify an instance of a handle class by modifying any variable that references it, e.g., continuing the previous example,

```
x.Description = 'input gain';
y.Description

ans =
input gain
```

Most Simulink data object classes are value classes. Exceptions include `Simulink.Signal` and `Simulink.Parameter` class.

To determine whether the value of a variable is an object (value class) or a handle to an object, see "Determine If an Object Is a Handle".

Copy Handle Objects

Use the copy method of a handle object to create copies of instances of that object. For example, `ConfigSet` is a handle object that represents model configuration sets. The following code creates a copy of the current model's active configuration set and attaches it to the model as an alternate configuration geared to model development.

```
activeConfig = getActiveConfigSet(gcs);
develConfig = copy(activeConfig);
develConfig.Name = 'develConfig';
attachConfigSet(gcs, develConfig);
```

Compare Data Objects

Simulink data objects provide a method, named `isequal`, that determines whether object property values are equal. This method compares the property values of one object with those belonging to another object and returns true (`1`) if all of the values are the same or false (`0`) otherwise. For example, the following code instantiates two signal objects (A and B) and specifies values for particular properties.

```
A = Simulink.Signal;
B = Simulink.Signal;
A.DataType = 'int8';
B.DataType = 'int8';
A.InitialValue = '1.5';
B.InitialValue = '1.5';
```

Afterward, use the `isequal` method to verify that the object properties of A and B are equal.

```
result = isequal(A,B)
```

```
result =

     1
```

Resolve Conflicts in Configuration of Signal Objects for Code Generation

If a signal is defined in the Signal Properties dialog box and a signal object of the same name is defined by using the command line or in the Model Explorer, the potential exists for ambiguity when the Simulink engine attempts to resolve the symbol representing the signal name. One way to resolve the ambiguity is to specify that a signal resolve to a `Simulink.Signal` data object. Select the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box.

To configure the signal data, use the Code Mappings editor or code mappings API to add the signal to the model code mappings and set the storage class and storage class properties. For Simulink Coder, see "Configure Signal Data for C Code Generation" (Simulink Coder). For Embedded Coder, see "Configure Signal Data for C Code Generation" (Embedded Coder).

Create Persistent Data Objects

To preserve data objects so that they persist when you close MATLAB, you can:

- Store the objects in a data dictionary or model workspace. To decide where to permanently store model data, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.
- Use the `save` command to save data objects in a MAT-file and the `load` command to restore them to the MATLAB base workspace in the same or a later session. Configure the model to load the objects from the MAT-file or a script file when the model loads.

To load data objects from a file when you load a model, write a script that creates the objects and configures their properties. Alternatively, save the objects in a MAT-file. Then use either the script or a load command as the `PreLoadFcn` callback routine for the model that uses the objects. Suppose that you save the data objects in a file named `data_objects.mat`, and the model to which they apply is open and active. At the command prompt, entering:

```
set_param(gcs, 'PreLoadFcn', 'load data_objects');
```

sets `load data_objects` as the model's preload function. Whenever you open the model, the data objects appear in the base workspace.

Definitions of the classes of saved objects must exist on the MATLAB path for them to be restored. If the class of a saved object acquires new properties after the object is saved, Simulink adds the new properties to the restored version of the object. If the class loses properties after the object is saved, only the properties that remain are restored.

See Also

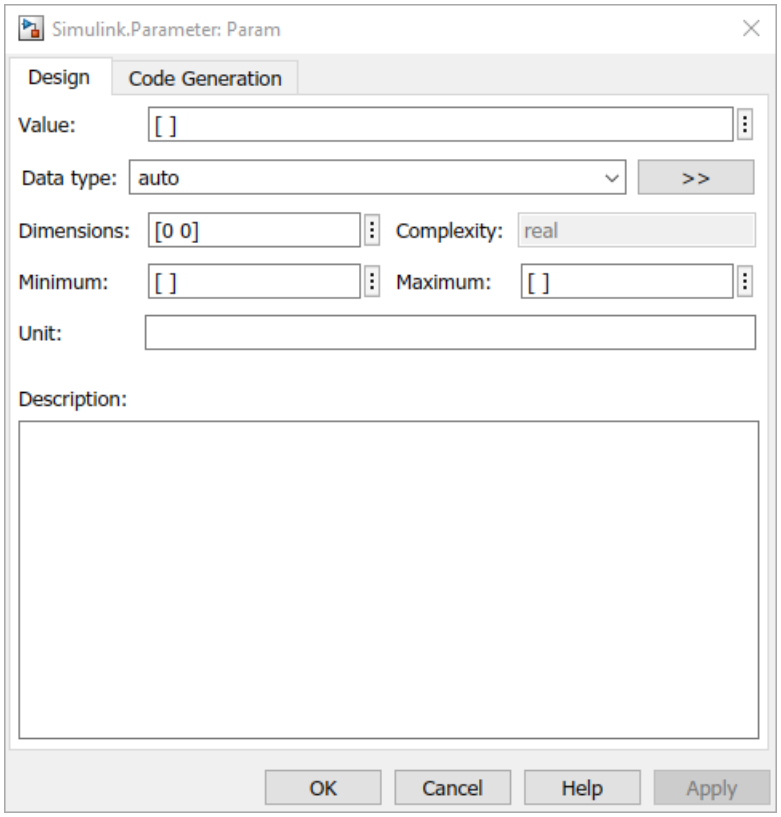
`Simulink.Breakpoint` | `Simulink.LookupTable` | `Simulink.Parameter` | `Simulink.Signal` | `disableimplicitsignalresolution`

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Create, Edit, and Manage Workspace Variables” on page 67-106
- “Define Data Classes” on page 67-96
- “Use Simulink.Signal Objects to Specify and Control Signal Attributes” on page 67-89
- “What Is a Data Dictionary?” on page 74-2
- “Configure Generated Code According to Interface Control Document” (Embedded Coder)
- “Symbol Resolution” on page 67-127

Simulink.Parameter Property Dialog Box

Create a `Simulink.Parameter` object to set the value of one or more block parameters in a model, such as the **Gain** parameter of a Gain block. For examples and programmatic information, see `Simulink.Parameter`.



Value

Ideal real-world value that the object stores. Block parameters that refer to the object use the value that you specify.

You can also use MATLAB syntax to specify the value.

Example Expression	Description
`15.23`	Specifies a scalar value
`[3 4; 9 8]`	Specifies a matrix
`3+2i`	Specifies a complex value
`struct('A',20,'B',5)`	Specifies a structure with two fields, `A` and `B`, with double-precision values `20` and `5`. Organize block parameters into structures (see “Organize Related Block Parameter Definitions in Structures” on page 37-19) or initialize the signal elements in a bus (see “Specify Initial Conditions for Bus Signals” on page 76-57).

Example Expression	Description
`=myVar + myOtherVar`	Specifies the expression `myVar + myOtherVar` where `myVar` and `myOtherVar` are other MATLAB variables or parameter objects. Simulink Coder preserves this mathematical relationship between the object and the variables.

To use a `Simulink.Parameter` object to store a value of a particular numeric data type, specify the ideal value with the **Value** property and control the type with the **Data type** property.

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 to avoid accumulating numerical error through repeated quantizations or data type saturation, especially for fixed-point data types.

When you specify an array with three or more dimensions, the **Value** property displays the array as an expression that contains a call to the `reshape` function. To edit the values in the array, modify the first argument of the `reshape` call, which contains all of the array values in a serialized vector. When you add or remove elements along a dimension, you must also correct the argument that represents the length of the modified dimension.

To more easily edit a large vector, 2-D matrix, or structure that you store in a `Simulink.Parameter` object, consider using the Variable Editor. See “Manage and Edit Workspace Variables” on page 37-14.

Data type

Data type of the parameter value in the `Value` property, specified as `'auto'` or a character vector. When you simulate the model or generate code, Simulink casts the value to the specified data type.

If you select `auto`, the default setting, the parameter object uses the same data type as the block parameters that use the object. See “Reduce Maintenance Effort with Data Type Inheritance” on page 37-44.

When you set the **Value** property to something other than a `double` number, the object typically sets the **Data type** property based on the value of the **Value** property. For example, when you set the **Value** property to `int8(5)`, the object sets the value of the **Data type** property to `int8`.

You can select a data type from the drop-down list or specify the name of a data type with text.

To explicitly specify a built-in data type (see “Data Types Supported by Simulink” on page 67-4), use one of these options:

- `double`
- `single`
- `half`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`

- `uint32`
- `boolean`

To specify a fixed-point data type, use the `fixdt` function. For example, specify `fixdt(1,16,5)`.

If you use a `Simulink.AliasType` or `Simulink.NumericType` object to create and share custom data types in your model, specify the name of the object.

To specify an enumerated data type, use the name of the type preceded by `Enum:`. For example, specify `Enum: myEnumType`.

When you store a structure or array of structures in the **Value** property of the object, the object sets the **Data type** property to `struct`. To specify a `Simulink.Bus` object as the data type, use the name of the bus object preceded by `Bus:`. For example, specify `Bus: myBusObject`.

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” on page 67-30.

Dimensions

Dimensions of the parameter value.

When you set the **Value** property of the object, the object sets the value of the **Dimensions** property to a `double` row vector. The vector is the same vector that the `size` function returns.

To use symbolic dimensions, see “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Complexity

Numeric complexity of the parameter value. Simulink determines the complexity from the parameter value that you specify in the **Value** property. This property is read only.

Minimum

Minimum value that the parameter can have. The default value is empty, which means the parameter value does not have a minimum. Specify a real `double` scalar.

If you store a complex number in the **Value** property, the **Minimum** property applies separately to the real and imaginary parts.

If you store a structure in the **Value** property, the object ignores the **Minimum** property. Instead, use a `Simulink.Bus` object as the data type of the parameter object and specify a minimum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21.

If **Value** is less than the minimum value or if the minimum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters” on page 37-52.

Maximum

Maximum value that the parameter can have. The default value is empty, which means the parameter value does not have a maximum. Specify a real `double` scalar.

If you store a complex number in the **Value** property, the **Maximum** property applies separately to the real and imaginary parts.

If you store a structure in the **Value** property, the object ignores the **Maximum** property. Instead, use a `Simulink.Bus` object as the data type of the parameter object and specify a maximum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object” on page 37-21.

If **Value** is greater than the maximum value or if the maximum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters” on page 37-52.

Stored Integer Minimum

For parameter objects with a fixed-point data type, the minimum value that the parameter can have, specified as a stored integer value. The value is derived from the real-world value **Minimum**. This property is available only in the property dialog box.

Stored Integer Maximum

For parameter objects with a fixed-point data type, the maximum value that the parameter can have, specified as a stored integer value. The value is derived from the real-world value **Maximum**. This property is available only in the property dialog box.

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” on page 9-2.

Storage class

Storage class of this parameter object. Simulink code generation toolboxes use this property to allocate memory for this parameter object in the generated code.

For more information, see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder) and “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Identifier

Alternative name for this parameter in the generated code.

Alignment

Data alignment boundary for code generation, specified in number of bytes. The starting memory address for the data allocated for the parameter is a multiple of the **Alignment** setting. The default value is `-1`, which specifies that the code generator determine an optimal alignment based on usage. Otherwise, specify a positive integer that is a power of 2, not exceeding 128. For more information, see “Data Alignment for Code Replacement” (Embedded Coder).

Argument

Specification to configure the parameter object as a model argument (see “Parameterize Instances of a Reusable Referenced Model” on page 8-64). This property appears only if the parameter object is in a model workspace.

Description

Custom description of this parameter object. Use this property to document the significance that the parameter object has in your algorithm.

If you have an Embedded Coder license, you can configure this description to appear in the generated code as a comment. See “Simulink data object descriptions” (Embedded Coder).

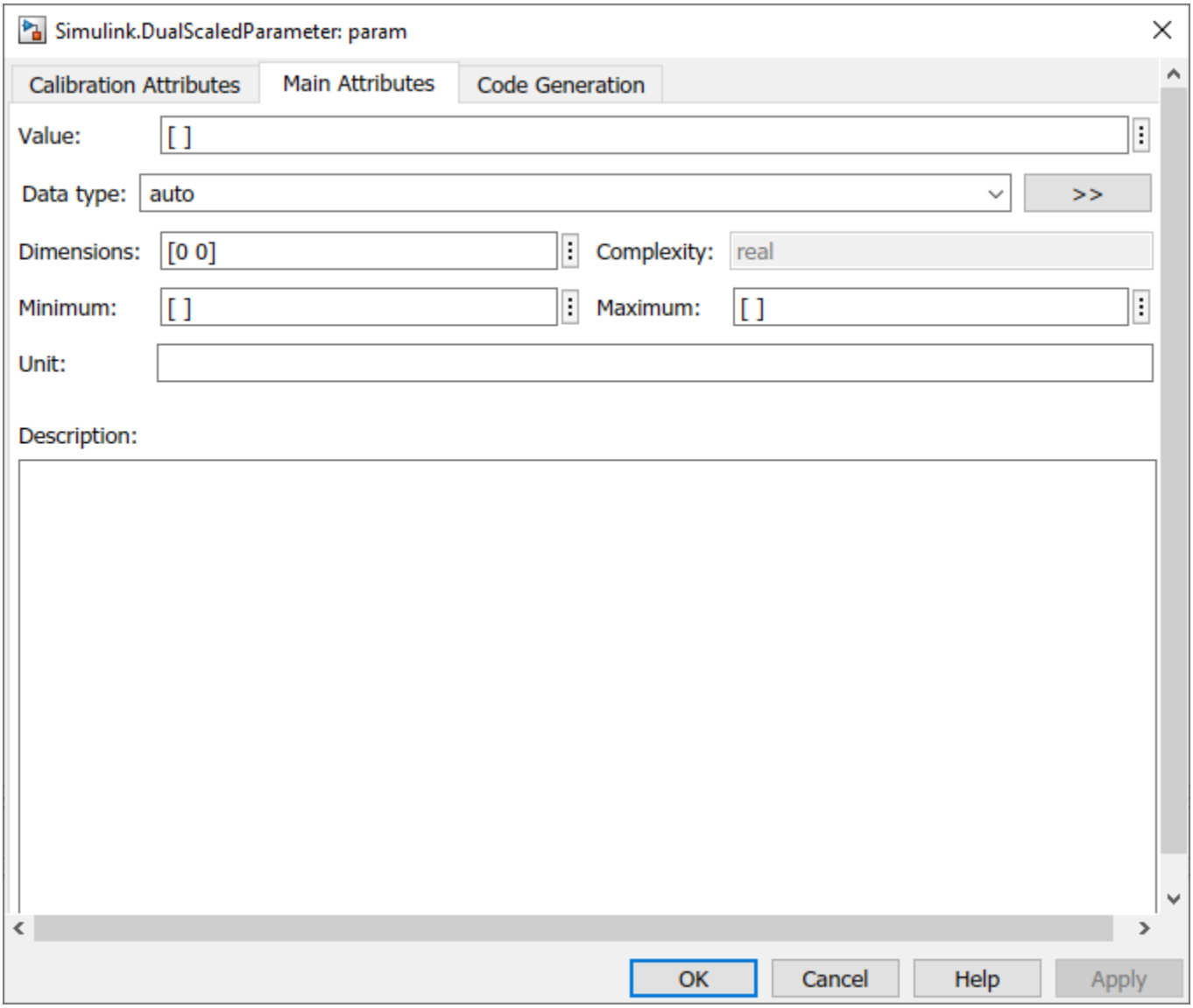
See Also

`Simulink.Parameter`

Simulink.DualScaledParameter Property Dialog Box

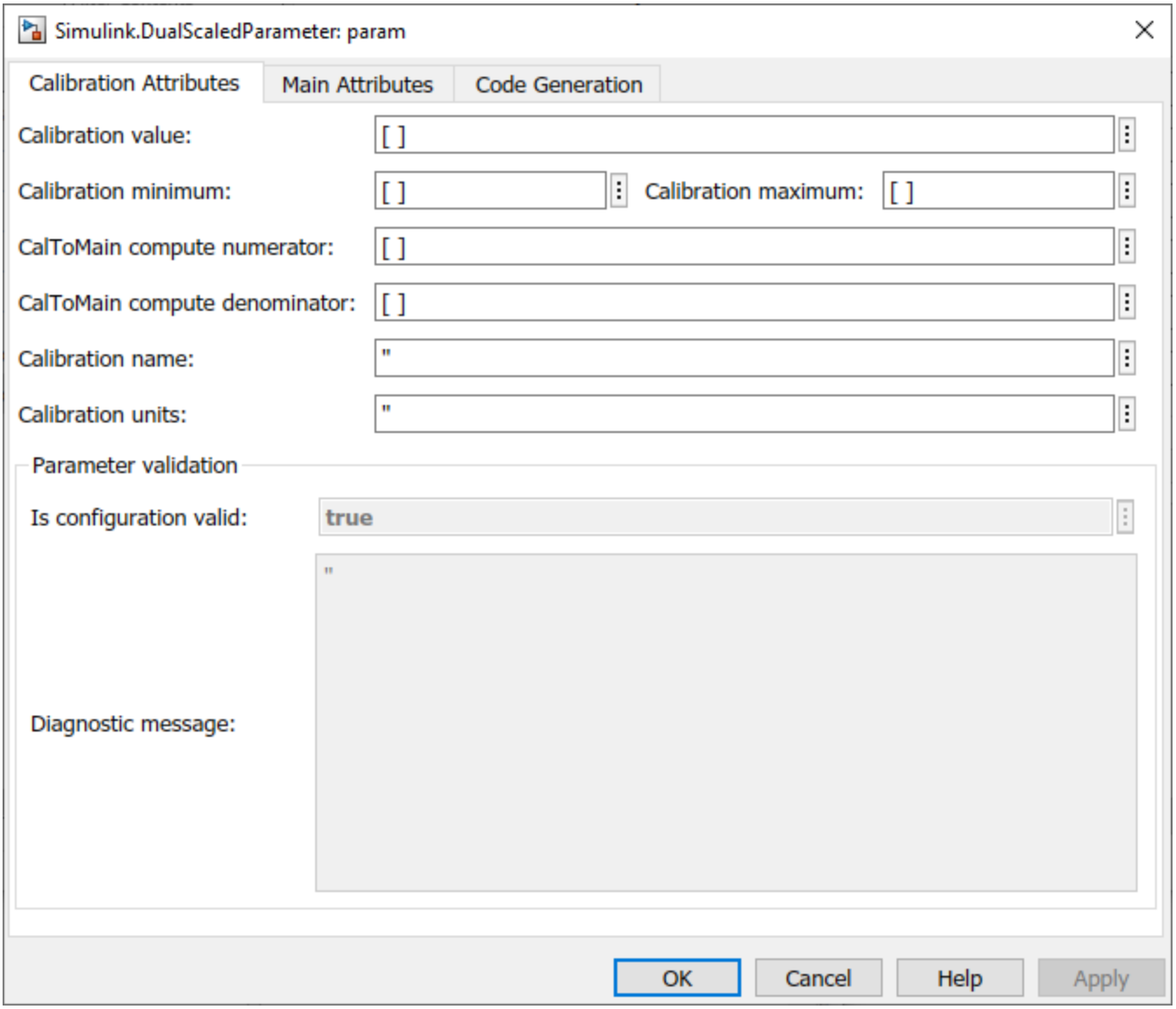
Create a `Simulink.DualScaledParameter` object so that you can store two scaled values of the same physical parameter value, for example, for the **Gain** parameter of a Gain block. For examples and programmatic information, see `Simulink.DualScaledParameter`.

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 computation method. The parameter uses the computation method and the calibration value to calculate the main value that Simulink uses.

Calibration minimum

Minimum value for the calibration parameter. The default value is `[ ]` (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration minimum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computation method. The parameter uses

the computation method and the calibration minimum value to calculate the minimum or maximum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration minimum sets the main minimum value. If it is decreasing, setting the calibration minimum sets the main maximum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Calibration maximum

Maximum value for the calibration parameter. The default value is `[]` (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration maximum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computation method. The parameter uses the computation method and the calibration maximum value to calculate the corresponding maximum or minimum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration maximum sets the main maximum value. If it is decreasing, setting the calibration maximum sets the main minimum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

CalToMain compute numerator

Specify the numerator coefficients `a` and `b` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real, double scalar values for `a` and `b`. For example, `[1 1]` or, for reciprocal scaling, `1`.

Once you have applied **CalToMain compute numerator**, you cannot change it.

CalToMain compute denominator

Specify the denominator coefficients `c` and `d` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real, double scalar values for `c` and `d`. For example, `[1 1]`.

Once you have applied **CalToMain compute denominator**, you cannot change it.

Calibration name

Specify the name of the calibration parameter. The default value is `''`. Specify a character vector value, for example, `'T1'`.

Calibration units

Specify the measurement units for this calibration value. This field is intended for use in documenting this parameter. The default value is `''`. Specify a character vector value, for example, `'Fahrenheit'`.

Is configuration valid

Simulink indicates whether the configuration is valid. The default value is `true`. If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the **Diagnostic message** field. You cannot set this field.

Diagnostic message

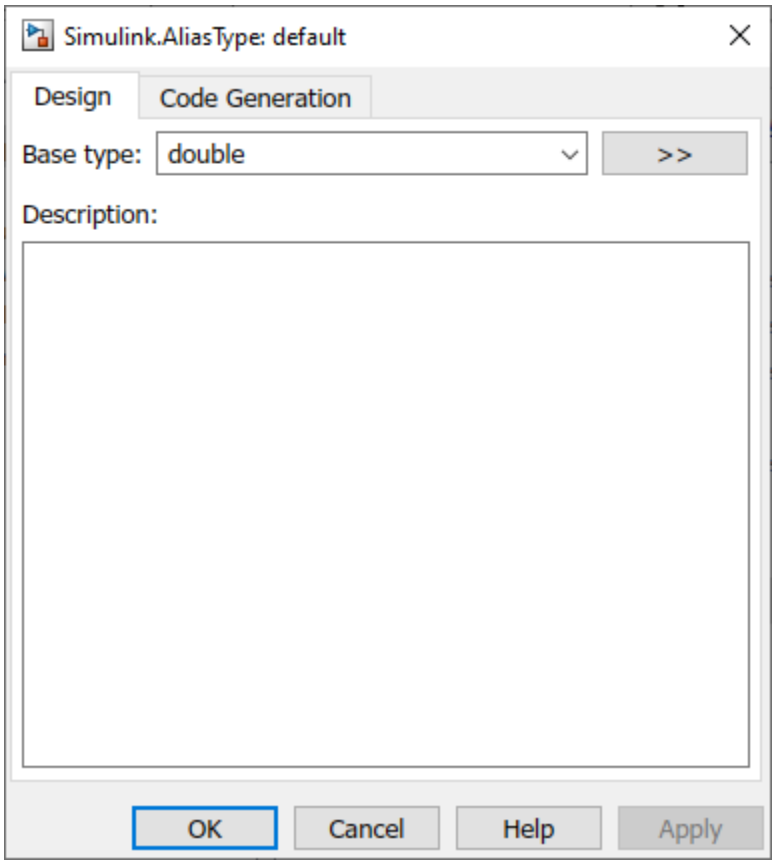
If you specify invalid parameter settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue. You cannot set this field.

See Also

`Simulink.DualScaledParameter`

Simulink.AliasType Property Dialog Box

Use a `Simulink.AliasType` object to rename data types for signal, state, and parameter data in a model. For examples and programmatic information, see `Simulink.AliasType`.



Base type

The data type to which this alias refers. The default is `double`. To specify another data type, such as `half`, select the data type from the adjacent drop–down list of standard data types or enter the data type name in the edit field.

To specify a fixed-point data type, you can use a call to the `fixdt` function, such as `fixdt(0,16,7)`. To specify the characteristics of the type interactively, expand the Data Type Assistant and set **Mode** to `Fixed point`. For information about using the Data Type Assistant, see “Specify Data Types Using Data Type Assistant” on page 67-30.

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 `DataTypeMode` is `Fixed-point: unspecified scaling`.

Note `Fixed-point: unspecified scaling` is a partially specified type whose definition is completed by the block that uses the `Simulink.NumericType`. Forbidding its use in alias types avoids creating aliases that have different base types depending on where they are used.

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

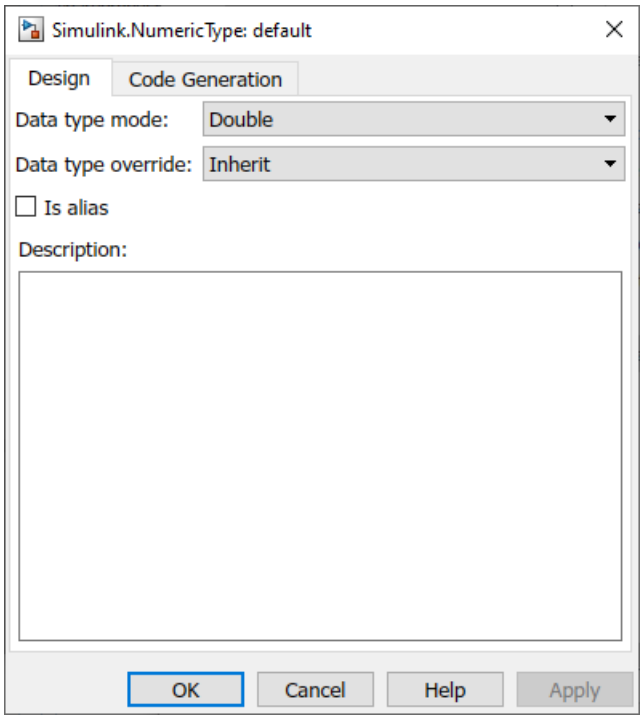
Describes the usage of the data type referenced by this alias.

See Also

`Simulink.AliasType`

Simulink.NumericType Property Dialog Box

Use a `Simulink.AliasType` object to set and share data types for signal, state, and parameter data in a model. For examples and programmatic information, see `Simulink.NumericType`.



Data type mode

Data type of this numeric type. The options are listed in this table.

Option	Description
`Double`	Same as the MATLAB `double` type.
`Single`	Same as the MATLAB `single` type.
`Boolean`	Same as the MATLAB `boolean` type.
`Fixed-point: unspecified scaling`	A fixed-point data type with unspecified scaling.
`Fixed-point: binary point scaling`	A fixed-point data type with binary-point scaling.
`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. 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`.

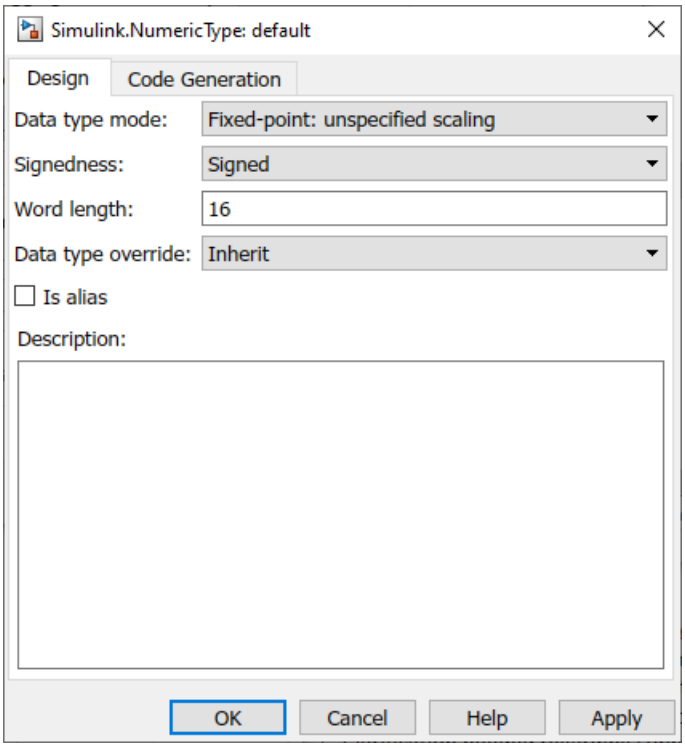
By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

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.

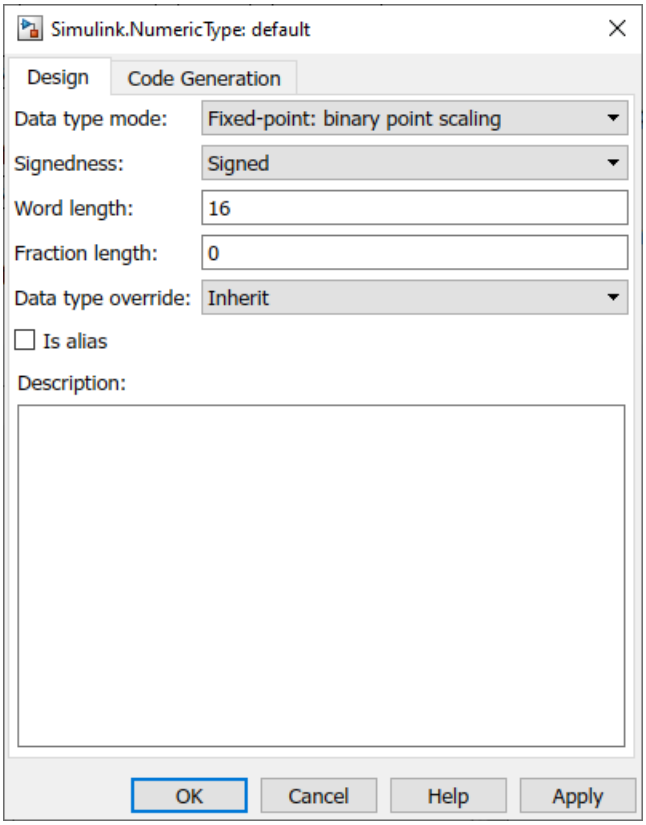


Word length

Word length in bits of the fixed-point data type. This option is enabled only for fixed-point data type modes.

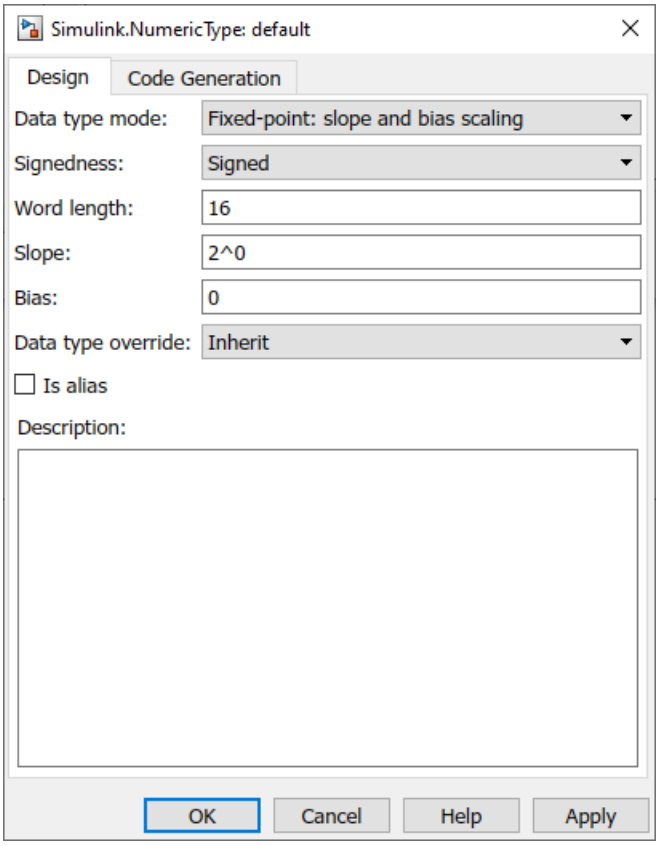
Fraction length

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.

See Also

`Simulink.NumericType`

Use Simulink.Signal Objects to Specify and Control Signal Attributes

A `Simulink.Signal` object enables you to assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on. For programmatic and reference information, see `Simulink.Signal`.

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 67-92 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” on page 67-129 for more information.

     When you use this technique, set **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** to a value other than `None`. To use only explicit resolution (a best practice), set the parameter to `Explicit only`.
   - **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” on page 75-38, “Using Signal Objects to Tune Initial Values” on page 75-40, and “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder).

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 “Specify Signal Ranges” on page 75-31 and “Model Configuration Parameters: Data Validity Diagnostics” 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 `Reusable`. If the storage class 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 `Reusable`, 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 “Highlight Signal Sources and Destinations” on page 75-25 for a description of techniques that you can use to trace the full extent of a signal.

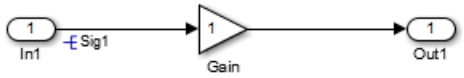
Signal Specification Block: An Alternative to Simulink.Signal

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:

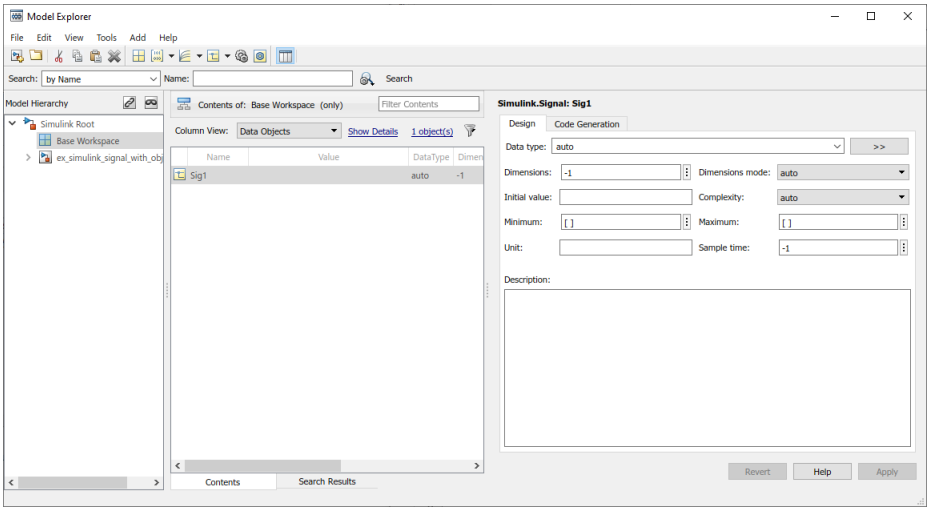
- Using a signal object simplifies the model and allows you to change signal property values without editing the model, but does not show signal property values directly in the block diagram.
- Using a Signal Specification block displays signal property values directly in the block diagram, but complicates the model and requires editing it to change signal property values.

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.

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.

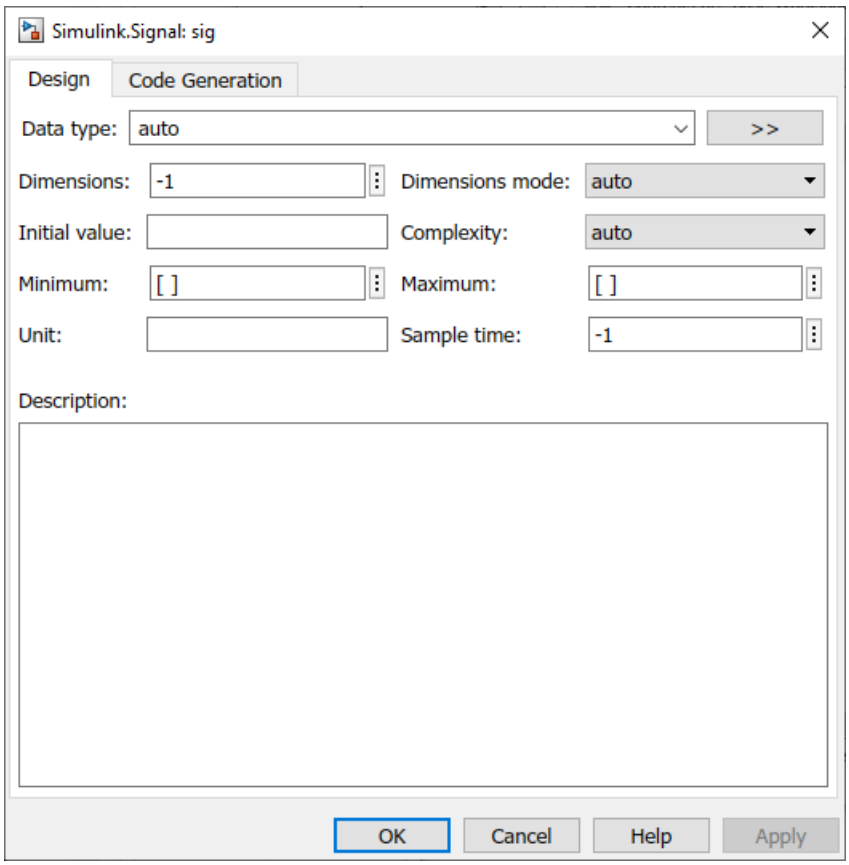
The structure you specify must contain a value for every element of the bus represented by the bus object.

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.

Property Dialog Box

For examples and programmatic information about `Simulink.Signal`, see `Simulink.Signal`.



Data type

Data type of the signal. The default entry, `auto`, specifies that Simulink should determine the data type. Use the adjacent dropdown list to specify built-in data types (for example, `uint8`) or a data type such as `'half'`. 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 67-91 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” on page 67-30 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 `M`x`N` 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” on page 7-3 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”).

Stored Integer Minimum

For signal objects with a fixed-point data type, the minimum value that the signal should have, specified as a stored integer value. The value is derived from the real-world value **Minimum**. This property is available only in the property dialog box.

Stored Integer Maximum

For signal objects with a fixed-point data type, the maximum value that the signal should have, specified as a stored integer value. The value is derived from the real-world value **Maximum**. This property is available only in the property dialog box.

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 character vector.

```
mySigObject.InitialValue='5.3';
```

```
mySigObject.InitialValue = 'myNumericVariable';
```

To specify an initial value for a signal that uses a numeric data type other than `double`, cast the initial value to the signal data type. For example, you can specify `single(73.3)` to use `73.3` as the initial value for a signal of data type `single`.

If you use a bus object as the data type for the signal object, set **Initial value** to a character vector containing either `0` or a MATLAB structure that matches the bus object. See “Bus Support” on page 67-91 for details.

If the initial value evaluates to a MATLAB structure, then in the **Configuration Parameters** dialog box, set “Underspecified initialization detection” to `simplified`.

If necessary, Simulink converts the initial value to ensure type, complexity, and dimension consistency with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model. Also, Simulink performs range checking of the initial value. The software alerts you when the initial value of the signal lies outside a range that corresponds to its specified minimum and maximum values and data type.

Classic initialization mode: In this mode, initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as `[]`):

- Output signals of conditionally executed subsystems and Merge blocks
- Block states

Simplified initialization mode: In this mode, initial values of signal objects associated with the following blocks are ignored. The initial values of the corresponding blocks are used instead.

- Outport blocks of conditionally executed subsystems
- Merge blocks

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 matching units. For more information, see “Unit Specification in Simulink Models” on page 9-2.

Storage class

Storage class of this signal. For more information, see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder) and “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder).

If you create the signal object in a model workspace, you must set the object storage class to `Auto`.

Identifier

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” (Embedded Coder). 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” (Embedded Coder).

See Also

`Simulink.Signal`

Define Data Classes

This example shows how to subclass Simulink data classes.

Use MATLAB class syntax to create a data class in a package. Optionally, assign properties to the data class and define storage classes.

Use an example to define data classes

1. View the `+SimulinkDemos` data class package in the folder *`matlabroot`*`/toolbox/simulink/simdemos/dataclasses` (open).

   This package contains predefined data classes.
2. Copy the folder to the location where you want to define your data classes.
3. Rename the folder `+mypkg` and add its parent folder to the MATLAB path.
4. Modify the data class definitions.

Manually define data class

1. Create a package folder `+mypkg` and add its parent folder to the MATLAB path.
2. Create class folders `@Parameter` and `@Signal` inside `+mypkg`.

   Note Simulink requires data classes to be defined inside `+Package/@Class` folders.
3. In the `@Parameter` folder, create a MATLAB file `Parameter.m` and open it for editing.
4. Define a data class that is a subclass of `Simulink.Parameter` using MATLAB class syntax.

```
classdef Parameter < Simulink.Parameter

end % classdef
```

To use a custom class name other than `Parameter` or `Signal`, name the class folders using the custom name. For example, to define a class `mypkg.myParameter`:

- Define the data class as a subclass of `Simulink.Parameter` or `Simulink.Signal`.

```
classdef myParameter < Simulink.Parameter

end % classdef
```

- In the class definition, name the constructor method as `myParameter` or `mySignal`.
- Name the class folder, which contains the class definition, as `@myParameter` or `@mySignal`.

Optional: Add properties to data class

The `properties` and `end` keywords enclose a property definition block.

```
classdef Parameter < Simulink.Parameter
    properties % Unconstrained property type
        Prop1 = [];
    end

    properties(PropertyType = 'logical scalar')
        Prop2 = false;
```

```
    end

    properties(PropertyType = 'char')
        Prop3 = '';
    end

    properties(PropertyType = 'char',...
      AllowedValues = {'red'; 'green'; 'blue'})
        Prop4 = 'red';
    end
end % classdef
```

If you add properties to a subclass of `Simulink.Parameter`, `Simulink.Signal`, or `Simulink.CustomStorageClassAttributes`, you can specify the following property types.

Property Type	Syntax
Double number	`properties(PropertyType = 'double scalar')`
int32 number	`properties(PropertyType = 'int32 scalar')`
Logical number	`properties(PropertyType = 'logical scalar')`
Character vector (char)	`properties(PropertyType = 'char')`
Character vector with limited set of allowed values	`properties(PropertyType = 'char', AllowedValues = {'a', 'b', 'c'})`

If you use MATLAB property validation (see “Validate Property Values”) instead of `PropertyType`, the properties are displayed as an edit field in the property dialog box of the class. If you use `PropertyType` and `AllowedValues`, then the property dialog box displays:

- A check box for logical scalar properties.
- A dropdown menu for character vectors and `AllowedValues`.

Optional: Add initialization code to data class

You can add a constructor within your data class to perform initialization activities when the class is instantiated. The added constructor cannot require an input argument.

In this example, the constructor initializes the value of object `obj` based on an optional input argument.

```
classdef Parameter < Simulink.Parameter
    methods
        function obj = Parameter(optionalValue)
            if (nargin == 1)
                obj.Value = optionalValue;
            end
        end
    end % methods
end % classdef
```

Optional: Define storage classes

Use the `setupCoderInfo` method to configure the `CoderInfo` object of your class. Then, create a call to the `useLocalCustomStorageClasses` method and open the Custom Storage Class Designer.

1. In the constructor within your data class, call the `useLocalCustomStorageClasses` method.

```
classdef Parameter < Simulink.Parameter
    methods
        function setupCoderInfo(obj)
            useLocalCustomStorageClasses(obj, 'mypkg');

            obj.CoderInfo.StorageClass = 'Custom';
        end
    end % methods
end % classdef
```

2. Open the Custom Storage Class Designer for your package.

```
cscdesigner('mypkg')
```

3. Define storage classes.

Optional: Define custom attributes for storage classes

1. Create a MATLAB file `myCustomAttribs.m` and open it for editing. Save this file in the `+mypkg/@myCustomAttribs`folder, where `+mypkg` is the folder containing the `@Parameter` and `@Signal` folders.
2. Define a subclass of `Simulink.CustomStorageClassAttributes` using MATLAB class syntax. For example, consider a storage class that defines data using the original identifier but also provides an alternate name for the data in generated code.

```
classdef myCustomAttribs < Simulink.CustomStorageClassAttributes
    properties(PropertyType = 'char')
        AlternateName = '';
    end
end % classdef
```

3. Override the default implementation of the `isAddressable` method to determine whether the storage class is writable.

```
classdef myCustomAttribs < Simulink.CustomStorageClassAttributes
        properties(PropertyType = 'logical scalar')
            IsAlternateNameInstanceSpecific = true;
        end

        methods
            function retVal = isAddressable(hObj, hCSCDefn, hData)
                retVal = false;
            end
        end % methods
end % classdef
```

4. Override the default implementation of the `getInstanceSpecificProps` method.

   For examples, see `CSCTypeAttributes_FlatStructure.m` in the folder *`matlabroot`*`\toolbox\simulink\simulink\dataclasses\+Simulink\@CSCTypeAttributes_FlatStructure` (open) and `CSCTypeAttributes_Unstructed.m` in the folder *`matlabroot`*`\toolbox\simulink\simulink\dataclasses\+mpt\@CSCTypeAttributes_Unstructed` (open).

Note This is an optional step. By default, all custom attributes are instance-specific and are modifiable for each data object. However, you can limit which properties are allowed to be instance-specific.

5 Override the default implementation of the `getIdentifiersForInstance` method to define identifiers for objects of the data class.

Note In its default implementation, this method queries the name or identifier of the data object and uses that identifier in generated code. By overriding this method, you can control the identifier of your data objects in generated code.

```
classdef myCustomAttribs < Simulink.CustomStorageClassAttributes
    properties(PropertyType = 'char')
        GetFunction = '';
        SetFunction = '';
    end

    methods
        function retVal = getIdentifiersForInstance(hCSCAttrib,...
 hCSCDefn, hData, identifier)
            retVal = struct('GetFunction',...
 hData.CoderInfo.CustomAttributes.GetFunction, ...
            'SetFunction', hData.CoderInfo.CustomAttributes.SetFunction);
        end%
    end % methods
end % classdef
```

6 If you are using grouped storage classes, override the default implementation of the `getIdentifiersForGroup` method to specify the identifier for the group in generated code.

For an example, see `CSCTypeAttributes_FlatStructure.m` in the folder *`matlabroot`*`\toolbox\simulink\simulink\dataclasses\+Simulink\@CSCTypeAttributes_FlatStructure` (open).

See Also

Related Examples

- “Data Objects” on page 67-58

Determine Where to Store Variables and Objects for Simulink Models

Model data are objects and variables that you create in workspaces such as the base workspace or a data dictionary. Model data include:

- Numeric values for block parameters, such as `Simulink.Parameter` objects and MATLAB variables
- Signals, such as `Simulink.Signal` objects
- Data types
- Model configuration sets
- Simulation input and output data

You can store, partition, and share model data in a location that is appropriate for your design. The storage locations that you choose can depend on:

- Your modeling goals.
- The model architecture (referenced models, subsystems, and other partitioning strategies) and component structure.
- The types of data that you use.

Types of Data

- Simulation data is the set of input data that you use to drive a simulation and the set of output data that a simulation generates. For example, you can use variables to store input data that a simulation acquires through Inport blocks. A simulation can export output data through, for example, Outport blocks, To Workspace blocks, and logged signals.

  You can store simulation data for your current MATLAB session in the base workspace. To permanently store this simulation data, save it in a MAT-file or script file. For more information about loading, generating, and storing simulation data, see “Comparison of Signal Loading Techniques” on page 70-21 and “Export Simulation Data” on page 72-2.
- Design data is the set of variables that you use to specify block parameters and signal characteristics in a model. For example, design data includes numeric MATLAB variables, parameter and signal data objects, data type objects, and bus objects.

  You can store design data in the base workspace, model workspaces, or the Design Data section of a data dictionary. To permanently store local design data with a model, use model workspaces. To share design data between models, use data dictionaries or the base workspace. Data dictionaries permanently store the data, and you can control the data scope to establish ownership, partition the data to ease readability and maintenance, and track changes. If you use the base workspace, to permanently store the data, you must save it in a MAT-file or script file.
- Configuration sets are sets of model configuration parameters. By default, configuration sets reside in the model file, so you do not need to store the sets separately from the model. However, you cannot share these sets with other models.

  To share configuration sets between models, you must create `Simulink.ConfigSet` objects. Each object represents a standalone configuration set. You can store these objects in the base workspace or in the Configurations section of a data dictionary. If you use data dictionaries, you

can define the scope of each configuration set, compare different configuration sets, and track changes. A data dictionary inherently partitions configuration sets from other kinds of data.

Store Data for Your Design

The table shows the techniques you can use to store, partition, and manage design data and configuration sets.

Modeling Scenario	Scenario Description	Storage Locations and Techniques
Rapid prototyping and model experimentation	You want to create temporary data, such as variables to specify numeric block parameters, while you learn to use Simulink. You want to experiment with modeling techniques. You do not need to permanently store the data that you create.	Store data in the base workspace so you can quickly create and change the data.
Standalone model	You have a single model that does not depend on other systems for data. The model stands alone because it is not a piece of a larger system.	Store data in the model workspace to improve model portability. Use a data dictionary to store data that you cannot store in the model workspace. Alternatively, store all of the model data in a data dictionary. If you use a dictionary, you can partition the data by using referenced dictionaries.
Standalone hierarchy of referenced models	You have a hierarchy of referenced models that does not depend on other systems for data. The hierarchy stands alone because it is not a piece of a larger system.	Store local model data in each model workspace. Store data that the models share, such as bus objects and configuration sets, in a data dictionary. Link all of the models in the hierarchy to the dictionary. For examples, see “Migrate Model Reference Hierarchy to Use Dictionary” on page 74-6 and “Using a Data Dictionary to Manage the Data for a Fuel Control System”.

Modeling Scenario	Scenario Description	Storage Locations and Techniques
System of components	One or more teams maintain the components of a system of models. A component is a single model or a hierarchy of referenced models that represents a piece of a larger system.	Store local model data in model workspaces. Store data that the models in a component share, such as bus objects and configuration sets, in a data dictionary. Link all of the models in the component to the dictionary. Use additional referenced dictionaries to store data that the components share. For an example, see “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27.

Storage Locations

Choose any of these locations to store data:

- The MATLAB base workspace. Use the base workspace to store variables while you experiment with temporary models.
- A model workspace. Use a model workspace to permanently store data that is local to a model.
- A data dictionary. Use data dictionaries to permanently store global data, share data between models, and track changes made to data.

The chart shows the capabilities and advantages of each storage location.

Capability	Base Workspace	Model Workspace	Data Dictionary
Data-model linkage	implicit	implicit	✓
Unified interface for defining data	✓	✓	✓
Model-data dependency	✓	✓	✓
Data entry comparison	✓	✓	✓
Data entry persistence		✓	✓
Options to remedy a missing variable	✓	✓	Additional options
Shared data	✓		✓
Data grouping			✓
Change tracking for data entries			✓
Change tracking for configuration sets			✓
Data entry merging and reconciliation			✓
Storage and partitioning of auxiliary data			✓
Requirements linking			✓

For information about the way that models interact with workspaces and workspace variables, see “Symbol Resolution” on page 67-127.

Temporary Data: Base Workspace

Use the base workspace to temporarily store data:

- While you learn to use Simulink
- When you need to quickly create variables while experimenting with modeling techniques
- When you do not need to store the data permanently

To create variables in the base workspace, you can use the MATLAB command prompt or the Model Explorer. All open models can use the data that you create in the base workspace.

If you use variables to specify numeric block parameters in the model, you can programmatically change the parameter values during simulation by using commands at the command prompt. To programmatically change the values of parameters that you store in the model workspace or data dictionaries, you must use the function interfaces for those storage locations.

To permanently store base workspace data before you end a MATLAB session, you can save the data in a MAT-file or a script file. During a later session, you can load the data from the file. However, if you make changes to the data in the base workspace, you must save the data to the file again. Consider instead using a model workspace or data dictionary to permanently store data.

Local Data: Model Workspace

Use a model workspace to store data that you use only in the associated model. This data can include:

- Constant parameters, for example, numeric variables that you use to specify block parameter values.
- Data objects, such as `Simulink.Signal` and `Simulink.Parameter` objects, that you use to control signal and parameter characteristics. However, signal objects in a model workspace can use only the `Auto` storage class. If you store an `AUTOSAR.Parameter` object in a model workspace, the code generator ignores the storage class that you specify for the object.
- `Simulink.NumericType` objects that you use to specify data types. However, you cannot use the object as a data type alias. You must set the `IsAlias` property to `false`.
- Model arguments.

You can improve model portability and establish data ownership by storing the data in the model workspace. In this case, the model file permanently stores the data.

In a model reference hierarchy, each model workspace acts as a unique namespace. Therefore, you can use the same variable name in multiple model workspaces. You can then assign a unique variable value for each model.

You can use the Model Explorer to manipulate model workspace data. Alternatively, you can use the command prompt or scripts in conjunction with the model workspace programmatic interface.

For more information about using model workspaces to store local data, see “Model Workspaces” on page 67-119.

Global and Shared Data: Data Dictionary

A data dictionary is a standalone file that permanently stores data. Use data dictionaries instead of the base workspace to partition data, track changes, control access, and share data. If you link a model to a data dictionary, you can still use variables in the base workspace by configuring access from either the model or the dictionary.

As you can with model workspaces, you can use data dictionaries to directly associate data with a model. You can use this association to scope the data and to establish ownership.

When you use dictionaries, you can partition the data by storing it in additional referenced dictionaries. However, each entry in a dictionary must use a unique name. You must manage each dictionary as a separate file.

Use a data dictionary to store data that multiple models or system components share. This data can include:

- Numeric variables that multiple models use to specify block parameter values.
- `Simulink.AliasType` and `Simulink.NumericType` objects that you use to specify data types in multiple models at once.
- Data objects, including signal objects (such as `Simulink.Signal`) that use a storage class other than `Auto`. If you have a Simulink Coder license, these objects can represent signals and tunable parameters that appear as global variables in the generated code.
- `Simulink.Bus` objects that you use to define signal interfaces between referenced models.
- `Simulink.ConfigSet` objects that you use to maintain configuration parameter uniformity across multiple models.
- Enumerated type definitions, which you store using `Simulink.data.dictionary.EnumTypeDefinition` objects.

You can use the Model Explorer to manipulate dictionary data. Alternatively, you can use the command prompt or scripts in conjunction with the data dictionary programmatic interface.

For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

Considerations for Code Generation

If you intend to generate C code from a model (Simulink Coder), take these considerations into account.

- If you apply a storage class other than `Auto` to a signal object (such as `Simulink.Signal`) to control the appearance of a signal or block state in the generated code, you cannot store the object in a model workspace. Store the object in the base workspace or a data dictionary. For more information about storage classes for signals and states, see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder).
- If you apply a storage class other than `Auto` to a parameter object (such as `Simulink.Parameter`), you can store the object in the base workspace, a model workspace, or a data dictionary. However, if you store the object in a model workspace, the code generator assumes that the containing model owns the parameter. For more information, see “Code Generation Impact of Storage Location for Parameter Objects” (Simulink Coder).

See Also

Related Examples

- “Introduction to Managing Data with Model Reference”
- “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110
- “Create, Edit, and Manage Workspace Variables” on page 67-106
- “Compare Capabilities of Model Components” on page 22-8
- “Model Workspaces” on page 67-119
- “Data Objects” on page 67-58
- “Symbol Resolution” on page 67-127

Create, Edit, and Manage Workspace Variables

To share information such as parameter values and signal data types between separate blocks and models, you use workspace variables. For example, you can create a numeric MATLAB variable in the base workspace and use the variable to set the value of the **Gain** parameter in multiple Gain blocks simultaneously (see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9). You can create a `Simulink.Bus` object to explicitly define the structure of a bus signal.

You can store workspace variables in the base workspace, model workspaces, or data dictionaries. To decide where to store variables, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.

Tools for Managing Variables

Use one or more of these techniques to create, modify, store, and migrate workspace variables:

- To share block parameter values and create `Simulink.Parameter` and `Simulink.Signal` objects (for example, in preparation for code generation), you can use the Model Data Editor. You can interact with all of the block parameters, signal lines, and block states in a model at once. You can also inspect tunable block parameters in a list that you can search, sort, and filter.

  - To create a variable, in the data table, begin editing the cell that corresponds to a block parameter value (in the **Value** column) or a signal or state name (in the **Name** column). Enter the name of the variable you want to create and click the action button ⋮ in the right side of the cell.

    If a block parameter value is already set to a simple numeric expression, you can create a variable for that expression. Click ⋮ in the right side of the cell that corresponds to the value, then select **Create variable**. In the Create New Data dialog box, set the name and location for the new variable, then click **Create**. The cell now displays the new variable.
  - To modify variables by using the columns in the data table, click the **Show/refresh additional information** button. Then, the data table contains rows that correspond to the variables and objects that the model uses.
  - To interact with one variable at a time (for example, to inspect all of the variable properties at once), open the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**) and select the relevant row in the data table. The Property Inspector shows the properties of the selected variable.

  For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.
- To interact with a small number of parameters, signals, or states at a time, use individual block parameter dialog boxes or the Property Inspector to create variables for sharing block parameter values and create and configure parameter and signal objects for code generation.

  In the dialog box or the Property Inspector, click the action button ⋮ next to the value of a block parameter, signal name, or state name.
- To create and edit any type or class of variable or object, move variables between workspaces, and inspect all of the variables in a workspace at once, use the Model Explorer. You can also rename variables and precisely analyze the way that an entire model or an individual block uses variables.

See **Model Explorer** and “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110.

Edit Variable Value or Property From Block Parameter

This example shows how to change the value of a **Gain** parameter (Gain block) whose value is set by a numeric variable. Modify the variable, not the block parameter.

1. Open the model `f14`. The model loads variables into the base workspace.
2. In the model, open the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**.
3. In the model, select the Gain block that uses the variable `Mw`.
4. In the Property Inspector, click the button ⋮ next to the value of the **Gain** parameter. Select **Open**.
5. In the **Data properties** dialog box, type a new value for the variable in the **Value** box and click **OK**.

Modify Structure and Array Variables Interactively

To inspect and modify a variable whose value is a structure or array, you can launch the Variable Editor by clicking the nearby button ⋮. Choose one of these techniques:

- In the Model Explorer, select the variable in the **Contents** pane. In the Dialog pane (the right pane), the button appears.
- In the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), on the **Parameters** tab, click the **Show/refresh additional information** button. In the data table, find the row that corresponds to the variable and, in the **Value** column, begin editing the value of the variable. The button appears in the right side of the cell.
- In a block dialog box or the Property Inspector, the button appears next to the value of a block parameter that uses the variable. Click the button and use the menu options to open the property dialog box for the variable. Then, in the property dialog box, click the button again to launch the Variable Editor. You can use this technique only for parameter objects such as `Simulink.Parameter`.

Ramifications of Modifying or Deleting a Variable

When you modify or delete a variable, the change can impact multiple blocks and models that use the variable. To assess the impact by determining where the variable is used, use the Model Explorer (see “Analyze Variable Usage in a Model” on page 67-108). However, you can analyze variable usage only for models that are open at the time of the analysis. Before you perform the analysis, open any models that you suspect use the variable.

Models and blocks use variables through name resolution (see “Symbol Resolution” on page 67-127). When you change the name of a variable without making corresponding changes to dependent blocks and models, the blocks and models generate errors. Instead, to rename a variable in the context of one or more models, see “Rename a Variable Throughout a Model” on page 67-108.

When a block or model cannot access a variable that it needs, it generates an error in the Diagnostic Viewer. In some cases, you can use buttons in the Diagnostic Viewer to fix the error (for example, by

restoring a deleted variable). To increase the likelihood that you can use the Diagnostic Viewer to recover from the absence of a variable, use these techniques:

- Store variables in a data dictionary instead of the base workspace. With a data dictionary, you gain additional options for recovery. For information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.
- For every model, keep the corresponding Simulink cache file available. For example, when you share the model with someone else, share the cache file, too. When you fetch the latest model design files from a source control system, fetch the cache file from the continuous integration system or latest build folder. The cache file preserves information that Simulink Coder can use to help you recover from the absence of a variable. For more information about Simulink cache files, see “Share Simulink Cache Files for Faster Simulation” on page 8-54.

Analyze Variable Usage in a Model

To analyze the ways in which a model uses variables, use the Model Explorer. You can:

- Determine where a variable is used in a model.
- Determine whether a model uses a variable.
- Determine which variables in a workspace are not used by a model.

For more information, see “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110.

Rename a Variable Throughout a Model

This example shows how to rename a variable in the Model Data Editor.

1. Open the model `f14`. The model loads variables into the base workspace.
2. In the model, on the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, inspect the **Parameters** tab.
3. In the model, click the Gain block labeled `Mw`.

   In the Model Data Editor, the **Value** column shows that the block uses the variable `Mw`. Suppose you want to rename this variable.
4. In the Model Data Editor, click the **Show/refresh additional information** button.

   Now, the data table contains rows that correspond to workspace variables that the model uses.
5. Activate the **Change scope** button.

   Now, the data table shows information about data items in subsystems.
6. In the **Filter contents** box, enter `Mw`.

   The data table shows rows that correspond to the variable and to blocks that use the variable.
7. In the row that represents `Mw`, right-click and select **Rename All**.
8. In the **Select a system** dialog box, click the name of the model `f14` to select it as the context for renaming the variable `Mw`.
9. Clear the **Search in referenced models** check box, since `f14` does not reference any models, and click **OK**.

With **Search in referenced models** selected, you can rename the target variable everywhere it is used in a model reference hierarchy. However, renaming the target variable in an entire hierarchy can take more time.

The **Update diagram to include recent changes** check box is cleared by default to save time by avoiding unnecessary model diagram updates. Select the check box to incorporate recent changes you made to the model by forcing a diagram update.

10 In the **Rename All** dialog box, type the new name for the variable in the **New name** box and click **OK**.

11 Click **Show/refresh additional information** again. Because the renaming operation changed the name of the variable and the values of some block parameters, for more accurate information in the Model Data Editor, you must click **Show/refresh additional information**.

Interact With Variables Programmatically

At the command prompt, you can create and modify variables in the base workspace by entering commands such as `myVar = 15;`. To programmatically create, modify, and store variables in a different workspace, such as a model workspace, use the programmatic interface for the target workspace. The table shows the interfaces and techniques that you can use to programmatically manage variables.

Target Workspace	Technique or Interface
Base workspace	Enter commands at the command prompt.
Model workspace	See `Simulink.ModelWorkspace`.
Data dictionary	See “Store Data in Dictionary Programmatically” on page 74-34.

To programmatically list the variables that a model uses or does not use, see `Simulink.findVars`.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic workspace interfaces. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27
- “Data Objects” on page 67-58

Edit and Manage Workspace Variables by Using Model Explorer

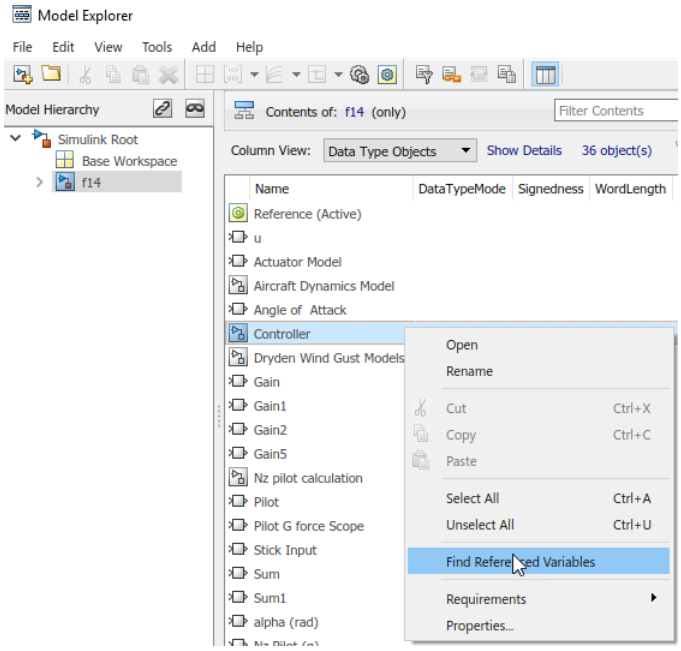
In this section...

To learn all of the techniques you can use to create, edit, and manage workspace variables, see “Create, Edit, and Manage Workspace Variables” on page 67-106.

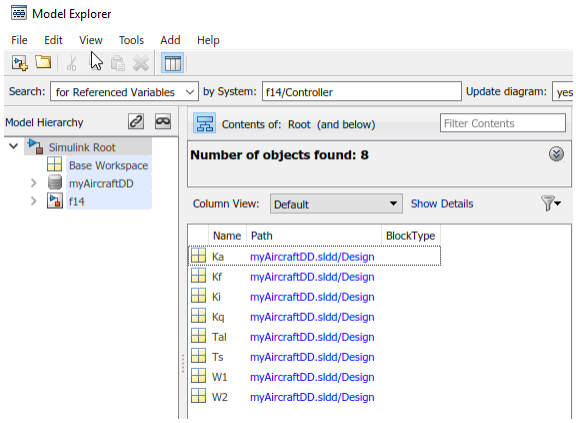
Finding Variables That Are Used by a Model or Block

In the Model Explorer, you can get a list of variables that a model or block uses. The following approach is one way to get that list of variables:

1. In the **Contents** pane, right-click the block for which you want to find the variables that it uses.
2. Select the **Find Referenced Variables** menu item.



Model Explorer returns results similar to these:



For performance, Model Explorer uses cached information from the last compiled version of the model. If you want to recompile the model, either do so manually or, in the Model Explorer, set the **Update diagram** field to `yes` and repeat the search.

You can also use the following approaches to find variables that a model or block uses:

- In the Model Explorer, in the **Model Hierarchy** pane, right-click a block or model node and select the **Find Referenced Variables** menu item.
- In the Model Explorer, in the search bar, use the `for Referenced Variables` search type option.
- In the Simulink Editor, right-click a block, subsystem, or in the canvas and select the **Find Referenced Variables** menu item. Clicking the canvas returns results for the whole model.

The `Simulink.findVars` function provides additional options for returning information about workspace variables that is not available from the Model Explorer or Simulink Editor.

For information about limitations when finding referenced variables, see the `Simulink.findVars` documentation.

Using the Set of Returned Variables

For a variable in the set of returned variables, you can find the blocks that use that variable (for details, see “Finding Blocks That Use a Specific Variable” on page 67-111). Also, you can export variables from the returned set of variables. For details, see “Export Workspace Variables” on page 67-116.

Finding Blocks That Use a Specific Variable

This example shows how to use Model Explorer to get a list of blocks that use a specific workspace variable.

1. Open the model `f14`.
2. Open Model Explorer.
3. In the **Model Hierarchy** pane, select the `Base Workspace` node.
4. In the **Contents** pane, right-click the variable `Mq` and select **Find Where Used**.
5. In the **Select a system** dialog box, select `f14`.

6 Clear the **Search in referenced models** check box, since `f14` does not reference any models, and click **OK**.

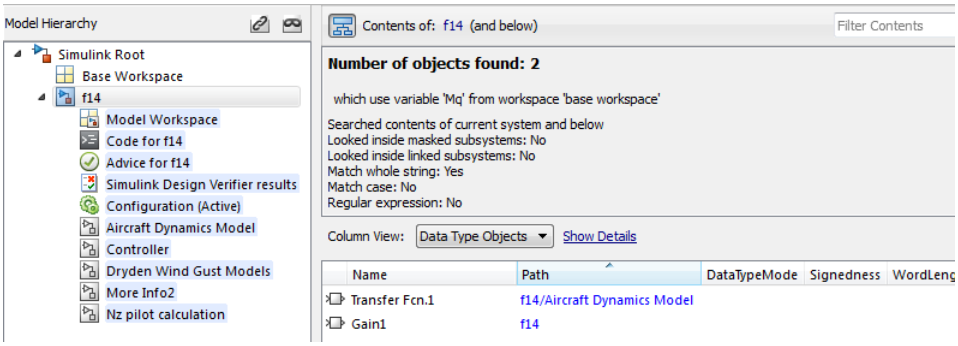
With **Search in referenced models** selected, you can find the target variable everywhere it is used in a model reference hierarchy. However, finding the target variable in an entire hierarchy can take more time.

The **Update diagram to include recent changes** check box is cleared by default to save time by avoiding unnecessary model diagram updates. Select the check box to incorporate recent changes you made to the model by forcing a diagram update.

7 Click **OK** in response to the message to update the model diagram.

Because you just opened the model, you must update the model diagram at least once before finding a variable. You could have selected **Update diagram to include recent changes** in the **Select a system** dialog box to force an initial diagram update, though you typically use that option when you make changes to the model while performing multiple searches with **Find Where Used**.

8 Model Explorer displays the search results:



The property columns whose values include `Mq` represent the block parameters that use the `Mq` variable. If those property columns are not already in the view, then the Model Explorer adds them to the end of the search results display.

You can also find blocks that use a specific variable by using one of these approaches:

- In the search bar, select the `for Variable Usage` search type option.
- In the **Search Results** pane, right-click a variable and select the **Find Where Used** menu item.
- In the Model Data Editor, right-click a workspace variable and select the **Find Where Used** menu item.

Finding Unused Workspace Variables

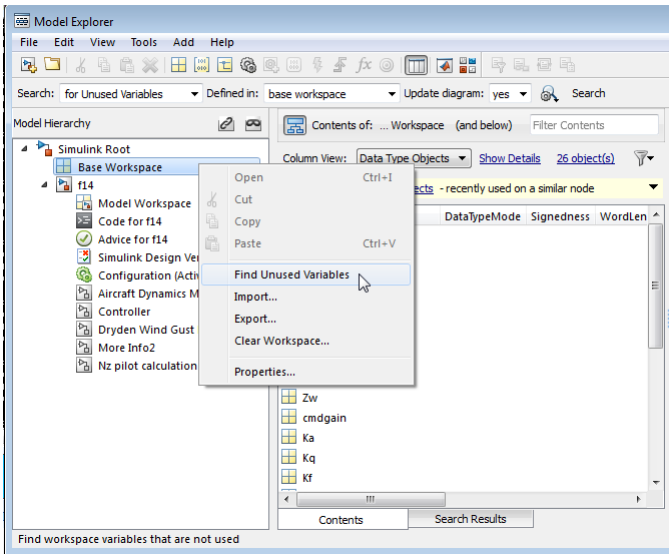
You can use the Model Explorer to get a list of variables that are defined in a workspace but not used by a model or block. One way to get that list of variables is to right-click a workspace name in the **Model Hierarchy** pane and select the **Find Unused Variables** menu item. For example:

1 Open the `f14` model.

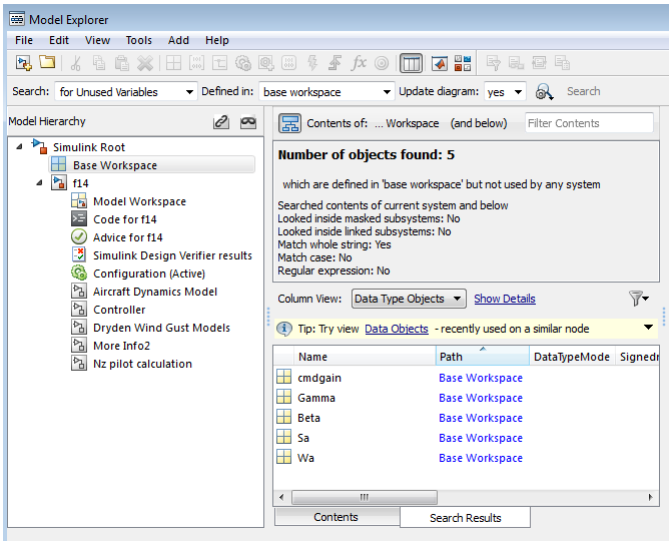
2 Open the Model Explorer.

3 In the search toolbar, set the **Update diagram** field to `yes`.

4 In the **Model Hierarchy** pane, right-click the `Base Workspace` node and select the **Find Unused Variables** menu item.



5 The Model Explorer displays output similar to this:



The `Simulink.findVars` function provides additional options for returning information about unused workspace variables that is not available from the Model Explorer or Simulink Editor.

Editing Workspace Variables

In the Model Explorer, you can use the Variable Editor to edit variables from the MATLAB base workspace or model workspace. The Variable Editor is available for editing large arrays and structures.

To open the Variable Editor:

1. In the **Contents** pane, select the variable.
2. In the Dialog pane (the right pane), click the button ⋮ near the value of the variable.
3. In the menu, select **Open Variable Editor**.

Alternatively, to open the Variable Editor from the **Contents** pane instead of the Dialog pane, begin editing the value of the variable by clicking the appropriate cell. The button appears in the cell.

Representation of Arrays with Three or More Dimensions

When the value of a variable or `Simulink.Parameter` object is an array with three or more dimensions, the **Value** column displays the array as an expression that contains a call to the `reshape` function.

To edit the values in the array, modify the first argument of the `reshape` call, which contains all of the array values in a serialized vector. When you add or remove elements along a dimension, you must also correct the argument that represents the length of the modified dimension.

Rename Variables

This example shows how to use Model Explorer to rename a variable everywhere it is used by blocks in Simulink models.

1. Open the model `sldemo_absbrake`. The model loads data to the MATLAB base workspace.
2. Open Model Explorer.
3. In the **Model Hierarchy** pane, select the base workspace.
4. In the **Contents** pane, right-click the base workspace variable `m` and select **Rename All**.
5. In the **Select a system** dialog box, click the name of the model `sldemo_absbrake` to select it as the context for renaming the variable `m`.
6. Clear the **Search in referenced models** check box and click **OK**. The model `sldemo_absbrake` references the model `sldemo_wheelspeed_absbrake`, but only `sldemo_absbrake` uses the variable `m`.

   With **Search in referenced models** selected, you can rename the target variable everywhere it is used in a model reference hierarchy. However, renaming the target variable in an entire hierarchy can take more time.

   The **Update diagram to include recent changes** check box is cleared by default to save time by avoiding unnecessary model diagram updates. Select the check box to incorporate recent changes you made to the model by forcing a diagram update.
7. Click **OK** in response to the message to update the model diagram.

   Since you just opened the model, you must update the model diagram at least once before renaming a variable. You could have selected **Update diagram to include recent changes** in the **Select a system** dialog box to force an initial diagram update, though you typically use that option when you make changes to the model while performing multiple variable renaming operations.
8. In the **Rename All** dialog box, type a new name for the variable in the **New name** box and click **OK**.

You can use the hyperlinks in the **Corresponding blocks** section of the **Rename All** dialog box to view the target blocks.

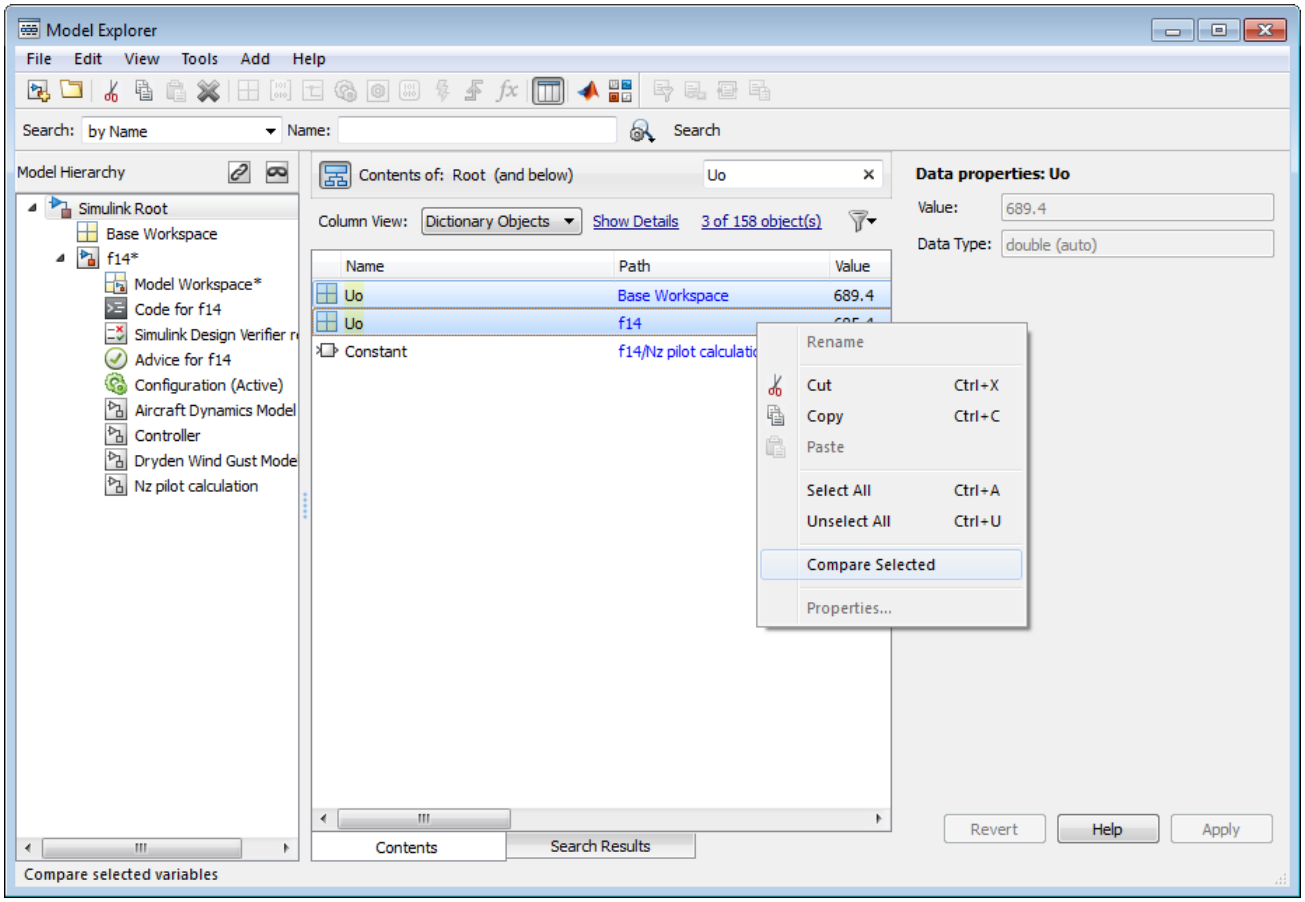
Note You can rename only variables that the function `Simulink.findVars` supports.

For help with renaming files, use a project. See “Automatic Updates When Renaming, Deleting, or Removing Files” on page 17-10.

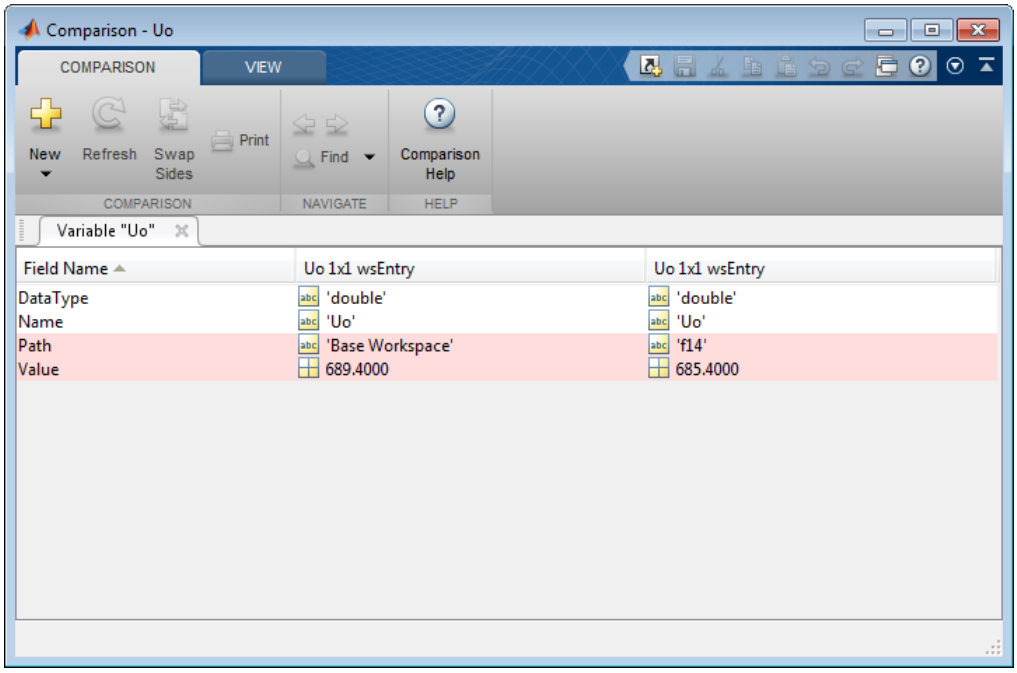
Compare Duplicate Workspace Variables

You can compare duplicate variables that are stored in the same workspace or in different workspaces. For example, you can compare a variable stored in the base workspace with its duplicate, which is stored in the model workspace.

1. Open a model and the Model Explorer.
2. In the search toolbar, search for the variable that is duplicated. Select the rows with the duplicate entries. Then, right-click and select **Compare Selected**.



3. Review the differences in the **Comparison Viewer**.



Export Workspace Variables

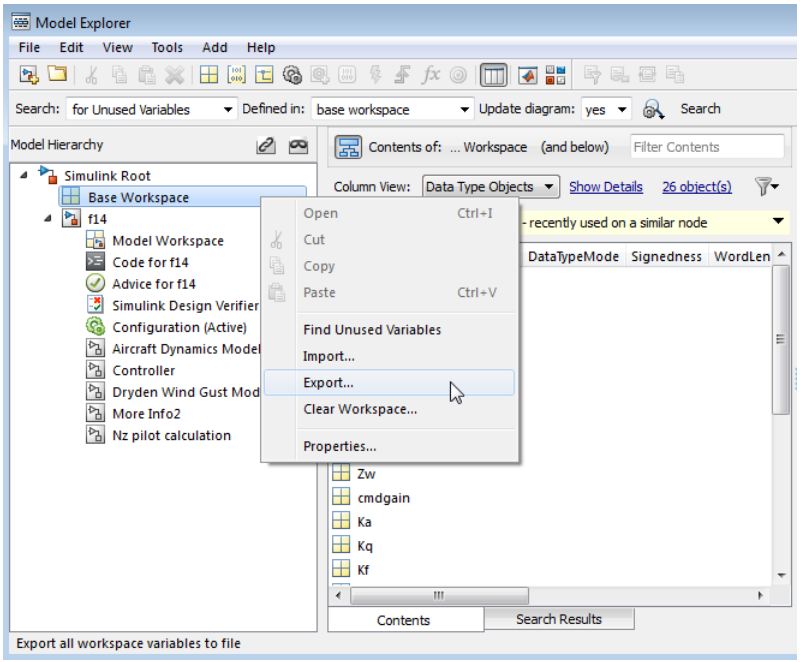
You can export (save) a set of variables listed in the Model Explorer, exporting either individual variables or all the variables in the base or model workspace.

One possible workflow is to export the set of variables returned with the **Find Referenced Variables** option or the `Simulink.findVars` function. For details, see “Finding Variables That Are Used by a Model or Block” on page 67-110.

Note All the variables that you export must be from the same workspace.

To export all the variables in a workspace in the Model Explorer to a MATLAB code file or MAT-file:

1. Select the variables that you want to export.
   a. To select all the variables in a workspace, right-click the workspace node (for example, `Base Workspace`) and select the **Export** menu item. For example:



b To select individual variables, in the **Contents** pane, select the variables that you want to export. Right-click one of the highlighted variables and select the **Export Selected** menu item.

If the **Contents** pane has data grouped by a property, selecting the top line in a group does not select all the variables in that group. For details about grouped data, see **Model Explorer**.

2 Specify whether to save the variables in a MATLAB code file or a MAT-file.

The MATLAB code file format is easier to read, is editable, and supports version control. The MAT-file format is binary, which has performance advantages.

If you specify a MATLAB code file format, the Model Explorer may create an associated MAT-file, reflecting the name of the MATLAB code file, but with an extension of `.mat` instead of `.m`.

3 Specify a name and location for the file.

4 If the file already exists, Model Explorer displays a dialog box asking you to choose one of these options:

- **Overwrite entire file**
  - Replaces all variables in the target file with the selected variables, which are stored in alphabetical order.
- **Update variables that exist in file and append new variables to file**
  - Updates existing variables in place and appends new variables.
- **Only update variables that exist in file**
  - Updates existing variables, but does not add any new variables, which eliminates potentially extraneous variables.

To permanently store workspace variables for a model, instead of using the base workspace, create a data dictionary. See “What Is a Data Dictionary?” on page 74-2.

Importing Workspace Variables

You can import (load) a set of variables from a file into the base workspace or into a model workspace using the Model Explorer. When you import variables into a workspace, the Model Explorer overwrites existing variables and adds any new variables.

To import variables into a workspace:

1. In the **Model Hierarchy** pane, right-click the workspace into which you want to import variables.
2. Select the **Import** menu item.
3. In the Import from File dialog box, select a MATLAB code file or MAT-file for the variables that you want to import.

Note If you import a MATLAB code file, then Simulink also imports the associated MAT-file.

See Also

Model Explorer | `Simulink.findVars`

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100

Model Workspaces

In this section...

Model Workspace Differences from MATLAB Workspace

Each model is provided with its own workspace for storing variable values.

The model workspace is similar to the base MATLAB workspace except that:

- Variables in a model workspace are visible only in the scope of the model.

  If both the MATLAB workspace and a model workspace define a variable of the same name, and the variable does not appear in any intervening masked subsystem or model workspaces, the Simulink software uses the value of the variable in the model workspace. A model's workspace effectively provides it with its own name space, allowing you to create variables for the model without risk of conflict with other models.
- When the model is loaded, the workspace is initialized from a data source.

  The data source can be a Model file, a MAT-file, a MATLAB file, or MATLAB code stored in the model file. For more information, see “Data source” on page 67-122.
- You can interactively reload and save MAT-file, MATLAB file, and MATLAB code data sources.
- To store a signal object in a model workspace, set the storage class of the object to `Auto`. Signal objects include `Simulink.Signal` and subclasses that you create.

  If you specify a storage class other than `Auto`, you must store signal objects in the base workspace or a data dictionary to ensure the objects are unique within the global Simulink context and accessible to all models.
- When you store MATLAB variables and parameter objects (such as `Simulink.Parameter`) in a model workspace, some tunability limitations apply. See “Tunability Considerations and Limitations for Other Modeling Goals” on page 37-36. In addition, if you store an `AUTOSAR.Parameter` object in a model workspace, the code generator ignores the storage class that you specify for the object.

Note When resolving references to variables used in a referenced model, the variables of the referenced model are resolved as if the parent model did not exist. For example, suppose a referenced model references a variable that is defined in both the parent model's workspace and in the MATLAB workspace but not in the referenced model's workspace. In this case, the MATLAB workspace is used.

Troubleshooting Memory Issues

When you use a workspace variable as a block parameter, Simulink creates a copy of the variable during the compilation phase of the simulation and stores the variable in memory. This can cause

your system to run out of memory during simulation, or in the process of generating code. Your system might run out of memory if you have:

- Large models with many parameters
- Models with parameters that have a large number of elements

This issue does not affect the amount of memory that is used to represent parameters in generated code.

Manipulate Model Workspace Programmatically

An object of the `Simulink.ModelWorkspace` class describes a model workspace. Simulink creates an instance of this class for each model that you open during a Simulink session. The methods associated with this class can be used to accomplish a variety of tasks related to the model workspace, including:

- Listing the variables in the model workspace
- Assigning values to variables
- Evaluating expressions
- Clearing the model workspace
- Reloading the model workspace from the data source
- Saving the model workspace to a specified MAT-file or MATLAB file
- Saving the workspace to the MAT-file or MATLAB file that the workspace designates as its data source

See Also

`Simulink.ModelWorkspace`

Related Examples

- “Specify Source for Data in Model Workspace” on page 67-121
- “Change Model Workspace Data” on page 67-124
- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Parameterize Instances of a Reusable Referenced Model” on page 8-64
- “Introduction to Managing Data with Model Reference”

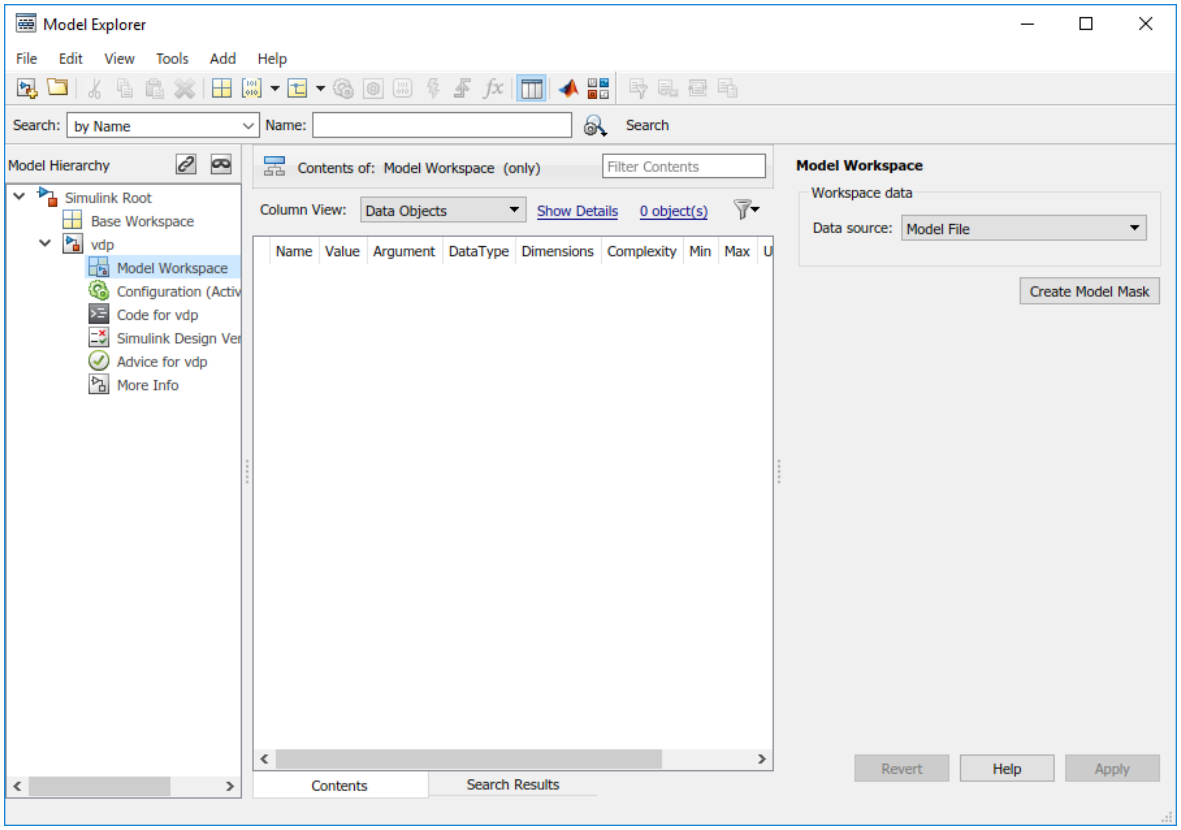
Specify Source for Data in Model Workspace

When you use a model workspace to contain the variables that a model uses, you can choose to store the variables in one of these sources:

- The model file, which can store static variable definitions.
- A separate MAT-file or MATLAB file. You can reload the variables from the external file into the model workspace at any time.
- Your own custom MATLAB code that creates variables. You can save the code as part of the model file, and reload the code at any time.

To specify a data source for a model workspace, in the Model Explorer, use the Model Workspace dialog box. To display the dialog box for a model workspace:

1. Open the Model Explorer. On the **Modeling** tab, click **Model Data Editor**.
2. In the **Model Hierarchy** pane, right-click the model workspace.



3. Select the **Properties** menu item, which opens the Model Workspace dialog box.

To use MATLAB commands to change data in a model workspace, see “Use MATLAB Commands to Change Workspace Data” on page 67-125.

Data source

The **Data source** field in the Model Workspace dialog box includes the following data source options for a workspace:

- `Model File`

  Specifies that the data source is the model itself.
- `MAT-File`

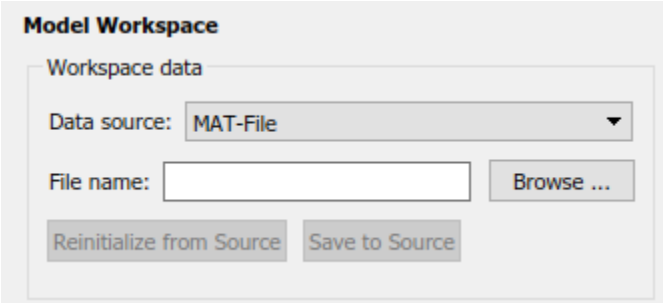
  Specifies that the data source is a MAT file. Selecting this option causes additional controls to appear (see “MAT-File and MATLAB File Source Controls” on page 67-122).
- `MATLAB File`

  Specifies that the data source is a MATLAB file. Selecting this option causes additional controls to appear (see “MAT-File and MATLAB File Source Controls” on page 67-122).
- `MATLAB Code`

  Specifies that the data source is MATLAB code stored in the model file. Selecting this option causes additional controls to appear (see “MATLAB Code Source Controls” on page 67-123).

MAT-File and MATLAB File Source Controls

Selecting `MAT-File` or `MATLAB File` as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.



File name

Specifies the file name or path name of the MAT-file or MATLAB file that is the data source for the selected workspace. If you specify a file name, the name must reside on the MATLAB path.

Reinitialize From Source

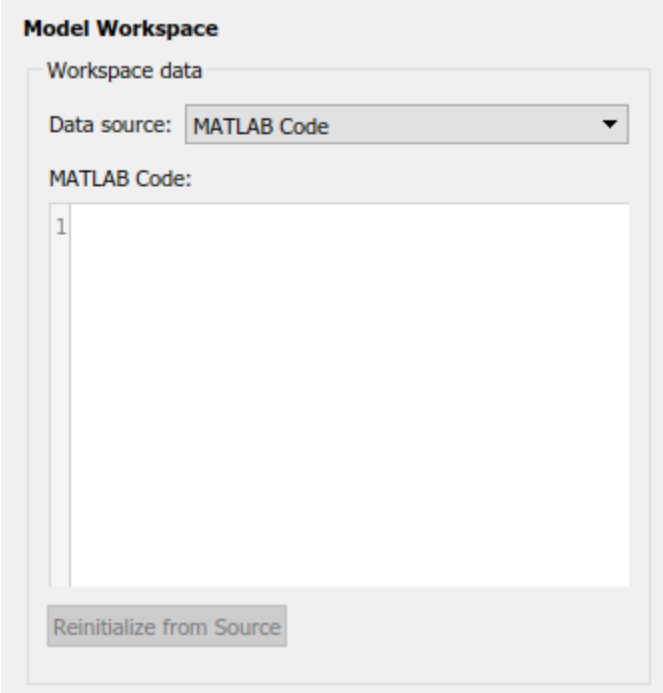
Clears the workspace and reloads the data from the MAT-file or MATLAB file specified by the **File name** field.

Save To Source

Saves the workspace in the MAT-file or MATLAB file specified by the **File name** field.

MATLAB Code Source Controls

Selecting `MATLAB Code` as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.



MATLAB Code

Specifies MATLAB code that initializes the selected workspace. To change the initialization code, edit this field, then select the **Reinitialize from source** button on the dialog box to clear the workspace and execute the modified code.

Reinitialize from Source

Clears the workspace and executes the contents of the **MATLAB Code** field.

Create Model Mask

Mask the model, which enables you to control how users of the model interact with model arguments. For more information, see “Introduction to System Mask” on page 39-48.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Model Workspaces” on page 67-119
- “Change Model Workspace Data” on page 67-124

Change Model Workspace Data

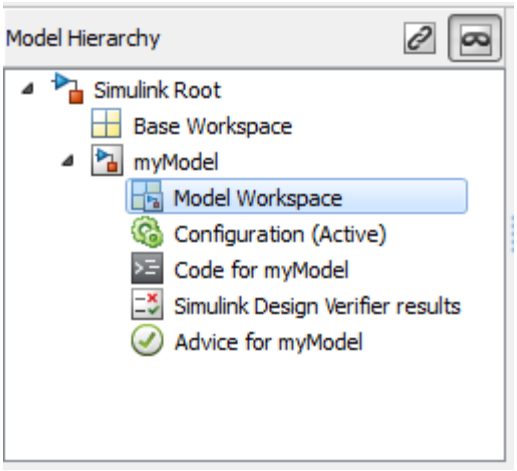
When you use a model workspace to contain the variables that a model uses, you choose a source to store the variables, such as the model file or an external MAT-file. To modify the variables at the source, you use a different procedure depending on the type of source that you selected.

Change Workspace Data Whose Source Is the Model File

If the data source of a model workspace is the model file, you can use Model Explorer or MATLAB commands to modify the stored variables (see “Use MATLAB Commands to Change Workspace Data” on page 67-125).

For example, to create a variable in a model workspace:

1. Open the Model Explorer. On the **Modeling** tab, click **Model Data Editor** or press **Ctrl+H**.
2. In the Model Explorer **Model Hierarchy** pane, expand the node for your model, and select the model workspace.



3. Select **Add** > **MATLAB Variable**.

   You can similarly use the **Add** menu or toolbar to add a `Simulink.Parameter` object to a model workspace.

To change the value of a model workspace variable:

1. Open the Model Explorer. On the **Modeling** tab, click **Model Explorer**.
2. In the Model Explorer **Model Hierarchy** pane, select the model workspace.
3. In the **Contents** pane, select the variable.
4. In the **Contents** pane or in **Dialog** pane, edit the value displayed.

To delete a model workspace variable:

1. Open the Model Explorer. On the **Modeling** tab, click **Model Explorer**.
2. In the Model Explorer **Model Hierarchy** pane, select the model workspace.
3. In **Contents** pane, select the variable.

4 Select **Edit** > **Delete**.

Change Workspace Data Whose Source Is a MAT-File or MATLAB File

You can use Model Explorer or MATLAB commands to modify workspace data whose source is a MAT-file or MATLAB file.

To make the changes permanent, in the Model Workspace dialog box, use the **Save To Source** button to save the changes to the MAT-file or MATLAB file.

1 Open the Model Explorer. On the **Modeling** tab, click **Model Explorer**.
2 In the Model Explorer **Model Hierarchy** pane, right-click the workspace.
3 Select the **Properties** menu item.
4 In the Model Workspace dialog box, use the **Save To Source** button to save the changes to the MAT-file or MATLAB file.

To discard changes to the workspace, in the Model Workspace dialog box, use the **Reinitialize From Source** button.

Changing Workspace Data Whose Source Is MATLAB Code

The safest way to change data whose source is MATLAB code is to edit and reload the source. Edit the MATLAB code and then in the Model Workspace dialog box, use **Reinitialize From Source** button to clear the workspace and re-execute the code.

To save and reload alternative versions of the workspace that result from editing the MATLAB code source or the workspace variables themselves, see “Export Workspace Variables” on page 67-116 and “Importing Workspace Variables” on page 67-118.

Use MATLAB Commands to Change Workspace Data

To use MATLAB commands to change data in a model workspace, first get the workspace for the currently selected model:

```
hws = get_param(bdroot, 'modelworkspace');
```

This command returns a handle to a `Simulink.ModelWorkspace` object whose properties specify the source of the data used to initialize the model workspace. Edit the properties to change the data source.

Use the workspace methods to:

- List, set, and clear variables
- Evaluate expressions in the workspace
- Save and reload the workspace

For example, the following MATLAB code creates variables specifying model parameters in the model workspace, saves the parameters, modifies one of them, and then reloads the workspace to restore it to its previous state.

```
hws = get_param(bdroot, 'modelworkspace');
hws.DataSource = 'MAT-File';
```

```
hws.FileName = 'params';
hws.assignin('pitch', -10);
hws.assignin('roll', 30);
hws.assignin('yaw', -2);
hws.saveToSource;
hws.assignin('roll', 35);
hws.reload;
```

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the model workspace. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

Create Model Mask

Mask the model, which enables you to control how users of the model interact with model arguments. For more information, see “Introduction to System Mask” on page 39-48.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Model Workspaces” on page 67-119
- “Specify Source for Data in Model Workspace” on page 67-121

Symbol Resolution

In this section...

Symbols

When you create a Simulink model, you can use symbols to provide values and definitions for many types of entities in the model. Model entities that you can define with symbols include block parameters, configuration set parameters, data types, signals, signal properties, and bus architecture.

A symbol that provides a value or definition must be a legal MATLAB identifier. Such an identifier starts with an alphabetic character, followed by alphanumeric or underscore characters up to the length given by the function `namelengthmax`. You can use the function `isvarname` to determine whether a symbol is a legal MATLAB identifier.

A symbol provides a value or definition in a Simulink model by corresponding to some item that:

- Exists in an accessible workspace
- Has a name that matches the symbol
- Provides the required information

Symbol Resolution Process

The process of finding an item that corresponds to a symbol is called *symbol resolution* or *resolving the symbol*. The matching item can provide the needed information directly, or it can itself be a symbol. A symbol must resolve to some other item that provides the information.

When the Simulink software compiles a model, it tries to resolve every symbol in the model, except symbols in MATLAB code that runs in a callback or as part of mask initialization. Depending on the particular case, the item to which a symbol resolves can be a variable, object, or function.

Simulink attempts to resolve a symbol by searching through the accessible workspaces in hierarchical order for a MATLAB variable or Simulink object whose name is the same as the symbol.

The search path is identical for every symbol. The search begins with the block that uses the symbol, or is the source of a signal that is named by the symbol, and proceeds upward. Except when simulation occurs via the `sim` command, the search order is:

1. Any mask workspaces, in order from the block upwards (see “Masking Fundamentals” on page 39-2).
2. The model workspace of the model that contains the block (see “Model Workspaces” on page 67-119).

3 The MATLAB base workspace (see “Create and Edit Variables”) or, if the model is linked to a data dictionary, the dictionary (see “What Is a Data Dictionary?” on page 74-2). If the data dictionary has the **Enable dictionary access to base workspace** property selected, the search treats the dictionary and the base workspace as a single namespace.

Note The **Input** and **Initial state** parameters do not load data from a data dictionary. When a model uses a data dictionary and you disable model access to the base workspace, the **Input** and **Initial state** parameters still access data in the base workspace.

If Simulink finds a matching item in the course of this search, the search terminates successfully at that point, and the symbol resolves to the matching item. The result is the same as if the value of that item had appeared literally instead of the symbol that resolved to the item. An object defined at a lower level shadows any object defined at a higher level.

If no matching item exists on the search path, Simulink attempts to evaluate the symbol as a function. If the function is defined and returns an appropriate value, the symbol resolves to whatever the function returned. Otherwise, the symbol remains unresolved, and an error occurs. Evaluation as a function occurs as the final step whenever a hierarchical search terminates without having found a matching workspace variable.

If the model that contains the symbol is a referenced model, and the search reaches the model workspace but does not succeed there, the search jumps directly to the base workspace or data dictionary *without* trying to resolve the symbol in the workspace of any parent model. Thus a given symbol resolves to the same item, irrespective of whether the model that contains the symbol is a referenced model. For information about model referencing, see “Model References”.

Numeric Values with Symbols

You can specify any block parameter that requires a numeric value by providing a literal value, a symbol, or an expression, which can contain symbols and literal values. Each symbol is resolved separately, as if none of the others existed. Different symbols in an expression can thus resolve to items on different workspaces, and to different types of item.

When a single symbol appears and resolves successfully, its value provides the value of the parameter. When an expression appears, and all symbols resolve successfully, the value of the expression provides the value of the parameter. If any symbol cannot be resolved, or resolves to a value of inappropriate type, an error occurs.

For example, suppose that the **Gain** parameter of a Gain block is given as `cos(a*(b+2))`. The symbol `cos` will resolve to the MATLAB cosine function, and `a` and `b` must resolve to numeric values, which can be obtained from the same or different types of items in the same or different workspaces. If the symbols resolve to numeric values, the value returned by the cosine function becomes the value of the **Gain** parameter.

Other Values with Symbols

Most symbols and expressions that use them provide numeric values, but the same techniques that provide numeric values can provide any type of value that is appropriate for its context.

Another common use of symbols is to name objects that provide definitions of some kind. For example, a signal name can resolve to a signal object (`Simulink.Signal`) that defines the

properties of the signal, and a Bus Creator block **Data type** parameter can name a bus object (`Simulink.Bus`) that defines the properties of the bus. You can use symbols for many purposes, including:

- Define data types
- Specify input data sources
- Specify logged data destinations

For hierarchical symbol resolution, all of these different uses of symbols, whether singly or in expressions, are the same. Each symbol is resolved, if possible, independently of any others, and the result becomes available where the symbol appeared. The only difference between one symbol and another is the specific item to which the symbol resolves and the use made of that item. The only requirement is that every symbol must resolve to something that can legally appear at the location of the symbol.

Limit Signal Resolution

Hierarchical symbol resolution traverses the complete search path by default. You can truncate the search path by using the **Permit Hierarchical Resolution** option of any subsystem. This option controls what happens if the search reaches that subsystem without resolving to a workspace variable. The **Permit Hierarchical Resolution** values are:

- `All`

  Continue searching up the workspace hierarchy trying to resolve the symbol. This is the default value.

- `None`

  Do not continue searching up the hierarchy.

- `ExplicitOnly`

  Continue searching up the hierarchy only if the symbol specifies a block parameter value, data store memory (where no block exists), or a signal or state that explicitly requires resolution. Do not continue searching for an implicit resolution. See “Explicit and Implicit Symbol Resolution” on page 67-129 for more information.

If the search does not find a match in the workspace, and terminates because the value is `ExplicitOnly` or `None`, Simulink evaluates the symbol as a function. The search succeeds or fails depending on the result of the evaluation, as previously described.

Explicit and Implicit Symbol Resolution

Models and some types of model entities have associated parameters that can affect symbol resolution. For example, suppose that a model includes a signal named Amplitude, and that a `Simulink.Signal` object named Amplitude exists in an accessible workspace. If the Amplitude signal's **Signal name must resolve to Simulink signal object** option is checked, the signal will resolve to the object. See “Signal Properties Controls” for more information.

If the option is not checked, the signal may or may not resolve to the object, depending on the value of **Configuration Parameters > Data Validity > Signal resolution**. This parameter can suppress resolution to the object even though the object exists, or it can specify that resolution occurs on the

basis of the name match alone. For more information, see “Model Configuration Parameters: Data Validity Diagnostics” > “Signal resolution”.

Resolution that occurs because an option such as **Signal name must resolve to Simulink signal object** requires it is called *explicit symbol resolution*. Resolution that occurs on the basis of name match alone, without an explicit specification, is called *implicit symbol resolution*.

Tip Implicit symbol resolution can be useful for fast prototyping. However, when you are done prototyping, consider using explicit symbol resolution, because implicit resolution slows performance, complicates model validation, and can have nondeterministic effects.

See Also

`isvarname`

Related Examples

- “Explicit and Implicit Symbol Resolution” on page 67-129
- “Create and Edit Variables”
- “Model Workspaces” on page 67-119

Configure Data Properties by Using the Model Data Editor

Models contain data items such as signals, block parameters (for example, the **Gain** parameter of a Gain block), and data stores. The Model Data Editor enables you to inspect and edit data items in a list that you can sort, group, and filter. You can then configure properties and parameters, such as data types and dimensions, without having to locate the items in the block diagram.

While creating and debugging a model, you can configure multiple data items at once by selecting the corresponding signals and blocks in the block diagram. Work with the selected items in the Model Data Editor instead of opening individual dialog boxes. Use this technique to more quickly view and compare properties of multiple signals that are close to each other in the diagram, for example, in a subsystem.

Use the Model Data Editor to configure:

- Instrumentation for signals and data stores, which means you want to view and collect the simulation values. For example, you can log signals to compare data in the Simulation Data Inspector.
- Design attributes such as data type, minimum and maximum value, and physical units. For example, you use these attributes to:
  - Specify the values of numeric block parameters.
  - Control the interaction (interface) between components through Inport and Outport blocks and data stores (see “Configure Data Interface for Component” on page 22-18).
  - Specify the dimensions of nonscalar signals in a model.

To open the Model Data Editor in a model, on the **Modeling** tab, click **Model Data Editor** or press **Ctrl+Shift+E**.

Note The Model Data Editor does not show information about data items in referenced models (which you reference with Model blocks). To work with data items in a referenced model, open the Model Data Editor in that model.

Configure Distant Data Items

The example model `sldemo_fuelsys_dd` represents the fueling system of a vehicle engine. The referenced model `sldemo_fuelsys_dd_controller` controls the rate of fuel flow to the engine. In this example, use the Model Data Editor to log signals in different subsystems and referenced models so you can inspect their data using the Simulation Data Inspector.

Explore Example Models

1. Open `sldemo_fuelsys_dd` and the referenced model `sldemo_fuelsys_dd_controller`.
2. Navigate to the `airflow_calc` subsystem.

   The Pumping Constant block contains a lookup table that describes the performance of a fuel pump. You can stream the output of this block to the Simulation Data Inspector.
3. Navigate to the root of the model and into the `fuel_calc` subsystem.
4. Navigate into the `feedforward_fuel_rate` subsystem.

The Outport block named `ff_fuel_rate` passes feedforward information to the fuel rate control algorithm.

5 Navigate back to the `fuel_calc` subsystem and into the `switchable_compensation` subsystem.

The Inport block named `ff_fuel_rate` carries the feedforward information. You can stream the output of this Inport block.

Log Signals for Data Inspection

1 Navigate to the root of the `sldemo_fuelsys_dd_controller` model.

2 In the Model Data Editor, inspect the **Signals** tab.

3 Set the **Change view** drop-down to `Instrumentation`.

4 Activate the **Change scope** button  to display the contents of the subsystems.

The Model Data Editor identifies all the signals in the model. The **Path** column appears.

5 In the **Filter Contents** box, type `ff_fuel_rate`.

The Model Data Editor updates the list of signals to include only those named `ff_fuel_rate`. You can click the link in the **Path** column to view where the signal resides within the model.

6 Select the **Log Data** check box for the signal whose path is `sldemo_fuelsys_dd_controller/fuel_calc/switchable_compensation`.

This instructs Simulink to send the data for the logged signals to the Simulation Data Inspector.

7 Filter the signals again using the text `Pumping Constant`.

The table contains one row that corresponds to the output of the Pumping Constant block.

8 Select the **Log Data** check box for the `Pumping Constant` signal.

9 Simulate the system model, `sldemo_fuelsys_dd`. During the simulation, double-click a Manual Switch block, such as `Engine Speed Selector`, to disturb the fuel control system.

10 When the simulation finishes, the **Simulation Data Inspector** button  is highlighted. This indicates that there is data to inspect and compare. Click the **Simulation Data Inspector** button.

11 In the left pane, expand the **Run** node that corresponds to the simulation run and select the check boxes for the signals whose data you want to inspect and compare.

The Simulation Data Inspector presents the values for the selected signals on the same graph.

Select Multiple Data Items from Block Diagram

In the example model `sldemo_househeat`, use the Model Data Editor to log the signals in the `Heater` subsystem for inspection using the Simulation Data Inspector.

1 In the example model `sldemo_househeat`, open the `Heater` subsystem.

2 Open the Model Data Editor and select the **Signals** tab.

The Model Data Editor identifies all the signals in the subsystem.

3. In the Model Data Editor, set the **Change view** drop-down list to `Instrumentation`.
4. Using the Simulink Editor, select all the signals in the subsystem. Optionally, do not select the output of the Constant block because the signal value does not change during the simulation.

   In response, the Model Data Editor highlights the rows that correspond to the signals you selected.
5. In the Model Data Editor, for any of the signals, click the check box in the **Log Data** column.

   The Model Data Editor selects the check box for all of the selected signals.
6. Simulate the model.
7. Open the Simulation Data Inspector and, in the leftmost pane, expand the **Run** node that corresponds to the simulation run. Select the check boxes for the signals whose values you want to inspect and compare.

Interact with a Model That Uses Workspace Variables

When you use workspace variables (such as numeric MATLAB variables and `Simulink.AliasType` objects) to share settings between data items, you can interact with those variables through the Model Data Editor. You do not need to work outside the Editor to configure the data items. In the Editor, click the **Show/refresh additional information** button, which finds variables that the model uses by updating the block diagram.

This example shows how to work with objects that a model uses to set block parameter values. You modify the value of a variable that the model `sldemo_fuelsys` uses.

1. Open the model.

   ```
   sldemo_fuelsys
   ```
2. Open the Model Data Editor **Parameters** tab.
3. In the Model Data Editor, click the **Show/refresh additional information** button.

   The data table now contains rows that correspond to variables and objects that the model uses.
4. In the model, navigate into the `fuel_rate_control` subsystem and then the `airflow_calc` subsystem.
5. In the Model Data Editor, next to the **Filter contents** box, select the **Filter using selection** button.

   With this button selected, when you select a block or signal in the block diagram, the data table shows only the data items and workspace variables that are relevant to that block or signal.
6. In the model, click the lookup table block labeled `Pumping Constant`.

   The Model Data Editor shows that the block uses three workspace variables. The block acquires some breakpoint values from the variable `SpeedVect`.

Now, you can use the columns in the Model Data Editor to configure the properties of `SpeedVect`.

You can further interact with a variable to:

- Configure other properties that the columns do not represent:

1. In the model, open the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**.
2. In the Model Data Editor, select the row that corresponds to the target variable or object. If the Property Inspector does not respond, select a different row and then select the target row again.
3. Use the Property Inspector to configure the target properties.

- Move the variable between workspaces and data dictionaries and configure the variable alongside other variables. Use the Model Explorer. To open the Model Explorer, in the Model Data Editor data table, double-click the icon in the leftmost column. For more information about using the Model Explorer, see “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110.
- Rename a variable everywhere it is used by blocks in Simulink models. In the Model Data Editor, right-click the variable and select **Rename All**. You can rename only variables that the function `Simulink.findVars` supports.
- Find blocks that use a specific variable. In the Model Data Editor, right-click the variable and select **Find Where Used**.

Find and Organize Data by Filtering, Sorting, and Grouping

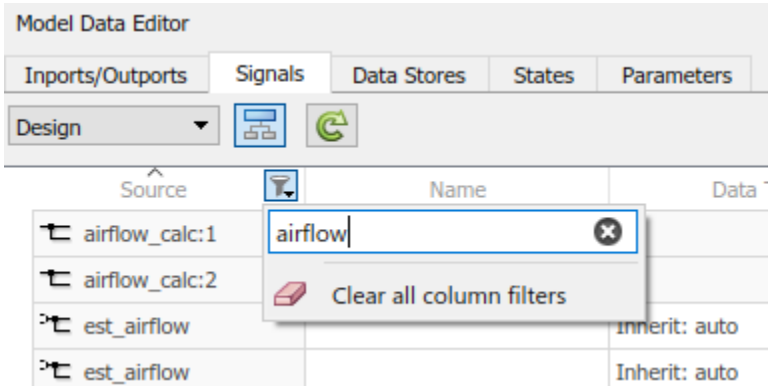
In the example model `sldemo_fuelsys_dd_controller`, variables and parameter objects set the values of block parameters. The variables and objects reside in a data dictionary. Use the Model Data Editor to display these dictionary entries together in a group.

1. In the example model, open the Model Data Editor and select the **Parameters** tab.
2. Activate the **Change scope** button to display the contents of the subsystems.
3. Click the **Show/refresh additional information** button to display rows that correspond to the dictionary entries.
4. Right-click the **Source** column header and select **Group by This Column**.

   The Model Data Editor groups the list by block or workspace (including a group for the dictionary entries).
5. Find the group labeled **Source: Dictionary**. Now, you can use the Model Data Editor to inspect and modify the attributes of the variables and objects in the dictionary.

The Model Data Editor allows you to filter a list of data items by using one or a combination of these methods:

- To filter the data table through a text search, use the **Filter contents** box.
- To filter based on the blocks or signals that you select in the model, next to the **Filter contents** box, select the **Filter using selection** button. Then, as you click blocks and signals in the model, the Model Data Editor shows you only the rows that are relevant to that block or signal. If you lasso multiple blocks or signals, the Model Data Editor shows only the rows that are relevant to those model elements.
- To filter on column-specific criteria, point to a column header and click the filter icon  . As you type in the text box, the editor applies a substring filter to the column contents. After the filter is applied, the column displays a smaller filter icon  next to the column header. To edit a filter, remove a filter, or remove all column filters, click this icon.



Inspect Individual Data Item

To focus on an individual data item, use one of these techniques:

- In the Model Data Editor, next to the **Filter contents** box, select the **Filter using selection** button. Then, in the model, click the block or signal that corresponds to the data item.

  Use this technique to configure the item by using the columns in the data table.
- In the model, open the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**. Then, in the data table, click the target row. The Property Inspector shows the properties of the data item. If the Property Inspector does not respond when you click the target row, click a different row and then click the target row again.

  Use this technique to inspect all of the properties that the Model Data Editor can access at once (in other words, the union of the columns available in the `Design` and `Instrumentation` views).
- In the model, open the Property Inspector. Then, in the data table, for the target row, double-click the cell in the leftmost column (the icon). In the model, select the highlighted block or signal.

  Use this technique to inspect all properties, including those that the Model Data Editor cannot access.

Navigate from Model Data Editor to Block Diagram

To navigate from a data item in the Model Data Editor to the block in the diagram that owns the data item, double-click the icon in the left-most column. The Simulink Editor then focuses on the relevant block. Use this technique to navigate to blocks when you select **Change scope** to view the contents of subsystems below the current system.

Columns in the Data Table

Use this table to find more information about the purpose of the columns in the Model Data Editor.

Column Name	Purpose and More Information
Source	Shows the name of the block that defines the data item. For signals, also shows the number of the block port that generates the signal. For workspace variables, shows the name of the workspace or data dictionary that contains the variable.
Signal Name or **Name**	Sets the name of the signal, state, or data store. For information about naming signals, see “Signal Names and Labels” on page 75-3. For parameters, displays the programmatic name of each parameter. For workspace variables, sets the name of the variable.
Data Type	“Control Signal Data Types” on page 67-6 and “Control Block Parameter Data Types” on page 37-44
Min and **Max**	“Specify Signal Ranges” on page 75-31 and “Specify Minimum and Maximum Values for Block Parameters” on page 37-52
Dimensions	“Determine Signal Dimensions” on page 75-19
Complexity	Sets the numeric complexity of the data item.
Sample Time	“What Is Sample Time?” on page 7-2
Unit	“Unit Specification in Simulink Models” on page 9-2
Test Point	“Configure Signals as Test Points” on page 75-43
Log Data	“Iterate Model Design Using the Simulation Data Inspector” on page 29-71
Resolve	Corresponds to the **Signal name must resolve to Simulink signal object** check box in the Signal Properties dialog box and similar check boxes in block dialog boxes for states and data stores. See “Use Signal Objects” on page 67-59.
Shared	Corresponds to the **Share across model instances** parameter of the Data Store Memory block. See Data Store Memory.
Initial Value	Sets the initial value of the state or data store. See “Initialize Signal Values” on page 75-9.
Value	“Set Block Parameter Values” on page 37-2
Argument	Configures a variable in a model workspace as a model argument. See “Parameterize Instances of a Reusable Referenced Model” on page 8-64.

Column Name	Purpose and More Information
Path	Shows the location of the block in the model and provides a link to the block in the Simulink Editor. Visible when you click the **Change Scope** button.

Two Entries Per Cell in the Data Table

When a cell contains two entries (for instance, in the **Data Type** column), the entry on the right side of the cell indicates compiled information. The compiled information shows you the value that the data item uses for simulation.

For example, the default data type setting for most signals in a model is `Inherit: Inherit via internal rule`. With this setting, after you update the block diagram, Simulink chooses a specific data type, such as `single`, for the signal to use for simulation. In the Model Data Editor, the cell in the **Data Type** column shows `Inherit: Inherit via internal rule` on the left side and `single` on the right side.

Model Data Editor Limitations

- You cannot access these attributes by using the Model Data Editor:
  - Any settings related to code generation. Instead, use the Code Mappings editor or code mappings API.
  - For mask parameters:
    - Any settings for tunable mask parameters other than the parameter value.
    - Any settings for nontunable mask parameters.

    Note that some built-in blocks are masked and can have tunable or nontunable mask parameters.
  - Any settings for parameters of Simscape blocks.
  - Any settings for data items in referenced models. Instead, open the Model Data Editor in the referenced models.
  - Any settings for variables that are not defined in the base workspace, a model workspace, or a data dictionary. For example, you cannot access the attributes of variables created by mask initialization code.
  - On the **Parameters** tab, the data type, minimum value, and maximum value of a Constant block. Use the **Signals** tab instead.

  For some settings that you cannot access with the Model Data Editor, you can use the Property Inspector instead (see “Parameters”). Open the Inspector and select the target data item in the model, not in the Model Data Editor. For mask parameters, use the mask dialog box or the Mask Editor as described in “Masking Fundamentals” on page 39-2.
- The Model Data Editor does not show Stateflow data. However, the Model Data Editor shows the data for Simulink Functions that you define inside Stateflow charts.

  To manage Stateflow data, events, and messages in a chart, see “Manage Data, Events, and Messages in the Symbols Pane” (Stateflow).

- On the **Parameters** tab, these variables are not available:
  - Variables used by non-tunable block parameters. For example, the minimum and maximum parameters on a Gain block or the `Sample time` on a Constant block.
  - Variant control variables
  - Variables used for symbolic dimensions

See Also

Related Examples

- “Use the Model Data Editor for Batch Editing” on page 67-8
- “Configure Data Interface for Component” on page 22-18
- “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder)
- “Decide How to Visualize Simulation Data” on page 30-2
- “Configure Generated Code According to Interface Control Document Interactively” (Embedded Coder)

Upgrade Level-1 Data Classes

Simulink no longer supports level-1 data classes. You must upgrade data classes that you created using the level-1 data class infrastructure, which was removed in a previous release.

Run the following utility function while specifying the destination folder for the upgraded classes.

Note Property types defined in level-1 data classes that are not subclasses of `Simulink.Parameter`, `Simulink.Signal`, or `Simulink.CustomStorageClassAttributes` are not preserved during an upgrade. Only subclasses of these three classes will preserve attributes `PropertyType` and `AllowedValues`.

1. This command upgrades all your level-1 data class packages. You cannot upgrade selected data packages.

   ```
   Simulink.data.upgradeClasses('C:\MyDataClasses')
   ```

   Here, `C:\MyDataClasses` is the destination folder for your level-2 data classes.

   Note Do not place your upgraded level-2 classes and their equivalent level-1 classes in the same folder.

   `Simulink.data.upgradeClasses` uses the `packagedefn.mat` file in your level-1 class packages for the upgrade and creates level-2 classes in the specified destination folder. Then, `Simulink.data.upgradeClasses` adds the folder to top of the MATLAB path and saves the path.

   Note If `Simulink.data.upgradeClasses` cannot save the MATLAB path because of restricted access, a warning appears. In this case, manually add the folder to the top of the MATLAB path and save the path using `savepath`.

2. You can change the location of the level-2 package folders after they have been generated. However, you will need to update your MATLAB path so that MATLAB can find these package folders.
3. Resave MAT-files and models that contain level-1 data objects.
4. Retain your level-1 classes on the MATLAB path until you have resaved all of your models and MAT-files that contain level-1 data objects. Any models or MAT-files that contain level-1 data objects will continue to load successfully while your level-1 data classes are on the MATLAB path.

   Note You cannot use both level-1 and level-2 data classes at the same time. Level-2 classes need to be above the level-1 classes on the MATLAB path so that they are found by MATLAB.

See Also

`Simulink.Parameter` | `Simulink.Signal`

Related Examples

- “Define Data Classes” on page 67-96
- “Data Objects” on page 67-58

Associating User Data with Blocks

You can use the `set_param` command to associate your own data with a block. For example, the following command associates the value of the variable `mydata` with the currently selected block.

```
set_param(gcb, 'UserData', mydata)
```

The value of `mydata` can be any MATLAB data type, including arrays, structures, objects, and Simulink data objects.

Use `get_param` to retrieve the user data associated with a block.

```
get_param(gcb, 'UserData')
```

The following command saves the user data associated with a block in the model file of the model containing the block.

```
set_param(gcb, 'UserDataPersistent', 'on');
```

Note If persistent `UserData` for a block contains any Simulink data objects, the directories containing the definitions for the classes of those objects must be on the MATLAB path when you open the model containing the block.

See Also

Related Examples

- “Specify Block Properties” on page 36-4

Support Limitations for Simulink Software Features

The software does not support the following Simulink software features. Avoid using these unsupported features.

Not Supported	Description
Variable-step solvers	The software supports only fixed-step solvers. For more information, see “Fixed Step Solvers in Simulink” on page 25-21.
Callback functions	The software does not execute model callback functions during the analysis. The results that the analysis generates, such as the harness model, may behave inconsistently with the expected behavior. • If a model or any referenced model calls a callback function that changes any block parameters, model parameters, or workspace variables, the analysis does not reflect those changes. • Changing the storage class of base workspace variables on model callback functions or mask initializations is not supported. • Callback functions called prior to analysis, such as the `PreLoadFcn` or `PostLoadFcn` model callbacks, are fully supported.
Model callback functions	The software only supports model callback functions if the `InitFcn` callback of the model is empty.
Algebraic loops	The software does not support models that contain algebraic loops. For more information, see “Algebraic Loop Concepts” on page 3-27.
Masked subsystem initialization functions	The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.
Variable-size signals	The software does not support variable-size signals. A variable-size signal is a signal whose size (number of elements in a dimension), in addition to its values, can change during model execution. For more information, see “Variable-Size Signal Basics” on page 77-2.
Multiword fixed-point data types	The software does not support multiword fixed-point data types larger than 128 bits.

Not Supported	Description
Nonzero start times	Although Simulink allows you to specify a nonzero simulation start time, the analysis generates signal data that begins only at zero. If your model specifies a nonzero start time: • If you do not select the **Reference input model in generated harness** parameter (the default), the harness model is a subsystem. The analysis sets the start time of the harness model to 1 and continues the analysis. • If you select the **Reference input model in generated harness** parameter, a Model block references the harness model. The software cannot change the start time of the harness model, so the analysis stops and you see a recommendation to set the **Start time** parameter to 0. • Simulink Design Verifier assumes zero start time for analysis and generates signal data that begins at zero. Zero start time might impact the reporting of the objective status. For example, in the test generation analysis, the software might report some objectives as `Undecided with Testcases`. For more information, see “Simulation Basics” on page 25-2.
Nonfinite data	The software does not support nonfinite data (for example, `NaN` and `Inf`) and related operations. In the Relational Operator block, the software assigns the output as follows: • If the **Relational operator** parameter is `isFinite`, the output is always 1. • If the **Relational operator** parameter is `isNan` or `isInf`, the output is always 0. In the MATLAB Function block, the software assigns the return value as follows: • For the `isFinite` function, the output is always 1. • For the `isNan` and `isInf` functions, the output is always 0.
Concurrent execution	The software does not support models that are configured for concurrent execution.
Signals with nonzero sample time offset	The software does not support models with signals that have nonzero sample time offsets.
Models with no output ports	The software only supports models that have one or more output ports.
Large floating-point constants outside the range `[-realmax/2, realmax/2]`	The use of large floating-point constants can cause out of memory errors or substantial loss of precision. Avoid using such constants if possible.
Symbolic Dimensions	The software does not support symbolic dimensions for test generation, property proving, or design error detection.

Not Supported	Description
Simulink Strings	Models that contain blocks with string data types as block parameters are not supported. For more information, see “Simulink Strings” on page 67-40.
Row-major Algorithms	The software does not support row-major algorithms for block simulation. For more information see, “Use algorithms optimized for row-major array layout”. 1 The model is incompatible for Simulink Design Verifier analysis when in **Configuration Parameters**: • In **Code Generation** > **Interface** pane, the **Array layout** parameter is set to `Row-major`. • In **Math and Data types** pane, the parameter `Use algorithms optimized for Row-major array layout` is set to `on`. 2 Simulink Design Verifier will display incompatibility message if the model contains a MATLAB Function block that uses `coder.rowMajor` directive.

See Also

More About

- “Supported and Unsupported Simulink Blocks” on page 67-145
- “Support Limitations for Stateflow Software Features” on page 67-154

Supported and Unsupported Simulink Blocks

The software provides various levels of support for Simulink blocks:

- Fully supported
- Partially supported
- Not supported

If your model contains unsupported blocks, you can enable automatic stubbing. Automatic stubbing considers the interface of the unsupported blocks, but not their behavior. If any of the unsupported blocks affect the simulation outcome, however, the analysis may achieve only partial results. For details about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).

To achieve 100% coverage, avoid using unsupported blocks in models that you analyze. Similarly, for partially supported blocks, specify only the block parameters that the software recognizes.

The following tables summarize the analysis support for Simulink blocks. Each table lists the blocks in a Simulink library and describes support information for that particular block.

Additional Math and Discrete Library

The software supports all blocks in the Additional Math and Discrete library.

Commonly Used Blocks Library

The Commonly Used Blocks library includes blocks from other libraries. Those blocks are listed under their respective libraries.

Continuous Library

Block	Support Notes
Derivative	Not supported
Integrator	Not supported and not stubbable
Integrator Limited	Not supported and not stubbable
PID Controller	Not supported
PID Controller (2 DOF)	Not supported
Second Order Integrator	Not supported and not stubbable
Second Order Integrator Limited	Not supported and not stubbable
State-Space	Not supported and not stubbable
Transfer Fcn	Not supported and not stubbable
Transport Delay	Not supported
Variable Time Delay	Not supported
Variable Transport Delay	Not supported
Zero-Pole	Not supported and not stubbable

Discontinuities Library

The software supports all blocks in the Discontinuities library.

Discrete Library

Block	Support Notes
Delay	Supported
Difference	Supported
Discrete Derivative	Supported
Discrete Filter	Supported
Discrete FIR Filter	Supported
Discrete PID Controller	Supported
Discrete PID Controller (2 DOF)	Supported
Discrete State-Space	Not supported
Discrete Transfer Fcn	Supported
Discrete Zero-Pole	Not supported
Discrete-Time Integrator	Supported
Memory	Supported
Tapped Delay	Supported
Transfer Fcn First Order	Supported
Transfer Fcn Lead or Lag	Supported
Transfer Fcn Real Zero	Supported
Unit Delay	Supported
Zero-Order Hold	Supported

Logic and Bit Operations Library

The software supports all blocks in the Logic and Bit Operations library.

Lookup Tables Library

Block	Support Notes
Cosine	Supported
Direct Lookup Table (n-D)	Supported
Interpolation Using Prelookup	Not supported when: • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `4`. or • The **Interpolation method** parameter is `Linear` and the **Number of sub-table selection dimensions** parameter is not `0`.

Block	Support Notes
1-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`.
2-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`.
n-D Lookup Table	Not supported when: • The **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`. or • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `5`.
Lookup Table Dynamic	Supported
Prelookup	Not supported when output is an array of buses
Sine	Supported

Math Operations Library

Block	Support Notes
Abs	Supported
Add	Supported
Algebraic Constraint	Supported
Assignment	Supported
Bias	Supported
Complex to Magnitude-Angle	Supported
Complex to Real-Imag	Supported
Divide	Supported
Dot Product	Supported
Find Nonzero Elements	Not supported
Gain	Supported
Magnitude-Angle to Complex	Supported
Math Function	Supported
Matrix Concatenate	Supported
MinMax	Supported
MinMax Running Resettable	Supported
Permute Dimensions	Supported
Polynomial	Supported
Product	Supported
Product of Elements	Supported
Real-Imag to Complex	Supported

Block	Support Notes
Reciprocal Sqrt	Not supported
Reshape	Supported
Rounding Function	Supported
Sign	Supported
Signed Sqrt	Not supported
Sine Wave Function	Not supported
Slider Gain	Supported
Sqrt	Not Supported
Squeeze	Supported
Subtract	Supported
Sum	Supported
Sum of Elements	Supported
Trigonometric Function	Supported if **Function** is `sin`, `cos`, or `sincos`, and **Approximation method** is `CORDIC`.
Unary Minus	Supported
Vector Concatenate	Supported
Weighted Sample Time Math	Supported

Model Verification Library

The software supports all blocks in the Model Verification library.

Model-Wide Utilities Library

Block	Support Notes
Block Support Table	Supported
DocBlock	Supported
Model Info	Supported
Timed-Based Linearization	Not supported
Trigger-Based Linearization	Not supported

Ports & Subsystems Library

Block	Support Notes
Atomic Subsystem	Supported
Code Reuse Subsystem	Supported
Configurable Subsystem	Supported
Enable	Supported

Block	Support Notes
Enabled Subsystem	Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” (Simulink Design Verifier). Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation.
Enabled and Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” (Simulink Design Verifier). Simulink Design Verifier treats Enabled and Triggered Subsystems as short-circuited during test generation.
For Each	Supported with the following limitations: • When For Each Subsystem contains one or more Simulink Design Verifier Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported. • When the mask parameters of the For Each Subsystem are partitioned, not supported.
For Each Subsystem	Supported with the following limitations: • When For Each Subsystem contains one or more Simulink Design Verifier Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported. • When the mask parameters of the For Each Subsystem are partitioned, not supported.
For Iterator Subsystem	Supported
Function-Call Feedback Latch	Supported
Function-Call Generator	Supported
Function-Call Split	Supported
Function-Call Subsystem	Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” (Simulink Design Verifier). Not supported when the Function-Call Subsystem is invoked using function-call triggers passed via root-level Inport blocks. For more information see, “Export-Function Models Overview” on page 10-97.

Block	Support Notes
If	Parameter configurations are not supported. The analysis ignores parameter configurations that you specify for an If block.
If Action Subsystem	Supported
In Bus Element	Supported
Inport	Supported
Model	Supported except for the limitations described in “Support Limitations for Model Blocks” (Simulink Design Verifier).
Out Bus Element	Supported
Outport	Supported
Resettable Subsystem	Supported
Subsystem	Supported
Variant Transitions in Stateflow	Supported. Only the active variant is analyzed.
Switch Case	Supported
Switch Case Action Subsystem	Supported
Trigger	Supported
Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” (Simulink Design Verifier). Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation.
Variant Subsystem	Not supported when the **Generate preprocessor conditionals** parameter is enabled. Only the active variant is analyzed.
While Iterator Subsystem	Supported

Signal Attributes Library

The software supports all blocks in the Signal Attributes library.

Signal Routing Library

Block	Support Notes
Bus Assignment	Supported
Bus Creator	Supported
Bus Selector	Supported

Block	Support Notes
Data Store Memory	Supported
Data Store Read	Supported
Data Store Write	Supported
Demux	Supported
Environment Controller	Supported
From	Supported
Goto	Supported
Goto Tag Visibility	Supported
Index Vector	Supported
Manual Switch	The Manual Switch block is compatible with the software, but the analysis ignores this block in a model. The analysis does not flag the coverage objectives for this block as satisfiable or unsatisfiable. Model coverage data is collected for the Manual Switch block.
Merge	Supported
Multiport Switch	Supported
Mux	Supported
Selector	Supported
Switch	Supported
Vector Concatenate	Supported

Sinks Library

Block	Support Notes
Display	Supported
Floating Scope	Supported
Outport (Out1)	Supported
Out Bus Element	Supported
Scope	Supported
Stop Simulation	Not supported and not stubbable
Terminator	Supported
To File	Supported
To Workspace	Supported
XY Graph	Supported

Sources Library

Block	Support Notes
Band-Limited White Noise	Not supported

Block	Support Notes
Chirp Signal	Not supported
Clock	Supported
Constant	Supported unless **Constant value** is `inf`.
Counter Free-Running	Supported
Counter Limited	Supported
Digital Clock	Supported
Enumerated Constant	Supported
From File	Not supported. When MAT-file data is stored in MATLAB `timeseries` format, not stubbable.
From Workspace	Not supported
Ground	Supported
Inport (In1)	Supported
In Bus Element	Supported if `Simulink.Bus` type is defined for the In Bus Element.
Pulse Generator	Supported
Ramp	Supported
Random Number	Not supported and not stubbable
Repeating Sequence	Not supported
Repeating Sequence Interpolated	Not supported
Repeating Sequence Stair	Supported
Signal Builder	Not supported
Signal Editor	Not supported
Signal Generator	Not supported
Sine Wave	Not supported
Step	Supported
Uniform Random Number	Not supported and not stubbable

User-Defined Functions Library

Block	Support Notes
Initialize Function	Not supported
Interpreted MATLAB Function	Not supported
Level-2 MATLAB S-Function	For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” (Simulink Design Verifier).
MATLAB Function	For limitations, see “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier).

Block	Support Notes
MATLAB System	• Decision, Condition and MCDC Coverage objectives are supported in Test Generation. Enhanced MCDC, Relational Boundary and Custom Test objectives are not supported. • Custom Proof objectives are not supported in Property Proving. • For further limitations, see “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier).
Reset Function	Not supported
S-Function Builder	For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” (Simulink Design Verifier).
Terminate Function	Not supported

See Also

More About

- “Support Limitations for Simulink Software Features” on page 67-142
- “Support Limitations for Stateflow Software Features” on page 67-154

Support Limitations for Stateflow Software Features

Simulink Design Verifier does not support the following Stateflow software features. Avoid using these unsupported features in models that you analyze.

In this section...

ml Namespace Operator, ml Function, ml Expressions

The software does not support calls to MATLAB functions or access to MATLAB workspace variables, which the Stateflow software allows. See "Access MATLAB Functions and Workspace Data in C Charts" (Stateflow).

C or C++ Operators

The software does not support the `sizeof` operator, which the Stateflow software allows.

C Math Functions

The software supports calls to the following C math functions:

- `abs`
- `ceil`
- `fabs`
- `floor`
- `fmod`
- `labs`
- `ldexp`
- `pow` (only for integer exponents)

The software does not support calls to other C math functions, which the Stateflow software allows. If automatic stubbing is enabled, which it is by default, the software eliminates these unsupported functions during the analysis.

For information about C math functions in Stateflow, see "Call C Library Functions in C Charts" (Stateflow).

Note For details about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).

Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart

The software does not support atomic subcharts that call exported graphical functions, which the Stateflow software allows.

Note For information about exported functions, see “Export Stateflow Functions for Reuse” (Stateflow).

Atomic Subchart Input and Output Mapping

If an input or output in an atomic subchart maps to chart-level data of a different scope, the software does not support the chart that contains that atomic subchart.

For an atomic subchart input, this incompatibility applies when the input maps to chart-level data of output, local, or parameter scope. For an atomic subchart output, this incompatibility applies when the output maps to chart-level data of local scope.

Recursion and Cyclic Behavior

The software does not support recursive functions, which occur when a function calls itself directly or indirectly through another function call. Stateflow software allows you to implement recursion using graphical functions.

In addition, the software does not support recursion that the Stateflow software allows you to implement using a combination of event broadcasts and function calls.

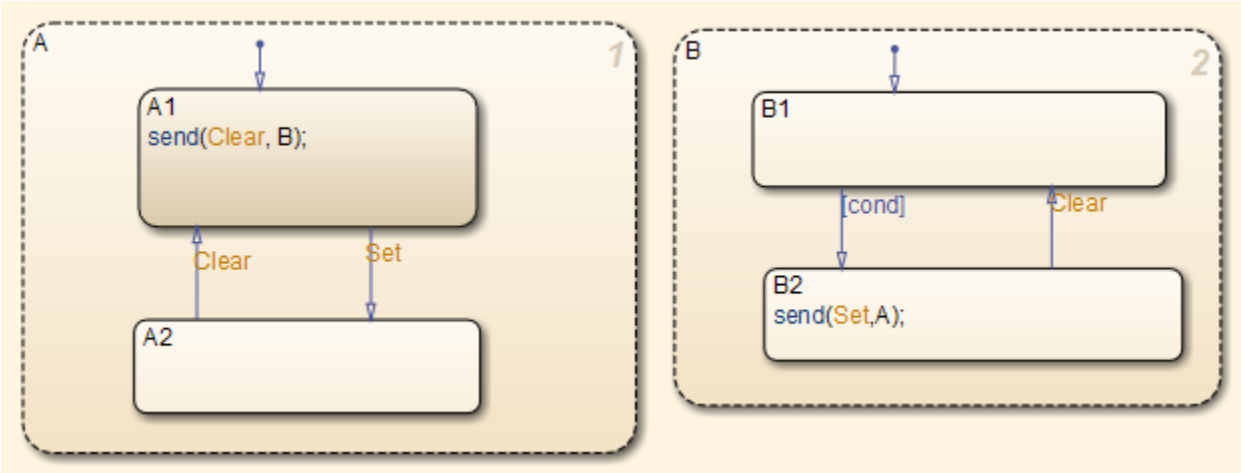
Note For information about avoiding recursion in Stateflow charts, see “Avoid Unwanted Recursion in a Chart” (Stateflow).

Stateflow software also allows you to create *cyclic behavior*, where a sequence of steps is repeated indefinitely. If your model has a chart with cyclic behavior, the software cannot analyze it.

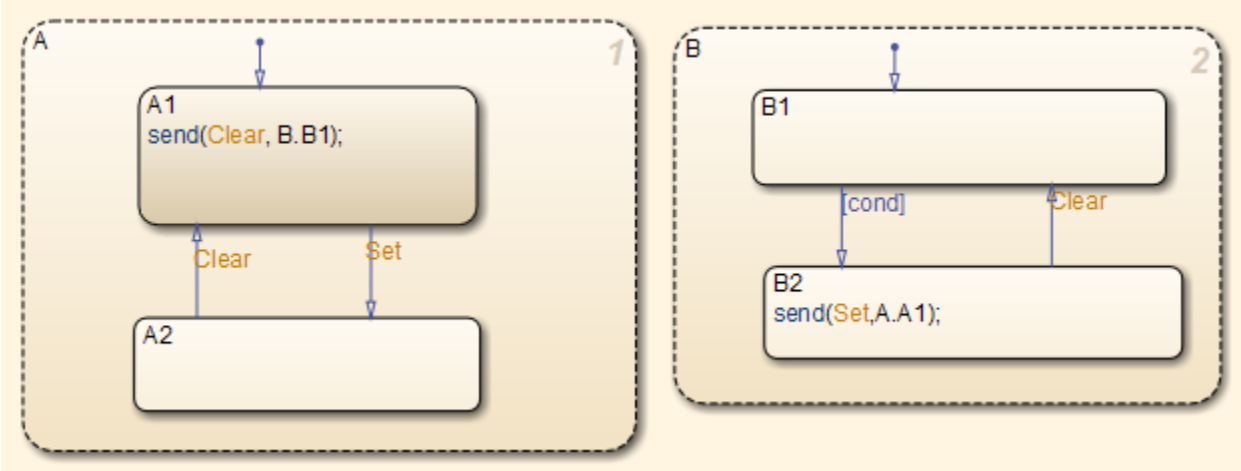
Note For information about cyclic behavior in Stateflow charts, see “Cyclic Behavior” (Stateflow).

However, you can modify a chart with cyclic behavior so that it is compatible, as in the following example.

The following chart creates cyclic behavior. State A calls state A1, which broadcasts a `Clear` event to state B, which calls state B2, which broadcasts a `Set` event back to state A, causing the cyclic behavior.



If you change the `send` function calls to use directed event broadcasts so that the Set and Clear events are broadcast directly to the states B1 and A1, respectively, the cyclic behavior disappears and the software can analyze the model.



Note For information about the benefits of directed event broadcasts, see “Broadcast Local Events to Synchronize Parallel States” (Stateflow).

Custom C/C++ Code

If your model consists of custom C/C++ code, Simulink Design Verifier supports analysis based on these settings:

- If you enable import custom code and custom code analysis options, the software supports custom C/C++ code for analysis. For more information, see “Import custom code” and “Enable custom code analysis”.
- If you enable import custom code option and the custom code analysis option is set to `Off`, the model is compatible for analysis, but calls to the custom code are stubbed during analysis.
- If the import custom code option is set to `Off`, the custom code is not supported and the model is incompatible for analysis.

Machine-Parented Data

The software does not support machine-parented data (i.e., defined at the level of the Stateflow machine), which the Stateflow software allows.

For more information, see “Best Practices for Using Data in Charts” (Stateflow).

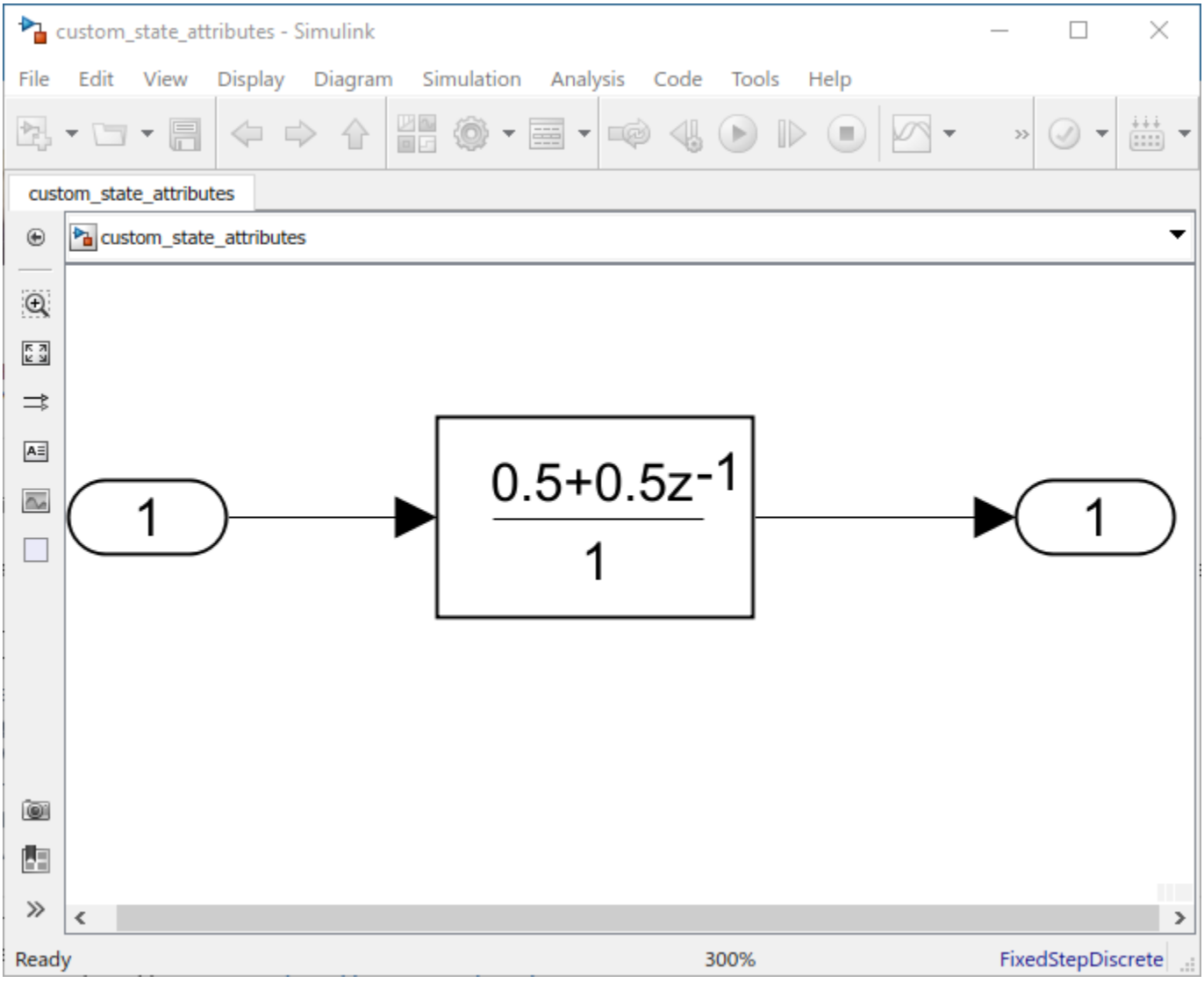
Textual Functions with Literal String Arguments

The software does not support literal string arguments to textual functions in a Stateflow chart.

Custom State Attributes in Discrete FIR Filter block

This example shows how to customize the state attributes of the Discrete FIR Filter block using the Model Data Editor. The Model Data Editor enables you to inspect and edit data items in a list that you can sort, group, and filter. For more information on using the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Consider a simple model that contains the Discrete FIR Filter block.

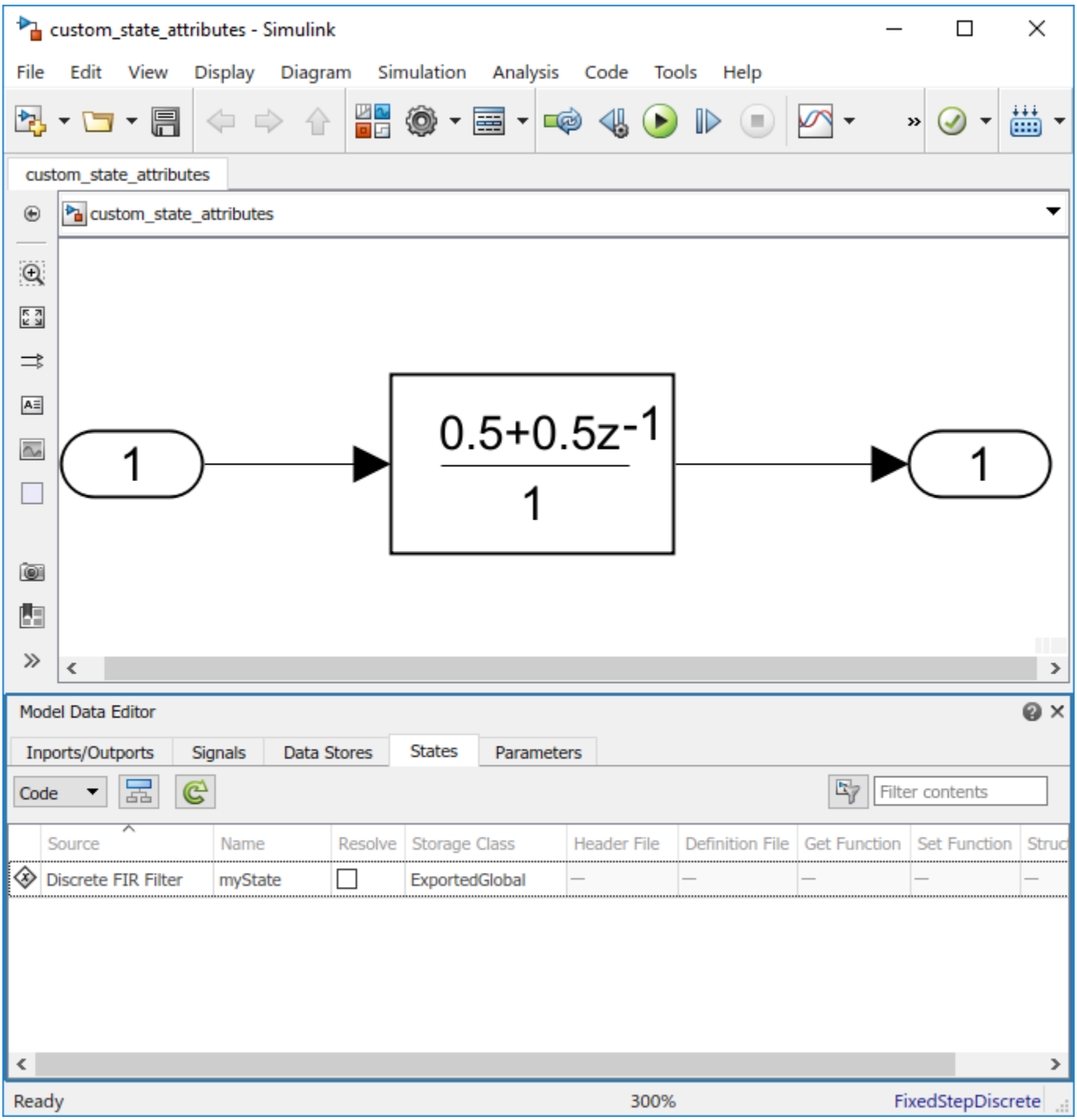


Using the Code Mappings editor or code mappings API, you can configure the state of the Discrete FIR Filter to appear in the generated code as a separate global variable. This is done by declaring the storage class of the state as `ExportedGlobal`. For details on how the generated code stores internal states, see “How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder). For more details on storage classes and how to apply them to the states, see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder).

Open Model Data Editor

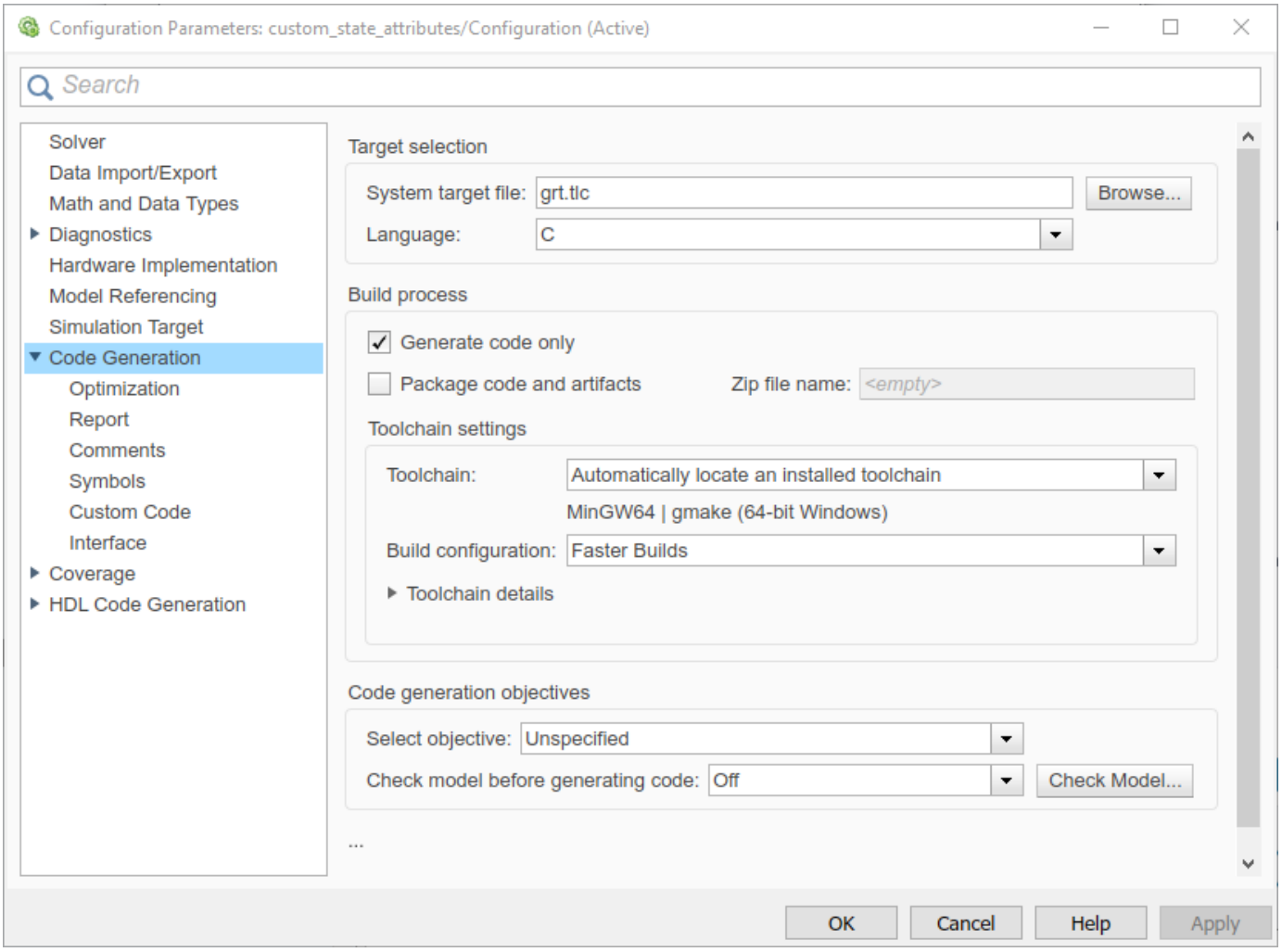
Open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**.

Under the **States** tab, enter the **Name** as `myState`. In the coder app, set the **Storage Class** to `ExportedGlobal`. You can alternatively select the **Resolve** check box, which requires the state name to resolve to a Simulink signal object.

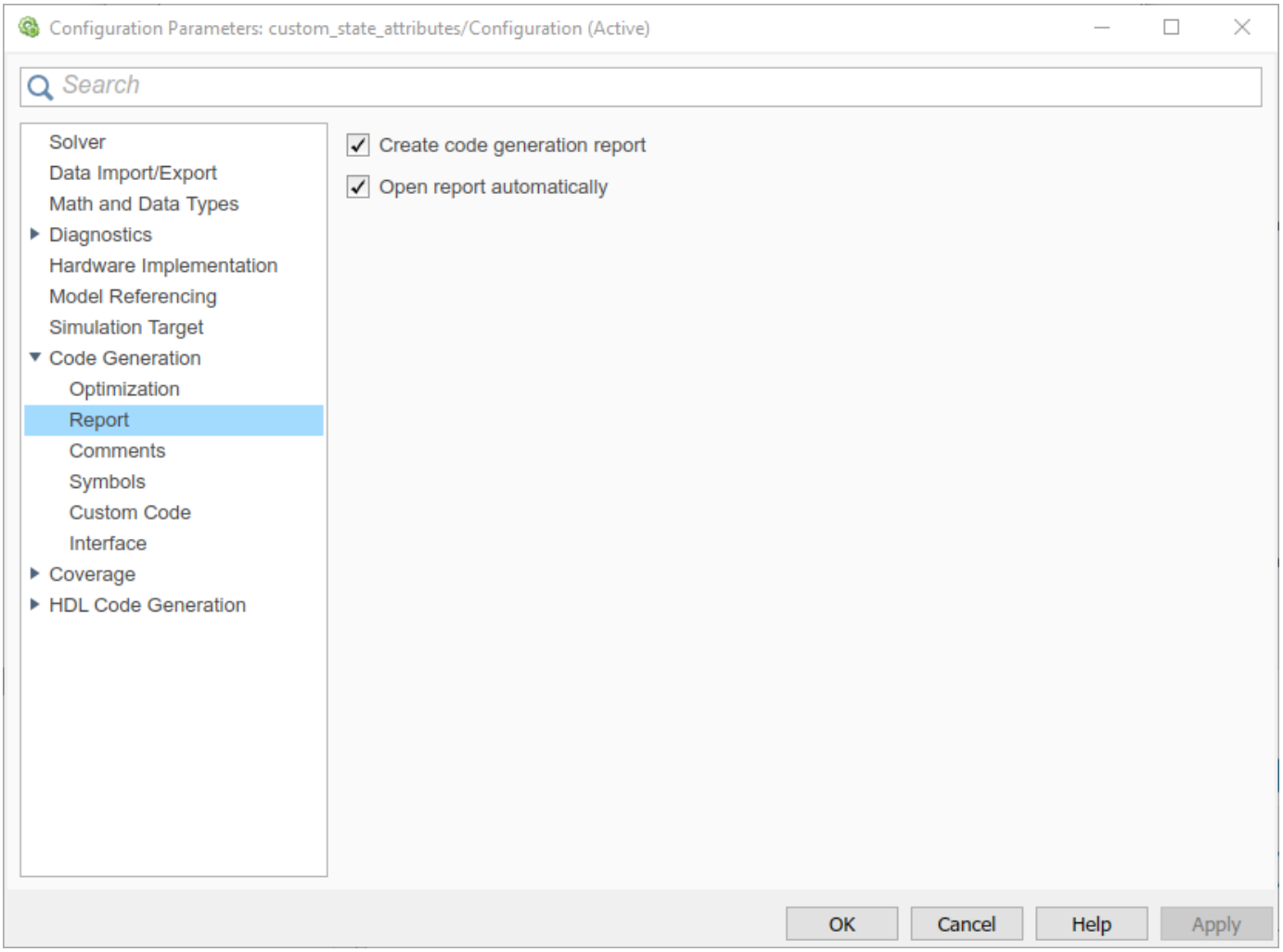


Build the Model and Inspect the Generated Code

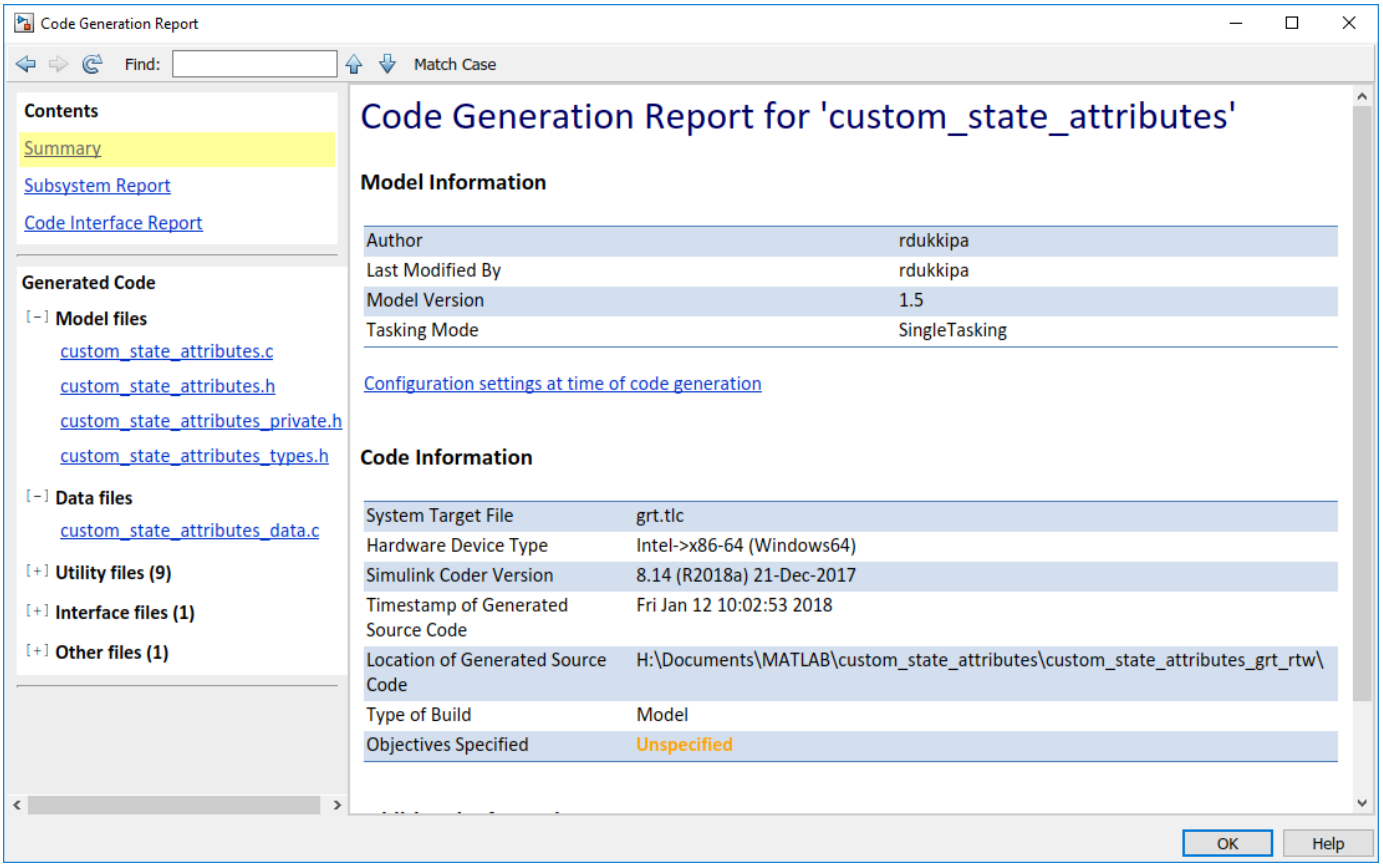
This example configures the model to generate code only. Open the Configuration Parameters by clicking the configuration button  in the Simulink editor. In the **Code Generation** pane, select **Generate code only**. Click **Apply**.



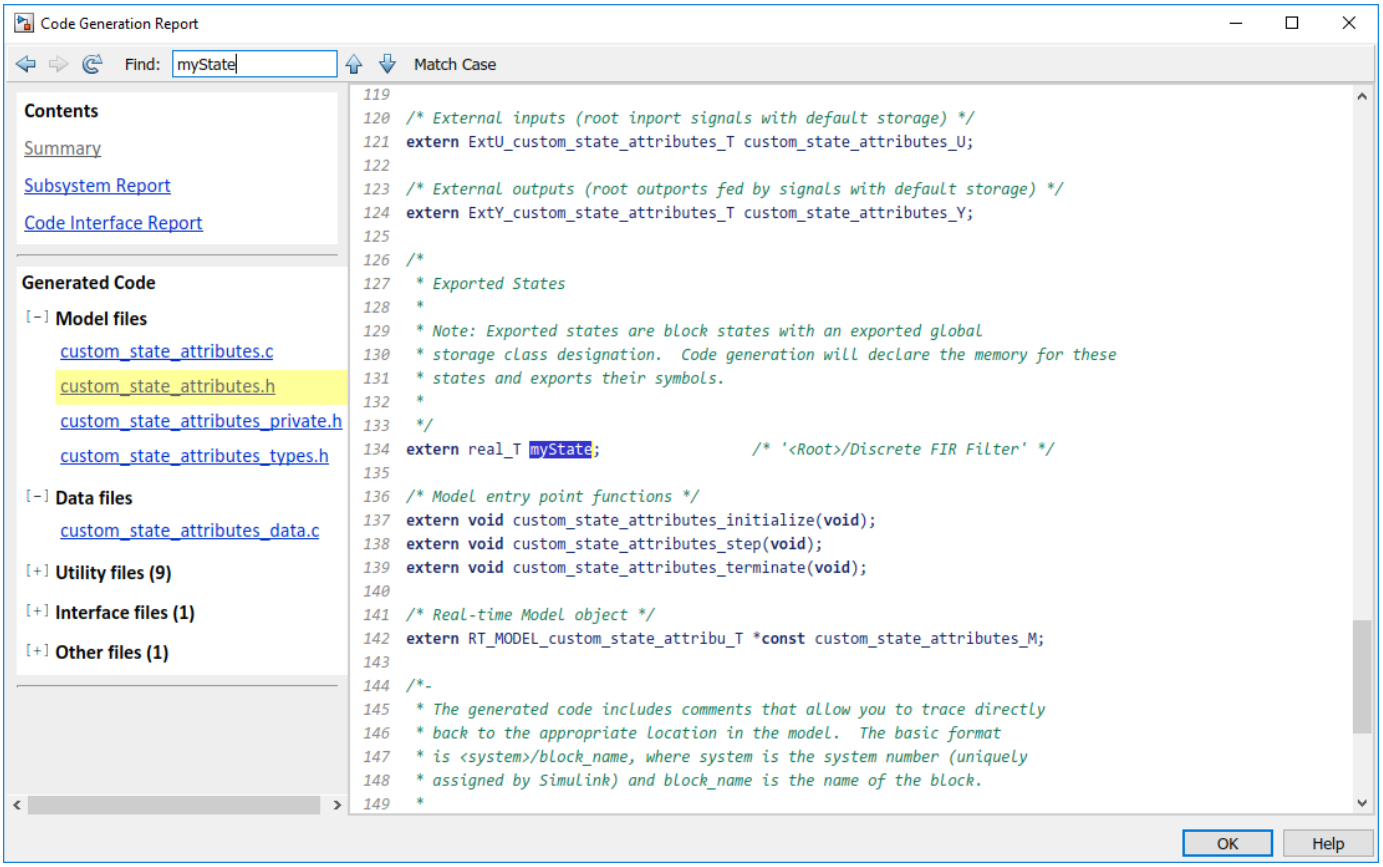
In the **Report** pane, select **Create code generation report** and **Open report automatically**. Click **Apply**. These settings create a report and automatically open the report in a web browser.



To initiate the build, click the build model button  in the Simulink editor or press **Ctrl+B**. The build process writes the code generation report files to the `html` subfolder of the build folder. Next, the build process automatically opens a MATLAB web browser window and displays the code generation report. Using this report, you can view and analyze the generated code. For more information on the generated report, see “Reports for Code Generation” (Simulink Coder).



In the `custom_state_attributes.h` file, you can see that the filter state is declared as an external variable since the storage class is `ExportedGlobal`.



If you change the storage class and rebuild the model, you can see the generated code reflect the change. With the ability to customize the state attributes, you can streamline and customize how the state appears in the generated code.

See Also

Blocks

Discrete FIR Filter

More About

- “Configure Data Properties by Using the Model Data Editor” on page 67-131
- “How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder)
- “C Code Generation Configuration for Model Interface Elements” (Simulink Coder)
- “Reports for Code Generation” (Simulink Coder)

Enumerations and Modeling

Simulink Enumerations

Enumerated data is data that is restricted to a finite set of values. An *enumerated data type* is a MATLAB class that defines a set of *enumerated values*. Each enumerated value consists of an *enumerated name* and an *underlying integer* which the software uses internally and in generated code.

Before you begin to use enumerations in a modeling context, you should understand information provided in “Enumerations”.

To define an enumeration for use in Simulink models, choose one of these techniques:

- Use the function `Simulink.defineIntEnumType`. The enumeration exists for the duration of your MATLAB session.
- Create a permanent enumeration class by subclassing one of these built-in classes:
  - Many of the built-in integer data types such as `int8` and `uint16`
  - `Simulink.IntEnumType`
- Use the function `Simulink.importExternalCTypes` to create a Simulink representation of an enumerated data type (`enum`) that your external C code defines.

  Use this technique to help you:

  - Replace existing C code with a Simulink model.
  - Integrate existing C code for simulation in Simulink (for example, by using the Legacy Code Tool).
  - Generate C code (Simulink Coder) that you can compile with existing C code into a single application.

For more information, see “Define Simulink Enumerations” on page 68-6.

The following examples show how to use enumerations in Simulink and Stateflow.

Example	Shows How To Use...
Data Typing in Simulink	Data types in Simulink, including enumerated data types
Modeling a CD Player/Radio Using Enumerated Data Types	Enumerated data types in a Simulink model that contains a Stateflow chart

For information on using enumerations in Stateflow, see “Enumerated Data” (Stateflow).

Simulink Constructs that Support Enumerations

Overview

In general, all Simulink tools and constructs support enumerated types for which the support makes sense given the purpose of enumerated types: to represent program states and to control program logic. The Simulink Editor, Simulink Debugger, Port Value Displays, referenced models, subsystems, masks, buses, data logging, and most other Simulink capabilities support enumerated types without imposing any special requirements.

Enumerated types are not intended for mathematical computation, so no block that computes a numeric output (as distinct from passing a numeric input through to the output) supports enumerated types. Thus an enumerated type is not considered to be a numeric type, even though an enumerated value has an underlying integer. See “Enumerated Values in Computation” on page 68-17 for more information.

Most capabilities that do not support enumerated types obviously could not support them. Therefore, the Simulink documentation usually mentions enumerated type nonsupport only where necessary to prevent a misconception or describe an exception. See “Simulink Enumeration Limitations” on page 68-4 for information about certain constructs that could support enumerated types but do not.

Block Support

The following Simulink blocks support enumerated types:

- Constant (but Enumerated Constant is preferable)
- Data Type Conversion
- Data Type Conversion Inherited
- Data Type Duplicate
- Display
- MATLAB Function
- Enumerated Constant
- Floating Scope
- From File
- From Workspace
- Inport
- Interval Test
- Interval Test Dynamic
- Multiport Switch
- Outport
- Probe (input only)
- Relational Operator
- Relay (output only)
- Repeating Sequence Stair
- Scope
- Signal Specification
- Switch

- Switch Case
- To File
- To Workspace

All members of the following categories of Simulink blocks support enumerated types:

- Bus-capable blocks (see “Bus-Capable Blocks” on page 76-36)
- Pass-through blocks:
  - With state, like the Data Store Memory and Unit Delay blocks.
  - Without state, like the Mux block.

Many Simulink blocks in addition to those named above support enumerated types, but they either belong to one of the categories listed above, or are rarely used with enumerated types. The Data Type Support section of each block reference page describes all data types that the block supports.

Class Support

The following Simulink classes support enumerated types:

- `Simulink.Signal`
- `Simulink.Parameter`
- `Simulink.AliasType`
- `Simulink.BusElement`

Logging Enumerated Data

Top-level model output ports, To Workspace blocks, and Scope blocks can all export enumerated values. Signal and State logging work with enumerated data in the same way as with any other data. All logging formats are supported. The From File block does not support enumerated data. Use the From Workspace block instead, combined with some technique for transferring data between a file and a workspace. See “Save Run-Time Data from Simulation” for more information.

Importing Enumerated Data

Top-level model input ports and From Workspace blocks can output enumerated signals during simulation. Data must be provided in a `Structure`, `Structure with Time`, or `TimeSeries` object. No interpolation occurs for enumerated values between the specified simulation times. From File blocks produce only data of type `double`, so they do not support enumerated types. See “Load Signal Data for Simulation” for more information.

Simulink Enumeration Limitations

Enumerations and Scopes

When a Scope block displays an enumerated signal, the vertical axis displays the names of the enumerated values only if the scope was open during simulation. If you open the Scope block for the

first time before any simulation has occurred, or between simulations, the block displays only numeric values. When simulation begins, enumerated names replace the numeric values, and thereafter appear whenever the Scope block is opened.

When a Floating Scope block displays multiple signals, the names of enumerated values appear on the Y axis only if all signals are of the same enumerated type. If the Floating Scope block displays more than one type of enumerated signal, or any numeric signal, no names appear, and any enumerated values are represented by their underlying integers.

Enumerated Types for Switch Blocks

The control input of a Switch block can be of any data type supported by Simulink. However, the `u2 ~=0` mode is not supported for enumerations. If the control input has an enumeration, choose one of the following methods to specify the criteria for passing the first input:

- Select `u2 >= Threshold` or `u2 > Threshold` and specify a threshold value of the same enumerated type as the control input.
- Use a Relational Operator block to do the comparison and then feed the Boolean result of this comparison into the control port of the Switch block.

Nonsupport of Enumerations

The following limitations exist when using enumerated data types with Simulink:

- Packages cannot contain enumeration class definitions.
- The If Action block does not support enumerations.
- Generated code does not support logging enumerated data.
- Custom Stateflow targets do not support enumerated types.

See Also

`Simulink.data.getEnumTypeInfo` | `Simulink.defineIntEnumType` | `enumeration`

Related Examples

- “Use Enumerated Data in Simulink Models” on page 68-6
- “Define Enumerations for MATLAB Function Blocks” on page 44-84
- “Use Enumerated Data in Generated Code” (Simulink Coder)
- “Manipulate Enumerations in Data Dictionary” on page 74-14

Use Enumerated Data in Simulink Models

In this section...

Enumerated data is data that is restricted to a finite set of values. An *enumerated data type* is a MATLAB class that defines a set of *enumerated values*. Each enumerated value consists of an *enumerated name* and an *underlying integer* which the software uses internally and in generated code.

For basic conceptual information about enumerations in Simulink, see “Simulink Enumerations” on page 68-2.

For information about generating code with enumerations, see “Use Enumerated Data in Generated Code” (Simulink Coder).

Define Simulink Enumerations

To define an enumerated data type that you can use in Simulink models, use one of these methods:

- Define an enumeration class using a `classdef` block in a MATLAB file.
- Use the function `Simulink.defineIntEnumType`. You do not need a script file to define the type. For more information, see the function reference page.
- Use the function `Simulink.importExternalCTypes` to create a Simulink representation of an enumerated data type (`enum`) that your external C code defines.

Workflow to Define a Simulink Enumeration Class

1. Create a class definition on page 68-6.
2. Optionally, customize the enumeration on page 68-7.
3. Optionally, save the enumeration in a MATLAB file on page 68-9.
4. Optionally, permanently store the enumeration definition in a Simulink data dictionary. See “Permanently Store Enumerated Type Definition” on page 68-11.

Create Simulink Enumeration Class

To create a Simulink enumeration class, in the class definition:

- Define the class as a subclass of `Simulink.IntEnumType`. You can also base an enumerated type on one of these built-in integer data types: `int8`, `uint8`, `int16`, `uint16`, and `int32`.
- Add an `enumeration` block that specifies enumeration values with underlying integer values.

Consider the following example:

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
end
```

The first line defines an integer-based enumeration that is derived from built-in class `Simulink.IntEnumType`. The enumeration is integer-based because `IntEnumType` is derived from `int32`.

The `enumeration` section specifies three enumerated values.

Enumerated Value	Enumerated Name	Underlying Integer
`Red(0)`	`Red`	`0`
`Yellow(1)`	`Yellow`	`1`
`Blue(2)`	`Blue`	`2`

When defining an enumeration class for use in the Simulink environment, consider the following:

- The name of the enumeration class must be unique among data type names and base workspace variable names, and is case-sensitive.
- Underlying integer values in the `enumeration` section need not be unique within the class and across types.
- Often, the underlying integers of a set of enumerated values are consecutive and monotonically increasing, but they need not be either consecutive or ordered.
- For simulation, an underlying integer can be any `int32` value. Use the MATLAB functions `intmin` and `intmax` to get the limits.
- For code generation, every underlying integer value must be representable as an integer on the target hardware, which may impose different limits. See “Configure a System Target File” (Simulink Coder) for more information.

For more information on superclasses, see “Convert to Superclass Value”. For information on how enumeration classes are handled when there is more than one name for an underlying value, see “How to Alias Enumeration Names”.

Customize Simulink Enumeration

About Simulink Enumeration Customizations

You can customize a Simulink enumeration by implementing specific static methods in the class definition. If you define these methods using the appropriate syntax, you can change the behavior of the class during simulation and in generated code.

The table shows the methods you can implement to customize an enumeration.

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value	Usage Context
`getDefaultValue`	Specifies the default enumeration member for the class.	First member specified in the enumeration definition	A character vector containing the name of an enumeration member in the class (see “Instantiate Enumerations” on page 68-15)	Simulation and code generation
`getDescription`	Specifies a description of the enumeration class.	`''`	A character vector containing the description of the type	Code generation
`getHeaderFile`	Specifies the name of a header file. The method `getDataScope` determines the significance of the file.	`''`	A character vector containing the name of the header file that defines the enumerated type	Code generation
`getDataScope`	Specifies whether generated code exports or imports the definition of the enumerated data type. Use the method `getHeaderFile` to specify the generated or included header file that defines the type.	`'Auto'`	One of: `'Auto'`, `'Exported'`, or `'Imported'`	Code generation
`addClassNameToEnumNames`	Specifies whether to prefix the class name in generated code.	`false`	`true` or `false`	Code generation

For more examples of these methods as they apply to code generation, see “Customize Enumerated Data Type” (Simulink Coder).

Specify a Default Enumerated Value

Simulink and related generated code use an enumeration's default value for ground-value initialization of enumerated data when you provide no other initial value. For example, an enumerated signal inside a conditionally executed subsystem that has not yet executed has the enumeration's default value. Generated code uses an enumeration's default value if a safe cast fails, as described in “Type Casting for Enumerations” (Simulink Coder).

Unless you specify otherwise, the default value for an enumeration is the first value in the enumeration class definition. To specify a different default value, add your own `getDefaultValue` method to the `methods` section. The following code shows a shell for the `getDefaultValue` method:

```
function retVal = getDefaultValue()
  % GETDEFAULTVALUE Specifies the default enumeration member.
  % Return a valid member of this enumeration class to specify the default.
  % If you do not define this method, Simulink uses the first member.
  retVal = ThisClass.EnumName;
end
```

To customize this method, provide a value for *`ThisClass.EnumName`* that specifies the desired default.

- *`ThisClass`* must be the name of the class within which the method exists.
- *`EnumName`* must be the name of an enumerated value defined in that class.

For example:

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
  methods (Static)
    function retVal = getDefaultValue()
      retVal = BasicColors.Blue;
    end
  end
end
```

This example defines the default as `BasicColors.Blue`. If this method does not appear, the default value would be `BasicColors.Red`, because that is the first value listed in the enumerated class definition.

The seemingly redundant specification of *`ThisClass`* inside the definition of that same class is necessary because `getDefaultValue` returns an instance of the default enumerated value, not just the name of the value. The method, therefore, needs a complete specification of what to instantiate. See “Instantiate Enumerations” on page 68-15 for more information.

Save Enumeration in a MATLAB File

You can define an enumeration within a MATLAB file.

- The name of the definition file must match the name of the enumeration exactly, including case. For example, the definition of enumeration `BasicColors` must reside in a file named `BasicColors.m`. Otherwise, MATLAB will not find the definition.
- You must define each class definition in a separate file.
- Save each definition file on the MATLAB search path. MATLAB searches the path to find a definition when necessary.

  To add a file or folder to the MATLAB search path, type `addpath` *`pathname`* at the MATLAB command prompt. For more information, see “What Is the MATLAB Search Path?”, `addpath`, and `savepath`.
- You do not need to execute an enumeration class definition to use the enumeration. The only requirement, as indicated in the preceding bullet, is that the definition file be on the MATLAB search path.

Change and Reload Enumeration Classes

You can change the definition of an enumeration by editing and saving the file that contains the definition. You do not need to inform MATLAB that a class definition has changed. MATLAB automatically reads the modified definition when you save the file. However, the class definition changes do not take full effect if any class instances (enumerated values) exist that reflect the previous class definition. Such instances might exist in the base workspace or might be cached.

The following table explains options for removing instances of an enumeration from the base workspace and cache.

If In Base Workspace...	If In Cache...
Do one of the following: • Locate and delete specific obsolete instances. • Delete everything from the workspace by using the `clear` command.	• Delete obsolete instances by closing all models that you updated or simulated while the previous class definition was in effect. • Clear functions and close models that are caching instances of the class.

Similarly, if you defined an enumeration class by using `Simulink.defineIntEnumType`, you can redefine that class, using the same function, even if instances exist. However, you cannot change `StorageType` for the class while instances exist.

For more information about applying enumeration changes, see “Automatic Updates for Modified Classes”.

Import Enumerations Defined Externally to MATLAB

If you have enumerations defined externally to MATLAB that you want to import for use within the Simulink environment, you can do so programmatically with calls to one of these functions:

- `Simulink.defineIntEnumType` — Defines an enumeration that you can use in MATLAB as if it is defined by a class definition file. In addition to specifying the enumeration class name and values, each function call can specify:

  - Character vector that describes the enumeration class.
  - Which of the enumeration values is the default.

  For code generation, you can specify:

  - Header file in which the enumeration is defined for generated code.
  - Whether the code generator applies the class name as a prefix to enumeration members — for example, `BasicColors_Red` or `Red`.

  As an example, consider the following class definition:

```
classdef BasicColors < Simulink.IntEnumType
    enumeration
        Red(0)
        Yellow(1)
        Blue(2)
    end
    methods (Static = true)
        function retVal = getDescription()
            retVal = 'Basic colors...';
        end
        function retVal = getDefaultValue()
            retVal = BasicColors.Blue;
        end
        function retVal = getHeaderFile()
            retVal = 'mybasiccolors.h';
        end
        function retVal = addClassNameToEnumNames()
```

```
            retVal = true;
        end
    end
end
```

The following function call defines the same class for use in MATLAB:

```
Simulink.defineIntEnumType('BasicColors', ...
     {'Red', 'Yellow', 'Blue'}, [0;1;2],...
     'Description', 'Basic colors', ...
     'DefaultValue', 'Blue', ...
     'HeaderFile', 'mybasiccolors.h', ...
     'DataScope', 'Imported', ...
     'AddClassNameToEnumNames', true);
```

- `Simulink.importExternalCTypes` — Creates Simulink representations of enumerated data types (`enum`) that your existing C code defines.

If a MATLAB Function block in your model uses the enumerated type, configure the model configuration parameters to include (`#include`) the type definition from your external header file. See “Control Imported Bus and Enumeration Type Definitions” on page 44-124.

Permanently Store Enumerated Type Definition

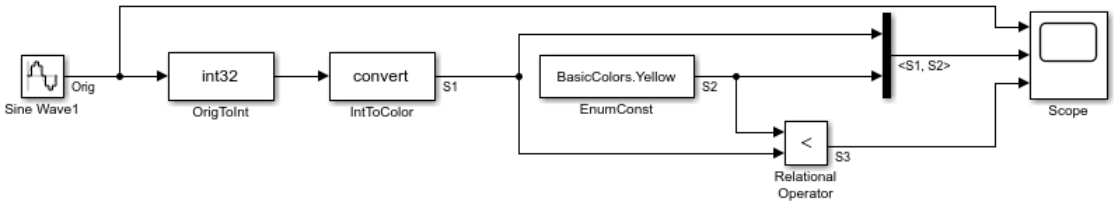
Whether you define an enumeration by using a class file or by using the function `Simulink.defineIntEnumType`, you can permanently store the enumeration definition in a Simulink data dictionary. Models that are linked to the dictionary can use the enumeration. For more information, see “Enumerations in Data Dictionary” on page 74-12.

Simulate with Enumerations

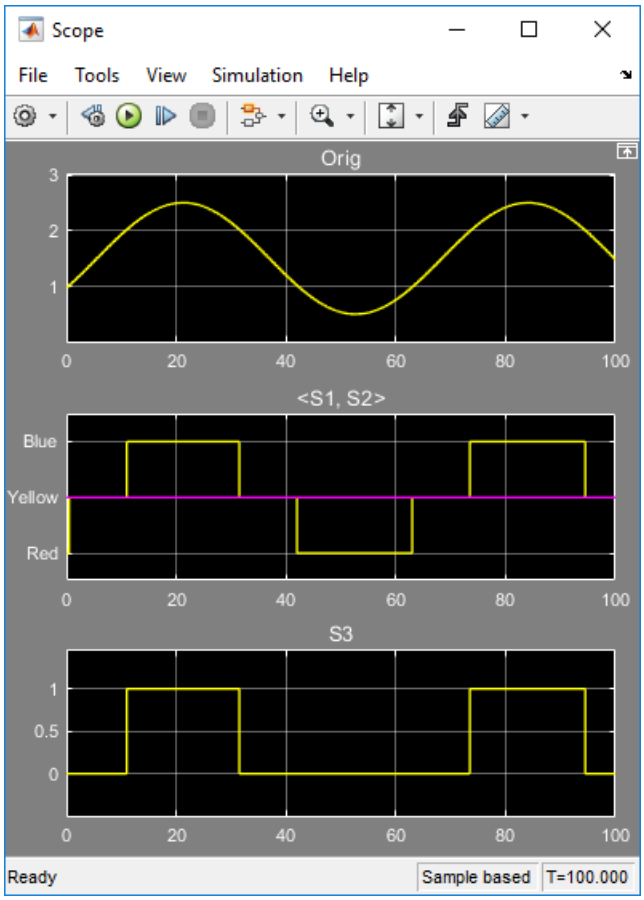
Consider the following enumeration class definition — `BasicColors` with enumerated values `Red`, `Yellow`, and `Blue`, with `Blue` as the default value:

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
  methods (Static)
    function retVal = getDefaultValue()
      retVal = BasicColors.Blue;
    end
  end
end
```

Once this class definition is known to MATLAB, you can use the enumeration in Simulink and Stateflow models. Information specific to enumerations in Stateflow appears in “Enumerated Data” (Stateflow). The following Simulink model uses the enumeration defined above:



The output of the model looks like this:



The Data Type Conversion block OrigToInt specifies an **Output data type** of `int32` and **Integer rounding mode:** `Floor`, so the block converts the Sine Wave block output, which appears in the top graph of the Scope display, to a cycle of integers: `1`, `2`, `1`, `0`, `1`, `2`, `1`. The Data Type Conversion block IntToColor uses these values to select colors from the enumerated type `BasicColors` by referencing their underlying integers.

The result is a cycle of colors: `Yellow`, `Blue`, `Yellow`, `Red`, `Yellow`, `Blue`, `Yellow`, as shown in the middle graph. The Enumerated Constant block EnumConst outputs `Yellow`, which appears in the second graph as a straight line. The Relational Operator block compares the constant `Yellow` to each value in the cycle of colors. It outputs `1` (`true`) when `Yellow` is less than the current color, and `0` (`false`) otherwise, as shown in the third graph.

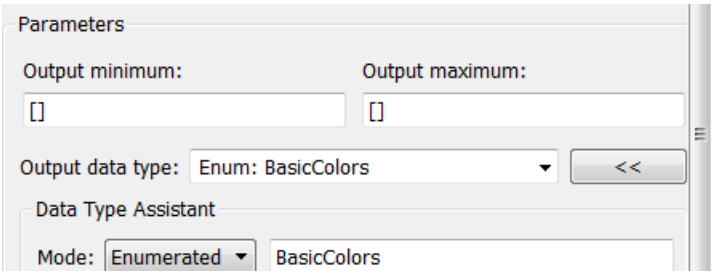
The sort order used by the comparison is the numeric order of the underlying integers of the compared values, *not* the lexical order in which the enumerated values appear in the enumerated class definition. In this example the two orders are the same, but they need not be. See “Specify Enumerations as Data Types” on page 68-13 and “Enumerated Values in Computation” on page 68-17 for more information.

Specify Enumerations as Data Types

Once you define an enumeration, you can use it much like any other data type. Because an enumeration is a class rather than an instance, you must use the prefix ? or `Enum:` when specifying the enumeration as a data type. You must use the prefix `?` in the MATLAB Command Window. However, you can use either prefix in a Simulink model. `Enum:` has the same effect as the `?` prefix, but `Enum:` is preferred because it is more self-explanatory in the context of a graphical user interface.

Depending on the context, type `Enum:` followed by the name of an enumeration, or select `Enum: <class name>` from a menu (for example, for the **Output data type** block parameter) , and replace `<class name>`.

To use the Data Type Assistant, set the **Mode** to `Enumerated`, then enter the name of the enumeration. For example, in the previous model, the Data Type Conversion block IntToColor, which outputs a signal of type `BasicColors`, has the following output signal specification:



You cannot set a minimum or maximum value for a signal defined as an enumeration, because the concepts of minimum and maximum are not relevant to the purpose of enumerations. If you change the minimum or maximum for a signal of an enumeration from the default value of `[]`, an error occurs when you update the model. See “Enumerated Values in Computation” on page 68-17 for more information.

Get Information About Enumerated Data Types

The functions `enumeration` and `Simulink.data.getEnumTypeInfo` return information about enumerated data types.

Get Information About Enumeration Members

Use the function `enumeration` to:

- Return an array that contains all enumeration values for an enumeration class in the MATLAB Command Window
- Get the enumeration values programmatically
- Provide the values to a Simulink block parameter that accepts an array or vector of enumerated values, such as the **Case conditions** parameter of the Switch Case block

Get Information About Enumerated Class

Use the function `Simulink.data.getEnumTypeInfo` to return information about an enumeration class, such as:

- The default enumeration member
- The name of the header file that defines the type in generated code
- The data type used in generated code to store the integer values underlying the enumeration members

Enumeration Value Display

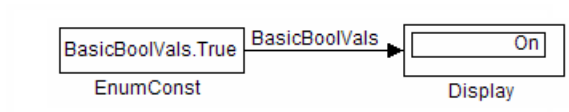
Wherever possible, Simulink displays enumeration values by name, not by the underlying integer value. However, the underlying integers can affect value display in Scope and Floating Scope blocks.

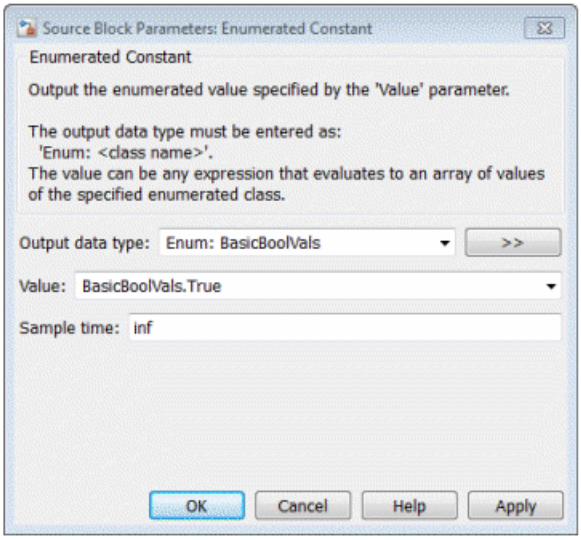
Block...	Affect on Value Display...
Scope	When displaying an enumerated signal, the names of the enumerated values appear as labels on the Y axis. The names appear in the order given by their underlying integers, with the lowest value at the bottom.
Floating Scope	When displaying signals that are of the same enumeration, names appear on the Y axis as they would for a Scope block. If the Floating Scope block displays mixed data types, no names appear, and any enumerated values are represented by their underlying integers.

Enumerated Values with Non-Unique Integers

More than one value in an enumeration can have the same underlying integer value, as described in “Specify Enumerations as Data Types” on page 68-13. When this occurs, the value on an axis of Scope block output or in Display block output always is the first value listed in the enumerated class definition that has the shared underlying integer. For example:

```
classdef BasicBoolVals < Simulink.IntEnumType
  enumeration
    On(1)
    Off(0)
    True(1)
    False(0)
  end
end
```





Although the Enumerated Constant block outputs `True`, both `On` and `True` have the same underlying integer, and `On` is defined first in the class definition `enumeration` section. Therefore, the Display block shows `On`. Similarly, a Scope axis would show only `On`, never `True`, no matter which of the two values is input to the Scope block.

Instantiate Enumerations

Before you can use an enumeration, you must instantiate it. You can instantiate an enumeration in MATLAB, in a Simulink model, or in a Stateflow chart. The syntax is the same in all contexts.

Instantiating Enumerations in MATLAB

To instantiate an enumeration in MATLAB, enter *`ClassName.EnumName`* in the MATLAB Command Window. The instance is created in the base workspace. For example, if `BasicColors` is defined as in “Create Simulink Enumeration Class” on page 68-6, you can type:

```
bcy = BasicColors.Yellow

bcy =

    Yellow
```

Tab completion works for enumerations. For example, if you enter:

```
bcy = BasicColors.<tab>
```

MATLAB displays the elements and methods of `BasicColors` in alphabetical order:

```
addClassNameToEnumNames
Blue
getDefaultValue
getDescription
getHeaderFile
getStorageType
Red
Yellow
```

Double-click an element or method to insert it at the position where you pressed <tab>. See “Code Suggestions and Completions” for more information.

Casting Enumerations in MATLAB

In MATLAB, you can cast directly from an integer to an enumerated value:

```
bcb = BasicColors(2)

bcb =

    Blue
```

You can also cast from an enumerated value to its underlying integer:

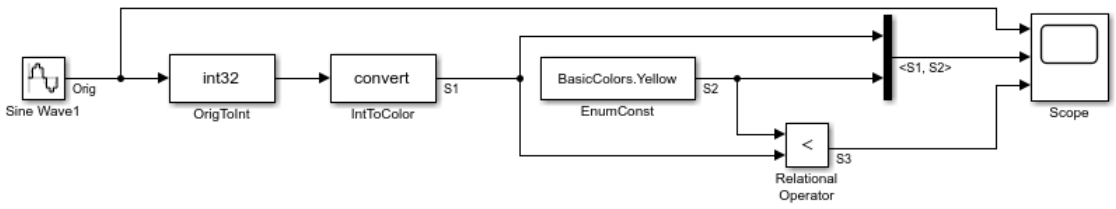
```
>> bci = int32(bcb)

bci =

    2
```

In either case, MATLAB returns the result of the cast in a 1x1 array of the relevant data type.

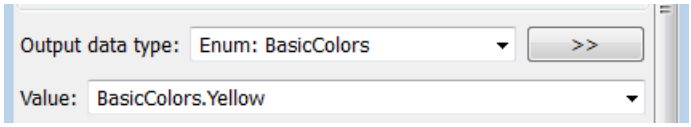
Although casting is possible, use of enumeration values is not robust in cases where enumeration values and the integer equivalents defined for an enumeration class might change.

Instantiating Enumerations in Simulink (or Stateflow)

To instantiate an enumeration in a Simulink model, you can enter *`ClassName.EnumName`* as a value in a dialog box. For example, consider the following model:



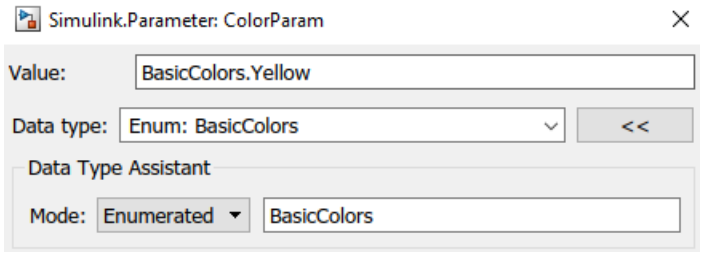
The Enumerated Constant block EnumConst, which outputs the enumerated value `Yellow`, defines that value as follows:



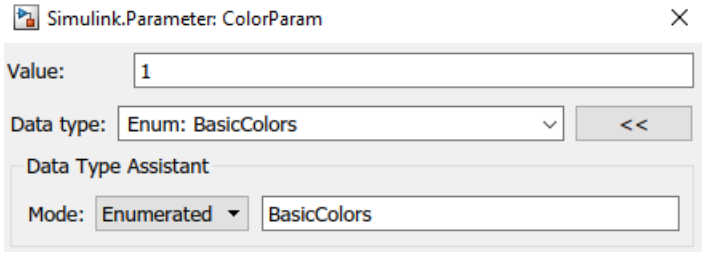
You can enter any valid MATLAB expression that evaluates to an enumerated value, including arrays and workspace variables. For example, you could enter `BasicColors(1)`, or if you had previously executed `bcy = BasicColors.Yellow` in the MATLAB Command Window, you could enter `bcy`. As another example, you could enter an array, such as `[BasicColors.Red, BasicColors.Yellow, BasicColors.Blue]`.

You can use a Constant block to output enumerated values. However, that block displays parameters that do not apply to enumerated types, such as **Output Minimum** and **Output Maximum**.

If you create a `Simulink.Parameter` object as an enumeration, you must specify the **Value** parameter as an enumeration member and the **Data type** with the `Enum:` or ? prefix, as explained in “Specify Enumerations as Data Types” on page 68-13.



You *cannot* specify the integer value of an enumeration member for the **Value** parameter. See “Enumerated Values in Computation” on page 68-17 for more information. Thus, the following fails even though the integer value for `BasicColors.Yellow` is `1`.



The same syntax and considerations apply in Stateflow. See “Enumerated Data” (Stateflow) for more information.

Enumerated Values in Computation

By design, Simulink prevents enumerated values from being used as numeric values in mathematical computation, even though an enumerated class is a subclass of the MATLAB `int32` class. Thus, an enumerated type does not function as a numeric type despite the existence of its underlying integers. For example, you cannot input an enumerated signal directly to a Gain block.

You can use a Data Type Conversion block to convert in either direction between an integer type and an enumerated type, or between two enumerated types. That is, you can use a Data Type Conversion block to convert an enumerated signal to an integer signal (consisting of the underlying integers of

the enumerated signal values) and input the resulting integer signal to a Gain block. See “Casting Enumerated Signals” on page 68-18 for more information.

Enumerated types in Simulink are intended to represent program states and control program logic in blocks like the Relational Operator block and the Switch block. When a Simulink block compares enumerated values, the values compared must be of the same enumerated type. The block compares enumerated values based on their underlying integers, not their order in the enumerated class definition.

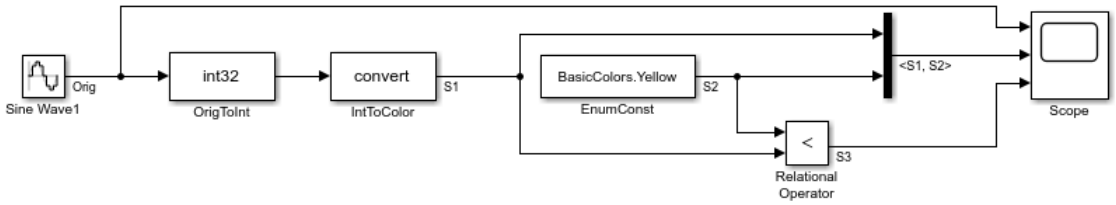
When a block like the Switch block or Multiport Switch block selects among multiple data signals, and any data signal is of an enumerated type, all the data signals must be of that same enumerated type. When a block inputs both control and data signals, as Switch and Multiport Switch do, the control signal type need not match the data signal type.

Casting Enumerated Signals

You can use a Data Type Conversion block to cast an enumerated signal to a signal of any numeric type, provided that the underlying integers of all enumerated values input to the block are within the range of the numeric type. Otherwise, an error occurs during simulation.

Similarly, you can use a Data Type Conversion block to cast a signal of any integer type to an enumerated signal, provided that every value input to the Data Type Conversion block is the underlying integer of some value in the enumerated type. Otherwise, an error occurs during simulation.

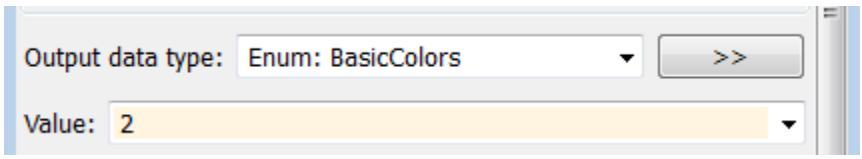
You cannot use a Data Type Conversion block to cast a numeric signal of any non-integer data type to an enumerated type. For example, the model used in “Simulate with Enumerations” on page 68-11 needed two Data Conversion blocks to convert a sine wave to enumerated values.



The first block casts `double` to `int32`, and the second block casts `int32` to `BasicColors`. You cannot cast a complex signal to an enumerated type regardless of the data types of its real and imaginary parts.

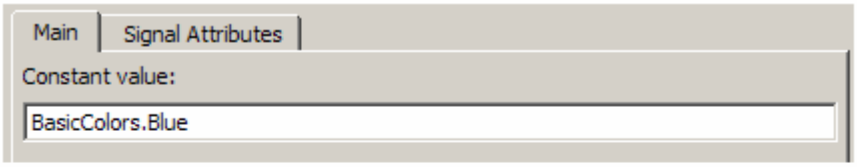
Casting Enumerated Block Parameters

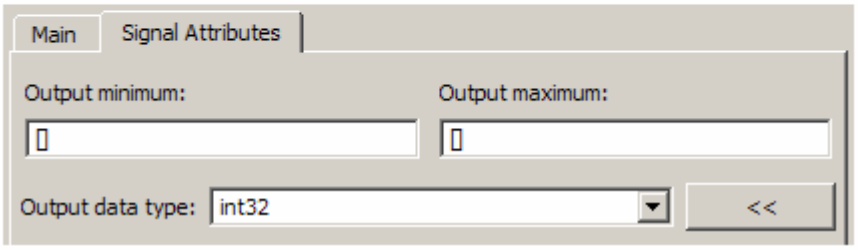
You cannot cast a block parameter of any numeric data type to an enumerated data type. For example, suppose that an Enumerated Constant block specifies a **Value** of 2 and an **Output data type** of `Enum: BasicColors`:



An error occurs because the specifications implicitly cast a `double` value to an enumerated type. The error occurs even though the numeric value corresponds arithmetically to one of the enumerated values in the enumerated type.

You cannot cast a block parameter of an enumeration to any other data type. For example, suppose that a Constant block specifies a **Constant value** of `BasicColors.Blue` and an **Output data type** of `int32`.





An error occurs because the specifications implicitly cast an enumerated value to a numeric type. The error occurs even though the enumerated value's underlying integer is a valid `int32`.

See Also

`Simulink.data.getEnumTypeInfo` | `Simulink.defineIntEnumType` | `enumeration`

Related Examples

- “Define Enumerations for MATLAB Function Blocks” on page 44-84
- “Define Enumerated Data Types” (Stateflow)
- “Use Enumerated Data in Generated Code” (Simulink Coder)
- “Simulink Enumerations” on page 68-2
- “Manipulate Enumerations in Data Dictionary” on page 74-14

Create Data to Use as Simulation Input

Create and Edit Signal Data

In this section...

Use the Signal Editor to create and edit input signals that you can organize for multiple simulations. You can then save the signal data to a MAT-file for simulation or to map to root-level ports. You can access the Signal Editor in the following ways:

- `signalEditor` function — Signal Editor starts from the command line.
- From the Root Inport Mapper on page 71-7 — To create a MAT-file for your new signal data, select **Signals > New MAT-File**. To link in an existing signal data file from an existing scenario and edit the signals in that file, use the **Signals > Edit MAT-File**.
- From the Signal Editor block

Signal Editor works only with MAT-files.

You can manipulate signals in these ways:

- Create and edit multiple signals in multiple data sets.
- Use signal notations to create more complicated signals using MATLAB expressions.
- Use existing scenarios to get existing data sets for which you can edit and create signals.
- Create and edit multidimensional signals.
- Edit signals imported as registered custom file types. For more information, see “Import Custom File Type” on page 69-34.

While editing signal data:

- Use tabular editing or MATLAB to modify signal data.
- Modify signal properties such as name, interpolation, and unit properties.
- Drag and drop signals to change signal hierarchies for buses and data sets.
- Use signal notations and variables to replace signal data.

Alternatively, you can import data from external sources and edit them in Signal Editor. For more information, see “Link in Signal Data from Signal Builder Block and Simulink Design Verifier Environment” on page 69-15.

Differences Between the Root Inport Mapper Signal Editor and Other Signal Editors

Generally, the Signal Editor user interface is the same regardless of how you access it. Here are the differences in the Root Inport Mapper Signal Editor:

- FILE section **Save and Sync** and **SAVE** commands save and synchronize to the Root Inport Mapper.
- **Insert** section **Scenario** command always has the option, **Scenario from Model**.

If you start the function with a model name, the `signalEditor` function Signal Editor user interface shows the option **Scenario from Model** in the **Insert** section.

Table Editing Data Support

The Signal Editor user interface supports all signal data types that Simulink supports and that are editable.

Mouse, Keyboard, and Touchscreen Shortcuts

Edit actions:

Action	Keyboard	Mouse	Multi-Touch
Insert point	Ctrl+P	Click 	Tap
Insert line	Ctrl+L	Click 	Pan and pinch
Draw	Ctrl+D	Click and draw	Pan
Select a point	Ctrl+T	Click  , then click and select point or area To select all areas, double-click	Pan and pinch To select all areas, double-tap
Move a point	Ctrl+M	Click  and drag	Tap and move
Change data of a point		Click 	
Continuously delete points on a line		Click 	
		Three mouse clicks	Triple tap
Expand along the *x*-axis		**Ctrl**+mouse pan	Pry *x*-axis
Expand along the *y*-axis		**Shift**+mouse pan	Pry *y*-axis

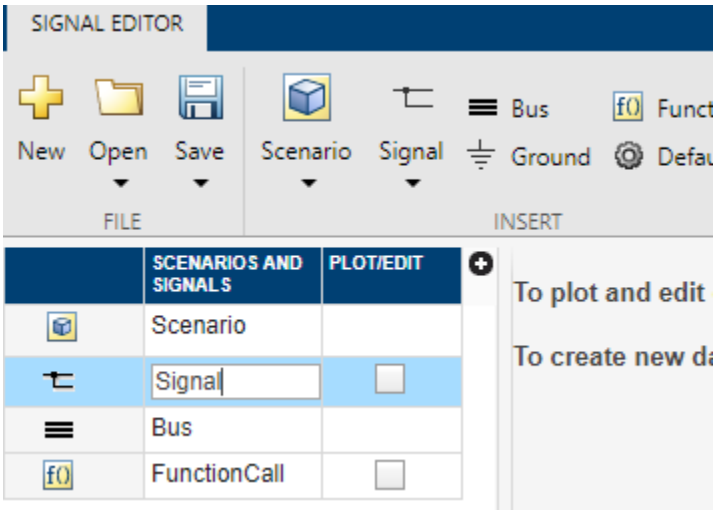
Zoom actions:

Type of Zoom or Pan	Button to Click
Zoom in along the T and Y axes.	
Zoom in along the time axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom in along the data value axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom only in *x* while zooming in *xy*.	Zoom in *xy* while pressing **Ctrl**
Zoom only in *y* while zooming in *xy*.	Zoom in *xy* while pressing **Shift**
Zoom out from the graph.	
Fit the plot to the graph. After selecting the icon, click the graph to enlarge the plot to fill the graph.	
Pan the graph up, down, left, or right. Select the icon. On the graph, hold the left mouse button and move the mouse to the area of the graph that you want to view.	

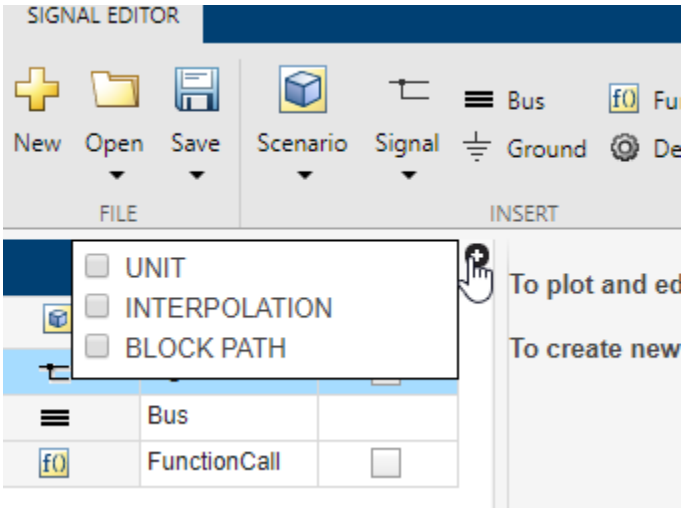
Change Signal Names and Hierarchy Orders

In the **Scenarios and Signals** section, you can change signal names and hierarchy order, create duplicates of signals, and delete signals. Simulink ignores leading and trailing spaces in signal names.

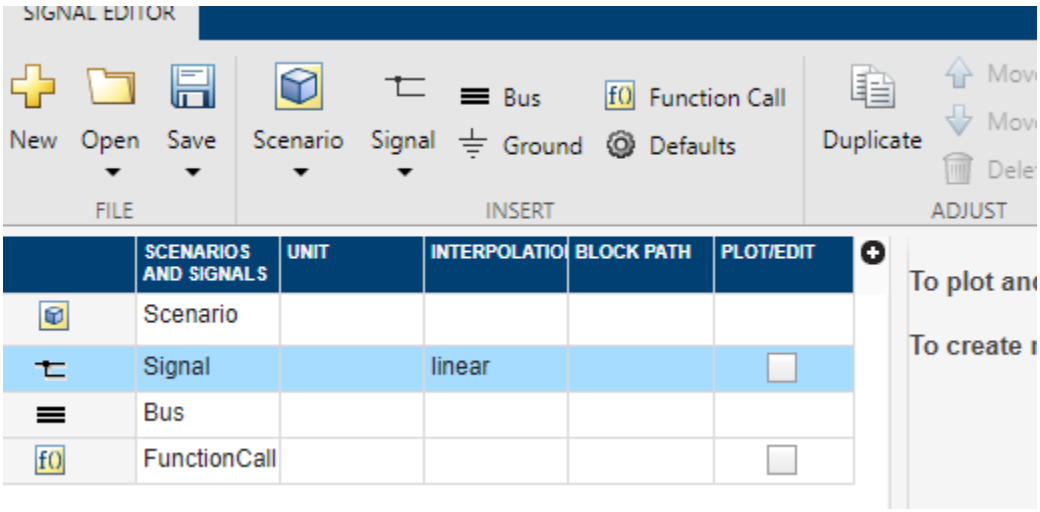
- To change a signal name, double-click the name and change it.



- To change the unit or interpolation of a signal, click the plus sign and click the **Unit** or **Interpolation** check boxes.



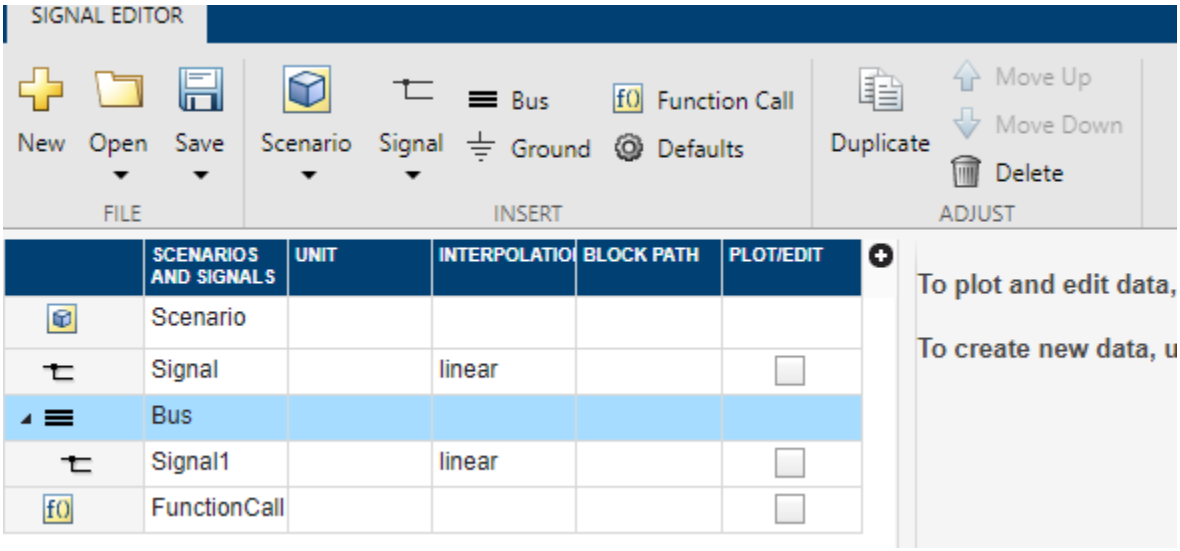
The pane updates with **Unit** and **Interpolation** columns.



- In the **Unit** column, enter an appropriate unit expression. For a suggested list of unit expressions, see allowed units.
- In the **Interpolation** column, from the drop-down list, select `linear` or `zero order hold`.

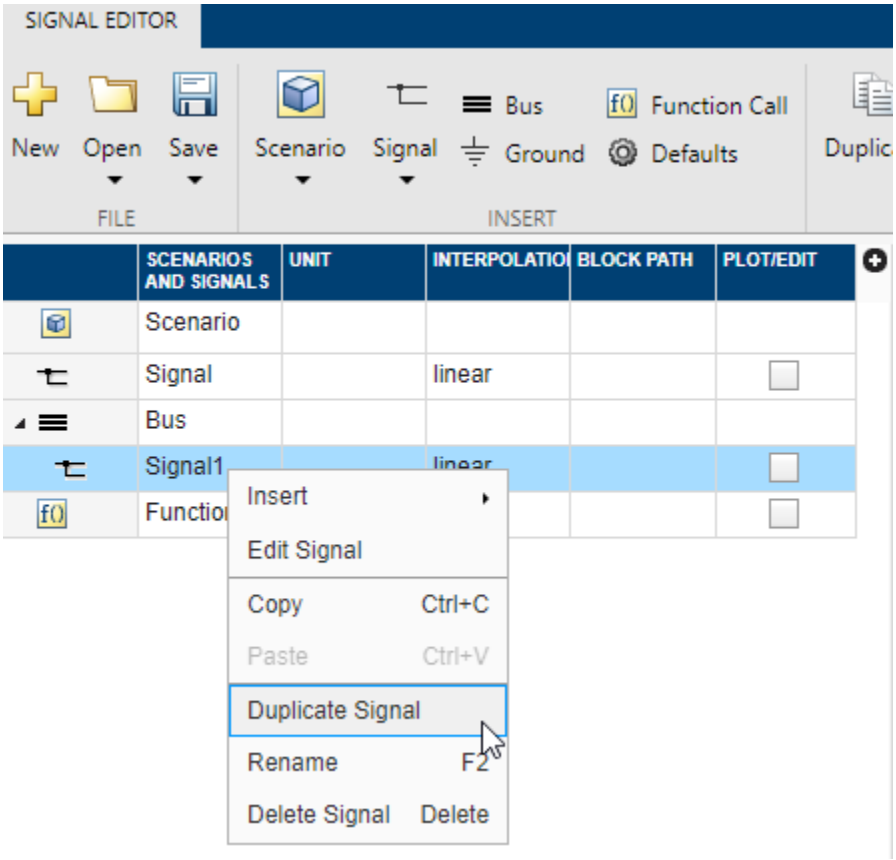
  Note Interpolation affects only the plotting of signals in the Signal Editor user interface.

- To change the order of a signal in the hierarchy, drag and drop it. For example, you can drag and drop signals into a bus.



Alternatively, use the **Move Up** and **Move Down** buttons in the **Adjust** section.

- To copy a signal and paste it under the original, right-click it and select **Duplicate Signal**.

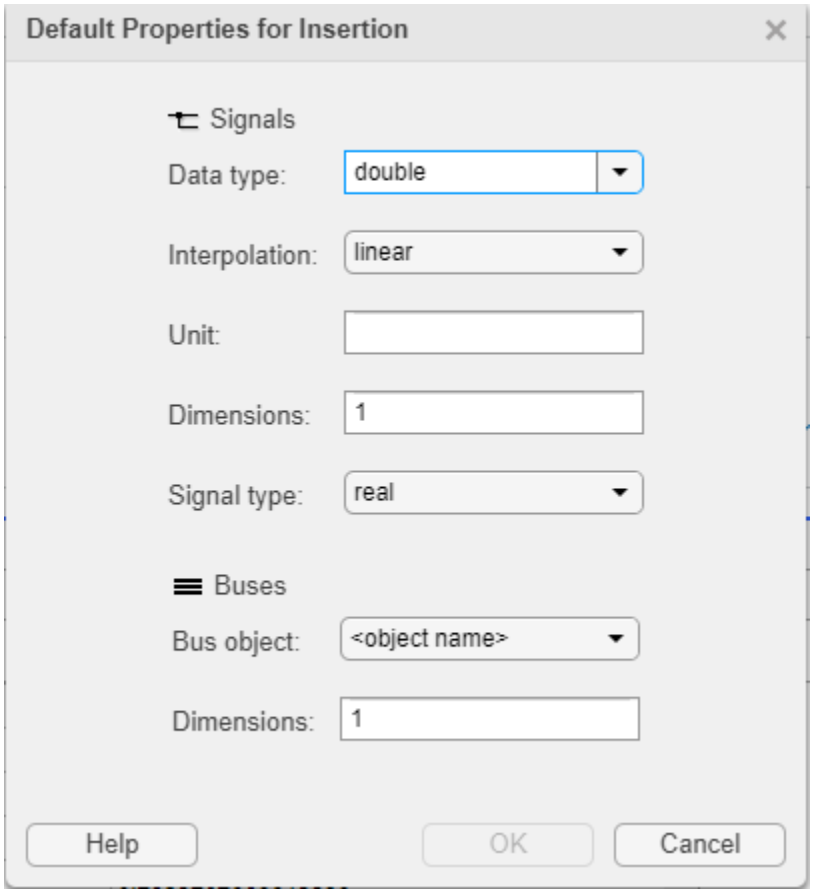


Alternatively, use the **Duplicate** button in the **Adjust** section. You can also adjust the default properties of the signal you duplicate. For more information, see “Create Signals with the Same Properties” on page 69-7.

- To copy a signal and paste it elsewhere in hierarchy, select **Copy** and then **Paste**.

Create Signals with the Same Properties

To create signals of the same predefined type, use the **Duplicate** button in the **Adjust** section. To change the predefined signal type, click the **Defaults** icon, . A Default Properties for Insertion dialog box displays.



- **Data type** — From the drop-down list, select the signal data type.
- **Enumeration** — When you select the `Enum` data type, this parameter displays. Enter the class name of your enumeration.

  If you define an enumeration class that contains the same integer value multiple times, for example:

```
classdef(Enumeration) hEnumColors_duplicateValues < Simulink.IntEnumType
  enumeration
    Red(118)
    Yellow(-14)
    Blue(90)
    Green(87)
    White(-14)
    Black(198)
    Brown(90)
    Pink(118)
    Purple(90)
  end
```

```
  methods (Static = true)
    function retVal = getDefaultValue()
      retVal = hEnumColors_duplicateValues.Blue;
    end
  end
end
```

The Signal Editor treats the first enumeration value (`Red (118)`) as the canonical value and equates all subsequent instances of the same underlying integer `118` to the enumerated name `Red`. In other words, `Pink` equals `Red`.

- **Interpolation** — From the drop-down list, select `linear` or `zero order hold`.
- **Unit** — Enter an appropriate unit expression. For a suggested list of unit expressions, see Allowed Units.
- **Dimensions** — Enter the number of dimensions for the signal.
- **Signal type** — From the drop-down list, select `real` or `complex`.
- **Bus object** — From the drop-down list, select the bus object for which to define the dimensions. If you leave the **Bus object** parameter at the default `<object name>`, Signal Editor adds empty buses.
- **Dimensions** — Enter the number of dimensions for the bus object.

Add and Edit Multidimensional Signals

To add multidimensional signals, use either of these options:

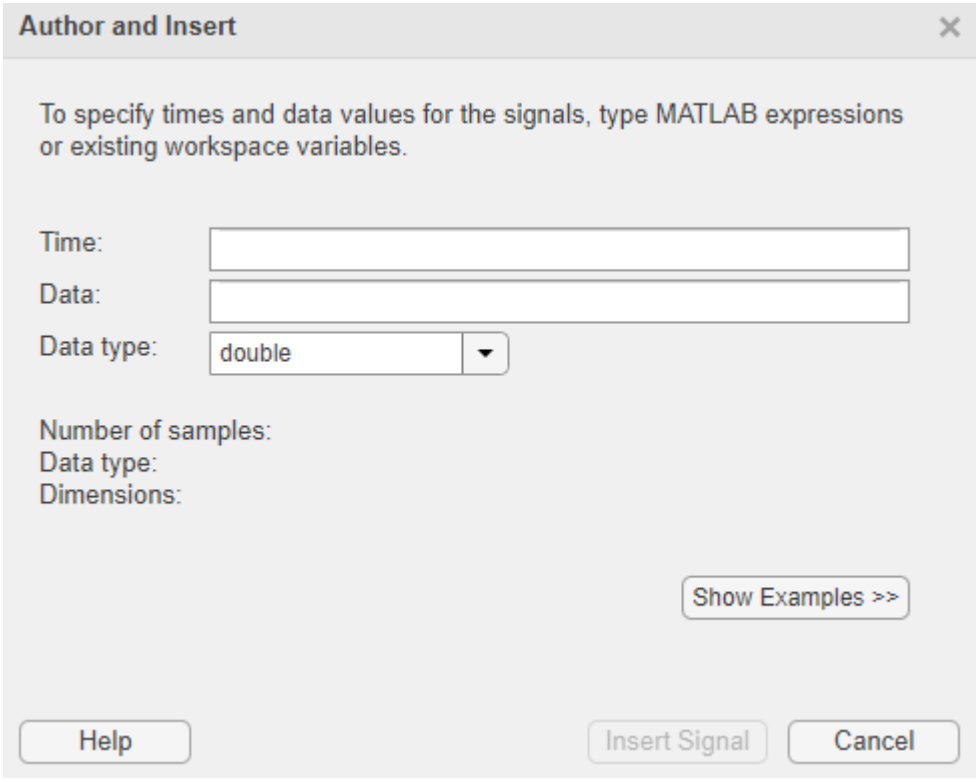
- Change the **Dimensions** property in the Default Properties for Insertion dialog box, and then use **Insert > Blank Signal** to insert a new blank signal. For a multidimensional blank signal, enter a dimension greater than 1.
- Enter multidimensional signal data in the Author and Insert dialog box **Data** parameter, such as `[(1:10)' (1:10)']`.

When you click the **Plot/Edit** check box for the signal, the tabular area displays the signal with columns for each dimension. You can edit the data individually in the tabular area, or click the replace button  to replace the signal with a MATLAB expression. Use the Author and Replace Signal Data dialog box as though you are inserting a new expression with the Author and Insert dialog box. For more information, see “Create Signals with MATLAB Expressions and Variables” on page 69-24.

Tip When replacing a signal, the signal dimension and complexity of the new signal must be the same as the signal being replaced.

For example, to create signal data with two columns and time from 1 to 10:

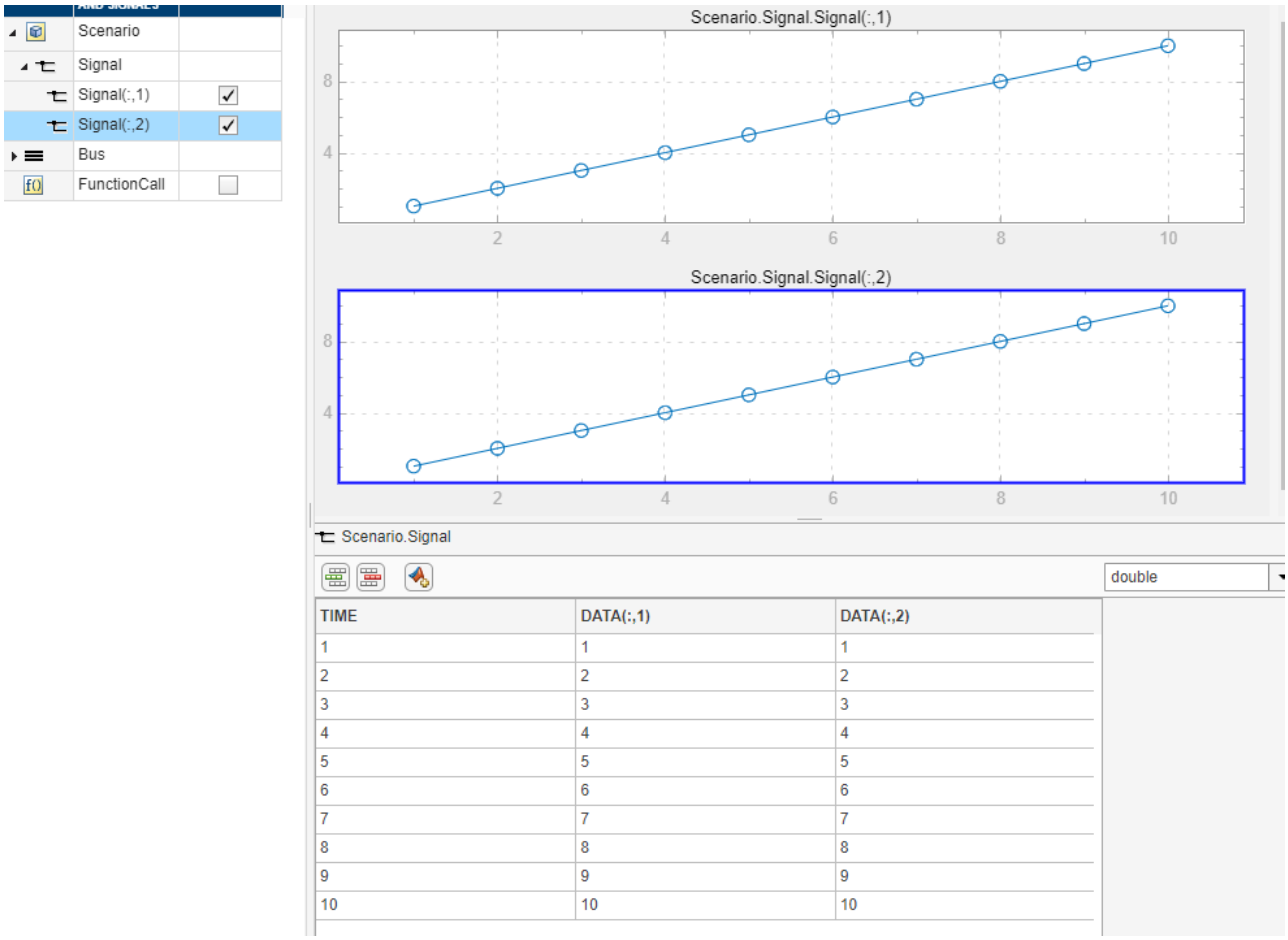
1. In the Signal Editor, select **Signal > Author Signal**.
2. Enter signal data with two columns and time from 1 to 10:
   - **Time** — `[1:10]`
   - **Data** — `[(1:10)' (1:10)']`



3 Click **Insert Signal**.

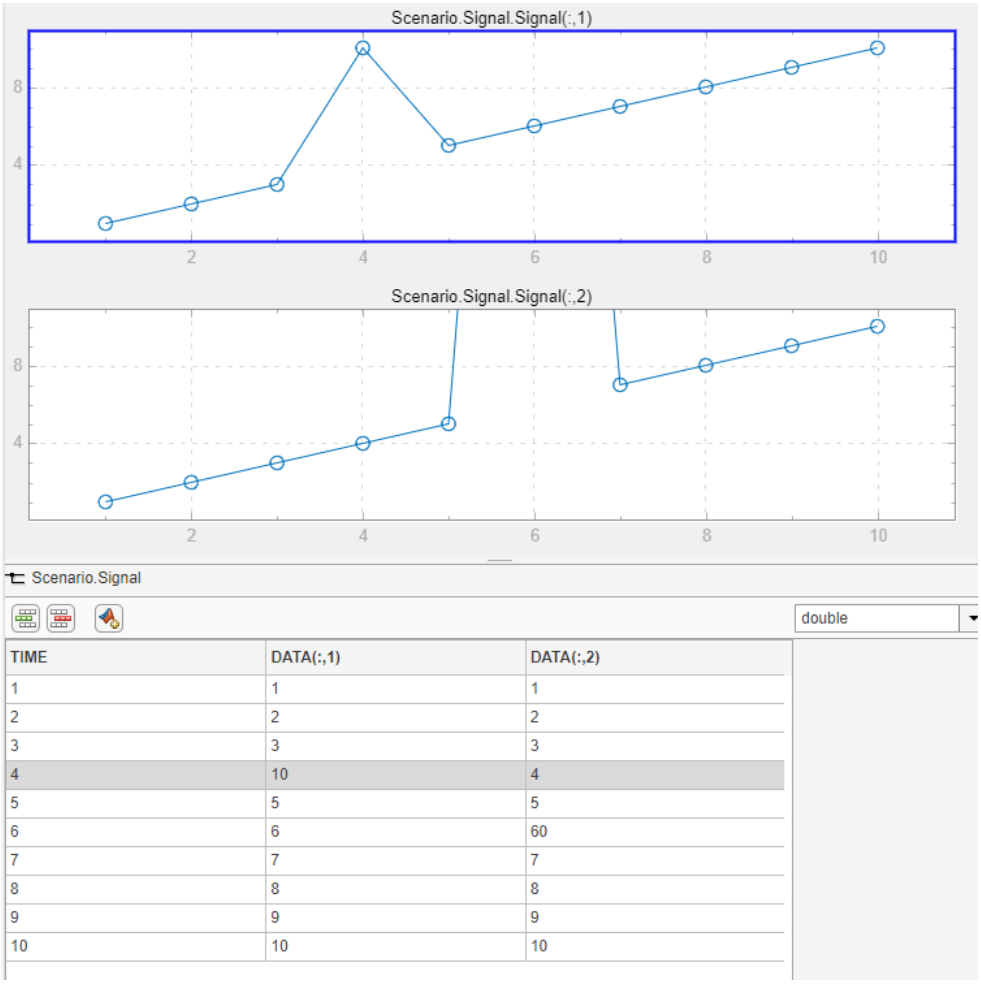
The hierarchy updates with the new signal data.

4 Expand the new signal and click the **Plot/Edit** check boxes for the new signal data. Observe the associated plots and the tabular data for the signal.



Tip If the data does not plot as expected, use the **Fit to window** button in the **Zoom & Pan** section. For multidimensional signals, also make sure that you have the right plot selected for the column data you are editing.

5 You can edit the data directly in the table.



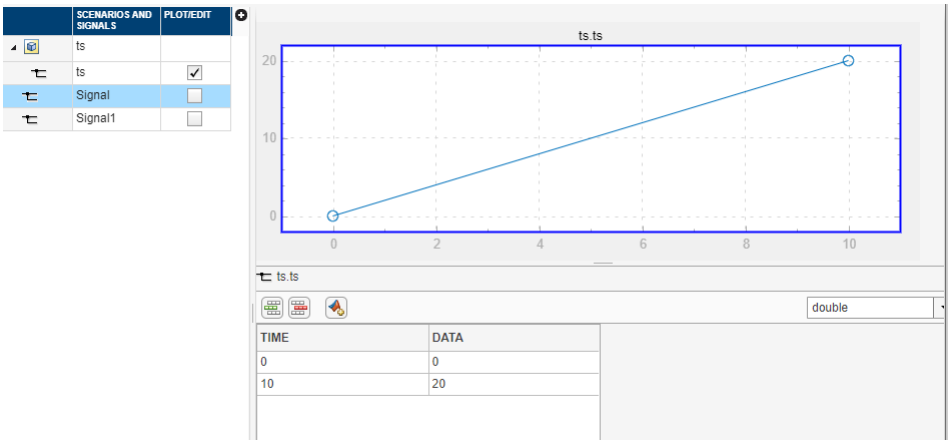
TIME	DATA(:,1)	DATA(:,2)
1	1	1
2	2	2
3	3	3
4	10	4
5	5	5
6	6	60
7	7	7
8	8	8
9	9	9
10	10	10

Or you can replace data completely with a new expression by clicking the **Replace** button and entering a new time range and data in the Author and Replace Signal Data dialog box.

Work with Data in Signals

This example describes how to add and delete data to the signals in the linked scenario. To create a model and data to work with, see “Add Signals to Scenarios” on page 69-17.

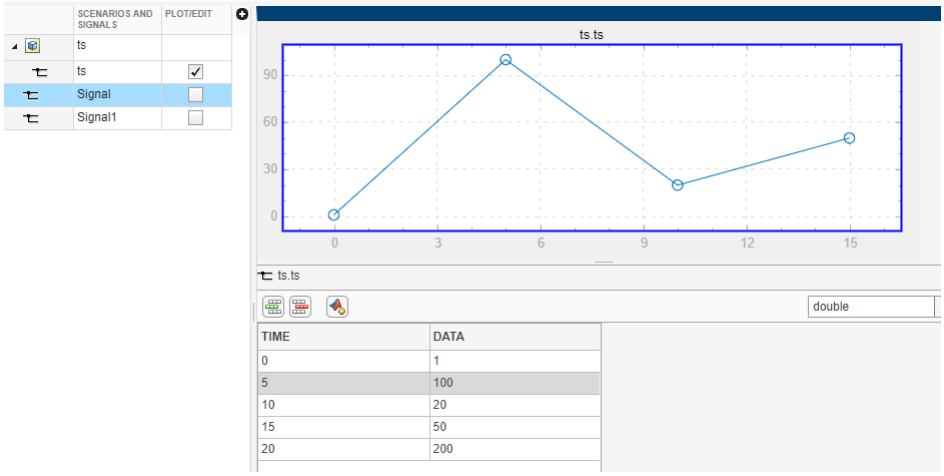
1 In the Signal Editor, in the **Scenarios and Signals** section, click the plot check box for the signal *ts*.



2 Add some data to the signal `ts`.

   a Click the add row icon and add some signals. To add a signal row between other signals, click the signal before and click the add row icon.

   b When done, click **Apply**. Clicking **Apply** updates the plot.



3 Remove the time 20 line from the signal. Select `20` and click .

4 Alternatively, if you want to replace all the signal data for `ts` with a signal defined with signal notations, click the replace button and use the Author and Replace Signal Data dialog box to define new data.

Draw a Ramp Using Snap to Grid for Accuracy

This example describes how to create a ramp signal by selecting some points in the canvas.

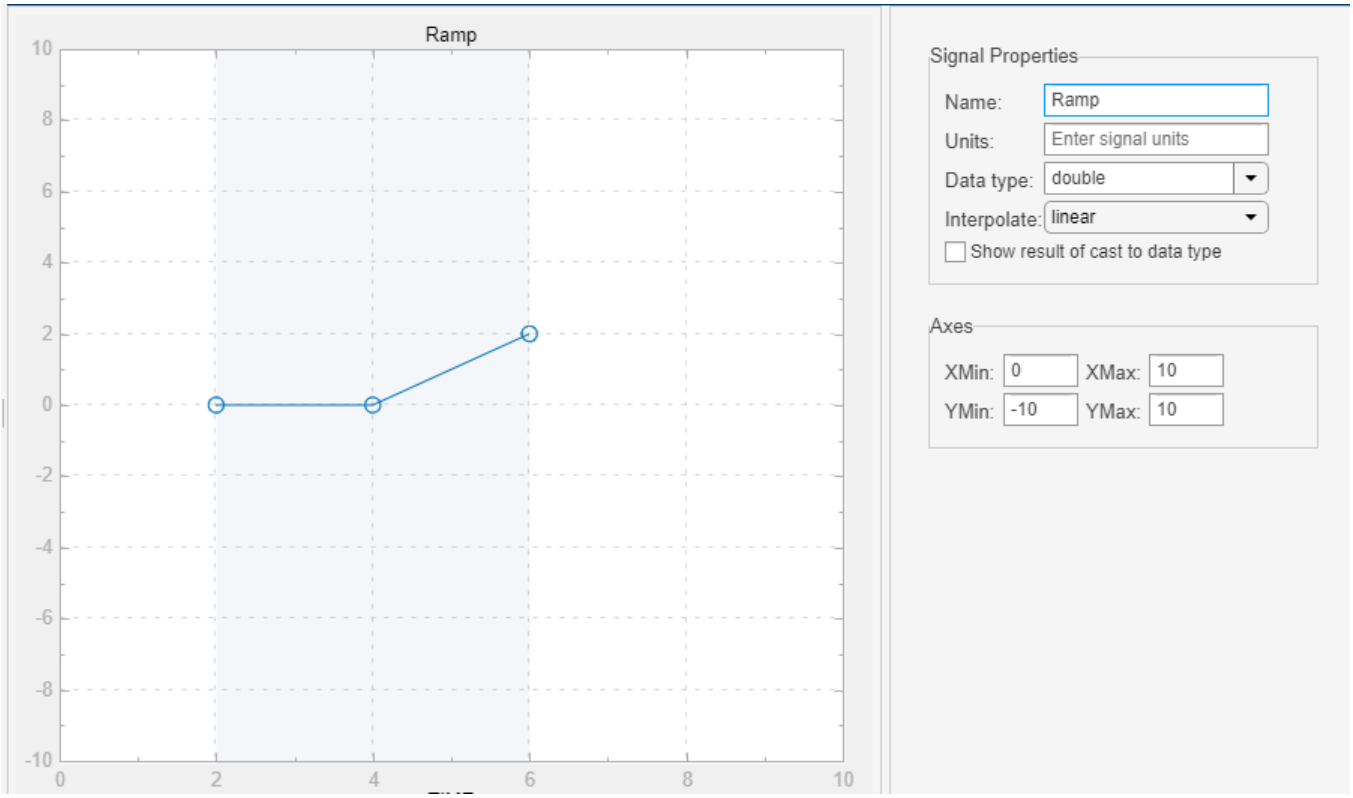
1 In the Signal Editor tab, select **Signal** > **Draw Signal**.

2 To line up the signal data values along horizontal and vertical lines, select **Snap X to Grid** and **Snap Y to Grid**.

3 In the canvas, add three points:

- Two points horizontal to each other
- One point set to the right at an angle to the other signals

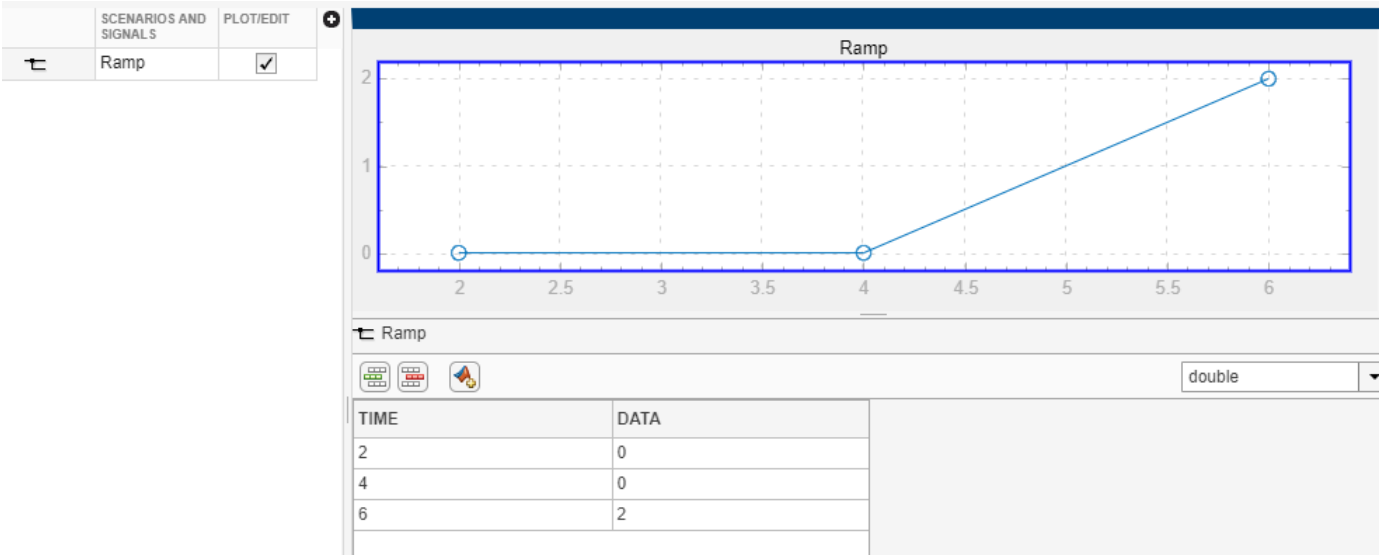
4 In the Signal Properties section, in **Name**, change the signal name to `Ramp` and press **Enter**.



5 To add the signal to the Signal Editor, in the **Insert** section, click  .

6 To return to the main Signal Editor window and check that the signal has been added, click  .

7 To observe the drawn signal data In Signal Editor, click the **Plot/Edit** check box for the `Ramp` signal name in the hierarchy.



Save and Send Changes to the Root Inport Mapper Tool

When you are done adding and modifying signals and scenarios, use the **Save and Sync** button to save the changes to a MAT-file. The Signal Editor also sends the data to the Root Inport Mapper Tool:

- If the Root Inport Mapper tool has the scenario loaded, the Root Inport Mapper tool updates with the new data.
- If the Root Inport Mapper tool has the scenarios mapped and your changes affect the mapping, the Root Inport Mapper tool unmaps the scenario.

See Also

Signal Editor | `linspace` | `signalBuilderToSignalEditor` | `signalEditor`

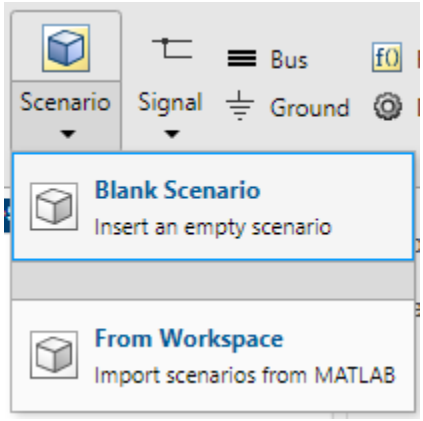
Related Examples

- “Map Root Inport Signal Data” on page 71-7
- “View and Inspect Signal Data” on page 71-17
- “Import Signal Data for Root Inport Mapping” on page 71-14
- “Exporting Signal Group Data” on page 75-84
- “Map Signal Data to Root Input Ports” on page 71-18
- “Root Inport Mapping Scenarios” on page 71-34
- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Create Custom File Type for Import to Signal Editor” on page 69-36

Use Scenarios and Insert Signals in Signal Editor

Use Scenarios to Group and Organize Inputs

Signal Editor uses scenarios to group and organize sets of inputs to be saved to a MAT-file for a single simulation. To create signal data using existing data sets from existing scenarios, or create an empty scenario into which to add signals, use the Signal Editor **Scenario** menu.



Action	Option
To create a scenario from the root inports of a model	Select **Scenario** > **Scenario from Model**. (Available only when accessing Signal Editor from the Root Inport Mapper.) You can also use the `signaleditor` function with a model argument. **Note** When using this option, the resulting scenario contains signals with the data types and dimensions of the inport ports.
To create an empty scenario and create signals from scratch	Select **Scenario** > **Blank Scenario**.
To import scenarios from MATLAB workspace	Select **Scenario** > **From Workspace**.

After you have your scenario:

- To begin inserting signals, use the other options in the **Insert** section. For more information, see “Create Signals and Signal Data” on page 69-20.
- To change the signal order in the hierarchy or change the name of a signal, see “Change Signal Names and Hierarchy Orders” on page 69-4.

Link in Signal Data from Signal Builder Block and Simulink Design Verifier Environment

You can use **Signals** > **Edit MAT-File** to link in MAT-file data from these sources for editing.

- Signal Builder blocks.
- Simulink Design Verifier.

Link in Data from Signal Builder

You can link in and edit data exported from the Signal Builder block in a MAT-file or MATLAB. Use one of these methods to export the data:

- Signal Builder block **File > Export Data > To MAT-file** option, then link in the MAT-file.
- `signalbuilder get` function with data sets, then perform either of these steps:
  - Import the data sets in the workspace and save to a MAT-file
  - Save the data sets in the workspace to a MAT-file and import the MAT-file

For more information on exporting from a Signal Builder block, see “Exporting Signal Group Data” on page 75-84.

Link in Test Vectors from Simulink Design Verifier Environment

You can link in and edit Simulink Design Verifier test vectors. This workflow requires a Simulink Design Verifier license.

Before linking in, use the Simulink Design Verifier `sldvsimdata` function to convert a Simulink Design Verifier test structure to a set of `Simulink.SimulationData.Dataset` objects. This file contains a test vector structure `sldvData`. Save the output to a MAT-file and then import that file into Signal Editor.

Insert Signals

To insert signals into scenarios, select the scenario, then click a signal type from the **Insert** section.

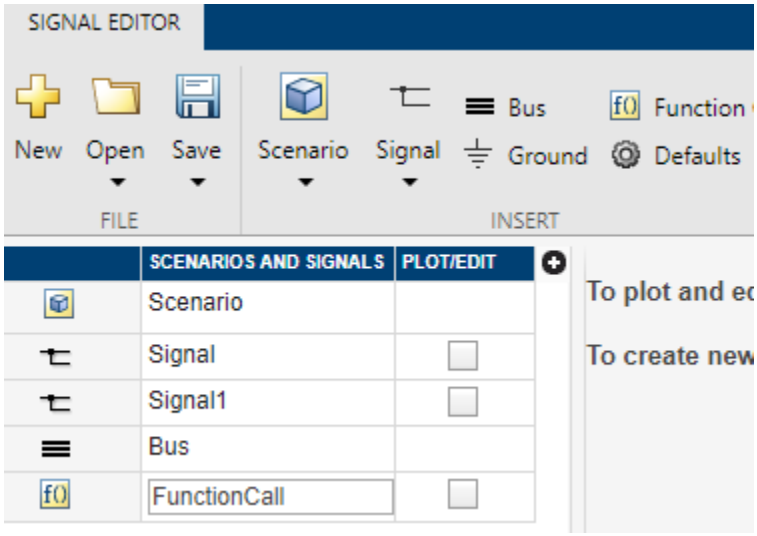
- **Signal**

  Use the **Signal** split button to specify how you want to define the signal data:

  - **Blank Signal** — Add signal data directly into a table (see “Work with Basic Signal Data” on page 69-20).
  - **Author Signal** — Author signal data using signal notations and variables (“Create Signals with MATLAB Expressions and Variables” on page 69-24).
- **Bus**
- **Ground**
- **Function Call**

  If you need a function-call signal for a root inport with explicit periodic sample time, insert a ground signal instead. Simulink then executes the function-call automatically.

The new signals appear in the **Scenarios and Signals** section.



You can also insert multiple signals of the same type. For more information, see “Create Signals with the Same Properties” on page 69-7.

To change the signal order in the hierarchy or change the name of a signal, see “Change Signal Names and Hierarchy Orders” on page 69-4.

To edit the properties of a signal:

- For tabular editing, see “Work with Basic Signal Data with a Tabular Editor” on page 69-20.
- To replace the signal data or edit the signal notation using MATLAB expressions, see “Replace Signal Data with MATLAB Expressions” on page 69-28.

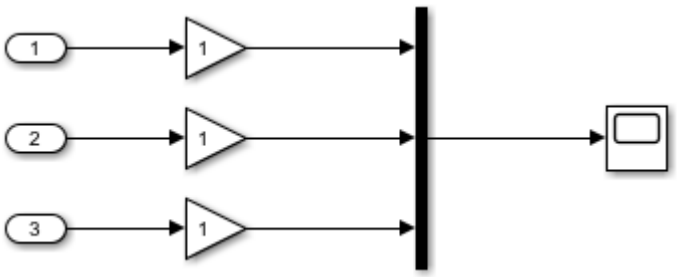
Add Signals to Scenarios

This example describes how to create a scenario to be linked to from the Root Inport Mapper tool. You can then start the Signal Editor to manipulate and add signals to this scenario.

1. In the MATLAB Command Window, create some data by typing:

   ```
   ts = timeseries([0;20],[0;10]);
   ```

2. In Simulink Editor, create a model that contains three Inport blocks, three Gain blocks, a Mux block, and a Scope block. Connect these blocks as shown:



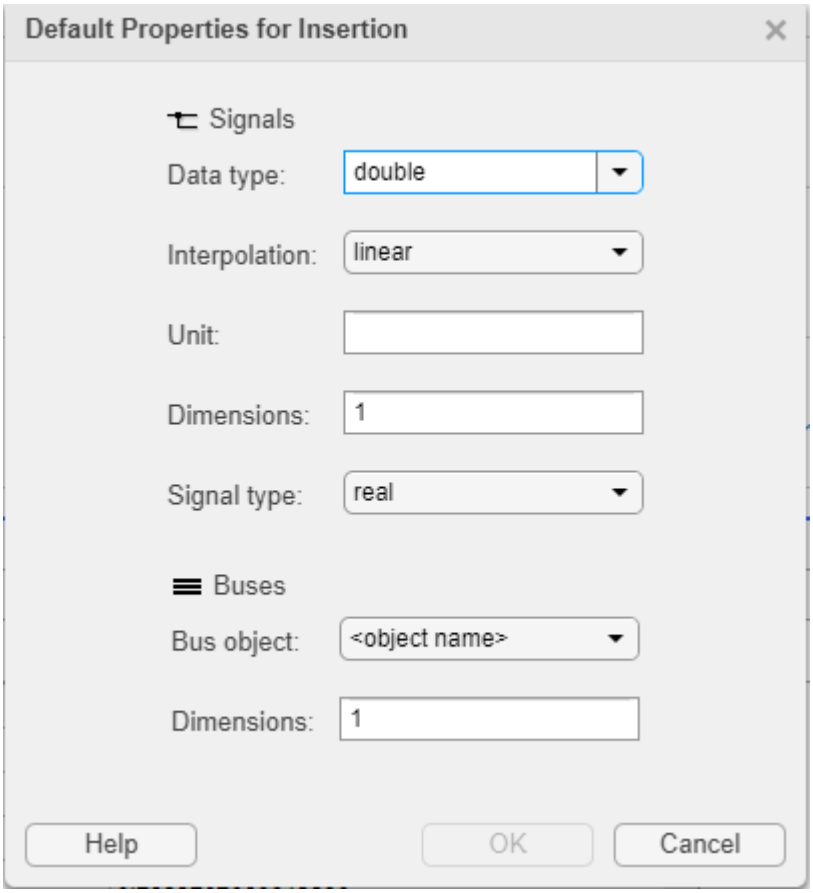
3. Set the gain for the Gain blocks to `5`, `10`, and `15`, respectively.

4. Click one of the Inport blocks, then click the **Connect Input** button.

   The Root Inport Mapper tool displays.
5. In Root Inport Mapper **Link** section, select **From Workspace**.
6. In the From Workspace window, enter a name to store the MAT-file, then click **OK**.
7. In the **Scenario** section of the Signal Editor, click **Signals** > **Edit MAT-File**.
8. In the Edit Signal File window, select the new MAT-file and click **OK**.

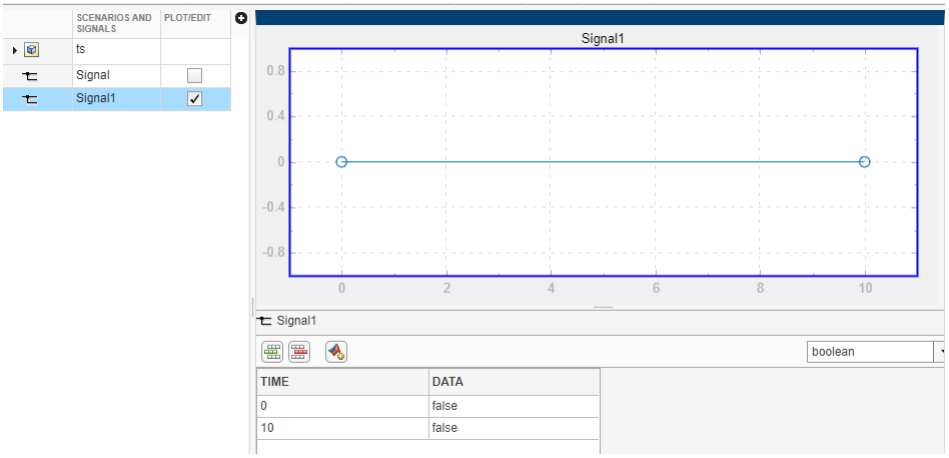
   The Signal Editor displays.
9. Add a signal, *`Signal`*, to the scenario. Right-click the scenario and select **Insert** > **Blank Signal**.

   This action adds *`Signal`* with these default properties.



   Alternatively, insert signals by clicking a signal type from the **Insert** section or using the **Signal** split button to author a signal using signal notations.
10. Change the default properties of signals you want to add. In the **Insert** section, select **Defaults**. In the Default Properties for Insertion dialog box, change the data type to `boolean`, then right-click the scenario and select **Insert** > **Blank Signal**.

    This action adds *Signal1* with the data type `boolean`.
11. To check that the data type is `boolean`, click the plot check box for *`Signal1`*.



See Also

Signal Editor | `signalEditor`

More About

- “Create and Edit Signal Data” on page 69-2
- “Work with Basic Signal Data” on page 69-20
- “Create Signals with MATLAB Expressions and Variables” on page 69-24
- “Create Freehand Signal Data Using Mouse or Multi-Touch Gestures” on page 69-31

Work with Basic Signal Data

Create Signals and Signal Data

In the **Signal Editor** tab, create signal data either from existing model data (scenarios) or start with an empty scenario. To a scenario, add signals and data by:

- Adding signal data directly into a table (“Work with Basic Signal Data” on page 69-20).
- Authoring signal data using signal notations and variables (“Create Signals with MATLAB Expressions and Variables” on page 69-24).
- Drawing signal points and lines (“Create Freehand Signal Data Using Mouse or Multi-Touch Gestures” on page 69-31)

After inserting the signal, view, plot, and edit the data by clicking the **Plot/Edit** check box. The plot opens in the **Edit** tab.

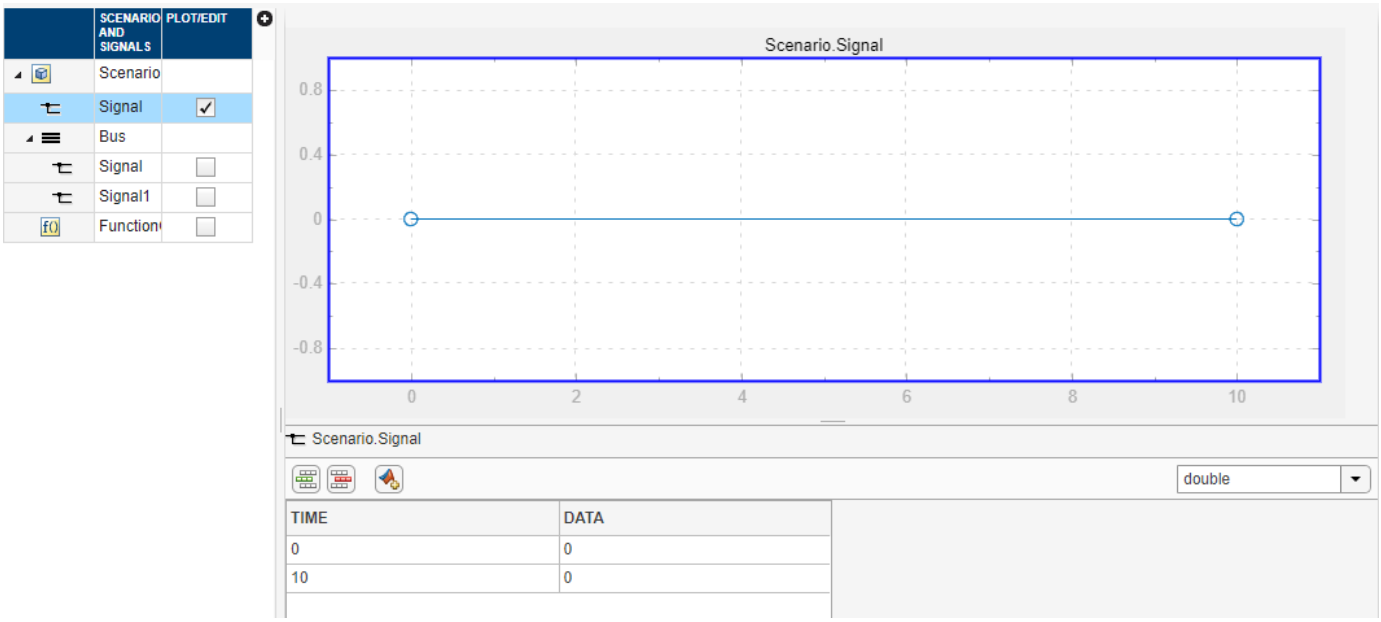
If the data does not plot as expected, use the **Fit to window** button in the **Zoom & Pan** section. For multidimensional signals, also make sure that you have the right plot selected for the column data you are editing.

Explore the plots using the **Measure** and **Zoom & Pan** sections on the toolbar.

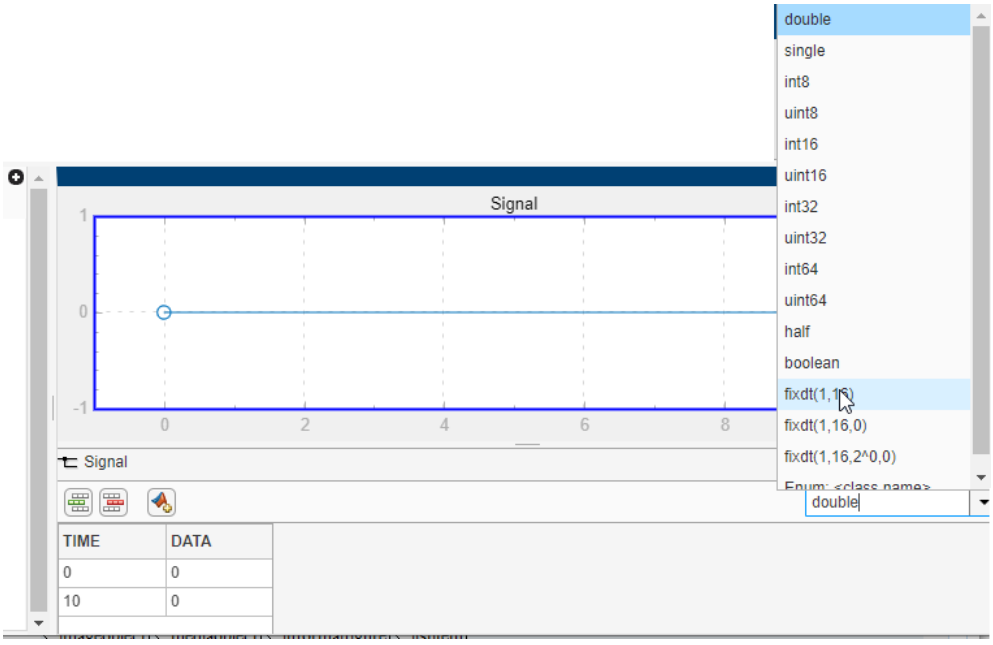
- In the **Measure** section, use the **Data Cursors** button to display one or two cursors for the plot. These cursors display the T and Y values of a data point in the plot. To view a data point, click a point on the plot line.
- In the **Zoom & Pan** section, select how you want to zoom and pan the signal plots. Zooming is only for the selected axis.

Work with Basic Signal Data with a Tabular Editor

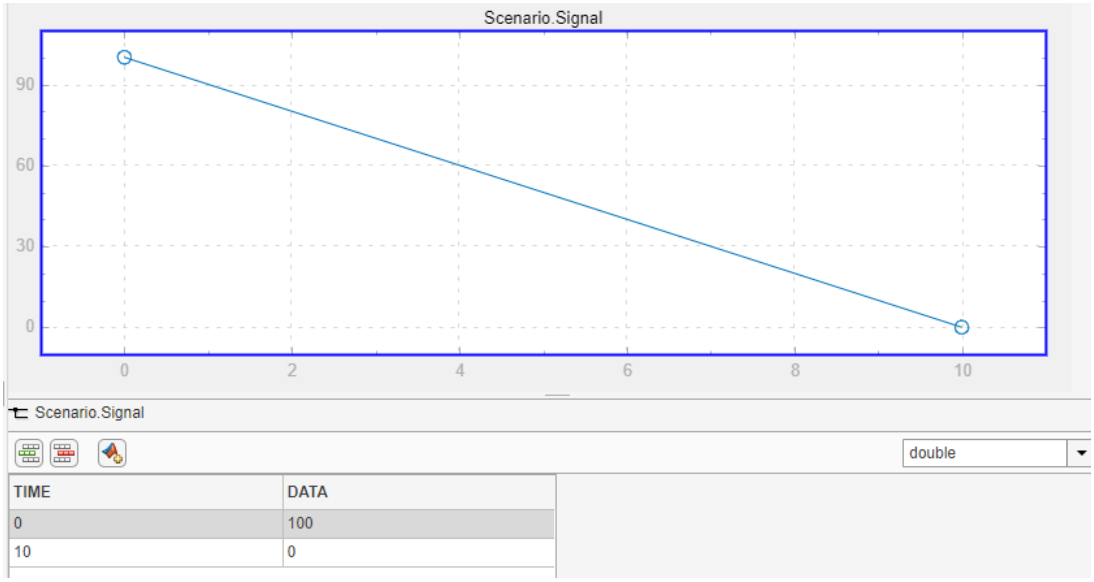
To add and edit basic signal data, select a signal and click the associated **Plot/Edit** check box. The **Edit** tab opens with a plot of the signal. Each data point in the signal is demarcated with a circle (marker), which you can toggle off and on through the context menu **Show > Toggle Markers** option. Under the signal plot is a tabular editor.



- To insert or delete a data row for a signal, use  or  , respectively.
- To change the data type for signal data, select the type from the drop-down.



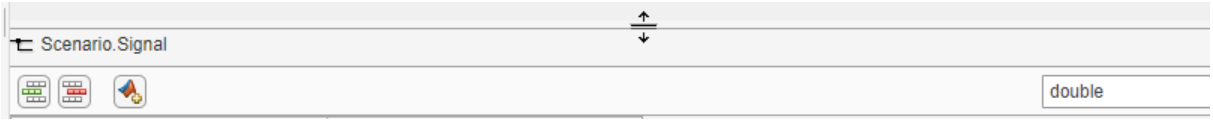
- To change the time or data for each signal, edit the associated column of the data row, then click anywhere in the canvas to update the plot of the signal. You may need to click **Fit to View** in the toolstrip to adjust the plot axes.



Note If the data is fixed-point data, hovering over the data in the table displays a summary of the data

- Ideal Value — Requested value.
- Fixed-Point Value — Value resulting from casting the ideal value as a fixed-point value.
- Absolute Error — Absolute error of value.
- Relative Error — Difference between cast value and the original value.
- Additional error information, such as whether the error is an overflow or underflow.

- To change the size of the plot or tabular area, move the separator up and down.



- To create multidimensional signals, use one of these methods. For more information, see “Add and Edit Multidimensional Signals” on page 69-8.
  - In the Default Properties for Insertion dialog box, enter a dimension greater than `1` in the **Dimensions** parameter.
  - In the Author and Insert dialog box, enter a MATLAB expression that creates multidimensional signals in the **Data** parameter

Instead of using the tabular editor to define signal data, you can use MATLAB expressions.

See Also

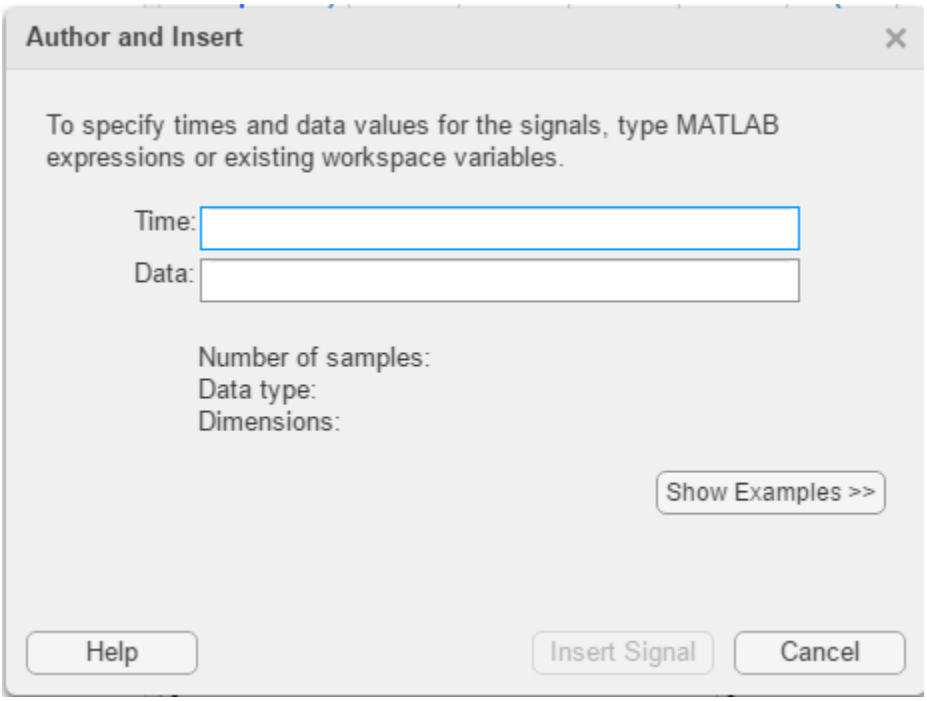
Signal Editor | `signalEditor`

More About

- “Create and Edit Signal Data” on page 69-2
- “Use Scenarios and Insert Signals in Signal Editor” on page 69-15
- “Create Signals with MATLAB Expressions and Variables” on page 69-24
- “Create Freehand Signal Data Using Mouse or Multi-Touch Gestures” on page 69-31

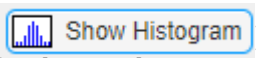
Create Signals with MATLAB Expressions and Variables

To add signals using MATLAB expressions and variables, select the Signal Editor **Signal** > **Author Signal** option.



- **Time** — Enter the range of time for the data.
- **Data** — Enter the MATLAB expression for the signal.
- **Data type** — Select or enter the signal data type.
  - `double`
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uin16`
  - `int32`
  - `uint32`
  - `boolean`
  - `fixdt(1,16)`
  - `fixdt(1,16,0)`
  - `fixdt(1,16,2^0,0)`
  - `string`
  - `Enum: <class name>`

If you enter your time and data and then select a fixed-point data type, the Signal Editor displays a fixed-point proposed data type for your data.

- To help you select a fixed-point data type, click the **Show Histogram** button (

). Clicking this button displays a plot of the signal data using the selected fixed-point data type. The graph displays:

Column	Information
Values	The negative, positive, and zero signal values.
Potential Overflows	Bins the signal values that may overflow.
In-Range	Bins the signal values that are within acceptable range.
Potential Underflows	Bins the signal values that may underflow.

  To see the difference that a data type may have on the histogram, select:

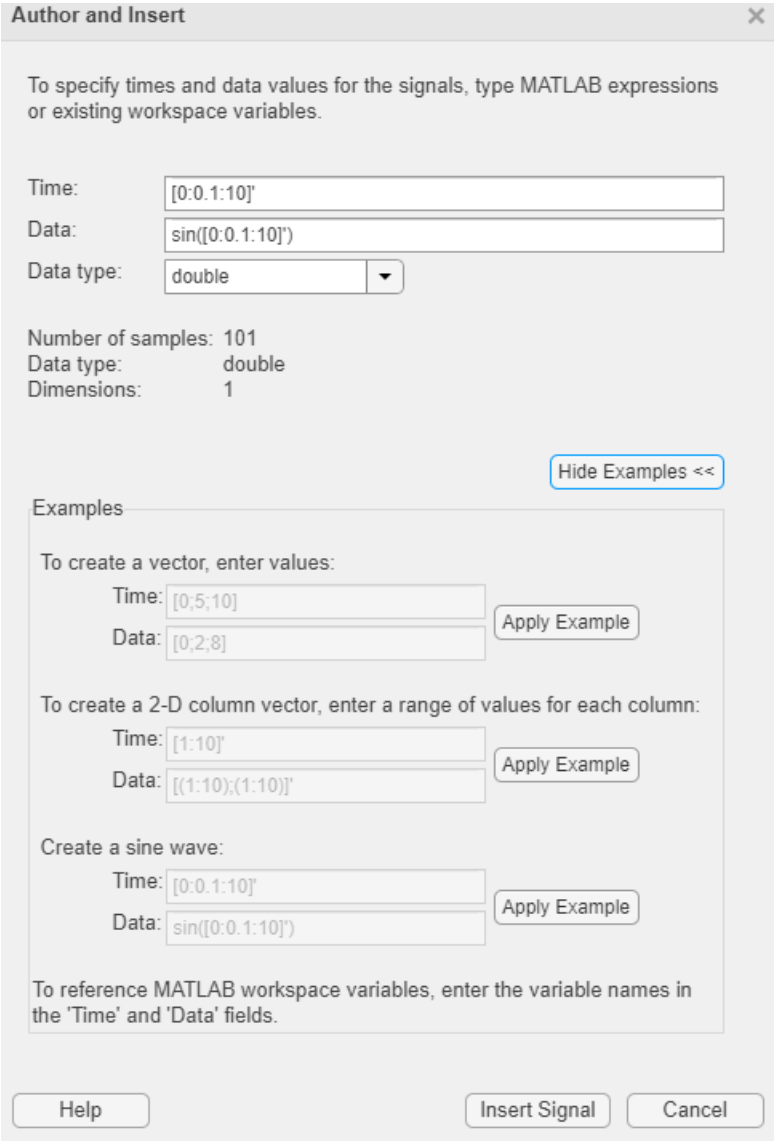
  - User Specified
  - Binary Scaling
  - Slope & Bias Scaling

  For more information, see “Histogram Plot of Signal” (Fixed-Point Designer).
- To apply the proposed fixed-point data type to your data, click the **Use proposed data type button** ().

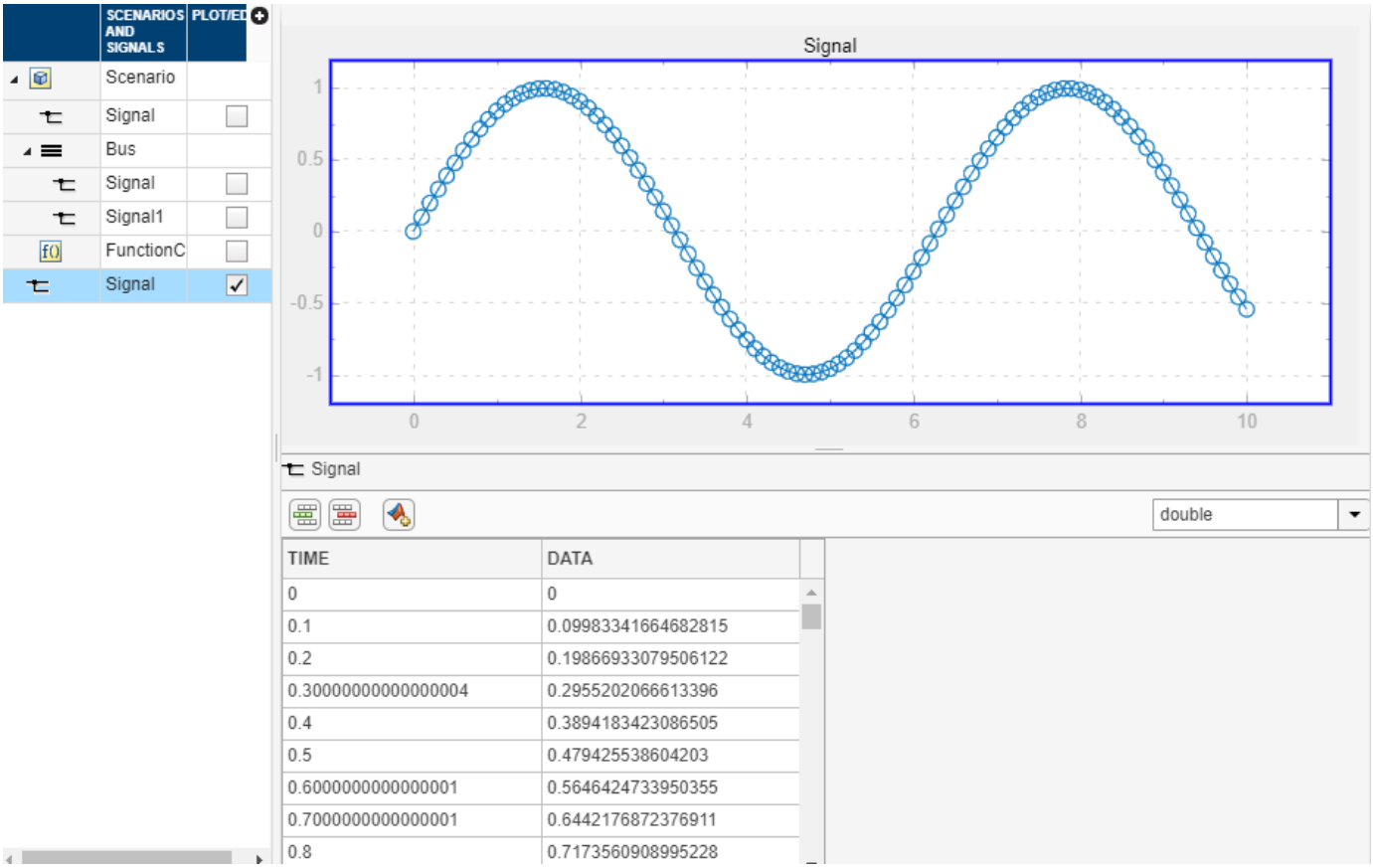
When you click **Insert Signal**, the interface evaluates the signal, updates the signal information in the dialog box, and adds the signal to the **Scenarios and Signals** section. In addition, the number of samples, signal data type, and signal dimension also appear.

To see example **Time** and **Data** entries, click the **Show Examples** button. To experiment with the signals from these example signal notations, click the **Apply Example** button for the associated example. You can also modify the examples before inserting.

For example, to create a sine wave, click the **Apply Example** button next to that example and click the **Insert Signal** button. Observe that dialog box displays the number of samples, signal data type, and signal dimensions.



To view the signal, cancel the Author and Insert dialog box, navigate to the **Scenarios and Signals** section, and click the **Plot/Edit** button for the new signal. If a signal has real and imaginary parts, both parts display in one plot. The tabular editor also reflects the signal data.



Explore the plots using the plot context menus **Align** and **Zoom & Pan**.

If the data does not plot as expected, use the **Fit to view** button in the **Zoom & Pan** section. For multidimensional signals, also make sure that you have the right plot selected for the column data you are editing.

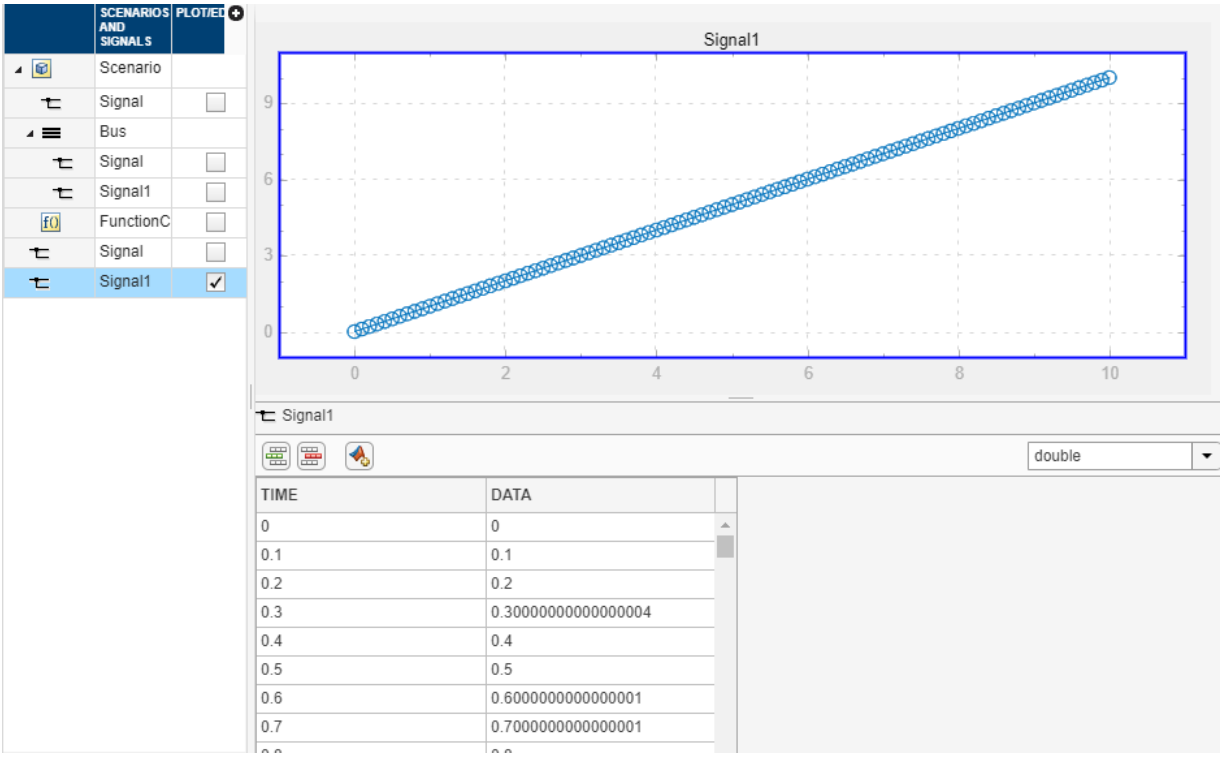
- In the **Measure** section, use the **Data Cursors** button to display one or two cursors for the plot. These cursors display the T and Y values of a data point in the plot. To view a data point, click a point on the plot line.
- In the **Zoom & Pan** section, select how you want to zoom and pan the signal plots. Zooming is only for the selected axis.

Type of Zoom or Pan	Button to Click
Zoom in along the T and Y axes.	
Zoom in along the time axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom in along the data value axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom out from the graph.	

Type of Zoom or Pan	Button to Click
Fit the plot to the graph. After selecting the icon, click the graph to enlarge the plot to fill the graph.	
Pan the graph up, down, left, or right. Select the icon. On the graph, hold the left mouse button and move the mouse to the area of the graph that you want to view.	

Tip To produce signals with linearly spaced values for time, use the `linspace` function, for example:

- **Time** — `linspace(0,10,101)`
- **Data** — `[0:0.1:10]`



To edit signal data without using MATLAB expressions, see “Work with Basic Signal Data” on page 69-20.

Replace Signal Data with MATLAB Expressions

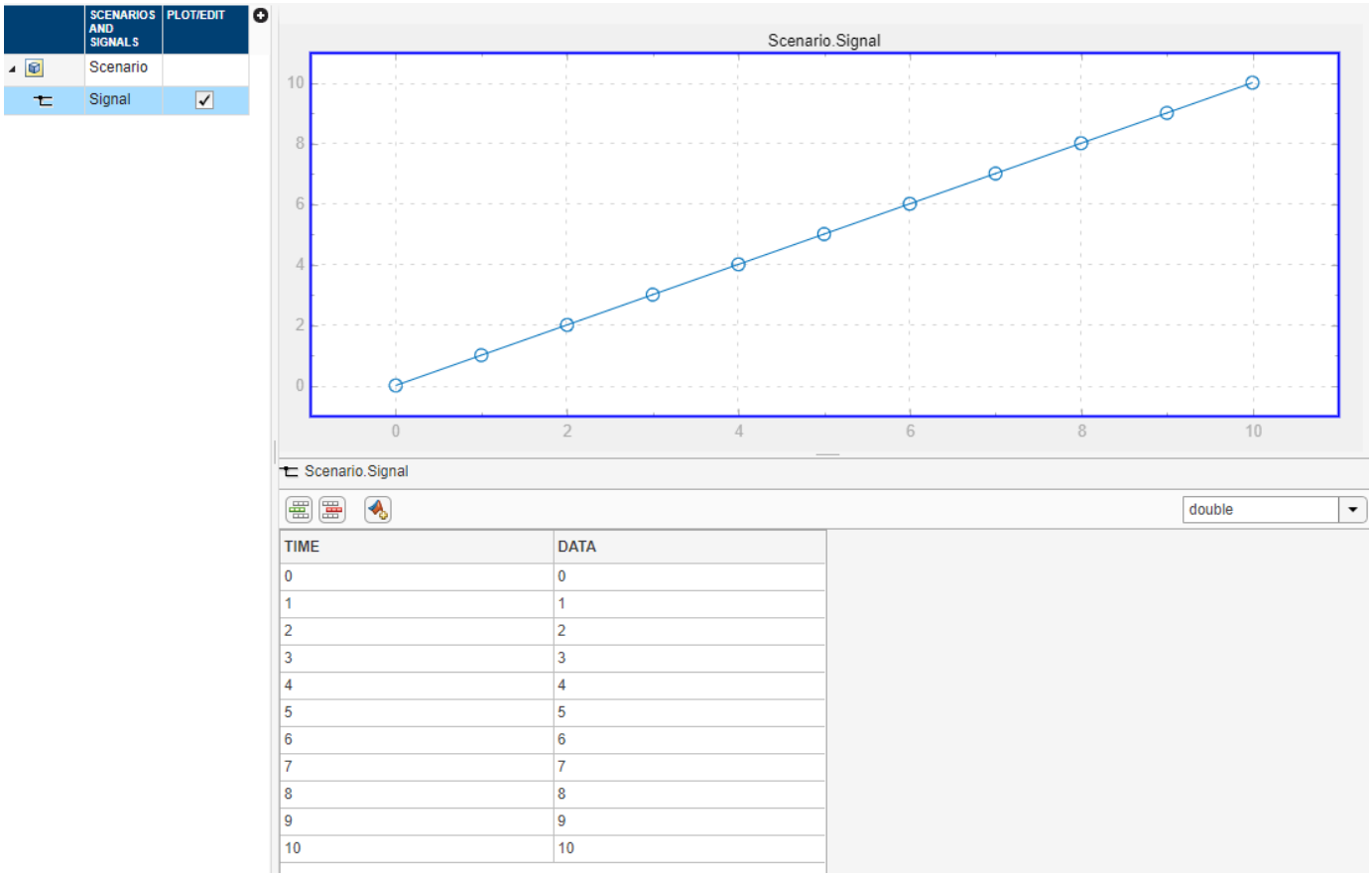
You can replace signal data using MATLAB expressions at any time, regardless of how you created the original signal data. To replace signal data, select that signal in the **Scenarios and Signals** section, and then click the replace button ( ). Use the Author and Replace Signal Data dialog box as if you were inserting a new expression with the Author and Insert dialog box. For more information, see “Create Signals with MATLAB Expressions and Variables” on page 69-24.

Note You cannot change data types to or from a fixed-point data type.

Tip When replacing a signal, the signal dimension and complexity of the new signal must be the same as the signal being replaced.

Tip To produce signals with linearly spaced values for time, use the `linspace` function. For example, using:

- **Time** — `linspace(0,10,11)`
- **Data** — `[0:10]`



See Also

Signal Editor | `signalEditor`

More About

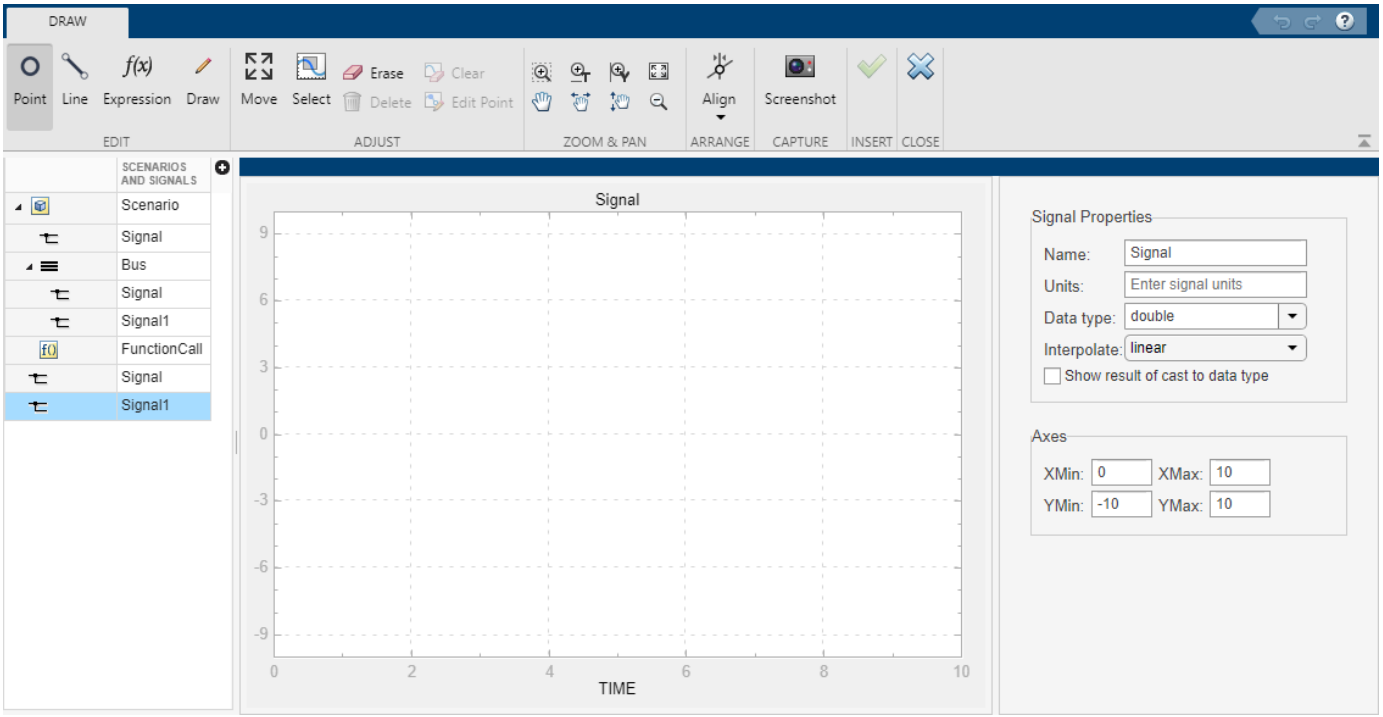
- “Create and Edit Signal Data” on page 69-2
- “Use Scenarios and Insert Signals in Signal Editor” on page 69-15
- “Work with Basic Signal Data” on page 69-20

- “Create Freehand Signal Data Using Mouse or Multi-Touch Gestures” on page 69-31

Create Freehand Signal Data Using Mouse or Multi-Touch Gestures

Freehand signal data is data that you add graphically. Signal Editor allows you to add freehand signal data using a mouse or touchscreen (if available and supported). While using a touchscreen, use common multi-touch gestures such as tap, pan, pinch, and double-tap (select all).

To add freehand signal data, in the **Signal Editor** tab, select **Signal** > **Draw Signal** option. A **Draw** tab opens.



You can draw your own signal lines or points, use MATLAB expression to enter data, move signal lines or points across the canvas, take an image of the lines of the canvas, and add the data to the Signal Editor.

To create signal data, in the **Edit** section:

- Insert a single data point by clicking **Point** and then clicking in the graph area. If you add additional points, the tool connects the points.
- Insert a signal line by clicking **Line** and then clicking in the graph area. To extend the line, click on the endpoints of the line and draw the extension.

  Tip When you insert signals or signal lines, the canvas adds a shaded area bound by the leftmost and rightmost endpoints. While in **Insert Line** mode, you cannot add more points within this shaded area. You add additional line points outside the shaded area.

- Draw a signal freehand by clicking **Draw**.

- Draw a signal using MATLAB expressions by clicking **Expression** ($f(x)$, Ctrl+E). Enter time and data values that create signal points that are scalars or vectors whose number of points match the time points.

To select or move signal data, in the **Signal** section:

- Select a point or line by clicking **Select** and then selecting the point or area to be selected.
- Move a point or line by clicking **Move**.

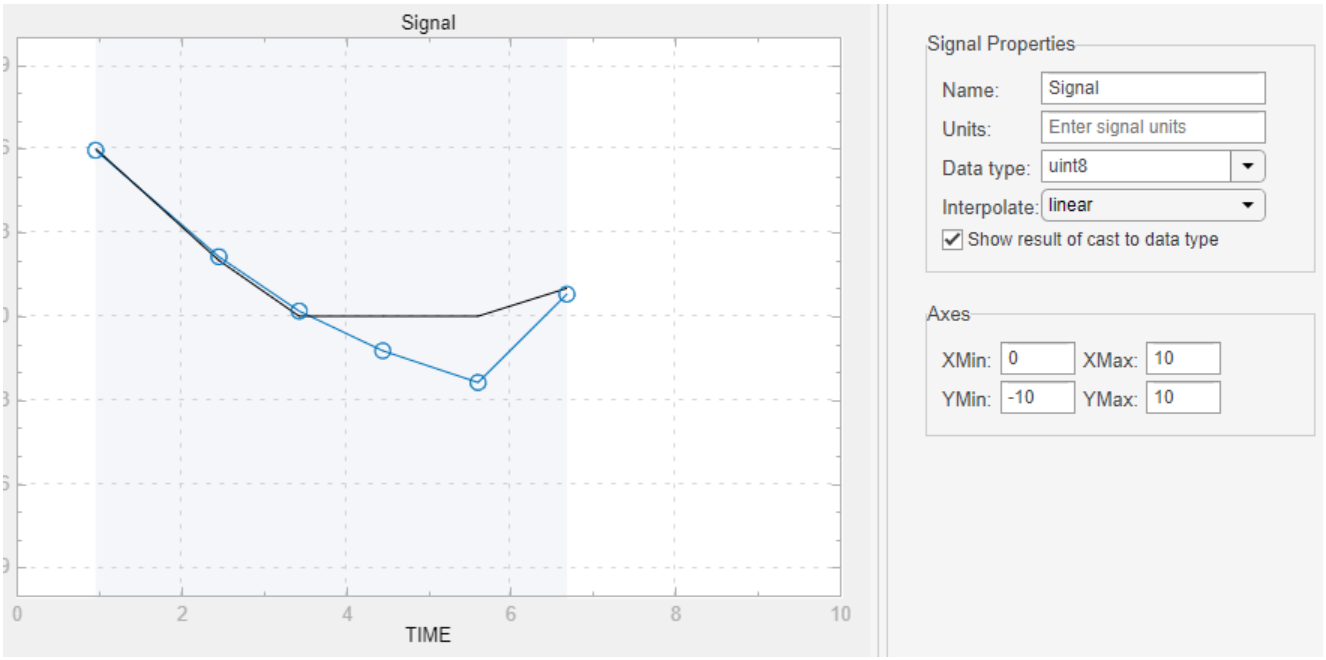
To edit or remove the signal data, in the **Edit** section:

- Change the data of a point by clicking **Edit Point**. In the Edit Point dialog box, enter the new **Time** and **Data** values. These values must be scalar.

To work with the canvas, use the tools in the **Align**, **Show**, and **Zoom** sections.

- Snap the canvas graph to various grids using the tools in the **Align** section. Snapping to grid helps you better control data accuracy while drawing signals. For example, consider snapping to the *x*-grid to uniformly sample signal values or snapping to the *y*-grid to control the amplitude of a ramp signal.
- Toggle the display of data points, grid lines, and data markers in the canvas, explore the commands in the **Align** and **Show** sections. To zoom in and out of the canvas, see **Zoom & Pan**. These actions are touchscreen supported with the pan and pinch gestures. For more information, see “Create Signals and Signal Data” on page 69-20.

To work with signal properties, use the parameters in the **Signal Properties** section. For example, to see the changed values of your signal as you change data types in the **Data type** parameter, select the **Show result of cast to data type**. The original line is blue, the changed values are represented by the black line. This graphic illustrates a signal line with negative double values recast as uint8.



To capture an image of the signal data, in the **Capture** section, click **Screenshot** (). In the Save a screenshot dialog box, specify a name and graphic type for the file.

When you are done drawing the signal, in the **Insert** section:

1. To add the signal to the Signal Editor, click **Insert signal** ().
2. To return to the main Signal Editor window and check that the signal has been added, click **Close draw tab** (). Signal Editor discards any noninserted signal data.
3. To observe the drawn signal data In Signal Editor, click the signal name in the hierarchy.

For an example of how to add a freehand ramp signal, see “Draw a Ramp Using Snap to Grid for Accuracy” on page 69-12.

See Also

Signal Editor | `signalEditor`

More About

- “Create and Edit Signal Data” on page 69-2
- “Use Scenarios and Insert Signals in Signal Editor” on page 69-15
- “Work with Basic Signal Data” on page 69-20
- “Create Signals with MATLAB Expressions and Variables” on page 69-24

Import Custom File Type

To edit signals that are in your own custom data format or a non-MAT-file, use the **Open > Import** option on the **Signal Editor** tab. Custom file types:

- Are external to MATLAB or Simulink, such as Microsoft Excel or JSON format files.
- Contain signal data whose format does not conform to those listed in “Forms of Input Data” on page 70-36.

Simulink provides these file types.

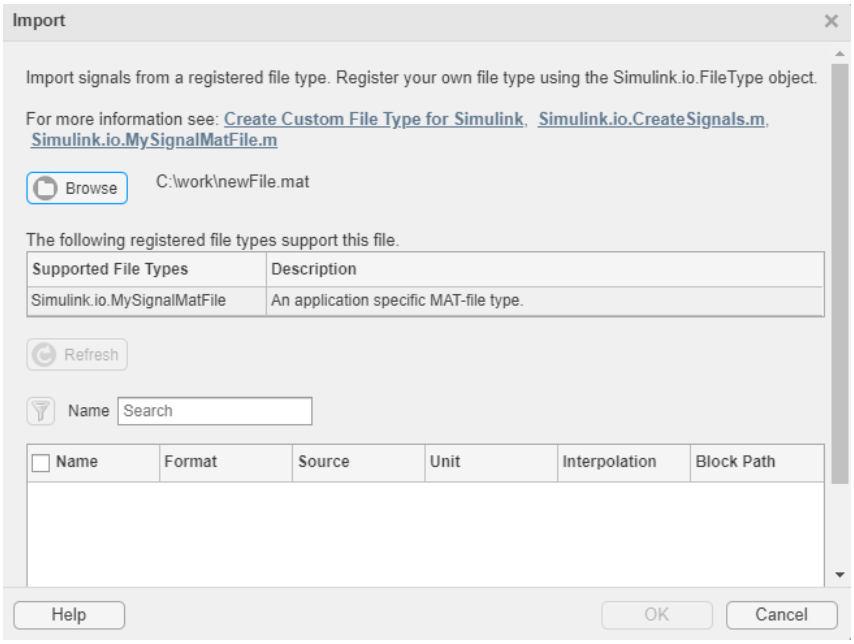
- `Simulink.io.SignalBuilderSpreadsheet` — Signal Builder file type
- Example file types
  - `Simulink.io.MySignalMatFile`
  - `Simulink.io.CreateSignals`

To import custom file types that you develop in-house, you can create and register your own custom file type reader. For information, see “Create Custom File Type for Import to Signal Editor” on page 69-36. Afterwards, use this workflow to import custom file types into Signal Editor.

Note Before starting, check that nobody is editing the custom file type class file. Editing the custom file type class file while trying to import it as a reader causes unexpected behavior.

1. Check that your custom file types have been registered in Simulink. In the **Signal Editor** tab, select **Open > Import**. The **Export** dialog box displays.
2. Click **Browse**.
3. From the list of custom MAT-files, select the one that contains your signals, such as *`custompath`*`/newFile.mat`.

   All registered file types appear in the Supported File Types table.



4 Select the signals you want to see in Signal Editor and click **OK**. To search for a signal name, enter it in **Name**.

5 In the **Signal Editor** tab, observe that the signals are now in the hierarchy. You can edit these signals as you would any signal.

See Also

Signal Editor | `signalEditor`

More About

- “Create Custom File Type for Import to Signal Editor” on page 69-36
- “Export Signals to Custom Registered File Types” on page 69-39

Create Custom File Type for Import to Signal Editor

By default, Simulink supports signals in the forms listed in “Forms of Input Data” on page 70-36. To import file types containing signals that are not of the supported format, create and register your own custom file type reader. Simulink supports custom file type readers written with `Simulink.io.FileType`.

Simulink provides these file types.

- `Simulink.io.SignalBuilderSpreadsheet` — Signal Builder file type
- Example file types
  - `Simulink.io.MySignalMatFile`
  - `Simulink.io.CreateSignals`

Creating a file reader requires you to be familiar with object-oriented programming. It is intended for an advanced audience.

1. To contain your package folders, create a folder and add that folder path to the MATLAB path.
2. To that folder, add the custom file that contains your signals, such as `mySignals.mat`.

   In that folder, create a `+Simulink` folder, and inside that folder, create a `+io` folder.
3. Create a class that inherits from the `Simulink.io.FileType`.

   ```
   classdef MyFileType < Simulink.io.FileType
   ```
4. Save this class to *`yourfolder`*`/+Simulink/+io`.
5. To register and interact with Signal Editor, implement these static methods:
   - `Simulink.io.FileType.isFileSupported`
   - `Simulink.io.FileType.getFileTypeDescription`
6. Implement these public methods:
   - `validateFileNameImpl`
   - `whosImpl`

     At run time, call `whosImpl` via `whos` when you run the `Simulink.io.FileType` object. `whos` has the same syntax as `whosImpl`.
7. Check if your class is registered. In the **Signal Editor** tab, select **Open** > **Import**, and in the **Import** dialog box window, click **Browse**.

   The custom file that contains your signals, such as *`custompath`*`/mySignals.mat`, appears in the file browser.
8. Select the custom file that contains your custom signals.
9. Return to the class file and implement these additional public methods:
   - `loadAVariableImpl`
   - `loadImpl`

     At run-time, call `loadImpl` via `load` when you run the `Simulink.io.FileType` object. `load` has the same syntax as `loadImpl`.

10 To import the custom signals, use the `import` method.

   `dataOnFile` `= import(`*`reader`*`)`, where *reader* is the file type object for the reader, specified as a `Simulink.io.FileType` object. The output, *dataOnFile*, is a structure with the fields *`structure`*`.Data`, which is a cell array of signals, and *`structure`*`.Name`, which is a cell array of the corresponding signal names. For example, *`dataOnFile`*`.Data` is the cell array of signals and *`dataOnFile`*`.Name` contains the corresponding signal names.

11 Return to the Signal Editor **Open** > **Import** and try to import again.

After you successfully import your custom signals, you can manipulate them in Signal Editor. When done, and if you have implemented the `exportImpl` method, you can export the results by calling the `export` method for your reader at run time. Alternatively, you can use the export dialog from “Export Signals to Custom Registered File Types” on page 69-39.

For example implementations, see:

- `open('Simulink.io.CreateSignals')` — Implementation of how to create signals
- `open('Simulink.io.MySignalMatFile')` — Implementation of how to register custom file types to import into Simulink.

Define New FileType Object for Use in Simulink

A `FileType` object is a component you can use to create readers for signals that exist in formats not currently supported in Simulink. Write the reader in MATLAB and use the Signal Editor **Open** > **Import** option to register the reader and import the custom format file.

Note Before importing, check that all editors for the custom file type class file are closed. Editing the custom file type class file while trying to import it as a reader causes unexpected behavior.

Define FileType Object

1 Create a `FileType` object for use in Simulink. This example creates a reader for signals with a custom format.

2 Create a class definition text file to define your `FileType` object.

3 On the first line of the class definition file, specify the name of your `FileType` and subclass from `Simulink.io.FileType`. The `Simulink.io.FileType` base class enables you to use all the basic `FileType` object methods.

4 For your class:

   a Add the appropriate basic `FileType` object methods to register and interact with Signal Editor.

   b Validate the signal formats.

   c Determine the contents of the signal file.

   d Load variables from the signal file.

   e Import the signals.

   See the reference pages for each method and the full class definition file below for the implementation of each of these methods. To see the full class definition for a custom signal reader, run: `open('Simulink.io.MySignalMatFile')`.

See Also

`Simulink.io.FileType` | `Simulink.io.FileType.isFileSupported` | `Simulink.io.SignalBuilderSpreadsheet` | `exportImpl` | `getFileTypeDescription` | `loadAVariableImpl` | `loadImpl` | `validateFileNameImpl` | `whosImpl`

More About

- “Import Custom File Type” on page 69-34
- “Export Signals to Custom Registered File Types” on page 69-39
- “Forms of Input Data” on page 70-36

Export Signals to Custom Registered File Types

To export signals from Signal Editor to your own custom file types, on the **Signal Editor** tab, select **Save** > **Export**. Custom file types:

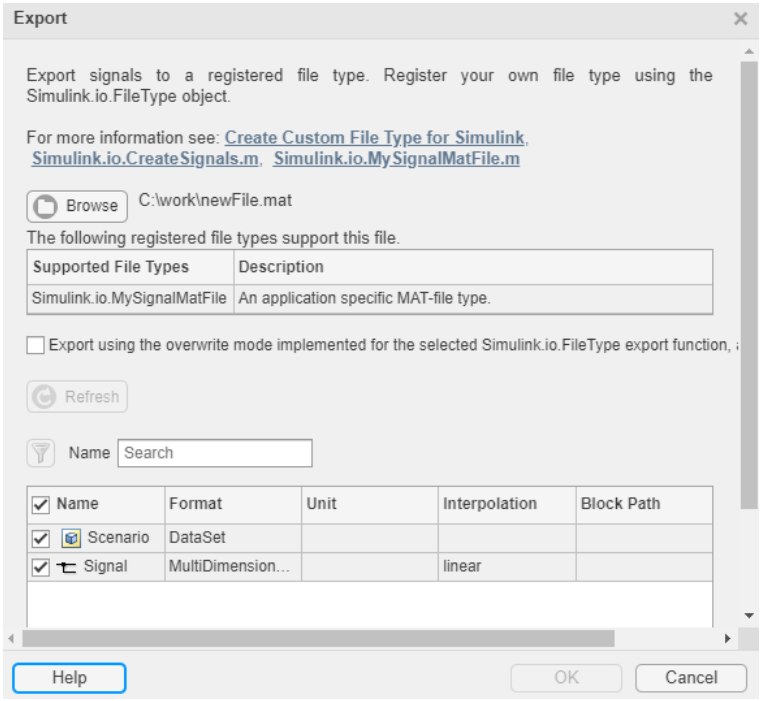
- Are external to MATLAB or Simulink, such as Microsoft Excel or JSON format files.
- Contain signal data whose format does not conform to those listed in “Forms of Input Data” on page 70-36.
- When exporting a `Simulink.io.SignalBuilderSpreadsheet` file type, it must contain a dataset represented with `Simulink.SimulationData.Dataset` objects that contain `timeseries` data with vector data.

Simulink provides these file types.

- `Simulink.io.SignalBuilderSpreadsheet` — Signal Builder file type
- Example file types
  - `Simulink.io.MySignalMatFile`
  - `Simulink.io.CreateSignals`

1. In Signal Editor, create signals to export to your custom file type.
   a. Click **Scenario** > **Blank Scenario**.
   b. Click **Signal** > **Author Signal**.
   c. Click **Show Examples**. Select **Apply Example** for the 2-D column vector.
2. Check that your custom file types have been registered in Simulink. In the **Signal Editor** tab, select **Save** > **Export**. The **Export** dialog box displays.
3. Click **Browse**.
4. From the list of custom MAT-files, select the one that contains your signals, such as *custompath*`/newFile.mat`.

   All registered file types appear in the Supported File Types table.



The signals in your custom file type appear in the file browser.

5 Select the signals you want to export from Signal Editor, then click **OK**.

See Also

`Simulink.io.FileType` | `Simulink.io.FileType.isFileSupported` | `Simulink.io.SignalBuilderSpreadsheet` | `exportImpl` | `getFileTypeDescription` | `loadAVariableImpl` | `loadImpl` | `validateFileNameImpl` | `whosImpl`

More About

- “Import Custom File Type” on page 69-34
- “Forms of Input Data” on page 70-36

Load Simulation Input Data

Provide Signal Data for Simulation

In this section...

A Simulink model performs algorithms on input signal data and produces output signals. The model defines what input data to use at the start of simulation and what output to capture at the end of simulation. As you create and simulate your model, you:

1. “Identify Model Signal Data Requirements” on page 70-2
2. “Load Input Signal Data” on page 70-5
3. “Log Output Signal Data” on page 70-6

As you create, debug, and test a model, you can use different sets of input signal data for simulation. You can use logged simulation data as input to another simulation.

Identify Model Signal Data Requirements

To use system-generated signal data, use source blocks such as a Sine Wave block. Source blocks do not require the use of a variable or external data source. If you cannot configure source blocks to meet your modeling requirements, then supply the signal data.

As you determine your signal data requirements, identify the:

- Blocks (including subsystems and Model blocks) that you need to provide data for — Design interfaces for blocks and for model components, including data types of signals.
- Range characteristics of signals, such as sample time, dimensions, and data type.
- Storage location for data for each input signal — Determine where to store signal data: in workspace variables, a MAT-file, or an external data file such as an Microsoft Excel spreadsheet.

Create a list of equation variables and constant coefficients, and then determine the coefficient values from published sources or by performing experiments on the system.

For information about storage locations for signal data, see “Signal Data Storage for Loading” on page 70-2.

Signal Data Storage for Loading

- “MATLAB Workspace for Signal Data” on page 70-3
- “Source and Signal Editor Blocks for Signal Data” on page 70-4
- “MAT-Files for Signal Data” on page 70-4
- “Spreadsheets for Signal Data” on page 70-4

Store signal data for loading into a model in these locations:

- MATLAB (base) workspace, or function workspace
- Model workspace
- Function workspace
- Masking workspace
- Blocks
- MAT-files
- Spreadsheets

The MATLAB (base) workspace is the most common workspace to use for loading signal data.

MATLAB Workspace for Signal Data

Consider using the MATLAB (base) workspace when you want to:

- Use a small amount of signal data for iterative simulations.
- Use signal data logged during one simulation as input for another simulation.
- Have multiple models use the same signal data.

Create Signal Data in the MATLAB Workspace

- At the MATLAB command line or editor, create the signal data.
- Use the `xlsread` function to read data from an Excel spreadsheet into the MATLAB workspace.
- Use the `csvread` function to read data from a CSV spreadsheet into the MATLAB workspace.
- Use a model callback to load signal data.
- Use one of these Simulink logging techniques:
  - Signal logging
  - To Workspace block
  - Scope block
  - The **Configuration Parameters** > **Data Import/Export** pane, the **Output**, **States**, or **Final states** parameters.
  - Data store
  - The `sim` command configured to log simulation data

Load Signal Data from the MATLAB Workspace

To load signal data from a workspace, use one of these techniques:

- Add a From Workspace block.
- Use a root-level input port.
  - Specify workspace variables in the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.
  - Use the Root Inport Mapper tool to specify the data for the **Input** parameter.

Source and Signal Editor Blocks for Signal Data

Source blocks, such as the Sine Wave block, generate signals that you can use as inputs to other blocks. Source blocks do not store signal data. Source blocks can be useful for initial prototyping of a model when the generated signal data serves your modeling requirements.

To define scenarios to use as inputs to a model, you can use the Signal Editor block. The Signal Editor block stores the scenario definitions.

Consider using a source block to:

- Avoid having to create the data manually.
- Reduce memory consumption — source blocks do not store signal data.
- Graphically represent in the model the kind of signal data.

Consider using a Signal Editor block to:

- Create and import scenarios for use in testing.

  You can use scenarios with Simulink and with these products:

  - Simulink Test
  - Simulink Coverage
  - Simulink Design Verifier
- Switch between scenarios quickly.

MAT-Files for Signal Data

Consider storing signal data in a MAT-file to:

- Load a large amount of signal data efficiently.
- Reuse the same signal data in different models.
- Reduce memory requirements for the model.
- Use different sets of signal data with the same model, with minimal model updates.

Store Signal Data in a MAT-File

To create a MAT-file to store signal data to import, you can use:

- A To File block
- The Signal Editor user interface
- MATLAB to create signal data that you store in a MAT-file
- `Simulink.saveVars` function to save to a MAT-file the simulation signal data that Simulink stores as workspace variables

Load Signal Data from a MAT-File

To load signal data from a MAT-file into a model, you can use a From File block.

Spreadsheets for Signal Data

Consider using an Excel or CSV spreadsheet to:

- Use an existing spreadsheet that already has the necessary signal data or that you can update easily to contain the signal data.
- Load a large amount of signal data efficiently.
- Reduce memory requirements for the model.
- Use different sets of signal data with the same model, with minimal model updates.
- Share the signal data with other people who do not have Simulink installed.

Store Signal Data in a Spreadsheet

Use one of these approaches:

- Create the signal data directly in the spreadsheet. For spreadsheet requirements, see “Storage Formats” on page 70-62.
- Export MATLAB signal data to an Excel or CSV spreadsheet using the `xlswrite` or `csvwrite` function.

Load Signal Data from a Spreadsheet

Use the From Spreadsheet.

The From Spreadsheet block loads Microsoft Excel on all platforms. This block loads CSV spreadsheets only on Microsoft Windows platforms.

The From Spreadsheet block incrementally loads the data directly from the spreadsheet, to minimize memory consumption.

Load Input Signal Data

You can use various sources for input signal data for simulating a model. You can:

- Use existing data from a file, such as a spreadsheet.
- Write a MATLAB script to define variables for the signal data. For example, you can create `Dataset` format data that you can use with all the signal loading techniques.
- Use data logged from a previous simulation.

You can use several different approaches to load data into a model, including:

- Root-level input ports — Import signal data from a workspace, using the **Input** configuration parameter to import it to a root-level input port of a Inport, Enable, or Trigger block. You can specify the input data directly in the **Input** parameter. To import multiple signals to root-level input ports, consider using the Root Inport Mapping tool on page 71-2. That tool updates the **Input** parameter based on the signal data that you import and map to root-level input ports.
- Source blocks — Add a source block, such Sine Wave block, to generate signals to input to another block.
- From File block — Read data from a MAT-file, outputting the data as a signal.
- From Spreadsheet block — Read data from Microsoft Excel spreadsheets or CSV spreadsheets, outputting the data as one or more signals.

To determine the approach to meet the input signal data requirements of your model, see “Comparison of Signal Loading Techniques” on page 70-21.

Log Output Signal Data

You can save signal values to the MATLAB workspace or to a MAT-file during simulation for later retrieval and postprocessing. Saving simulation data is also known as logging or exporting simulation data.

To determine which approach to use for logging signal data, see “Export Simulation Data” on page 72-2.

Saving simulation data in `Dataset` format simplifies postprocessing by providing a common format for the results of various logging techniques. Using `Dataset` format stores the data as MATLAB `timeseries`objects, which you can process with MATLAB. Simulink provides tools for converting data logged in other formats to `Dataset` format.

For more information about logging output signal data, see “Save Run-Time Data from Simulation”.

See Also

Blocks
From File | From Spreadsheet | From Workspace | To File | To Workspace

Related Examples

- “Signal Data Storage for Loading” on page 70-2
- “Comparison of Signal Loading Techniques” on page 70-21
- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Export Simulation Data” on page 72-2
- “Create and Edit Signal Data” on page 69-2

Load Big Data for Simulations

In this section...

Simulating models with many time steps and signals can use and create data that is too large to fit into working memory on your computer. When your simulation input data does not fit into memory, you can choose one of several strategies to use that data as simulation input with root-level Inport blocks. These strategies work for loading data stored in `Dataset` format in a Version 7.3 MAT-file, including data logged from another simulation.

1. When individual input signals are too large to fit into memory, you can use a `matlab.io.datastore.SimulationDatastore` object to access the signal data. The data from the `SimulationDatastore` object loads into the simulation incrementally in chunks that fit into memory.
2. When your simulation inputs are specified by a `Simulink.SimulationData.Dataset` object in a file that is too large to load into memory, you can stream the entire contents of the `Dataset` object into your model using a `Simulink.SimulationData.DatasetRef` object.
3. When the signals fit into memory and are stored in a file that is too large to load into memory, you can load individual signals from the file into memory using a `Simulink.SimulationData.DatasetRef` object.

Note When you want to use data logged in one simulation as input for another, you can also stream data into the model using a `matlab.io.datastore.sdidatastore` object. The `sdidatastore` object references data in the Simulation Data Inspector repository on disk, so you do not have to save the logged data to a file. Consider using a `sdidatastore` object as simulation input for iterative workflows.

All big data loading strategies are for the special case when your data does not fit into memory and can require extra steps. These examples use data that fits fully into memory to illustrate the steps required for big data loading. When your simulation inputs fit into memory, consider using other loading techniques.

Stream Individual Signals Using SimulationDatastore Objects

When individual signals in your input data are too large to fit into memory, you can create `matlab.io.datastore.SimulationDatastore` objects for those signals and stream them into your model. To create a `SimulationDatastore` object for a signal you want to stream into your model, first create a `Simulink.SimulationData.DatasetRef` object to reference the `Dataset` object that contains your signal of interest. For example, create a `DatasetRef` for logged data from a simulation of the `slexAircraftExample` model.

```
logsout_DSR = Simulink.SimulationData.DatasetRef('aircraftData.mat','logsout');
```

You can create a `SimulationDatastore` object for your desired signal by indexing into the `DatasetRef` object with curly braces or using the `getAsDatastore` method for the `DatasetRef`

object. In both cases, the `SimulationDatastore` object exists in the `Values` property of the returned `Simulink.SimulationData.Signal` object.

When you know the index of the signal within the `Dataset` object, you can index into the `DatasetRef` object with curly braces to create a `SimulationDatastore` for your signal.

```
alphaRad_ds = logsout_DSR{4}
```

```
alphaRad_ds =
  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
             Name: 'alpha, rad'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 4
            Values: [1x1 matlab.io.datastore.SimulationDatastore]


  Methods, Superclasses
```

To create a `SimulationDatastore` object for a signal using the signal index, name, or block path, use the `getAsDatastore` method. For example, create a `SimulationDatastore` object for the `Stick` signal.

```
stick_ds = logsout_DSR.getAsDatastore('Stick')
```

```
stick_ds =
  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
             Name: 'Stick'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [1x1 matlab.io.datastore.SimulationDatastore]


  Methods, Superclasses
```

Because the `Values` properties of the `stick_ds` and `alphaRad_ds` `Simulink.SimulationData.Signal` objects are `SimulationDatastores`, the signal data streams into your model. You can include a `SimulationDatastore` backed `Signal` object as an element in a `Dataset` object or as an item in the `Input` parameter comma-separated list.

Stream an Entire Dataset Using a DatasetRef Object

When your simulation inputs are specified in a `Dataset` in a file that is too large to load into memory, you can create a `Simulink.SimulationData.DatasetRef` object to stream your simulation inputs into your model. When you specify a `DatasetRef` object for the Input parameter on the Data Import/Export pane, all the signals in the `Dataset` object used to create the `DatasetRef` stream into your

model. Use one of the other big data loading techniques to load or stream individual signals from a `Dataset` object.

When the file where your simulation input `Dataset` is stored contains other `Datasets` and data, you can use the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function to view a list of the `Dataset` objects contained in the file. Previewing the variable names in the `Dataset` object is particularly useful when the file contents do not fit into memory.

```
datasetNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('aircraftData.mat')
```

```
datasetNames = 1x3 cell
    {'logsout'}    {'xout'}    {'yout'}
```

Create a `DatasetRef` object for `logsout`.

```
logsout_DSR = Simulink.SimulationData.DatasetRef('aircraftData.mat','logsout');
```

You can load `logsout_DSR` using the `Input` parameter the same way you would load a `Simulink.SimulationData.Dataset` object. Each signal in the `Dataset` object used to create the `DatasetRef` streams into the model in chunks that fit into memory.

Load Individual Signals from a DatasetRef Object

When your simulation input signals individually fit into memory and are stored in a `Dataset` object in a file that does not fit into memory, use a `Simulink.SimulationData.DatasetRef` object to load each signal of interest into memory. Then, you can load the signals as simulation inputs for your model.

First, create the `DatasetRef` object to reference the `Dataset` object in the file that contains the signals you want to load. For example, create a `DatasetRef` object for data logged to file from a simulation of the `slexAircraftExample` model.

```
logsout_DSR = Simulink.SimulationData.DatasetRef('aircraftData.mat','logsout');
```

You can use the `get` or `getElement` methods to load individual signals into memory with the `DatasetRef` object. Both methods load the specified element into memory, using the same syntax. You can specify the signal you want to load into memory using its index within the `Dataset` object or its name. If you don't know the name of the signal you want to load, use the `getElementNames` method to see the names of the elements in the `Dataset` object referenced by the `DatasetRef` object.

```
elNames = logsout_DSR.getElementNames
```

```
elNames = 15x1 cell
    {0x0 char    }
    {0x0 char    }
    {0x0 char    }
    {'alpha, rad'}
    {'q, rad/sec'}
    {0x0 char    }
    {'qGust'     }
    {'wGust'     }
    {0x0 char    }
    {0x0 char    }
```

```
    {0x0 char    }
    {'Stick'     }
    {0x0 char    }
    {0x0 char    }
    {0x0 char    }
```

Load the `qGust` signal into memory using its name.

```
qGust = logsout_DSR.getElement('qGust')
```

```
qGust =
  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
             Name: 'qGust'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 2
            Values: [1x1 timeseries]


  Methods, Superclasses
```

You can add the `qGust` signal to a `Dataset` object of simulation input signals to load to the root-level Inport blocks in your model, or you can specify `qGust` as an item in the `Input` parameter comma-separated list.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `matlab.io.datastore.SimulationDatastore`

Related Examples

- “Work with Big Data for Simulations” on page 72-29
- “Log Data to Persistent Storage” on page 72-31
- “Analyze Big Data from a Simulation” on page 72-35

Stream Data from a MAT-File as Input for a Parallel Simulation

This example shows how to use `Simulink.SimulationData.DatasetRef` objects and the `parsim` function to stream input data from a version 7.3 MAT-file for parallel simulations. Consider following the steps outlined in this example when the inputs for your simulation are too large to load into memory. For example, you can use data logged to persistent storage from one set of parallel simulations as input for another.

This example uses `parsim` to run multiple simulations of a model, with each simulation using unique input data. The model is based on the sldemo_suspn_3dof model, modified to use Inport blocks as the source for inputs instead of the Signal Editor block. The model simulates the response of a suspension system to different road conditions. The MAT-file used in this example contains multiple `Simulink.SimulationData.Dataset` objects representing various road conditions. The example uses `DatasetRef` objects to stream the contents of an entire referenced `Dataset` object as simulation input.

You can also stream data for individual signals into parallel simulations run with `parsim` using `matlab.io.datastore.SimulationDatastore` objects. For details on creating `SimulationDatastore` objects, see “Stream Individual Signals Using SimulationDatastore Objects” on page 70-7.

Load Model and Access Input Data

Load the `ex_sldemo_suspn_3dof_parsim_stream` model. The model receives input data through two Inport blocks, and each `Dataset` object used as simulation input contains two elements: one for each Inport.

```
mdl = 'ex_sldemo_suspn_3dof_parsim_stream';
open_system(mdl)
```

You can use the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function to evaluate the contents of the MAT-file containing the input data without loading the data into memory. The function returns a cell array that contains elements for the name of each `Simulink.SimulationData.Dataset` variable the file contains. Use the function to access the variable names and determine the number of test cases in the file.

```
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('suspn_3dof_test_cases.mat
numTestCases = numel(varNames);
```

You can stream the test case data into the model using `Simulink.SimulationData.DatsetRef` objects. The `DatasetRef` object references a variable in the file used to create it and loads the variable data incrementally. Create a `DatasetRef` object for each `Simulink.SimulationData.Dataset` object in the test cases file.

```
for idx1 = 1:numTestCases
    inputData(idx1) = Simulink.SimulationData.DatasetRef('suspn_3dof_test_cases.mat',...
                                                          varNames{idx1});
end
```

Configure and Run Parallel Simulations

To use the set of test case inputs as input for a set of parallel simulations, create an array of `Simulink.SimulationInput` objects that you can pass to the `parsim` function. Use the `setExternalInput` function to specify a `Simulink.SimulationData.DatasetRef` object corresponding to a test case as data to stream as simulation input.

```
in(1:numTestCases) = Simulink.SimulationInput(mdl);

for idx2 = 1:numTestCases
    in(idx2) = setExternalInput(in(idx2),inputData(idx2));
end
```

Use the `parsim` function to run a simulation for each test case. When you have the Parallel Computing Toolbox™, the `parsim` function runs simulations in parallel. Without the Parallel Computing Toolbox, the `parsim` function runs the simulations in serial.

The `parsim` function creates a worker pool based on the Parallel Computing Toolbox configuration. By default, `parsim` uses a local pool. If you use remote workers, you can use the `AttachedFiles` name-value pair to send the MAT-file containing the test case input data to each worker. When you specify the `AttachedFiles` name-value pair, `parsim` sends a copy of the file to each worker, which can take some time for large files. For streaming input data from a large file, local workers may be faster because the workers have access to the file without creating and sending copies. When you use remote workers, consider storing the MAT-file in a location that all remote workers can access and creating `DatasetRef` objects that reference that copy of the file.

```
out = parsim(in);
```

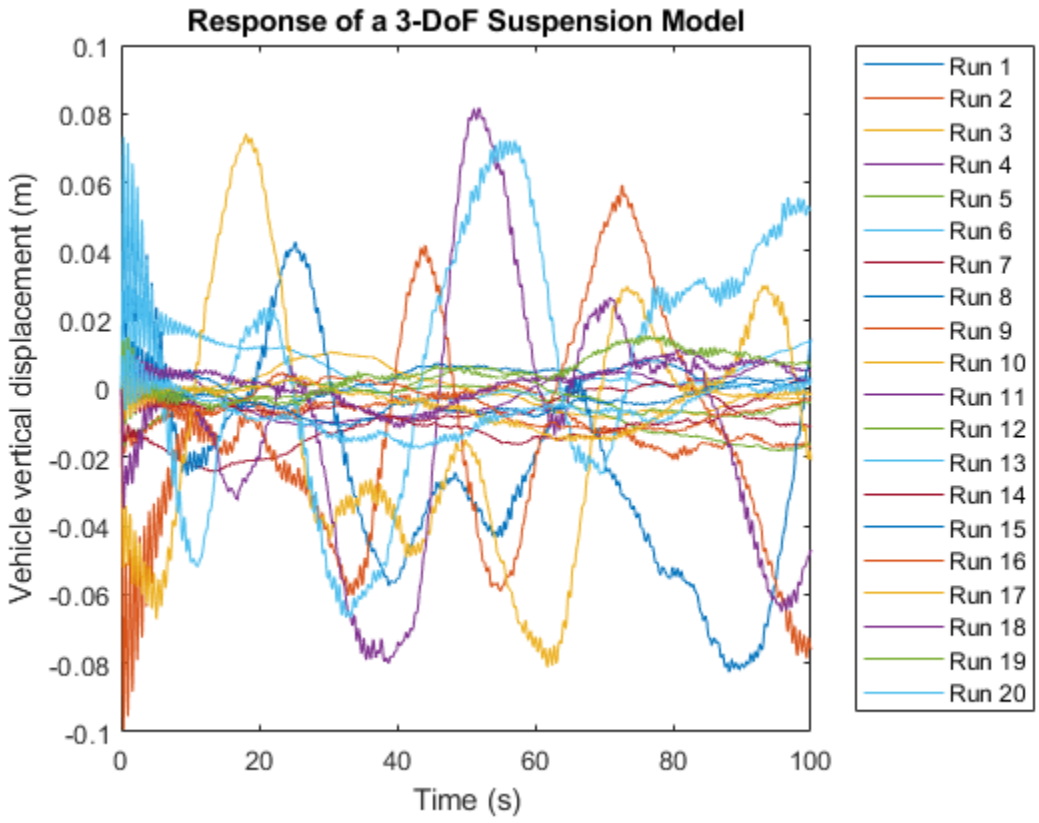
```
[20-May-2020 08:58:17] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
[20-May-2020 08:59:10] Starting Simulink on parallel workers...
[20-May-2020 08:59:50] Configuring simulation cache folder on parallel workers...
[20-May-2020 08:59:50] Loading model on parallel workers...
[20-May-2020 09:00:02] Running simulations...
[20-May-2020 09:00:18] Completed 1 of 20 simulation runs
[20-May-2020 09:00:18] Completed 2 of 20 simulation runs
[20-May-2020 09:00:18] Completed 3 of 20 simulation runs
[20-May-2020 09:00:18] Completed 4 of 20 simulation runs
[20-May-2020 09:00:19] Completed 5 of 20 simulation runs
[20-May-2020 09:00:19] Completed 6 of 20 simulation runs
[20-May-2020 09:00:21] Completed 7 of 20 simulation runs
[20-May-2020 09:00:21] Completed 8 of 20 simulation runs
[20-May-2020 09:00:21] Completed 9 of 20 simulation runs
[20-May-2020 09:00:21] Completed 10 of 20 simulation runs
[20-May-2020 09:00:21] Completed 11 of 20 simulation runs
[20-May-2020 09:00:21] Completed 12 of 20 simulation runs
[20-May-2020 09:00:24] Completed 13 of 20 simulation runs
[20-May-2020 09:00:24] Completed 14 of 20 simulation runs
[20-May-2020 09:00:24] Completed 15 of 20 simulation runs
[20-May-2020 09:00:24] Completed 16 of 20 simulation runs
[20-May-2020 09:00:24] Completed 17 of 20 simulation runs
[20-May-2020 09:00:24] Completed 18 of 20 simulation runs
[20-May-2020 09:00:27] Completed 19 of 20 simulation runs
[20-May-2020 09:00:27] Completed 20 of 20 simulation runs
[20-May-2020 09:00:27] Cleaning up parallel workers...
```

View Simulation Results

You can access the simulation results programmatically when the simulations finish. Create a plot showing the vertical displacement for the vehicle for all the road profile test cases.

```
if isempty(out(1).ErrorMessage)
    legend_labels = cell(1,numTestCases);
```

```
    for i = 1:numTestCases
        if isempty(out(i).ErrorMessage)
            simOut = out(i);
            ts = simOut.logsout.get('vertical_disp').Values;
            ts.plot;
            legend_labels{i} = ['Run ' num2str(i)];
        end
            hold all
    end
    title('Response of a 3-DoF Suspension Model')
    xlabel('Time (s)');
    ylabel('Vehicle vertical displacement (m)');
    legend(legend_labels,'Location','NorthEastOutside');
end
```



You can also view `parsim` simulation results using the Simulation Manager. To view results in the Simulation Manager, use the `ShowSimulationManager` name-value pair for `parsim`. With the Simulation Manager, you can monitor the progress of the runs, view simulation data, and show the `parsim` results in the Simulation Data Inspector.

Close Parallel Workers

When you have finished running parallel simulations, you can close the worker pool.

```
delete(gcp('nocreate'));
```

Related Topics

“Load Big Data for Simulations” on page 70-7

“Log Data to Persistent Storage” on page 72-31

“Run Parallel Simulations Using parsim” on page 27-5

Overview of Signal Loading Techniques

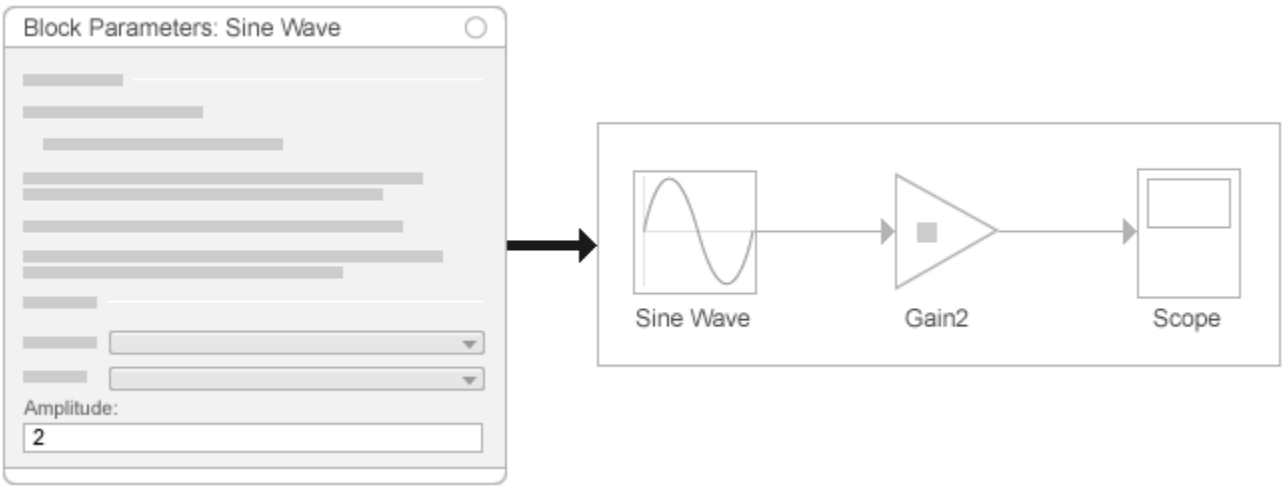
In this section...

Simulink provides several techniques for importing signal data into a model. Each of the signal data loading techniques uses blocks to represent signal data sources visually.

For additional details about which technique to use to meet specific modeling requirements, see “Comparison of Techniques” on page 70-22.

Source Blocks

You can add a source block, such as a Sine Wave block, to generate signals to input to another block. To specify how to generate the signal, use the Block Parameters dialog box. For example, in the Sine Wave Block Parameters dialog box, you can specify the `sim` function to use and time-based or sample-based data.



The output data types of source blocks vary. For example, a Sine Wave block outputs a vector of real doubles.

For an example of using a source block, see “Build and Edit a Model Interactively” on page 1-8.

Recommended Uses

- Do initial prototyping in a model, when the generated signal data serves your modeling requirements

- Avoid creating the data manually.
- Reduce memory consumption. Source blocks do not store signal data.
- Make the kind of signal data visually clear in the model.

Limitations

Source blocks generate signals based on a predefined algorithm. To use actual data from an external source or to test a model without having to modify the model, use a different signal loading technique.

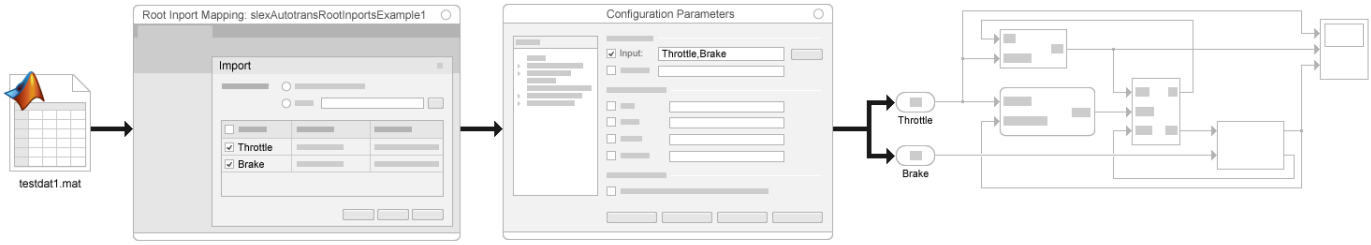
Root-Level Input Ports

You can import signal data from a workspace and apply it to a root-level input port using one of these blocks:

- Enable
- Inport
- Trigger block that has an edge-based (rising, falling, or either) trigger type

The root-level input ports load external inputs from the MATLAB (base), model, or mask workspace. These blocks import data from the workspace based on the value of the **Configuration Parameters > Data Import/Export > Input** parameter or a `sim` command argument. For an example, see “Load Data to Model a Continuous Plant” on page 70-29.

To import many signals to root-level input ports, consider using the Root Inport Mapper tool. This tool updates the **Input** configuration parameter based on the signal data that you import and map to root-level input ports. For an example, see “Map Data Using Root Inport Mapper Tool” on page 71-2.



Recommended Uses

Use root input ports to:

- Import many signals to many blocks
- Test your model as a referenced model in a wider context with signals from the workspace, without modifying your model

For importing signal data to meet most modeling requirements and to maintain model flexibility, root-level inport mapping is a convenient technique. Root-level inport mapping:

- Displays signal data for you to inspect without loading all the signal data into MATLAB memory
- Provides memory-efficient signal viewing

Requirements

To ensure that the Simulink variable solver executes at the times that you specify in the imported data, set the **Configuration Parameters** > **Data Import/Export** > **Additional parameters** > **Output options** parameter to `Produce additional output`.

Limitations

- You cannot use input ports to import buses in external modes. To import bus data in rapid accelerator mode, use `Dataset` format.
- The Root Inport Mapper tool supported map modes depend on the data type of a signal. For details, see “Choose a Base Workspace and MAT-File Format” on page 71-10.

From File Block

A From File block reads data from a MAT-file and outputs the data as a signal.

For an example, see “From File Block Loading Timeseries Data”.

Recommended Uses

Consider using a From File block for loading:

- Large amounts of data. For a Version 7.3 MAT-file, the From File block loads data incrementally from the MAT-file during simulation.

  Tip To convert a Version 7.0 file to Version 7.3 (for example, `my_data_file.mat` that contains the variable `var`), at the MATLAB command line, enter:

  ```
  load('my_data_file.mat')
  save('my_data_file.mat', 'var', '-v7.3')
  ```

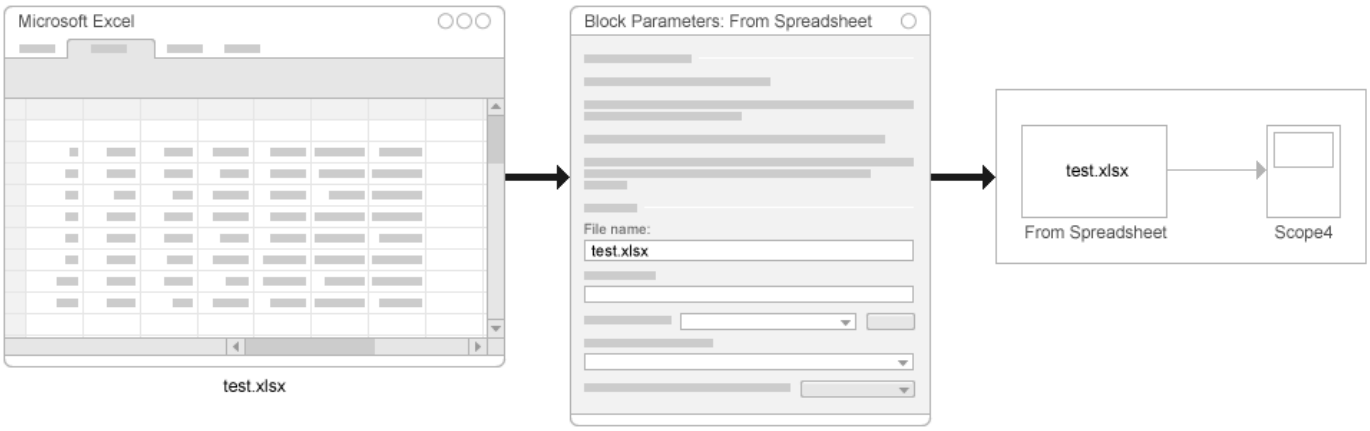
- Data that was exported to a To File block. The From File block reads data written by a To File block without any you modifying the data or making other special provisions.
- Data stored in a MAT-file that is separate from the model file.

Limitations

- For *Version 7.0* or earlier MAT-file, the From File block reads only array-format data.
- Version 7.3 and Version 7.0 or earlier MAT-files handle multiple variables differently. See “MAT-File Variable” on page 70-64.
- The From File block supports reading nonvirtual bus signals in MATLAB `timeseries` format.
- For array data, the From File block reads only double signal data.
- Code generation that involves building ERT or GRT targets, or using SIL or PIL simulation modes, has some special considerations. See “Code Generation”.

From Spreadsheet Block

The From Spreadsheet block reads data from Microsoft Excel spreadsheets (all platforms) or CSV spreadsheets (Microsoft Windows platform with Microsoft Office only) and outputs the data as one or more signals.



Recommended Uses

Use the From Spreadsheet block for loading:

- Large Microsoft Excel or CSV spreadsheets. The From Spreadsheet block incrementally reads data from the spreadsheet during simulation, rather than loading the data into Simulink memory.
- Spreadsheets that you expect to modify. The From Spreadsheet block handles changes to worksheet values automatically, because it loads data directly from the spreadsheet.

Limitations

- You cannot import bus data.
- The From Spreadsheet file has requirements for the spreadsheet data. Organize Excel spreadsheet data using the format described in “Supported Microsoft Excel File Formats” on page 71-12.
- Linux and Mac platforms do not support using a From Spreadsheet block to import data from a CSV spreadsheet.

From Workspace Block

The From Workspace block reads signal data from a workspace and outputs the data as a signal. In the Block Parameters dialog box, in the **Data** parameter, enter a MATLAB expression that specifies the workspace data.

For an example of how to use a From Workspace block, see “Use From Workspace Block for Test Case” on page 70-33.

Recommended Uses

Use the From Workspace block for loading:

- A small set of signal data to perform local, temporary testing
- Data from the MATLAB (base), model, mask, or function workspace
- Variable-size signals
- Data that you saved using a To Workspace block in MATLAB `timeseries` format, without manual changes to the data
- Data saved in a previous simulation by a To Workspace block in either `Timeseries` or `Structure with Time` format for use in a later simulation

Limitations

The data expressions that you specify must evaluate to one of these types of data:

- A `timeseries` or `timetable` object
- A structure of `timeseries` or `timetable` objects
- A structure, with or without time
- A two-dimensional matrix

Signal Editor Block

Using a Signal Editor block, you can create interchangeable scenarios to use in a model.

For examples of how to use a Signal Editor block, see:

- “Load Data for an Input Test Case” on page 70-32
- “Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode”

Recommended Uses

Use the Signal Editor block to create and load scenarios to use in testing.

These products integrate the Signal Editor block into their workflows:

- Simulink Test
- Simulink Coverage
- Simulink Design Verifier

Limitations

- Function-calls
- Array of buses
- Buses while using rapid accelerator mode
- `timetable` objects
- Ground signals

The Signal Editor block supports dynamic strings. It does not support strings with maximum length. In addition, strings in the Signal Editor block cannot output:

- Non-scalar MATLAB strings.
- String data that contains missing values.
- String data that contains non-ASCII characters.

See Also

Related Examples

- “Load Data to Root-Level Input Ports” on page 70-35
- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Load Data Using the From File Block” on page 70-60
- “Load Data Using the From Workspace Block” on page 70-65
- “Load Signal Data That Uses Units” on page 70-59

More About

- “Comparison of Signal Loading Techniques” on page 70-21

Comparison of Signal Loading Techniques

In this section...

Techniques

Simulink provides several techniques for importing signal data into a model. Each signal data loading technique uses a block to represent signal data sources visually. You can use a:

- Source on page 70-15 block, such as the Sine Wave block, to generate signal data as input to another block
- Root-level input port on page 70-16 (Inport, Enable, or Trigger block). Loading signal data to root-level input ports, either manually or by using the Root Inport Mapper tool. “Root-level input ports” refers to both approaches and “Root Inport Mapper tool” refers specifically to using that tool.
- From File on page 70-17 block
- From Spreadsheet on page 70-18 block
- From Workspace on page 70-19 block
- Signal Editor block

Impact of Loading Techniques on Block Diagrams

To test reusable systems, it is helpful to separate signal data loading from the block diagram. Loading root-level input ports provides a good framework for testing complex systems on an ongoing basis. Using the Root Inport Mapper tool allows you to visualize the signal data that is loaded.

To perform temporary testing on standalone models, adding data loading blocks can be simpler and make the source of the signal data visible from within the block diagram.

To avoid adding data loading blocks to a model, load the signal data to root-level input ports. You can change the data to use by changing the **Configuration Parameters > Data Import/Export > Input** parameter. You do not need to add or change blocks, or reset block parameters. You can use the Root Inport Mapper tool to update the **Input** parameter so that it reflects the mapping of signal data to the appropriate ports.

Test Harness Models

You can use a test harness model with different test cases to load:

- Different signal data to a port
- Signal data to different ports

The Signal Editor block is useful in test harness models to simplify loading data to multiple input ports.

Alternatively, you can use the Root Inport Mapper tool to create scenarios that you can use instead of creating separate test harness models. Creating separate test harness models can be simpler to

create than setting up root inport mapping. However, you then need to manage the separate test harness models. For an example of using root inport mapping instead of a test harness, see “Converting Harness-Driven Models to Use Harness-Free External Inputs” on page 71-22

Comparison of Techniques

Each technique addresses many of these modeling considerations:

Purpose of Importing Signal Data

The model development phase you are in and your goals for loading signal data can influence the signal loading technique that you choose.

Modeling Goal	Supported Techniques
Perform local, temporary testing by importing a small set of signal data	All From File, From Spreadsheet, and From Workspace blocks work well for this goal. Root-level input ports for reusable systems.
Test a model that you want to use as a referenced model	Root-level input ports.
Verify a model by using multiple test cases	Root Inport Mapper tool, using exported signal data. Signal Editor block.
Represent a continuous plant	All Root-level input ports work well for this goal.
Test a discrete algorithm	All Root-level input ports work well for this goal.

Model Development Phase

Modeling Requirement	Suggested Signal Loading Technique
Initial prototyping	Signal values that source blocks generate meet your requirements, use Source blocks on page 70-15. From File, From Spreadsheet, and From Workspace blocks.
System testing, sharing, and code generation	Root-level input port on page 70-16. You can use the Root Inport Mapper tool to create and map signal data to load Signal Editor block

For many models, loading signal data to a root Inport block is an effective approach. The Root Inport Mapping tool on page 71-2 provides a convenient way to load data for several signals to root inports.

Signal Data

The amount, source, and kind of the signal data can influence the signal loading technique that you choose.

Signal Data	Supported Techniques
Large data set	From File and From Spreadsheet blocks work well for large data sets, because they incrementally load the data. You can log big simulation data to persistent storage and then incrementally load data from a file to root-level Inport blocks.
Data exported by using a To File block	From File block.
Data exported by using a To Workspace block	From Workspace block.
Excel or CSV spreadsheet	From Spreadsheet block, which can import Microsoft Excel (all platforms) or CSV (Microsoft Windows platform with Microsoft Office only) spreadsheet data directly into Simulink.
Variable-size signals	From Workspace block.

Data Format or Type

Each of the signal loading techniques supports a wide range of data formats for signal data (such as array or `Dataset`). A few signal loading techniques have some limitations for specific formats.

Note Some of the Root Inport Mapper tool map modes do not support all the data types that you can use with the tool. For details, see “Choose a Base Workspace and MAT-File Format” on page 71-10.

Data Format or Type	Supported Techniques
Array	All. For array data in a Version 7.0 MAT-file, the From File block loads only double signal values. Use Version 7.3 MAT-files for other types of signal data.
Structure with time	All.
Structure without time	All.
MATLAB `timeseries`	All.
`Simulink.SimulationData.Dataset`	All.
Enumeration	All.
Fixed-point	From File block has a word length limit of 32 or fewer bits.
Function-call	Root-level input ports (select the **Output function call** parameter).

Bus Support

You can use any of the signal loading techniques to load bus data. However, for some kinds of bus data, you need to use a specific technique.

Type of Bus or Bus Element	Supported Techniques
Virtual and nonvirtual buses	All techniques support both types of buses. The Signal Editor block supports only non-virtual buses. Root-level input ports and the Signal Editor block do not support loading bus data in rapid accelerator mode.
Partial bus specification	From File and Signal Editor use ground values for unspecified bus elements.
Array of buses signals	Root-level input ports.

Time Points

The kind of time points in signal data impacts the signal loading technique that you choose.

Time Points for Signal Data	Supported Techniques
Single time point	All.
Continuous	All.
Discrete	All.
Repeated sequence without time	Structure data by using root-level input ports and From Workspace block.

Location for Data Storage

Whether you want to store the signal data with the model or separate from the model impacts the signal loading technique that you choose.

Location	Supported Techniques
In the base or model workspace	From Workspace block. Root-level input ports or a Trigger, Enable, or Function-Call Subsystem block.
In a MAT-file separate from the model file	From File and Signal Editor blocks. You can log big simulation data to persistent storage and then incrementally load data from a file to root-level Inport blocks.
In an Excel or CSV spreadsheet	From Spreadsheet block. **Tip** For Excel and CSV spreadsheet requirements, see “Storage Formats” on page 70-62. Loading CSV data is supported only for Microsoft Windows platforms.

Signal Data Inspection

The Root Inport Mapper tool, From File block, and Signal Editor block each provide an interface for plotting and inspecting the signal data to load.

Handling of Loaded Data

How Simulink processes the signal data as it loads it into a model impacts the signal logging technique that you choose.

Data Loading Handling	Supported Techniques
Incremental data loading	From File and From Spreadsheet blocks.
Interpolation	All.
Extrapolation	From File, From Spreadsheet, and Signal Editor blocks. For information about From Workspace extrapolation, see “Form output after final data value by”.
Zero-crossing detection	All except root-level input ports.
Fast restart	All techniques.

Simulation Mode

All signal loading techniques support all simulation modes except for SIL or PIL. Some techniques have limitations for specific simulation modes.

Simulation Modes	Supported Techniques
Normal and accelerator	All
Rapid accelerator	All, with these limitations: • Root-level input ports only support array and structure data formats. • The From Workspace block does not support `timeseries` format. • The Signal Editor block does not support buses in this mode.
ERT/GRT	All From Workspace and From File blocks are not tunable.
SIL or PIL	From Workspace block
External mode	From Workspace block Root-level input ports load ground values in external mode.

See Also

Related Examples

- “Load Data to Root-Level Input Ports” on page 70-35
- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Load Data Using the From File Block” on page 70-60
- “Load Data Using the From Workspace Block” on page 70-65
- “Load Big Data for Simulations” on page 70-7
- “Load Signal Data That Uses Units” on page 70-59

More About

- “Overview of Signal Loading Techniques” on page 70-15

Load Data Logged In Another Simulation

In this section...

A common source of signal data to load into a model is data that you log from a simulation. You can use the signal data captured from a simulation as roundtrip input to:

- Simulate the same model again from a known starting point.
- Test simulation results.
- Simulate another model starting with the captured signal values from a model. For example, you can log signal data when you simulate a model. Then load the signal data from that simulation as inputs to a second model that you want to reference from the first model.

You can capture the signal data from a simulation in a workspace or in a file. Use one of these techniques to capture signal data from a simulation:

- Signal logging
- To Workspace block
- To File block
- Scope block
- In the **Configuration Parameters** > **Data Import/Export** pane, the **Output**, **States**, or **Final states** parameters
- Data store
- The `sim` command configured to log simulation data

For an example of using simulation data for roundtrip signal data loading, see “Load Data to Model a Continuous Plant” on page 70-29.

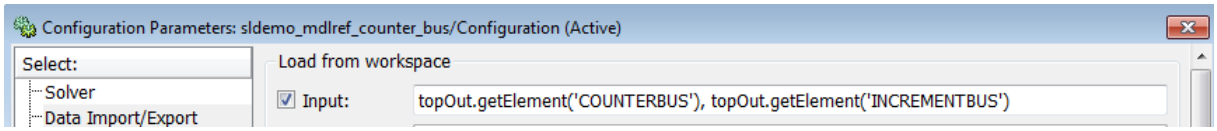
Load Logged Data

Here is a workflow for using signal logging data for standalone simulation of a referenced model. You can use a similar approach for other data logged in `Dataset` format.

1. Use the default signal logging output variable, `logsout`, or specify a variable using the **Configuration Parameters** > **Data Import/Export** > **Signal logging** edit box.
2. Simulate the parent model.

   The signal logging output is a `Simulink.SimulationData.Dataset` object.
3. Use the `Simulink.SimulationData.Dataset.getElement` method to access the logged data. The logging data for individual signals is stored in `Simulink.SimulationData.Signal` objects.
4. For the referenced model that you want to simulate standalone, use the `Simulink.SimulationData.Signal.getElement` method to specify signal elements for the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.

   For example:



5 Simulate the referenced model.

For an example of loading signal logging data for a model that uses model referencing, see the open the `sldemo_mdlref_bus` model. After you open the model, double-click the blue block labeled **Interface Specification** and see the sections called:

- Logging Model Reference Signals
- Loading Data

Also, the "Load Data to Model a Continuous Plant" on page 70-29 example illustrates loading signal logging data.

To import signal logging data for array of buses signals, see "Import Array of Buses Data" on page 70-49.

Configure Logging to Meet Loading Requirements

Different logging techniques support different data formats. Most logging techniques support the `Dataset` format, which provides a consistent data format for logged signal data. You can use the `Simulink.SimulationData.Dataset` constructor to convert other data formats to `Dataset` format.

To log only the data that you require, use the **Configuration Parameters** > **Data Import/Export** > **Logging intervals** parameter to specify start and stop time intervals.

See Also

Classes
`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.Signal`

Related Examples

- "Load Data to Model a Continuous Plant" on page 70-29
- "Save Run-Time Data from Simulation"
- "Export Signal Data Using Signal Logging" on page 72-41
- "Import Array of Buses Data" on page 70-49

Load Data to Model a Continuous Plant

A continuous plant model uses signal data that is smooth and uninterrupted in time. There is signal data for each time value. A continuous plant model uses a continuous solver (any solver other than an explicit discrete solver). The solver can be fixed-step or variable. The model includes blocks from the Continuous library in Simulink, such as an Integrator block.

To load data to represent a continuous plant, consider using either a root-level input port or a From Workspace block. Using a From Workspace block can be useful when loading data to a port buried deep within a model.

For the signal data:

- Specify a time vector and signal values extracted from a continuous plant. For example, extract from data that you acquire experimentally or from the results of a previous simulation.
- Use any of the data formats listed in “Specify Input Data” on page 70-35. Here are recommended formats for the following imported data sources:
  - Another simulation — `Dataset`
  - An equation — MATLAB time expression
  - Experimental data — MATLAB `timeseries`, structure with time, a structure without time, or a data array

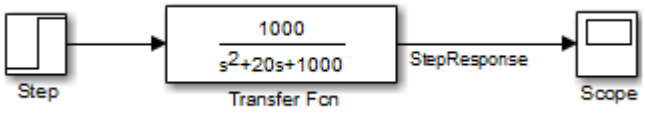
  For structure data, see “Specify Time Data” on page 70-40.

Use Simulation Data to Model a Continuous Plant

This example illustrates how to use logged data from the simulation of one model in the simulation of a second model. For more information, see “Load Data Logged In Another Simulation” on page 70-27.

When using data from a simulation that uses a variable-step solver for simulation in another model, the second simulation must read the data at the same time steps as the first simulation.

1. Open the `ex_data_import_continuous` model.



   This model uses the `ode15s` solver and produces continuous signals.

2. To use the output of this model as input to the simulation of another model, log the signal that you want to use. In the Simulink Editor, select that signal, and click **Log Signals**.

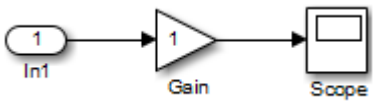
   Note To enable signal logging, select the **Configuration Parameters** > **Data Import/Export** > **Signal logging** parameter. This model has **Signal logging** enabled.

3. Simulate the model.

   Simulating the model saves the logged signal to the workspace in the `Simulink.SimulationData.Dataset` object, `logsout`.

Use the `Simulink.SimulationData.Dataset.getElement` method to access the logged data. The logged data for an individual signal is stored in a `Simulink.SimulationData.Signal` object. The `Dataset` object created by this model contains one logged signal: `StepResponse`.

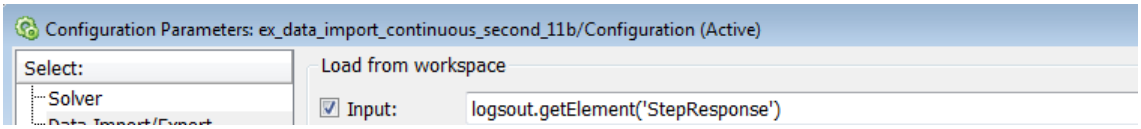
4 Open the second model, `ex_data_import_continuous_second`.



You can configure this second model to simulate using the logged data from the first model. In this example, the second model uses a root-level Inport block to load the logged data as input for the simulation. The Inport block has the **Interpolate data** option selected.

5 In the second model, select the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.

Use the `Simulink.SimulationData.Signal.getElement` method to specify the `StepResponse` signal element:



6 Specify that for the second model, the Simulink solver runs at the time steps specified in the saved data (`u`). In the Data Import/Export pane, set the **Output options** parameter to `Produce additional output` and the **Output times** parameter to:

```
logsout.getElement('StepResponse').Values.Time
```

7 Simulate the second model.

Note Simulink does not feed minor time step data through root input ports. For details about minor time steps, see “Types of Sample Time” on page 7-13.

See Also

Related Examples

- “Load Data to Test a Discrete Algorithm” on page 70-31
- “Load Data for an Input Test Case” on page 70-32

More About

- “Overview of Signal Loading Techniques” on page 70-15

Load Data to Test a Discrete Algorithm

Discrete signals are signals that you define using evenly spaced time values. One signal value is read at each time step, using the sample time of the source block.

Use a structure that has an empty time vector, which results in the model using the sample time of the source block. Using this approach avoids possible mismatches between the vector and the Simulink time steps. The double-precision rounding used by computers and the values expected by Simulink can differ.

Suppose that you want to import signal data for this simple model.

1. In the Block Parameters dialog box for the Inport block:
   - Set the sample time.
   - Clear the **Interpolate data** parameter.
2. For the data that you want to import, specify a structure variable that does *not* include a time vector. For example, for the variable called `import_var`:

   ```
   import_var.time = [];
   import_var.signals.values = [0; 1; 5; 8; 10];
   import_var.signals.dimension = 1;
   ```

   The input for the first time step is read from the first element of an input port value array. The value is `0`. The value for the second time step is read from the second element of the value array (`1`), and so on.

   For details about how to specify the signal value and dimension data, see “Loading Data Structures to Root-Level Inputs” on page 70-39.
3. Select the **Configuration Parameters > Data Import/Export > Input** parameter and specify `import_var` for the data to import.

If you are using a From Workspace block to import data, use a similar approach. In addition, set the **Form output after final data value by** parameter to a value other than `Extrapolation`.

See Also

Related Examples

- “Load Data to Model a Continuous Plant” on page 70-29
- “Load Data for an Input Test Case” on page 70-32

More About

- “Overview of Signal Loading Techniques” on page 70-15

Load Data for an Input Test Case

In this section...

For most input test cases, you try to minimize the number of time points. The signal data you load includes samples with ramps and discontinuities.

Guidelines for Importing a Test Case

Typically when importing a test case data, you want to minimize the number of time points. The test data focuses on discontinuities in the signal data.

- Create a signal that has ramps and steps. In other words, the signal has one or more discontinuities.
- Create the signal using the fewest points possible.
- Have the Simulink solver execute at the specified discontinuities.

To import this signal in Simulink, use a From Workspace, From File, or Signal Editor block, all of which support zero-crossing detection.

You can load data of these types:

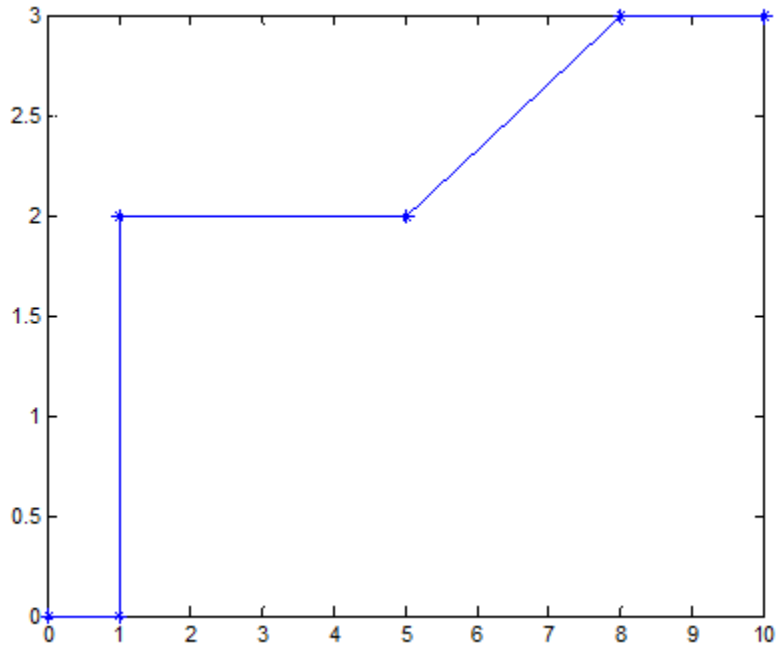
- A `Simulink.SimulationData.Dataset`
- Array
- `Simulink.SimulationData.Signal`
- Structure
- A structure array containing data for all input ports (not supported by Signal Editor block)
- Empty matrix — Use an empty matrix for ports for which you want to use ground values, without having to create data values
- Time expression (not supported by Signal Editor block)

Specify a time vector and signal values, but specify only the time steps at points where the shape of the output jumps. For details about specifying a time vector, see “Specify Time Data” on page 70-40.

Use any of the input data formats described in “Forms of Input Data” on page 70-36, except for MATLAB time expressions.

Example of Test Case Data

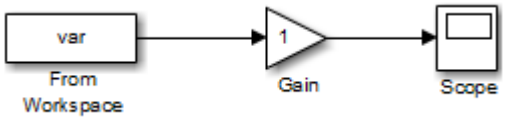
The following is an example of test case data:



The following two examples use this test case data.

Use From Workspace Block for Test Case

1. Open the model `ex_data_import_test_case_from_workspace`.



2. Enable zero-crossing detection. In the From Workspace block dialog box, select **Enable zero-crossing detection**. Zero-crossing detection allows you to capture discontinuities accurately.
3. Create a signal structure for the test case. At each discontinuity, enter a duplicate entry in the time vector, which generates a zero crossing and forces the variable-step solver to take a time step at this exact time. For details, see “Load Data Using the From Workspace Block” on page 70-65.

   Define the `var` structure representing the test case:

   ```
   var.time = [0 1 1 5 5 8 8 10];
   var.signals.values = [0 0 2 2 2 3 3 3]';
   var.signals.dimensions = 1;
   ```

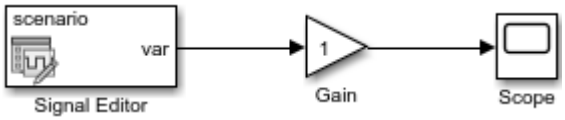
4. To import the test case structure, in the From Workspace block dialog box, in the **Data** parameter, specify `var`.
5. Simulate the model. The Scope block reflects the test case data.

Use Signal Editor Block for Test Case

Instead of using a From Workspace block, you can use a Signal Editor block to either:

- Create a signal interactively
- Import a signal from a MAT-file

1 Create a model with Signal Editor, Gain, and Scope blocks.



2 Create a structure and save it in a MAT-file:

```
scenario = Simulink.SimulationData.Dataset;
time = [0 1 1 5 5 8 8 10];
data = [0 0 2 2 2 3 3 3]';
scenario{1} = timeseries(data,time);
scenario{1}.Name = 'var';
save var.mat scenario
```

3 Open the Signal Editor dialog box by double-clicking the Signal Editor block.

4 In the File name parameter, enter `var.mat`.

5 In the **Active scenario** parameter, select `scenario`. Click **OK**.

The Scope block display reflects the test case data from the MAT-file.

See Also

Related Examples

- “Load Data to Model a Continuous Plant” on page 70-29
- “Load Data to Test a Discrete Algorithm” on page 70-31

More About

- “Overview of Signal Loading Techniques” on page 70-15

Load Data to Root-Level Input Ports

In this section...

You can load data from a workspace to a root-level inport modeled using one of these blocks:

- Inport block
- Enable block
- Trigger block that has an edge-based (rising, falling, or either) trigger type

These blocks import data from the workspace based on the value of the **Configuration Parameters > Data Import/Export > Input** parameter.

Tip To import many signals to root-level input ports, consider using the Root Inport Mapper tool. For more information, see “Map Root Inport Signal Data” on page 71-7.

You can also import data from a workspace using a From Workspace block. For details, see the From Workspace documentation and “Load Data for an Input Test Case” on page 70-32.

Specify Input Data

You can specify input data manually, using the **Input** configuration parameter. To load many signals to root-level input ports, consider using the Root Inport Mapping tool, which automatically specifies in the **Input** parameter the data you map using the tool. For details, see “Map Data Using Root Inport Mapper Tool” on page 71-2.

1. Select the **Configuration Parameters > Data Import/Export > Input** parameter.

   Note The use of the **Input** configuration parameter is independent of the setting for the **Format** configuration parameter for saving logged data.

2. Enter an external input specification in the adjacent edit box and click **Apply**. For a list of the forms of data you can specify, see “Forms of Input Data” on page 70-36.

In the **Input** box, specify the signal input using one of these approaches:

- Create data at run-time for each simulation time step using the input `u = UT(t)` for either a MATLAB function (expressed as a string) or MATLAB expression.
- Specify the data directly, using one of the input data forms described in “Forms of Input Data” on page 70-36.

Comma-Separated List

If you specify `Dataset` data, specify only one `Dataset` object for the **Input** parameter. Do not include it in a comma-separated list.

Each variable or expression must evaluate to an appropriate object that corresponds to a specific root-level input port in the model. Each variable or expression in the list must evaluate to the appropriate object that corresponds to one of the root-level input ports of the model. The first item corresponds to the first root-level input port, the second to the second root-level input port, and so on. The dimensions for each data sample must match the dimensions of the data specified in the input block parameter.

For an Enable or Trigger block, the signal driving the enable or trigger port must be the last item in the comma-separated list. If you have both an enable and a trigger port, then specify:

- The enable port as the next-to-last item in the list
- The trigger port as the last item

Use an empty matrix to specify ground values for a port. For example, to load data for input ports `in1` and `in3`, and to use ground values for port `in2`, enter the following in the **Input** parameter:

```
in1, [], in3
```

Forms of Input Data

You can provide input data with the following formats:

- `Simulink.SimulationData.Dataset` — Collection of logged data in MATLAB `timeseries` format. For more information, see “Loading Dataset Data to Root-Level Inputs” on page 70-37.
- MATLAB `timeseries` — For more information, see:
  - “Loading MATLAB Timeseries Data to Root-Level Inputs” on page 70-38
  - “Load Bus Data to Root-Level Input Ports” on page 70-46
- `Simulink.SimulationData.DatasetRef` — For more information, see “Load Big Data for Simulations” on page 70-7
- MATLAB `timetable` — For more information, see “Loading MATLAB Timetable Data to Root-Level Inputs” on page 70-39.
- Array — See “Loading Data Arrays to Root-Level Inputs” on page 70-42.
- `Simulink.SimulationData.Signal` — For more information, see “Load Data Logged In Another Simulation” on page 70-27.
- `matlab.io.datastore.SimulationDatastore` – For more information, see “Work with Big Data for Simulations” on page 72-29.
- `matlab.io.datastore.sdidatastore`
- Structure — To simplify the specification of external input data, you can load data for a subset of root-level input port blocks. This approach avoids having to create data structures for the ports for

which you want to use ground values. For information about ground values, see “Initialize Signals and Discrete States” on page 75-37. For more information about loading structure data, see “Loading Data Structures to Root-Level Inputs” on page 70-39.

- Structure array containing data for all input ports.
- Empty matrix — Use an empty matrix for ports for which you want to use ground values, without having to create data values.
- Time expression — For more information, see “Loading MATLAB Time Expressions to Root Inports” on page 70-44.

Note When you specify `timetable` data to load, the `timetable` must contain data for only one signal.

For information about importing bus data, see “Load Bus Data to Root-Level Input Ports” on page 70-46.

Time Values for the Input Parameter

Time values that you specify in the **Input** parameter do not control the times the solver uses. Solvers have their own logic for propagating time and might require input data at an arbitrary time value. The **Interpolate** parameter setting for the root-level input block (for example, the root-level Inport block) specifies how to handle output at time steps for which no corresponding workspace data exists.

The time values specified in the **Input** parameter cannot be sparse or include `NAN` or `Inf` values.

Data Loading

If you select the **Interpolate data** option for the corresponding Inport, Enable, or Trigger block, Simulink linearly interpolates or extrapolates input values as necessary.

Simulink resolves symbols used in the external input specification as described in “Symbol Resolution” on page 67-127. The `sim` command provides some data import capabilities that are available only for programmatic simulation.

If you use a `Simulink.SimulationData.Dataset` object that includes a `matlab.io.datastore.SimulationDatastore` object as an element, then the data stored in persistent storage is streamed in from a file. For more information, see “Load Big Data for Simulations” on page 70-7.

Loading Dataset Data to Root-Level Inputs

You can use a Dataset object as a value for the **Configuration Parameters > Data Import/Export > Input** parameter. Specify only one `Dataset` object and do not include it in a comma-separated list. The number of elements in the `Dataset` must match the number of root-level input ports.

Dataset Elements

A `Dataset` object can include elements with different data types.

For individual non-bus signal data, you can specify these types of data for `Dataset` elements:

- `timeseries`
- `timetable`
- `matlab.io.datastore.SimulationDatastore`
- `double` vectors or structure of `double` data
- a `Simulink.SimulationData.Signal`, `Simulink.SimulationData.State`, or `Simulink.SimulationData.DataStoreMemory` object
- An array that meets one of these requirements:
  - An array with time in the first column and the remaining columns each corresponding to an input port. See “Loading Data Arrays to Root-Level Inputs” on page 70-42.
  - An `nx1` array for a root inport that drives a function-call subsystem.
- Structure — See “Loading Data Structures to Root-Level Inputs” on page 70-39.

For bus signals, use a structure with a data element for each leaf signal, using one of these formats:

- A MATLAB `timeseries` object
- A MATLAB `timetable` object
- A `matlab.io.datastore.SimulationDatastore` object
- An empty matrix
- Another structure, with data elements for each signal that are consistent with these requirements for a structure for bus data

Note When you specify `timetable` data to load, the `timetable` can contain data for only one signal.

Create a Dataset Object for Inport Blocks

To generate a `Simulink.SimulationData.Dataset` object from the root-level Inport blocks in a model, you can use the `createInputDataset` function. Signals in the generated dataset have the properties of the Inport blocks and the corresponding ground values at model start and stop times. You can create `timeseries` and `timetable`objects for the time and values for signals for loading. The other signals use ground values. Each `timetable` object must contain data for only one signal.

You can load into a root-level input port data specified by a MATLAB `timeseries` object that resides in a workspace.

Note This documentation about importing MATLAB timeseries data includes examples of root Inport blocks. Unless specifically noted otherwise, the examples are applicable to root-level Enable, Trigger, and From Workspace blocks.

Loading MATLAB Timeseries Data to Root-Level Inputs

Time Dimension

When you create a MATLAB `timeseries` object to import data to Simulink, the time dimension (number of time samples) depends on the dimension and the type of signal data.

Signal Data Dimension or Type	Time Dimension Alignment	Example of timeseries Constructor

Scalar or a 1D vector	First	Constructor for a scalar signal. Time is aligned with the first dimension.
		`t = (0:10)';`
		`ts = timeseries(sin(t), t);`

2D (including row and column vectors) or greater	Last	Constructor for a matrix signal. Time is aligned with the last dimension.
		`t = 0;`
		`ts = timeseries([1 2; 3 4], t);`

2D row vector, and there is only one time step	Last	`'InterpretSingleRowDataAs3D', true`
		For example:
		`t = 0;`
		`ts = timeseries([1 2], t, 'InterpretSin`

Enum Data

If you specify an `enum` in `timetable` data, clear the **Interpolate data** parameter for the corresponding Inport block.

Loading MATLAB Timetable Data to Root-Level Inputs

In general, you can load MATLAB `timetable` data the same way you load MATLAB `timeseries` data. Each `timetable` must contain data for only one signal.

Loading Data Structures to Root-Level Inputs

Data Structures

You can load to a root-level input port data from the workspace in the form of a structure, whose name you specify in the **Configuration Parameters > Data Import/Export > Input** parameter. For information about defining MATLAB structures, see “Create Structure Array”.

You can specify structures for the model as a whole or on a per-port basis. For information about specifying per-port structures for the **Input** parameter, see “Structures for All Ports or for Each Port” on page 70-40.

The structure always includes a signals substructure, which contains a values field and a dimensions field. Depending on the modeling task that you want to perform, the structure can also include a time field. The form of a structure that you use depends on the type of signals for which you are importing data:

- Discrete signals (the signal is defined at evenly spaced values of time) — Use a structure that has an empty time vector. Specify a `signals` field, which contains an array of substructures, each of which corresponds to a model input port.
- Continuous signals (the signal is defined for all values of time) — The approach that you use depends on whether the data represents a smooth curve (continuous) or a curve that has

discontinuities (jumps) over its range (discrete). Specify a `signals` field, which contains an array of substructures, each of which corresponds to a model input port. You can specify a `time` field, which contains a time vector. See “Specify Time Data” on page 70-40.

For examples of importing data for discrete and continuous signals, see:

- “Load Data to Test a Discrete Algorithm” on page 70-31
- “Load Data to Model a Continuous Plant” on page 70-29
- “Load Data for an Input Test Case” on page 70-32

Structures for All Ports or for Each Port

You can specify one structure to provide input to all root-level input ports in a model, or you can specify a separate structure for each port.

The per-port structure format consists of a separate structure-with-time or structure-without-time for each port. The input data structure for each has only one `signals` field. To specify this option, enter the names of the structures in the **Input** text field as a comma-separated list, `in1, in2,..., inN`. The value `in1` is the data for first input port in the model, `in2` for the second input port, and so on.

To specify one structure for all ports:

- The `values` field must contain an array of inputs for the corresponding input port. If you specify a time vector, each input must correspond to a time value specified in the `time` field.

  If the inputs for a port are scalar or vector values, the `values` field must be an `M-by-N` array. If you specify a time vector, `M` must be the number of time points specified by the `time` field and `N` is the length of each vector value.

  If the inputs for a port are matrices (2-D arrays), the `values` field must be an `M-by-N-by-T` array. `M` and `N` are the dimensions of each matrix input and `T` is the number of time points. For example, suppose that you want to input 51 time samples of a 4-by-5 matrix signal into one of the input ports in your model. Then, the corresponding `dimensions` field of the workspace structure must equal `[4 5]` and the `values` array must have the dimensions `4-by-5-by-51`.
- The `dimensions` field specifies the dimensions of the input. If each input is a scalar or vector (1-D array) value, the `dimensions` field must be a scalar value specifying the length of the vector (1 for a scalar). If each input is a matrix (2-D array), the `dimensions` field must be a two-element vector whose:

  - First element specifies the number of rows in the matrix
  - Second element specifies the number of columns

Note Set the **Port dimensions** parameter of the Inport or the Trigger block to be the same value as the `dimensions` field of the corresponding input structure. If the values differ, you get an error message when you try to simulate the model.

Specify Time Data

You can specify a time vector of doubles as part of the data structure to import. For example, specify a time vector when importing signal data to represent a continuous plant or to create a test case. To test a discrete algorithm, use a structure with an empty time vector. This table provides additional recommendations for specifying time values, based on the kind of signal data you want to load.

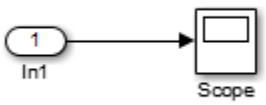
Signal Data	Time Data Recommendation
Inport or Trigger block with a discrete sample time	Do not specify a time vector. Simulink loads one signal value at each time step.

Evenly spaced discrete signals	Use an expression in this form:
	`timeVector = timeStep * [startTime:numSteps-1]'`
	The vector is transposed. Also, because the start time is a time step, you need specify the number of steps you want minus 1. For example, to specify 50 time values at 0.2 time steps:
	`T1 = 0.2 * [0:49]'`
	Note Do *not* use an expression in this form:
	`timeVector = [startTime:timeStep:endTime]'`
	For example, do not use:
	`T2 = [0:0.2:10]'`
	This time vector form is not equivalent to the form that multiplies by time steps (`T1`), because of double-precision rounding used by computers. Simulink expects exact values, with no double-precision rounding. Using the `T2` form can lead to mismatches between the provided time vector and the times steps taken by Simulink, resulting in unexpected simulation results.

Unevenly spaced values	Use any valid MATLAB array expression; for example, `[1:5 5:10]` or `(1 6 10 15)`. The From Workspace, From File, and Signal Editor blocks support zero-crossing detection. If the root-level input port is connected to one of those blocks, you can specify a zero-crossing time by using a duplicate time entry.

Examples of Specifying Signal and Time Data

In the first example, consider the following model that has a single input port:



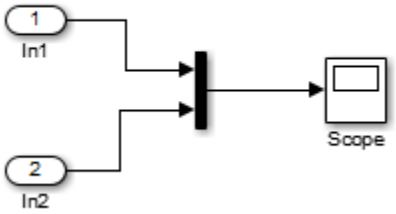
1. Create an input structure for loading 11 time samples of a two-element signal vector of type `int8` into the model:

```
N = 10
Ts = 0.1
a.time = Ts*[0:N]';
```

```
c1 = int8([0:1:10]');
c2 = int8([0:10:100]');
a.signals(1).values = [c1 c2];
a.signals(1).dimensions = 2;
```

2. In the **Configuration Parameters** > **Data Import/Export** > **Input** parameter edit box, specify the variable `a`.
3. In the Inport block dialog box, in the **Signal Attributes** tab, set **Port dimensions** to 2 and **Data type** to `int8`.

As another example, consider a model that has two inputs.



Suppose that you want to input a sine wave into the first port and a cosine wave into the second port. Define a structure, `a`, as follows, in the MATLAB workspace:

```
a.time = 0.1*[0:1]';
a.signals(1).values = sin(a.time);
a.signals(1).dimensions = 1;
a.signals(2).values = cos(a.time);
a.signals(2).dimensions = 1;
```

Enter the structure name (`a`) in the **Configuration Parameters** > **Data Import/Export** > **Input** parameter edit box.

Note In this model you do not need to specify the dimension and data type, because the default values are `1` and `double`.

Loading Data Arrays to Root-Level Inputs

You can load to a root-level input port data from the workspace in the form of a data array, which you specify in the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.

This import format consists of a real (noncomplex) matrix of data type `double`. The first column of the matrix must be a vector of times in ascending order. The remaining columns specify input values.

- Each column represents the input for a different Inport or Trigger block signal (in sequential order).
- Each row is the input value for the corresponding time point.

For a Trigger block, the signal that drives the trigger port must be the last data item.

The total number of columns of the input matrix must equal `n + 1`, where `n` is the total number of signals entering the model input ports.

Specify the Input Expression

The default input expression for a model is `[t,u]` and the default input format is `Array`. If you define `t` and `u` in the MATLAB workspace, simply select the **Configuration Parameters** > **Data Import/Export** > **Input** parameter to input data from the model workspace.

Suppose that you have a model with two Inport blocks:

- The `In1` block accepts two signals (the block has the **Port dimensions** parameter set to `2`).
- The `In2` block accepts one signal (the block uses the default value for the **Port dimensions** parameter).

You define `t` and `u` in the MATLAB workspace:

```
numSteps = 9;
timeStep = 0.1;
t = (timeStep*(0:numSteps))';
u = [sin(t),cos(t),4*cos(t)];
```

When the simulation runs, the signal data `sin(t)` and `cos(t)` are assigned to `In1` and the signal data `4*cos(t)` is assigned to `In2`. Signal data is input for 100 time points.

Note The array input format allows you to load only real (noncomplex) scalar or vector data of type `double`. Use the structure format to input complex data, matrix (2-D) data, and data types other than `double`.

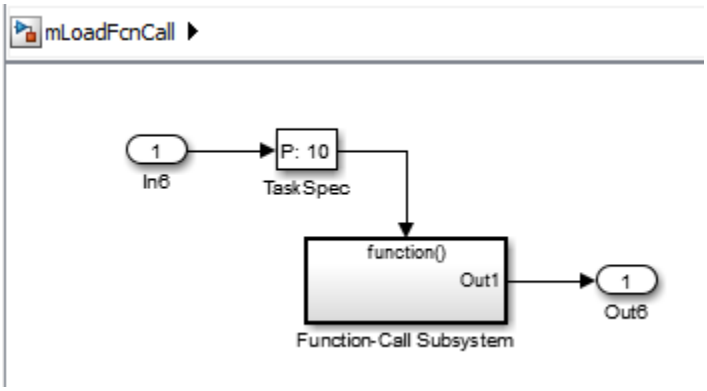
Arrays for Input Ports Driving Function-Call Subsystems

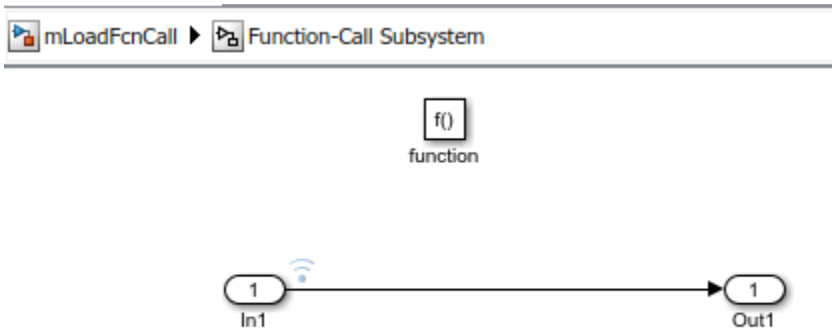
You can use an array to drive a Function-Call Subsystem through a root-level input port. You can use an array or an array that is an element of a `Dataset` object. The array must be an `nx1` array. For the root-level Inport block, select the **Output function call** parameter.

For example, this `Dataset` object has an array element `x`:

```
ds = Simulink.SimulationData.Dataset;
x = [1 3 7 8]';
ds = ds.addElement(x,'theElementName');
```

This model uses the `ds` data set in the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.





When you simulate the model, the time values of the logged signal data in the Function-Call Subsystem show that the Function-Call Subsystem was triggered only for the times specified in an array stored in `ds`.

```
>> logsout{1}.Values.Time

ans =

     1
     3
     7
     8
```

Loading MATLAB Time Expressions to Root Inports

Specify the Input Expression

You can use a MATLAB time expression to load data from a workspace into a root-level input port. To use a time expression, enter the expression as a string (enclosed in single quotes) in the **Input** field of the **Data Import/Export** pane. The time expression can be any MATLAB expression that evaluates to a row vector equal in length to the number of signals entering the input ports of the model. Suppose that a model has one vector Inport that accepts two signals. Also, suppose that `timefcn` is a user-defined function that returns a row vector two elements long. Here are valid input time expressions for such a model:

```
'[3*sin(t), cos(2*t)]'

'4*timefcn(w*t)+7'
```

The expression is evaluated at each step of the simulation, applying the resulting values to the input ports of the model. Simulink defines the variable `t` when it runs the simulation. Also, you can omit the time variable in expressions for functions of one variable. For example, the expression `sin` is interpreted as `sin(t)`.

See Also

Blocks
Enable | Inport | Trigger

Classes
`Simulink.SimulationData.Dataset` | `addElement` | `timeseries`

Related Examples

- “Load Bus Data to Root-Level Input Ports” on page 70-46
- “Loading MATLAB Timeseries Data to Root-Level Inputs” on page 70-38
- “Loading Data Arrays to Root-Level Inputs” on page 70-42
- “Loading MATLAB Time Expressions to Root Inports” on page 70-44
- “Loading Data Structures to Root-Level Inputs” on page 70-39
- “Comparison of Signal Loading Techniques” on page 70-21

More About

- “Loading Data Arrays to Root-Level Inputs” on page 70-42

Load Bus Data to Root-Level Input Ports

In this section...

You can import bus data to top-level input ports by manually specifying the data in the **Input** configuration parameter or by using the Root Inport Mapper tool. For information about importing bus data using the Root Inport Mapper tool, see “Import Bus Data” on page 71-15.

Imported Bus Data Requirements

You can import bus (virtual, nonvirtual, or array of buses) data to a top-level input port defined by a bus object (see `Simulink.Bus`). In the top-level Inport block, set **Data type** to `Bus` and specify the name of a bus object. To specify data values for bus signals, use a structure of:

- MATLAB `timeseries` objects
- MATLAB `timetable` objects
- A combination of `timeseries` and `timetable` objects

Bus elements for which you do not include a field in the structure use ground values. You can use an empty matrix to specify to use ground values.

Note When you specify `timetable` data to load, the `timetable` must contain data for only one signal.

The structure of `timeseries` or `timetable` (or both) objects must match the bus elements in terms of:

- Hierarchy
- Name of the structure field, which must match the bus element name. (The `name` property of the `timeseries` object does not need to match the bus element name.)
- Data type
- Dimensions
- Complexity

The order of the structure fields does not have to match the order of the bus elements.

You can include the structure as an element of a `Dataset` object. You can use a structure in a comma-separated list. You can specify an empty matrix in a comma-separated list. The empty matrix uses the ground values for the bus signal.

For example, to load data for input ports `in1` and `in3`, and to use ground values for port `in2`, enter the following in the **Input** parameter:

```
in1, [], in3
```

Initialize Bus Signals

You can initialize bus signals, including using partial specification of initialization data. For details, see “Specify Initial Conditions for Bus Signals” on page 76-57.

For details about importing array of bus data to a root Inport block, see “Import Array of Buses Data” on page 70-49.

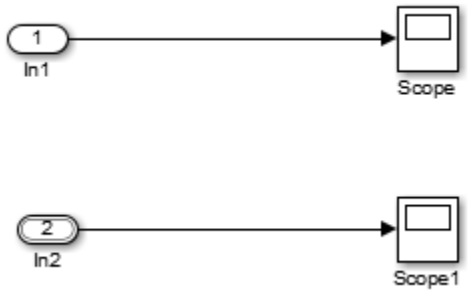
Limitations for Importing Bus Data to Top-Level Inputs

The signal data that you can use the Root Inport Mapper tool to import and map to a top-level Inport block can include bus data. You cannot use that tool to map bus signals to a top-level Enable or Trigger block.

You cannot use input ports to import buses in external modes. To import bus data in rapid accelerator mode, use `Dataset` format.

Import Bus Data to a Top-Level Inport

This model has two Inport blocks connected to Scope blocks. The data type of the `In1` block is inherited (nonbus data) and the data type of the `In2` block is defined by the bus object `BusObject`. The model has a callback that loads `BusObject` and its sub-bus `BusObject1`.



The `BusObject` bus object has two elements:

- `c`
- `s1`, which is a nested bus that has two elements:
  - `a`
  - `b`

1. Open the .
2. Create a MATLAB `timeseries` object for `In1`, for which you want to import nonbus data.

   For example:

   ```
   t1 = (1:10)';
   d1 = sin(t1);
   in1 = timeseries(d1,t1);
   ```

3 Create an input structure, which can consist of MATLAB `timeseries` objects or MATLAB `timetable` objects, or a combination of those types of objects. Create one `timeseries` or `timetable` object for each leaf bus element for which you do not want to use ground values. This example uses ground values for the `b` bus element, so it does not need a `timeseries` or `timetable` object for that element.

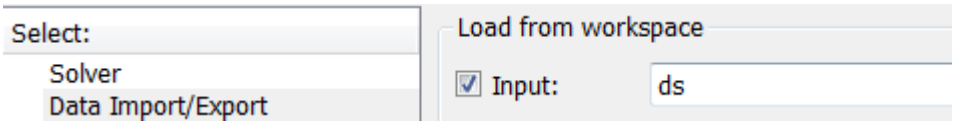
```
t2 = (1:5)';
d2 = cos(t2);
in2.c = timeseries(d1,t1);
in2.s1.a = timetable(seconds(t2),d2);
```

The MATLAB `timeseries` objects that you create must match the corresponding bus elements, as described in “Imported Bus Data Requirements” on page 70-46.

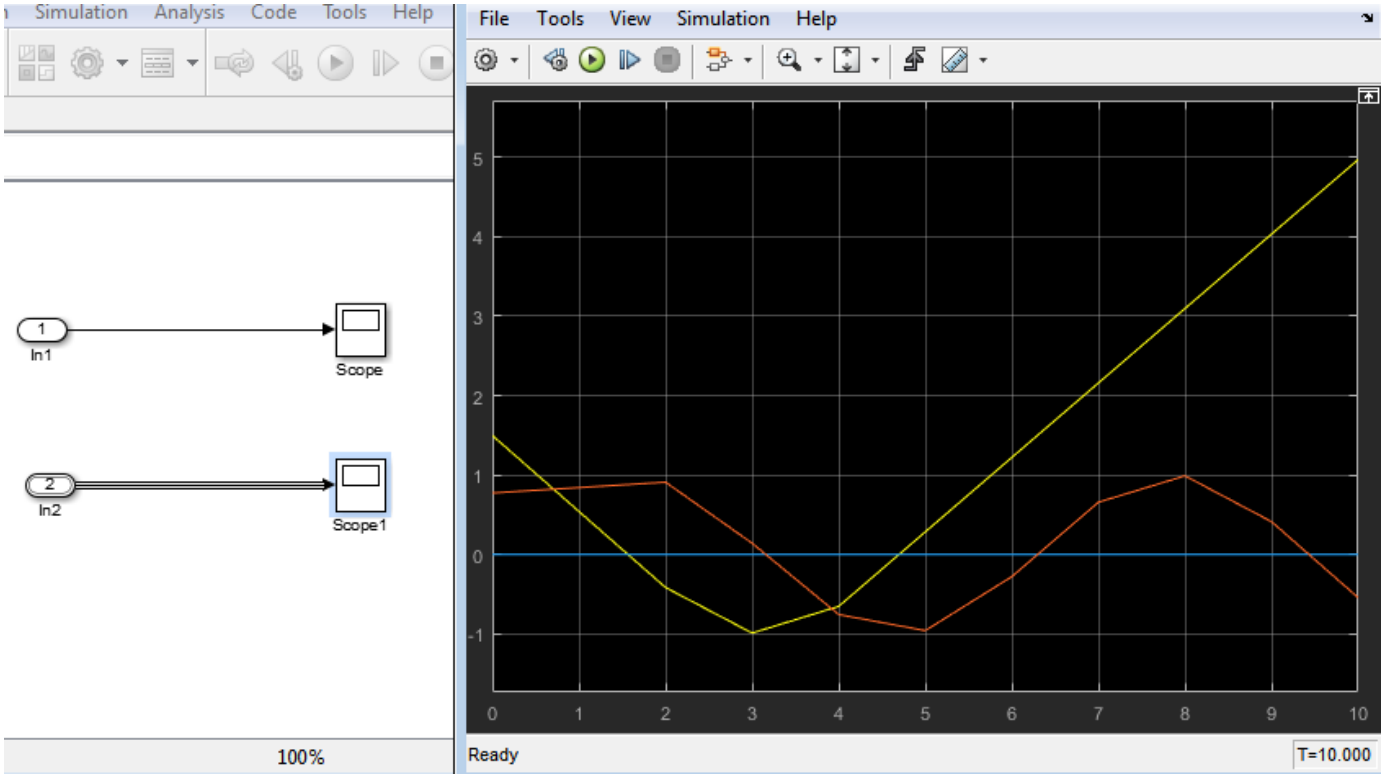
4 Create a `Dataset` object and add `in1` and `in2` to the data set.

```
ds = Simulink.SimulationData.Dataset;
ds = ds.addElement(in1,'in1_signal');
ds = ds.addElement(in2,'in2_signal');
```

5 In the **Configuration Parameters** > **Data Import/Export** > **Input** parameter edit box, enter the `Dataset` object `ds`.



6 Simulate the model. The Scope block connected to In2 shows the imported bus data.



Get Information About Bus Objects

To determine the number of MATLAB `timeseries` objects and data type, complexity, and dimensions needed for creating a structure of `timeseries` objects from a bus, use these methods:

- `Simulink.Bus.getNumLeafBusElements`
- `Simulink.Bus.getLeafBusElements`

For example, for the bus object `BusObject`:

```
num_el = BusObject.getNumLeafBusElements
```

```
num_el =

     3
```

```
el_list = BusObject.getLeafBusElements
```

```
el_list =

  3x1 BusElement array with properties:

    Min
    Max
    DimensionsMode
    SampleTime
    Description
    Units
    Name
    DataType
    Complexity
    Dimensions
```

```
el_list(1).Dimensions
```

```
ans =

     1
```

Create Structures of Timeseries Objects from Buses

If you have `timeseries` objects defined, you can use them to create a structure of `timeseries` objects based on a bus object. Use the `Simulink.SimulationData.createStructOfTimeseries` function. For example, if you have defined `timeseries` objects `ts1`, `ts2`, and `ts3`, and you have a bus object `MyBusObject`, you can use this command to create a structure of `timeseries` objects:

```
input = Simulink.SimulationData.createStructOfTimeseries(...
'MyBusObject',{ts1,ts2,ts3});
```

The number of `timeseries` objects in the cell array must match the number of leaf elements in the bus object. The data type, dimensions, and complexity of each `timeseries` object must match those attributes of the corresponding bus object leaf node.

Import Array of Buses Data

To import (load) array of buses data using a root Inport block, use an array of structures of MATLAB `timeseries` objects.

Note You cannot use an Enable, Trigger, From Workspace, or From File block to import data for an array of buses.

Full Specification of Data

You can use logged data for an array of buses signal from a previous simulation as roundtrip input to a root-level Inport in a subsequent simulation run. The logged data is a full specification of data for the Inport block.

If you construct an array of structures of MATLAB `timeseries` objects to specify fully the data to import:

- Specify the structure fields in the same order as the signals in the bus signals.
- Do not include more fields in the structure than there are signals in the bus.

  For leaf fields, match exactly the data type, dimensions, and complexity of the corresponding signal in the bus.

Partial Specification of Data

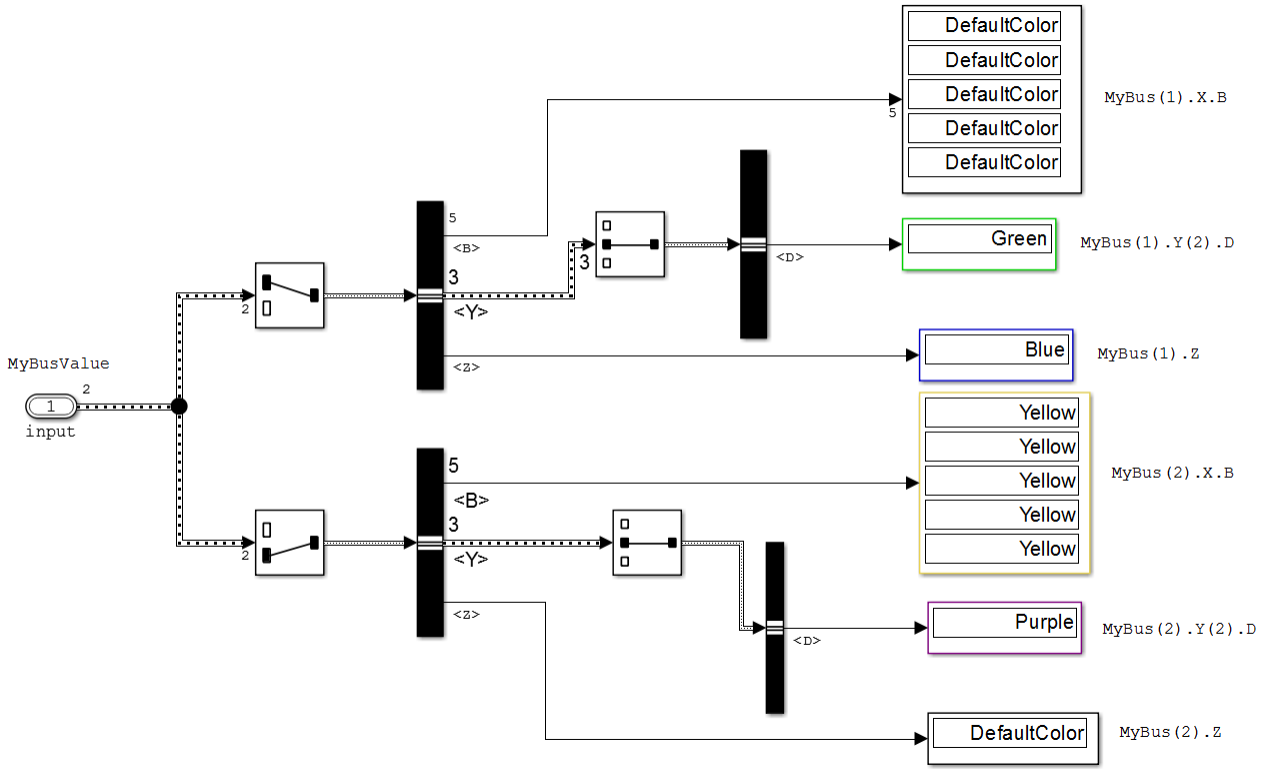
To specify partial data for array of buses, create a MATLAB array of structures with MATLAB `timeseries` objects at the leaf nodes.

The structure that you create to specify partial data must be consistent with these rules:

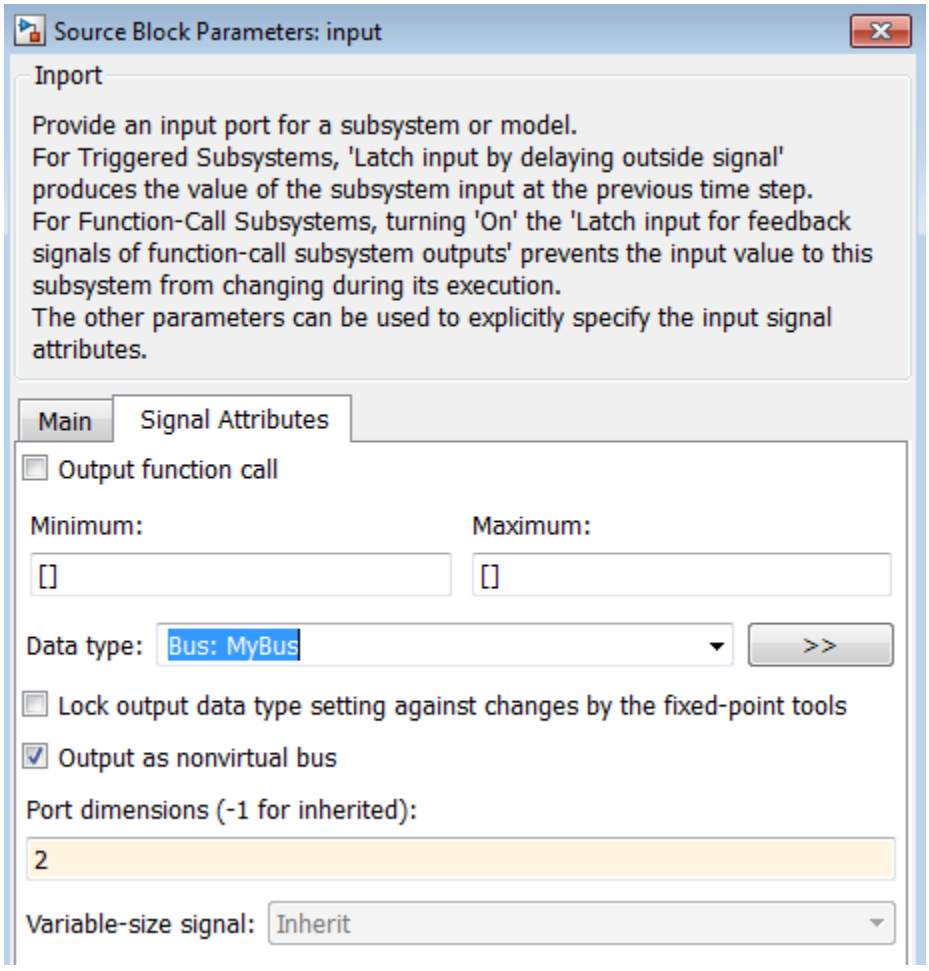
- You can omit fields, including leaf nodes and subbranches. You can also omit dimensions. If you do not specify a field, Simulink uses the ground value for that field.
- For nested bus nodes, make the dimension of each field equal to, or smaller than, the dimension for the corresponding node of the array of buses.

This example shows how you can specify partial data to be imported using a root Inport block whose data type is defined as bus object `MyBus`. You can open the model (`ex_partial_loading_aob_model`) and the MATLAB code that defines the data to import (ex_partial_loading_aob_data.m).

When you simulate `ex_partial_loading_aob_model`, you see:

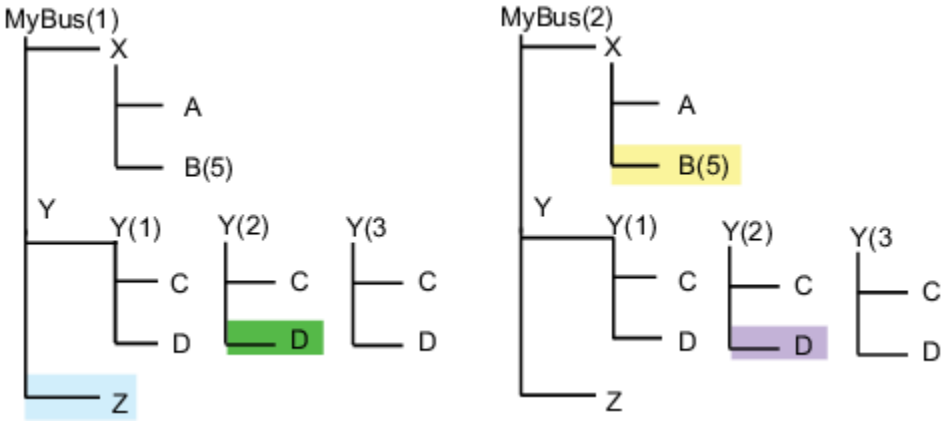


The `input` Inport block uses the `MyBus` bus object as its data type.



The `MyBus` array of buses includes `MyBus(1)` and `MyBus(2)`. The port dimension is set to 2 to reflect the two buses in the array of buses, and **Output as nonvirtual bus** is enabled.

Here are the elements of the array of buses, which includes `MyBus(1)` and `MyBus(2)`. The color highlighting shows the nodes of the array of buses for which data is being imported.



Here is MATLAB code that defines the data to import. The color that highlights the code matches the color of the corresponding node in the array of buses. To view the code used in this model, open the MATLAB code file ex_partial_loading_aob_data.m.

```
%% Create timeseries data for leaf signals
N = 10; Ts = 1;
Time = ((0:N)*Ts)';
[m, n] = size(Time);

ZData = repmat(ex_partial_loading_aob_basicColors.Blue,m,n);
BData = repmat(ex_partial_loading_aob_basicColors.Yellow,m,5);
D1Data = repmat(ex_partial_loading_aob_basicColors.Green,m,n);
D2Data = repmat(ex_partial_loading_aob_basicColors.Purple,m,n);

% Create individual timeseries objects to represent Z, B, D1 and D2
ZT = timeseries(ZData, Time, 'Name', 'Z');
BT = timeseries(BData, Time, 'Name', 'B');
D1T = timeseries(D1Data, Time, 'Name', 'D1');
D2T = timeseries(D2Data, Time, 'Name', 'D2');

%% Construct MyBusValue(1)

% Y(2).D
Y(2) = struct('D',D1T);

% Set X.B to empty to allow the second element to support specification for
% this field
X = struct('B',[]);

% MyBusValue(1)
MyBusValue(1) = struct('Z',ZT,'Y',Y,'X',X);
clear X Y;

%% Construct MyBusValue(2)

% Specify timeseries for MyBusValue(2).X.B
MyBusValue(2).X.B = BT;

% Specify timeseries for MyBusValue(2).Y(2).D
MyBusValue(2).Y(2).D = D2T;
```

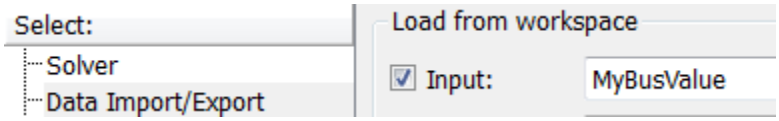
In the code that defines the import data:

- The `timeseries` object `MyBusValue` specifies the data for the highlighted nodes.
- The `timeseries` object `BT` for `MyBus(2)`, because `BT` is a leaf node, it must match exactly the dimensions, data type, and complexity of the corresponding bus element.

- The structure specifies data for `Y(2)`. You can skip the first and last nested buses of `Y` (that is, `Y(1)` and `Y(3)`).

This example specifies data for `Y(2)`; you can skip the first and last nested buses of `Y` (that is, `Y(1)` and `Y(3)`).

After you define the `MyBusValue` variable for the import data, set the **Configuration Parameters** > **Data Import/Export** > **Input** parameter to `MyBusValue`.



See Also

`Simulink.Bus`

Related Examples

- “Load Data to Root-Level Input Ports” on page 70-35
- “Import Bus Data” on page 71-15
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Import Array of Buses Data” on page 70-49
- “Nonvirtual Buses at Model Interfaces” on page 76-55

More About

- “Imported Bus Data Requirements” on page 70-46
- “Load Data to Root-Level Input Ports” on page 70-35
- “Map Root Inport Signal Data” on page 71-7
- “Virtual Bus” on page 76-2
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44

Load Input Data for a Bus Using In Bus Element Blocks

You can use In Bus Element blocks to load external input data for a bus. Using In Bus Element blocks allows flexibility in the design and implementation of external interfaces for buses. You can use an In Bus Element block to load data for an element of a bus or to load data for an entire bus. You can also use multiple In Bus Element blocks to select the same bus element.

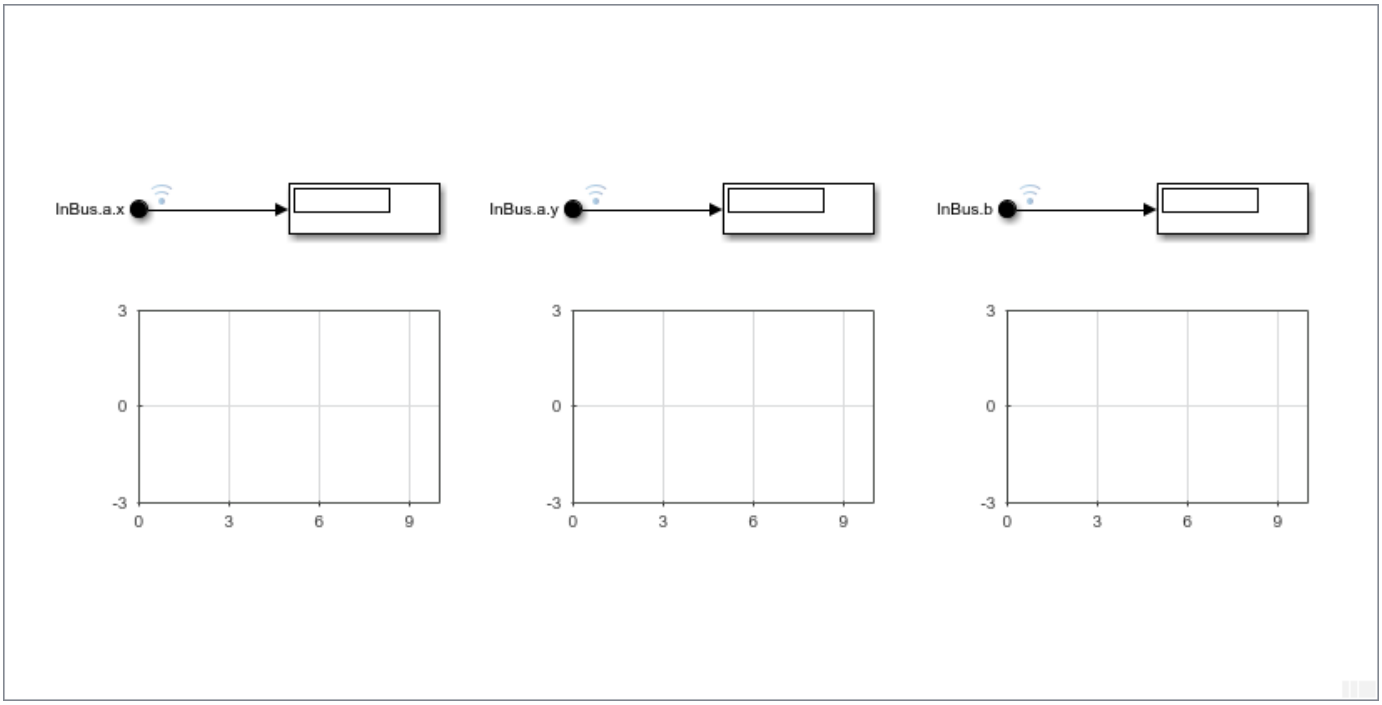
This example shows how to use In Bus Element blocks to load input data for bus elements. To load data for an entire bus using the In Bus Element block, you must specify the data type for the In Bus Element block using a `Simulink.Bus` object.

Open and Inspect the Model

Open the `ex_load_inbuselement` model.

```
open_system('ex_load_inbuselement.slx')
```

One or more In Bus Element blocks can define a port in a model. The `ex_load_inbuselement` model has one port, `InBus`, defined by three In Bus Element blocks. The label for each In Bus Element block indicates the bus element the block selects. The `InBus` port corresponds to a bus containing a nested bus, `a`, with signals `x` and `y`, and an individual signal, `b`.



Use the **Input** parameter on the **Data Import/Export** pane to specify workspace data to load as simulation input for the port in the model. The **Input** parameter for the `ex_load_inbuselement` model loads the variable `struct1`, which maps to the port according to the **Port Number** defined in the In Bus Element block dialog box.

Create Input Data

Ports that load bus data accept structures composed of individual `timeseries`, `timetable`, and `matlab.io.datastore.SimulationDatastore` objects. The hierarchy of the structure must

match the hierarchy of the bus. Ports that use In Bus Element blocks to select bus elements allow partial specification and overspecification of data.

This example creates the structure using three `timeseries` signals that correspond to a sine wave, a line, and a constant.

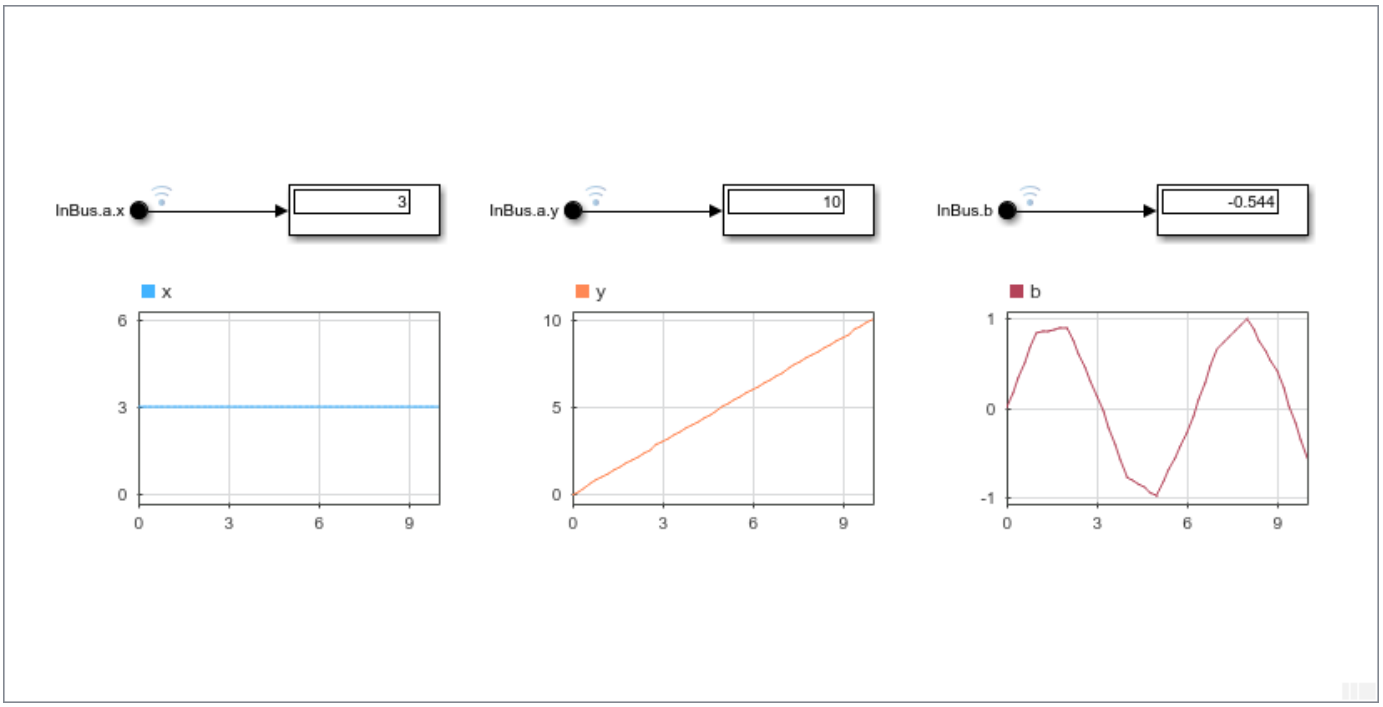
```
time = linspace(0,10,11);
sineData = sin(time);
constData = 3*ones(11,1);
lineData = linspace(0,10,11);

tsSine = timeseries(sineData,time);
tsConst = timeseries(constData,time);
tsLine = timeseries(lineData,time);
```

Construct the structure, `struct1`, to provide the input data for the port in the model. The structure field names must match the bus element names.

```
struct1.a.x = tsConst;
struct1.a.y = tsLine;
struct1.b = tsSine;
```

Fully Specify Input Data

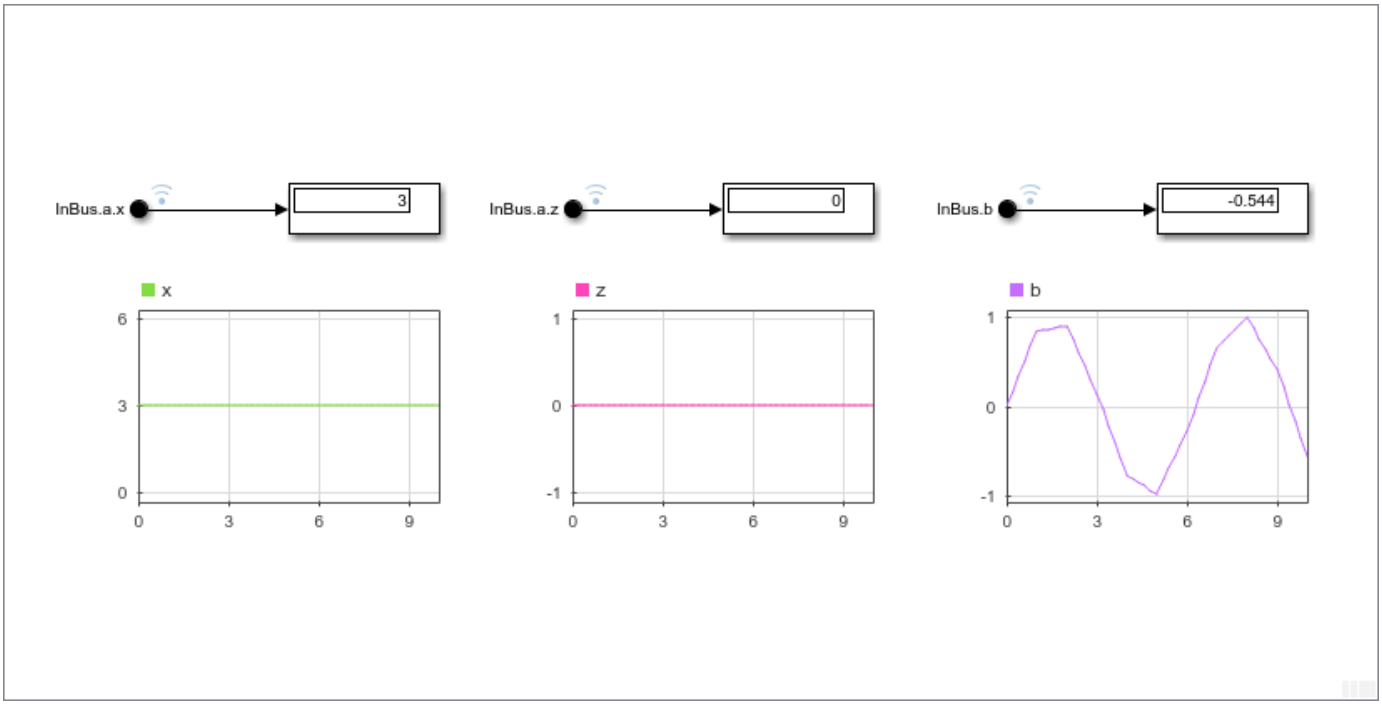
The data created in the previous section fully specifies data for the signals selected by the In Bus Element blocks in the model. Simulate the model and observe the signals on the Dashboard Scope blocks.



Partially Specify Input Data

When you use In Bus Element blocks to select bus elements, you can partially specify data using a structure that does not include one or more selected elements. Elements without data defined in the structure use ground signal values for simulation.

For example, edit the label for the In Bus Element block that selects element `InBus.a.y` so that the In Bus Element block selects `InBus.a.z`. The structure that maps to the `InBus` port does not contain a field for `z` in the nested structure, `a`. When you simulate the model, without modifying the structure, the Dashboard Scope shows ground for `InBus.a.z`.



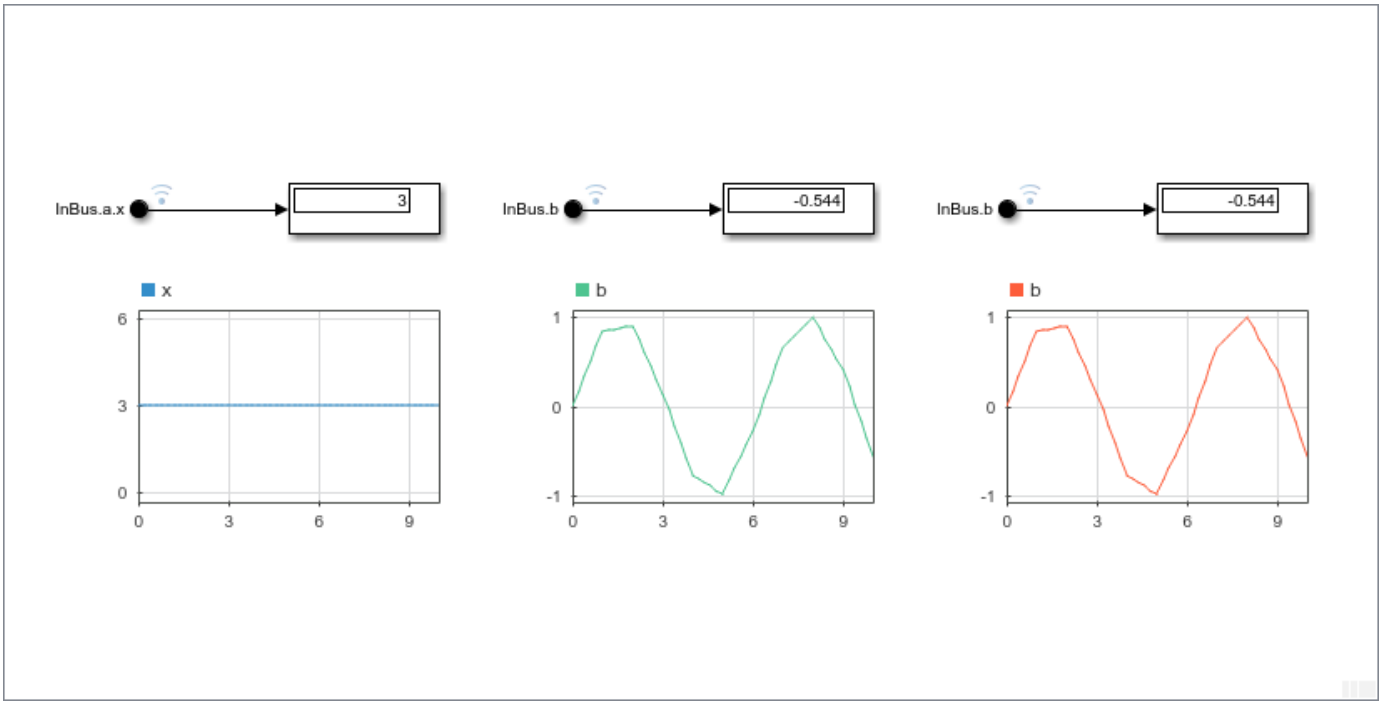
Change the In Bus Element port block that selects `InBus.a.z` back to select `InBus.a.y`.

Overspecify Input Data

When you use In Bus Element blocks to select bus elements, you can overspecify data for the port. Overspecified input data contains signals that are not selected by any of the In Bus Element blocks in the model.

For example, change the label for the In Bus Element block that selects `InBus.a.y` to select `InBus.b`. Now, none of the In Bus Element blocks selects `InBus.a.y`, while the structure still contains the data for that element.

Simulate the model and observe the signals on the Dashboard Scope blocks.



See Also

Blocks
In Bus Element | Inport

Objects
`Simulink.Bus` | `matlab.io.datastore.SimulationDatastore` | `timeseries` | `timetable`

Functions
`Simulink.SimulationData.createStructOfTimeseries`

More About

- “Simplify Subsystem and Model Interfaces with Buses” on page 76-24
- “Load Bus Data to Root-Level Input Ports” on page 70-46

Load Signal Data That Uses Units

Signal data logged in a previous simulation using signal logging or the To File or To Workspace block can include units information for `Dataset` or `Timeseries` logging formats.

Logging top-level model Outport block data from a previous simulation contains units information if:

- These **Data Import/Export** configuration parameter settings:
  - **Output** is enabled.
  - **Format** is `Dataset`.
- For the Outport blocks that you log, in the Block Parameters dialog box, you set the **Unit** parameter.

Otherwise, to include units in signal data that you load, for the `Units` property of the MATLAB `timeseries` objects that you want to load, specify a `Simulink.SimulationData.Unit` object.

Loading Bus Signals That Have Units

When you input a bus signal to a root-level Inport or Outport block, or you use a From File or From Workspace block, the output data type of the block must be a bus object. When you load the data from these blocks, the units in loaded data must match the units specified for the bus elements in the bus object. If the units for the loaded data do not match the units for a bus element in the bus object, Simulink uses the units specified in the bus object.

See Also

Classes
`Simulink.BusElement` | `Simulink.SimulationData.Unit`

Related Examples

- “Log Signal Data That Uses Units” on page 72-24

More About

- “Units in Simulink”
- “Unit Consistency Checking and Propagation” on page 9-9

Load Data Using the From File Block

In this section...

To load signal data into a model using a From File block:

1. Create a MAT-file with the signal data that you want to load. See “Create Data for a From File Block” on page 70-62.
2. Add a From File block to a model. Connect the From File block to the block that the From File provides input to.
3. Double-click the From File block and specify:
   - The path to the file that you want to load data from
   - The data format for the From File block output
   - How the data is loaded, including sample time, how data for missing data points is handled, and whether to use zero-crossing detection

Data Loading

For a Version 7.0 and earlier MAT-file, the From File block loads the complete, uncompressed data from the file into memory at the start of simulation. For a Version 7.3 MAT-file, the From File block incrementally loads data from the file during simulation.

For each simulation time hit for which the MAT-file contains no matching timestamp, Simulink uses interpolation or extrapolation to obtain the needed data. You specify the interpolation and extrapolation methods.

During simulation, the From File block cannot load data from a MAT-file that a To File block is exporting data to.

Sample Time

The From File block **Sample time** parameter specifies the sample time to load data from a MAT-file. The timestamps in the file must be monotonically nondecreasing. For details, see the From File block documentation.

Simulation Time Hits Without Corresponding Time Data

If a simulation time hit does not have a corresponding MAT-file timestamp, then the From File block output depends on:

- Whether the simulation time hit occurs before the first timestamp, within the range of timestamps, or after the last timestamp
- The interpolation or extrapolation methods that you select
- The data type of the MAT-file data

For details about interpolation and extrapolation options, see the documentation for these From File block parameters:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

Duplicate Timestamps

Sometimes the MAT-file includes duplicate timestamps (two or more data values that have the same timestamp). In such cases, the From File block action depends on when the simulation time hit occurs, relative to the duplicate timestamps in the MAT-file.

Suppose that the MAT-file contains the following data, with three data values having a timestamp value of 2:

```
time stamps:    0 1 2 2 2 3 4
data values:    2 3 6 4 9 1 5
```

The following table describes the From File block output.

Simulation Time, Relative to Duplicate Timestamp Values in MAT-File	From File Block Action
Before the duplicate timestamps	Uses the first of the duplicate timestamp values as the basis for interpolation. (In this example, that timestamp value is 6.)
At or after the duplicate timestamps	Uses the last of the duplicate timestamp values as the basis for interpolation. (In this example, that timestamp value is 9.)

Detect Zero Crossings

Zero-crossing detection locates a discontinuity in timestamps, without resorting to excessively small time steps. By default, the From File block does not enable zero-crossing detection.

For the From File block, zero-crossing detection occurs only at timestamps in the file. Simulink examines only the timestamps, not the data values.

For bus signals, Simulink detects zero-crossings across all leaf bus elements.

For more information, see the From File block documentation of the **Enable zero-crossing detection** parameter.

Create Data for a From File Block

NaN Values Not Supported

Do not include NaN values in a MAT-file that you load into a From File block.

Data Saved by a To File Block

The From File block loads data that was written by a To File block without any modifications to the data or any other special provisions.

Supported MAT-File Versions

The supported MAT-file versions are:

- Version 7.0 or earlier
- Version 7.3

For a Version 7.0 and earlier MAT-file, the From File block loads the complete, uncompressed data from the file into memory when the simulation begins. For a Version 7.3 MAT-file, the From File block incrementally loads data from the file during simulation.

For more information about MAT-files, see “MAT-File Versions”.

Convert Version 7.0 and Earlier Version MAT-Files

If you have a Version 7.0 or earlier version MAT-file that you want to use with the From File block, consider converting the file to Version 7.3. Use a Version 7.3 MAT-file if you want the From File block to load data incrementally during simulation or you want to use MATLAB `timeseries` data. For example, to convert a Version 7.0 file named `my_data_file.mat` that contains the variable `var`, at the MATLAB command prompt, enter:

```
load('my_data_file.mat')
save('my_data_file.mat', 'var', '-v7.3')
```

Storage Formats

When the From File block loads data from a MAT-file, that data must be stored in array format or as a MATLAB `timeseries` object.

Array Data

You can use the array format only for vector, double, noncomplex signal data.

For a Version 7.0 MAT-file, the From File block loads array data, but not MATLAB `timeseries` data.

The array format for stored data is a matrix containing two or more rows. The matrix in the MAT-file must have the following form:

$$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & & & \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

The first element of each column contains a timestamp. The remainder of each column contains data for the corresponding output values. Each element must be a double. Elements cannot include a `NaN`, `Inf`, or `-Inf`.

For data stored using the array format, the width of the From File output depends on the number of rows in the matrix. For a matrix containing *m* rows, the block outputs a vector of length *m*–1.

MATLAB Timeseries Data

To use bus data with a From File block, use the MATLAB `timeseries` format.

MATLAB `timeseries` format data can have:

- Any dimensionality and complexity
- Any built-in data type, including `Boolean`
- A fixed-point data type with a word length of up to 32 bits
- An enumerated data type

When you load `timeseries` data using the From File block, the data type of the time data must be `double`.

The MATLAB `timeseries` format supports the following simulation and code generation modes:

- Normal
- Accelerator
- Rapid accelerator
- Model reference accelerator

See the From File block documentation for an example of creating a MAT-file with MATLAB `timeseries` data load with a From Workspace block.

Timestamps

The timestamps in the file must be monotonically nondecreasing.

Bus Data

The From File block supports loading nonvirtual bus signals.

The data must be in a MATLAB structure that matches the bus hierarchy. Each leaf of the structure must be a MATLAB `timeseries` object.

The structure can underspecify the bus signal, but must not overspecify the bus signal. The structure cannot have any elements that do not have corresponding signals in the bus.

The structure does not require a `timeseries` object for every element in the bus hierarchy. However, the structure must have a `timeseries` object for at least one of the signals in the bus. For signals that do not specify data, the From File block outputs the ground values.

MAT-File Variable

If a MAT-file contains only one variable, then the From File block uses that variable. If the MAT-file contains more than one variable:

- For Version 7.3 MAT-files, the From File block uses the variable that is first alphabetically.
- For Version 7.0 or earlier MAT-files, the From File block uses the first variable. However, for these versions, the ordering algorithm for variables is complicated. Use a MAT-file that contains only the variable with the data that you want the From File block to load.

See Also

Blocks
From File | To File

Related Examples

- “Comparison of Signal Loading Techniques” on page 70-21

Load Data Using the From Workspace Block

In this section...

To load signal data with a From Workspace block:

1. Create a workspace variable with the signal data that you want to load.
2. Add a From Workspace block to a model. Connect the From Workspace block to the block that the From Workspace block provides input to.
3. Double-click the From Workspace block and configure:

   - The workspace data to load
   - The data format for the From Workspace block output
   - How the data is loaded, including sample time, how data for missing data points are handled, and whether to use zero-crossing detection

   Suppose that the workspace contains a column vector of times named `T` and a column vector of corresponding signal values named `U`. Entering the expression `[T U]` for **Data** parameter yields the required input array. If the required array or structure exists in the workspace, enter the name of the structure or matrix in the **Data** parameter.

An alternative to using a From Workspace block for loading workspace data is to load data to a root-level input port. For more information, see “Root-Level Input Ports” on page 70-16.

Specify the Workspace Data

Double-click the From Workspace block, and 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 `timeseries` or `timetable` object

  Real signals of type `double` can be in any format that the From Workspace block supports. For complex signals and real signals of a data type other than `double`, use any format except `Array`.

- A structure of `timeseries` or `timetable` objects

  For bus data, use a structure of `timeseries` or `timetable` objects. Match the bus hierarchy and specify a `timeseries` or `timetable` object for each leaf signal in the bus. Set up the data the

same way as you do for loading bus signals to a root-level Inport block. For details, see “Load Bus Data to Root-Level Input Ports” on page 70-46.

- A structure, with or without time

  For details, see “Specify Structure Data for the From Workspace Block” on page 70-66.

- A two-dimensional matrix

  You can use a matrix to specify only one-dimensional signals. The first element of each matrix row is a timestamp. The rest of each row is a scalar or vector of signal values.

Note When you specify `timetable` data to load, each `timetable` object can contain data for only one signal.

Specify Structure Data for the From Workspace Block

You can use a structure for one-dimensional or multidimensional signals, with or without time values. For the structure, use this format:

- A `signals.values` field, which contains a column vector of signal values.
- An optional `signals.dimensions` array, which contains the dimensions of the signal.
- An optional `time` vector of doubles, which is a column vector of timestamps.

  The nth `time` element is the timestamp of the nth `signals.values` element.

The form of a structure that you use depends on whether you are importing data for:

- Discrete signals (the signal is defined at evenly spaced values of time) — Use a structure that has an empty time vector.
- Continuous signals (the signal is defined for all values of time) — The approach that you use depends on whether the data represents a smooth curve or a curve that has discontinuities (jumps) over its range.

For examples, see:

- “Load Data to Test a Discrete Algorithm” on page 70-31
- “Use From Workspace Block for Test Case” on page 70-33

For both discrete and continuous signals, specify a `signals` field, which contains an array of substructures, each of which corresponds to a model input port.

Each `signals` substructure must contain two fields: `values` and `dimensions`.

- The `values` field must contain an array of inputs for the corresponding input port. If you specify a time vector, each input must correspond to a time value specified in the `time` field.

  If the inputs for a port are scalar or vector values, the `values` field must be an `M-by-N` array. If you specify a time vector, `M` must be the number of time points specified by the `time` field and `N` is the length of each vector value.

  If the inputs for a port are matrices (2-D arrays), the `values` field must be an `M-by-N-by-T` array. `M` and `N` are the dimensions of each matrix input and `T` is the number of time points.

Suppose that you want to input 51 time samples of a 4-by-5 matrix signal into one of your model input ports. Then, the corresponding `dimensions` field of the workspace structure must equal `[4 5]` and the `values` array must have the dimensions `4-by-5-by-51`.

- The `dimensions` field specifies the dimensions of the input. If each input is a scalar or vector (1-D array) value, the `dimensions` field must be a scalar value that specifies the length of the vector (1 for a scalar). If each input is a matrix (2-D array), the `dimensions` field must be a two-element vector whose first element specifies the number of rows in the matrix and whose second element specifies the number of columns.

For continuous signals, you can specify a `time` field, which contains a time vector. How you specify the time values depends on the kind of signal data that you want.

For information about defining MATLAB structures, see “Create Structure Array”.

Signal Data	Time Data Recommendation

Evenly spaced discrete signals	Use an expression in this form:
	`timeVector = timeStep * [startTime:numSteps-1]'`
	The vector is transposed. Also, because the start time is a time step, you need specify the number of steps you want minus 1. For example, to specify 50 time values at 0.2 time steps:
	`T1 = 0.2 * [0:49]'`
	Note Do *not* use an expression in this form:
	`timeVector = [startTime:timeStep:endTime]'`
	For example, do not use:
	`T2 = [0:0.2:10]'`
	This time vector form is not equivalent to the form that multiplies by time steps (`T1`), because of double-precision rounding used by computers. Simulink expects exact values, with no double-precision rounding. Using the `T2` form can lead to unexpected simulation results.

Unevenly spaced values	Use any valid MATLAB array expression; for example, `[1:5 5:10]` or `(1 6 10 15)`. The From Workspace, From File, and Signal Editor blocks support zero-crossing detection. If the root-level input port is connected to one of those blocks, you can specify a zero-crossing time by using a duplicate time entry.

If you load a structure that does not specify a time vector:

1. Set **Sample time (-1 for inherited)** to a value other than `0` (continuous).
2. Clear **Interpolate data**.
3. Set **Form output after final data value by** to a value other than `Extrapolation`.

Use Data from a To File Block

You can use the From Workspace block to load data exported by a To Workspace block in a previous simulation for use in a later simulation. Save the To Workspace block data in either `Timeseries` or `Structure with Time` format. Loading data that was exported to a file by a To File block using MATLAB `timeseries` does not require that you change the data.

If you set the To File block **Save format** parameter to `Array`, transpose the exported array data. The data saved by the To File block contains columns with consecutive timestamps, followed by the corresponding data. The transposed data contains rows with consecutive timestamps, followed by the corresponding data. To provide the required format, use MATLAB `load` and `transpose` commands with the MAT-file. To avoid transposing the data again, resave the transposed data.

Load Dataset Data

To use workspace data that is in the `Simulink.SimulationData.Dataset` format, extract a `timeseries` or `timetable` object from the `Dataset` object. For example, if you use signal logging with the `Dataset` format and use the default output variable `logsout`, for a single logged signal enter:

```
logsout.get(1).values
```

Specifying Variable-Size Signals

You can use a To Workspace block (with the `Structure` or `Structure With Time` format) or a root Outport block to log variable-size signals. Then use the To Workspace variable with the From Workspace block.

Alternatively, create a MATLAB structure that contains variable-size signal data. For each `values` field in the structure, include a `valueDimensions` field that specifies the run-time dimensions for the signal. For details, see Simulink Models Using Variable-Size Signals on page 77-9.

Store Data for Model Linked to Data Dictionary

When you use a From Workspace block in a model that is linked to a data dictionary, you must choose the location to store the data that the block refers to. Set the value of the **Data** parameter based on the workspace or dictionary that contains the target data to load. For more information, see “Load Data Using the From Workspace Block” on page 70-65.

Sample Time

The From Workspace block **Sample time** parameter specifies the sample time to load data from a workspace. The timestamps in the workspace data must be monotonically nondecreasing. For details, see “Specify Sample Time” on page 7-3.

Interpolate Missing Data Values

To use linear Lagrangian interpolation to compute data values for time hits that occur between the time hits for which the workspace supplies the data, select **Interpolate data**.

For variable-size signals, clear **Interpolate data**.

Specify Output After Final Data

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**

In the From Workspace block documentation, see the **Form output after final data value by** parameter.

Detect Zero Crossings

By default, the From Workspace block does not enable zero-crossing detection. Zero-crossing detection locates discontinuities, without resorting to excessively small time steps.

The **Enable zero-crossing detection** parameter applies only if the sample time is continuous (`0`).

If you select the **Enable zero-crossing detection** parameter, and if an input array contains multiple entries for the same time hit, Simulink detects a zero crossing at that time hit.

For bus signals, Simulink detects zero crossings across all leaf bus elements.

See Also

Blocks
From Workspace | Signal Editor | To Workspace

Related Examples

- “Use From Workspace Block for Test Case” on page 70-33
- “Load Data Using the From Workspace Block” on page 70-65
- “Comparison of Signal Loading Techniques” on page 70-21

Load State Information

In this section...

Import Initial States

To initialize a simulation, you can use:

- Final state information (with or without `ModelOperatingPoint` object) from a previous simulation
- State information that you create in MATLAB

Use **Configuration Parameters** > **Data Import/Export** parameters to import initial states.

1. Enable the **Initial state** parameter.
2. In the **Initial state** edit box, enter the name of the variable for the state information that you want to use for initialization.

The initial values that the variable specifies override the initial state values that the blocks in the model specify in initial condition parameters.

You can specify `Dataset`, structure, or structure with time data.

Initialize a State

You can initialize a specific state. This example creates an initial state structure for the `x2` state of the `vdp` model. The `x1` state is not initialized in the structure. Therefore, during simulation, Simulink uses the value in the Integrator block associated with the `x1` state.

1. Open the model.

   ```
   open_system('vdp');
   ```

2. Set the `SaveFormat` model argument to `'Structure'`.

   ```
   set_param('vdp','SaveFormat','Structure');
   ```

3. Obtain an initial state structure.

   ```
   states = Simulink.BlockDiagram.getInitialState('vdp');
   ```

4. Set the initial value of the signals structure element associated with `x2` to 2.

   ```
   states.signals(2).values = 2;
   ```

5. Remove the signals structure element associated with `x1`.

   ```
   states.signals(1) = [];
   ```

6. Use the `states` variable for the `vdp` model. Select the initial state configuration parameter.

   ```
   set_param('vdp','LoadInitialState','on','InitialState','states');
   ```

7 Simulate the model and examine the initial values of `x2` and `x1`.

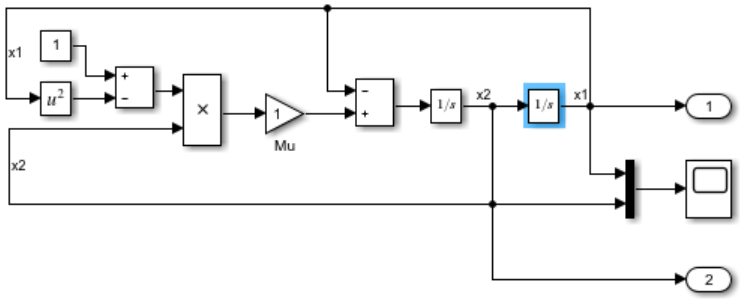
```
sim('vdp');
states
```

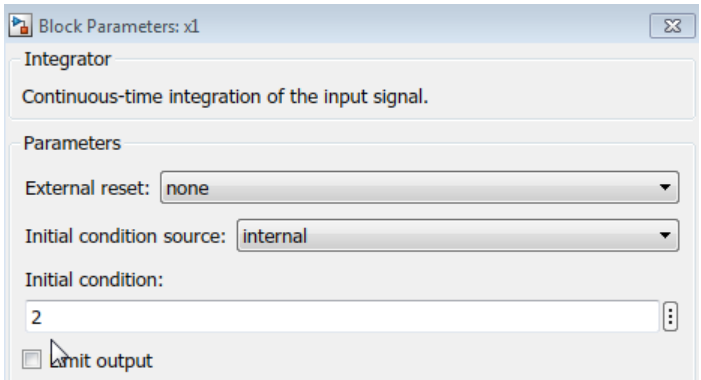
```
states =

  struct with fields:

      time: 0
    signals: [1×1 struct]
```

```
states.signals
```

```
ans =

  struct with fields:

              values: 2
          dimensions: 1
               label: 'CSTATE'
           blockName: 'vdp/x2'
           stateName: ''
    inReferencedModel: 0
          sampleTime: [0 0]
```

When you simulate the model, both states have the initial value of 2. The initial value of the `x2` state is assigned in the `states` structure, while the initial value of the `x1` state is assigned in its Integrator block.





Initialize States in Referenced Models

To initialize the states of a top model and the models that it references, use the structure or structure with time format or use operating point.

If the top model is in rapid accelerator mode, you cannot load discrete state data.

See Also

Related Examples

- “Save State Information” on page 72-81

More About

- “State Information” on page 72-76

Load Simulation Inputs Using the Root Inport Mapper

Map Data Using Root Inport Mapper Tool

In this section...

Use the Root Inport Mapper tool to import, visualize, and map signal and bus data to root-level inports.

Root-level inport mapping meets most modeling requirements and maintains model flexibility (for supported signal data, see “Create Signal Data for Root Inport Mapping” on page 71-9).

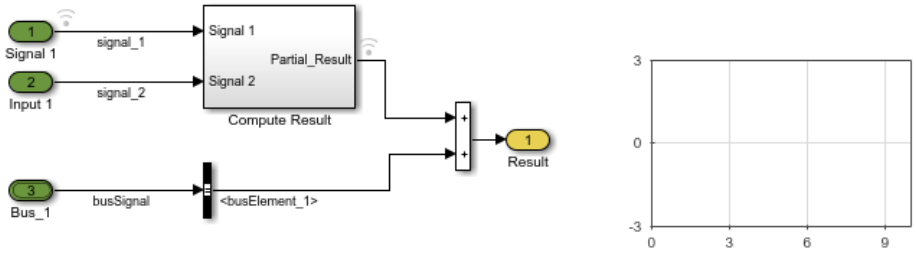
- Test your model with signals from the workspace and use your model as a referenced model in a larger context without any modification. Test signals in your model without disconnecting the inports and connecting sources to them.
- Use the Root Inport Mapper tool to update the **Input** parameter based on the signal data that you import and map to root-level inports.
- Visually inspect signal data without loading all the signal data into MATLAB memory.

To use the Root Inport Mapper tool:

1. Create signal data in the MATLAB workspace.
2. For a Simulink model, import the data from the workspace. You can visualize the data you import.
3. Map the data to root-level inports.
4. Simulate the model.
5. Save the Root Inport Mapper scenario.

The Model

This model has three root-level Inport blocks. Two of the Inport blocks output scalar signals and the other Inport block outputs bus data. Open the model.



This example shows how you can use the Root Inport Mapper tool to test the model with data. This approach can be useful for performing standalone testing of a model that another model references.

Create Signal Data

You can define the signal data as MATLAB `timeseries` objects.

1. Define the time values for the signal data.

   ```
   sampleTime = 0.01;
   endTime = 10;
   numberOfSamples = endTime * 1/sampleTime +1;
   timeVector = (0:numberOfSamples) * sampleTime;
   ```

2. Create the data for the two scalar signals. Naming the data variable to match the name of the corresponding signal makes it easier to map data to signals.

   ```
   signal_1 = timeseries(sin(timeVector)*10,timeVector);
   signal_2 = timeseries(rand(size(timeVector)),timeVector);
   ```

3. Create the signals for the bus.

   ```
   busSignal.busElement_1 = timeseries(cos(timeVector)*2,timeVector);
   busSignal.busElement_2 = timeseries(randn(size(timeVector)),timeVector);
   ```

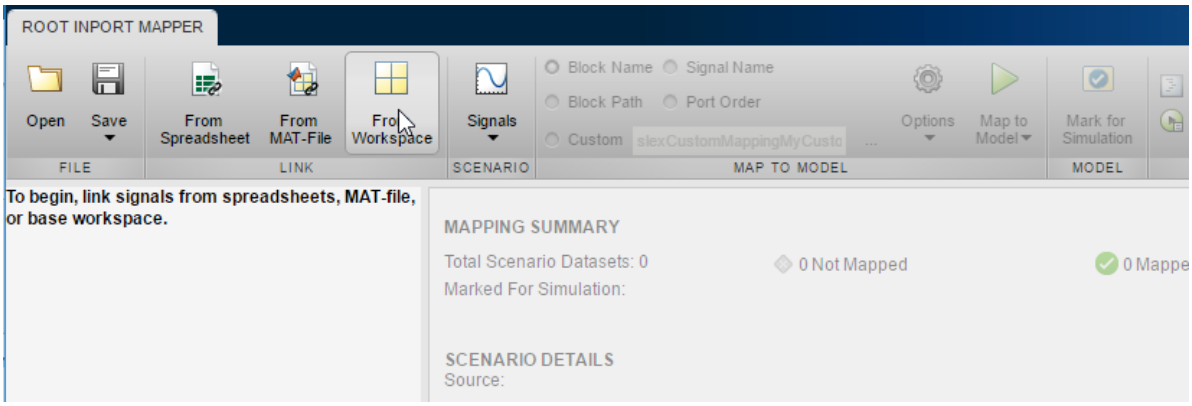
4. Create the bus object for the output data type of the `Bus_1` Inport block. You can create the bus object from the bus signal that you defined. Use a bus object for bus signals that cross model reference boundaries.

   ```
   busInfo = Simulink.Bus.createObject(busSignal);
   ```

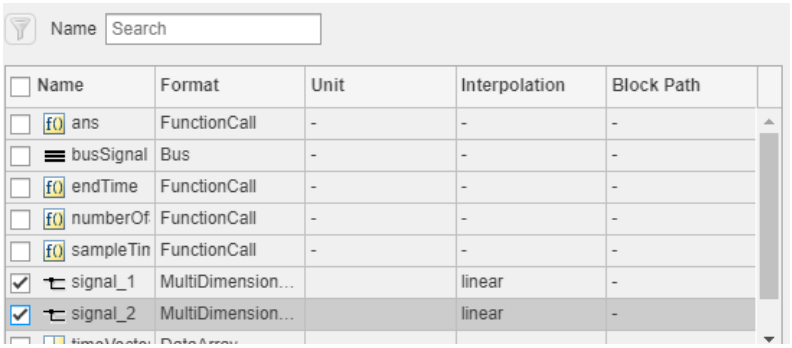
Import and Visualize Workspace Signal Data

Import the signal data that you created from the workspace into the Root Inport Mapper tool. Then you can use the tool to visualize the imported data.

1. Open the Root Inport Mapper tool. Open the Block Parameters dialog box for one of the Inport blocks in the model and click **Connect Input**.
2. In the Root Inport Mapper tool, select the **From Workspace** button.



3. In the Import dialog box, specify a MAT-file to save signals to.
4. To clear the data variables, click the **Name** check box. Then click the check boxes for the `busSignal`, `signal_1`, and `signal_2` signals.

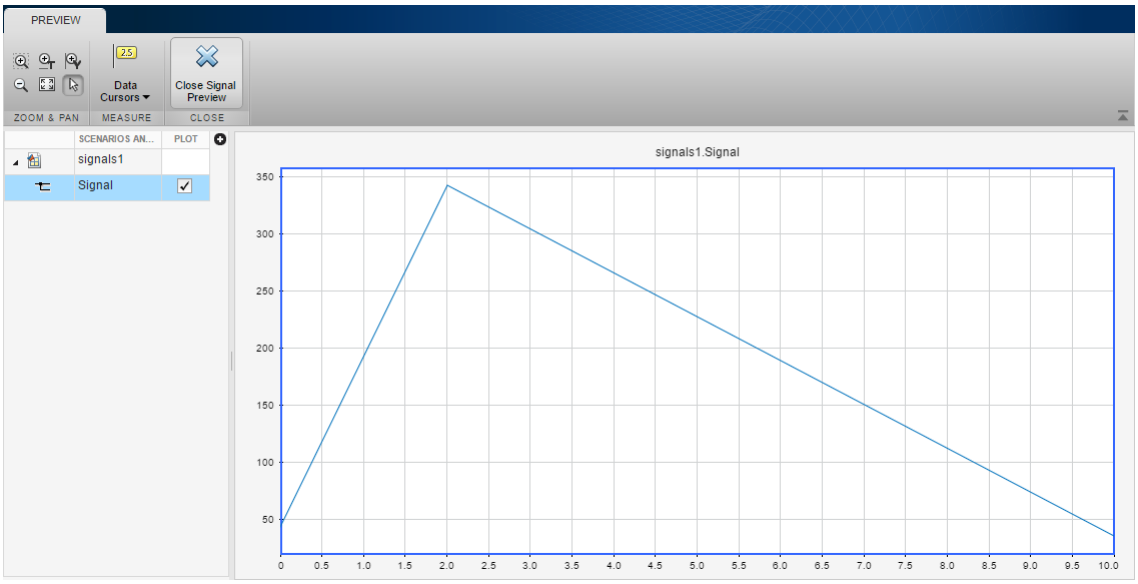


Although in this example you select all the signals, you can select a subset of signals.

5 You can visualize signals. In the Root Inport Mapper dialog box toolbar, click **Signals > Edit MAT-File**.

6 In the Select Linked MAT-file window, select the MAT-file to which you saved the signals and click **OK**.

The **Signal Editor** user interface appears. You can select signals to plot. For example, to see a plot of `signal_1`, in the Navigation pane, expand the scenario data set (in this example, the top node, `untitled`) and then expand the `signal_1` entry. Select the check box for `signal_1(1,1,:)` to plot the data.



7 Close the **Signal Editor** user interface by clicking the **Close** button.

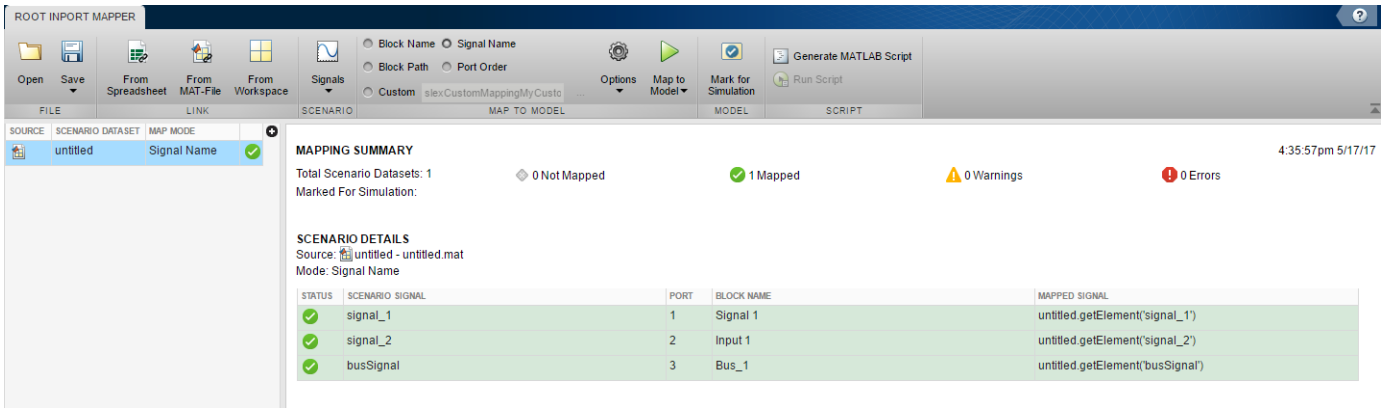
Map the Data to Inports

After you import the data, you map which data to use for specific Inport blocks.

1 Select the map mode, which specifies the criteria the mapping uses. In the toolbar, select the **Signal Name** option button.

The signals in this model have names, so mapping based on signal names makes it very clear which data is going to an Inport block.

2. You can specify options for the mapping. In the toolbar, select **Options**. Select **Update Model**, which updates the model after you do the mapping. Compiling the model verifies that signal dimensions and data types match between the data and the Inport blocks.
3. Map the data. In the Navigation pane, select the scenario data set. In the toolbar, click **Map to Model**. The dialog box shows the mapped data.



Save the Mapping and Data

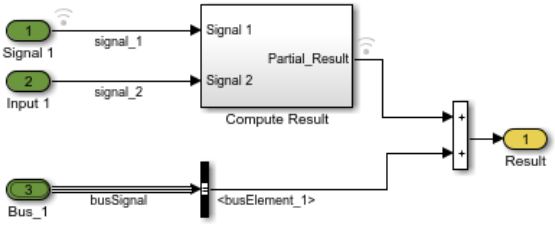
If you want to reuse the mapping and data that you have set up, you can save it as a scenario. In the Root Inport Mapper tool, click **Save** > **Save As** and save the scenario as an `.mldatx` file.

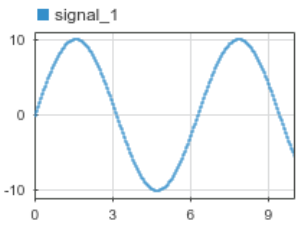
Simulate the Model

1. In the Navigation pane, select the scenario data set.
2. In the toolbar, click **Mark for Simulation**.

   The model is now set up to simulate using the workspace signal data that you mapped to root-level Inport blocks.
3. Simulate the model.

   This model includes a Dashboard block that shows the data used during simulation for `signal_1`. The plot matches the plot you did when you visualized the data as part of the data import process.





See Also

Related Examples

- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Import Signal Data for Root Inport Mapping” on page 71-14
- “View and Inspect Signal Data” on page 71-17
- “Map Signal Data to Root Input Ports” on page 71-18
- “Create and Use Custom Map Modes” on page 71-32
- “Root Inport Mapping Scenarios” on page 71-34
- “Comparison of Signal Loading Techniques” on page 70-21

Map Root Inport Signal Data

In this section...

To import, visualize, and map signal and bus data to root-level input ports, use the Root Inport Mapper tool or the `getRootInportMap` function. At the top level of a model or referenced model, root-level input ports include:

- Inport blocks
- Enable blocks
- Trigger blocks

Root-level input ports import data from the MATLAB workspace based on the value of the **Configuration Parameters > Data Import/Export > Input** parameter.

Root-level inport mapping imports signal data in a way that meets most modeling requirements and maintains model flexibility. You can:

- Test your model with signals from the workspace and use your model as a referenced model in a larger context without any modification.
- Update the **Input** parameter based on the signal data you import and map to root-level inports.
- Visually inspect signal data without loading all the data into MATLAB memory.

Tip To determine whether another data import technique meets your specific modeling requirements (such as the amount of data or the storage location) better, see “Comparison of Signal Loading Techniques” on page 70-21.

Open the Root Inport Mapper Tool

Use either of these approaches to open the Root Inport Mapper tool:

- In the **Configuration Parameters > Data Import/Export** pane, click **Connect Input**.
- In the block parameters dialog box for the Inport block, select **Connect Input**.

Command-Line Interface

You can use the `getRootInportMap` to create a custom object to map signals to root-level input ports and `getSlRootInportMap` to create a custom mapping mode. For more information, see “Create and Use Custom Map Modes” on page 71-32.

Import and Mapping Workflow

1 Identify and possibly create signal data to import and map on page 71-9.

2. Import on page 71-14 and inspect signal data on page 71-17.
3. Map the imported signal data on page 71-7. For example, you can map the signal data by block path or signal name.
4. Simulate the model using the mapped data. After associating a scenario with the model, you can generate scripts for simulation with scenarios on page 71-31 to perform batch simulations.
5. Optionally, save the current Root Inport Mapper scenario on page 71-34 for future reference or to share with other people.

Tip To extend the Root Inport Mapper tool map modes, you can create a custom mapping file function to map data to root-level inports.

Choose a Map Mode

To specify how you want the Root Inport Mapper tool to map the signal data to a model, select from these map modes in the **MAP TO MODEL** section of the toolbar:

- Block name — Connect signal data to ports based the name of a root-level input port block.
- Block path — Connect signal data to ports based the path of a root-level input port block.
- Signal name — Connect signal data to ports based on the name of the signal on a port.
- Port order — Connect sequential port numbers to the imported data.
- Custom — Connect signal data to ports based on the definitions in a custom mapping file.

Each supported input format supports one or more mapping modes. To import MATLAB `timeseries` data, for example, you use any mapping mode. To import data array signal data, use the port order mapping mode.

See Also

Functions
`getRootInportMap` | `getSlRootInportMap`

Related Examples

- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Create and Edit Signal Data” on page 69-2
- “Import Signal Data for Root Inport Mapping” on page 71-14
- “View and Inspect Signal Data” on page 71-17
- “Map Signal Data to Root Input Ports” on page 71-18
- “Create and Use Custom Map Modes” on page 71-32
- “Root Inport Mapping Scenarios” on page 71-34

More About

- “Create Signal Data for Root Inport Mapping” on page 71-9

Create Signal Data for Root Inport Mapping

In this section...

The first step for using the Root Inport Mapper tool is to know the source of signal data to import and map. You can use existing data (for example, in a Microsoft Excel spreadsheet), create data in a MAT-file, or use the Signal Editor interface to create signal data.

For a summary of the other steps involved in using the Root Inport Mapper tool, see “Import and Mapping Workflow” on page 71-7.

Choose a Source for Data to Import and Map

You can import data from these sources.

- Base workspace — You can selectively import data from the base workspace. For more information about supported data formats, see “Choose a Base Workspace and MAT-File Format” on page 71-10.
- Data files — You can selectively import signals contained in MAT-files and Microsoft Excel files. Each time that you import the contents of the file, the contents overwrite data already loaded for the file in the Root Inport Mapper tool.

  For more information, see “Choose a Base Workspace and MAT-File Format” on page 71-10 and “Supported Microsoft Excel File Formats” on page 71-12.

  Tip To load input data for a simulation from a Microsoft Excel spreadsheet, consider using the From Spreadsheet block. The From Spreadsheet block incrementally loads data from the spreadsheet during simulation. If you use a From Spreadsheet block, you do not need to do anything to handle changes to sheet values.

You can also use the Signal Editor interface to create and edit signal data. For more information, see “Create and Edit Signal Data” on page 69-2.

Choose a Naming Convention for Signals and Buses

When identifying signals to import, consider using a naming convention for signals and buses such that this grouping of data (scenario) is interchangeable. For example, you can have two MAT-files with the same set of variables named with the naming convention but different data values. Then, you can switch the scenarios of input data into and out of a model easily.

Choose a Base Workspace and MAT-File Format

The Root Inport Mapper tool supports the MATLAB data types or formats described in the table for imported signal data. For each data type, you can use the mapping modes indicated in the table.

Data Formats	Block Name	Block Path	Signal Name	Port Order	Custom
`Simulink.SimulationData.Dataset`	✓	✓	✓	✓	✓
MATLAB `timeseries`	✓		✓	✓	✓
MATLAB `timetable`	✓		✓	✓	✓
`Simulink.SimulationData.Signal`	✓	✓	✓	✓	✓
`Stateflow.SimulationData.State`	✓	✓	✓	✓	✓
Structure with time and structure without time				✓	
Data array				✓	
Array of buses	✓		✓	✓	✓
Asynchronous function-call signal on page 10-75	✓		✓	✓	✓

Note If your MAT-file or base workspace contains data in a format that the Root Inport Mapper tool does not support, the tool ignores that data.

Note Although the Root Inport Mapper tool accepts these formats, it can only link in a `Simulink.SimulationData.Dataset` object. To convert the data in your MAT-file to a `Simulink.SimulationData.Dataset` object, in the Root Inport Mapper From dialog box, select the **Convert signals into a scenario dataset and save to MAT-file** check box. Alternatively, use the `convertToSLDataset` function to convert your data.

Note When you specify a `timetable` as an element in the `Dataset` or a bus, the `timetable` must contain data for only one signal.

Dataset Signal Data

If data sets have nonunique element names, use the **Port Order** map mode.

MATLAB Timeseries Signal Data

If you have MATLAB timeseries data that includes enumeration data, and the enumeration class is not on your MATLAB path, the tool ignores that timeseries data.

Structure Signal Data

When converting structure signal data to datasets, the signals are named using the value contained in the label field of the signal field of the structure signal.

Array Signal Data

The Root Inport Mapper tool tries to map the data array to a single input port. In this case, you can choose any of the map modes.

Bus Signal Data for Root Inport Mapping

The signal data that you import and map to a root-level Inport block can include bus data. You cannot map bus signals to a root-level Enable or Trigger block.

1. In the MATLAB workspace, create or load a bus object on page 76-44 for the bus data you want to import and map.
2. If you create a bus object in the base workspace, save the bus object definition to a MAT-file, such as `d_myBusObj.mat`.
3. Create a separate MAT-file that contains the bus data you want to import for the bus object. Use one of these approaches:
   - Use an existing MAT-file that already contains a MATLAB struct or `Simulink.SimulationData.Dataset` object.
   - Create the bus in the base workspace and then save it to a MAT-file.
4. Set up the model to load the bus object.
   - For root-level Inport blocks that you map signals to, set the **Data type** field to `Bus`. Specify the name of the variable for the bus object to be used for signal mapping.
   - Load into the model the MAT-file that includes the bus objects used for mapping. For example, use a `PreLoadFcn` callback function. For details, see “Alternative Workflows to Load Mapping Data” on page 71-27.

Create Signal Data in a MAT-File for Root Inport Mapping

You can create signal data in a MAT-file to use for root-inport mapping. For example, you can import three signals (`signal1`, `signal2`, and `signal3`) and save the signals in a MAT-file. The `Simulink.SimulationData.Signal` objects include signal names, block names, block paths, and port order index values.

You can use the `convertToSLDataset` function to convert MAT-file contents to `Simulink.SimulationData.Dataset` objects.

1. In MATLAB, create three `Simulink.SimulationData.Signal` objects, specifying signal names, block paths, and port order index values.

```
signal1 = Simulink.SimulationData.Signal;
signal1.Name = 'signal1';
signal1.BlockPath = Simulink.SimulationData.BlockPath('Out1');
signal1.PortType = 'inport';
signal1.PortIndex = 1;
```

```
signal2 = Simulink.SimulationData.Signal;
signal2.Name = 'signal2';
signal2.BlockPath = Simulink.SimulationData.BlockPath('Out2');
signal2.PortType = 'inport';
signal2.PortIndex = 2;

signal3 = Simulink.SimulationData.Signal;
signal3.Name = 'signal3';
signal3.BlockPath = Simulink.SimulationData.BlockPath('Out3');
signal3.PortType = 'inport';
signal3.PortIndex = 3;
```

2. In the MATLAB workspace, select `signal1`, `signal2`, and `signal3`. Right-click the selection, and in the context menu, click **Save as**. Save the file as `mySigData.mat`.
3. Open the MAT-file.

```
open mySigData.mat

ans =

    signal1: [1x1 Simulink.SimulationData.Signal]
    signal2: [1x1 Simulink.SimulationData.Signal]
    signal3: [1x1 Simulink.SimulationData.Signal]
```

You can use the **Signal Name**, **Block Name**, **Block Path**, or **Port Order** map mode with this MAT-file. Based on your map mode, the Root Inport Mapper tool maps the signal data from the MAT-file to corresponding ports.

Supported Microsoft Excel File Formats

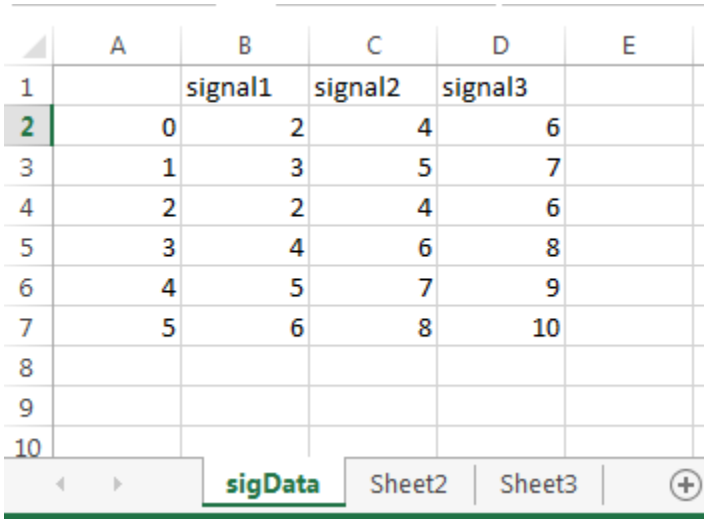
You can use the Root Inport Mapper tool to import data from Excel spreadsheets. You can also use the Root Inport Mapper tool to import signal data in CSV files on a Windows system with Microsoft Office installed. The Root Inport Mapper tool does not support Excel spreadsheet charts.

- Use sheet names that follow MATLAB variable name rules. If you import from a sheet whose name does not follow these rules, the Root Inport Mapper tool uses a modified sheet name. This modified sheet name follows the MATLAB variable name rules. For example, if you have a sheet name `Group Name`, the Root Inport Mapper uses the modified name `GroupName`.
- Use the first row of a sheet to specify signal names. Either specify a signal name for every signal or do not specify any signal names. If you do not specify any signal names, the tool assigns signal names using the format `Signal#`.
- For time values, use the first column of the remaining rows. The time values must increase for each row.
- Put signal values in the remaining columns.
- During import, the Root Inport Mapper tool converts formatted numbers from Excel spreadsheets to doubles.
- The Root Inport Mapper tool does not support block path mapping mode for spreadsheets.

This example of a Microsoft Excel spreadsheet is set up for root-inport mapping.

- The sheet name is `sigData`, which is a valid MATLAB variable name.
- The first row contains the signal names `signal1`, `signal2`, and `signal3`.

- The first column has six time values that increase for each row.
- In each row with a time value, columns to the right of the first column contain signal data values for each signal.



	A	B	C	D	E
1		signal1	signal2	signal3	
2	0	2	4	6	
3	1	3	5	7	
4	2	2	4	6	
5	3	4	6	8	
6	4	5	7	9	
7	5	6	8	10	
8					
9					
10					

sigData | Sheet2 | Sheet3

See Also

Related Examples

- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Create and Edit Signal Data” on page 69-2
- “Import Signal Data for Root Inport Mapping” on page 71-14
- “View and Inspect Signal Data” on page 71-17
- “Map Signal Data to Root Input Ports” on page 71-18
- “Root Inport Mapping Scenarios” on page 71-34

More About

- “Map Root Inport Signal Data” on page 71-7

Import Signal Data for Root Inport Mapping

In this section...

Import Signal Data

Before you can import data, identify the signals that you want to import and set up the data to use with root-level inport mapping. See “Create Signal Data for Root Inport Mapping” on page 71-9. For a summary of the other steps involved in using the Root Inport Mapper tool, see “Import and Mapping Workflow” on page 71-7.

For data import purposes, the Root Inport Mapper From MAT-File and From Workspace dialog boxes provide a **Convert signals into a scenario dataset and save to MAT-file** check box, selected by default. To convert the data in your MAT-file to a `Simulink.SimulationData.Dataset` object, select this check box. Alternatively, use the `convertToSLDataset` function to convert your data.

Note The Root Inport Mapper tool uses the term link to refer to the act of importing Simulink data. The sources from which you can link your data are in the LINK section of the tool.

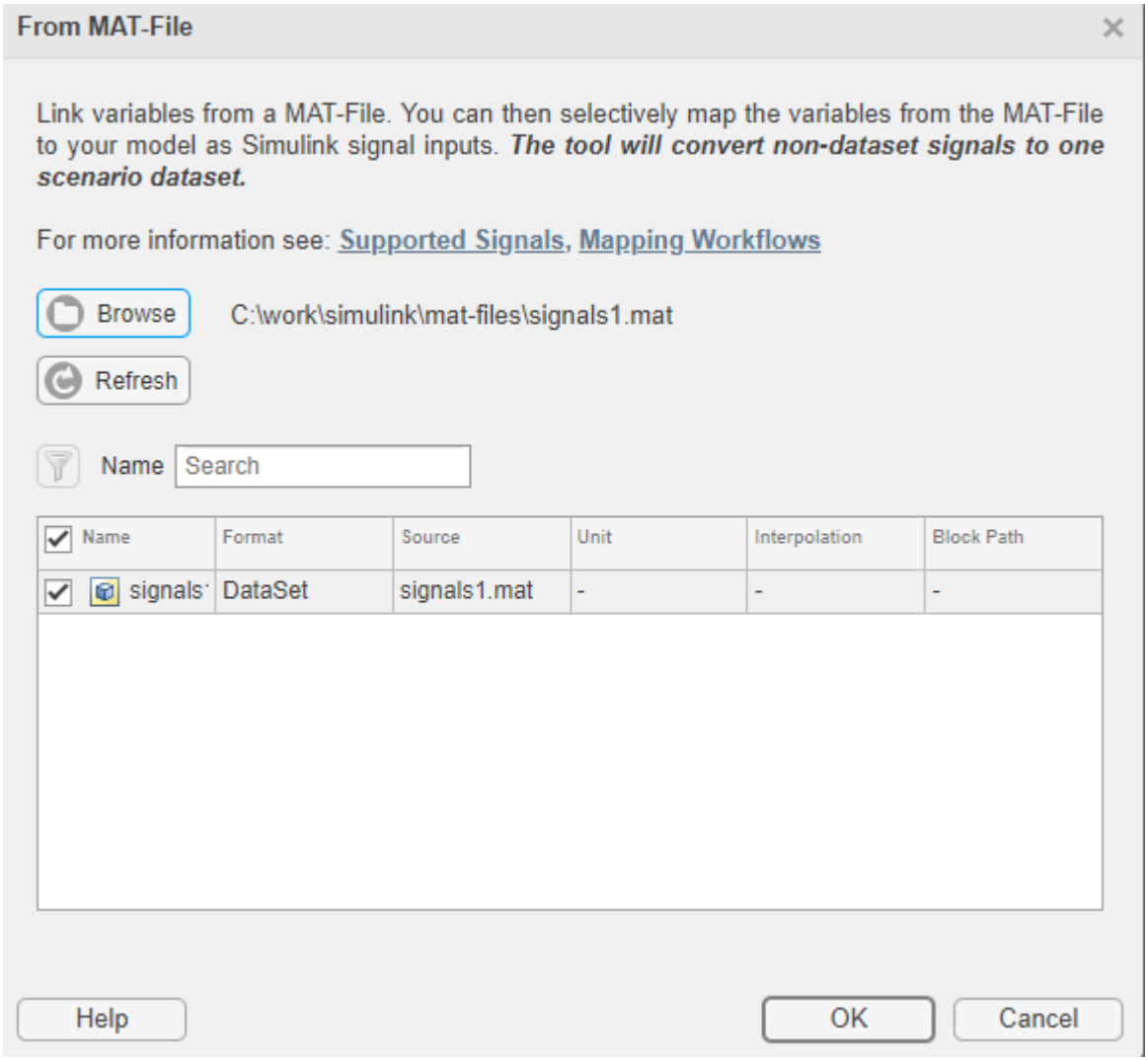
To import signal data for root-inport mapping:

1. Open the Root Inport Mapper tool. In the **Configuration Parameters > Data Import/Export** pane, click **Connect Input**.
2. In the LINK section, select the data source.

   - To browse to the MAT-file or spreadsheet file that contains the signals you want to import, select **From Spreadsheet** or **From MAT-File** and browse to the MAT-file or spreadsheet file. To return to the LINK section, click **OK**.

     Note To import bus data for root inport mapping, see “Import Bus Data” on page 71-15.
   - To display a list of base workspace variables that you can import, select **From Workspace**. Select the variables that you want to import and click **OK**.

   The From dialog box displays the contents of the spreadsheet, file, or base workspace.



3 Select the data that you want to import, and then click **OK**.

The Root Inport Mapper tool displays the imported data.

Alternatively, you can create signal data to map using the **Signals > New MAT-File** option. For more information, see “Create and Edit Signal Data” on page 69-2.

Import Bus Data

Use bus objects for bus data that you want to import and map to root inports or bus element ports.

Store the bus objects in a MAT-file. Use a different MAT-file that contains the bus data that you want to import for the bus object. This file can be an existing MAT-file that already contains a MATLAB `struct`. You can also create the bus in the base workspace and save it to a MAT-file. For more information, “Bus Signal Data for Root Inport Mapping” on page 71-11.

To import the bus data, in the LINK section of the Root Inport Mapper toolstrip, click **From MAT-File**. Select the MAT-file that contains the bus data and click **OK**.

Import Signal Data from Other Sources

Use the Root Inport Mapper tool to import signals from other sources.

- To import signals from models that contain Signal Editor blocks, see “Import Data from Signal Editor” on page 71-16.
- You can import Excel spreadsheet data. The Root Inport Mapper tool imports a worksheet as a `Simulink.SimulationData.Dataset` object that contains timeseries elements.
- To import test vectors from Simulink Design Verifier, see “Import Test Vectors from Simulink Design Verifier Environment” on page 71-16.

Import Data from Signal Editor

You can import and map data from the Signal Editor block in a MAT-file. For more information, see “Load Data with Interchangeable Scenarios Using Signal Editor Block” on page 71-37.

Import Test Vectors from Simulink Design Verifier Environment

You can import and map Simulink Design Verifier test vectors. This workflow requires a Simulink Design Verifier license.

Before importing, use the Simulink Design Verifier `sldvsimdata` function to convert a Simulink Design Verifier test structure to a `Simulink.SimulationData.Dataset` object. This file contains a test vector structure `sldvData`. Save the output to a MAT-file and then import that file into the Root Inport Mapper tool.

See Also

Signal Editor

Related Examples

- “Map Data Using Root Inport Mapper Tool” on page 71-2
- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Create and Edit Signal Data” on page 69-2
- “View and Inspect Signal Data” on page 71-17
- “Map Signal Data to Root Input Ports” on page 71-18
- “Root Inport Mapping Scenarios” on page 71-34

More About

- “Map Root Inport Signal Data” on page 71-7
- “Exporting Signal Group Data” on page 75-84

View and Inspect Signal Data

After you import signal or bus data, you can view and inspect signal data using tools such as:

- Signal Editor on page 69-2
- Simulation Data Inspector on page 29-2
- The `Simulink.SimulationData.Dataset` or `Simulink.SimulationData.DatasetRef` `plot` method, where the Signal Preview window contains an **Open Simulation Data Inspector** button. Click this button to plot the data using the Simulation Data Inspector.

For a summary of the other steps involved in using the Root Inport Mapper tool, see "Import and Mapping Workflow" on page 71-7.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `plot`

Related Examples

- "Map Data Using Root Inport Mapper Tool" on page 71-2
- "Create Signal Data for Root Inport Mapping" on page 71-9
- "Load Data to Root-Level Input Ports" on page 70-35
- "Map Signal Data to Root Input Ports" on page 71-18
- "Root Inport Mapping Scenarios" on page 71-34

More About

- "Map Root Inport Signal Data" on page 71-7

Map Signal Data to Root Input Ports

In this section...

After you import data, map signal data to root input ports by selecting map modes and options and selecting data.

For a summary of the other steps involved in using the Root Inport Mapper tool, see “Import and Mapping Workflow” on page 71-7.

Select Map Mode

To map signal data to root-level ports, use one of these map modes in the Map To Model section of the Root Inport Mapper toolstrip. The mapping mode you select from the toolstrip, such as **Block Name** or **Port Order**, is maintained between MATLAB sessions and models. You do not have to select the map mode each time you want to map signal data to root input ports.

Goal	Map Mode
Assign signals to ports according to the name of the root-inport block. If the name of a signal or bus element matches the name of a root-input port block, the data is mapped to the corresponding port.	**Block Name**
Assign signals to ports according to the block path of the root-input port block. If the block path of a signal matches the block path of a root-inport block, the data is mapped to the corresponding port.	**Block Path**
Assign signals 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, the signal is mapped to the corresponding port.	**Signal Name**
Assign sequential port numbers to the imported data, starting at 1. Map signals to the corresponding input ports. If there is more data than input ports, the remaining data is mapped to enable and then trigger input ports. If the data is not in the form of a dataset, it is processed in the order in which it appears in the data file.	**Port Order**

Goal	Map Mode
Assign signals to ports according to the definitions in a custom file. To create a custom mapping mode, see “Create and Use Custom Map Modes” on page 71-32.	**Custom**

Set Options for Mapping

If you want to set up mapping options, in the Map To Model section on the Root Inport Mapper toolstrip, click **Options**. The option you select from the toolstrip, such as **Update Model** or **Allow Partial**, is maintained between MATLAB sessions and models. You do not have to select the option each time you want to map signal data to root inports.

To map the signals, see “Map Data” on page 71-20.

Goal	Option
Update the model and review the data types of root-level input ports and imported data.	**Update Model**. Compare the signal data and input port parameters to the root-level port and display the results. If you do not select this option, the tool maps the imported data to the root-level input port but does not update the model.
Use strong data typing when mapping data from spreadsheets.	**Use Strong Data Typing with Spreadsheets**. Clear this check box to allow the Root Inport Mapper tool to automatically convert spreadsheet input signals to the data types of the corresponding root inports. The Root Inport Mapper tool can cast the spreadsheet data to only these data types: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. If you select this check box or if the root input port is not one of these data types, you may receive a data type mismatch error.
Import bus data that is only partially defined.	**Allow partial**. Confirm that any partially specified bus data you import maps properly to root-level input ports.
Identify unassigned root input ports and detect incomplete input data sets.	**Notify of Missing Signals**. Show inputs with missing signals.

Select Data to Map

To specify a subset of scenarios to map, click the down arrow on the **Map to Model** button. You can choose different mapping modes for different scenarios.

Goal	Option
Map all the scenario datasets (default).	**Map All**
Map the datasets of the scenarios currently selected in the Scenario Dataset section.	**Map Selected**
Map the disconnected datasets.	**Map Unconnected**
Map datasets that previously failed a mapping.	**Map Failed**

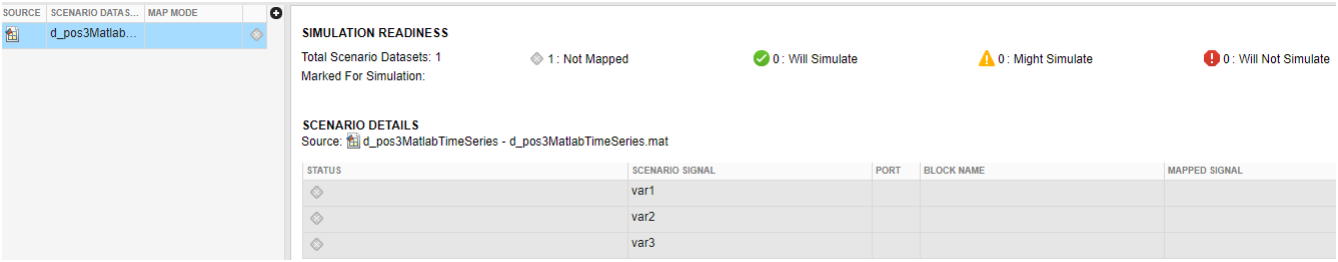
Goal	Option
Map datasets that previously caused warnings.	**Map Warned**

Map Data

After you import signals or buses, you can map data.

1. On the Root Inport Mapper toolbar, click **Map to Model**.

   The results of a signal mapping appear in the **Scenario Dataset** tab.
2. In the FILE section, click a data set to see the mapping results



- The **Simulation Readiness** section lists the input data and the status of the mapping.

  Note See “Understand Mapping Results” on page 71-20.
- The mapping definition for the input data is applied to the model.

After you save and close the model, when you load input data of the same scenario to the workspace, the model uses the mapping defined for that scenario.

For an example of mapping signal data to root-level inputs, see “Converting Harness-Driven Models to Use Harness-Free External Inputs” on page 71-22.

After you save the mapping definition for a model, you can automate data loading. For more information, see “Alternative Workflows to Load Mapping Data” on page 71-27.

Understand Mapping Results

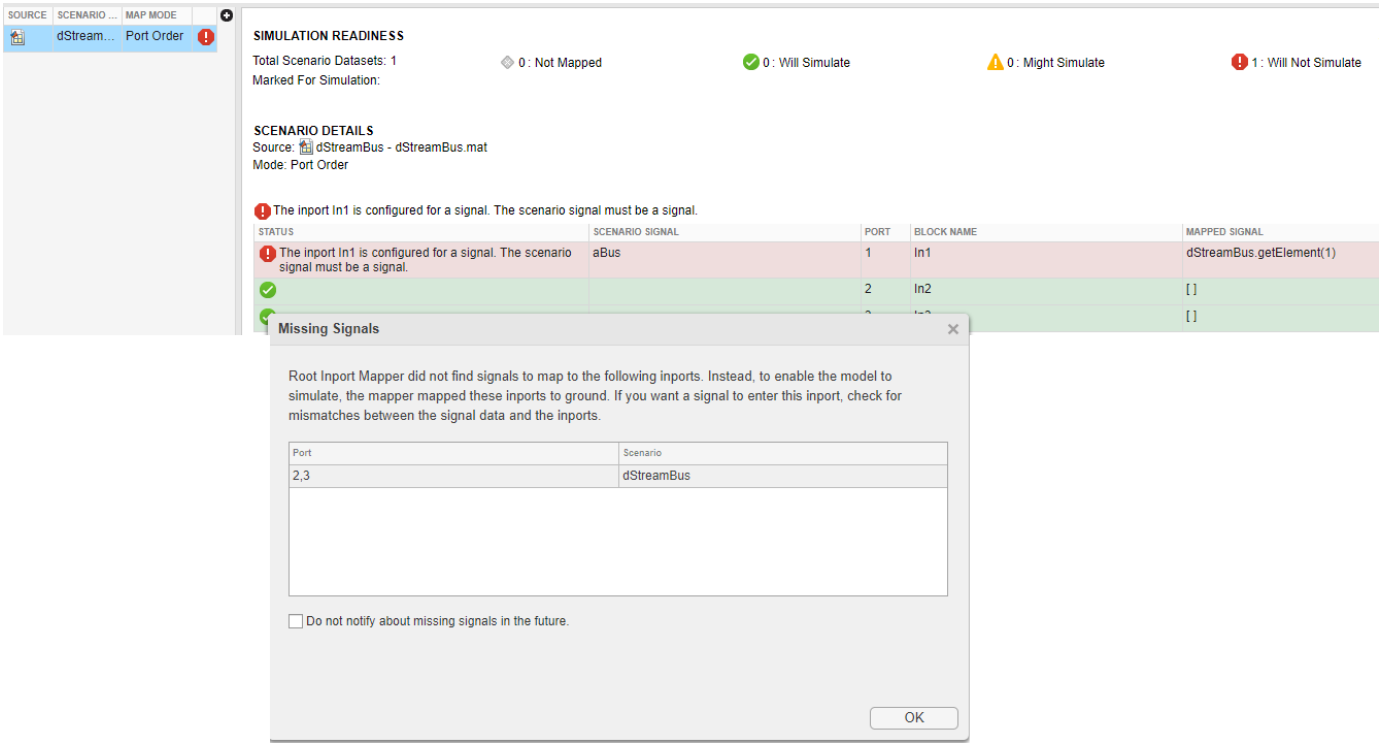
When you complete the import and map process, the **Simulation Readiness** section displays the results in the status area. The results depend on whether you select the **Update Model** option when you set up the mapping.

Status	Update Model	Continue Without Update Model
	The properties of the mapped data and the input port are appropriate for simulation.	The data type, dimension, and signal type properties of the data and input port are compatible.

Status	Update Model	Continue Without Update Model
	Not applicable	Comparison of data and root-level port data type, dimension, and signal type properties cannot determine whether there is a match. If you do not update the model before mapping, the tool cannot evaluate whether all the data types match unless you explicitly specify the input port data types. Confirm that you set these block parameters correctly: Inport block parameter **Data type** is not set to `Inherit:auto`. Inport block parameter **Dimension** is not set to `-1`. Inport block parameter **Signal type** cannot be `auto`.
	The properties of the mapped data and the input port are not appropriate for simulation.	One or more of the data types, dimensions, or signal types of the signal data are not compatible with the root-level input port.

To enable the model to simulate, if Root Inport Mapper does not find input port signals to map, it maps these input ports to ground and displays the mapped signal as empty (`[ ]`).

This figure shows mapping successes, failures, and ground assignments. If there are issues, the status column displays suggested resolutions. Read through and diagnose the issues.



Sometimes the **Simulation Readiness** section shows a warning or error, but your investigation of the elements indicates that there is no problem with mapping the data. In these cases, if you did not select the **Update Model** check box from the **Options** menu, select it and click **Map to Model** again.

In the Root Inport Mapper tool, clicking **Mark for Simulation** selects the **Input** check box in the **Data Import/Export** pane in the model Configuration Parameters dialog box. It also sets the value to the imported data variables. To apply the changes to the model configuration, in the **Data Import/Export** pane, click **OK**.

If your model uses configuration references to reference configuration sets, you cannot mark the model for simulation. To use this data to simulate the model with the Root Inport Mapper tool, use the Model Explorer to activate a configuration set first.

This graphic illustrates the application of the changes to the model configuration for the model in “Map Data” on page 71-20.

To inspect the imported data, you can:

- Connect the output to a scope, simulate the model, and observe the data.
- Log the signals and use the Simulation Data Inspector tool to observe the data.

To highlight the Inport block that is associated with the signal, select an item in the **Simulation Readiness** section. The selected Inport block is outlined with blue.

Note When the input is a bus, click the levels of the bus object to see the individual elements in the bus.

Converting Harness-Driven Models to Use Harness-Free External Inputs

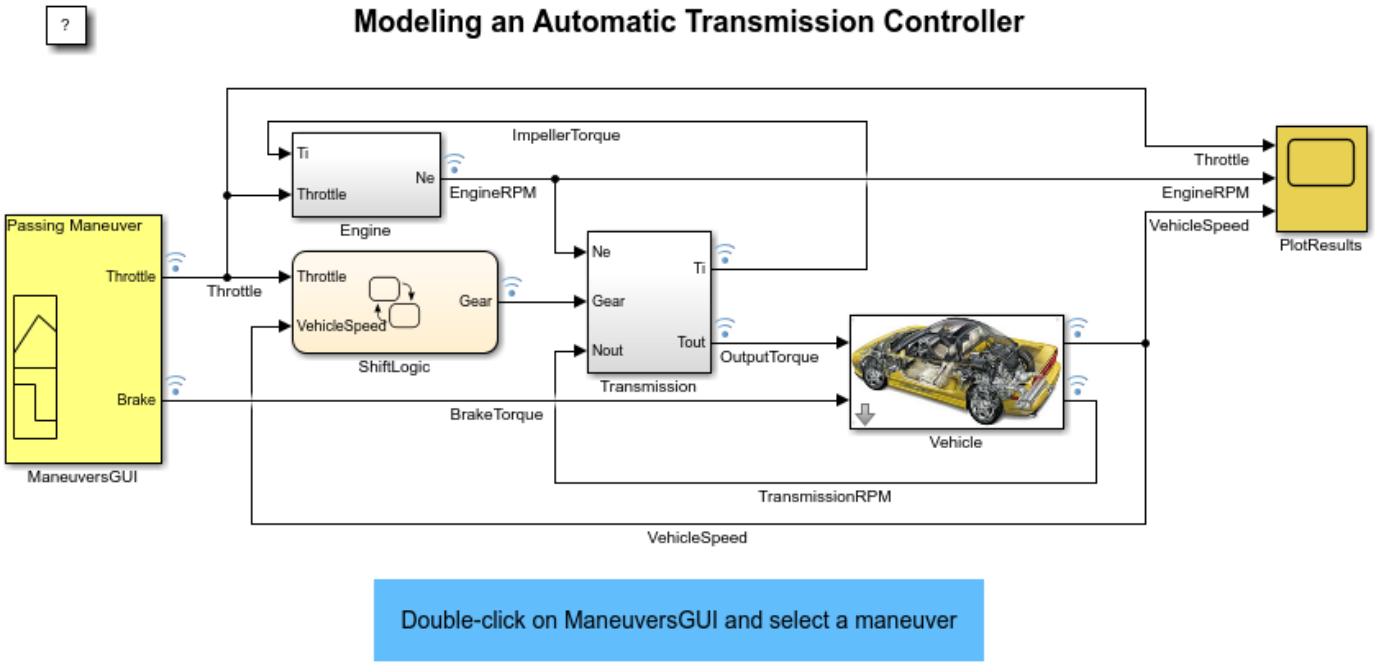
This example shows how to convert a harness model that uses a Signal Builder block as an input to a harness-free model with root inports. The example collects data from the harness model and stores it in MAT-files for the harness-free model. After storing the data, the example removes the Signal Builder block from the harness model and adds root inports to create a harness-free model. Then, the data in the MAT-files is mapped to the root inports of the model.

Save Harness Data to MAT-Files

Before converting the model to be harness-free, collect the test cases in the harness.

For this example, you will modify the model sldemo_autotrans from the **Modeling an Automatic Transmission Controller** example.

Open the example model. In the MATLAB Command Window, type `sldemo_autotrans`.



Exporting Signal Builder Block Groups

Export data that defines Signal Builder block signal groups to a MAT-file from the Signal Builder window. To export Signal Builder signal data, formatted as Simulink.SimulationData.Dataset, to a MAT-file, open the Signal Builder window and select **File** > **Export Data** > **To MAT-File**. In the dialog, enter a name for the MAT-file to contain the data and the number of the group you want to export. For this example, the file name is `slexAutotransRootInportPassingManeuver.mat` and the group number is 1 for the Passing Maneuver group.

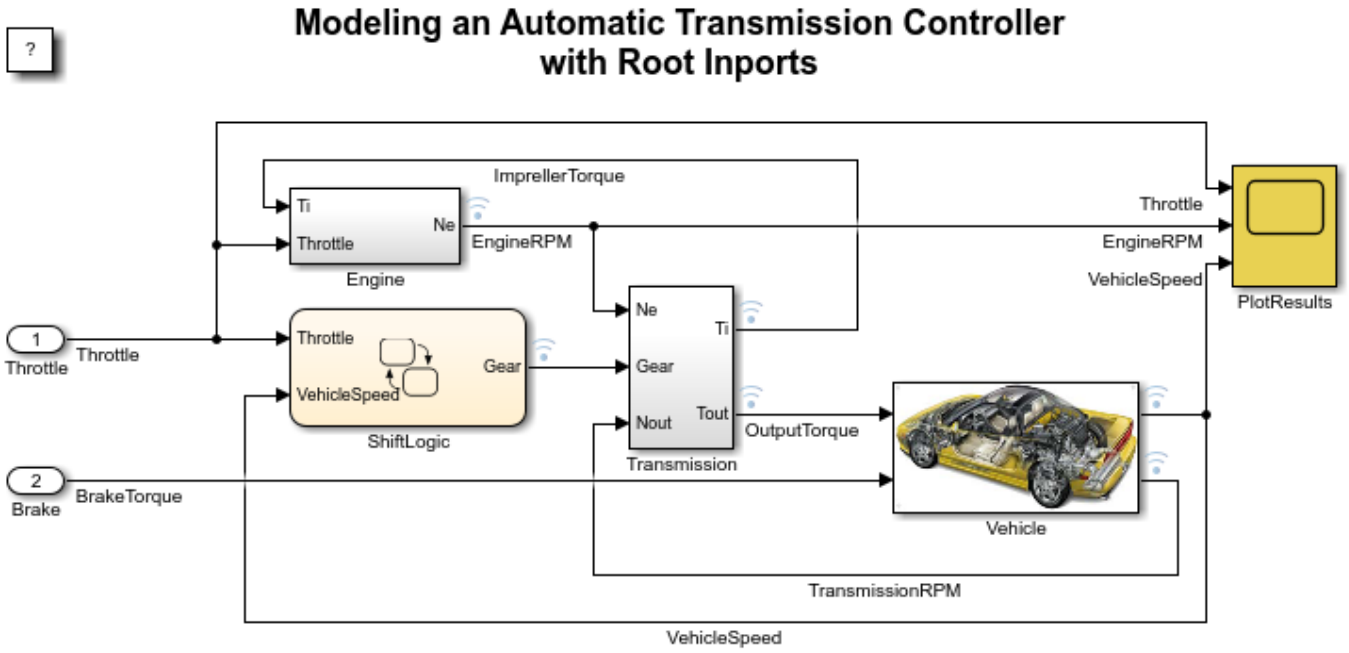
Remove the Signal Builder Block

Remove the Signal Builder block named ManeuversGUI and replace it with two inports.

1. Delete the Signal Builder block named ManeuversGUI.
2. From the **Simulink/Commonly Used Blocks** library, drag two inport blocks into the model.
3. Connect the input ports to the lines previously connected to the Signal Builder block.
4. Rename the inport ports. Name the input port connected to the Throttle line **Throttle**. Name the input port connected to the BrakeTorque line **Brake**.

Save the model as `slexAutotransRootInportsExample1.slx` or use the example `slexAutotransRootInportsExample.slx`.

The remaining steps of this example use the model `slexAutotransRootInportsExample.slx`. If you saved the model with a different name use your model name in the steps going forward.



Set Up Harness-Free Inputs

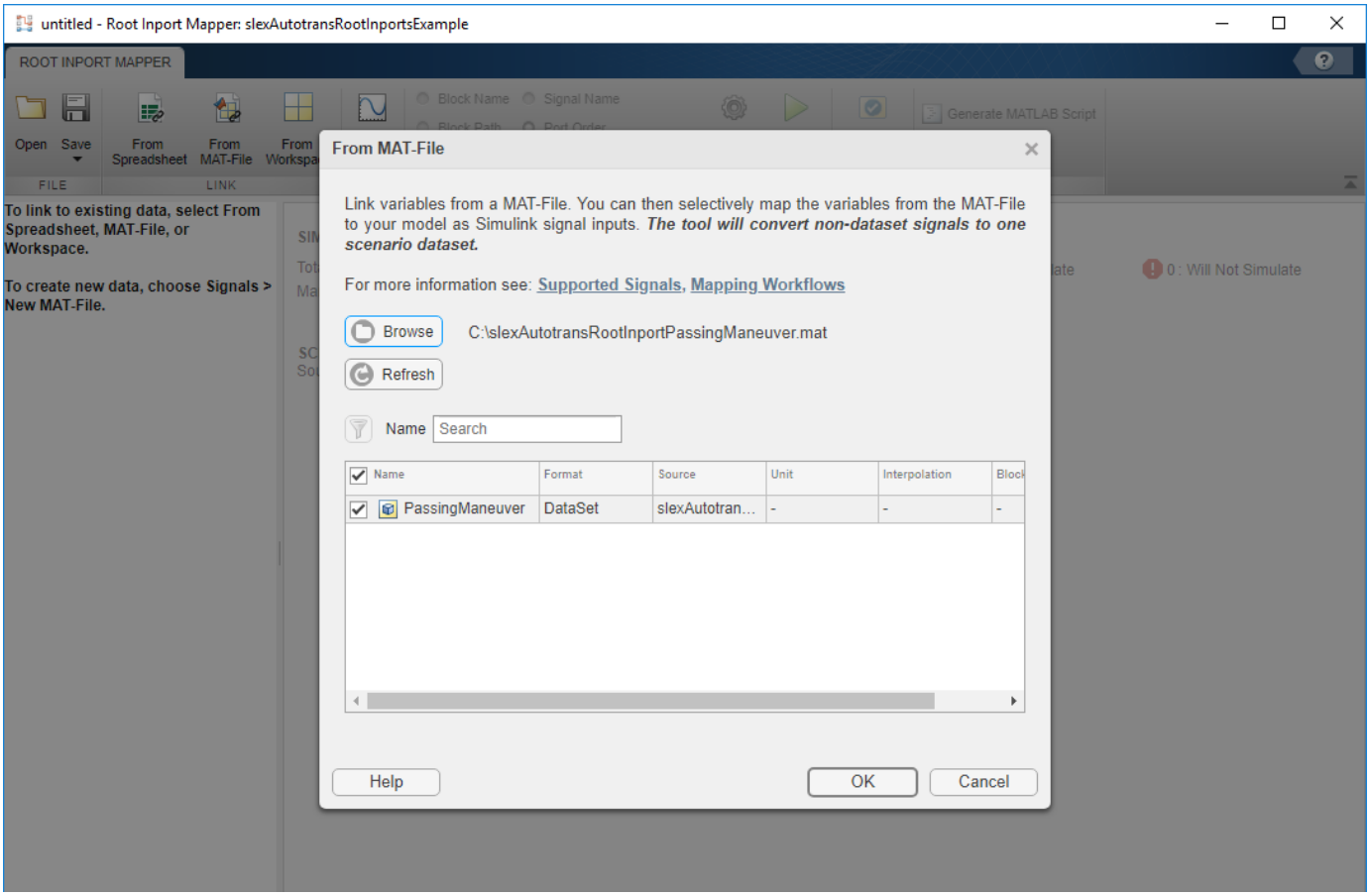
Now that the model is harness-free, set up the inputs that you already saved (See "Save Harness Data to MAT-Files").

In the Modeling tab, select Model Settings. In the Data Import/Export pane, click the **Connect Input** button.

Map Signals to Root Inport

The Root Inport Mapper tool opens.

The example uses this tool to set up the model inputs from the MAT-file and map those inputs to an input port, based on a mapping algorithm. To select the MAT-file that contains the input data, click the **From MAT-File** button on the Root Inport Mapper toolbar. When the link dialog appears, click the Browse button. In the browser, select the MAT-file that you saved earlier.



Select a Mapping Mode

When you select the MAT-file `slexAutotransRootInportPassingManeuver.mat` that contains the input data, determine the root input port to which to send input data. Simulink matches input data with input ports based on one of five criteria:

- **Port Order** - Maps in the order it appears in the file to the corresponding port number.
- **Block Name** - Maps by variable name to the corresponding root inport with the matching block name.
- **Signal Name** - Maps by variable name to the corresponding root inport with the matching signal name.
- **Block Path** - Maps by the BlockPath parameter to the corresponding root inport with the matching block path.
- **Custom** - Maps using a MATLAB function.

Earlier in this example, you saved input data to variables of the same name as the harness signals Throttle and Brake, and added input ports with names matching the variables. Given the set of conditions for the input data and the model input ports, the best choice for a mapping criteria is **Block Name**. Using this criteria, Simulink tries try to match input data variable names to the names of the input ports. To select this option:

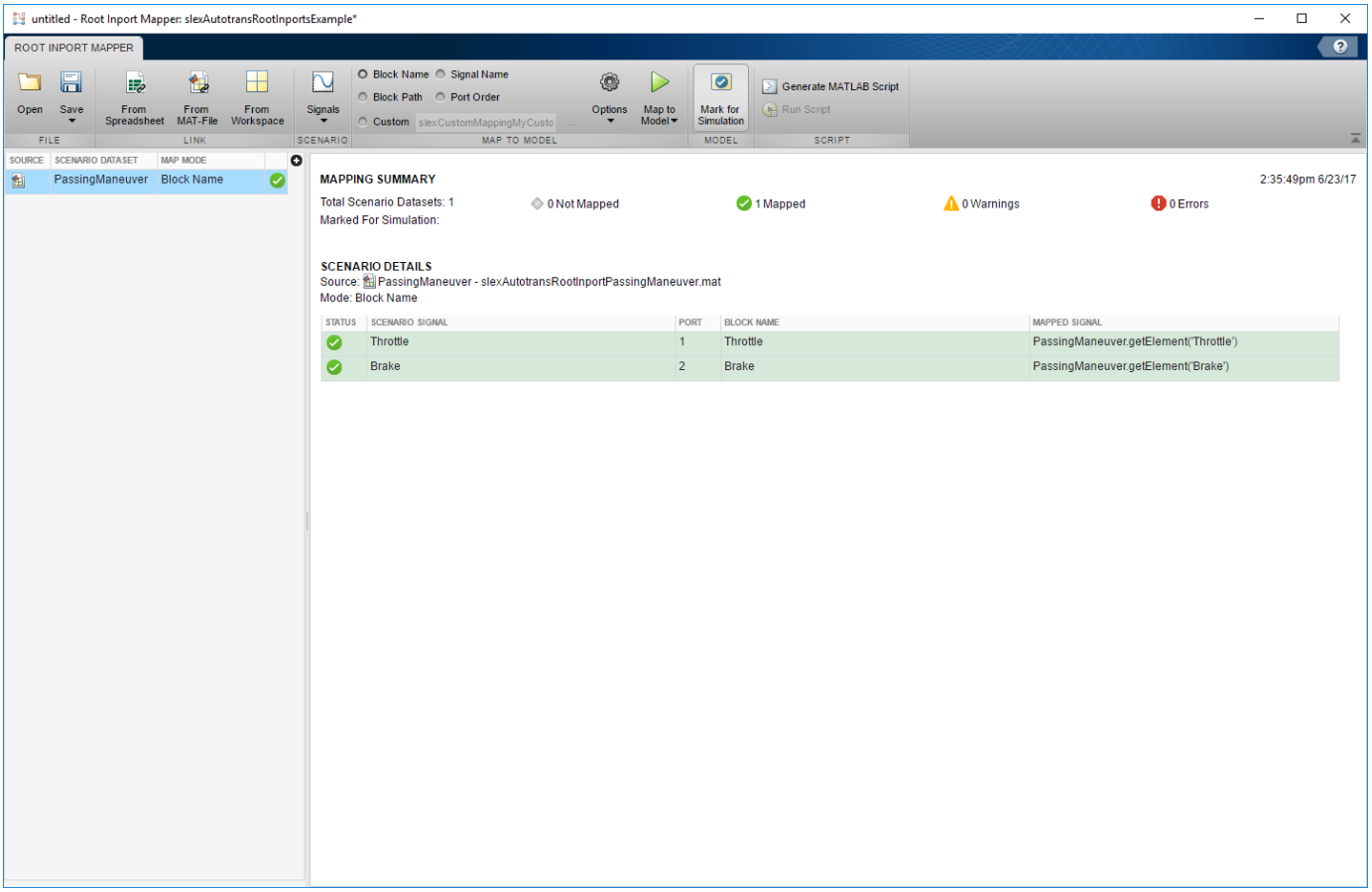
1 Click the **Block Name** radio button.

2 Click the **Options** button and select **Update Model**. This verifies the mapping.

3 Click the **Map** button.

When compiling the data, Simulink evaluates inports against the following criteria to determine compatibility issues. The status of this compatibility is reflected by the table colors green, orange, or red. Warnings and errors are flagged with diagnostic messages. If the **Options > Update Model** option is not selected, Root Inport Mapper determines the compatibility status by evaluating these block parameters and assigned signals:

- **Data Type** - Double, single, enum, ...
- **Complexity** - Real or complex
- **Dimensions** - Signal dimensions vs port dimensions

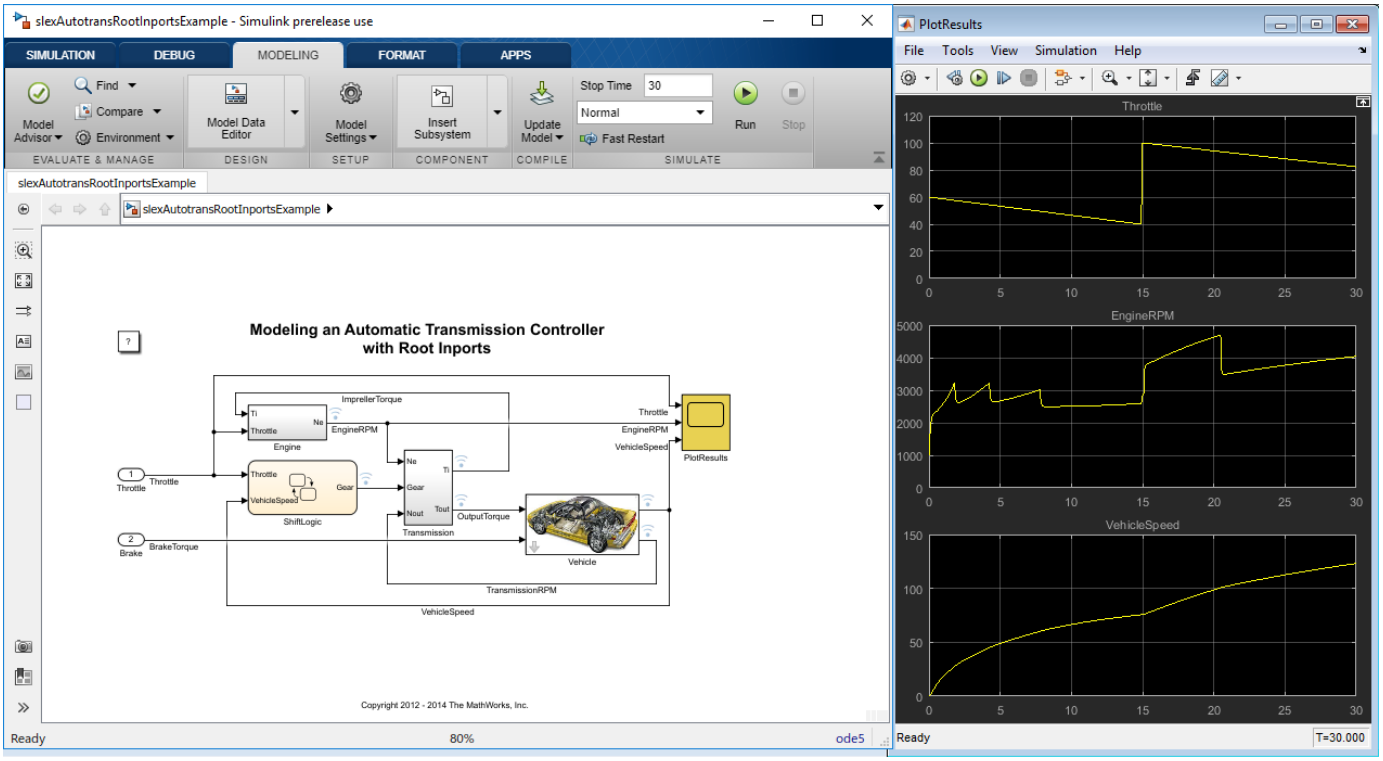


Finalize the Inputs to the Model

Review the results of the mapping compatibility. Click the Scenario Dataset 'PassingManeuver' in the scenario dataset list. To prepare for simulation, click **Mark for Simulation**. This action applies the mapping variables to the Configuration Parameter **Data Import/Export > External Input** text box. If this text box has content, it is overwritten.

Simulating the Model

With the changes applied you can now simulate the model and view the results. Run the model. To view the results of the simulation, double-click the Scope Block **PlotResults**.



Alternative Workflows to Load Mapping Data

After you save the mapping definition to a model, you can automate data loading and simulation. Consider one of the following methods.

Command Line or Script

To load data and simulate the model from the MATLAB command line, use commands similar to:

```
load('signaldata.mat');
simout = sim('model_name');
```

To automate testing and load different signal groups, consider using a script.

The following example code creates timeseries data and simulates a model after loading each signal group. It:

- Creates signal groups with variable names *In1*, *In2*, and *In3*, and saves these variables to MAT-files.
- Simulates a model after loading each signal group.

Note The variable names must match the import data variables in the **Configuration Parameters** > **Data Import/Export** > **Input** parameter.

```
% Create signal groups
fileName = 'testCase';
for k =1 :3
```

```
    % Create the timeseries data
    var1 = timeseries(rand(10,1));
    var2 = timeseries(rand(10,1));
    var3 = timeseries(rand(10,1));

    %create a dataset
    ds = Simulink.SimulationData.Dataset();
    ds = ds.addElement( var1, 'var1');
    ds = ds.addElement( var2, 'var2');
    ds = ds.addElement( var3, 'var3');

    % Save the data
    save([fileName '_' num2str(k) '.mat' ],'ds');
end
clear all

% After mapping and saving the model loop over signal groups and simulate
% Set the filename to append testcase # to
fileName = 'testCase';
% Loop backwards to preallocate
for k=3:-1:1
   % Load the MAT-file.
   load([fileName '_' num2str(k) '.mat']);

   % Simulate the model
   simOut{k} = sim('model_name');
end
```

Use the PreLoadFcn Model Callback

When you are satisfied with the data and mapping, you can configure your model to load a MAT-file containing the signal group into the MATLAB workspace. Call the `load` function in the `PreLoadFcn` callback for the model.

1. After saving the MAT-file, on the **Modeling** tab, click the **Model Settings** drop-down and select **Model Properties**.
2. In the Model Properties window, select the **Callbacks** tab and then the `PreLoadFcn` node.
3. Enter a command to load the MAT-file containing the signal data. For example:

   ```
   load d_signal_data.mat;
   ```
4. Click **OK** and save the model.

See Also

Related Examples

- “Create and Use Custom Map Modes” on page 71-32
- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Import Signal Data for Root Inport Mapping” on page 71-14
- “Create and Edit Signal Data” on page 69-2
- “View and Inspect Signal Data” on page 71-17
- “Root Inport Mapping Scenarios” on page 71-34

More About

- “Map Root Inport Signal Data” on page 71-7

Preview Signal Data

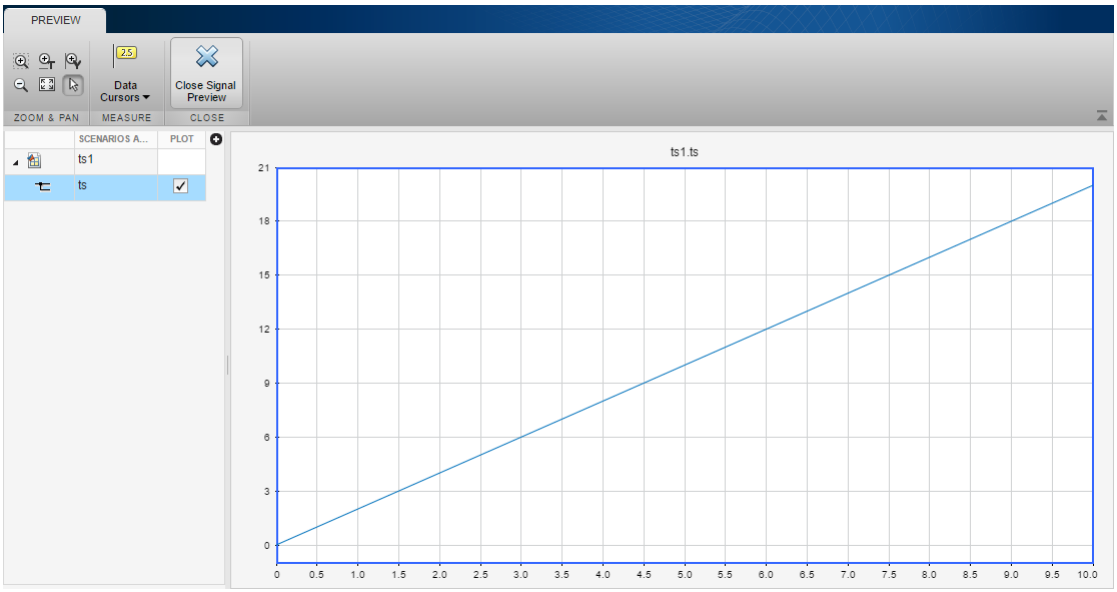
Preview input signal or bus data with the **Signal Preview** window. You can access this window from:

- From File block
- `Simulink.SimulationData.DatasetRef` and `Simulink.SimulationData.Dataset` `plot` methods

1 Preview input signal or bus data with the **Signal Preview** window.

- For From File block, browse to a MAT-file that contains the data you want to preview, then plot the data by clicking  .
- For `Simulink.SimulationData.Dataset` or `Simulink.SimulationData.DatasetRef` elements, use the `plot` method on the dataset.

  If you view and inspect signal using the `Simulink.SimulationData.Dataset` or `Simulink.SimulationData.DatasetRef` `plot` method, the Signal Preview window contains an **Open Simulation Data Inspector** button. Click this button to plot the data using the Simulation Data Inspector.



2 Explore the plots using the **Measure** and **Zoom & Pan** sections on the toolbar.

- In the **Measure** section, use the **Data Cursors** button to display one or two cursors for the plot. These cursors display the T and Y values of a data point in the plot. To view a data point, click a point on the plot line.
- In the **Zoom & Pan** section, select how you want to zoom and pan the signal plots. Zooming is only for the selected axis.

Type of Zoom or Pan	Button to Click
Zoom in along the T and Y axes.	

Type of Zoom or Pan	Button to Click
Zoom in along the time axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom in along the data value axis. After selecting the icon, on the graph, drag the mouse to select an area to enlarge.	
Zoom out from the graph.	
Fit the plot to the graph. After selecting the icon, click the graph to enlarge the plot to fill the graph.	
Pan the graph up, down, left, or right. Select the icon. On the graph, hold the left mouse button and move the mouse to the area of the graph that you want to view.	

See Also

From File | `Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `plot`

Generate MATLAB Scripts for Simulation with Scenarios

After associating a scenario with the model, you can generate a MATLAB script to perform batch simulations. These scripts enable you to connect multiple sets of input signals to your Simulink model for interactive or batch simulation. You can run simulations multiple times and quickly generate data. This topic assumes that you have a scenario ready to run (see “Root Inport Mapping Scenarios” on page 71-34).

1. Associate your scenario with the model.
2. In the **SCRIPT** section, click **Generate MATLAB Script** and supply a script name when prompted.
3. To run the script, click **Run Script**.
4. To evaluate the results of the simulation, see the base workspace.

The resulting script uses the `Simulink.SimulationInput` object and `parsim` function.

See Also

`Simulink.SimulationInput` | `parsim`

More About

- “Root Inport Mapping Scenarios” on page 71-34

Create and Use Custom Map Modes

You can create custom map modes to supplement the map modes that the Root Inport Mapper tool provides (see “Choose a Map Mode” on page 71-8).

For a summary of the other steps involved in using the Root Inport Mapper tool, see “Import and Mapping Workflow” on page 71-7.

Create Custom Mapping File Function

If you do not want to use the map modes in the Root Inport Mapper tool, create a custom mapping file function. For example, consider creating a custom mapping file function if:

- Your signal data contains a common prefix that is not in your model.
- You want to map a signal explicitly.

When the data contains a signal name that does not match one of the block names, a custom mapping function is useful for block name mapping.

For examples, see these files in the folder *`matlabroot`*`/help/toolbox/simulink/examples` (open).

File	Description
`BlockNameIgnorePrefixMap.m`	Custom mapping file function that ignores the prefix of a signal name when importing signals
`BlockNameIgnorePrefixData.mat`	MAT-file of signal data to be imported
`ex_BlockNameIgnorePrefixExample`	Model file into which you can import and map data

In addition, see “Using Mapping Modes with Custom-Mapped External Inputs”.

To create a custom mapping file function:

1. Create a MATLAB function with these input parameters:
   - Model name
   - Signal names specified as a cell array of character vectors
   - Signals specified as a cell array of signal data
2. In the function, call the `getRootInportMap` function to create a variable that contains the mapping object (for an example, see `BlockNameIgnorePrefixMap.m`).
3. Save and close the MATLAB function file.
4. Add the path for the new function to the MATLAB path.

To use the custom mapping file function:

1. Open the model that you want to import data to (for example, `ex_BlockNameIgnorePrefixExample`).
2. Open the Configuration Parameters dialog box for the model and select the **Data Import/Export** pane.

3. In the **Load from workspace** section, click **Connect Input**.
4. Import your signal (for example, `BlockNameIgnorePrefixData.mat`).
5. In the **MAP TO MODEL** section of the toolstrip, click **Custom**.
6. In the **Custom** text box, select the MATLAB function file (for example, `BlockNameIgnorePrefixMap.m`) using the browser.

   By default, this text box contains `slexcustomMappingMyCustomMap`, which is the custom function for "Attaching Input Data to External Inputs via Custom Input Mappings".

   Tip The Root Inport Mapper tool parses your custom code. Parsing reorders output alphabetically and verifies that data types are consistent.
7. Click **Options** and select the **Compile** check box.
8. Click **Map**.

   The model is compiled and the Root Inport Mapper tool gets updated.

   To understand the mapping results, see "Understand Mapping Results" on page 71-20.
9. Save and close the model.

After you save the mapping definition for a model, you can automate data loading. The next time that you load input data of the same signal group into the workspace, the model uses the mapping definition during simulation. For more information, see "Alternative Workflows to Load Mapping Data" on page 71-27.

Custom Mapping Modes Similar to Simulink Modes

If your custom mapping mode is similar to a Simulink mapping mode, use the `getSlRootInportMap` function in your custom mapping file function to perform the data mapping.

For an example of a custom mapping function that uses this function, see "Using Mapping Modes with Custom-Mapped External Inputs".

Command-Line Interface for Input Variables

Use the `getInputString` function to supply a set of input variables to:

- The `sim` command
- A list of input variables that you can paste in the **Configuration Parameters** > **Data Import/ Export** > **Input** parameter

See Also

Related Examples

- "Map Signal Data to Root Input Ports" on page 71-18

More About

- "Create Signal Data for Root Inport Mapping" on page 71-9

Root Inport Mapping Scenarios

In this section...

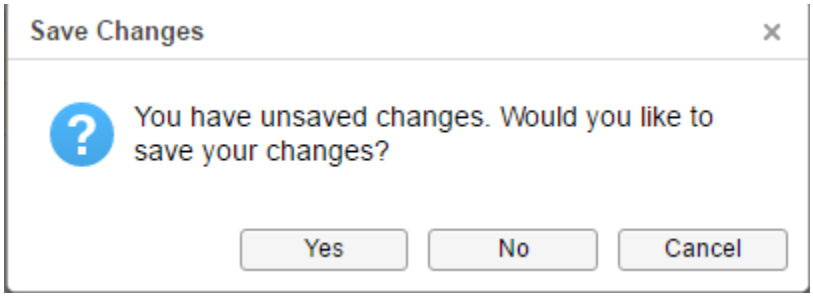
Use the Root Inport Mapper tool to create scenarios, save scenarios, and load previously saved scenarios. The Root Inport Mapper tool uses scenarios to save a snapshot of the current state of the imported and mapped signals in an MLDATX file. A scenario file contains information about the:

- Location of signal files (MAT-file or Microsoft Excel files)
- Location of the model
- Map mode
- Mapping options
- Mapped state

When sharing scenario files, include the scenario file and signal files (MAT-file or Microsoft Excel. Place the signal files in the last known location (where you used the Root Inport Mapper tool most recently) or the MATLAB path.

Open Scenarios

To open an existing scenario, click **Open**. If you are working in another scenario, the Root Inport Mapper tool displays a message.



To...	Click...
Open a new scenario. Remove the existing scenario without saving it.	**No**
Cancel opening a scenario.	**Cancel**
To save the existing scenario, click **Yes**. Then click the **Open** button again to open an existing scenario.	**Yes**

Save Scenarios

When the **Save** icon turns blue or when the model name in the title bar is has an asterisk (*), you can save a scenario.

1. On the Root Inport Mapper toolbar, select **Save** > **Save As**.
2. In the Save As dialog box, browse to a writable folder, specify a scenario file name, and then click **Save**.

   - To save the signals and the scenario file, click **Yes**.

     If a MAT-file is already associated with the scenario, the tool appends the base workspace variables to this file.

Save a scenario to an existing file (the file from which the scenario was last loaded):

1. On the Root Inport Mapper tool toolbar, click **Save**.
2. Browse to the `.mldatx` file in which to save the scenario, and then click **Save**.

   If you have not saved the signals from the scenario, the tool prompts you to save the signals to a MAT-file.

Goal	Action
Overwrite the existing `.mldatx` file.	**Yes**
Exit the dialog box. The tool does not save the scenario.	**No**

Open Existing Scenarios

You can open previously saved scenario files in one of the following ways:

- Double-click the previously saved scenario file (*.`mldatx`). The Root Inport Mapper tool opens and loads the model. Alternatively, right-click the file and select **Open**.
  - When loading scenario files, the tool looks for the associated model and MAT-file or Microsoft Excel file in the last known location, and then looks on the MATLAB path. If the tool cannot find the model or signal files in these two locations, an error occurs.
  - If the previously saved scenario has mapped signals, when you open the scenario, the tool applies the mapping. Also, the tool adds the signals to the base workspace so that you can simulate the model.
- Open the Root Inport Mapper tool for the model, click **Open**, and select the previously saved scenario file.

If the model is already open, the new scenario overwrites the existing scenario for the model. If there are unsaved changes in the open scenario, respond to the prompt.

Goal	Click
Save the existing scenario and associated data before loading the new scenario.	**Yes**

Goal	Click
Open the new scenario without saving the existing scenario. This option also removes the data in the existing scenario.	**No**

Work with Multiple Scenarios

You can open and work with multiple scenario files simultaneously. Working with multiple scenario files lets you view, edit, group, and nest multiple scenario files. Use multiple scenario files to test more complex systems with interrelated components.

As you open each multiple scenario, the Root Inport Mapper tool adds its data set to the SCENARIO DATASET section. If the scenario contains only signals, convert the signals to the `Simulink.SimulationData.Dataset` format:

1. To convert the signals to a `Simulink.SimulationData.Dataset` format, use the `convertToSLDataset` function.
2. Link to the new data set. You do not need to reopen the scenario.

See Also

Related Examples

- “Import Signal Data for Root Inport Mapping” on page 71-14
- “Create and Edit Signal Data” on page 69-2
- “Map Signal Data to Root Input Ports” on page 71-18

More About

- “Create Signal Data for Root Inport Mapping” on page 71-9
- “Map Root Inport Signal Data” on page 71-7

Load Data with Interchangeable Scenarios

To easily exchange scenarios within models, use the Signal Editor block. This block displays, creates, edits, and switches scenarios, where scenarios contain information about groups of signals, such as:

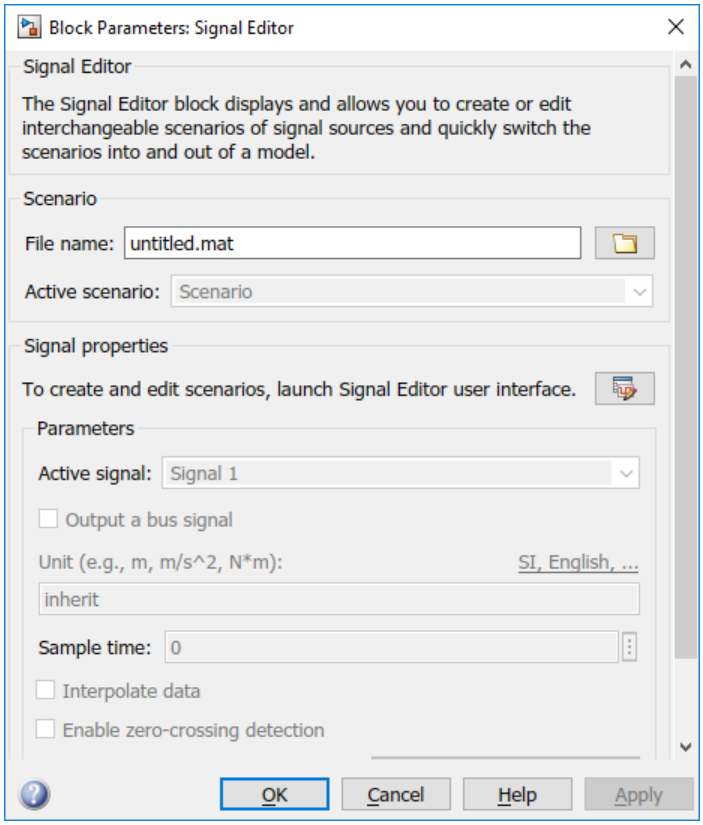
- Location of signal files (MAT-file or Microsoft Excel files)
- Location of the associated model
- Map mode
- Mapping options
- Mapped state

Use scenarios to exchange groups of signals in your model, such as when running multiple simulations or working with test harnesses.

Tip The Signal Editor displays, creates, and edits interchangeable scenarios and is better integrated with other Simulink capabilities such as units, signals in MAT-files, and signal edit and creation. Consider using the Signal Editor block in models where you use the Signal Builder block. The Signal Builder block is not recommended to work with signal groups. For more information on benefits of the Signal Editor block, see “Replace Signal Builder Block with Signal Editor Block” on page 71-39.

Load Data with Interchangeable Scenarios Using Signal Editor Block

The Signal Editor block gets groups of signals (scenarios) from MAT-files.



The block has three sections.

- Scenario — Set up a scenario by specifying the MAT-file containing the list of scenarios and selecting the active (current) scenario.
- Signal properties — Use the Signal Editor user interface to examine, create, and edit scenarios and associated signals. The Signal Editor user interface organizes the signals according to the scenarios that contain them.
- Parameters — Select the active signal to be output and set up characteristics, such as unit, sample time, linear interpolation, and so forth, for the active signal. To change the active signal, use the drop-down list.

Explore the Signal Editor Block

This example shows how to use the Signal Editor block with the “Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode” example, which runs multiple simulations of a Monte Carlo study in parallel using Parallel Computing Toolbox. Parallel execution leverages the multiple cores of your host machine to run many simulations more quickly. If you do not have Parallel Computing Toolbox, this example runs the simulations in serial. The model simulates vehicle dynamics based on the interaction between road and suspension for different road profiles. This example stores its road profile scenarios in the `matlab\toolbox\ssldemo_suspn_3dof_sigData.mat` file.

1. In the `sldemo_suspn_3dof` model, open the Signal Editor (named Road Profiles) block.

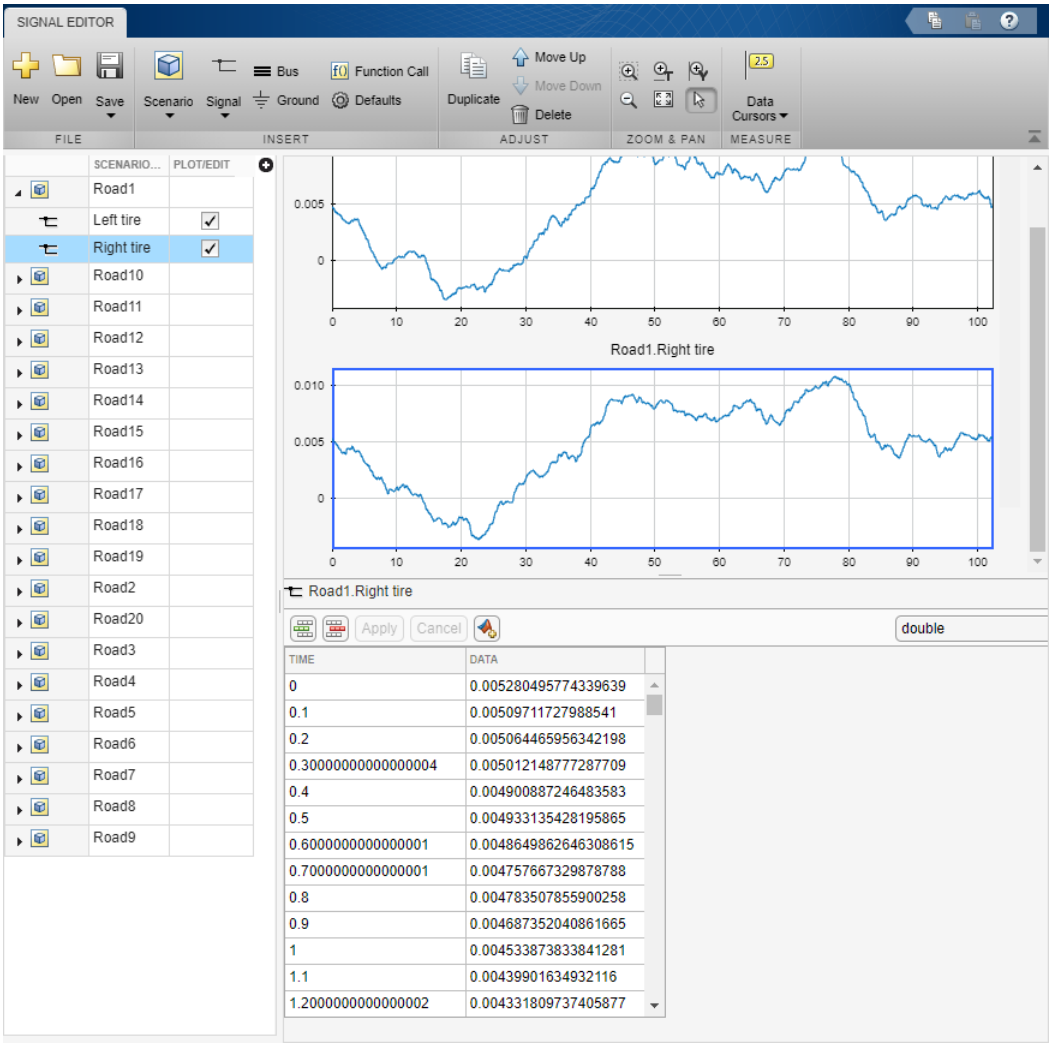
   The Signal Editor block **File Name** parameter contains the MAT-file `sldemo_suspn_3dof_sigData.mat`.

   Observe that:

   - **Active scenario** automatically sets to the first scenario, `Road1`. This setting means that the remainder of the block parameters apply to the signals in that scenario. To change the active scenario, select a new scenario from the list of scenarios.
   - **Active signal** automatically sets to the first signal `Left tire` in the active scenario, `Road1`. This setting means that the remainder of the settings in the **Parameter** section apply to the active signal. To change the active signal, select a new signal from the list of signals.

2. Explore and edit the scenarios in the MAT-file. Click  .

   The Signal Editor user interface displays a list of scenarios contained in the MAT-file. Explore and change the scenarios and associated signals. If you make changes, you can save them back to the MAT-file by clicking **Save**.



The output from the block is the active signal data. To simulate the model sequentially using each scenario and signal, use the `parsim` function. This function simulates dynamic systems multiple times in parallel or serial. For an example on how to use `parsim`, see “Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode”.

Replace Signal Builder Block with Signal Editor Block

The Signal Editor provides similar functionality to the Signal Builder block, but with greater flexibility. Replace the Signal Builder with the Signal Editor block. The benefits of Signal Editor block include:

- Signal data storage in a MAT-file outside the model
- Signal editing and creation
- Support for Simulink signal attributes, such as dimension and complexity
- Support for standard Simulink data types, including bus and fixed-point
- Unique signal-level data types for outputs

- Multiple rates for outputs
- Support for Simulink units
- Zero crossing on page 3-10 detection and data interpolation unique to each signal

To port signal data and properties from the Signal Builder block to Signal Editor block, use the `signalBuilderToSignalEditor` function. For the current model, this function stores to a MAT-file the signal data and properties from an existing Signal Builder block, adds a Signal Editor block to the current model, and modifies the Signal Editor block to reference the new MAT-file.

For an example see “Replace Signal Builder Block with Signal Editor Block”.

Considerations

Converting from the Signal Builder block to the Signal Editor block is relatively straightforward using the `signalBuilderToSignalEditor` function. However, take into account these considerations:

- Internal storage format and preprocessing of data differs between the Signal Builder and Signal Editor blocks. When using the variable step solver, different simulation time steps and mismatched output occur in the two blocks. To minimize the difference between the outputs of both blocks, you can:
  - Reduce the value of **Max step size** of the variable step solver.
  - Insert more data points in the input signal of the Signal Editor block to better represent its shape.
  - Use the fixed-step solver or set the sample time for both blocks to the same discrete sample time (greater than 0). For more information on discrete sample times, see “Discrete Sample Time” on page 7-13.
- The Signal Builder block supports only doubles. To change the data type or otherwise change the signals after conversion, click the  button in the Signal Editor block to access the Signal Editor user interface.

Get Number of Scenarios and Signals

To programmatically get the total number of scenarios and signals in the Signal Editor block, use the `get_param` `NumberOfScenarios` and `NumberOfSignals` properties. The values of these properties are character vectors. To convert these values to doubles, use the `str2double` function.

`NumberOfScenarios` and `NumberOfSignals` are read-only properties available only through `get_param`. The block dialog box does not provide these values.

See Also

Signal Editor | `parsim` | `signalBuilderToSignalEditor`

More About

- “Create and Edit Signal Data” on page 69-2
- “Signal Groups” on page 75-50

Importing and Exporting Simulation Data

Export Simulation Data

In this section...

Exporting (logging) simulation data provides a baseline for analyzing and debugging a model. Use standard or custom MATLAB functions to generate simulated system input signals and to graph, analyze, or otherwise postprocess the system outputs.

Simulation Data

Simulation data can include any combination of signal, time, output, state, and data store logging data.

Exporting simulation data involves saving signal values to the MATLAB workspace or to a MAT-file during simulation for later retrieval and postprocessing. Exporting data is also known as “data logging” or “saving simulation data.”

You can have data logged in several formats:

- `Simulink.SimulationData.Dataset`
- Array
- Structure
- Structure with time
- MATLAB timeseries
- `ModelDataLogs`

> **Note** The `ModelDataLogs` format is supported for backward compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model logs data in `Dataset` format.

Consider converting data logged in other formats to `Dataset` format to simplify post-processing. For more information, see “Dataset Conversion for Logged Data” on page 72-12.

You can also use exported data as the input for simulating a model.

Approaches for Exporting Signal Data

Exporting simulation data often involves exporting signal data. You can use various approaches for exporting signal data.

Export Approach	Usage	Documentation
Connect a Scope block to a signal.	If you use a Scope block for viewing results during simulation, consider also using the Scope block to export data. Save output at a sample rate other than the base sample rate. Scopes store data and can be memory intensive.	Scope
Connect a signal to a To File block.	Consider using a To File block for exporting large amounts of data. Save output at a sample rate other than the base sample rate. Use the MAT-file only after the simulation has completed.	To File
Connect a signal to a To Workspace block.	Document in the diagram the workspace variables used to store signal data. Save output at a sample rate other than the base sample rate.	To Workspace
Connect a signal to a root-level Outport block.	Consider using this approach for logging data in a top-level model, if the model already includes an Outport block.	Outport
Set the signal logging properties for a signal.	Use signal logging to avoid adding blocks, such as Scope, To File, and To Workspace blocks, to your model. Log signals based on individual signal rates. Data is available when simulation is paused or completed. Use signal logging to log array of buses signals.	“Export Signal Data Using Signal Logging” on page 72-41
Configure Simulink to export time, state, and output data.	To capture complete information about the simulation as a whole, consider exporting this data. Use the **Output** parameter to save root Outport block data during simulation. Outputs and states are logged at the base sample rate of the model.	“Data Format for Logged Simulation Data” on page 72-7 “Limit Amount of Exported Data” on page 72-26 “Samples to Export for Variable-Step Solvers” on page 72-38

Export Approach	Usage	Documentation
Log a data store.	Log a data store to share data throughout a model hierarchy, capturing the order of all data store writes.	“Log Data Stores” on page 73-30
Use the `sim` command to log simulation data programmatically.	Use `sim` to export the time, states, and signal simulation data to one data object. Select the **Return as single object** parameter when simulating the model using the `sim` command inside a function or a `parfor` loop.	`sim`

Enable Simulation Data Export

To export the states and root-level output ports of a model to the MATLAB base workspace during simulation of the model, use one of these interfaces:

- **Configuration Settings > Data Import/Export** pane (for details, see “Model Configuration Parameters: Data Import/Export”)
- `sim` command

In both approaches, specify:

- The kinds of simulation data that you want to export:
  - Signal logging
  - Time
  - Output
  - State or final state
  - Data store

  Each kind of simulation data export has an associated default variable. You can specify your own variables for the exported data.
- The characteristics of the logged data, including:
  - “Data Format for Logged Simulation Data” on page 72-7
  - “Limit Data Logged” on page 72-46
  - “Samples to Export for Variable-Step Solvers” on page 72-38

View Logged Data Using Simulation Data Inspector

To inspect exported simulation data interactively, consider using the **Simulation Data Inspector**.

The Simulation Data Inspector has some limitations on the kinds of logged data that it displays. See “View Data in the Simulation Data Inspector” on page 29-2.

Memory Performance

Optimization for Logged Data

When exporting simulation data in a simulation mode other than rapid accelerator, Simulink optimizes memory usage in the following situations.

- When time steps happen at regular intervals, Simulink uses compressed time representation. Simulink stores the value for the first timestamp, the length of the interval (time step), and the total number of timestamps.
- When multiple signals use identical timestamp sequences, the signals share a single stored timestamp sequence. Sharing a single stored timestamp can reduce memory use for logged data by as much as a factor of two. The difference in memory performance can be a critical performance factor, particularly when logging bus signals that have thousands of bus elements.

Logging to Persistent Storage

You can encounter memory issues when you log many signals in a long simulation that has many time steps. Logging to persistent storage can address this kind of memory issue.

To log to persistent storage, in the **Configuration Parameters > Data Import/Export** pane, select **Log Dataset data to file** option. Specify the kinds of logging (for example, signal logging and states logging).

- For logging output and states data, set the **Format** parameter to `Dataset`.
- If you select the **Final states** parameter, clear the **Save final operating point** parameter.

Using a `Simulink.SimulationData.DatasetRef` object to access signal logging and states logging data loads data into the model workspace incrementally. Accessing data for other kinds of logging loads all the data at once.

For details, see “Log Data to Persistent Storage” on page 72-31.

See Also

Blocks
Outport | Scope | To File | To Workspace

Functions
`sim`

Related Examples

- “Load Data Logged In Another Simulation” on page 70-27
- “Data Format for Logged Simulation Data” on page 72-7
- “Limit Amount of Exported Data” on page 72-26
- “Log Data to Persistent Storage” on page 72-31

More About

- “Overview of Signal Loading Techniques” on page 70-15

- “Comparison of Signal Loading Techniques” on page 70-21
- “Data Format for Logged Simulation Data” on page 72-7
- “Signal Data Storage for Loading” on page 70-2
- Simulation Data Inspector on page 29-144

Data Format for Logged Simulation Data

In this section...

Data Format for Block-Based Logged Data

You can use the Scope, To File, or To Workspace blocks to export simulation data. Each of these blocks has a data format parameter.

Data Format for Model-Based Logged Data

The data format for model-based exporting of simulation data specifies how Simulink stores the exported data.

Simulink uses different data formats, depending on the kind of data that you export. For details, see:

- “Signal Logging Format” on page 72-7
- “Logged Data Store Format” on page 72-7
- “Time, State, and Output Data Format” on page 72-7

Signal Logging Format

Signal logging always uses `Dataset` format. You can specify whether to log data for individual signals as `timeseries` or `timetable` objects.

To control how `Dataset` elements are saved, set the **Dataset signal format** configuration parameter. The default is `timeseries`. For details, see “Dataset signal format”.

The **Dataset signal format** parameter applies to signal logging, as well as output and states data when you set the **Format** parameter to `Dataset`.

Logged Data Store Format

When you log data store data, Simulink uses a `Simulink.SimulationData.Dataset` object.

For details, see “Accessing Data Store Logging Data” on page 73-32.

Time, State, and Output Data Format

For exported time, states, and output data, use one of the following formats:

- “Dataset” on page 72-8 (default)

- “Array” on page 72-8
- “Structure with Time” on page 72-9
- “Structure” on page 72-10

If you select the **Configuration Parameters > Data Import/Export > Output** check box, Simulink logs fixed-point data as double. To log fixed-point data, consider using one of these approaches:

- Signal logging — For details, see “Export Signal Data Using Signal Logging” on page 72-41.
  1. In the Simulink Editor, select one or more signals.
  2. Click **Log Signals**.
- To File block
- To Workspace block — In the To Workspace Block Parameters dialog box, enable the **Log fixed-point data as a fi object** parameter.

For information about the format for logged final state data, see “State Information” on page 72-76.

Dataset

The default setting for the **Format** parameter is `Dataset`. The `Dataset` format:

- Stores logged data in `timeseries` or `timetable` objects. You can work with data saved in a `timeseries` or `timetable` object in MATLAB without a Simulink license.
- Supports logging multiple data values for a given time step, which can be required for logging data in a For Iterator Subsystem, a While Iterator Subsystem, and Stateflow.
- Does not support rapid accelerator simulation, logging states information inside a function-call subsystem, or code generation.

Signal logging always uses the `Dataset` format. Logging states and output data using the `Dataset` format allows you to post-process simulation data without writing custom code for different types of logged data. When you log states and outputs using the `Dataset` format, the data also automatically streams to the Simulation Data Inspector during simulation.

Array

If you select this `Array` option, Simulink saves the states and outputs of a model in a state and in an output array, respectively.

The state matrix has the name specified in the **Configuration Parameters > Data Import/Export** pane (for example, `xout`). Each row of the state matrix corresponds to a time sample of the states of a model. Each column corresponds to an element of a state. For example, suppose that your model has two continuous states, each of which is a two-element vector. Then the first two elements of each row of the state matrix contain a time sample of the first state vector. The last two elements of each row contain a time sample of the second state vector.

The model output matrix has the name specified in the **Configuration Parameters > Data Import/Export** pane (for example, `yout`). Each column corresponds to a model output port, and each row to the outputs at a specific time.

Note Use array format to save your model outputs and states only if the logged data meets *all* these conditions:

- Data is all scalars or all vectors (or all matrices for states)
- Data is all real or all complex
- Data all has the same data type
- Data includes bus signals

If your model outputs and states do not meet these conditions, use the `Structure` or `Structure with time` output formats (see “Structure with Time” on page 72-9).

Structure with Time

If you select this format, Simulink saves the model states and outputs in structures that have their names specified in the **Configuration Parameters** > **Data Import/Export** pane. By default, the structures are `xout` for states and `yout` for output.

The structure used to save outputs has two top-level fields:

- `time`

  Contains a vector of the simulation times.
- `signals`

  Contains an array of substructures, each of which corresponds to a model output port.

Each substructure has four fields:

- `values`

  Contains the outputs for the corresponding output port.

  - If outputs are scalars or vectors — `values` field is a matrix each of whose rows represent an output at the time specified by the corresponding time vector element.
  - If the outputs are matrix (2-D) values — `values` field is a 3-D array of dimensions M-by-N-by-T. M-by-N is the dimensions of the output signal and T is the number of output samples.
  - If T = 1 — MATLAB drops the last dimension. Therefore, the `values` field is an M-by-N matrix.
- `dimensions`

  Specifies the dimensions of the output signal.
- `label`

  Specifies the label of the signal connected to the output port, S-Function block, or the type of state (continuous or discrete). The label is `DSTATE` or `CSTATE`, except for S-Function block state labels. For S-Function block state labels for discrete states, the label is the name of the state (instead of `DSTATE`).
- `blockName`

  Specifies the name of the corresponding output port or block with states.
- `inReferencedModel`

  If the `signals` field records the final state of a block that resides in the referenced model, contains a value of `1`. Otherwise, the value is false (`0`).

The following example shows the structure-with-time format for a nonreferenced model.

```
xout.signals(1)

ans =

               values: [296206x1 double]
           dimensions: 1
                label: 'CSTATE'
            blockName: 'vdp/x1'
    inReferencedModel: 0
```

The structure used to save states has a similar organization. The states structure has two top-level fields:

- `time`

  The `time` field contains a vector of the simulation times.

- `signals`

  The field contains an array of substructures, each of which corresponds to one of the states of the model.

Each `signals` structure has four fields: `values`, `dimensions`, `label`, and `blockName`. The `values` field contains time samples of a state of the block specified by the `blockName` field. The `label` field for built-in blocks indicates the type of state: either `CSTATE` (continuous state) or `DSTATE` (discrete state). For S-Function blocks, the label contains whatever name is assigned to the state by the S-Function block.

The time samples of a state are stored in the `values` field as a matrix of values. Each row corresponds to a time sample. Each element of a row corresponds to an element of the state. If the state is a matrix, the matrix is stored in the `values` array in column-major order. For example, suppose that the model includes a 2-by-2 matrix state and that 51 samples of the state are logged during a simulation run.

The `values` field for this state would contain a 51-by-4 matrix. Each row corresponds to a time sample of the state, and the first two elements of each row correspond to the first column of the sample. The last two elements correspond to the second column of the sample.

Note Simulink can read back simulation data saved to the MATLAB workspace in the `Structure with time` output format. See “Examples of Specifying Signal and Time Data” on page 70-41 for more information.

Structure

This format is the same as for `Structure with time` output format, except that Simulink does not store simulation times in the `time` field of the saved structure.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.forEachTimeseries`

Related Examples

- “Export Simulation Data” on page 72-2
- “Log Signal Data That Uses Units” on page 72-24
- “Load Data to Root-Level Input Ports” on page 70-35

More About

- “Comparison of Signal Loading Techniques” on page 70-21
- “Dataset Conversion for Logged Data” on page 72-12
- “Map Root Inport Signal Data” on page 71-7
- “State Information” on page 72-76

Dataset Conversion for Logged Data

In this section...

Why Convert to Dataset Format?

You can use the `Simulink.SimulationData.Dataset` constructor to convert a MATLAB workspace variable that contains data that was logged in one of these formats to `Dataset` format:

- Array
- Structure
- Structure with time
- MATLAB timeseries
- `ModelDataLogs`

Converting data from other Simulink logging formats to `Dataset` format simplifies writing scripts to post-process data logged. For example, a model with multiple To Workspace blocks can use different data formats. Converting the logged data to `Dataset` format avoids the need to write special code to handle different formats.

Different simulation modes have different levels of support for data logging formats. Switching between normal and accelerator modes can require changes to the logging formats used.

The conversion to `Dataset` format also makes it easier to take advantage of features that require `Dataset` format. You can easily convert data logged in earlier releases that used a format other than `Dataset` to work well with `Dataset` data in a more recent release.

The `Dataset` format:

- Uses MATLAB `timeseries` objects to store logged data, which allows you to work with logging data in MATLAB without a Simulink license. For example, to manipulate the logged data, you can use MATLAB time-series methods such as `filter`, `detrend`, and `resample`.
- Supports logging multiple data values for a given time step, which is important for Iterator subsystem and Stateflow signal logging.

By default, the resulting `Dataset` object uses the variable name as its name. You can use a name-value pair to specify a `Dataset` name.

You can use the `concat` method to combine `Dataset` objects into one concatenated `Dataset` object.

Results of Conversion

`Dataset` objects hold data as elements. To display the elements of a `Dataset` variable, enter the variable name at the MATLAB command prompt. The elements of `Dataset` objects are different types, depending on the data they store. For example, signal logging stores data as `Simulink.SimulationData.Signal` elements and state logging in `Dataset` format stores data as

`Simulink.SimulationData.State` elements. Each element holds data as a MATLAB time-series object. At conversion, the elements and time-series field populate as much as possible from the converted object.

Format	Conversion Result Notes
MATLAB time series	If you log nonbus data, during conversion, the software first adds the data as a `Simulink.SimulationData.Signal` object. It then adds that object as an element of the newly created `Dataset`. If you log bus data in time-series format, one time series corresponds to each element of a bus. Converting arranges the logged data as a structure with time-series objects as leaf nodes. This structure hierarchy matches the bus hierarchy. Conversion of this type of structure of time-series objects adds the whole structure to a `Simulink.SimulationData.Signal` object. It then adds that object as an element of the data set. Time-series objects hold relevant information such as block path and timestamps. The conversion tries to preserve this information.
Structure and structure with time	Structure and structure with time formats do not always contain as much information as if you log in `Dataset` format. However, before converting structure and structure with time formats, the data structure must have `time` and `signals` fields. Conversion populates a `Simulink.SimulationData.Signal` object with the structure and adds it as an element of the data set. If other information is available, converting also adds it to the element or time-series values. For example, if the structure has a field called `blockName`, converting adds it to the block path. Otherwise, the block path is empty. When scope data is logged in structure format, the logged structure has a `PlotStyle` field. The software uses this field to set the interpolation in the `Dataset` object.
Array	Arrays contain little information. For example, there is no block path information. Conversion adds the array to a `Simulink.SimulationData.Signal` object and adds it as an element of the `Dataset` object. The conversion leaves unavailable information such as block path and timestamp fields as either empty or with default values.

Format	Conversion Result Notes
`ModelDataLogs`	Converts data from `ModelDataLogs` format to `Dataset` format. **Note** The `ModelDataLogs` format is no longer used for signal logging.

Dataset Conversion Limitations

- Converting logged data to `Dataset` format results in a `Dataset` object that contains all the information that the original logged data included. However, if there is no corresponding information for the other `Dataset` properties, the conversion uses default values for that information.
- To log variable-size signals, use the To Workspace block. If you convert data logged with To Workspace to be `Dataset` format, you lose the information about the variable-size signals.
- When you log a bus signal in array, structure, or structure with time formats, the logged data is organized with:

  - The first column containing the data for the first signal in the bus
  - The second column containing data for the second bus signal, and so on

  When you convert that data to `Dataset`, the `Dataset` preserves that organization. But if you log the bus signal in `Dataset` format without conversion, the conversion captures the bus data as a structure of time-series objects.
- If the logged data does not include a time vector, when you convert that data to `Dataset`, the conversion inserts a time vector. There is one time step for each data value. However, the simulation time steps and the `Dataset` time steps can vary.
- `Dataset` format ignores the specification of frame signals. Conversion of structure or structure with time data to `Dataset` reshapes the data for logged frame signals.

See Also

`Simulink.SimulationData.Dataset`

Related Examples

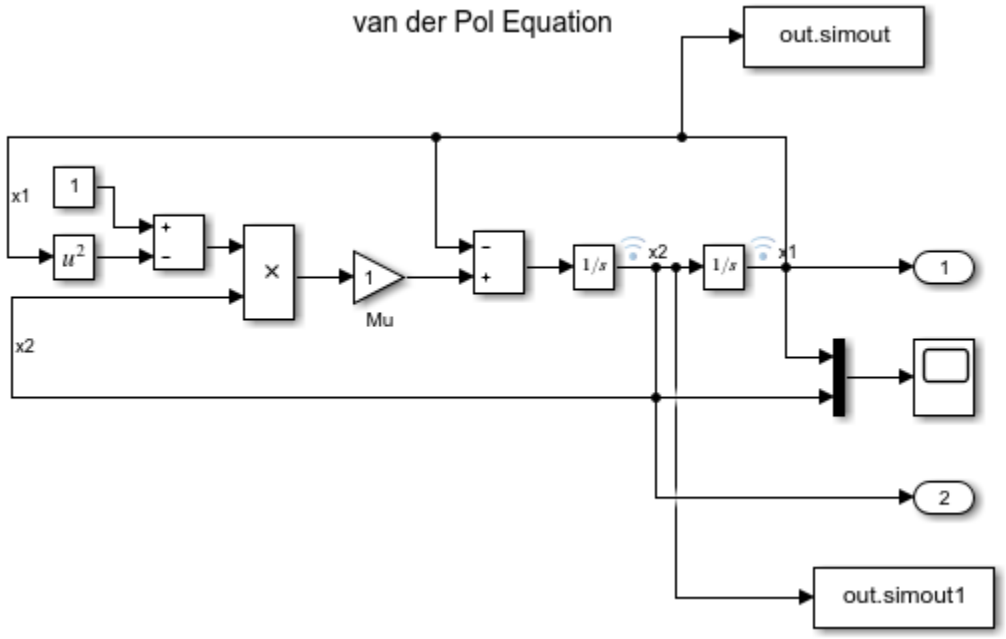
- “Convert Logged Data to Dataset Format” on page 72-15

Convert Logged Data to Dataset Format

In this section...

Convert Workspace Data to Dataset

This example shows how to convert MATLAB time-series data to `Dataset` format. `myvdp_timeseries` is the `vdp` model with two To Workspace blocks configured for *simout* and *simout1* logging data in MATLAB timeseries format. Consider using a procedure like this one if you have models that use To Workspace blocks to log data to MATLAB timeseries format.



Use the `Simulink.SimulationData.Dataset` constructor to convert the MATLAB timeseries data to `Dataset` format and then concatenate the two data sets.

1. Starting with the `vdp` model, add two To Workspace blocks to the model as shown.
2. Set the **Save format** parameter of both blocks. Set `Timeseries`.
3. Save the model as `myvdp_timeseries`.
4. Simulate the model.

   The simulation logs data using the To Workspace blocks.
5. Access the signal logging format, `logsout`.

   ```
   logsout
   ```

   ```
   logsout =
   ```

```
  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'logsout'
    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
```

6 Convert the MATLAB time-series data from both To Workspace blocks to `Dataset`.

```
ds = Simulink.SimulationData.Dataset(simout);
ds1 = Simulink.SimulationData.Dataset(simout1);
```

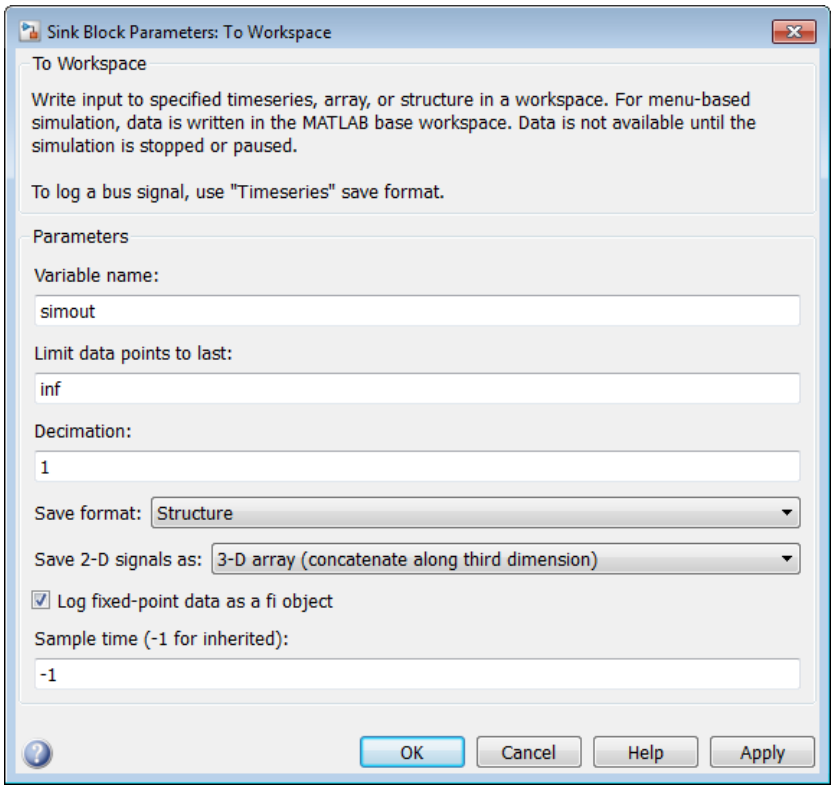
ds is the variable name of the first To Workspace block data. *ds1* is the variable name of the second To Workspace block data.

7 Concatenate both datasets into *dsfinal*. Observe that the format of *dsfinal* matches that of `logsout`.

```
dsfinal = ds.concat(ds1)

dsfinal =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'simout'
    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
```

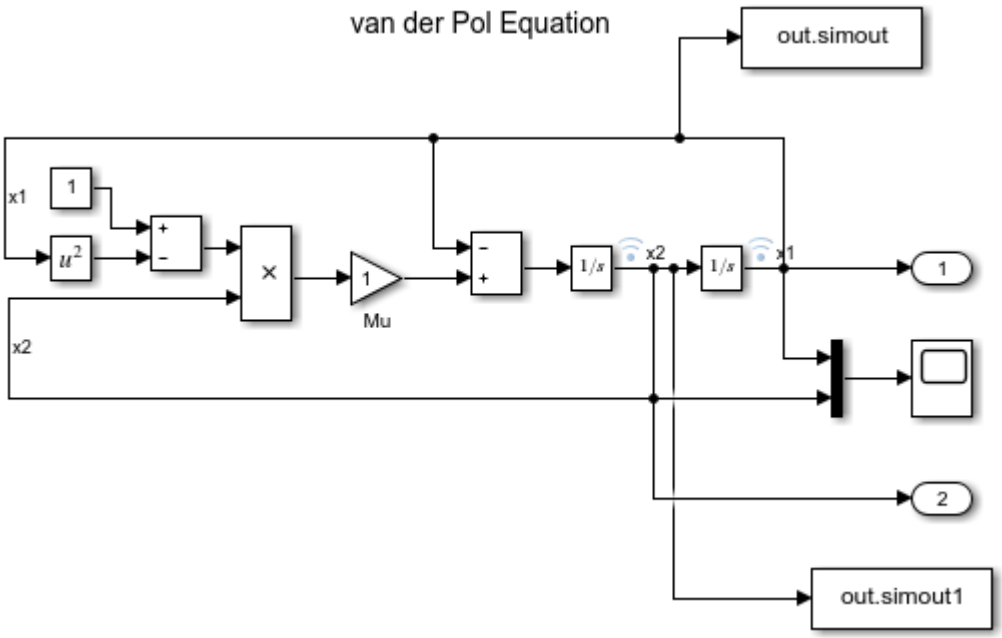
Convert Structure Without Time to Dataset

This example shows how to convert structure without time data to `Dataset` format. `myvdp_structure` is the `vdp` model with two To Workspace blocks configured for simout and simout1 logging data in structure format, as shown.



If you have models that use To Workspace blocks to log data to structure format, consider using a procedure like this one to convert them to `Dataset` format.

1 Starting with the `vdp` model, add two To Workspace blocks to the model as shown.



2 In the **Save format** parameter of both blocks, select `Structure`.

3. Enable signal logging for the two signals going to the two To Workspace blocks to log in `Ds` format.
4. Save the model as `myvdp_structure`.
5. Simulate the model.

   The simulation logs data using the To Workspace blocks.
6. Convert the structure data from both To Workspace blocks to `Dataset`.

   ```
   ds = Simulink.SimulationData.Dataset(simout);
   ds1 = Simulink.SimulationData.Dataset(simout1);
   ```

   simout is the variable name of the first To Workspace block data. *simout1* is the variable name of the second To Workspace block data.

   With the conversion of structure without time or an array, time starts at t=0 and increments by 1.
7. Get the values of the first element in *ds*.

   ```
   ds.get(1).Values.Time
   ```

   ```
   ans =

        0
        1
        2
        3
        .
        .
        .
       61
       62
       63
   ```
8. Get the time values of the first element from signal logging.

   ```
   logsout.get(1).Values.Time
   ```

   ```
   ans =

             0
        0.0001
        0.0006
        0.0031
        .
        .
        .
       19.2802
       19.6802
       20.0000
   ```
9. Observe the discrepancy in timestamps between

   - Data logged in structure without time that you convert to `Dataset` format
   - Data logged in `Dataset` format

Programmatically Access Logged Dataset Format Data

When you use the default `Dataset` signal logging format, Simulink saves the logging data in a `Simulink.SimulationData.Dataset` object. For information about extracting signal data from that object, see the `Simulink.SimulationData.Dataset` reference page.

The `Simulink.SimulationData.Dataset` object contains a `Simulink.SimulationData.Signal` object for each logged signal.

For bus signals, the `Simulink.SimulationData.Signal` object contains a structure of MATLAB `timeseries` objects.

The `Simulink.SimulationData.Dataset` class provides two methods for accessing the signal logging data and its associated information.

Name	Description
`get` You can also use the `getElement` method, which shares syntax and behavior as the `get` method.	Get element or collection of elements from the dataset, based on index, name, or block path.
`numElements`	Get number of elements in the dataset.

For example of accessing signal logging data that uses the `Dataset` format, see `Simulink.SimulationData.Dataset`.

Access Array of Buses Signal Logging Data

Signal logging data for an array of buses uses `Dataset` signal logging format.

The general approach to access data for a specific signal in an array of buses is:

1. Use a `Simulink.SimulationData.Dataset.get` (or `getElement`) method to access a specific signal in the logged data (by default, the `logsout` variable).
2. To get the values, index within the array of buses.
3. Index again to get data for a specific bus.

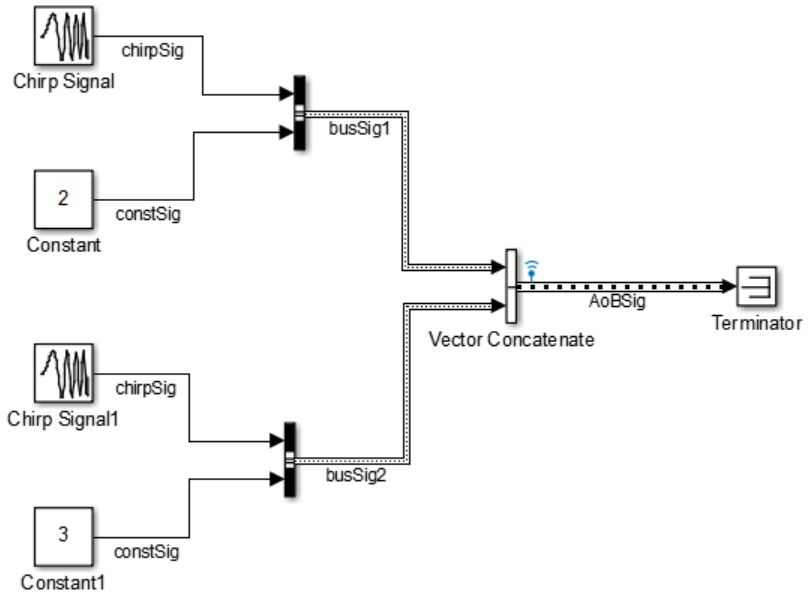
For example, to obtain the signal logging data for the Constant6 block in the `ex_log_nested_aob` model, for the `topBus` signal that feeds the Terminator block:

```
logsout.getElement('topBus').Values.a(2,2).firstConst.data
```

Here are additional examples of accessing array of buses signal logging data. For another example that shows how to log array of buses data, see `sldemo_mdlref_bus`.

Simple Array of Buses

The `ex_log_simple_aob` model includes an array of buses signal `AoBSig` that combines two bus signals (`busSig1` and `busSig2`).



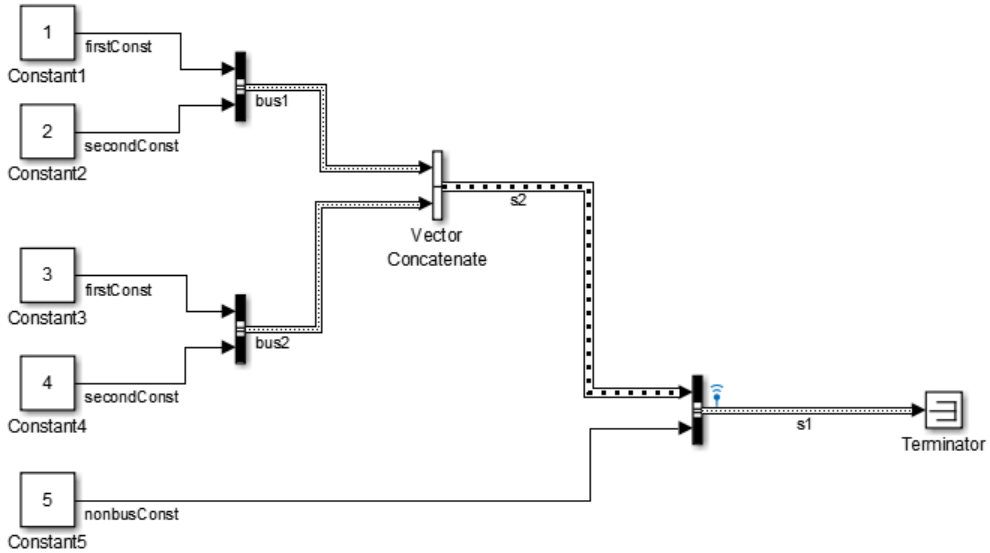
To access the signal logging data for the array of buses signal, navigate through the structure hierarchy and use an index to access a specific node. This example shows navigation to the `chirpSig` signal value in `busSig2`.

```
logsout.getElement('AoBSig').Values(2).chirpSig.Data
```

```
ans=

   0
 0.9585
```

Array of Buses in a Bus

The `ex_log_aob_in_bus` model has an array of buses (`s2`) that feeds into bus `s1`.



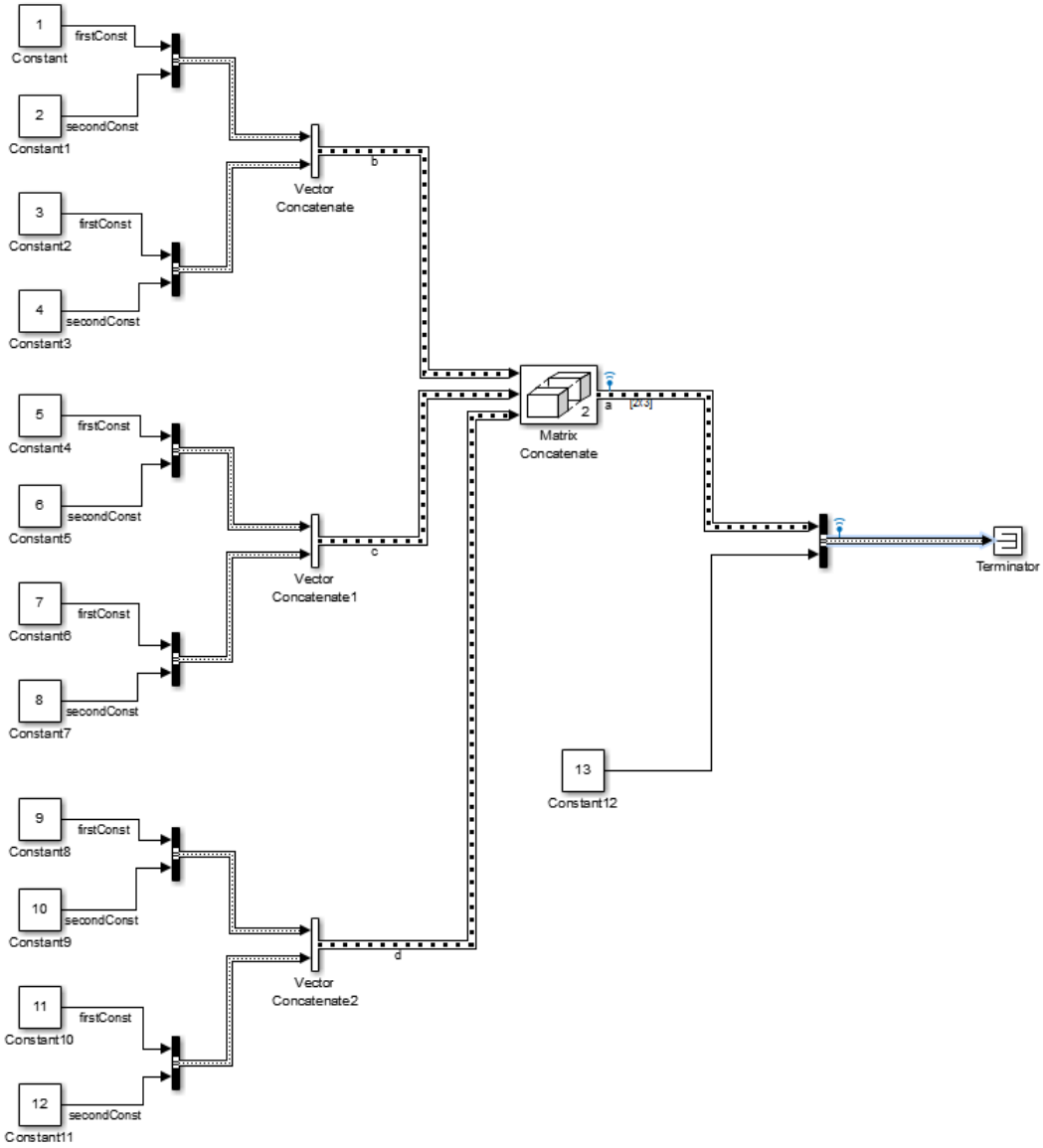
This example shows navigation to the `Constant3` block, which is a signal in `bus2`.

```
logsout.getElement('s1').Values.s2(2).firstConst.Data
```

```
ans=

  3
  3
  3
  3
  3
  3
```

Nested Arrays of Buses

The `ex_log_nested_aob` model has an array of buses (`a`) that is made up of three arrays of buses: `b`, `c`, and `d`. The Matrix Concatenate block combines the nested arrays of buses into array of buses `a`.



This example shows navigation to the `Constant6` block.

```
logsout.getElement('topBus').Values.a(2,2).firstConst.Data
```

```
ans=

  7
  7
  7
  7
  7
  7
  7
  7
  7
```

```
7
7
```

Accessing Data for Signals with a Duplicate Name

For a model with multiple signals that have the same signal name, signal logging data includes a `Simulink.SimulationData.Signal` object for each signal that has a duplicate name.

To access a specific signal that has a duplicate name, use *one* of these approaches:

- To find the data for the specific signal, visually inspect the displayed output of `Simulink.SimulationData.Signal` objects.
- Use the `Simulink.SimulationData.Dataset.getElement` method, specifying the block path for the source block of the signal.
- To iterate through the signals with a duplicate signal name, create a script using the `Simulink.SimulationData.Dataset.getElement` method with an index argument.
- Use the Signal Properties dialog box to specify a different name. Consider using this approach when the signals with a duplicate name do not appear in multiple instances of a referenced model in normal mode.

  1. In the model, right-click the signal.
  2. In the context menu, select **Properties**.
  3. In the Signal Properties dialog box, set **Logging name** to `Custom` and specify a different name than the signal name.
  4. Simulate the model and use the `Simulink.SimulationData.Dataset.getElement` method with a name argument.

Tip Alternatively, you can use the Signal Logging Selector to access a specific signal. For details, see “Override Signal Logging Settings with Signal Logging Selector” on page 72-58.

Handling Newline Characters in Signal Logging Data

To handle newline characters in logging names in signal logging data that uses `Dataset` format, use a `sprintf` command within a `getElement` call. For example:

```
topOut.getElement(sprintf('INCREMENT\nBUS'))
```

See Also

`Simulink.SimulationData.Dataset`

More About

- “Dataset Conversion for Logged Data” on page 72-12

Log Signal Data That Uses Units

To have logged data include the units specified for signals, use the `Dataset` or `Timeseries` logging format, which stores logging information in MATLAB `timeseries` objects.

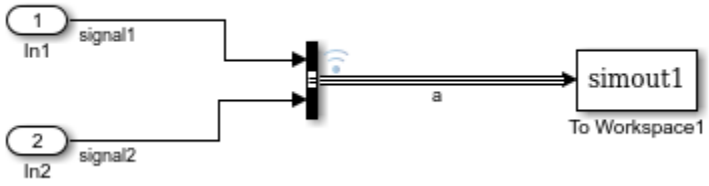
Signal logging uses `Dataset` format. Output logging (**Configuration Parameters** > **Data Import/Export** > **Output**) uses `Dataset` as the default format. The default save format for the To File and To Workspace blocks is `Timeseries`.

If you use `Dataset` or `Timeseries` format for signal logging or for To File block or To Workspace block logging, the logged data includes units information.

To capture units information for output logging:

1. Set the **Format** configuration parameter to `Dataset`.
2. In the Block Parameters dialog box for Outport blocks for which you want to capture units information, set the **Unit** parameter to match the units of the input signal.

For example, in this model the `In1` block has its **Unit** parameter set to `newton` and `In2` block uses `m` (meters). Open the model. After you simulate the model, you can see the units for the logged data.



- You can view the units in the signal logging data for `signal1` of the bus signal `b`.

```
logsout.get('a').Values.signal1.DataInfo
```

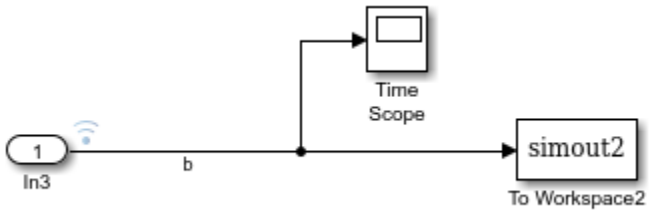
```
tsdata.datametadata
  Package: tsdata

  Common Properties:
              Units: newton (Simulink.SimulationData.Unit)
      Interpolation: linear (tsdata.interpolation)
```

- You can view the units in the data logged in the To Workspace block.

```
 simout1.signal2.DataInfo.Units
```

```
ans =

  Units with properties:

    Name: 'm'
```

This example model shows how to view the data logged in a Time Scope block. Open the model.



To use the Time Scope block to log data, in the scope select **Configuration Properties** > **Logging** > **Log data to workspace** and specify a variable (`ScopeData` in this example). The `In3` block uses `m` (meters). Simulate the model and then at the MATLAB command line, enter:

```
ScopeData.get(1).Values.DataInfo.Units

ans =

  Units with properties:

    Name: 'm'
```

See Also

Related Examples

- “Load Signal Data That Uses Units” on page 70-59

More About

- “Units in Simulink”
- “Unit Consistency Checking and Propagation” on page 9-9

Limit Amount of Exported Data

In this section...

You can use several options to reduce the amount of data logged during a simulation. Limiting the amount of exported simulation data reduces memory usage and speeds up simulation. However, if you limit the amount of simulation data, the logged data can skip some time steps that are critical for testing and analyzing the model.

You can use multiple techniques for the same simulation.

Technique	Description
Specify a decimation factor	Skip samples when exporting data.
Limit data points	Limit the number of samples saved to be only the most recent samples
Specify an interval for logging	Specify ranges of time steps for logging

Alternatively, for logging large amounts of data that can cause memory issues, consider logging to persistent storage. This approach preserves all the logging data, minimizing MATLAB workspace memory usage. For details, see “Log Data to Persistent Storage” on page 72-31.

Decimation

To skip samples when exporting data, apply a decimation factor. For example, a decimation factor of 2 saves every other sample. By default, decimation is set to 1, which does not skip any samples.

The approach you use to specify a decimation factor depends on the kind of logging data.

Data	How to Specify
Signal logging	Right-click the signal. In the Signal Properties dialog box, select the **Decimation** parameter.
Data store logging	From the Block Parameters dialog box for that block, open the **Logging** tab. Apply a decimation factor using the **Decimation** parameter.
State and output	Enter a value in the field to the right of the **Decimation** label.

Limit Data Points

To limit the number of samples saved to be only the most recent samples, set the **Limit Data Points** parameter.

The approach you use depends on the kind of logging data.

Data	How to Specify
Signal logging	Right-click the signal. In the Signal Properties dialog box, select the **Limit Data Points to Last** parameter.
Data store logging	From the Block Parameters dialog box for that block, open the **Logging** tab. Select the **Limit Data Points to Last** parameter.
Time, state, and output logging	Select the **Limit data points** configuration parameter and for the **Maximum number of data points** configuration parameter, specify the limit.

Logging Intervals

To specify an interval for logging, use the **Configuration Parameters > Data Import/Export > Logging intervals** parameter. Limiting logging to a specified interval allows you to examine specific logged data without changing the model or adding complexity to a model.

The logging intervals apply to data logged for:

- Time
- States
- Output
- Signal logging
- The To Workspace block
- The To File block

The logging intervals do not apply to final state logged data, scopes, or streaming data to the Simulation Data Inspector.

The intervals specified with **Logging intervals** establish the set of times to which the **Decimation** and **Limit data points to last** parameters apply. For example, suppose that you set the logging interval `[2,4;7,9]` with a fixed-step solver with a fixed-step size of 1. The logged times are 2, 3, 4, 7, 8, and 9.

See Also

Related Examples

- “Export Signal Data Using Signal Logging” on page 72-41
- “Load Data Using the From File Block” on page 70-60
- “Load Data Using the From Workspace Block” on page 70-65

More About

- “Decimation”
- “Limit data points”

- “Logging intervals”

Work with Big Data for Simulations

Simulation of models with many time steps and signals can involve big data that is too large to fit into the RAM of your computer. Such situations include:

- Logging simulation data (signal logging, output port logging, and state logging)
- Loading input signal data for simulating a model
- Running multiple or parallel simulations

To work with big data for simulations, store the data to persistent storage in a MAT-file. Using big data techniques for simulations requires additional steps beyond what you do when the data is small enough to fit in workspace memory. As you develop a model, consider logging and loading simulation data without using persistent storage unless you discover that your model has big data requirements that overload memory.

Big Data Workflow

This example is a high-level workflow for handling big data that one simulation produces and that another simulation uses as input. For more detailed information about the major workflow tasks, see:

- “Log Data to Persistent Storage” on page 72-31
- “Load Big Data for Simulations” on page 70-7
- “Analyze Big Data from a Simulation” on page 72-35

Tip This example uses a `SimulationDatastore` object for streaming data into a model. Alternatively, you can stream a `DatasetRef` object directly into a model.

1. Configure two models to log several signals.
2. Simulate the models, logging the data to persistent storage for each model.

   ```
   sim(mdl1,'LoggingToFile','on','LoggingFileName','data1.mat');
   sim(mdl2,'LoggingToFile','on','LoggingFileName','data2.mat');
   ```

   Logging that involves big data requires saving the data to persistent storage as a v7.3 MAT-file. Only the data logged in `Dataset` format is saved to the file. Data logged in other formats, such as `Structure with time`, is saved in memory, in the base workspace.

   The data that you log to persistent storage is streamed during the simulation in small chunks, to minimize memory requirements. The data is stored in a file that contains `Dataset` objects for each set of logged data (for example, `logsout` and `xout`).
3. Create `DatasetRef` objects (`dsr1` and `dsr2`) for specific sets of logged signals. Then create `SimulationDatastore` objects (`dst1` and `dst2`) for values of elements of the `DatasetRef` objects. This example code creates a `SimulationDatastore` for the 12th element of `logsout` for the first simulation. For the second simulation, the example code creates a signal with values being a `SimulationDatastore` object for the seventh element of `logsout`. You can use curly braces for indexing.

   ```
   dsr1 = Simulink.SimulationData.DatasetRef('data1.mat','logsout');
   dsr2 = Simulink.SimulationData.DatasetRef('data2.mat','logsout');
   ```

```
dst1 = dsr1{12};
dst2 = dsr2{7};
```

4. Use `SimulationDatastore` objects as an external input for another simulation. To load the `SimulationDatastore` data, include it in a `Dataset` object. The datastore input is incrementally loaded from the MAT-file. The third input is a `timeseries` object, which is loaded into memory as a whole, not incrementally.

```
input = Simulink.SimulationData.Dataset;
input{1} = dst1;
input{2} = dst2;
ts = timeseries(rand(5,1),1,'Name','RandomSignals');
input{3} = ts;
sim(mdl3,'ExternalInput','input');
```

5. Use MATLAB big data analysis to work with the `SimulationDatastore` objects. Create a `timetable` object by reading the values of a `SimulationDatastore` object. The `read` function reads a portion of the data. The `readall` function reads all the data.

```
tt = dst1.Values.read;
```

6. Set the MATLAB session as the global execution environment (`mapreducer`) for working with the tall `timetable`. Create a tall `timetable` from a `SimulationDatastore` object and read a `timetable` object with in-memory data.

```
mapreducer(0);
ttt = tall(dst1.Values);
```

Tip For another example showing how to work with big simulation data, see Working with Big Data.

See Also

Functions

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `matlab.io.datastore.SimulationDatastore` | `timeseries`

Related Examples

- Working with Big Data
- “Log Data to Persistent Storage” on page 72-31
- “Load Big Data for Simulations” on page 70-7
- “Analyze Big Data from a Simulation” on page 72-35
- “Run Multiple Simulations” on page 27-2
- “Large Files and Big Data”

Log Data to Persistent Storage

In this section...

When to Log to Persistent Storage

In some cases, logging simulation data can create large amounts of data that are too large for your computer to hold in working memory while also running efficiently. Such situations can include simulations that log many signals, simulations that run for a long time with many time steps, and parallel simulations. When your simulation configuration creates a large amount of data, you can log that data to persistent storage, rather than logging it to working memory.

You can store logged simulation data to persistent storage in a MAT-file. You control logging to persistent storage at the model level. You can enable and disable the feature by changing one model configuration parameter (**Log Dataset data to file**) without changing the model layout.

If you use `Dataset` format for logging, you can log each of these kinds of data to persistent storage:

- Signal logging — Uses `Dataset` format only.
- States — Defaults to `Dataset` format. You can use other formats.
- Final states — Requires that you clear the **Save final operating point** parameter
- Output — Defaults to `Dataset` format.
- Data stores — Uses `Dataset` format only.

By default, logging to persistent storage is disabled, so that logged data is stored in the MATLAB workspace. For most models, logging to the workspace is simpler because it avoids loading and saving logging files. Compared to accessing data logged to memory, accessing data logged to persistent storage requires some additional steps. For short simulations, logging to the MATLAB workspace can be faster and possibly use less memory than logging to persistent storage.

Limitations for Logging to Persistent Storage

- Only data logged in `Dataset` format is stored in the MAT-file. Data logged in other formats is stored in the MATLAB workspace.
- To use persistent storage for logging final states data, you cannot enable the **Configuration Parameters > Data Import/Export + Save final operating point**.
- The Simulation Stepper and fast restart do not support logging to persistent storage.
- During simulation, you cannot load data from the persistent storage file directly into the model. Create objects that reference the data in the file and then load the referencing object.

Alternative Approaches for Reducing Logging Memory Usage

When you need to simulate a model that creates a large amount of data and you do not want to log the simulation data to persistent storage, consider using one of these alternatives.

- Limit the amount of simulation data stored in the workspace.

  You can limit the amount of simulation data stored in the workspace by using one or more of these techniques. For details, see “Limit Amount of Exported Data” on page 72-26.

Technique	Description
Specify a decimation factor	Skip samples when exporting data.
Limit data points	Limit the number of samples saved to be only the most recent samples.
Specify intervals for logging	Specify ranges of time steps for logging.

  If you limit the amount of simulation data stored in the workspace, the logged data may not contain some time steps that are critical for testing and analyzing the model.
- Use a To File block for each signal that you want to log.

  Connecting a To File block to signals that you want to log stores the logged data in a MAT-file, rather than in the MATLAB workspace. However, this approach:

  - Is a per-signal approach that can clutter a model with multiple To File blocks attached to individual signals.
  - Creates a separate MAT-file for each To File block, instead of the single file created when you log to persistent storage.

Log to Persistent Storage

1. Specify the kinds of logging to perform (for example, signal logging and output logging) and the variable names for the logging data.
2. In the model diagram, mark selected signals for signal logging.
3. Use `Dataset` format for logging the data. Data that is logged in any other format is stored in the workspace.

   - Signal logging and data store logging use `Dataset` format only. The default format for output, states, and final states logging is `Dataset`.
   - For final states logging, clear the **Save final operating point** configuration parameter.
4. Enable logging to persistent storage and specify an output MAT-file name.

   - Select the **Log Dataset data to file** configuration parameter.
   - Specify the MAT-file to use. Do not use a file name from one locale in a different locale.
5. To save the logged `Dataset` data using `timeseries` or `timetable` elements, set the **Dataset signal format** configuration parameter. The default format is `timeseries`. The `timetable` format is helpful for MATLAB combining logged data from multiple simulations. For details about the `timetable` format, see “Dataset signal format”.
6. Simulate the model.

Enable Logging to Persistent Storage Programmatically

You can programmatically log to persistent storage. To enable logging to persistent storage, use the `LoggingToFile` and `LoggingFileName` name-value pairs with either the `sim` command or `set_param` command.

To enable the logging approaches that you want to use, set these parameters to `'on'`, as applicable:

- `SignalLogging`
- `SaveState`
- `SaveFinalState`
- `SaveOutput`
- `DSMLogging`

To log output, states, and final states data to persistent storage, set the `SaveFormat` parameter to `'Dataset'`.

To log final states data to persistent storage, set the `SaveOperatingPoint` to `'off'`.

How Simulation Data Is Stored

Logging to persistent storage saves logged simulation data in the specified MAT-file. The data is stored as a `Simulink.SimulationData.Dataset` objects for each type of logging that uses `Dataset` format. The `Dataset` elements are stored as either `timeseries` or `timetable` objects, depending on how you set the **Dataset signal format** parameter. For details about the `timetable` format, see “Dataset signal format”.

The `Dataset` object name in the file is the name of the variable that you used for logging. For example, if you use the default signal logging variable `logsout`, the `Dataset` object in the MAT-file is `logsout`.

Save Logged Data from Successive Simulations

The approach you use for saving data logged from successive simulations depends on whether you are performing parallel simulations.

Without Using Parallel Simulations

Each time you simulate a model without using parallel simulation, Simulink overwrites the contents of the MAT-file unless you change the name of the file between simulations. When you use a `Simulink.SimulationData.DatasetRef` object that references data in the MAT-file to retrieve data in the file, it retrieves the most recent version of the data. To preserve data from an earlier simulation, use one of these approaches:

- Between simulations, use the **Configuration Parameters** > **Data Import/Export** pane to specify a different name for the MAT-file for logging.
- Between simulations, save a copy of the MAT-file. Use a different file name than the name that you specify as the MAT-file for persistent storage, or move the MAT-file.
- Programmatically specify a new file name for each simulation run.

If you run multiple simulations that overlap in time, use a unique MAT-file for each model that you log to persistent storage.

If you change the file name used for logging to persistent storage, then to access the logged data, use one of these approaches:

- Create a `Simulink.SimulationData.DatasetRef` object.

- To match the new file name, change the `Location` property of the `DatasetRef` objects.

For details about using `DatasetRef` objects to access logged data, see “Load Big Data for Simulations” on page 70-7.

With Parallel Simulations

For parallel simulations, for which you specify an array of input objects, if you log to file, Simulink:

- Creates a MAT-file for each simulation
- Creates `Simulink.SimulationData.DatasetRef` objects to access output data in the MAT-file and includes those objects in the `SimulationOutput` object data
- Enables the `CaptureErrors` argument for simulation

For more information about parallel simulations, see “Run Multiple Simulations” on page 27-2.

See Also

Functions

“Dataset signal format” | `Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `timeseries` | `timetable`

Related Examples

- “Work with Big Data for Simulations” on page 72-29
- “Load Big Data for Simulations” on page 70-7
- “Analyze Big Data from a Simulation” on page 72-35
- “Run Multiple Simulations” on page 27-2

Analyze Big Data from a Simulation

In this section...

To access data logged to a MAT-file for analysis in MATLAB, use references to the data in the MAT-file.

Create DatasetRef Objects to Access Logged Datasets

When you log to a MAT-file, Simulink stores a `Simulink.SimulationData.Dataset` object in the specified MAT-file. The elements of the `Dataset` object in the file are `Dataset` objects. There is one `Dataset` object for each set of logged simulation data. For example, a file may contain a `Dataset` object that contains a `Dataset` object for logged signal data and another `Dataset` object for logged states data.

To access simulation `Dataset` format data for a set of logged simulation data, create `Simulink.SimulationData.DatasetRef` objects. You can access individual elements of the dataset using a `DatasetRef` object. For details, see “Load Individual Signals from a DatasetRef Object” on page 70-9.

Use SimulationDatastore Objects to Access Signal Data

To access leaf signals in a logged `Dataset`, create a `matlab.io.datastore.SimulationDatastore` object for the signal, based on the `DatasetRef` object for the `Dataset` that contains the signal. For details, see

“Stream Individual Signals Using SimulationDatastore Objects” on page 70-7.

You can operate on data referenced by a `SimulationDatastore` object. For example, you can get the data in a chunk to be read into memory from the MAT-file. For an example, see `matlab.io.datastore.SimulationDatastore`.

Create Timetables for MATLAB Analysis

When you read a `SimulationDatastore` object, using the `read` or `readall` method the output is in MATLAB `timetable` format. For details about the timetable format, see “Dataset signal format”.

You can use a `SimulationDatastore` object to create a `timetable` for the signal values and read a `timetable` object with in-memory data. For example, for `SimulationDatastore` object `dst1`:

```
tt = dst1.Values.read;
ttt = tall(dst1.Values);
```

Create Tall Timetables

You can create a tall `timetable`:

```
mapreducer(0);
ttt = tall(dst1.Values);
```

Access Persistent Storage Metadata

If you use persistent storage for several simulations, you can have multiple MAT-files. When you run multiple simulations using batch processing, you get multiple MAT-files if you specify a different persistent storage MAT-file for each simulation. For parallel simulations, Simulink produces a separate MAT-file for each simulation run. To help you identify and understand the context of the simulation data included in a MAT-file, Simulink stores metadata about logging to persistent storage.

A `Simulink.SimulationMetadata` object includes in its `ModelInfo` structure a `LoggingInfo` structure with two fields:

- `LoggingToFile` — Indicates whether logging to persistent storage is enabled (`'on'` or `'off'`)
- `LoggingFileName` — Specifies the resolved file name for the persistent storage MAT-file (if `LoggingToFile` is `'on'`).

The MAT-file used for persistent storage contains a `SimulationMetadata` variable that stores the same simulation metadata as the `Simulink.SimulationMetadata` object. The `SimulationMetadata` is a system-generated name, not a variable name that you specify.

To access the persistent logging storage metadata, use one of these alternatives:

- View simulation metadata by using the `SimulationOutput` object `SimulationMetadata` property.
- Use tab completion to access `SimulationMetadata` object properties such as `ModelInfo` and to access field names.
- Display simulation metadata in the Variable Editor. Click the `SimulationOutput` object and use one of these approaches:
  - Select the **Explore Simulation Metadata** check box (which displays the data in a tree structure).
  - Double-click the **SimulationMetadata** row.

Access Error Information

You can view error message and information about the stack and causes for simulation data by using the `SimulationOutput` object `ErrorMessage` property. For parallel simulations, if you are logging to file, Simulink enables the `CaptureErrors` argument for simulation.

See Also

Functions

“Dataset signal format” | `Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef` | `createInputDataset` | `matlab.io.datastore.SimulationDatastore` | `timeseries` | `timetable`

Related Examples

- “Work with Big Data for Simulations” on page 72-29
- “Log Data to Persistent Storage” on page 72-31
- “Load Big Data for Simulations” on page 70-7

Samples to Export for Variable-Step Solvers

In this section...

Output Options

Use the **Output options** list under **Configuration Parameters > Data Import/Export > Additional parameters** to control how much output the simulation generates when your model uses a variable-step solver.

- `Refine output` (default)
- `Produce additional output`
- `Produce specified output only`

Refine Output

The `Refine output` option provides additional output points when the simulation output does not include as many points as you would like. This parameter provides an integer number of output points between time steps. For example, a refine factor of 2 provides output midway between the time steps and at the steps. The default refine factor is 1.

Suppose that a sample simulation generates output at these times:

```
0, 2.5, 5, 8.5, 10
```

Choosing `Refine output` and specifying a refine factor of 2 generates output at these times:

```
0, 1.25, 2.5, 3.75, 5, 6.75, 8.5, 9.25, 10
```

To get smoother output more efficiently, change the refine factor instead of reducing the step size. When you change the refine factor, the solver generates additional points by evaluating a continuous extension formula at sample points. This option changes the simulation step size so that time steps coincide with the times that you specify for additional output.

The refine factor applies to variable-step solvers and is most useful when you are using `ode45`. The `ode45` solver can take large steps. However, when you graph simulation output, the output from this solver sometimes is not sufficiently smooth. In such cases, run the simulation again with a larger refine factor. A value of such as 4 for `ode45` can provide much smoother results.

Note This option helps the solver locate zero crossings, although it does not ensure that Simulink detects all zero crossings (see “Zero-Crossing Detection” on page 3-10).

Produce Additional Output

Use the `Produce additional output` option to specify directly those additional times at which the solver generates output. When you select this option, the **Data Import/Export** pane displays an

Output times configuration parameter. In this parameter, enter a MATLAB expression that evaluates to an additional time or a vector of additional times. The solver produces hit times at the output times that you specify, in addition to the times it requires for accurate simulation.

Suppose that a sample simulation generates output at these times:

```
0, 2.5, 5, 8.5, 10
```

Choosing the `Produce additional output` option and specifying `[0:10]` generates output at these times:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

and perhaps at additional times, depending on the step size chosen by the variable-step solver.

Tips

- This option helps the solver locate zero crossings, although it does not ensure that Simulink detects all zero crossings (see “Zero-Crossing Detection” on page 3-10).
- Set the **Output times** configuration parameter to a value other than the default empty matrix (`[]`).
- For triggered subsystems and function-call subsystems, the calling function must inherit the sampling rate.

Produce Specified Output Only

Simulink generates output at the start and stop times, in addition to the times that you specify.

Suppose that a sample simulation generates output at these times:

```
0, 2.5, 5, 8.5, 10
```

Choosing the `Produce specified output only` option and specifying `[1:9]` generates output at these times:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

This option changes the simulation step size so that time steps coincide with the times that you specify for producing output. The solver can hit other time steps to accurately simulate the model. However, the output does not include these points. This option is useful when you are comparing different simulations to check that the simulations produce output at the same times.

Tips

- This option helps the solver locate zero crossings, although it does not ensure that Simulink detects all zero crossings (see “Zero-Crossing Detection” on page 3-10).
- Set the **Output times** configuration parameter to a value other than the default empty matrix (`[]`).
- In normal, accelerator, and rapid accelerator modes, Simulink generates output at the start and stop times, as well as at the times that you specify.
- When you simulate a model in normal mode, triggered subsystems and function-call subsystems use:

  - The times that you specify
  - All the time steps in between the values that you specify
  - The simulation start and stop times
- For triggered subsystems and function-call subsystems, the calling function must inherit the sampling rate.

See Also

Related Examples

- “Limit Amount of Exported Data” on page 72-26

More About

- “Zero-Crossing Detection” on page 3-10
- “Output options”

Export Signal Data Using Signal Logging

In this section...

Signal Logging

To capture signal data from a simulation, usually you can use signal logging. Mark the signals that you want to log and enable signal logging for the model. For details, see “Configure a Signal for Logging” on page 72-44 and “Enable Signal Logging for a Model” on page 72-54.

For a summary of other approaches to capture signal data, see “Export Simulation Data” on page 72-2.

Signal Logging Workflow

To collect and use signal logging data, perform these tasks.

1. Mark individual signals for signal logging. See “Configure a Signal for Logging” on page 72-44.
2. Enable signal logging for a model. See “Enable Signal Logging for a Model” on page 72-54.
3. Simulate the model.
4. Access the signal logging data. See “View and Access Signal Logging Data” on page 72-67.

Log Subsets of Signals

One approach for testing parts of a model as you develop it is to mark a superset of signals for logging and then override signal logging settings to select different subsets of signals for logging. You can use the Signal Logging Selector or a programmatic interface. See “Override Signal Logging Settings” on page 72-57.

Use this approach to log signals in models that use model referencing. For an example, see “Viewing Signals in Model Reference Instances”.

Additional Signal Logging Options

With the basic signal logging workflow, you can specify additional options related to the data that signal logging collects and to how that data is displayed. You can:

- Specify a name for the signal logging data for a signal. See “Specify Signal-Level Logging Name” on page 72-45.
- Control how much data the simulation generates for a signal. See “Limit Data Logged” on page 72-46.
- Review the signal logging configuration for a model. See “View the Signal Logging Configuration” on page 72-49.
- Specify the samples for export for models with variable-step solvers. See “Samples to Export for Variable-Step Solvers” on page 72-38.

Signal Logging in Rapid Accelerator Mode

Signal logging in rapid accelerator mode does not log the following kinds of signals. When you update or simulate a model that contains these signals, Simulink displays a warning that those signals are not logged.

- Signals inside Stateflow charts
- Signals that use a custom data type

If you set the **Configuration Parameters > Solver > Periodic sample time constraint** parameter to `Ensure sample time independent`, you cannot use signal logging in rapid accelerator mode.

Signal Logging Limitations

- Rapid accelerator mode supports signal logging, with the requirements and limitations described in “Signal Logging in Rapid Accelerator Mode” on page 72-42.
- Top-model and Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes support signal logging. For a description of limitations, see “Top-Model SIL/PIL Limitations” (Embedded Coder) and “Model Block SIL/PIL Limitations” (Embedded Coder).
- Array of buses signals support signal logging, with the requirements described in “Import Array of Buses Data” on page 70-49.
- You cannot log bus signals directly in For Each subsystems.
- You cannot log a signal inside a referenced model that is inside a For Each subsystem if either of these conditions exists:
  - The For Each subsystem is in a model simulating in rapid accelerator mode.
  - The For Each subsystem itself is in a model referenced by a Model block in accelerator mode.
- You cannot log signals that feed Function-Call subsystems or Action subsystems.
- You cannot log an input signal to a Merge block. You can log a Merge block output signal.
- For Integrator and Discrete-Time Integrator blocks that have the **Show state port** parameter enabled, you cannot log the state port signal.
- If you configure a bus signal or bus element for signal logging that is an input to a subsystem, you cannot automatically refactor the subsystem interface to use In Bus Element and Out Bus Element blocks. For details about that refactoring, see “Simplify Bus Interfaces in Subsystems” on page 76-25.
- You cannot log local data in Stateflow Truth Table blocks.

See Also

More About

- “Enable Signal Logging for a Model” on page 72-54
- “Configure a Signal for Logging” on page 72-44
- “View and Access Signal Logging Data” on page 72-67
- “View the Signal Logging Configuration” on page 72-49
- “Log Signals in For Each Subsystems” on page 72-71

- “Export Simulation Data” on page 72-2

Configure a Signal for Logging

In this section...

Mark a Signal for Logging

Enable logging by marking a signal, using one of the following techniques:

The Simulink Editor menu options are generally the simplest way to mark signals for logging.

A signal for which you enable logging is a *logged signal*. By default, Simulink displays a logged signal indicator  for each logged signal.

Enable Logging Using Simulink Toolstrip

1. In the Simulink Editor, select one or more signals.
2. On the **Simulation** tab, click **Log Signals**.

Enable Logging Using Signal Properties

1. In the Simulink Editor, right-click the signal.
2. From the context menu, select **Properties**.
3. In the Signal Properties dialog box, in the **Logging and accessibility** tab, select **Log signal data**.
4. Click **OK**.

Alternatively, you can select the **Log Selected Signals** from the context menu that appears when you right-click the selected signal.

Enable Logging Using the Model Data Editor

The Model Data Editor displays a flat list of signals in your model. You can sort, group, and filter the list. Use this technique to enable logging for:

- Many signals at once.
- Signals that are not close to each other in the block diagram.
- Signals that are difficult to locate in a large model or subsystem hierarchy.

To select signals to log using the Model Data Editor:

1. Open the Model Data Editor. On the **Modeling** tab, click the **Model Data Editor** button.
2. Select the **Signals** tab in the Model Data Editor.
3. Select `Instrumentation` in the drop-down.
4. Check the boxes in the **Log Data** column for signals you would like to log.

For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Programmatic Interface

To enable signal logging programmatically for selected blocks, use the output `DataLogging` property. Set this property using the `set_param` command. For example:

1. At the MATLAB Command Window, open a model. Type

   ```
   vdp
   ```

2. Get the port handles of the signal that you want to log. For example, for the Mu block output port signal.

   ```
   ph = get_param('vdp/Mu','PortHandles')
   ```

3. Enable signal logging for the desired output port signal.

   ```
   set_param(ph.Outport(1),'DataLogging','on')
   ```

   The logged signal indicator appears.

Logging Referenced Model Signals

You can log any logged signal in a referenced model. Use the Signal Logging Selector to configure signal logging for a model reference hierarchy. For details, see “Models with Model Referencing: Overriding Signal Logging Settings” on page 72-60.

Specify Signal-Level Logging Name

You can specify a signal-level logging name to the object that Simulink uses to store logging data for a signal. Specifying a signal-level logging name can be useful for signals that are unnamed or that share a duplicate name with another signal in the model hierarchy. Specifying signal-level logging names, rather than using the names that Simulink generates, can make the logged data easier to analyze.

To specify a signal-level logging name, use *one* of the following approaches:

- “Signal-Level Logging Name in the Editor” on page 72-46
- “Signal-Level Logging Name in Model Explorer” on page 72-46
- “Signal-Specific Logging Name Specified Programmatically” on page 72-46

If you do not specify a custom signal-level logging name, Simulink uses the signal name. If the signal does not have a name, the action Simulink uses a blank name.

Note The signal-level logging name is distinct from the model-level signal logging name. The model-level signal logging name is the name for the object containing all the logged signal data for the

whole model. The default model-level signal logging name is `logsout`. For details about the model-level signal logging name, see “Specify a Name for Signal Logging Data” on page 72-56.

Signal-Level Logging Name in the Editor

1. In the Simulink Editor, right-click the signal.
2. From the context menu, select **Signal Properties**.
3. Specify the logging name:
   a. In the Signal Properties dialog box, select the **Logging and accessibility** tab.
   b. From the **Logging name** list, select `Custom`.
   c. Enter the logging name in the adjacent text field.

Signal-Level Logging Name in Model Explorer

1. In the Model Explorer **Model Hierarchy** pane, select the node that contains the signal for which you want to specify a logging name.
2. If the **Contents** pane does not display the `LoggingName` property, add the `LoggingName` property to the current view. For details about column views, see **Model Explorer**.
3. Enter a logging name for one or more signals using the `LoggingName` column.

Signal-Specific Logging Name Specified Programmatically

Enable signal logging programmatically for selected blocks with the output port `DataLogging` property. Set this property using the `set_param` command.

1. At the MATLAB Command Window, open a model. For example, type:

   ```
   vdp
   ```

2. Get the port handles of the signal that you want to log. For example, for the Mu block output port signal:

   ```
   ph = get_param('vdp/Mu','PortHandles');
   ```

3. Enable signal logging for the desired output port signal:

   ```
   set_param(ph.Outport(1),'DataLogging','on');
   ```

   The logged signal indicator appears.

4. Issue commands that use the `DataLoggingNameMode` and `DataLoggingName` parameters. For example:

   ```
   set_param(ph.Outport(1),'DataLoggingNameMode','Custom');
   set_param(ph.Outport(1),'DataLoggingName','x2_log');
   ```

Limit Data Logged

You can limit the amount of data logged for a signal by:

- Specifying a decimation factor
- Limiting the number of samples saved to be only the most recent samples

You can limit data logged for a signal by using the Signal Properties dialog box, the Model Explorer, the Signal Logging Selector, or programmatically. The following sections describe the first two approaches.

Use Signal Properties to Limit Logged Data

1. In the Simulink Editor, right-click the signal.
2. From the context menu, select **Signal Properties**.
3. In the Signal Properties dialog box, click the **Logging and accessibility** tab. Then select one or both of these options:
   - **Limit data points to last**
   - **Decimation**

Use Model Explorer to Limit Data Logged

1. In the Model Explorer **Model Hierarchy** pane, select the node that contains the signal for which you want to limit the amount of data logged.
2. If the **Contents** pane does not display the `DataLoggingDecimation` property or the `DataLoggingLimitDataPoints` property, add one or both of those properties to the current view. For details about column views, see **Model Explorer**.
3. To specify a decimation factor, edit the `Decimation` and `DecimateData` properties. To limit the number of samples logged, edit the `LimitDataPoints` property.

Set Sample Time for a Logged Signal

To set the sample time for a logged signal, in the Signal Properties dialog box, use the **Sample Time** option. This option:

- Separates design and testing, because you do not need to insert a Rate Transition block to have a consistent sample time for logged signals
- Reduces the amount of logged data for a continuous time signal, for which setting decimation is not relevant
- Eliminates the need to postprocess logged signal data for signals with different sample times

Usage Notes

Do not specify a sample time for:

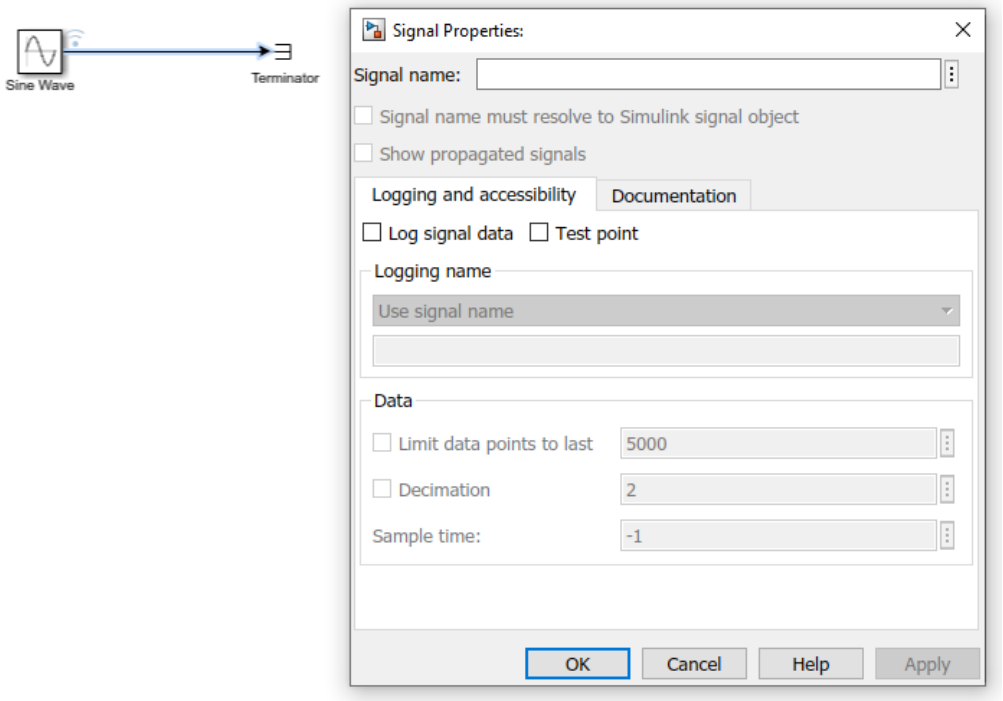
- Frame-based signals
- Conditional subsystems (for example, function-call or triggered subsystems) and conditional referenced models, which require an inherited sample time

If you simulate in SIL mode, signal logging ignores the sample times you specify for logged signals.

When you mark a signal for signal logging, Simulink inserts a hidden To Workspace block. When you specify a sample time for a logged signal, Simulink inserts a hidden Rate Transition block and a hidden To Workspace block.

Specifying a sample time for signal logging does not affect the simulation result. However, it is possible that the signal logging output for a logged signal varies depending on whether you specify a sample rate. For example, the interpolation method can differ depending on whether you specify a

sample time for signal logging. Suppose that a model includes a continuous signal and the sample time is inherited (`-1`). The logged output for that signal shows that the interpolation method is `linear`.




```
logsout.get(1).Values.DataInfo
```

```
tsdata.datametadata
  Package: tsdata

  Common Properties:
              Units: ''
      Interpolation: linear (tsdata.interpolation)
```

If you change the sample time to be continuous (`0`), the logged output for that signal shows that the interpolation method is `zoh` (zero-order hold).

See Also

Related Examples

- “Export Signal Data Using Signal Logging” on page 72-41
- “View the Signal Logging Configuration” on page 72-49
- “Enable Signal Logging for a Model” on page 72-54
- “Override Signal Logging Settings” on page 72-57
- “Log Signal Data That Uses Units” on page 72-24

View the Signal Logging Configuration

In this section...

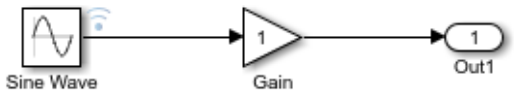
Approaches for Viewing the Signal Logging Configuration

Signal Logging Configuration Viewing Approach	Usage	Documentation
In the Simulink Editor, view signal logging indicators.	Consider using this approach for models that have few signals marked for signal logging and have a shallow model hierarchy. This approach avoids leaving the Simulink Editor. Open the Signal Properties dialog box for each signal.	“View Signal Logging Configuration Using the Simulink Editor” on page 72-50
Use the Signal Logging Selector.	Consider using this approach for models with deep hierarchies. View a model that has signal logging override settings for some signals. View the configuration as part of specifying a subset of signals for logging from all signals marked for signal logging. View signal logging configuration without displaying the signal logging indicators in the model. View signal logging configuration information such as decimation and output options in one window.	“View Logging Configuration Using the Signal Logging Selector” on page 72-51
Use the Model Explorer.	View signal logging configuration in the context of other model component properties. Adjust the column view to display signal logging properties, if necessary.	“View Signal Logging Configuration Using the Model Explorer” on page 72-52

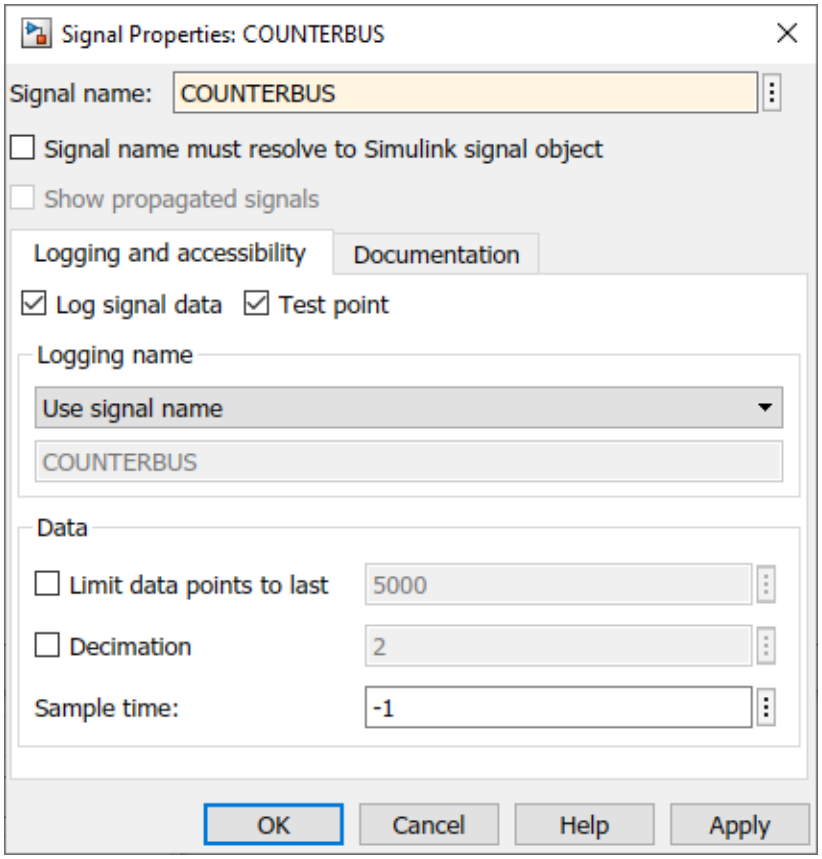
Signal Logging Configuration Viewing Approach	Usage	Documentation
Use MATLAB commands	Get the handles of the signals in the model and find the ones that have data logging enabled.	“Programmatically Find Signals Configured for Logging” on page 72-53

View Signal Logging Configuration Using the Simulink Editor

By default, the Simulink Editor displays an indicator on each signal marked for logging. For example, this model logs the output signal of the Sine Wave block.



To view the signal properties, right-click the signal and select **Properties**.



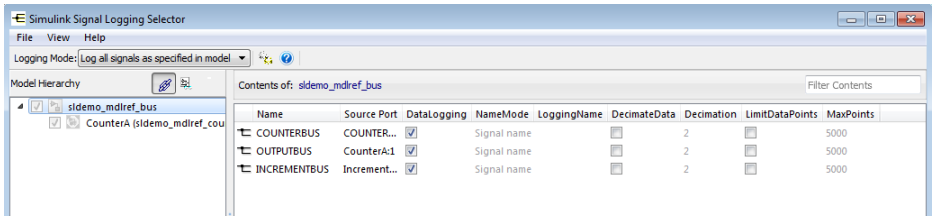
If you programmatically override logging for a signal, the Simulink Editor continues to display the signal logging indicator for that signal. When you simulate the model, Simulink displays a red signal logging indicator for all signals marked to be logged, reflecting any overrides. For details about configuring a signal for logging, see “Configure a Signal for Logging” on page 72-44.

A logged signal can also be a test point. See “Configure Signals as Test Points” on page 75-43 for information about test points.

To hide logging indicators, on the **Debug** tab, click to clear **Information Overlays > Log & Testpoint**.

View Logging Configuration Using the Signal Logging Selector

In the **Simulation** tab, click **Model Settings**. Then, click **Configure Signals to Log**.



The **Contents** pane shows the signals marked for logging in the node selected in the **Model Hierarchy** pane. When no signals are marked for logging in a node, the **Contents** pane is empty. Use the arrow to the left of a hierarchical node to expand or collapse the contents of the node in the **Model Hierarchy** pane.

When a model includes referenced models, the check box in the **Model Hierarchy** pane indicates the override configuration for the model corresponding to the node.

Check Box	Signal Logging Configuration
	For the top-level model node, logs all logged signals in the top model. For a Model block node, logs all logged signals in the model reference hierarchy for that block.
	For the top-level model node, disables logging for all logged signals in the top-level model. For a Model block node, disables logging for all signals in the model reference hierarchy for that block.
	For the top-level model node, logs all logged signals that have the `DataLogging` setting enabled. For a Model block node, logs all logged signals in the model reference hierarchy for that block that have the `DataLogging` setting enabled.

View Configuration of Subsystems and Linked Libraries

The following table describes default **Model Hierarchy** pane display of subsystems, masked subsystems, and linked library nodes.

Node	Display Default
Subsystem	Displays subsystems all that include logged signals
Masked subsystem	Does not display masked subsystems

Node	Display Default
Linked library	Displays all subsystems that include logged signals

You can control how the **Model Hierarchy** pane displays subsystems, masked subsystems, and linked libraries. Use icons at the top of the **Model Hierarchy** pane or use the **View** menu, using the same approach as you use in the Model Explorer. For details, see **Model Explorer** and “Manage Existing Masks” on page 39-12.

- To display all subsystems, including subsystems that do not include signals marked for logging, select the icon or **View > Show All Subsystems**. This subsystem setting also applies to masked subsystems, if you specify to display masked subsystems.
- To display masked subsystems with logged signals, use the icon or **View > Show Masked Subsystems**
- To display linked libraries, use the icon or **View > Show Library Links**

Filtering Signal Logging Selector Contents

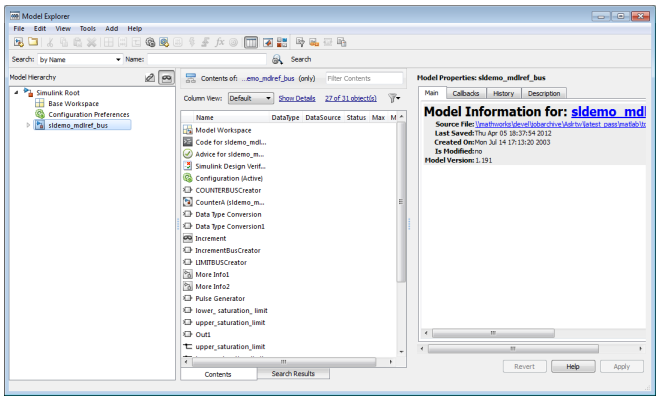
To find a specific signal or property value for a signal, use the **Filter Contents** edit box. Use the same approach as you use in the Model Explorer; for details, see **Model Explorer**.

Highlighting a Block in a Model

To use the Model Hierarchy pane to highlight a block in model, right-click the block or signal and select **Highlight block in model**.

View Signal Logging Configuration Using the Model Explorer

1. To access the logging configuration information for referenced models, open the model for which you want to view the signal logging configuration. Select the top-level model in a model reference hierarchy.
2. In the **Contents** pane, set **Column View** to the `Signals` view.



For further information, see **Model Explorer**.

Programmatically Find Signals Configured for Logging

Use MATLAB commands to get the handles of the signals in the model and find the ones that have data logging enabled. For example:

```
mdlsignals = find_system(gcs,'FindAll','on','LookUnderMasks','all',...
        'FollowLinks','on','type','line','SegmentType','trunk');
ph = get_param(mdlsignals,'SrcPortHandle')
for i=1: length(ph)
    get_param(ph{i},'datalogging')
end
```

See Also

Related Examples

- “Configure a Signal for Logging” on page 72-44
- “View Logging Configuration Using the Signal Logging Selector” on page 72-51

Enable Signal Logging for a Model

In this section...

Enable and Disable Logging at the Model Level

To log a signal, mark it for logging. For details, see “Configure a Signal for Logging” on page 72-44.

Enable or disable logging globally for all signals that you mark for logging in a model. By default, signal logging is enabled. Simulink logs signals if the **Configuration Parameters > Data Import/Export > Signal logging** parameter is checked. If the option is not checked, Simulink ignores the signal logging settings for individual signals.

When signals are marked for logging, the signal data logs to the workspace and to the Simulation Data Inspector. You can disable signal logging through the Configuration Parameters dialog box or programmatically.

- In the Configuration Parameters dialog box, clear the **Configuration Parameters > Data Import/Export > Signal logging** parameter check box.
- From the command line, use the `SignalLogging` parameter.

  ```
  set_param(bdroot,'SignalLogging','off')
  ```

Selecting a Subset of Signals to Log

You can select a subset of signals to log for a model that has:

- Signal logging enabled
- Logged signals

For details, see “Override Signal Logging Settings” on page 72-57.

Specify Format for Dataset Signal Elements

Logged signal data is saved in `Dataset` format (as `Simulink.SimulationData.Dataset` objects). To specify whether you want the data for individual signals in the dataset to use MATLAB `timeseries` or `timetable` elements, set the **Dataset signal format** configuration parameter. The default is `timeseries`. For details, see “Dataset signal format”.

Migrate Scripts That Use Legacy ModelDataLogs API

For scripts that simulate a model created in a release earlier than R2016a that uses `ModelDataLogs` format for logging, update the code to log in `Dataset` format.

If you have already logged signal data in the `ModelDataLogs` format, you can use the `Simulink.ModelDataLogs.convertToDataset` function to update the `ModelDataLogs` signal logging data to use `Dataset` format. For example, to update the `older_model_dataset` from `ModelDataLogs` format to `Dataset` format:

```
new_dataset = logsout.convertToDataset('older_model_data')
```

Converting a model from using `ModelDataLogs` format to using `Dataset` format can require that you modify your existing models and to code in callbacks, functions, scripts, or tests. The following table identifies possible issues to address after converting to `Dataset` format. The table provides solutions for each issue.

Possible Issue After Conversion to Dataset Format	**Solution**

Code in existing callbacks, functions, scripts, or tests that used the `ModelDataLogs` programmatic interface to access data can result in an error.	Check for code that uses `ModelDataLogs` format access methods. Update that code to use `Dataset` format access methods.
	
	For example, suppose that existing code includes the following line:
	
	`logsout.('Subsystem Name').X.data`
	
	Replace that code with a `Dataset` access method:
	
	`logsout.getElement('x').Values.data`

Mux block signal names are lost.	The `Dataset` format treats Mux block signals as a vector. To identify signals by signal names, replace Mux blocks with Bus Creator blocks.
Signal Viewer cannot be used for signal logging.	Simulink does not log signal logging data in the Signal Viewer. Use the signal logging output variable to view the logged data.
The `unpack` method generates an error.	The `unpack` method, which is supported for `Simulink.ModelDataLogs` and `Simulink.SubsysDataLogs` objects, is *not* supported for `Simulink.SimulationData.Dataset` objects. For example, if the data in `mlog` has three fields: `x`, `y`, and `z`, then: For `ModelDataLogs` format data, the `mlog.unpack` method creates three variables in the base workspace. For `Dataset` format data, access methods by names. For example: `x = logsout.getElement('x').Values`

Possible Issue After Conversion to Dataset Format	Solution
The `ModelDataLogs` and `Dataset` formats have different naming rules for unnamed signals.	If necessary, add signal names. In `ModelDataLogs` format, for an unnamed signal coming from a block, Simulink assigns a name in this form: `SL_BlockName+<portIndex>` For example, `SL_Gain1`. In `Dataset` format, elements do not need a name, so Simulink leaves the signal name empty. For both `ModelDataLogs` and `Dataset` formats, Simulink assigns the same name to unnamed signals that come from Bus Selector blocks.
Test points in referenced models are not logged.	Consider enabling signal logging for test points in a referenced model.
Script uses `who` or `whos` functions.	Consider using `find` instead.

Specify a Name for Signal Logging Data

You use the model-level signal logging name to access the signal logging data for a model. The default name for the signal logging data is `logsout`. Specifying a model-level signal logging name can make it easier to identify the source of the logged data. For example, you could specify the signal logging name `car_logsout` to identify the data as being the signal logging data for the `car` model.

To specify a different model-level signal logging name, use either of these approaches:

- In the edit box next to the **Configuration Parameters** > **Data Import/Export** > **Signal logging** parameter, enter the signal logging name.
- Use the `SignalLoggingName` parameter, specifying a signal logging name. For example:

```
set_param(bdroot, 'SignalLoggingName', 'heater_model_signals')
```

See Also

Related Examples

- “Configure a Signal for Logging” on page 72-44
- “Export Signal Data Using Signal Logging” on page 72-41

Override Signal Logging Settings

In this section...

Benefits of Overriding Signal Logging Settings

As you develop a model, you may want to override the signal logging settings for a specific simulation run. You can override signal logging properties without changing the model in the Simulink Editor.

To reduce memory overhead and to facilitate the analysis of simulation logging results, override signal logging properties. By overriding signal logging settings, you can avoid recompiling a model.

Overriding signal logging properties is useful when you want to:

- Focus on only a few signals by disabling logging for most of the signals marked for logging. You can mark a superset of signals for logging, and then select different subsets of them for logging.
- Exclude a few signals from the signal logging output.
- Override specific signal logging properties, such as decimation, for a signal.
- Collect only what you need when running multiple test vectors.

Two Interfaces for Overriding Signal Logging Settings

Use either of two interfaces to override signal logging settings:

- “Override Signal Logging Settings with Signal Logging Selector” on page 72-58
- “Override Signal Logging Settings from MATLAB” on page 72-62

You can use a combination of the two interfaces. The Signal Logging Selector creates `Simulink.SimulationData.ModelLoggingInfo` objects when saving the override settings. The command-line interface has properties whose names correspond to the Signal Logging Selector interface. For example, the `Simulink.SimulationData.ModelLoggingInfo` class has a `LoggingMode` property, which corresponds to the **Logging Mode** parameter in the Signal Logging Selector.

Scope of Signal Logging Setting Overrides

When you override signal logging settings, Simulink uses those override settings when you simulate the model.

Simulink saves in the model the signal logging override configuration that you specify. However, Simulink does not change the signal logging settings in the Signal Properties dialog box for each signal in the model.

In the Signal Logging Selector, if you override some signal logging settings, and then set the **Logging Mode** to `Log all signals as specified in model`, the logging settings defined in the model appear in the Signal Logging Selector. The override settings are greyed out, indicating that you cannot override these settings. To reactivate the override settings, set **Logging Mode** to `Override signals`. Using the Signal Logging Selector to override logging for a specific signal does not affect the signal logging indicator for that signal.

If you close and then reopen the model, the logging setting overrides that you made are in effect, if logging mode is set to override signals for that model. When the model displays the signal logging indicators, it displays the indicators for all logged signals, including logged signals that you have overridden.

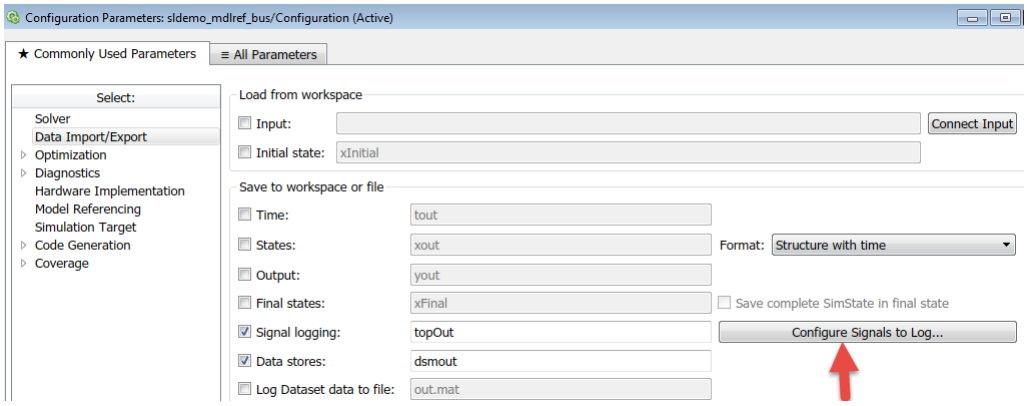
Note Simulink rebuilds a model in the following situation:

1. The model contains one or more signals marked for signal logging.
2. You simulate the model in rapid accelerator mode.
3. You use the Signal Logging Selector or MATLAB command line to modify the signal logging configuration.
4. You simulate the model in rapid accelerator mode again.

Override Signal Logging Settings with Signal Logging Selector

1. Open the Signal Logging Selector, using one of the following approaches:

   - In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.

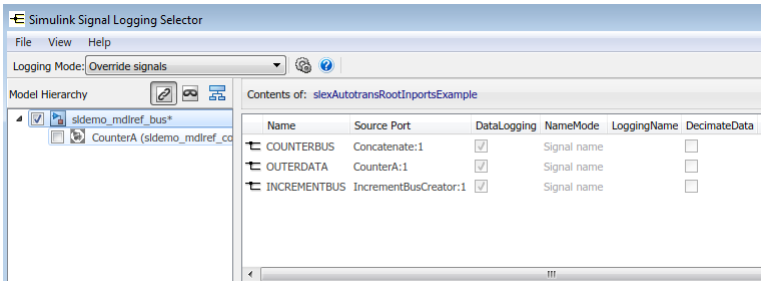


   Tip To enable the **Configure Signals to Log** button, select the **Signal logging** configuration parameter.

   - For a model that includes a Model block, you can also use the following approach:

     a. In the Simulink Editor, right-click a Model block.

     b. In the context menu, select **Log Referenced Signals**.



2 Set **Logging Mode** to `Override signals`.

Note The `Override signals` setting affects all levels of the model hierarchy. This setting can result in turning off logging for any signal throughout the hierarchy, based on existing settings. To review settings, select the appropriate node in the **Model Hierarchy** pane.

3 View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See “View Logging Configuration Using the Signal Logging Selector” on page 72-51.

4 Override signal logging settings. Use one of the following approaches, depending on whether your model uses model referencing:

- “Models Without Model Referencing: Overriding Signal Logging Settings” on page 72-59
- “Models with Model Referencing: Overriding Signal Logging Settings” on page 72-60

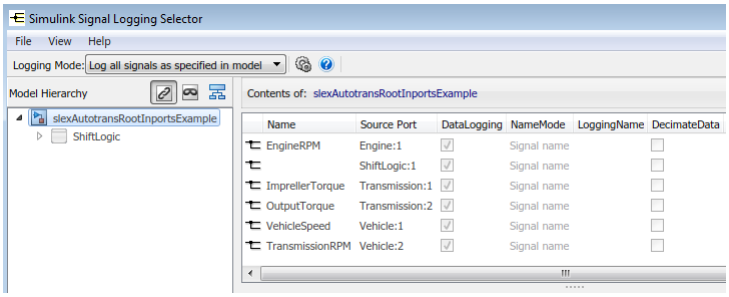
Tip To open the **Configuration Parameters > Data Import/Export** pane from the Signal Logging Selector, use the  button.

Models Without Model Referencing: Overriding Signal Logging Settings

If your model does not use model referencing (that is, the model does not include any Model blocks), override signal logging settings using the following procedure.

1 Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.

- If necessary, select **Signal logging** to enable the **Configure Signals to Log** button.



2 Set **Logging Mode** to `Override signals`.

3 View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See “View Logging Configuration Using the Signal Logging Selector” on page 72-51.

4. In the **Contents** pane table, select the signal whose logging settings you want to override.
5. Override logging settings:
   - To disable logging for a signal, clear the `DataLogging` check box for that signal.
   - To override other signal logging settings (for example, decimation), ensure that the `DataLogging` check box is selected. Then, edit values in the appropriate columns.

Models with Model Referencing: Overriding Signal Logging Settings

If your model uses model referencing (that is, the model includes at least one Model block), override signal logging settings using one or more of these procedures:

- "Enable Logging for All Logged Signals" on page 72-60
- "Disable Logging for All Signals in Node" on page 72-60
- "Override Signal Logging for a Subset of Signals" on page 72-61
- "Override Other Signal Logging Properties" on page 72-61

Enable Logging for All Logged Signals

By default, Simulink logs all the logged signals in a model, including the logged signals throughout model reference hierarchies.

If logging is disabled for any logged signals in the top-level model or in the top-level Model block in a model reference hierarchy, then in the **Model Hierarchy** pane, the check box to the left of that node is:

- Solid (), if logging is disabled for some of signals.
- Empty (), if logging is disabled for all the signals.

To enable logging of all logged signals for a node:

1. Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.
2. Set **Logging Mode** to `Override signals`.
3. View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See "View Logging Configuration Using the Signal Logging Selector" on page 72-51.
4. In the **Model Hierarchy** pane, select the check box to the left of the node, so that the check box has a check mark ().
   - For the top-level model, logging is enabled for all logged signals in the top-level model, but not for logged signals in model reference hierarchies.
   - For a Model block at the top of a model referencing hierarchy, logging is enabled for the whole model reference hierarchy for the selected referenced model.

Disable Logging for All Signals in Node

If signal logging is enabled for any signals in a model node, then in the **Model Hierarchy** pane, the check box to the left of the node is:

- Solid (), if logging is enabled for some signals.
- Checked (), if logging is enabled for all signals.

To disable logging for all logged signals in a node of a model:

1. Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.
2. Set **Logging Mode** to `Override signals`.
3. View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See “View Logging Configuration Using the Signal Logging Selector” on page 72-51.
4. In the **Model Hierarchy** pane, clear the check box to the left of the node, so that the check box is empty ().
   - For the top-level model, logging is disabled for all logged signals in the top-level model, but not for logged signals in model reference hierarchies.
   - For a Model block at the top of a model referencing hierarchy, logging is disabled for the whole model reference hierarchy for the selected referenced model.

Override Signal Logging for a Subset of Signals

To log some, but not all, logged signals in a model node:

1. Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.
2. Set **Logging Mode** to `Override signals`.
3. View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See “View Logging Configuration Using the Signal Logging Selector” on page 72-51.
4. In the **Model Hierarchy** pane, ensure that the check box for the top-level model or Model block is either solid (), if logging is disabled for some of the signals, or empty (), if logging is disabled for all the signals. Click the check box to cycle through different states.
5. In the **Contents** pane table, for the signals that you want to log, select the check box in the `DataLogging` column.

   To enable logging for multiple signals, hold the **Shift** or **Ctrl** key and select a range of signals or individual signals. Select the check box in the `DataLogging` column of one of the highlighted signals.

Override Other Signal Logging Properties

In addition to overriding the setting for the `DataLogging` property for a signal, you can override other signal logging properties, such as decimation.

1. Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, click the **Configure Signals to Log** button.
2. Set **Logging Mode** to `Override signals`.

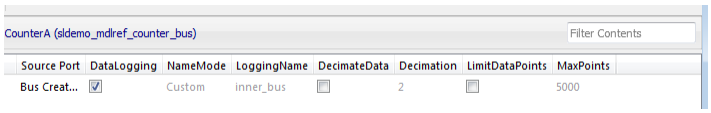
3. View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See “View Logging Configuration Using the Signal Logging Selector” on page 72-51.
4. In the **Model Hierarchy** pane, ensure that the check box for the top-level model or Model block is solid ( ) if logging is disabled for some signals, or empty ( ), if logging is disabled for all signals. Click the check box to cycle through different states.
5. In the **Contents** pane table, for the signals for which you want to override logging properties, enable logging by selecting the check box in the `DataLogging` column.

   To enable logging for multiple signals, hold the **Shift** or **Ctrl** key and select a range of signals or individual signals. Select the check box in the `DataLogging` column of one of the highlighted signals.
6. In the **Contents** pane table, modify the settings for properties, such as `DecimateData` and `Decimation`.

Override Signal Logging Settings from MATLAB

The MATLAB command-line interface for overriding signal logging settings includes:

- The `DataLoggingOverride` model parameter — Use to view or set signal logging override values for a model
- The following classes:
  - `Simulink.SimulationData.ModelLoggingInfo` — Specify signal logging override settings for a model. This class corresponds to the overall Signal Logging Selector interface.
  - `Simulink.SimulationData.SignalLoggingInfo` — Override settings for a specific signal. This class corresponds to a row in the logging property table in the Signal Logging Selector:

  - `Simulink.SimulationData.LoggingInfo` — Overrides for signal logging settings such as decimation. This class corresponds to the editable columns in a row in the logging property table in the Signal Logging Selector.

To query a model for its signal logging override status, use the `DataLoggingOverride` parameter.

To configure signal logging from the command line, use methods and properties of the three classes listed above. To apply the configuration, use `set_param` with the `DataLoggingOverride` model parameter.

The following sections describe how to use the command-line interface to perform some common signal logging configuration tasks.

- “Create a Model Logging Information Object” on page 72-63
- “Specify Which Models to Log” on page 72-63
- “Log a Subset of Signals” on page 72-64
- “Override Other Signal Logging Properties” on page 72-65

Create a Model Logging Information Object

To use the command-line interface for overriding signal logging settings, first create a `Simulink.SimulationData.ModelLoggingInfo` object. For example, use the following commands to create the model logging override object for the `ex_bus_logging` model and automatically add each logged signal in the model to that object:

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdlref_counter_bus')));
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
'ex_bus_logging')
```

```
mi =

  ModelLoggingInfo with properties:

                   Model: 'ex_bus_logging'
             LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                 Signals: [1x4 Simulink.SimulationData.SignalLoggingInfo]
```

The `LoggingMode` property is set to `OverrideSignals`, which configures the model logging override object to log only the signals specified in the `Signals` property.

To apply the model override object settings, use:

```
set_param(ex_bus_logging, 'DataLoggingOverride', mi);
```

Simulink saves the settings when you save the model.

You can control the kinds of systems from which to include logged signals. By default, the `Simulink.SimulationData.ModelLoggingInfo` object includes logged signals from:

- Libraries
- Masked subsystems
- Referenced models
- Active variants

As an alternative, you can use the `Simulink.SimulationData.ModelLoggingInfo` constructor and specify a `Simulink.SimulationData.SignalLoggingInfo` object for each signal. To ensure that you specified valid signal logging settings for a model, use the `verifySignalAndModelPaths` method with the `Simulink.SimulationData.ModelLoggingInfo` object for the model.

Specify Which Models to Log

To specify whether to use the signal logging settings as specified in the model and all referenced models, or to override those settings, use the `LoggingMode` property of a `Simulink.SimulationData.ModelLoggingInfo` object.

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. See the `Simulink.SimulationData.ModelLoggingInfo` documentation.

This example shows how to log all signals as specified in the top model and all referenced models. The signal logging output is stored in `topOut`.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
```

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdlref_counter_bus')));
mi = Simulink.SimulationData.ModelLoggingInfo...
     ('ex_bus_logging');
mi.LoggingMode = 'LogAllAsSpecifiedInModel'
```

```
mi =

  ModelLoggingInfo with properties:

                    Model: 'ex_bus_logging'
              LoggingMode: 'LogAllAsSpecifiedInModel'
    LogAsSpecifiedByModels: {}
                   Signals: []
```

To apply the model override object settings, use:

```
set_param(ex_bus_logging, 'DataLoggingOverride', mi);
```

The following example shows how to log only signals in the top model:

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdlref_counter_bus')));
mi = Simulink.SimulationData.ModelLoggingInfo...
     ('ex_bus_logging');
mi.LoggingMode = 'OverrideSignals';
mi = mi.setLogAsSpecifiedInModel('ex_bus_logging',true);
```

To apply the model override object settings, use:

```
set_param(ex_bus_logging,'DataLoggingOverride', mi);
```

Simulink saves the settings when you save the model.

Log a Subset of Signals

For a simple model with a limited number of logged signals, you could create an empty `Simulink.SimulationData.ModelDataLogInfo` object. Then create `Simulink.SimulationData.SignalLoggingInfo` objects for each of the signals that you want to log, and assign those objects to the model logging information object.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdlref_counter_bus')));
mdl = 'ex_bus_logging';
blk = 'ex_bus_logging/IncrementBusCreator';
blkPort = 1;

load_system(mdl);

ov = Simulink.SimulationData.ModelLoggingInfo(mdl);

so = Simulink.SimulationData.SignalLoggingInfo(blk,blkPort);

ov.Signals(1) = so;
```

```
% apply this object so the model
set_param(mdl,'DataLoggingOverride',ov);

% Simulate
sim(mdl);

% observe that only the signal
topOut
```

To apply the model override object settings, use:

```
set_param(mdl, 'DataLoggingOverride', ov);
```

Simulink saves the settings when you save the model.

For a model that uses model referencing, or that is complex, to specify a subset of logged signals to log, consider using the `findSignal` method with a `Simulink.SimulationData.ModelLoggingInfo` object. For example, to log only one signal from the referenced model instance referenced by:

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
 'examples', 'ex_mdlref_counter_bus')));

mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
             'ex_bus_logging');
pos = mi.findSignal({'ex_bus_logging/CounterA' ...
                     'ex_mdlref_counter_bus/Bus Creator'}, 1)



pos =

     4

for idx=1:length(mi.Signals)
  mi.Signals(idx).LoggingInfo.DataLogging = (idx == pos);
end
```

To apply the model override object settings, use:

```
set_param(ex_bus_logging,'DataLoggingOverride', mi);
```

Simulink saves the settings when you save the model.

Override Other Signal Logging Properties

In addition to overriding the setting for the `DataLogging` property for a signal, you can override other signal logging properties, such as decimation.

Use `Simulink.SimulationData.LoggingInfo` properties to override signal logging properties. The following example shows how to set the decimation override settings.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
 'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
```

```
'examples', 'ex_mdlref_counter_bus')));
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel...
    ('ex_bus_logging');
pos = mi.findSignal({'ex_bus_logging/CounterA' ...
                     'ex_mdlref_counter_bus/Bus Creator'}, 1);
mi.Signals(pos).LoggingInfo.DecimateData = true;
mi.Signals(pos).LoggingInfo.Decimation = 2;
```

To apply the model override object settings, use:

```
set_param(ex_bus_logging,'DataLoggingOverride', mi);
```

Simulink saves the settings when you save the model.

See Also

Related Examples

- “Configure a Signal for Logging” on page 72-44
- “Export Signal Data Using Signal Logging” on page 72-41
- “Viewing Signals in Model Reference Instances”

More About

- “Override Signal Logging Settings with Signal Logging Selector” on page 72-58

View and Access Signal Logging Data

In this section...

You can view logged signal data during simulation, using the Simulation Data Inspector, or for paused or stopped simulations, using other visualization interfaces. See “Decide How to Visualize Simulation Data” on page 30-2.

Alternatively, you can access signal logging data programmatically, using MATLAB commands, as described in this topic.

Tip If you do not see logging data for a signal that you marked in the model for signal logging, check the logging configuration. Use the Signal Logging Selector to enable logging for a signal whose logging is overridden. For details, see “View the Signal Logging Configuration” on page 72-49 and “Override Signal Logging Settings” on page 72-57.

Signal Logging Object

Simulink saves signal logging data in a `Simulink.SimulationData.Dataset` object, which is a MATLAB workspace variable. The default name of the signal logging variable is `logsout`. You can change the variable name. For details, see “Specify a Name for Signal Logging Data” on page 72-56.

You can specify whether you want the data for individual signals in a dataset to use MATLAB `timeseries` or `timetable` elements. Set the **Dataset signal format** configuration parameter (for details, see “Dataset signal format”).

Releases earlier than R2016a also supported a `ModelDataLogs` format. For details, see “Migrate Scripts That Use Legacy ModelDataLogs API” on page 72-54.

Access Data Programmatically

You can use the `Simulink.SimulationData.Dataset` API to access signal logging data programmatically. To access `Dataset` object elements, use indexing with curly braces. For example, you can access the first element of the `topOut` signal logging `Dataset` object using index `1`. This example is based on the use of the default setting of `timeseries` for the dataset elements. For details about `timeseries` and `timetable` format data, see “Dataset signal format”.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_bus_logging')));
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_mdlref_counter_bus')));
sim('ex_bus_logging')
topOut
```

```
Simulink.SimulationData.Dataset 'topOut' with 4 elements

                      Name          BlockPath
```

```
    1  [1x1 Signal]      COUNTERBUS    ex_bus_logging/COUNTERBUSCreator
    2  [1x1 Signal]      OUTPUTBUS     ex_bus_logging/CounterA
    3  [1x1 Signal]      INCREMENTBUS  ex_bus_logging/IncrementBusCreator
    4  [1x1 Signal]      inner_bus     ...erA|ex_mdlref_counter_bus/Bus Creator

  - Use braces { } to access, modify, or add elements using index.
```

```
element1 = topOut{1}
```

```
element1 =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
              Name: 'COUNTERBUS'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [1x1 struct]


  Methods, Superclasses
```

```
element1.Values
```

```
ans =

      data: [1x1 timeseries]
    limits: [1x1 struct]
```

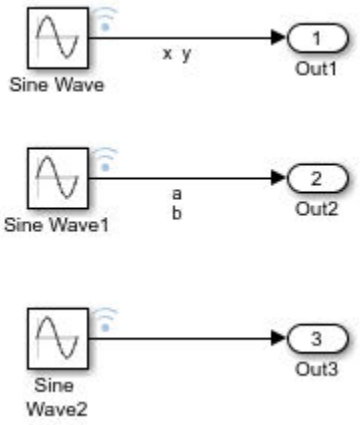
To search for specific elements in a `Dataset` object, use the `find` method. To return the names of the `Dataset` object elements, use the `getNames` method.

Tip To call a function on each specified MATLAB `timeseries` object, you can use the `Simulink.SimulationData.forEachTimeseries` function. For example, you can use this function to make it easy to resample every element of a structure of `timeseries` objects obtained by logging a bus signal.

Handling Spaces and Newlines in Logged Names

This example shows three signals that illustrate how signal logging names:

- A signal with a name that contains a space
- A signal with a name that contains a newline
- An unnamed signal that originates on a block with a name that contains a newline



Simulate the model and then look at the signal logging results in the `logsout` variable. You can see that the names in the `Dataset` object use a space where the signal name contained a space and a newline where the name contained a newline. The unnamed signal has an empty character array as its name.

```
logsout
```

```
logsout =

Simulink.SimulationData.Dataset 'logsout' with 3 elements

                        Name  BlockPath
                        ____  _____________________________________
    1  [1x1 Signal]     x y   ex_signal_names_with_spaces/Sine Wave
    2  [1x1 Signal]     a b   ex_signal_names_with_spaces/Sine Wave1
    3  [1x1 Signal]     ''    ex_signal_names_with_spaces/Sine Wave2

  - Use braces { } to access, modify, or add elements using index.
```

You can access a signal with a name that contains a space by name or by index. You just need to include the space in the name you pass to the `getElement` function. To access a signal with a name that contains a newline, use the index.

```
>> logsout{2}
```

```
ans =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
              Name: 'a↵b'
    PropagatedName: ''
         BlockPath: [1×1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [1×1 timeseries]
```

Access Logged Signal Data in ModelDataLogs Format

Before R2016a, you could log signals in `ModelDataLogs` 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.

For more information, see `Simulink.ModelDataLogs`.

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Dataset` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.forEachTimeseries` | `find` | `get` | `getElementNames` | `numElements` | `setElement`

Related Examples

- “Save Run-Time Data from Simulation”
- “Export Signal Data Using Signal Logging” on page 72-41
- “Migrate Scripts That Use Legacy ModelDataLogs API” on page 72-54
- “Log Signals in For Each Subsystems” on page 72-71

Log Signals in For Each Subsystems

In this section...

The approach you use to log data for a signal in a For Each subsystem depends on whether the signal is a:

- Nonbus signal — Log directly in a For Each subsystem
- A bus or array of buses signal — Use one of these approaches:
  - Use a Bus Selector block to select the signals you want to log and mark those signals for signal logging. This approach works well for many models.
  - Attach the signal to an Outport block and log the signal outside of the For Each subsystem. Use this approach when you want to log a whole bus signal, and that bus signal includes many bus element signals.

Note You cannot log bus signals directly in a For Each subsystem.

You cannot log a signal inside a referenced model that is inside a For Each subsystem if either of these conditions exists:

- The For Each subsystem is in a model simulating in rapid accelerator mode.
- The For Each subsystem itself is in a model referenced by a Model block in accelerator mode.

The data for each logged signal in a For Each subsystem is saved in a separate `Dataset` element as a `Simulink.SimulationData.Signal` object. The format of the logged signal data depends on how you set the **Dataset signal format** configuration parameter:

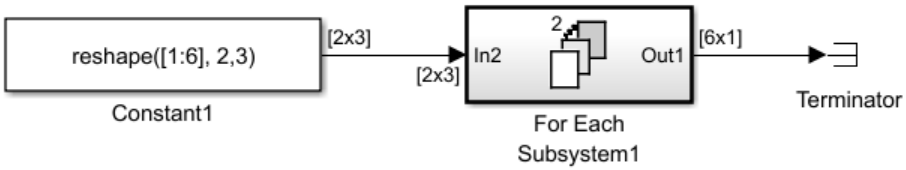
- If the setting is `timeseries`, then each signal object contains an array of MATLAB `timeseries` objects. The array keeps the data from different For Each iteration separate.
- If the setting is `timetable`, then each signal object contains a cell array of MATLAB `timetable` objects. The dimensions of this array match the number of For Each iterations. For example, if the For Each subsystem has three iterations, then the logged data has a `3x1` array of `timeseries` or `timetable` objects. For nested For Each subsystems, each layer of nesting adds another dimension to the logged data.

Log Signal in Nested For Each Subsystem

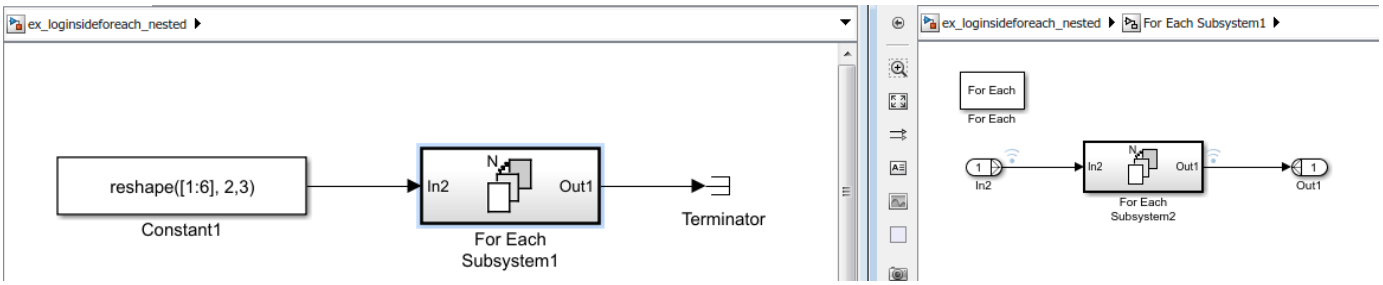
This example logs a signal in a nested For Each subsystem.

Open the `ex_loginsideforeach_nested` model.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_loginsideforeach_nested.slx')))
```



In the Simulink Editor, open the `For Each Subsystem1` block, and inside that subsystem, open the `For Each Subsystem2` block.



Simulate the model and examine the signal logging data for the first iteration of the top subsystem and the third iteration of the bottom subsystem. The `2x3 timeseries` results from two iterations at the first For Each level and three iterations at the second (nested) level

```
sim('ex_loginsideforeach_nested');
logsout.get('nestedDelay')
```

```
ans =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
  struct with fields:

              Name: 'nestedDelay'
    PropagatedName: ''
         BlockPath: [1×1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [2×3 timeseries]
```

Return the values of the `nestedDelay` object.

```
logsout.get('nestedDelay').Values(1,3)
```

```
timeseries

  Common Properties:
            Name: 'nestedDelay'
            Time: [5x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
```

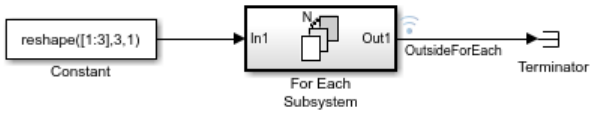
```
    Data: [1x1x5 double]
DataInfo: [1x1 tsdata.datametadata]
```

Log Bus Signals in For Each Subsystem

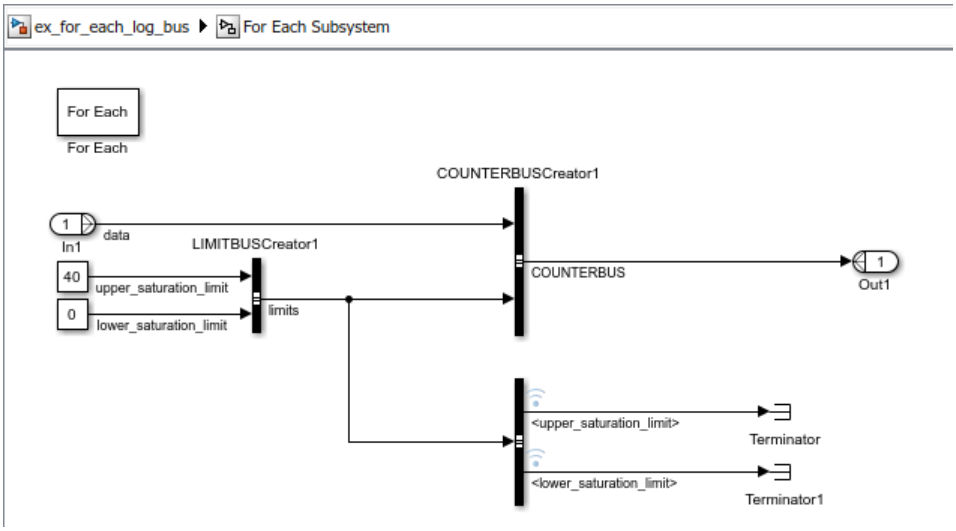
This example logs a two bus signal in a For Each subsystem. For one bus signal, you use a Bus Selector block and then log each selected signal. For the other bus signal, you use Outport blocks and log outside of the For Each subsystem.

Open the `ex_for_each_log_bus` model.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_for_each_log_bus.slx')))
```



In the Simulink Editor, open the `For Each Subsystem` block.



To log the signals in the `limits` bus signal, the signal is branched to a Bus Selector block, and each of the bus element signals is marked for signal logging.

To log the whole `COUNTERBUS` signal, the bus signal is connected to an Outport block. The output signal from the For Each subsystem is marked for signal logging. To have the bus signal cross the subsystem boundary, the Bus Creator block that creates the `COUNTERBUS` signal has the **Output data type** parameter set to `Bus: COUNTERBUS` and **Output as nonvirtual bus** check box selected.

Simulate the model and examine the signal logging output. Focus on one of the bus element signals logged inside the For Each subsystem and on the bus signal logged outside of the For Each subsystem.

```
sim('ex_for_each_log_bus');
logsout
```

```
Simulink.SimulationData.Dataset 'logsout' with 3 elements

                       Name                      BlockPath
                       ____________________      ____________________________________________
    1  [1x1 Signal]    OutsideForEach            ex_for_each_log_bus/For Each Subsystem
    2  [1x1 Signal]    <lower_saturation_limit>  ...g_bus/For Each Subsystem/Bus Selector
    3  [1x1 Signal]    <upper_saturation_limit>  ...g_bus/For Each Subsystem/Bus Selector

  - Use braces { } to access, modify, or add elements using index.
```

Return the values of the `lower_saturation_limit` object.

```
logsout{2}.Values
```

```
3×1 timeseries array with properties:

    Events
    Name
    UserData
    Data
    DataInfo
    Time
    TimeInfo
    Quality
    QualityInfo
    IsTimeFirst
    TreatNaNasMissing
    Length
```

Return the values of the `OutsideForEach` object.

```
logsout{1}.Values
```

```
 ans =

  3×1 struct array with fields:

    data
    limits
```

If the `Dataset` signal format is `timetable`, then the output is a cell array of `timetable` objects. For example:

```
out = sim('ex_for_each_log_bus','DatasetSignalFormat','timetable');
out.logsout{2}.Values
```

```
ans =
 3x1 cell array

  {11x1 timetable}
  {11x1 timetable}
  {11x1 timetable}
```

See Also

Blocks
For Each Subsystem

Functions
`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.Signal`

Related Examples

- “Export Signal Data Using Signal Logging” on page 72-41

State Information

In this section...

Simulation State Information

Some blocks maintain state information that they use during simulation. For example, the state information for a Unit Delay block is the output signal value from the previous simulation step. The block uses the state information to calculate the output value for the current simulation step.

Some examples of how saved state information is used include:

- Stopping a simulation for a model and using the saved state information as input when you restart the simulation.
- Simulating one model and using the saved state information as input for the simulation of another model that builds on the results of the first model.
- Examining changes in state information throughout a simulation.

Types of State Information

You can save these kinds of state information.

Type of State Information	Description	Configuration Parameters in Data Import/Export Pane
States for each simulation step	State information of blocks (referred to as partial state data) at each time step of a simulation	**States**
Final state	State information of blocks at the end of the simulation	**Final states**
Final state with `ModelOperatingPoint`	Final state with a `ModelOperatingPoint` object that captures additional internal information that Simulink uses during simulation	**Final States** and **Save final operating point**

`ModelOperatingPoint` provides more complete final simulation state information than final states information by itself does. However, if the requirements and limitations of using `ModelOperatingPoint` do not meet your modeling requirements, save final state information without `ModelOperatingPoint`.

Comparison of Operating Point and Final State Logging

Characteristic	Final State	Final State with Operating Point
Simulation mode	Supports all simulation modes	Normal or Accelerator.
Model reference	"State Information for Referenced Models" on page 72-79	See "Model Referencing" on page 25-47.
Resumed simulation	Not supported	Supported.
Saved state data	Only logged states — the continuous and discrete states of blocks — which are a subset of the complete simulation state of the model User data, run-time parameters, or logs of the model not saved	Complete state information. Does not save user data, run-time parameters, or logs of the model.
Block output	User data, run-time parameters, or logs of the model not saved	Simulink tries to save the output of a block as part of a `ModelOperatingPoint` object even if S-functions declare that no `ModelOperatingPoint` objects exist in the block. If the block output is of custom type, Simulink displays an error.
Readability	Use structure with time format for best readability	To examine a simplified view of the data, consider using the `loggedStates` property of the `Simulink.op.ModelOperatingPoint` class.
Restoring state data	Can save and restore in different simulation modes. If logged state information is not sufficient, you can obtain different results in the normal mode and the accelerator mode.	Cannot save in normal mode and restore in accelerator mode, or conversely save in accelerator mode and restore in normal mode.
Restoring multiple states	You can initialize only one out of multiple logged states in the model.	You restore all states in the model. You cannot load a subset of states.

Characteristic	Final State	Final State with Operating Point
Structural changes	You can make structural changes between simulation and restoring the simulation.	You cannot make structural changes to the model between when you save the `ModelOperatingPoint` object and when you restore the simulation using the `ModelOperatingPoint` object. For example, you cannot add or remove a block after saving the `ModelOperatingPoint` object without repeating the simulation and saving the new `ModelOperatingPoint` object.
Input to `model` function	To input to model function, use `Array` format with non-complex data of type `double`.	You cannot input the `ModelOperatingPoint` object to model function.
Code generation	Supported	Not supported.

For both `ModelOperatingPoint` and final state logging, Simulink saves state information at one of these points:

- At the final time step
- At the execution time at which the simulation paused or stopped

For additional information about `ModelOperatingPoint`, see “Limitations of Saving and Restoring Operating Point” on page 25-46.

Format for State Information Saved Without Operating Point

If you do not use the `ModelOperatingPoint` for saving state information, then use **Configuration Parameters** > **Data Import/Export** > **Format** to specify the data format for the saved state information.

You can set **Format** to:

- `Dataset` (default)
- `Array`
- `Structure`
- `Structure with time`

The default setting for the **Format** parameter is `Dataset`. The `Dataset` format:

- Stores logged data in `timeseries` or `timetable` objects. You can work with data saved in a `timeseries` or `timetable` object in MATLAB without a Simulink license.
- Supports logging multiple data values for a given time step, which can be required for logging data in a For Iterator Subsystem, a While Iterator Subsystem, and Stateflow.
- Does not support rapid accelerator simulation, logging states information inside a function-call subsystem, or code generation.

Signal logging always uses the `Dataset` format. Logging states data using the `Dataset` format allows you to post-process simulation data without writing custom code for different types of logged data. When you log states using the `Dataset` format, the data also automatically streams to the Simulation Data Inspector during simulation.

The `Array` option for the **Format** parameter exists for backward compatibility with models developed in earlier releases, when Simulink supported only the `Array` format for logging state information. The order of signals within the array depends on the block sorted order, which can change from one simulation to another when you change any of the following:

- The model (even without changing the signal)
- The simulation mode
- The code generation mode

The variation in signal order can present challenges when post-processing the logged data.

The `Structure` and `Structure with time` formats are useful when using state information to initialize a model for simulation, allowing you to:

- Associate initial state values directly with the full path name to the states. This association eliminates errors that can occur if Simulink reorders the states, but the order of the initial state array does not change correspondingly.
- Assign a different data type to the initial value of each state.
- Initialize only a subset of the states.

State Information for Referenced Models

When Simulink saves states in the structure or structure-with-time format, it adds an `inReferencedModel` subfield to the `signals` field of the structure. The value of this additional subfield is true (`1`) if the `signals` field records the final state of a block that resides in the referenced model. For example:

```
xout.signals(1)
```

```
ans =

               values: [101x1 double]
           dimensions: 1
                label: 'DSTATE'
            blockName: [1x66 char]
    inReferencedModel: 1
```

If the signals field records a referenced model state, its `blockName` subfield contains a compound path of a top model path and a referenced model path. The top model path is the path from the model root to the Model block that references the referenced model. The referenced model path is the path from the referenced model root to the block whose state the `signals` field records. The compound path uses a | character to separate the top and referenced model paths. For example:

```
>> xout.signals(1).blockName

ans =

sldemo_mdlref_basic/CounterA|sldemo_mdlref_counter/Previous Output
```

See Also

Classes
`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.Signal`

Related Examples

- “Save State Information” on page 72-81
- “Load State Information” on page 70-70

Save State Information

In this section...

Save State Information for Each Simulation Step

You can save state information for logged states for each simulation step during a simulation. That level of state information can be helpful for debugging.

1. Select the **Configuration Parameters** > **Data Import/Export** > **States** check box.
2. In the **States** edit box, you can specify a different variable for the state information, if you do not want to use the default `xout` variable.
3. Also in the **Data Import/Export** pane, set the **Format** parameter to `Dataset`, `Structure`, or `Structure with time`, unless you use array format for compatibility with a legacy model.

   `Dataset` format does not support:

   - Logging states information inside a function-call subsystem
   - Rapid accelerator simulation mode
   - Code generation
4. Click **Apply**.
5. Simulate the model.

Save Partial Final State Information

To save just the logged states (the continuous and discrete states of blocks):

1. Select the **Configuration Parameters** > **Data Import/Export** > **Final states** check box.
2. In the **Final states** edit box, you can specify a different variable for the state information, if you do not want to use the default `xFinal` variable.
3. Clear the **Save final operating point** parameter.
4. Set the **Format** parameter to `Dataset`, `Structure`, or `Structure with time`.
5. Click **Apply**.
6. Simulate the model.

Examine State Information Saved Without the Operating Point

If you enable the **Configuration Parameters** > **Data Import/Export** > **Final states** or **States** parameters, Simulink saves the state information in the format that you specify with the **Format** parameter. The default variable for **Final state** information is `xFinal`, and the variable for state information for States information is `xout`.

If a model has no states saved, then `xFinal` and `xout` are empty variables. To determine whether a model has states saved, use the `isempty(xout)` command.

Final State Information in Dataset Format

For example, suppose that you saved final state information in `Dataset` format, and use the default `xFinal` variable for the saved state information.

```
xFinal
```

```
xFinal =

Simulink.SimulationData.Dataset 'xFinal' with 2 elements

                        Name    BlockPath
                        ______  _________
    1  [1x1 State]      CSTATE  vdp/x1
    2  [1x1 State]      DSTATE  vdp/x2

  - Use braces { } to access, modify, or add elements using index.
```

Examine the first element of the state data set.

```
xFinal{1}
```

```
ans =

  Simulink.SimulationData.State
  Package: Simulink.SimulationData

  Properties:
         Name: 'CSTATE'
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
        Label: CSTATE
       Values: [1x1 timeseries]
```

Final State Information in Structure with Time Format

For example, suppose that you saved final state information in a structure with time format, and use the default `xFinal` variable for the saved state information.

To find the simulation time and number of states in the `vdp` model, enter the `xFinal` variable.

```
xFinal
```

```
xFinal =

       time: 20
    signals: [1x2 struct]
```

In this case, the simulation time is 20 and there are two states. To examine the first state, use this command.

```
xFinal.signals(1)
```

```
ans =

               values: 2.0108
```

```
          dimensions: 1
               label: 'CSTATE'
           blockName: 'vdp/x1'
           stateName: ''
    inReferencedModel: 0
```

The `values` and `blockName` fields of the first state structure show that the final value for the output signal of the x1 block is 2.018.

Note If you write a script to analyze state information, use a combination of `label` and `blockName` values to identify a specific state uniquely. Do not rely on the order of the states.

Save Final State Information with Operating Point

To save complete state information, save the `ModelOperatingPoint` object for a simulation.

1. Select the **Configuration Parameters** > **Data Import/Export** > **Final states** check box.
2. Also in the Data Import/Export pane, select the **Save final operating point** parameter.
3. In the edit box next to the **Save final operating point** parameter, enter a variable name for the `ModelOperatingPoint` object and click **Apply**.
4. Simulate the model.

For more information about using the operating point, see “Save and Restore Simulation Operating Point” on page 25-41.

See Also

Related Examples

- “Load State Information” on page 70-70

More About

- “State Information” on page 72-76

Working with Data Stores

Data Store Basics

A *data store* is a repository to which you can write data, and from which you can read data, without having to connect an input or output signal directly to the data store. Data stores are accessible across model levels, so subsystems and referenced models can use data stores to share data without using I/O ports.

When to Use a Data Store

Data stores can be useful when multiple signals at different levels of a model need the same global values, and connecting all the signals explicitly would clutter the model unacceptably or take too long to be feasible. Data stores are analogous to global variables in programs, and have similar advantages and disadvantages, such as making verification more difficult.

To share data between the instances of a reusable algorithm (for example, a subsystem in a custom library or a reusable referenced model), you can use a data store. For more information about data sharing for a reusable referenced model, see “Share Data Among Referenced Model Instances” on page 8-34.

Data Stores and Software Verification

Data stores can have significant effects on software verification, especially in the area of data coupling and control. Models and subsystems that use only input and output ports to pass data result in clean, well-specified, and easily verifiable interfaces in the generated code.

Data stores, like any type of global data, make verification more difficult. If your development process includes software verification, consider planning for the effect of data stores early in the design process.

For more information, see RTCA DO-331, “Model-Based Development and Verification Supplement to DO-178C and DO-278A,” Section MB.6.3.3.b.

Goto and From Blocks as a Signal Routing Alternative

In some cases, you may be able to use a simpler technique, Goto blocks and From blocks, to obtain results similar to those provided by data stores. The principal disadvantage of data Goto/From links is that they generally are not accessible across nonvirtual subsystem boundaries, while an appropriately configured data store can be accessed anywhere. See the Goto and From block reference pages for more information about Goto/From links.

Local and Global Data Stores

You can define two types of data stores:

- A *local data store* is accessible from anywhere in the model hierarchy that is at or below the level at which you define the data store, except from referenced models. You can define a local data store graphically in a model or by creating a model workspace signal object (`Simulink.Signal`).
- A *global data store* is accessible from throughout the model hierarchy, including from referenced models. Define a global data store only in the MATLAB base workspace, using a signal object. The only type of data store that a referenced model can access is a global data store.

In general, locate a data store at the lowest level in the model that allows access to the data store by all the parts of the model that need that access. Some examples of local and global data stores appear in “Data Store Examples” on page 73-10.

For information about using referenced models, see “Model References”.

Data Store Diagnostics

About Data Store Diagnostics

Simulink provides various run-time and compile-time diagnostics that you can use to help avoid problems with data stores. Diagnostics are available in the Model Configuration Parameters dialog box and the Data Store Memory block's parameters dialog box. The Simulink Model Advisor provides support by listing cases where data store errors are more likely because diagnostics are disabled.

Detecting Access Order Errors

You can use data store run-time diagnostics to detect unintended sequences of data store reads and writes that occur during simulation. You can apply these diagnostics to all data stores, or allow each Data Store Memory block to set its own value. The diagnostics are:

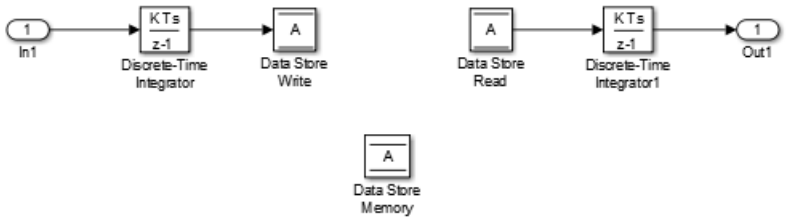
- **Detect read before write**
- **Detect write after read**
- **Detect write after write**

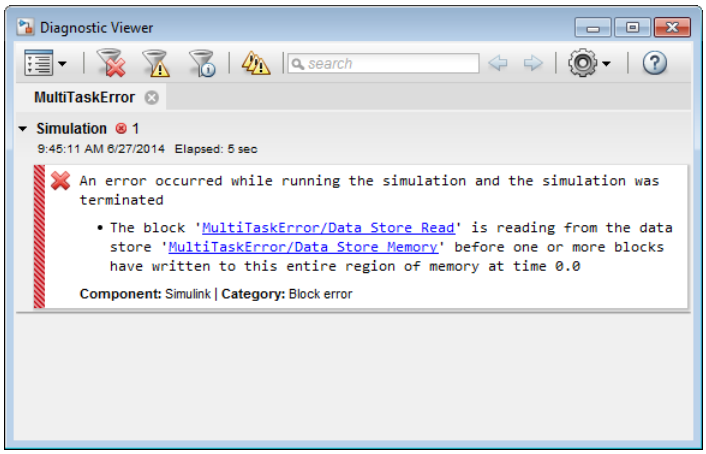
These diagnostics appear in the **Model Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block** pane, where each can have one of the following values:

- `Disable all` — Disables this diagnostic for all data stores accessed by the model.
- `Enable all as warnings` — Displays the diagnostic as a warning in the MATLAB Command Window.
- `Enable all as errors` — Halts the simulation and displays the diagnostic in an error dialog box.
- `Use local settings` — Allow each Data Store Memory block to set its own value for this diagnostic (default).

The same diagnostics also appear in each Data Store Memory block parameters dialog box **Diagnostics** tab. You can set each diagnostic to `none`, `warning`, or `error`. The value specified by an individual block takes effect only if the corresponding configuration parameter is `Use local settings`. See “Model Configuration Parameters: Data Validity Diagnostics” and the Data Store Memory documentation for more information.

The most conservative technique is to set all data store diagnostics to `Enable all as errors` in **Model Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block**. However, this setting is not best in all cases, because it can flag intended behavior as erroneous. For example, the next figure shows a model that uses block priorities to force the Data Store Read block to execute before the Data Store Write block:





An error occurred during simulation because the data store `A` is read from the Data Store Read block before the Data Store Write block updates the store. If the associated delay is intended, you can suppress the error by setting the global parameter **Detect read before write** to `Use local settings`, then setting that parameter to `none` in the **Diagnostics** pane of the Data Store Memory block dialog box. If you use this technique, set the parameter to `error` in all other Data Store Memory blocks aside from those that are to be intentionally excluded from the diagnostic.

Data Store Diagnostics and Models Referenced in Accelerator Mode

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`.

- **Detect read before write** (`ReadBeforeWriteMsg`)
- **Detect write after read** (`WriteAfterReadMsg`)
- **Detect write after write** (`WriteAfterWriteMsg`)

You can use the Model Advisor to identify models referenced in Accelerator mode for which Simulink ignores the configuration parameters listed above.

1 In the Simulink Editor, in the **Modeling** tab, click **Model Advisor**.

2. Select **By Task**.
3. Run the **Check diagnostic settings ignored during accelerated model reference simulation** check.

Data Store Diagnostics and the MATLAB Function Block

Diagnostics might be more conservative for data store memory used by MATLAB Function blocks. For example, if you pass arrays of data store memory to MATLAB functions, optimizations such as `A = foo(A)` might result in MATLAB marking the entire contents of the array as read or written, even though only some elements were accessed.

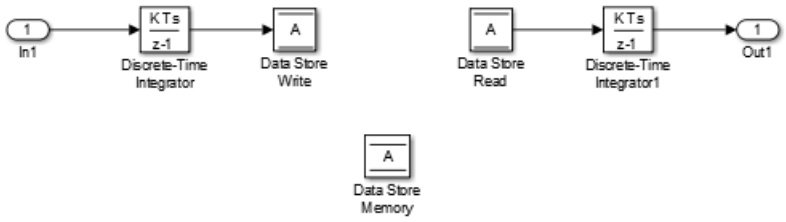
Detecting Multitasking Access Errors

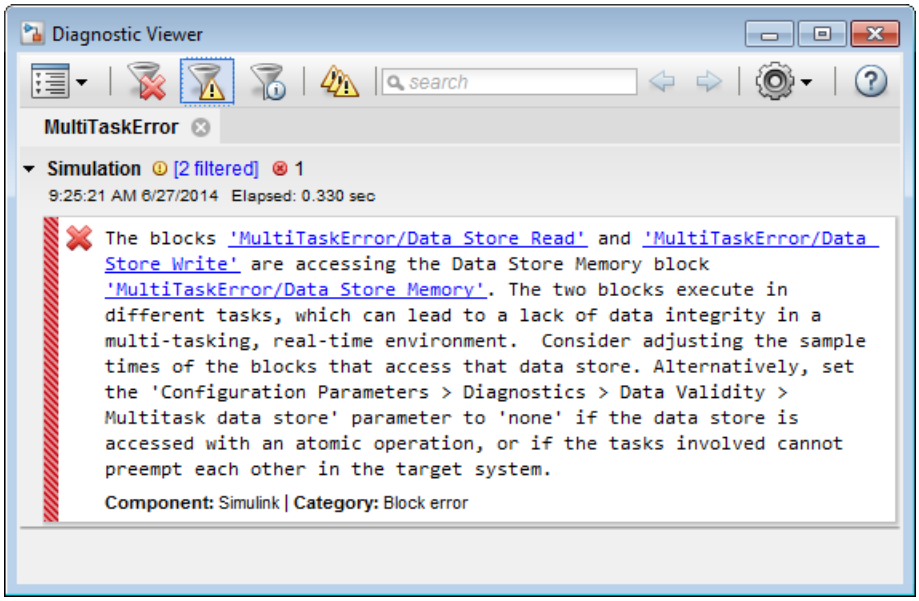
Data integrity may be compromised if a data store is read from in one task and written to in another task. For example, suppose that:

1. A task is writing to a data store.
2. A second task interrupts the first task.
3. The second task reads from that data store.

If the first task had only partly updated the data store when the second task interrupted, the resulting data in the data store is inconsistent. For example, if the value is a vector, some of its elements may have been written in the current time step, while the rest were written in the previous step. If the value is a multi-word, it may be left in an inconsistent state that is not even partly correct.

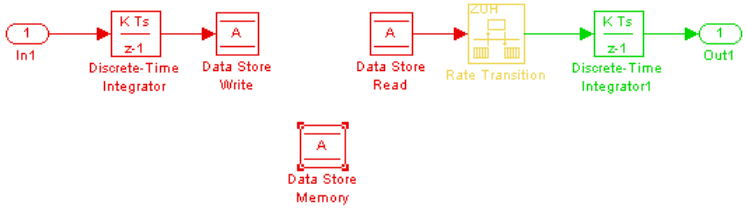
Unless you are certain that task preemption cannot cause data integrity problems, set the compile-time diagnostic **Model Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block > Multitask data store** to `warning` (the default) or `error`. This diagnostic flags any case of a data store that is read from and written to in different tasks. The next figure illustrates a problem detected by setting **Multitask data store** to `error`:





Since the data store A is written to in the fast task and read from in the slow task, an error is reported, with suggested remedy. This diagnostic is applicable even in the case that a data store read or write is inside of a conditional subsystem. Simulink correctly identifies the task that the block is executing within, and uses that task for the purpose of evaluating the diagnostic.

The next figure shows one solution to the problem shown above: place a rate transition block after the data store read, which previously accessed the data store at the slower rate.



With this change, the data store write can continue to occur at the faster rate. This may be important if that data store must be read at that faster rate elsewhere in the model.

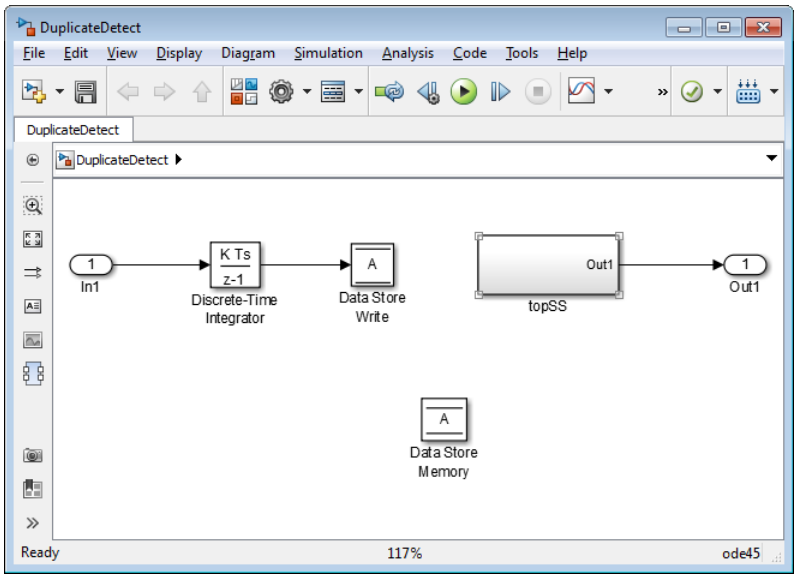
The **Multitask Data Store** diagnostic also applies to data store reads and writes in referenced models. If two different child models execute a data store's reads and writes in differing tasks, the error will be detected when Simulink compiles their common parent model.

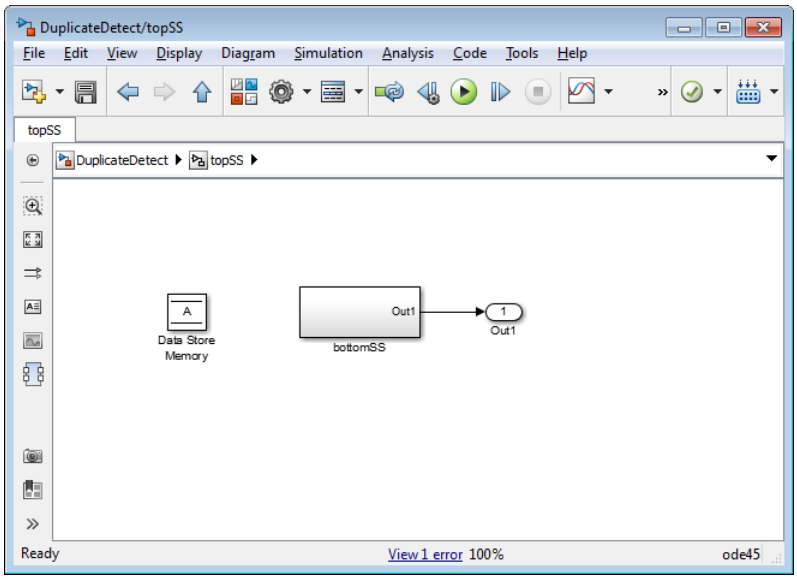
Detecting Duplicate Name Errors

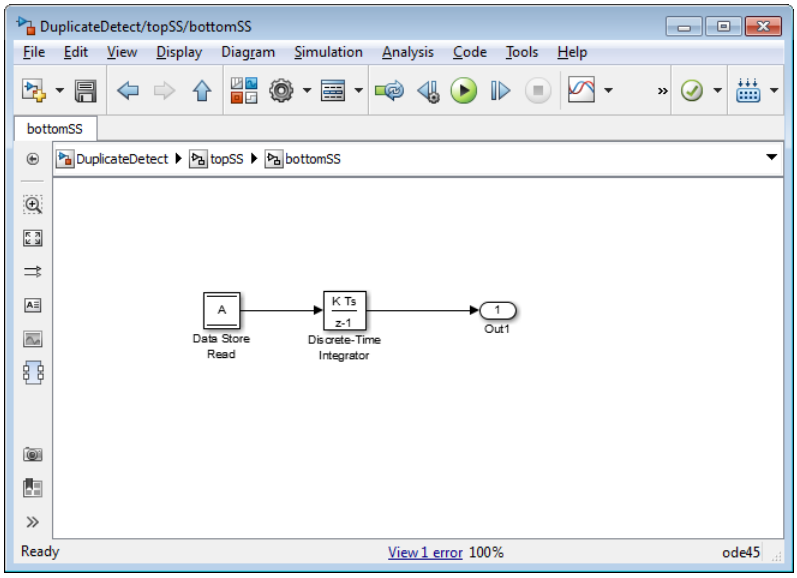
Data store errors can occur due to duplicate uses of a data store name within a model. For instance, data store shadowing occurs when two or more data store memories in different nested scopes have the same data store name. In this situation, the data store memory referenced by a data store read or write block at a low level may not be the intended store.

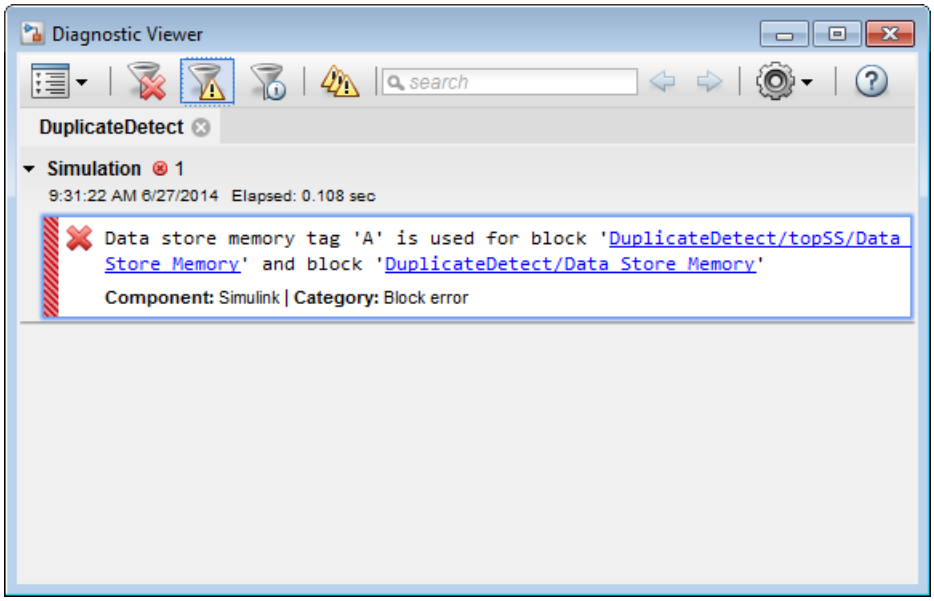
To prevent errors caused by duplicate data store names, set the compile-time diagnostic **Model Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block > Duplicate data store names** to `warning` or `error`. By default, the value of the diagnostic is `none`,

suppressing duplicate name detection. The next figure shows a problem detected by setting **Duplicate data store names** to `error`:









The data store read at the bottom level of a subsystem hierarchy refers to a data store named A, and two Data Store Memory blocks in the same model have that name, so an error is reported. This diagnostic guards against assuming that the data store read refers to the Data Store Memory block in the top level of the model. The read actually refers to the Data Store Memory block at the intermediate level, which is closer in scope to the Data Store Read block.

Data Store Diagnostics in the Model Advisor

The Model Advisor provides several diagnostics that you can use with data stores. See these sections for information about Model Advisor diagnostics for data stores:

“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues”

“Check data store block sample times for modeling errors”

“Check if read/write diagnostics are enabled for data store blocks”

Specify Initial Value for Data Store

In general, to specify an initial value for a data store, you can use the same techniques that you use for other blocks. See “Initialize Signals and Discrete States” on page 75-37.

With most blocks, you can take advantage of scalar expansion to minimize the effort of specifying an initial value for a nonscalar signal. When you specify a scalar initial value, each element in the signal uses that scalar.

However, when you set the **Dimensions** parameter to `-1` in a Data Store Memory block (the default), you cannot use scalar expansion. Instead, you must specify an initial value that has the same dimensions as the stored signal. To take advantage of scalar expansion of the initial value, set the **Dimensions** parameter to a specific value such as `[1 2]` or `[1 myDim]` (for symbolic dimensions).

See Also

Data Store Memory | Data Store Read | Data Store Write | `Simulink.Signal`

Related Examples

- “Data Stores in Generated Code” (Simulink Coder)
- “Model Global Data by Creating Data Stores” on page 73-10
- “Log Data Stores” on page 73-30
- “Data Objects” on page 67-58
- “Signal Basics” on page 75-2

Model Global Data by Creating Data Stores

In this section...

A *data store* is a repository to which you can write data, and from which you can read data, without having to connect an input or output signal directly to the data store. Data stores are accessible across model levels, so subsystems and referenced models can use data stores to share data without using I/O ports. To decide whether to use data stores, see “Data Store Basics” on page 73-2.

Data Store Examples

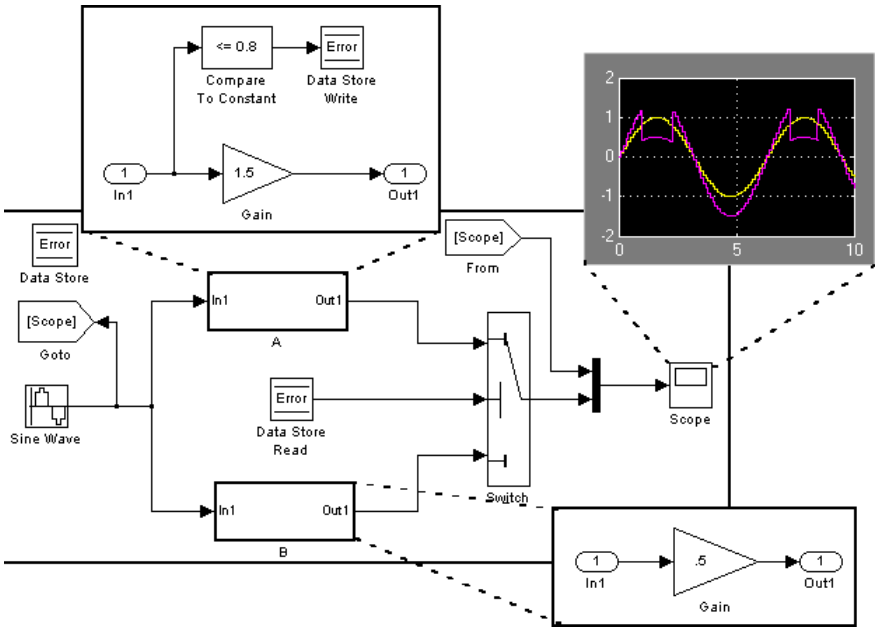
Overview

The following examples illustrate techniques for defining and accessing data stores. See “Order Data Store Access” on page 73-19 for techniques that control data store access over time, such as ensuring that a given data store is always written before it is read. See “Data Store Diagnostics” on page 73-3 for techniques you can use to help detect and correct potential data store errors without needing to run any simulations.

Note In addition to the following examples, see the sldemo_mdlref_dsm model, which shows how to use global data stores to share data among referenced models.

Local Data Store Example

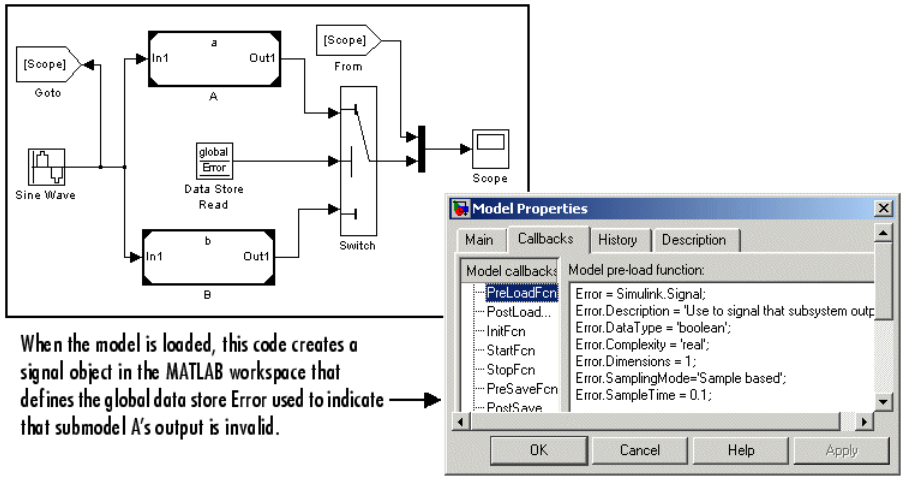
The following model illustrates creation and access of a local data store, which is visible only in a model or particular subsystem.



This model uses a data store to permit subsystem A to signal that its output is invalid. If subsystem A's output is invalid, the model uses the output of subsystem B.

Global Data Store Example

The following model replaces the subsystems of the previous example with functionally identical referenced models to illustrate use of a global data store to share data in a model reference hierarchy.



In this example, the top model uses a signal object in the MATLAB workspace to define the error data store. This is necessary because data stores are visible across model boundaries only if they are defined by signal objects in the MATLAB workspace.

Create and Apply Data Stores

Note To use buses and arrays of buses with data stores, perform *both* the following procedure and “Setting Up a Model to Use Data Stores with Buses and Arrays of Buses” on page 73-23.

The following is a general workflow for configuring data stores. You can perform the tasks in a different order, or separately from the rest, depending on how you use data stores.

1. Where applicable, plan your use of data stores to minimize their effect on software verification. For more information, see “Data Stores and Software Verification” on page 73-2.
2. Create data stores using the techniques described in “Data Stores with Data Store Memory Blocks” on page 73-13 or “Data Stores with Signal Objects” on page 73-16. For greater reliability, consider assigning rather than inheriting data store attributes, as described in “Specifying Data Store Memory Block Attributes” on page 73-13.
3. Add to the model Data Store Write and Data Store Read blocks to write to and read from the data stores, as described in “Access Data Stores with Simulink Blocks” on page 73-17.
4. Configure the model and the blocks that access each data store to avoid concurrency failures when reading and writing the data store, as described in “Order Data Store Access” on page 73-19.
5. Apply the techniques described in “Data Store Diagnostics” on page 73-3 as needed to prevent data store errors, or to diagnose them if they occur during simulation.
6. If you intend to generate code for your model, see “Data Stores in Generated Code” (Simulink Coder).

To create a data store, you create a Data Store Memory block or a `Simulink.Signal` object. The block or signal object represents the data store and specifies its properties. Every data store must have a unique name.

- A Data Store Memory block implements a local data store. See “Data Stores with Data Store Memory Blocks” on page 73-13.
- A `Simulink.Signal` object can act as a local or global data store. See “Data Stores with Signal Objects” on page 73-16.

Data stores implemented with Data Store Memory blocks:

- Support data store initialization
- Provide control of data store scope and options at specific levels in the model hierarchy
- Require a block to represent the data store
- Cannot be accessed within referenced models
- Cannot be in a subsystem that a For Each Subsystem block represents.

Data stores implemented with `Simulink.Signal` objects:

- Provide model-wide control of data store scope and options
- Do not require a block to represent the data store
- Can be accessed in referenced models, if the data store is global

Be careful not to equate local data stores with Data Store Memory blocks, and global data stores with `Simulink.Signal` objects. Either technique can define a local data store, and a signal object can define either a local or a global data store.

Data Stores with Data Store Memory Blocks

Creating the Data Store

To use a Data Store Memory block to define a data store, drag an instance of the block into the model at the topmost level from which you want the data store to be visible. The result is a local data store, which is not accessible within referenced models.

- To define a data store that is visible at every level within a given model, except within Model blocks, drag the Data Store Memory block into the root level of the model.
- To define a data store that is visible only within a particular subsystem and the subsystems that it contains, but not within Model blocks, drag the Data Store Memory block into the subsystem.

Once you have added the Data Store Memory block, use its parameters to define the data store's properties. The **Data store name** property specifies the name of the data store that the Data Store Write and Data Store Read blocks access. See Data Store Memory documentation for details.

You can specify data store properties beyond those definable with Data Store Memory block parameters by selecting the **Data store name must resolve to Simulink signal object** option and using a signal object as the data store name. See “Specifying Attributes Using a Signal Object” on page 73-14 for details.

Specifying Data Store Memory Block Attributes

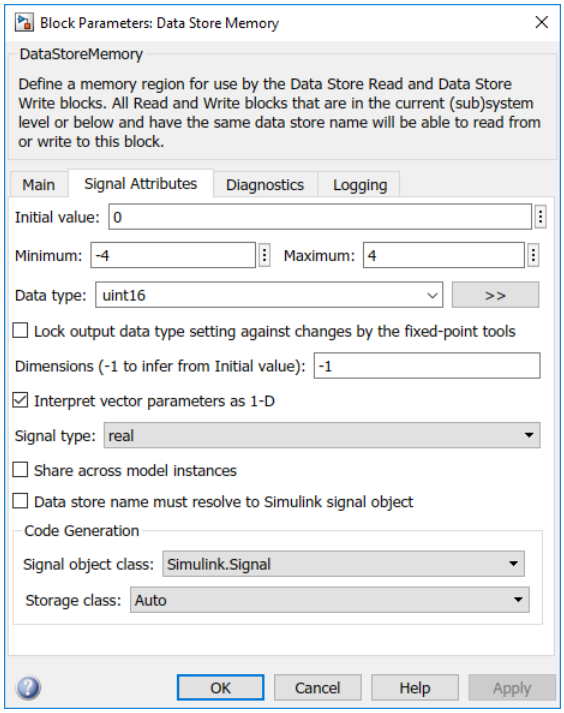
A Data Store Memory block can inherit three data attributes from its corresponding Data Store Read and Data Store Write blocks. The inheritable attributes are:

- Data type
- Complexity
- Sample time

However, allowing these attributes to be inherited can cause unexpected results that can be difficult to debug. To prevent such errors, use the Data Store Memory block dialog or a `Simulink.Signal` object to specify the attributes explicitly.

Specifying Attributes Using Block Parameters

You can use the Data Store Memory block dialog box or the Model Data Editor **Data Stores** tab (in the **Modeling** tab, click **Model Data Editor**) to specify the data type and complexity of a data store. In the next figure, the block dialog box sets the **Data type** to `uint16` and the **Signal type** to `real`.

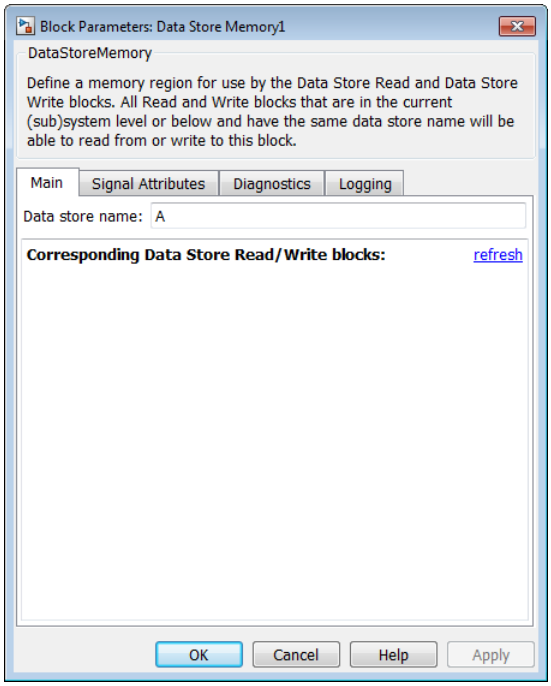


Specifying Attributes Using a Signal Object

You can use a `Simulink.Signal` object to specify data store attributes for a Data Store Memory block.

Tip To establish an implicit data store, as described in “Data Stores with Signal Objects” on page 73-16, use the same general approach as when you explicitly associate a signal object with a Data Store Memory block.

The next figure shows a Data Store Memory block that specifies resolution to a `Simulink.Signal` object, named `A`. To use a signal object for the data store, set **Data store name** to the name of the signal object. For compile-time checking, open the **Signal Attributes** tab and select the **Data store name must resolve to Simulink signal object** parameter.



Alternatively, on the Model Data Editor **Data Stores** tab (on the **Modeling** tab, click **Model Data Editor**), while editing the data store name, click the nearby action button ⋮ and select **Create and Resolve**. In the Create New Data dialog box, set **Value** to `Simulink.Signal`.

The signal object specifies values for all three data attributes that the data store would otherwise inherit. In this example, which defines a local data store, the `Simulink.Signal` object `A` has the following inherited properties: `DataType`, `Complexity`, and `SampleTime`.

```
A =

Simulink.Signal (handle)
         CoderInfo: [1x1 Simulink.CoderInfo]
       Description: ''
          DataType: 'auto'
               Min: []
               Max: []
              Unit: ''
        Dimensions: 1
    DimensionsMode: 'auto'
        Complexity: 'auto'
        SampleTime: -1
      InitialValue: 0
```

For more information about specifying signal object attributes for local and global data stores, see “Signal Object Attributes for Data Stores” on page 73-16.

Use Model Data Editor to Configure Data Store Memory Blocks in a List

Use the **Data Stores** tab in the Model Data Editor to configure the parameters of a Data Store Memory block. Use this technique to configure the data store without locating it in the model and to configure the data store together with other interface elements such as Inport and Outport blocks.

The Model Data Editor also shows you information for Data Store Read and Data Store Write blocks in the same list.

To open the Model Data Editor, in the **Modeling** tab, click **Model Data Editor**. For information about using the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Data Stores with Signal Objects

Creating the Data Store

To use a `Simulink.Signal` object to define a data store without using a Data Store Memory block, create the signal object in a workspace that is visible to every component that needs to access the data store. The name of the associated data store is the name of the signal object. You can use this name in Data Store Read and Data Store Write blocks, just as if it were the **Data store name** of a Data Store Memory block. Simulink creates an associated data store when you use the signal object for data storage.

Local and Global Data Stores

You can use a `Simulink.Signal` object to define either a local or a global data store.

- If you define the object in the MATLAB base workspace or a data dictionary, the result is a global data store, which is accessible in every model within Simulink, including all referenced models.
- If you create the object in a model workspace, the result is a local data store, which is accessible at every level in a model except any referenced models.

Signal Object Attributes for Data Stores

Those data store attributes that a signal object does not define have the same default values that they do in a Data Store Memory block. The property values of a signal object used as a data store have different requirements, depending on whether the data store is local or global.

Once you have created the object, set the properties of the signal object to the values that you want the corresponding data store properties to have. For example, the following commands define a data store named `Error` in the MATLAB base workspace:

```
Error = Simulink.Signal;
Error.Description = 'Use to signal that subsystem output is invalid';
Error.DataType = 'boolean';
Error.Complexity = 'real';
Error.Dimensions = 1;
Error.SampleTime = 0.1;
```

Attributes for Local Data Stores

For a local data store, for each parameter listed below, you can either set the value explicitly or you can have the data store inherit the value from the Data Store Write and Data Store Read blocks.

- `DataType`

- `Complexity`
- `SampleTime`

To define a local data store using a Data Store Memory block, you can use a signal object for the **Data store name** parameter. For compile-time checking, in the **Signal Attributes** tab, select the **Data store must resolve to Simulink signal object** parameter. The **Data store must resolve to Simulink signal object** parameter causes Simulink to display an error and stop compilation if Simulink cannot find the signal object or if the signal object properties are inconsistent with the signal object properties.

Attributes for Global Data Stores

The following table describes the parameter requirements for global data stores.

Parameter	Global Data Store Value
`DataType`	Must be set explicitly
`Complexity`	Must be set explicitly
`Dimensions`	Can be set or inherited
`SampleTime`	Can be set or inherited

Modify Attributes of Data Store Defined by Signal Object

You can use the Model Data Editor (in the **Modeling** tab, click **Model Data Editor**) to modify and inspect the attributes of data stores, Data Store Read, and Data Store Write blocks. On the **Data Stores** tab, to show the attributes of data stores that you define by using signal objects (such as `Simulink.Signal`), click the **Show/refresh additional information** button. Then, if a Data Store Read or Data Store Write block shown in the data table refers to a data store defined by a signal object, the table includes a row that corresponds to the object.

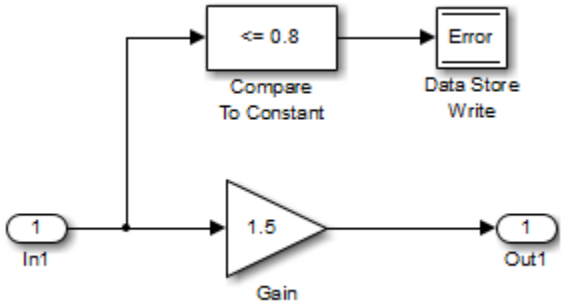
For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Access Data Stores with Simulink Blocks

Writing to a Data Store

To set the value of a data store at each time step:

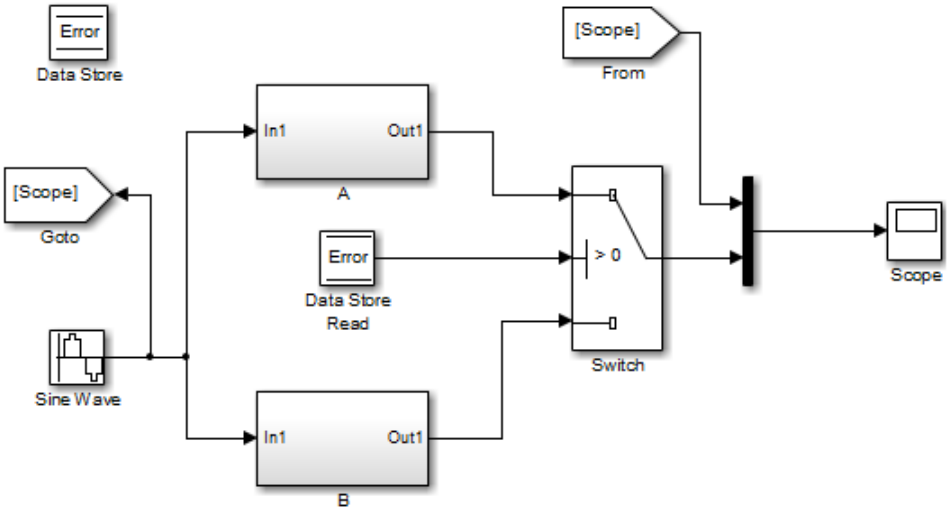
1. Create an instance of a Data Store Write block at the level of your model that computes the value.
2. Set the Data Store Write block **Data store name** parameter to the name of the data store to which you want it to write data.
3. Connect the output of the block that computes the value to the input of the Data Store Write block.



Reading from a Data Store

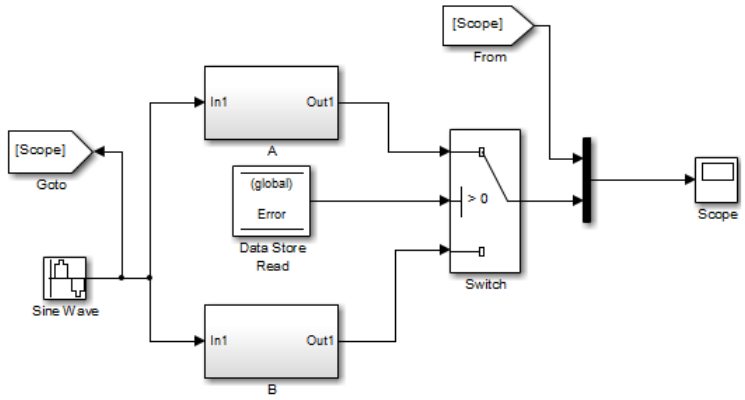
To get the value of a data store at each time step:

1. Create an instance of a Data Store Read block at the level of your model that needs the value.
2. Set the Data Store Read block **Data store name** parameter to the name of the data store from which you want it to read.
3. Connect the output of the Data Store Read block to the input of the block that needs the data store value.



Accessing a Global Data Store

When connected to a global data store (one that is defined by a signal object in the MATLAB workspace), a Data Store Read or Data Store Write block displays the word `global` above the data store name.



Order Data Store Access

About Data Store Access Order

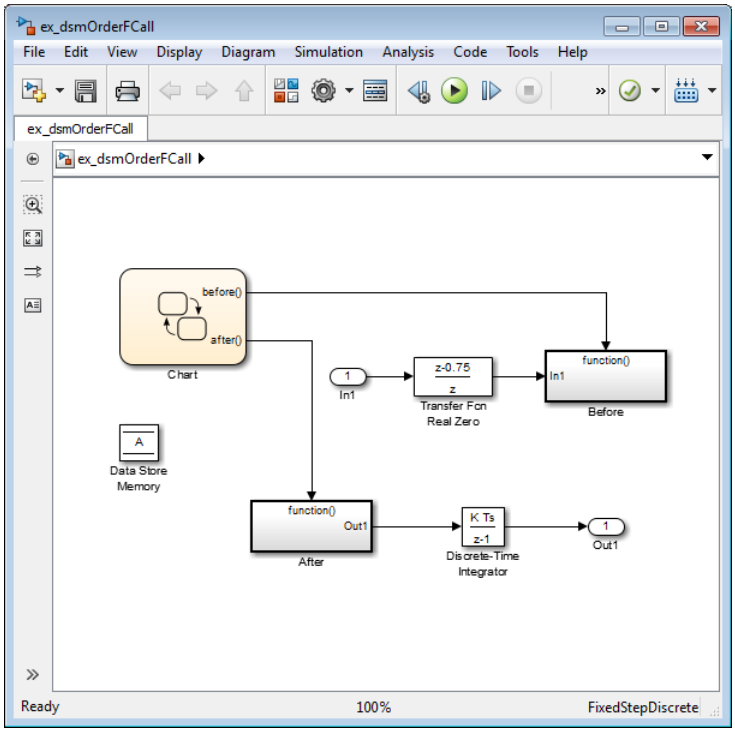
To obtain correct results from data stores, you must control the order of execution of the data store’s reads and writes. If a data store’s read occurs before its write, latency is introduced into the algorithm: the read obtains the value that was computed and stored in the previous time step, rather than the value computed and stored in the current time step.

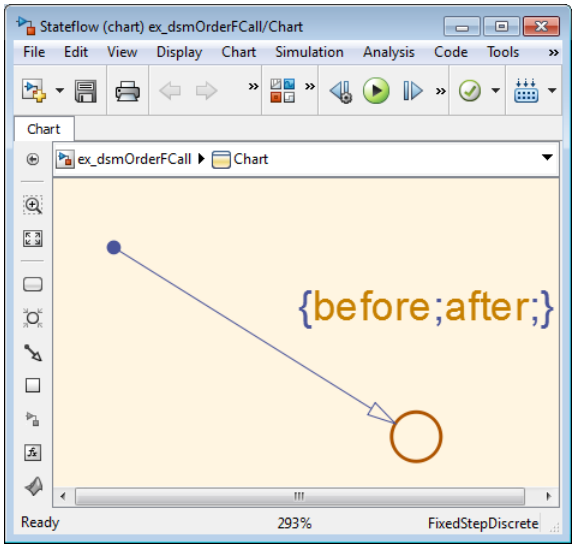
Such latency may cause the system to behave other than as designed, and in some cases may destabilize the system. Even if these problems do not occur, an uncontrolled access order could change from one release of Simulink to the next.

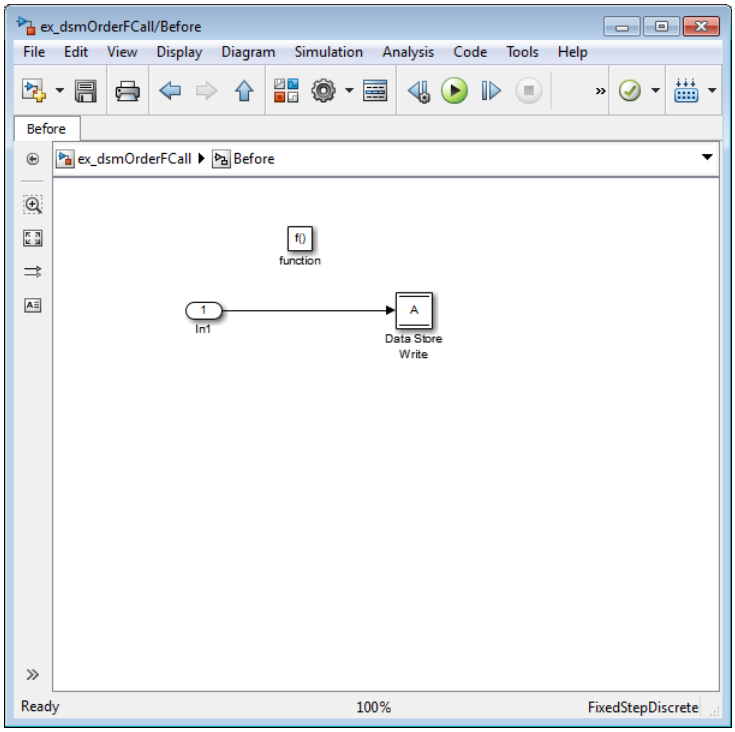
This section describes several strategies for explicitly controlling the order of execution of a data store’s reads and writes. See “Data Store Diagnostics” on page 73-3 for techniques you can use to detect and correct potential data store errors without running simulations.

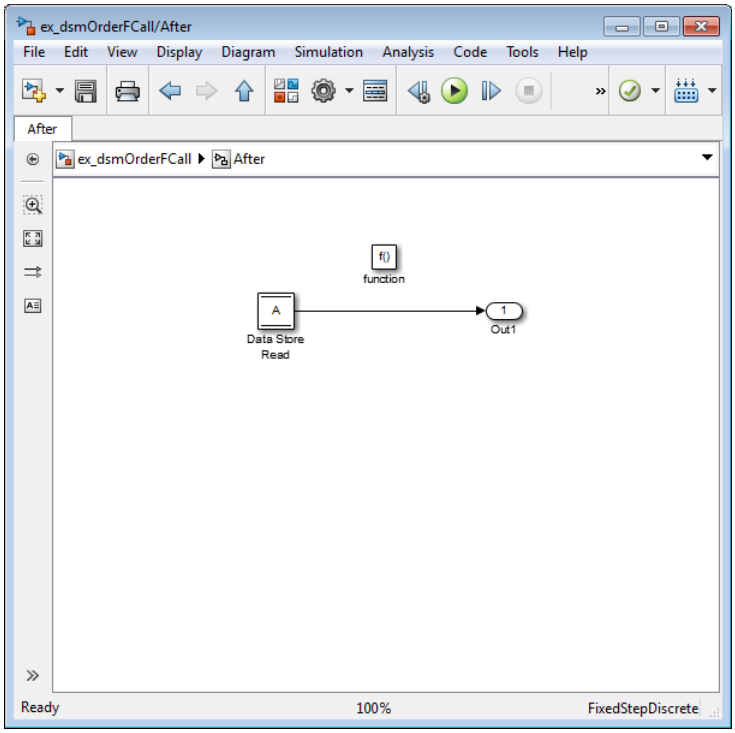
Ordering Access Using Function Call Subsystems

You can use function call subsystems to control the execution order of model components that access data stores. The next figure shows this technique:


ex_dsmOrderFCall
File Edit View Display Diagram Simulation Analysis Code Tools Help
ex_dsmOrderFCall
ex_dsmOrderFCall
before()
after()
Chart
1
In1
z-0.75
z
Transfer Fcn
Real Zero
function()
In1
Before
A
Data Store
Memory
function()
Out1
After
K Ts
z-1
Discrete-Time
Integrator
1
Out1
Ready
100%
FixedStepDiscrete


Stateflow (chart) ex_dsmOrderFCall/Chart
File Edit View Display Chart Simulation Analysis Code Tools
Chart
ex_dsmOrderFCall
Chart
{before;after;}
Ready
293%
FixedStepDiscrete

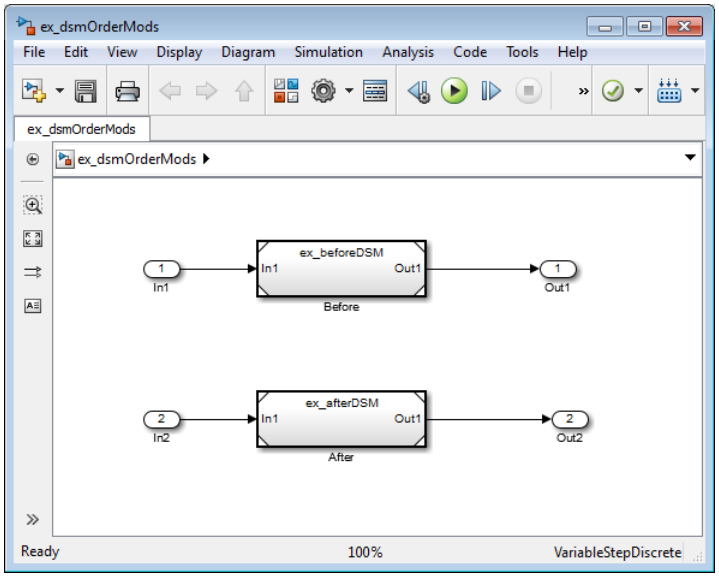


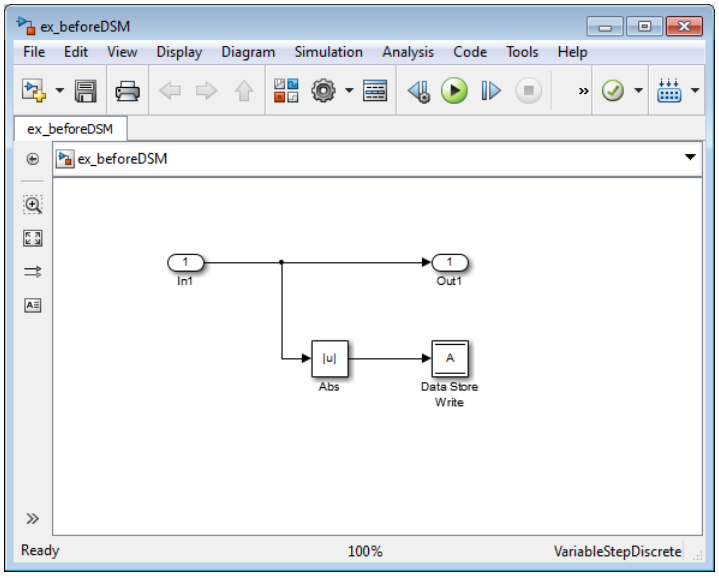


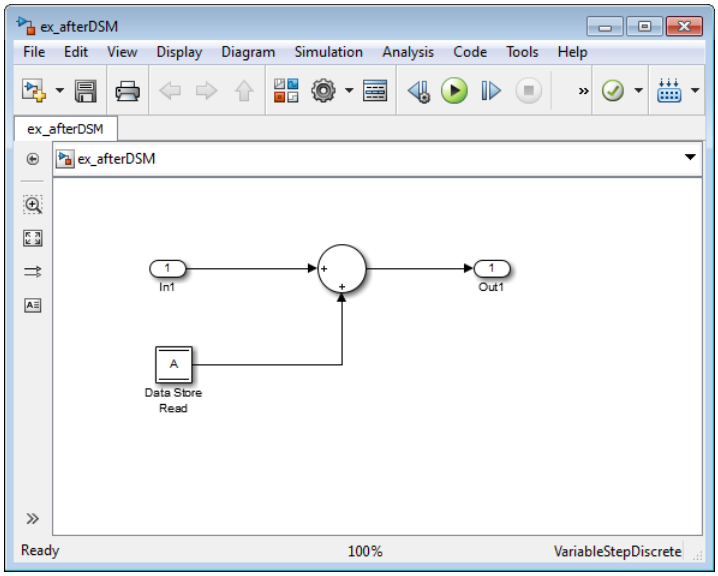
The subsystem `Before` contains the Data Store Write, and the Stateflow chart calls that subsystem before it calls the subsystem `After`, which contains the Data Store Read.

Ordering Access Using Block Priorities

You can embed data store reads and writes inside atomic subsystems or Model blocks whose priorities specify their relative execution order.







The Model block `beforeDSM` has a lower priority then `afterDSM`, so it is guaranteed to execute first. Since `beforeDSM` is atomic, all of its operations, including the Data Store Write, will execute prior to `afterDSM` and all of its operations, including the Data Store Read.

Data Stores with Buses and Arrays of Buses

Benefits of using data stores with buses and arrays of buses include:

- Simplifying the model layout by associating multiple signals with a single data store
- Producing generated code that represents the data in the store data as structures that reflect the bus hierarchy
- Writing to and reading from data stores without creating data copies, which results in more efficient data access

You cannot use a bus or array of buses that contains:

- Variable-dimension signals
- Frame-based signals

Setting Up a Model to Use Data Stores with Buses and Arrays of Buses

This procedure applies to local and global data stores, and to data stores defined with a Data Store Memory block or a `Simulink.Signal` object. Before performing the procedure, you must understand how to use data stores in a model, as described in “Create and Apply Data Stores” on page 73-12.

To use buses and arrays of buses with data stores:

1. Use the Bus Editor to define a bus object whose properties match the bus data that you want to write to and read from a data store. For details, see “Create and Specify Simulink.Bus Objects” on page 76-46.
2. Add a data store (using a Data Store Memory block or a `Simulink.Signal` object) for storing the bus data.

3 Specify the bus object as the data type of the data store. For details, see “Specify a Bus Object Data Type” on page 67-38.

4 If you use a MATLAB structure for the initial value of the data store, then set **Configuration Parameters > Diagnostics > Data Validity > Advanced parameters > Underspecified initialization detection** to `Simplified`. For details, see “Specify Initial Conditions for Bus Signals” on page 76-57 and “Underspecified initialization detection”.

5 (Optional) Select individual bus elements to write to or read from a data store. For details, see “Accessing Specific Bus and Matrix Elements” on page 73-24.

Accessing Specific Bus and Matrix Elements

Selecting Specific Bus or Matrix Elements

By default, a model writes and reads all bus and matrix elements to and from a data store.

To select specific bus or matrix elements to write to or read from a data store, use the **Element Assignment** pane of the Data Store Write block and the **Element Selection** pane of the Data Store Read block . Selecting specific bus or matrix elements offers the following benefits:

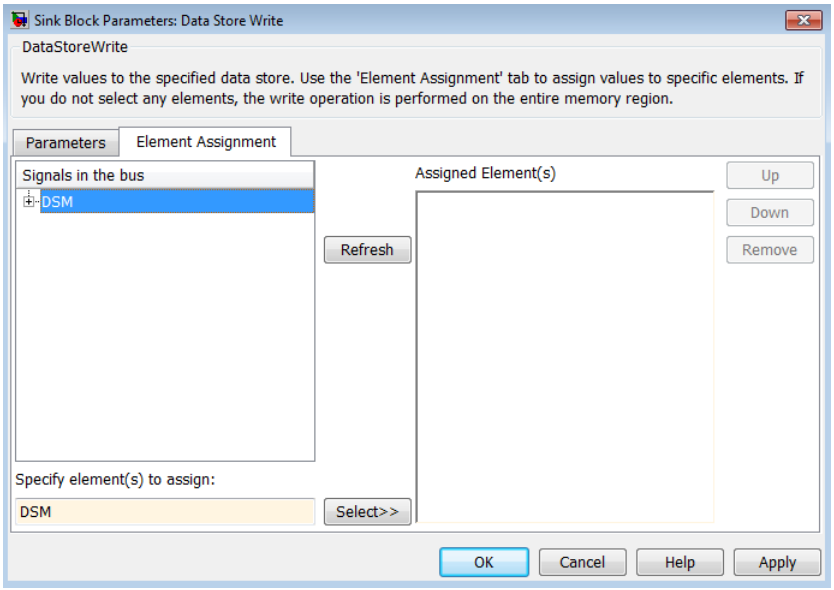
- Reducing the number of blocks in the model. For example, you can eliminate a Data Store Read and Bus Selector block pair or a Data Store Write and Bus Assignment block pair for each specific bus element that you want to access).
- Faster simulation of models with large buses and arrays of buses.

Writing Specific Elements to a Data Store

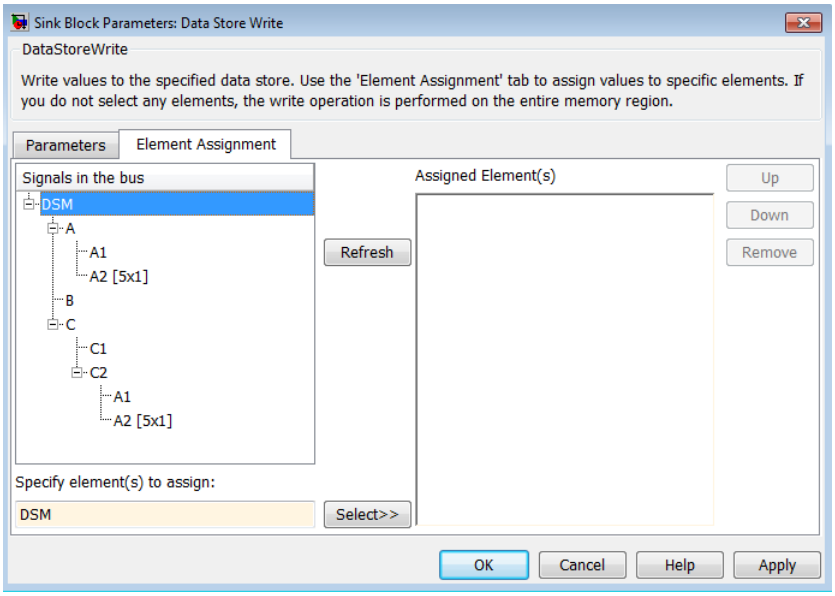
Note The following procedure describes how to use the Data Store Write block interface to write specific elements to a data store. You can also perform this task at the command line, using the `DataStoreElements` parameter to specify elements. For details, see “Specification using the command line” on page 73-27.

To assign specific bus or matrix elements to write to a data store:

1 Select the Data Store Write block and in the parameters dialog box, select the **Element Assignment** pane. For example, suppose you are using a bus with a data store named `DSM`:

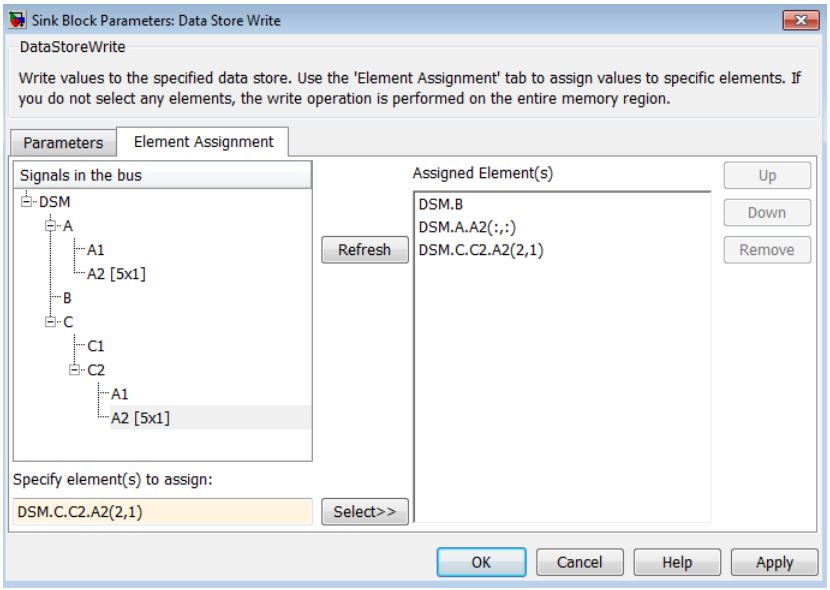


2 Expand all the elements in the **Signals in the bus** list.



3 Specify the elements that you want to write to the data store. For example:

- In the **Signals in the bus** list, click `B`. Then click **Select>>** to select the element `B`.
- To write all the elements of `A2` (in the `A` nested bus), select `A2[5x1]`. Then click **Select>>**.
- To write the second element of `A2` in the `C2` nested bus, select the `A2[5x1]` element. In the **Specify element(s) to assign** text box, edit the text to say `DSM.C.C2.A2(2,1)`.



For more examples, see “Specifying Elements to Assign or Select” on page 73-26.

4 (Optional) Reorder the assigned elements, which changes the order of the ports of the Data Store Write block.

- To reorder an assigned element, in the **Assigned element(s)** list, select the element that you want to move, and click **Up** or **Down**.
- To remove an assigned element, click **Remove**.

5 To apply the assigned elements, click **OK**.

The Data Store Write block has a port for each assigned element. The names of the selected elements that correspond to each port appear in the block icon. If you assign several signals, these additions may diminish the readability of the model. To improve readability, you can expand the size of the block or create multiple Data Store Write blocks.

Reading Specific Elements from a Data Store

Reading specific elements from a data store involves very similar steps as described in “Writing Specific Elements to a Data Store” on page 73-24. The Data Store Read block differs slightly from the Data Store Write block. A Data Store Read block has:

- An **Element Selection** pane instead of an **Element Assignment** pane
- A **Selected element(s)** list instead of an **Assigned element(s)** list

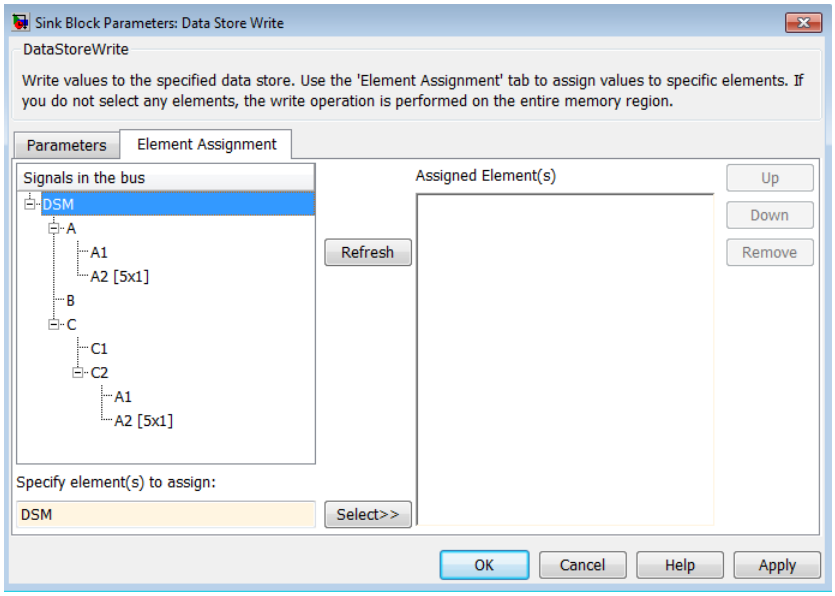
Specifying Elements to Assign or Select

Use MATLAB matrix element syntax to specify specific elements. For details about specifying matrices in MATLAB, see “Creating, Concatenating, and Expanding Matrices”.

Note To select matrix elements, you cannot use dynamic indexing with the **Element Assignment** and **Element Selection** panes of Data Store Read and Bus Assignment block pairs or Data Store Write and Bus Selector block pairs. You can, however, use a MATLAB Function block for dynamic indexing.

Valid element specifications

The following table shows examples of valid syntax for specifying elements to assign or select. These examples use the `A2` nested bus of the `A` bus, as shown in the bus hierarchy used in “Writing Specific Elements to a Data Store” on page 73-24.



Valid Syntax	Description
`DSM.A.A2(:,:)`	Selects all elements in every dimension
`DSM.A.A2([1,3,5],1)`	Selects the first, third, and fifth elements
`DSM.A.A2(2:5,1)`	Selects the second through the fifth element

Invalid element specifications

The following table shows examples of invalid syntax for specifying elements to assign or select. These examples use the `A2` nested bus of the `A` bus, as shown in the bus hierarchy used in “Writing Specific Elements to a Data Store” on page 73-24.

Invalid Syntax	Reason the Syntax Is Invalid
`DSM.A.A2(:)`	You must specify a colon for each dimension. For the bus hierarchy used in these examples, you must use two colons.
`DSM.A.A2(2:end,1)`	You cannot use the `end` operator.
`DSM.A.A2(idx,1)`	You cannot use variables to specify indices. Consider using a MATLAB Function block.
`DSM.A.A2(-1,1)`	The dimension –1 is not within the valid dimension bounds.

Specification using the command line

To set the elements to write to or read from, use the `DataStoreElements` parameter. Use a pound sign (#) to delimit multiple elements. For example, select the Data Store Write or Data Store Read block for which you want to specify elements and enter a command such as:

```
set_param(gcb, 'DataStoreElements', 'DSM.A#DSM.B#DSM.C(3,4)')
```

This specification results in the block now having three ports corresponding to the elements that you specified.

Rename Data Stores

Rename Data Store Defined by Block

Rename a data store everywhere it is used by Data Store Read and Data Store Write blocks in a model.

1. In a Data Store Memory block dialog box, type a new name in the **Data store name** box, and click **Rename All**.
2. In the **Rename All** dialog box, confirm the new data store name in the **New name** field, and click **OK**

Note You cannot use **Rename All** to rename a data store if you create a `Simulink.Signal` object in a workspace to control the code generated for the data store. Instead, you must rename the corresponding `Simulink.Signal` object using Model Explorer. For an example, see “Rename Data Store Defined by Signal Object” on page 73-28.

Rename Data Store Defined by Signal Object

This example shows how to rename a data store defined by a `Simulink.Signal` object. You can use Model Explorer to rename the object everywhere it is used by Data Store Read and Data Store Write blocks in a model or in a model reference hierarchy.

1. Open the model `sldemo_mdlref_dsm`. The model creates a `Simulink.Signal` object `ErrorCond` in the MATLAB base workspace and uses the object as a global data store in a model reference hierarchy.
2. Open Model Explorer.
3. In the **Model Hierarchy** pane, select the base workspace.
4. In the **Contents** pane, right-click the data store `ErrorCond` and select **Rename All**.
5. In the **Select a system** dialog box, click the name of the model `sldemo_mdlref_dsm` to select it as the context for renaming the data store `ErrorCond`.
6. Select **Search in referenced models** since `ErrorCond` is a global data store that is used in a referenced model. Click **OK**.

   The **Update diagram to include recent changes** check box is cleared by default to save time by avoiding unnecessary model diagram updates. Select the check box to incorporate recent changes you made to the model by forcing a diagram update.
7. Click **OK** in response to the message to update the model diagram.

   Since you just opened the model, you must update the model diagram at least once before renaming a variable such as `ErrorCond`. You could have selected **Update diagram to include**

recent changes in the **Select a system** dialog box to force an initial diagram update, though you typically use that option when you make changes to the model while performing multiple variable renaming operations.

8 In the **Rename All** dialog box, type the new name for the data store in the **New name** box and click **OK**.

Customized Data Store Access Functions in Generated Code

Embedded Coder provides a storage class that you can use to specify customized data store access functions in generated code. See “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder) and “Access Data Through Functions with Storage Class GetSet” (Embedded Coder).

See Also

Data Store Memory | Data Store Read | Data Store Write | `Simulink.Signal`

Related Examples

- “Data Stores in Generated Code” (Simulink Coder)
- “Log Data Stores” on page 73-30
- “Data Store Basics” on page 73-2
- “Data Objects” on page 67-58
- “Signal Basics” on page 75-2
- “Virtual Bus” on page 76-2

Log Data Stores

In this section...

Logging Local and Global Data Store Values

You can log the values of a local or global data store data variable for all the steps in a simulation. Two common uses of data store logging are for:

- Model debugging – view the order of all data store writes
- Confirming a model modification – use the logged data to establish a baseline for comparing results to identify the impact of a model modification

For an example of logging a global data store, see “Using Data Stores Across Multiple Models”.

Supported Data Types, Dimensions, and Complexity for Logging Data Stores

You can log data stores that use the following data types:

- All built-in data types
- Enumerated data types
- Fixed-point data types

You can log data stores that use any dimension level or complexity.

Data Store Logging Limitations

Limitations for using data store logging in a model are:

- To log data for a data store memory:
  - Simulate the top-level model in Normal mode.
  - For local data stores, the model containing the Data Store Memory block must be in Model Reference Normal mode.
  - Any block in a referenced model that writes to the data store memory must be executed in model reference Normal mode.
- You cannot log data stores that use custom data types, including buses.

Logging Data Stores Created with a Data Store Memory Block

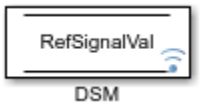
To log a local data store that you create with a Data Store Memory block:

1. In the model, open the Model Data Editor. In the **Modeling** tab, click **Model Data Editor**.
2. On the **Data Stores** tab, set the **Change view** drop-down list to `Instrumentation`.
3. In the data table, for the target data store, select the check box in the **Log Data** column.

   If the target data store does not appear in the table, click the **Change scope** button to display data stores that are defined in subsystems below your current system.
4. Optionally, to configure additional logging characteristics such as the maximum number of data points to log, open the Property Inspector (in the **Modeling** tab, under **Design**, click **Property Inspector**). Use the Property Inspector to open the block dialog box and inspect the **Logging** tab.
5. Enable data store logging with the **Model Configuration Parameters > Data Import/Export > Data stores** parameter.
6. Simulate the model.

Logging Icon for the Data Store Memory Block

When you enable logging for a model, and you configure a local data store for logging, the Data Store Memory block displays a blue icon. If you do not enable logging for the model, then the icon is gray.



Logging Data Stores Created with a Simulink.Signal Object

You can create local and global data stores using a `Simulink.Signal` object. See “Data Stores with Signal Objects” on page 73-16 for details.

To log a data store that you create with a `Simulink.Signal` object:

1. Create a `Simulink.Signal` object in a workspace that is visible to every component that needs to access the data store, as described in “Data Stores with Signal Objects” on page 73-16.
2. Use the name of the `Simulink.Signal` object in the **Data store name** block parameters of the Data Store Read and Data Store Write blocks that you want to write to and read from the data store.
3. From the MATLAB command line, set `DataLogging` (which is a property of the `LoggingInfo` property of `Simulink.Signal`) to `1`.

   For example, if you use a `Simulink.Signal` object called `DataStoreSignalObject` to create a data store, use the following command:

   ```
   DataStoreSignalObject.LoggingInfo.DataLogging = 1
   ```
4. Optionally, specify limits for the amount of data logged, using the following properties, which are properties of the `LoggingInfo` property of the `Simulink.Signal` object: `Decimation`, `LimitDataPoints`, and `MaxPoints`.

5 Enable data store logging with the **Model Configuration Parameters > Data Import/Export > Data stores** parameter.

6 Simulate the model.

Accessing Data Store Logging Data

The following Simulink classes represent data from data store logging and provide methods for accessing that data:

Class	Description
`Simulink.SimulationData.BlockPath`	Represents a fully specified Simulink block path; use for capturing the full model reference hierarchy
`Simulink.SimulationData.Dataset`	Stores logged data elements and provides searching capabilities; use to group `Simulink.SimulationData.Element` objects in a single object
`Simulink.SimulationData.DataStoreMemory`	Stores logging information from a data store during simulation

You can also convert data logged in formats other than Dataset. For more information, see “Dataset Conversion for Logged Data” on page 72-12.

Viewing Data Store Data

To view data store logging data from the command line, view the output data set in the base workspace. The default variable for the data store logging data set is `dsmout`.

The sldemo_mdlref_dsm model illustrates approaches for viewing data store logging data.

Accessing Elements in the Data Store Logging Data

To find an element in the data store logging data, based on the `Name` or `BlockType` property, use the `getElement` method of `Simulink.SimulationData.Dataset`. For example:

```
dsmout.getElement('RefSignalVal')

ans =
Simulink.SimulationData.DataStoreMemory
Package: Simulink.SimulationData

Properties:
                 Name: 'RefSignalVal'
            Blockpath: [1x1 Simulink.SimulationData.BlockPath]
                Scope: 'local'
  DSMWriterBlockPaths: [1x2 Simulinkl.SimulationData.BlockPath]
           DSMWriters: [101x1 uint32]
               Values: [101x1 timeseries]
```

To access an element by index, use the `Simulink.SimulationData.Dataset.getElement` method.

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Dataset`

Related Examples

- “Model Global Data by Creating Data Stores” on page 73-10
- “Data Store Basics” on page 73-2

Simulink Data Dictionary

What Is a Data Dictionary?

A data dictionary is a persistent repository of data that are relevant to your model. You can also use the base workspace to store design data that are used by your model during simulation. However, a data dictionary provides more capabilities.

The dictionary stores design data, which define parameters and signals, and include data that define the behavior of the model. The dictionary does not store simulation data, which are inputs or outputs of model simulation that enter and exit Inport and Outport blocks.

Dictionary Capabilities

Dictionary Capability	Benefit
Dictionary as data source	Entries in a dictionary are persistent. You do not need to reload data during development.
Explicit data-model linkage	You can define a data dictionary as the data source for a model. During model simulation and code generation, the model retrieves data from the data dictionary.
Version handling	You can: • Link a model to a data dictionary that includes model data saved in a previous version of Simulink. • Continue using the data dictionary of a model saved in a previous version of Simulink with versions of the model saved in later versions of Simulink. • Export (save) a data dictionary for use in models created with a previous version of Simulink.
Change tracking	When you modify an entry, its status is updated in the dictionary and stored as metadata that can be tracked. The dictionary also tracks who made the changes and when. You can also view or revert changes.
Entry comparison	Compare values of entries in two dictionaries.
Data grouping into reference dictionaries	Partition and organize data items into reference dictionaries.
Model-data dependency	Discover how entries are used in the model.
Additional options to remedy a missing variable	When a workspace variable that a model needs is not available, you have additional options for remediation. For example, if you renamed the variable in a dictionary, you can create a new variable by copying the old one.
Store and partition reference data	Store and partition data that are relevant to a model, such as equipment specifications, but not used by the model during simulation.
Unified interface for defining data	Use the Model Explorer to work with design data in a dictionary.
Incremental update in memory	Improved performance and scalability with minimal memory footprint.

Dictionary Capability	Benefit
Requirements traceability linking	Navigate from a data dictionary entry to a location in a requirements document.

Sections of a Dictionary

A Simulink data dictionary consists of four sections:

- **Design Data**: Contains the variables and data types that define parameters, signals, and design data that determine the behavior of the model. Design data created or imported in a dictionary are stored in this section.

  This section can store only certain classes and data types. See “Valid Design Data Classes” on page 74-9 for more information.
- **Configurations**: Contains configuration sets, which are objects of the `Simulink.ConfigSet` class, that determine how the model is configured during simulation. These objects control attributes such as sample time and simulation start time.

  When you store configuration sets in a data dictionary, you use configuration references to access the configuration sets. Models that are linked to a dictionary resolve configuration references to configuration sets in the dictionary. For more information about configuration references, see “Share a Configuration with Multiple Models” on page 13-10.

  This section can also store variant configuration objects, which belong to the `Simulink.VariantConfigurationData` class. These objects store information about variant configurations, active and default variant settings, and definitions of the control variable associated with each configuration.

  Note If you load a configuration set from the data dictionary that contains a component that is not available on your system, the parameters in the missing component are reset to their default values.
- **Embedded Coder Dictionary**: Contains code generation definitions for use with Embedded Coder. To inspect and modify code definitions stored in a data dictionary, use the **Embedded Coder Dictionary** not the Model Explorer.
- **Other Data**: Contains information that is relevant to your model but not used by the model during simulation. Use this section to store reference information such as data that describe physical equipment and processes that are represented by your model.

  This section can store almost any built-in or custom class or data type. See “Invalid Other Data Classes” on page 74-9 for more information.

Dictionary Usage for Models Created with Different Versions of Simulink

Simulink provides version handling for data dictionaries. When these events occur, Simulink synchronizes data in a dictionary for use with a model regardless of the Simulink version used to create the model:

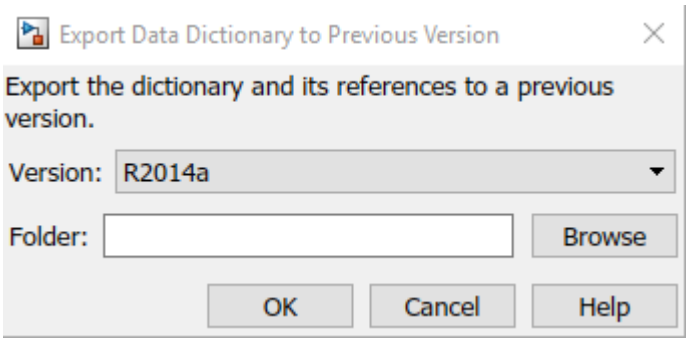
- You link a model to a data dictionary that was saved in a previous version of Simulink– for example, you link a model that you develop in R2018b with a dictionary saved in R2018a.

- You open a model that is linked to a data dictionary and was saved in a previous version of Simulink – for example, you developed a model that uses a data dictionary in R2018a and you open that model in R2018b to continue development.

To view the Simulink version in which a data dictionary is saved, in the Current Folder browser, click the data dictionary and find the **Saved in Simulink version** field in the **Details** pane. You also have the option to export (save) a data dictionary for use with models created using a different version of Simulink. To use a data dictionary you saved in a newer Simulink version in an older Simulink version, you need to export it first.

To export a data dictionary:

1. In the **Current Folder** pane of the MATLAB Command Window, navigate to the location of the data dictionary.
2. Double-click the name of the dictionary.
3. In the Model Explorer, right-click the name of the data dictionary. If you have made changes to the dictionary, in the context menu, select **Save Changes**.
4. Right-click the name of the data dictionary. In the context menu, select **Export to Previous Version**.
5. In the Export Data Dictionary to Previous Version dialog box, specify the previous version of Simulink in which you want to save the model. Specify the folder into which you want Simulink to place the new version of the dictionary. The folder that you specify cannot contain dictionaries that are part of the existing dictionary hierarchy. Then, click **OK**.



6. Verify that the new version of the data dictionary exists in the folder that you specified.

Manage and Edit Entries in a Dictionary

To create, modify, and view the entries in a data dictionary, use the Model Explorer. For more information, see “Create, Edit, and Manage Workspace Variables” on page 67-106 and “View and Revert Changes to Dictionary Data” on page 74-21.

To manage entries in a dictionary programmatically, see “Store Data in Dictionary Programmatically” on page 74-34.

Dictionary Referencing

You can reference one or more dictionaries in a parent dictionary. The data in the referenced dictionaries are visible in the parent dictionary. Use this technique to meaningfully partition data, especially for model reference hierarchies. For more information, see “Partition Dictionary Data

Using Referenced Dictionaries” on page 74-25 and “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27.

Import and Export File Formats

File Format	Import to Dictionary	Export from Dictionary
MAT-file	✓	✓
MATLAB script	✓	✓

Allow Access to Base Workspace

For information about the **Enable model access to base workspace** property and the **Enable dictionary access to base workspace** property, see “Continue to Use Shared Data in the Base Workspace” on page 74-10.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Using a Data Dictionary to Manage the Data for a Fuel Control System”
- “Migrate Models to Use Simulink Data Dictionary” on page 74-6
- “View and Revert Changes to Dictionary Data” on page 74-21
- “Store Data in Dictionary Programmatically” on page 74-34
- “Link Test Cases to Requirements Documents” (Simulink Requirements)

Migrate Models to Use Simulink Data Dictionary

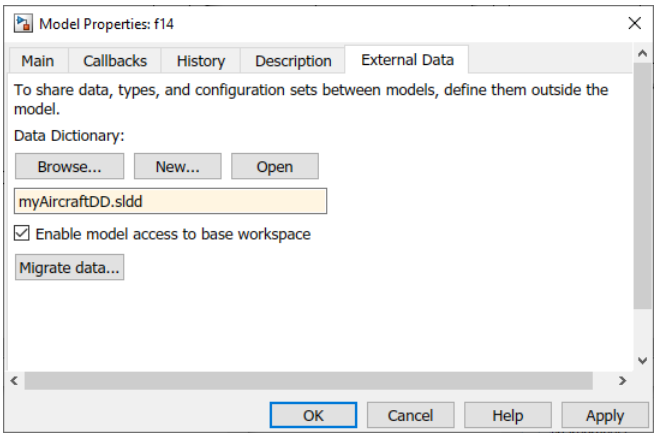
A Simulink data dictionary permanently stores model data including MATLAB variables, data objects, and data types. For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

Migrate Single Model to Use Dictionary

This example shows how to link a single standalone model to a single data dictionary.

Note Simulink does not import simulation data such as Timeseries objects into the data dictionary.

1. Open the `f14` model, which loads design data into the base workspace.
2. Save a copy of the model to your current folder. Open the copy.
3. In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
4. In the **Model Properties** dialog box, click **New** to create a data dictionary.



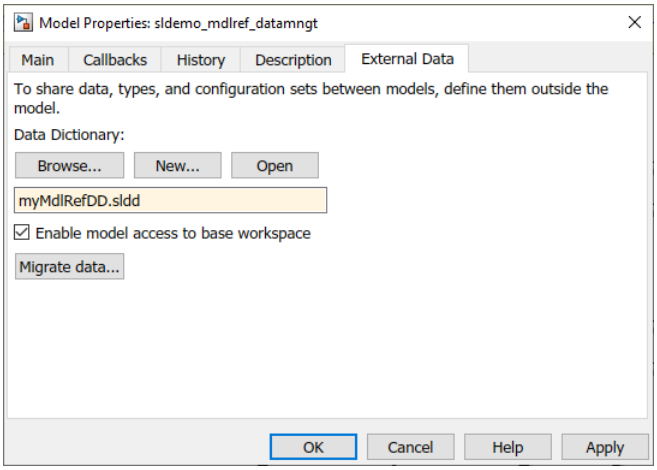
5. Name the data dictionary, save it, and click **Apply**.
6. Click **Migrate data**.
7. Click **Migrate** in response to the message about copying referenced variables.
8. (Optional) Clear **Enable model access to base workspace**.
9. Click **OK**.
10. To open the dictionary, in the Simulink Editor, click the model data badge  in the bottom left corner, then click the **External Data** link. To inspect the contents of the dictionary, in the Model Explorer **Model Hierarchy** pane, under the **External Data** node, expand the dictionary node.

Migrate Model Reference Hierarchy to Use Dictionary

This example shows how to link a parent model and all its referenced models to a single data dictionary.

1. Open the example model `sldemo_mdlref_datamngt`, which references the model `sldemo_mdlref_counter_datamngt`.

2 Save copies of the models to your current folder.
3 Open the top model, `sldemo_mdlref_datamngt`.
4 In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
5 In the **Model Properties** dialog box, click **New** to create a data dictionary.



6 Name the data dictionary, save it, and click **Apply**.
7 Click **Change all models** in response to the message about linking referenced models that do not already use a dictionary.
8 Click **Migrate data**.
9 Click **Migrate** in response to the message about copying referenced variables.
10 (Optional) Clear **Enable model access to base workspace**.
11 Click **OK**.

Considerations before Migrating to Data Dictionary

After you link a model to a data dictionary, you can choose to migrate data from the base workspace into the dictionary. If you choose to migrate data, take these considerations into account.

Check for Data-Loading Callbacks

You can use model callbacks such as the `PreLoadFcn` callback to load design data from a file into the base workspace when a model is loaded. For example, the following callback loads design data from the MAT file `myData.mat`.

```
load myData
```

After you migrate to a data dictionary, these callbacks will continue to load design data into the base workspace. Since the model then derives design data from the dictionary, manually remove or comment out these data-loading callbacks.

You can use the Dependency Analyzer to find data-loading callbacks. See “Analyze Model Dependencies” on page 17-40.

Check Scripts

A new model has access to the base workspace by default, but does not lose access when it is linked to a data dictionary. Scripts must be written with the assumption that the model can have access to the base workspace, the data dictionary, or both.

If you make explicit references to the base workspace by using the handle `base` in your scripts, consider changing these references.

Consider this example. Here, the script searches the base workspace for variable `sensor` and sets the parameter `enable` depending on the value of `sensor.noiseEnable`.

```
if evalin('base','sensor.noiseEnable')
    enable = 'Enabled';
else
    enable = 'Disabled';
end
```

When you migrate to a data dictionary, replace these explicit references to `base` as follows:

```
if Simulink.data.evalinGlobal(myExampleModel,...
'sensor.noiseEnable')
    enable = 'Enabled';
else
    enable = 'Disabled';
end
```

The `Simulink.data.evalinGlobal` function evaluates an expression in the global scope of the specified model. Here, the global scope can be in a data dictionary or the base workspace, if the model is not linked to a dictionary.

Check Tunable Parameters for Code Generation

- If your model is linked to a data dictionary, and the model does not have access to the base workspace (see “Continue to Use Shared Data in the Base Workspace” on page 74-10), Simulink ignores storage class information specified in the Model Parameter Configuration dialog box.
- If you use the Simulink interface to migrate a model to use a data dictionary, and you choose to migrate base workspace data, Simulink also migrates the storage class information of the model. If your model contains storage class information for variables in the base workspace, Simulink converts these variables into `Simulink.Parameter` objects during migration. Then, Simulink sets the storage class of these `Simulink.Parameter` objects using the storage class information from the model.
- If you migrate this model back to the base workspace, Simulink does not restore the storage class information in the model. To preserve the storage class for these variables, use the parameter objects from the data dictionary. You can also manually reset the storage class information in the model.
- If you set the `DataDictionary` property of a model from the command line, you can convert tunable variables to `Simulink.Parameter` objects using the `tunablevars2parameterobjects` function.

Data Used by Model References

When you use model referencing to break a large system of models into smaller components and subcomponents, you can create data dictionaries to segregate the design data. Design data is the set of workspace variables that the models use to specify block parameters and signal characteristics.

The models in a model reference hierarchy typically share data. Data ownership, the number of shared variables, and the complexity of your sharing strategy can influence the way that you use dictionaries.

Duplicate data definitions can exist in a model reference hierarchy under these conditions:

- Each model in the hierarchy can see only one definition.
- Definitions must be the same across models in the hierarchy.

For more information, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.

Valid Design Data Classes

You can import, store, or create MATLAB variables that use Simulink supported data types, such as `boolean` and `int32`, and structures in the **Design Data** section of a Simulink data dictionary. You can also use objects of these classes and objects of most classes that subclass these classes:

- `Simulink.AliasType`
- `Simulink.Bus`
- `Simulink.NumericType`
- `Simulink.Parameter`
- `Simulink.LookupTable`
- `Simulink.Breakpoint`
- `Simulink.Signal`
- `Simulink.Variant`
- `Simulink.data.dictionary.EnumTypeDefinition`
- `embedded.fi`
- `embedded.fimath`
- `numlti`

In addition, you can import, store, or create configuration objects of the following classes in the **Configurations** section of a Simulink data dictionary.

- `ConfigSet`
- `Simulink.VariantConfigurationData`

Invalid Other Data Classes

You can import, store, or create data objects of many built-in and custom classes or data types in the **Other Data** section of a Simulink data dictionary, except for the following:

- Arrays of objects created from built-in or custom classes
- Custom classes that have a property with any of these names:
  - `LastModified`
  - `LastModifiedBy`
  - `DataSource`
  - `Status`
  - `Variant`

Migration With From Workspace Blocks

If a model contains a From Workspace block that refers to a variable in the base workspace, you can migrate the model to a data dictionary. However, the migration process takes different actions depending on the nature of the variable that the block refers to:

- If the value of the variable is not a `timeseries` object, the migration process imports the variable to the Design Data section of the data dictionary. The block can still refer to the variable.
- If the value of the variable is a `timeseries` object (which a data dictionary cannot store) or a structure with fields identical to a timeseries object, the migration process does not import the variable. Then, when you try to update the diagram or simulate the model, the From Workspace block cannot find the variable and issues an error. In such a case, you can configure the block to refer to the base workspace variable by using the `evalin` function. See “Use with Data Dictionary”.

Data Dictionary Limitations

- Simulink cannot automatically migrate variables used only by inactive variant models into a data dictionary.
- You cannot import certain kinds of design data such as `meta` class objects and `timeseries` objects into the Design Data section of a data dictionary.
- Simulink does not allow implicit signal resolution for a model linked to a data dictionary. To use a data dictionary, set the model configuration parameter **Signal resolution** to `Explicit only` or `None`.
- If a model reference hierarchy is already linked to a data dictionary, you can protect a referenced model that is part of the hierarchy. However, if you migrate a model reference hierarchy that includes a protected model, simulation will fail.

  In other words, migrate a model to use a data dictionary before protecting it.

Continue to Use Shared Data in the Base Workspace

You can continue to store shared data in the base workspace and store model-specific data in the data dictionary by:

- Enabling access to the base workspace for the model.
- Enabling access to the base workspace from a data dictionary.

To enable access to the base workspace for a model, in the Model Properties dialog box, on the **External Data** tab, select **Enable model access to base workspace**. For a new model, this check box is selected by default. If the model is not linked to a data dictionary, this option must be selected.

You can also allow access to the base workspace from a data dictionary. For an existing dictionary, in the Model Explorer, select **Enable dictionary access to base workspace**.

When you allow base workspace access from a data dictionary, these limitations and ramifications apply:

- In general, you cannot interact with base workspace data through the dictionary.
  - When you inspect the contents of the dictionary in the Model Explorer, you cannot see base workspace data. To interact with base workspace data, in the **Model Hierarchy** pane, select the **Base Workspace** node.

  - With the programmatic interface of the data dictionary (see “Store Data in Dictionary Programmatically” on page 74-34), to interact with base workspace data, you can use only these functions with a `Simulink.data.dictionary.Section` object:

    - `assignin`
    - `exist`
    - `evalin`

    Consider using functions such as `Simulink.data.assigninGlobal` instead. See “Transition to Using Data Dictionary” on page 74-37.
- Change-tracking features, such as the ability to view and revert changes to dictionary entries (see “View and Revert Changes to Dictionary Entries” on page 74-21), do not apply to base workspace data.
- When you export data from a dictionary (see “Import and Export Dictionary Data” on page 74-16), Simulink ignores base workspace data.
- Simulink treats the base workspace and the dictionary as a single namespace. However you can define two variables with the same name, one in the base workspace and one in the dictionary. In this case, the variables must be identical and the variable in the dictionary is used.

Migrate Complicated Model Hierarchy with Shared Data

For examples, see “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27.

See Also

“Reference Protected Models from Third Parties” on page 8-13 | From Workspace

Related Examples

- “Import and Export Dictionary Data” on page 74-16
- “View and Revert Changes to Dictionary Data” on page 74-21
- “Programmatically Migrate Single Model to Use Dictionary” on page 74-38
- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Analyze Model Dependencies” on page 17-40

Enumerations in Data Dictionary

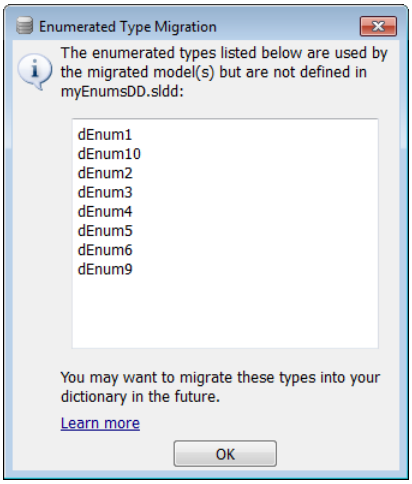
A Simulink data dictionary permanently stores model data including MATLAB variables, data objects, and data types including enumerated types. Enumeration classes defined in MATLAB by the data dictionary are owned by that dictionary and cannot be cleared by using `Simulink.clearIntEnumType`. When you close a data dictionary, the dictionary clears the enumeration classes that it owns. If an instance of an enumeration class exists when you close the dictionary, that enumeration class is not cleared and you become the owner of the class. As the owner of the class, you can then find or clear the class by using `Simulink.findIntEnumType` and `Simulink.clearIntEnumType`. For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

Migrate Enumerated Types into Data Dictionary

This example shows how to migrate enumerated types that are used by a model into a data dictionary.

Import Design Data

1. Open a model that uses enumerated types for design data or for blocks in the model.
2. In the Simulink Editor, on the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
3. In the **Model Properties** dialog box, click **New** to create a data dictionary.
4. Name the data dictionary, save it, and click **Apply**.
5. Click **Migrate data**.
6. Click **Migrate** in response to the message about copying referenced variables.
7. Simulink reports the enumerated types that were not imported into the data dictionary.



8. Click **OK**.

   A notification appears in the Simulink Editor, reporting that your model is now linked to the data dictionary.

Import Enumerated Types

Import the definitions of enumerated types only after you import all the design data that were creating using the types. When you import enumerated types to a data dictionary, Simulink disables

MATLAB files or P-files that contain the type definitions, causing variables that remain in the MATLAB base workspace to lose their definitions.

1. At the MATLAB command prompt, get the names of enumerated types that are used in model blocks.

```
% Find all variables and enumerated types used in model blocks
usedTypesVars = Simulink.findVars('EnumsReporting','IncludeEnumTypes',true);
% Here, EnumsReporting is the name of the model and
% usedTypesVars is an array of Simulink.VariableUsage objects

% Find indices of enumerated types that are defined by MATLAB files or P-files
enumTypesFile = strcmp({usedTypesVars.SourceType},'MATLAB file');

% Find indices of enumerated types that are defined using the function
% Simulink.defineIntEnumType
enumTypesDynamic = strcmp({usedTypesVars.SourceType},'dynamic class');

% In one array, represent indices of both kinds of enumerated types
enumTypesIndex = enumTypesFile | enumTypesDynamic;

% Use logical indexing to return the names of used enumerated types
enumTypeNames = {usedTypesVars(enumTypesIndex).Name}'
```

```
enumTypeNames =

    'dEnum1'
    'dEnum10'
    'dEnum2'
    'dEnum3'
    'dEnum4'
    'dEnum5'
    'dEnum6'
    'dEnum9'
```

2. Open the data dictionary, and represent it with a `Simulink.data.Dictionary` object.

```
ddConnection = Simulink.data.dictionary.open('myEnumsDD.sldd')
```

```
ddConnection =

  Dictionary with properties:

         DataSources: {0x1 cell}
    HasUnsavedChanges: 0
          NumberOfEntries: 3
```

3. Use the `importEnumTypes` method to import the enumerated types that are used by blocks in the model. The method saves changes made to the target dictionary, so before you use the method, confirm that unsaved changes are acceptable.

```
[successfulMigrations, unsuccessfulMigrations] = ...
importEnumTypes(ddConnection,enumTypeNames)
```

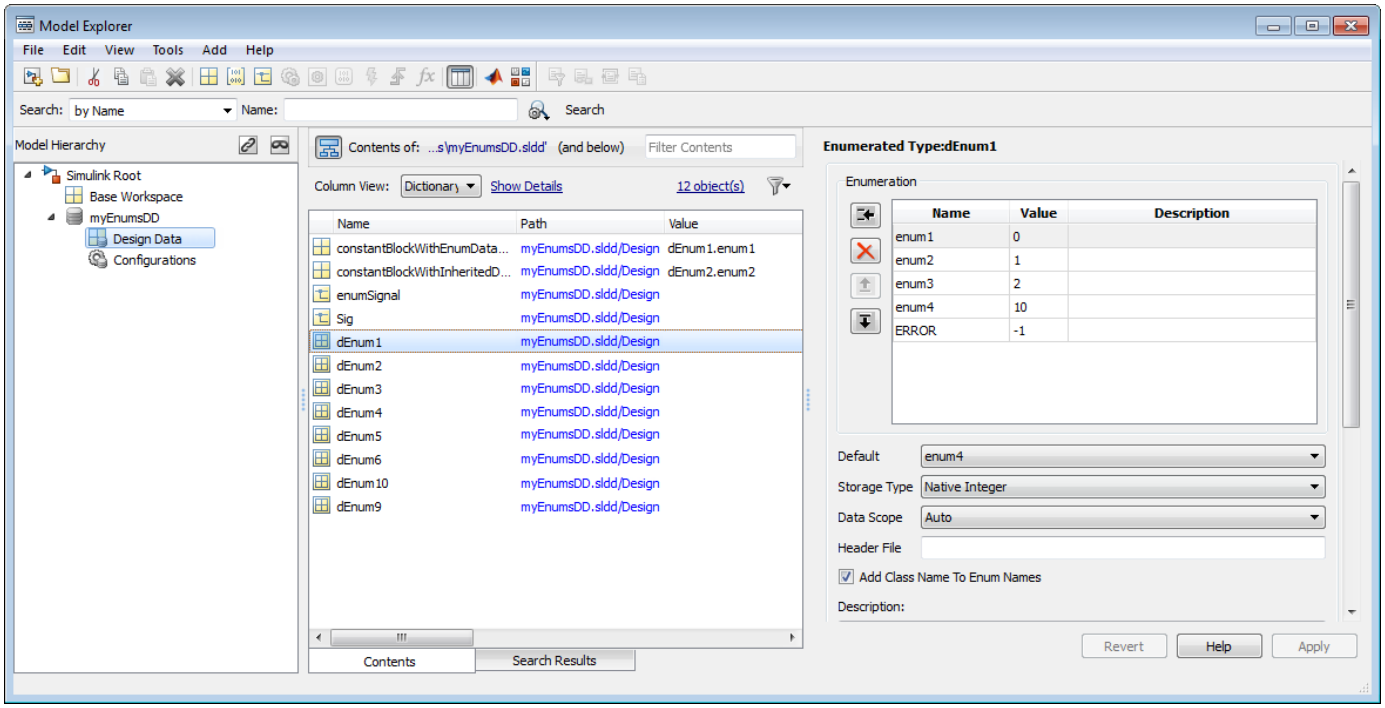
```
successfulMigrations =

1x6 struct array with fields:

    className
    renamedFiles


unsuccessfulMigrations =
```

```
1x2 struct array with fields:

    className
    reasons
```

When enumerated types are imported, `importEnumTypes` renames the enumerated class definition file by appending `.save` to the file name. For example, if the original enumerated class definition is named `Enum1.m`, Simulink renamed the file as `Enum1.m.save`.

The structure `unsuccessfulMigrations` reports enumerated types that are not migrated. In this example, two enumerated type instances are defined in the model workspace and can be imported after closing the model. Close the model to import these enumerated types.

4 Open the dictionary to view the migrated enumerated types.



Manipulate Enumerations in Data Dictionary

These examples show how to operate on existing enumerations in a data dictionary.

Rename Enumerated Type Definition

1 In the data dictionary, create a copy of the enumerated type, and rename the copy instead.

2 Find enumeration objects used by your model that are derived from the type with the old name.

3 Replace these objects with those derived from the renamed type.

4 Delete the type with the old name.

Rename Enumeration Members

Use one of the following approaches.

- Select the enumeration within the dictionary, and rename one or more enumeration members.
- If your model references enumeration members, change these references to match the renamed member.

Delete Enumeration Members

1 Find references in your model to an enumeration member you want to delete.
2 Replace these references with an alternate member.
3 Delete the original member from the enumeration.

Change Underlying Value of Enumeration Member

You can change the values of enumeration members when you represent these values as MATLAB variables or by using `Value` field of `Simulink.Parameter` objects.

1 Find references in your model to an enumeration member whose value you want to change.
2 Make a note of these references.
3 Change the value of the enumeration member.
4 Manually update references to the enumeration member in your model.

Remove Enumerated Types from Data Dictionary

If you decide that you no longer want to define an enumerated type in a dictionary, follow these steps.

1 Manually define the enumerated data type in MATLAB. See “Use Enumerated Data in Simulink Models” on page 68-6.
2 Delete instances of the enumeration class. If there is an instance of the enumeration class in existence when you delete the enumerated data type from the dictionary, the dictionary releases control of the enumeration class, but the class remains in memory as a dynamic enumeration class. You can then find or clear the class by using `Simulink.findIntEnumType` and `Simulink.clearIntEnumType`.
3 Delete the enumerated type from the dictionary.

See Also

`Simulink.data.dictionary.EnumTypeDefinition`

Related Examples

- “Use Enumerated Data in Simulink Models” on page 68-6
- “Simulink Enumerations” on page 68-2

Import and Export Dictionary Data

A Simulink data dictionary permanently stores model data including MATLAB variables, data objects, and data types. For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

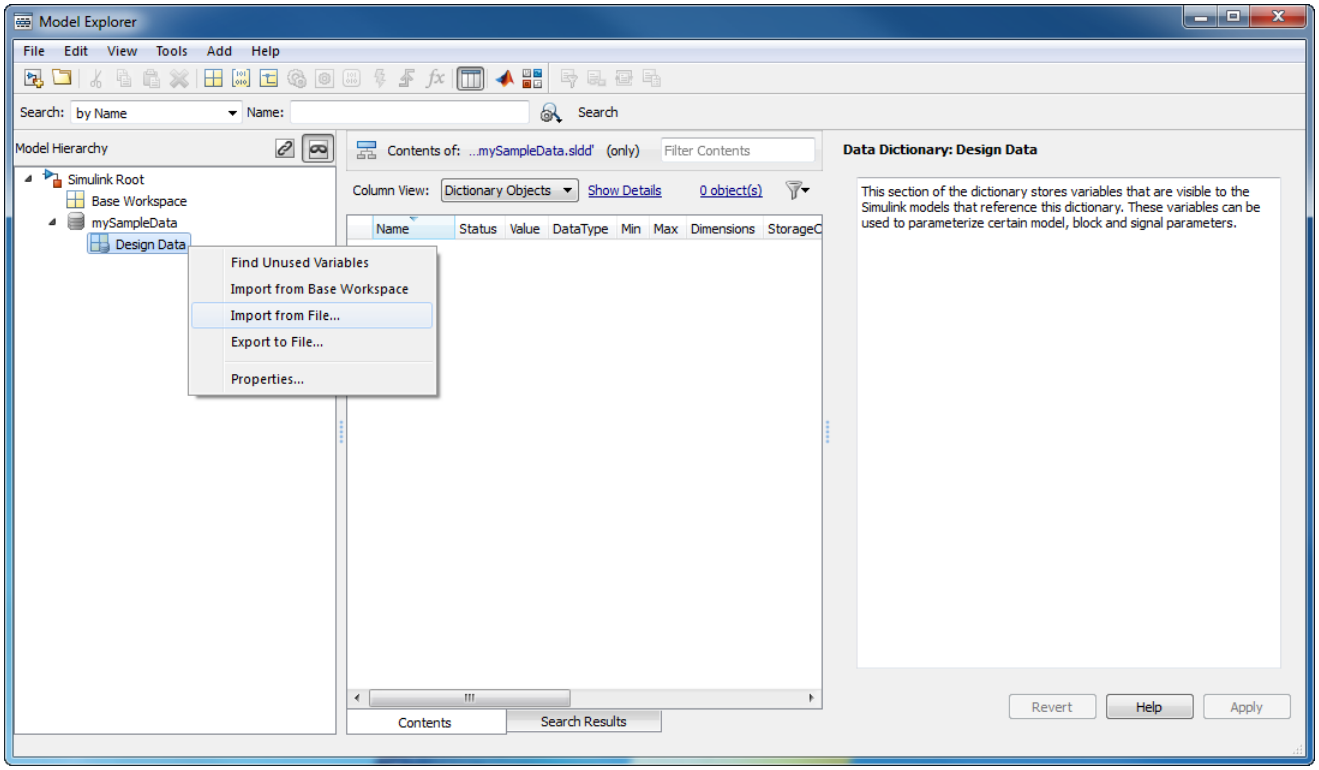
Import Data to Dictionary from File

You can import data from a MATLAB file or MAT-file to a data dictionary using the Model Explorer window. Import variables and data objects that are used by a model during simulation to the Design Data section of a dictionary. Import variables and objects that you want to store with a model, but that are not used by the model during simulation, to the Other Data section of a dictionary.

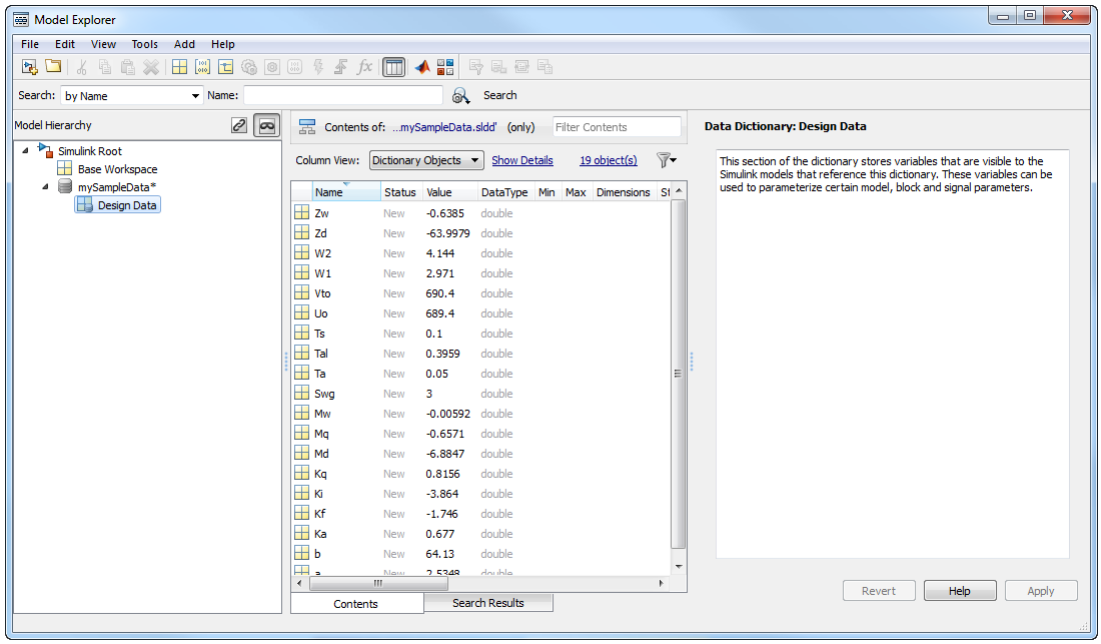
Import Design Data from File

This example shows how to import design data from a file into the Design Data section of a dictionary.

1. In the Simulink Editor, in the **Modeling** tab, click **Model Explorer** to open the Model Explorer.
2. Select **File** > **Open**. Then browse to an existing dictionary.
3. In the **Model Hierarchy** pane, right-click the **Design Data** section of the dictionary and select **Import from File**. Then browse to and select the MAT-file or MATLAB file that contains the data to import.



Design data from the MAT-file populate the dictionary. Data appear with **DataSource** set to the name of the dictionary.

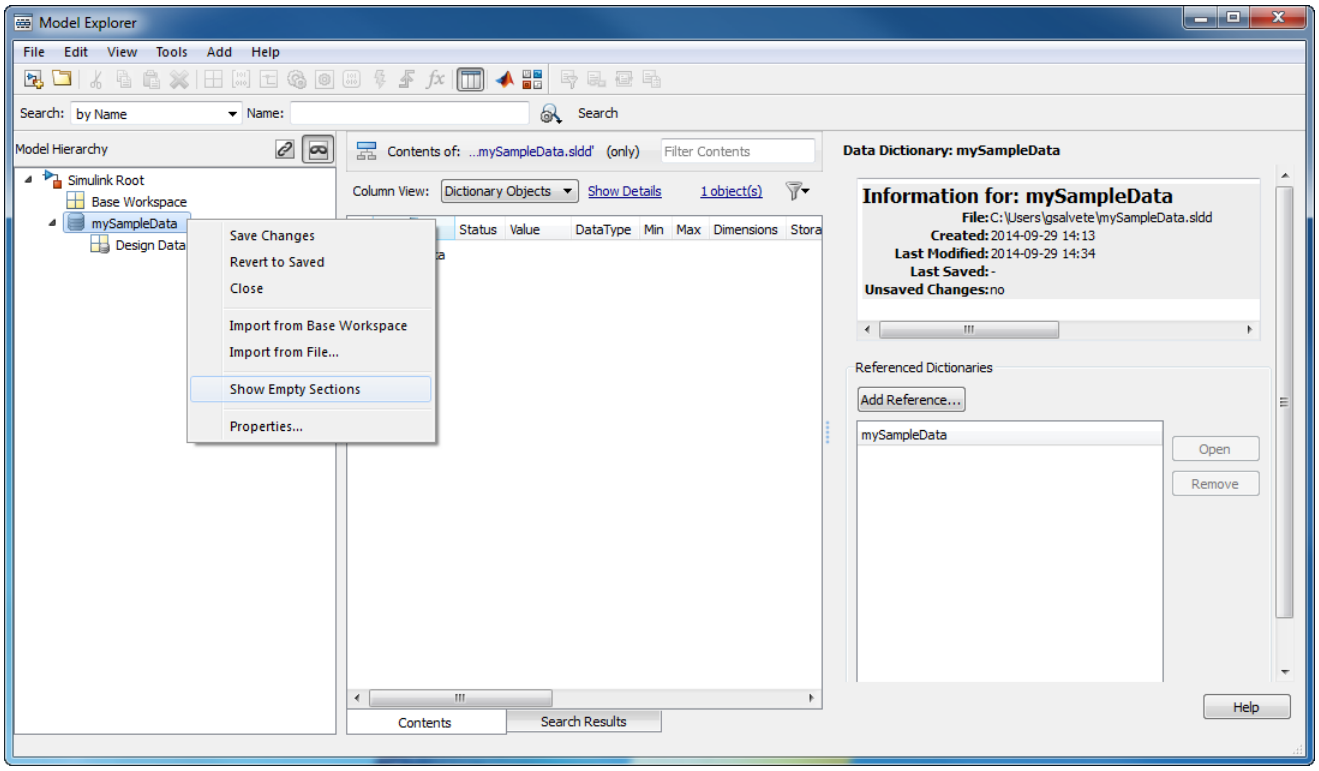


If you import from the same MAT-file again, Simulink only imports changed or new entries into the dictionary.

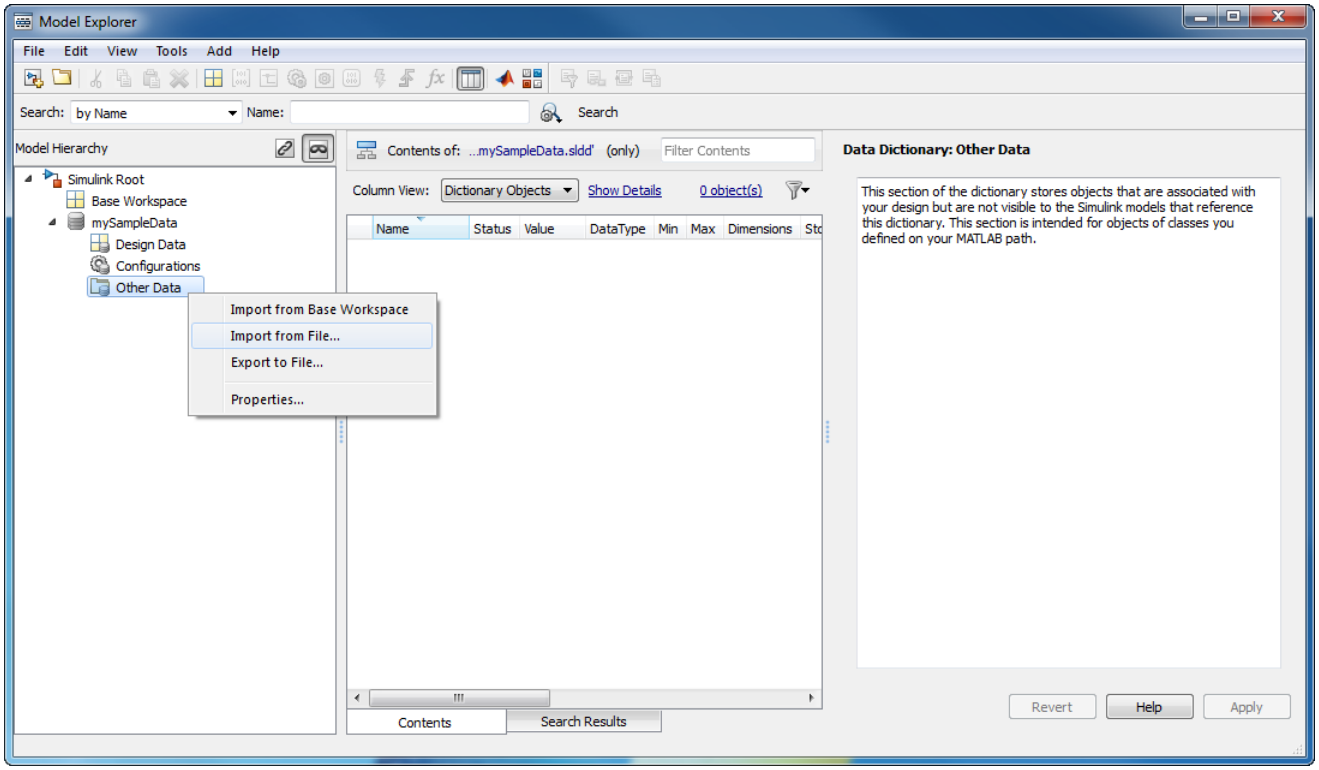
Import Other Data from File

This example shows how to import data from a file into the Other Data section of a data dictionary. Use this section to store reference information that is not used by Simulink during simulation, such as data that describe physical equipment and processes that are represented by your model.

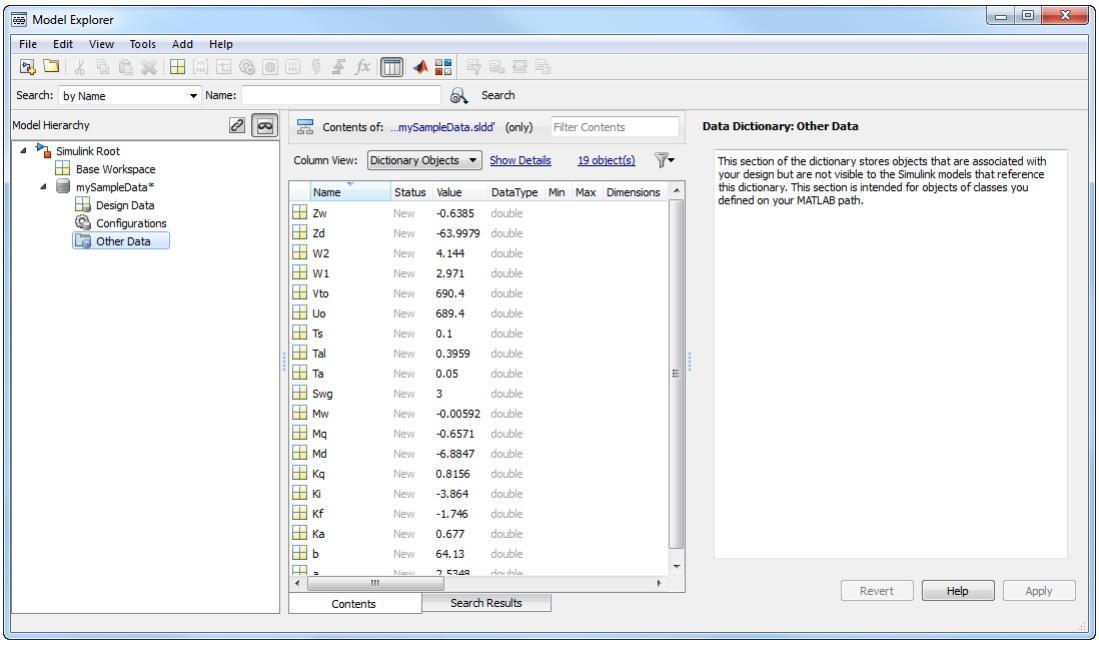
1. In the Simulink Editor, in the **Modeling** tab, click **Model Explorer** to open the Model Explorer.
2. Select **File > Open**. Then browse to an existing dictionary.
3. In the **Model Hierarchy** pane, right-click the dictionary node and select **Show Empty Sections**. Model Explorer reveals the **Other Data** and **Configurations** sections of the dictionary, even if they are empty, in addition to the **Design Data** section.



4 In the **Model Hierarchy** pane, right-click the **Other Data** section of the dictionary and select **Import from File**. Then browse to and select the MAT-file or MATLAB file that contains the reference data to import.



Data from the MAT-file populate the Other Data section of the dictionary. Data appear with **DataSource** set to the name of the dictionary.

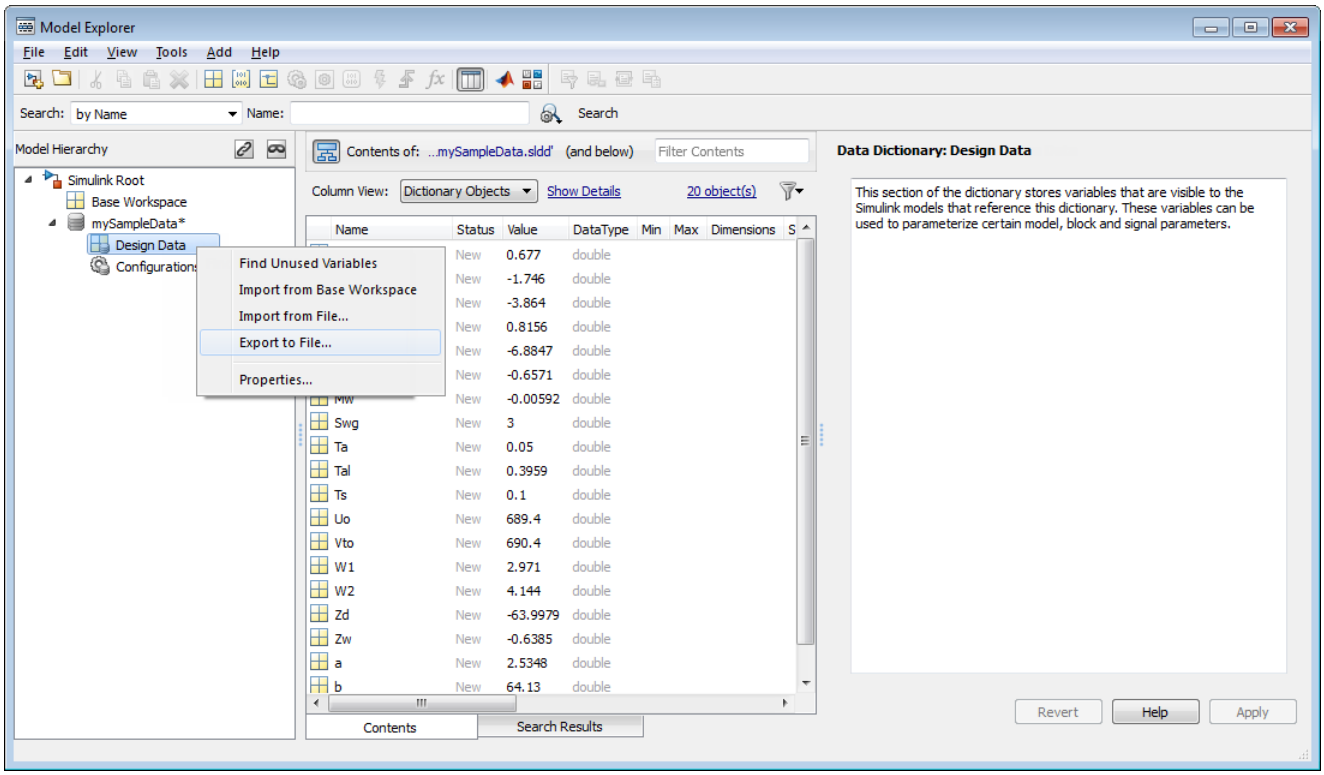


If you import from the same MAT-file again, Simulink only imports changed or new entries into the dictionary.

Export Design Data from Dictionary

This example shows how to export model design data from a data dictionary into a MAT-file or MATLAB script.

1. In the Simulink Editor, in the **Modeling** tab, click **Model Explorer** to open the Model Explorer.
2. Open a data dictionary using **File > Open Data Dictionary**.
3. In the **Model Hierarchy** pane, expand the dictionary node and select **Design Data > Export to File**. Then save the design data to a MAT-file or MATLAB script.



The dictionary does not export enumerated data types (which are stored as `Simulink.data.dictionary.EnumTypeDefinition` objects). To transfer or copy an enumerated type from one dictionary to another, use the Model Explorer to cut or copy and paste the object.

See Also

Related Examples

- “View and Revert Changes to Dictionary Data” on page 74-21
- “Migrate Models to Use Simulink Data Dictionary” on page 74-6

View and Revert Changes to Dictionary Data

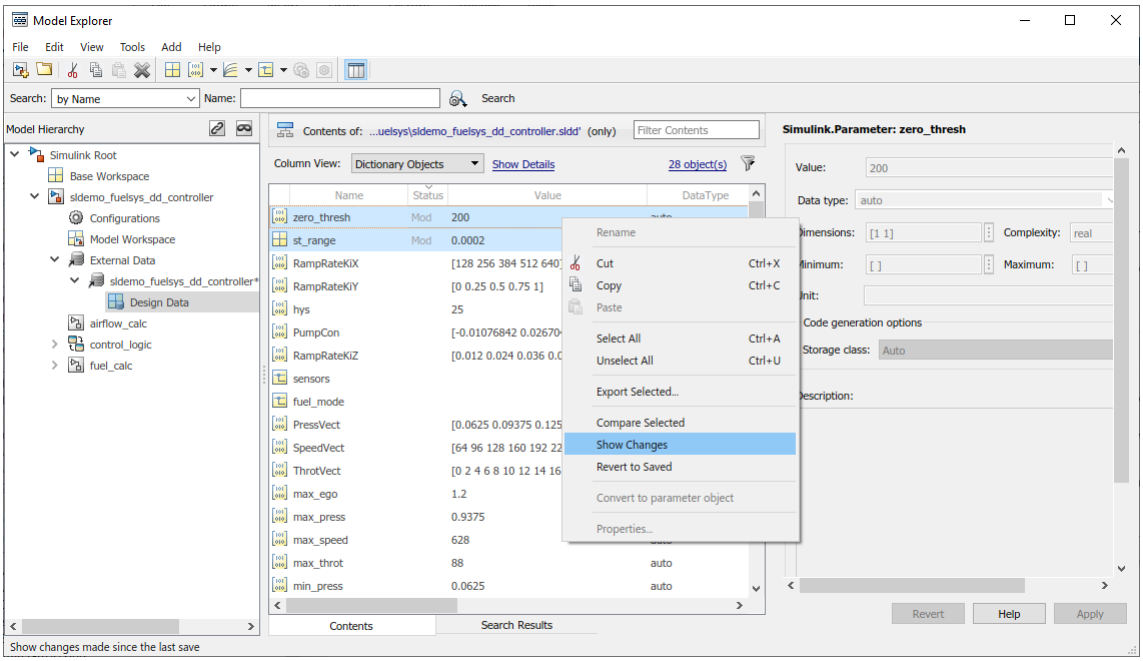
A Simulink data dictionary permanently stores model data including MATLAB variables, data objects, and data types. For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

View and Revert Changes to Dictionary Entries

This example shows how to view unsaved changes to dictionary entries, who made them, and when. You can view changes to entries in any section, including data stored in the Other Data section and configuration sets stored in the Configurations section.

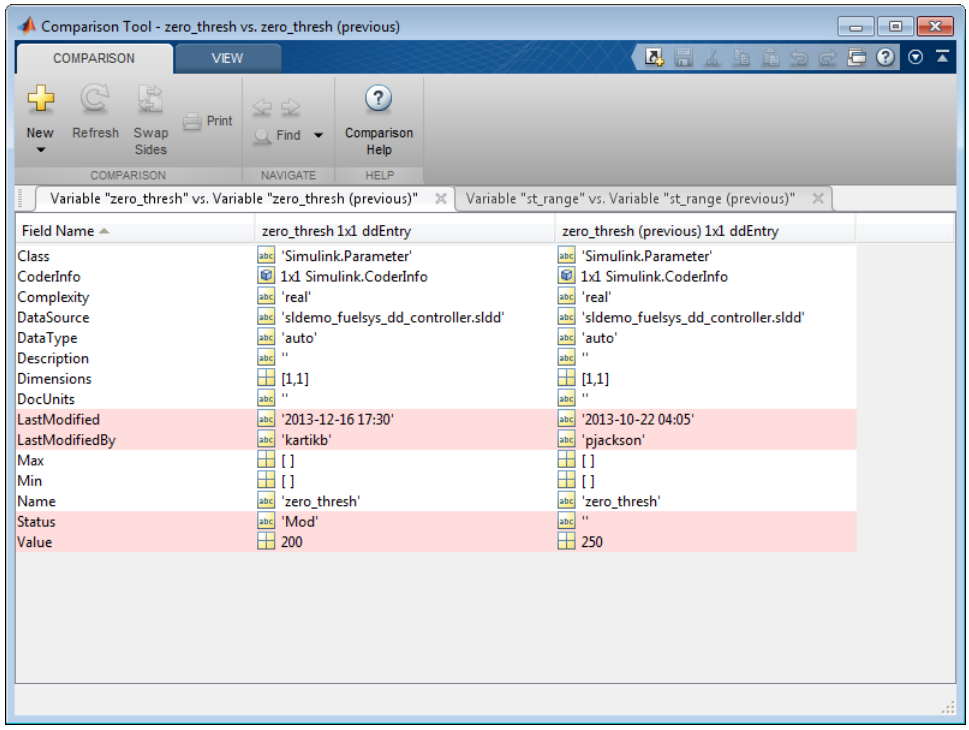
1. Open the `sldemo_fuelsys_dd_controller` model.
2. Open the data dictionary linked to this model. Click the model data badge  in the bottom left corner of the model, then click the **External Data** link.
3. In the Model Explorer **Model Hierarchy** pane, under the **External Data** node, select the **Design Data** node for `sldemo_fuelsys_dd_controller`.
4. In the **Contents** pane, change `st_range` to `0.0002` and `zero_thresh` to `200`.

   The **Status** column of these entries changes to `Mod`, indicating that they have been modified.
5. Click the heading of the **Status** column to sort the entries. Then, select the modified entries, which are indicated by the `Mod` status.



6. Right-click and select **Show Changes**.

   The Comparison Tool appears, displaying changed entries in separate tabs. The tool highlights changed values.



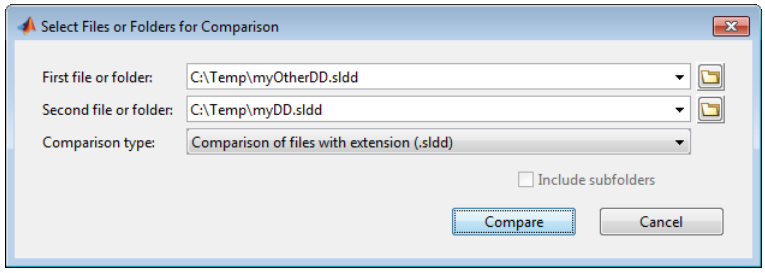
Note The Comparison Tool does not display changes to the data dictionary property **Enable dictionary access to base workspace**.

7 In the **Contents** pane of the Model Explorer, right-click `zero_thresh` and select **Revert to Saved**.

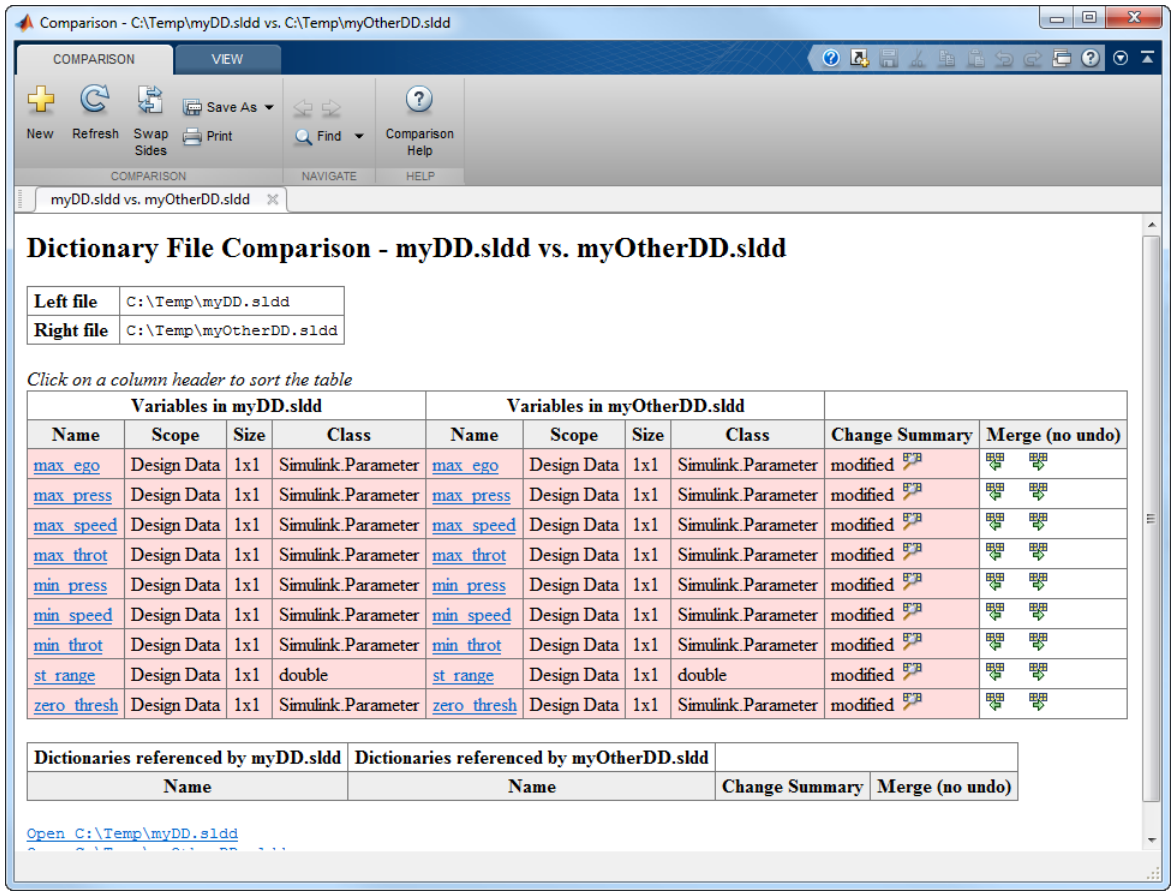
Simulink reverts `zero_thresh` to its value at the time of the last save action.

8 You can merge entries between dictionaries using the Comparison Tool. From the MATLAB desktop, on the **Home** tab, in the **File** section, click **Compare**.

9 Select the dictionaries to compare and merge.



10 In the comparison report, select the merge direction for each dictionary entry.



View and Revert Changes to Entire Dictionary

If you store model variables in a data dictionary, you can view and manage the changes that you make while you work. You can use the Comparison Tool to see the changes made to a dictionary, which compares the modified dictionary with the most recent saved version.

When you view the changes to a dictionary, you can choose to discard changes to individual entries or dictionary references, which reverts to the last saved state. You can use this technique to recover entries that you delete in your modified version or dictionary references that you remove.

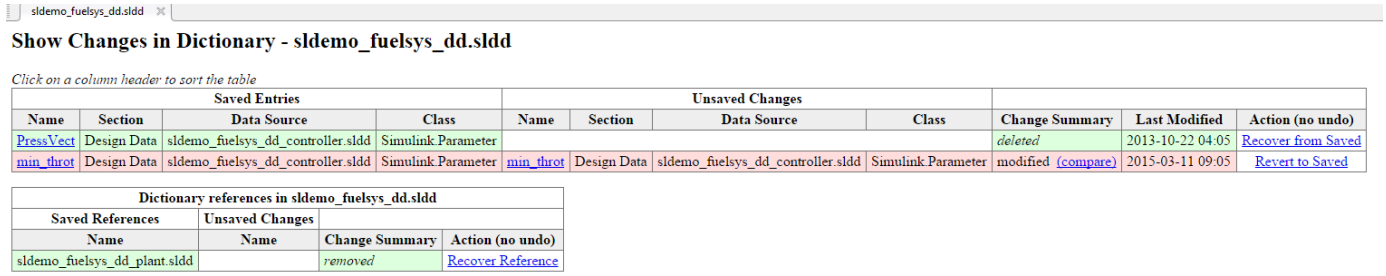
If you view changes to a dictionary that references other dictionaries, the Comparison Tool also reports changes made to the entries in the referenced dictionaries.

1. View the example data dictionary `sldemo_fuelsys_dd` in Model Explorer.

   ```
   dictionary = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
   show(dictionary)
   ```

   The dictionary contains entries that are defined in several referenced dictionaries, including `sldemo_fuelsys_dd_controller` and `sldemo_fuelsys_dd_plant`.
2. Run the script `ex_dictionary_changes`, which makes changes to `sldemo_fuelsys_dd`. Later, you can use the Comparison Tool to investigate the changes.
3. In the **Model Hierarchy** pane of Model Explorer, right-click the node `sldemo_fuelsys_dd` and select **Show Changes**.

The Comparison Tool displays the changes made to the dictionary.


sldemo_fuelsys_dd.sldd

Show Changes in Dictionary - sldemo_fuelsys_dd.sldd

Click on a column header to sort the table

Saved Entries				Unsaved Changes						
Name	Section	Data Source	Class	Name	Section	Data Source	Class	Change Summary	Last Modified	Action (no undo)
PressVect	Design Data	sldemo_fuelsys_dd_controller.sldd	Simulink.Parameter					*deleted*	2013-10-22 04:05	Recover from Saved
min_throt	Design Data	sldemo_fuelsys_dd_controller.sldd	Simulink.Parameter	min_throt	Design Data	sldemo_fuelsys_dd_controller.sldd	Simulink.Parameter	modified (compare)	2015-03-11 09:05	Revert to Saved

Dictionary references in sldemo_fuelsys_dd.sldd			
Saved References	Unsaved Changes		
Name	Name	Change Summary	Action (no undo)
sldemo_fuelsys_dd_plant.sldd		*removed*	Recover Reference

4. In the table at the top of the report, click **compare** in the Change Summary column of the row that corresponds to the entry `min_throt`.

   A new tab shows the changes made to `min_throt`. The script changed the parameter data type from `auto` to `int8` and the parameter value from `3` to `4`.

5. Click the tab that shows the changes made to the dictionary. In the Action column of the row that corresponds to the entry `min_throt`, click **Revert to Saved**.

   The entry reverts to the definition from the last saved version of the dictionary.

6. The remaining row in the report shows that the script deleted the entry `PressVect`, which was defined in the referenced dictionary `sldemo_fuelsys_dd_controller`. Click **Recover from Saved**, which recovers the entry in the referenced dictionary.

7. The table **Dictionary references in sldemo_fuelsys_dd.sldd** shows that the script removed the reference to the dictionary `sldemo_fuelsys_dd_plant`. In the Action column, click **Recover Reference**.

The report shows that there are no more unsaved changes to `sldemo_fuelsys_dd`.

See Also

Related Examples

- “Compare Revisions” on page 19-39
- “Import and Export Dictionary Data” on page 74-16
- “Migrate Models to Use Simulink Data Dictionary” on page 74-6
- “What Is a Data Dictionary?” on page 74-2

Partition Dictionary Data Using Referenced Dictionaries

This example shows how to partition a data dictionary into reference dictionaries that can be shared in a team. A Simulink data dictionary permanently stores model data including MATLAB variables, data objects, and data types.

Open dictionary for partitioning

1. Open the Model Explorer. In the Simulink Editor, in the **Modeling** tab, click **Model Explorer**.
2. Select **File > Open**.

   Browse and locate your dictionary.

Create reference dictionary

Use a reference dictionary to store a subset of entries from the main dictionary.

1. Select **File > New > Data Dictionary**.

   Name the reference dictionary and save it.

   Both dictionaries appear as nodes in the **Model Hierarchy** pane.
2. In the **Model Hierarchy** pane, select the dictionary that serves as the parent.
3. In the dialog box pane, in the **Referenced Dictionaries** section, click **Add**. Browse to the location of the reference dictionary and add it as a reference.

The **Referenced Dictionaries** section displays each directly referenced data dictionary as a top level node and the indirectly referenced data dictionaries as a flat list below each top level node. To see the full dependency tree of referenced dictionaries in the Dependency Analyzer, click **View Hierarchy**.

Move entries into reference dictionary

1. In the **Model Hierarchy** pane, select the **Design Data** node of the parent dictionary.
2. In the **Contents** pane, select the entries you want to move to the reference dictionary.
3. For one of the selected entries, set **DataSource** to the reference dictionary using the dropdown menu. You can also drag and drop entries between dictionaries.

   To make the **DataSource** column visible, click **Show Details** in the **Contents** pane. In the text box, enter `DataSource`, and add `DataSource` to the list of displayed columns.

Organize display of entries

1. Click the name of the **DataSource** column to sort the entries by the dictionaries that define them.
2. Right-click the name of the **DataSource** column and select **Group by This Column** to group the entries. The **Contents** pane creates a group for each dictionary that defines the entries.

See Also

Related Examples

- “Migrate Models to Use Simulink Data Dictionary” on page 74-6

- “Partition Data for Model Reference Hierarchy Using Data Dictionaries” on page 74-27
- “What Is a Data Dictionary?” on page 74-2

Partition Data for Model Reference Hierarchy Using Data Dictionaries

When you use model referencing to break a large system of models into smaller components and subcomponents, you can create data dictionaries to segregate the design data. Design data is the set of workspace variables that the models use to specify block parameters and signal characteristics. For basic information about data dictionaries, see “What Is a Data Dictionary?” on page 74-2.

To take this component-based approach to data management, create a shared dictionary that contains common data and a separate dictionary for each component that contains the data needed by that component.

Create a Dictionary for Each Component

This example shows how to partition design data into dictionaries. When you finish, each component in the system has a dictionary, and dictionary references allow the components to share data.

Explore Example Model Hierarchy

1. Navigate to the folder *`matlabroot`*`/help/toolbox/simulink/examples` (open).
2. Copy these files to a writable folder:
   - `ProjectData_Contr.mat`
   - `ProjectData_ContrSub1.mat`
   - `ProjectData_ContrSub2.mat`
   - `ProjectData_ContrSubs.mat`
   - `ProjectData_Plant.mat`
   - `ProjectData_System.mat`
   - `ex_SystemModel`
   - `ex_PlantComp_Lvl1`
   - `ex_PlantComp_Lvl2`
   - `ex_ContrComp`
   - `ex_ContrComp_Sub1_Lvl1`
   - `ex_ContrComp_Sub1_Lvl2`
   - `ex_ContrComp_Sub2_Lvl1`
   - `ex_ContrComp_Sub2_Lvl2`
3. Load the MAT-files to create design data in the base workspace.
4. Open the example model `ex_SystemModel`. This model is at the top of a reference hierarchy that includes the other example models.
5. In the model, update the diagram. Each bus signal in the model uses a `Simulink.Bus` object as a data type. The objects, `SensorBus` and `CtrlBus`, are in the base workspace.

   The referenced models `ex_PlantComp_Lvl1` and `ex_ContrComp` use the bus objects for root-level inputs and outputs, which means the plant and controller components share the objects.
6. In the base workspace, double-click the `Simulink.NumericType` object named `FloatType`. Signals, parameters, and other data items in the controller component use this shared data type.

7 In the Model Explorer **Model Hierarchy** pane, expand the node ex_SystemModel.

   Click the `Configurations` node. In the **Contents** pane, the node `Reference to SimConfigSet` appears. `SimConfigSet` is a `Simulink.ConfigSet` object in the base workspace. To maintain configuration parameter uniformity for simulation, all of the models in the hierarchy refer to `SimConfigSet`.

8 Right-click the node `Controller (ex_ContrComp)` and select **Open**.

9 In the Model Explorer **Model Hierarchy** pane, expand the new node `ex_ContrComp`. Click the `Configurations` node.

   In the **Contents** pane, the node `Reference to CodeGenConfigSet` appears. `CodeGenConfigSet` is a `Simulink.ConfigSet` object in the base workspace. To maintain configuration parameter uniformity for code generation, the models in the controller component refer to `CodeGenConfigSet`. The models in the plant component do not use `CodeGenConfigSet`.

10 In the **Model Hierarchy** pane, select **Base Workspace**. In the **Contents** pane, right-click the variable `diff` and select **Find Where Used**. In the **Select a system** dialog box, select `ex_SystemModel` and click **OK**. If you see a message about updating the diagram, click **OK**.

   In the **Contents** pane, the variable `diff` is used by Constant blocks in the models `ex_ContrComp_Sub1_Lvl1` and `ex_ContrComp_Sub1_Lvl2`, which make up the first controller subcomponent. Similarly, other models in the hierarchy share the base workspace variables `coeff`, `init`, `mu`, and `rho`.

The table shows the models that share each variable in the base workspace.

Variable Name	Models Using the Variable	Scope of Sharing
`CtrlBus`	Top-level models in the plant and controller components	Shared globally by entire system
`SensorBus`	Top-level models in the plant and controller components	Shared globally by entire system
`SimConfigSet`	All models in the hierarchy	Shared globally by entire system
`rho`	`ex_PlantComp_Lvl2` ,`ex_ContrComp_Sub1_Lvl2`, and `ex_ContrComp_Sub2_Lvl2`	Shared globally by entire system
`mu`	`ex_PlantComp_Lvl1` and `ex_PlantComp_Lvl2`	Shared by models in the plant component
`FloatType`	All models in the controller component	Shared by controller component and subcomponents
`CodeGenConfigSet`	All models in the controller component	Shared by controller component and subcomponents
`init`	`ex_ContrComp_Sub1_Lvl2` and `ex_ContrComp_Sub2_Lvl1`	Shared by controller subcomponents
`diff`	`ex_ContrComp_Sub1_Lvl1` and `ex_ContrComp_Sub1_Lvl2`	Shared by models in the first controller subcomponent

Variable Name	Models Using the Variable	Scope of Sharing
`coeff`	`ex_ContrComp_Sub2_Lvl1` and `ex_ContrComp_Sub2_Lvl2`	Shared by models in the second controller subcomponent

Suppose that separate teams of developers maintain the plant component and the controller components. You can use data dictionaries to store and scope the shared design data.

Create Shared Global Dictionary

Create a shared global data dictionary that contains the data shared globally by the entire system.

1. In the Model Explorer, select **File** > **New** > **Data Dictionary**.
2. Set the new dictionary name to `GlobalShare` and click **Save**.
3. In the **Model Hierarchy** pane, right-click the `GlobalShare`node and select **Show Empty Sections**.
4. In the **Model Hierarchy** pane, select **Base Workspace**. In the **Contents** pane, select the design data that are shared globally by the entire system: `CtrlBus`, `SensorBus`, and `rho`.
5. Right-click and select **Copy**.
6. In the **Model Hierarchy** pane, right-click the `Design Data` node under `GlobalShare` and select **Paste**.
7. Similarly, copy `SimConfigSet` from the **Base Workspace** and copy to the `Configurations` node under `GlobalShare`.

Create Dictionary for Plant Component

Create a data dictionary for data shared by models in the Plant component. Add a reference from this dictionary to the shared global dictionary.

1. In the Model Explorer, select **File** > **New** > **Data Dictionary**.
2. Set the new dictionary name to `Plant` and click **Save**.
3. In the **Model Hierarchy** pane, select the node `Plant`. In the Dialog pane, under **Referenced Dictionaries**, click **Add**.
4. Double-click `GlobalShare.sldd`.
5. In the **Model Hierarchy** pane, right-click the node `Plant` and select **Save Changes**.

Link Plant Component to Dictionary and Migrate Data

Link the Plant component to its component dictionary then migrate data shared by models in the Plant component from the base workspace to the dictionary.

1. Open the model `ex_PlantComp_Lvl1`.
2. In the model, update the diagram.
3. If the Diagnostic View displays an error for multiple inconsistent definitions of `SimConfigSet`, select **Delete others** next to the `GlobalShare` instance. This fix keeps the definition in the `GlobalShare` dictionary and removes other definitions that can be seen by the model.
4. In the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
5. In the dialog box, click **Browse**.

6. Double-click `Plant.sldd`.
7. In the **Model Properties** dialog box, click **Apply**. Click **Change all models** in response to the message about linking referenced models.
8. In the **Model Properties** dialog box, click **Migrate data**.
9. In the Migrate Data dialog box, select **Include data from referenced models** and then click **Migrate**.
10. (Optional) In the **Model Properties** dialog box, clear **Enable model access to base workspace**.
11. Remove the previous method for loading model data. In the **Model Properties** dialog box, on the **Callbacks** tab, clear the `PreLoadFcn` for the model.
12. Click **OK**.

Create Dictionary for Controller Component

Create a data dictionary that contains the data shared by models in the controller component. This dictionary can also reference the shared global dictionary.

1. In the Model Explorer, select **File** > **New** > **Data Dictionary**.
2. Set the new dictionary name to `Controller` and click **Save**.
3. In the **Model Hierarchy** pane, select the node `Controller`. In the Dialog pane, under **Referenced Dictionaries**, click **Add**.
4. Double-click `GlobalShare.sldd`.
5. In the **Model Hierarchy** pane, right-click the node `Controller` and select **Save Changes**.

Link Controller Component to Dictionary and Migrate Data

Link the Controller component to its component dictionary then migrate data shared by models in the Controller component from the base workspace to the dictionary.

1. Open the model `ex_ContrComp`.
2. If the Diagnostic View displays an error for multiple inconsistent definitions of `SimConfigSet`, select **Delete others** next to the `GlobalShare` instance. This fix keeps the definition in the `GlobalShare` dictionary and removes other definitions that can be seen by the model.
3. In the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
4. In the dialog box, click **Browse**.
5. Double-click `Controller.sldd`.
6. In the **Model Properties** dialog box, click **Apply**. Click **Change all models** in response to the message about linking referenced models.
7. In the **Model Properties** dialog box, click **Migrate data**.
8. In the Migrate Data dialog box, select **Include data from referenced models** and then click **Migrate**.
9. (Optional) In the **Model Properties** dialog box, clear **Enable model access to base workspace**.
10. Remove the previous method for loading model data. In the **Model Properties** dialog box, on the **Callbacks** tab, clear the `PreLoadFcn` for the model.
11. Click **OK**.

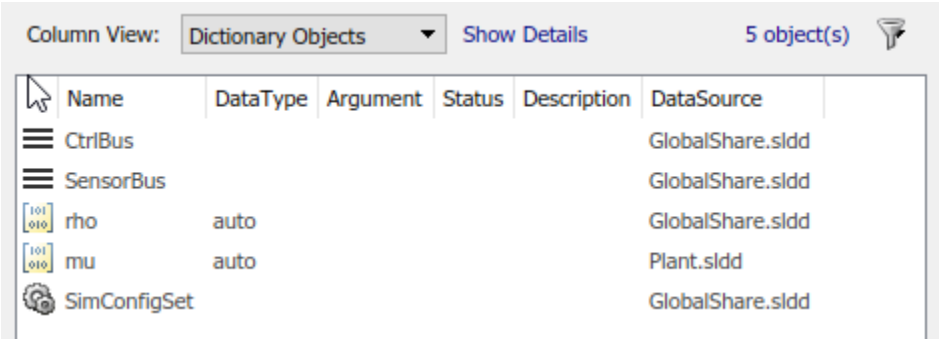
Link System to Global Dictionary

Finally, link the top model to the global dictionary.

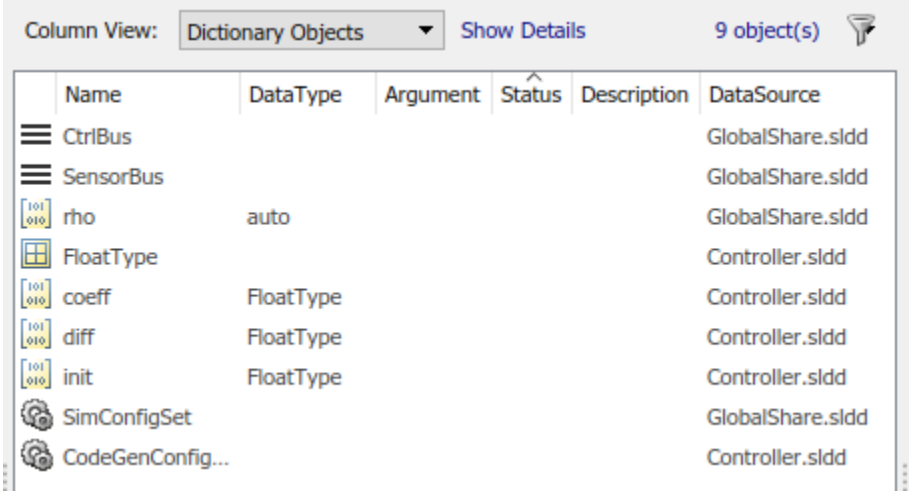
1. Open the model `ex_SystemModel`.
2. In the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
3. In the dialog box, click **Browse**.
4. Double-click `GlobalShare.sldd`.
5. In the **Model Properties** dialog box, click **OK**. Click **Change this model only** in response to the message about linking referenced models.

Inspect Data Storage

In the Model Explorer **Model Hierarchy** pane, select the dictionary node Plant. In the **Contents** pane, to view the contents of `Plant.sldd`, click **Show Current System and Below**  . The contents of the Design Data and Configurations sections appear.



Similarly, view the contents of `Controller.sldd`.



The **DataSource** column shows the variables and objects that each dictionary stores.

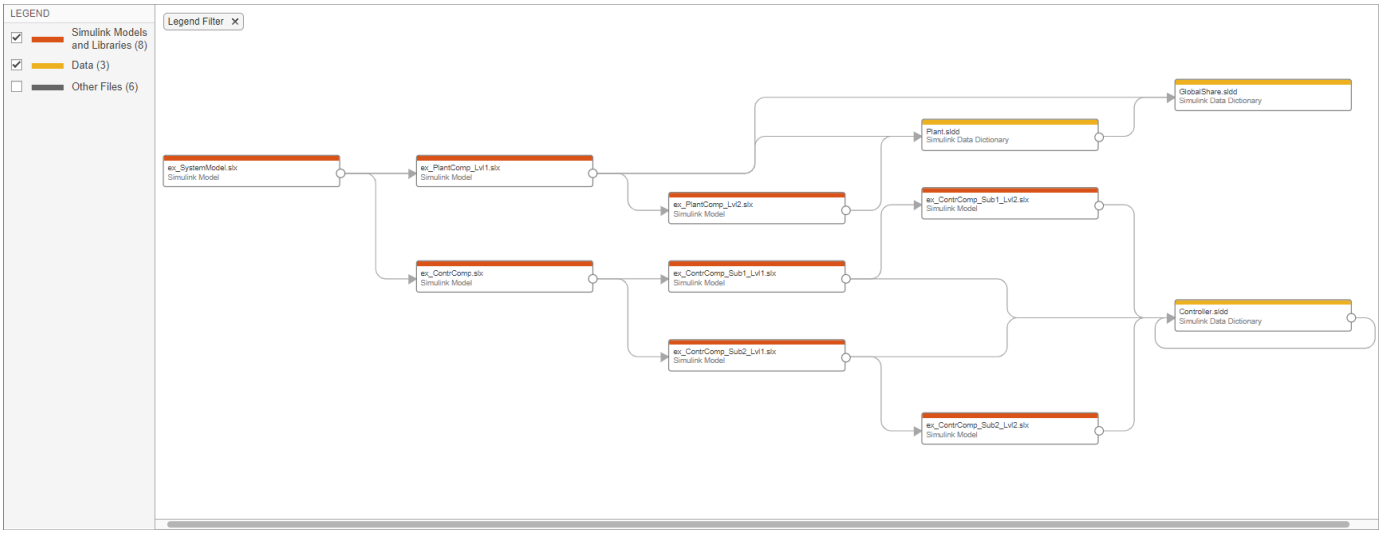
All of the globally shared variables, such as `CtrlBus` and `SensorBus`, reside in `GlobalShare.sldd`. The variable `init`, which both of the controller subcomponents share, resides in `Controller.sldd`.

If the development team assigned to the controller component must make changes to the globally shared variables, they access the `GlobalShare` dictionary file. Similarly, if the team must make changes to the variable `init`, they must access the `Controller` dictionary file.

Inspect Dictionary Hierarchy

To view the entire dictionary and model hierarchy, perform a dependency analysis.

1. Open your saved model `ex_SystemModel`.
2. On the **Modeling** tab, in the **Design** section, click **Dependency Analyzer**.



The system model, `ex_SystemModel`, is linked to the dictionary `GlobalShare.sldd`. The plant component and the controller component are each linked to a separate dictionary. To access the shared data, the component dictionaries reference the dictionary `GlobalShare.sldd`. These dictionaries form a reference hierarchy.

Strategies to Discover Shared Data

To learn how the models in a model reference hierarchy share data, use these techniques:

- In an open model, on the Modeling tab, select **Find** > **Find Ref Variables**. The Model Explorer displays the variables that the model uses, as well as the variables that referenced models use. You can then right-click a variable and select **Find Where Used** to display all of the models that use the variable. For more information, see “Edit and Manage Workspace Variables by Using Model Explorer” on page 67-110.
- At the command prompt, use the function `Simulink.findVars` to determine the variables a model uses. You can then use the function `intersect` to determine the variables two models, components, or subcomponents share.

See Also

Related Examples

- “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100
- “Using a Data Dictionary to Manage the Data for a Fuel Control System”
- “Introduction to Managing Data with Model Reference”
- “Store Data in Dictionary Programmatically” on page 74-34
- “Compare Capabilities of Model Components” on page 22-8
- “What Are Projects?” on page 16-3

Store Data in Dictionary Programmatically

In this section...

A data dictionary stores Simulink model data and offers more data management features than the MATLAB base workspace or the model workspace (see “What Is a Data Dictionary?” on page 74-2). To interact with the data in a dictionary programmatically:

1. Create a `Simulink.data.Dictionary` object that represents the target dictionary.
2. Create a `Simulink.data.dictionary.Section` object that represents the target section, for example the Design Data section. Use the object to interact with the entries stored in the section and to add entries.
3. Optionally, create `Simulink.data.dictionary.Entry` objects that each represent an entry in the target section. Use these objects to interact with individual entries in the target section.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38.

To programmatically interact with the **Embedded Coder** section of a data dictionary, see “Create Code Definitions Programmatically” (Embedded Coder).

Add Entry to Design Data Section of Data Dictionary

1. 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');
   ```

2. Add an entry to the Design Data section of `myDictionary_ex_API.sldd` an entry `myNewEntry` with value `237`.

   ```
   addEntry(dDataSectObj,'myNewEntry',237)
   ```

Rename Data Dictionary Entry

Rename an entry in the Design Data, Configurations, or Other Data section of a data dictionary.

1. 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');
   ```

2. Rename the data dictionary entry.

   ```
   fuelFlowObj.Name = 'fuelFlowNew';
   ```

Increment Value of Data Dictionary Entry

1. 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');
   ```

2. Store the value of the target entry in a temporary variable. Increment the value of the temporary variable by one.

   ```
   temp = getValue(fuelFlowObj);
   temp = temp+1;
   ```

3. Set the value of the target entry by using the temporary variable.

   ```
   setValue(fuelFlowObj,temp)
   ```

Data Dictionary Management

Use `Simulink.data.Dictionary` objects to interact with entire data dictionaries.

Goal	Use
Represent existing data dictionary with `Simulink.data.Dictionary` object	`Simulink.data.dictionary.open`
Create and represent data dictionary with `Simulink.data.Dictionary` object	`Simulink.data.dictionary.create`
Interact with data dictionary	`Simulink.data.Dictionary` class
Import variables to data dictionary from MATLAB base workspace	`importFromBaseWorkspace` method
Add reference dictionary to a data dictionary	`addDataSource` method
Remove reference dictionary from a data dictionary	`removeDataSource` method

Goal	Use
Save changes to data dictionary	`saveChanges` method
Discard changes to data dictionary	`discardChanges` method
View a list of entries stored in data dictionary	`listEntry` method
Import enumerated type definitions to data dictionary	`importEnumTypes` method
Return file name and path of data dictionary	`filepath` method
Show data dictionary in Model Explorer window	`show` method
Hide data dictionary from Model Explorer window	`hide` method
Close connection between data dictionary and `Simulink.data.Dictionary` object	`close` method
Identify data dictionaries that are open	`Simulink.data.dictionary.getOpenDictionaryPaths`
Close all connections to all open data dictionaries	`Simulink.data.dictionary.closeAll`

Dictionary Section Management

Data dictionaries store data as entries contained in sections, and by default all dictionaries have at least three sections named Design Data, Other Data, and Configurations. Use `Simulink.data.dictionary.Section` objects to interact with data dictionary sections.

Goal	Use
Represent data dictionary section with `Section` object.	`getSection` method
Interact with data dictionary section	`Simulink.data.dictionary.Section` class
Import variables to data dictionary section from MAT-file or MATLAB file	`importFromFile` method
Export entries in data dictionary section to MAT-file or MATLAB file	`exportToFile` method
Delete entry from data dictionary section	`deleteEntry` method
Evaluate MATLAB expression in data dictionary section	`evalin` method
Search for entries in data dictionary section	`find` method
Determine whether entry exists in data dictionary section	`exist` method

Dictionary Entry Manipulation

A variable that is stored in a data dictionary is called an entry of the dictionary. Entries have additional properties that store status information, such as the time and date the entry was last modified. Use `Simulink.data.dictionary.Entry` objects to manipulate data dictionary entries.

Goal	Use
Represent data dictionary entry with `Entry` object	`getEntry` method
Add data dictionary entry to section and represent with `Entry` object	`addEntry` method
Manipulate data dictionary entry	`Simulink.data.dictionary.Entry` class
Assign new value to data dictionary entry	`setValue` method
Display changes made to data dictionary entry	`showChanges` method
Save changes made to data dictionary	`saveChanges` method
Discard changes made to data dictionary entry	`discardChanges` method
Search in an array of data dictionary entries	`find` method
Return value of data dictionary entry	`getValue` method
Delete data dictionary entry	`deleteEntry` method
Store enumerated type definition in dictionary	`Simulink.data.dictionary.EnumTypeDefinition` class

Transition to Using Data Dictionary

Using a data dictionary can complicate programmatic interaction with model data. If you link a model to a dictionary:

- You can no longer interact with the model data by using simple commands at the command prompt. Instead, you must use the programmatic interface of the dictionary (`Simulink.data.Dictionary`).
- When you select the dictionary property **Enable dictionary access to base workspace** (see “Continue to Use Shared Data in the Base Workspace” on page 74-10), depending on the storage location of the target data, you must use either simple commands or the programmatic interface.

To help transition from using the base workspace to using data dictionaries, consider using these functions. The functions operate on model data regardless of the storage location of the data.

Goal	Use
Change value of data dictionary entry or workspace variable in context of Simulink model	`Simulink.data.assigninGlobal`

Goal	Use
Evaluate MATLAB expression in context of Simulink model	`Simulink.data.evalinGlobal`
Determine existence of data dictionary entry or workspace variable in context of Simulink model	`Simulink.data.existsInGlobal`

Programmatically Migrate Single Model to Use Dictionary

To change the data source of a Simulink model from the MATLAB base workspace to a new data dictionary, use this example code as a template.

```
% Define the model name and the data dictionary name
modelName = 'f14';
dictionaryName = 'myNewDictionary.sldd';

% Load the target model
load_system(modelName);

% Identify all model variables that are defined in the base workspace
varsToImport = Simulink.findVars(modelName,'SourceType','base workspace');
varNames = {varsToImport.Name};

% Create the data dictionary
dictionaryObj = Simulink.data.dictionary.create(dictionaryName);

% Import to the dictionary the model variables defined in the base
% workspace, and clear the variables from the base workspace
[importSuccess,importFailure] = importFromBaseWorkspace(dictionaryObj,...
    'varList',varNames,'clearWorkspaceVars',true);

% Link the dictionary to the model
set_param(modelName,'DataDictionary',dictionaryName);
```

Note This code does not migrate the definitions of enumerated data types that were used to define model variables. If you import model variables of enumerated data types to a data dictionary but do not migrate the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else. To migrate enumerated data type definitions to a data dictionary, see “Enumerations in Data Dictionary” on page 74-12.

Import Directly From External File to Dictionary

This example shows how to use a custom MATLAB function to import data directly from an external file to a data dictionary without creating or altering variables in the base workspace.

1. Create a two-dimensional lookup table in one sheet of a Microsoft Excel workbook. Use the upper-left corner of the sheet to provide names for the two breakpoints and for the table. Use column B and row 2 to store the two breakpoints, and use the rest of the sheet to store the table. For example, your lookup table might look like this:

	A	B	C	D	E	F	G	H
1		bkpt2Name						
2	bkpt1Name	tableName	0.1	0.2	0.3	0.4	0.5	
3		0.25	0.35	0.6	0.85	1.1	1.35	
4		0.5	0.45	0.7	0.95	1.2	1.45	
5		0.75	0.55	0.8	1.05	1.3	1.55	
6		1	0.65	0.9	1.15	1.4	1.65	
7		1.25	0.75	1	1.25	1.5	1.75	
8								

Save the workbook in your current folder as `my2DLUT.xlsx`.

2. Copy this custom function definition into a MATLAB file, and save the file in your current folder as `importLUTToDD.m`.

```
function importLUTToDD(workbookFile,dictionaryName)
    % IMPORTLUTTODD(workbookFile,dictionaryName) imports data for a
    % two-dimensional lookup table from a workbook directly into a data
    % dictionary. The two-dimensional lookup table in the workbook can be
    % any size but must follow a standard format.

    % Read in the entire first sheet of the workbook.
    [data,names,~] = xlsread(workbookFile,1,'');

    % Divide the raw imported data into the breakpoints, the table, and their
    % names.
    % Assume breakpoint 1 is in the first column and breakpoint 2 is in the
    % first row.
    % Assume cells A2, B1, and B2 define the breakpoint names and table name.
    bkpt1 = data(2:end,1);
    bkpt2 = data(1,2:end);
    table = data(2:end,2:end);
    bkpt1Name = names{2,1};
    bkpt2Name = names{1,2};
    tableName = names{2,2};

    % Prepare to import to the Design Data section of the target data
    % dictionary.
    myDictionaryObj = Simulink.data.dictionary.open(dictionaryName);
    dDataSectObj = getSection(myDictionaryObj,'Design Data');

    % Create entries in the dictionary to store the imported breakpoints and
    % table. Name the entries using the breakpoint and table names imported
    % from the workbook.
    addEntry(dDataSectObj,bkpt1Name,bkpt1);
    addEntry(dDataSectObj,bkpt2Name,bkpt2);
    addEntry(dDataSectObj,tableName,table);

    % Save changes to the dictionary and close it.
    saveChanges(myDictionaryObj)
    close(myDictionaryObj)
```

3. At the MATLAB command prompt, create a data dictionary to store the lookup table data.

```
myDictionaryObj = Simulink.data.dictionary.create('myLUTDD.sldd');
```

4. Call the custom function to import your lookup table to the new data dictionary.

```
importLUTToDD('my2DLUT.xlsx','myLUTDD.sldd')
```

5. Open the data dictionary in Model Explorer.

```
show(myDictionaryObj)
```

Three new entries store the imported breakpoints and lookup table. These entries are ready to use in a 2-D Lookup Table block.

Programmatically Partition Data Dictionary

To partition a data dictionary into reference dictionaries, use this example code as a template. You can use reference dictionaries to make large data dictionaries more manageable and to contain standardized data that is useful for multiple models.

```
% Define the names of a parent data dictionary and two
% reference data dictionaries
parentDDName = 'myParentDictionary.sldd';
typesDDName = 'myTypesDictionary.sldd';
paramsDDName = 'myParamsDictionary.sldd';

% Create the parent data dictionary and a
% Simulink.data.Dictionary object to represent it
parentDD = Simulink.data.dictionary.create(parentDDName);

% Create a Simulink.data.dictionary.Section object to represent
% the Design Data section of the parent dictionary
designData_parentDD = getSection(parentDD,'Design Data');

% Import some data to the parent dictionary from the file partDD_Data_ex_API.m
importFromFile(designData_parentDD,'partDD_Data_ex_API.m');

% Create two reference dictionaries
Simulink.data.dictionary.create(typesDDName);
Simulink.data.dictionary.create(paramsDDName);

% Create a reference dictionary hierarchy by adding reference dictionaries
% to the parent dictionary
addDataSource(parentDD,typesDDName);
addDataSource(parentDD,paramsDDName);

% Migrate all Simulink.Parameter objects from the parent data dictionary to
% a reference dictionary
paramEntries = find(designData_parentDD,'-value','-class','Simulink.Parameter');
for i = 1:length(paramEntries)
    paramEntries(i).DataSource = 'myParamsDictionary.sldd';
end

% Migrate all Simulink.NumericType objects from the parent data dictionary
% to a reference dictionary
typeEntries = find(designData_parentDD,'-value','-class','Simulink.NumericType');
for i = 1:length(typeEntries)
    typeEntries(i).DataSource = 'myTypesDictionary.sldd';
end

% Save all changes to the parent data dictionary
saveChanges(parentDD)
```

Make Changes to Configuration Set Stored in Dictionary

You can store a configuration set (a `Simulink.ConfigSet` object) in the Configurations section of a dictionary. To change the setting of a configuration parameter in the set programmatically:

1. Create a `Simulink.data.dictionary.Entry` object that represents the configuration set (which is an entry in the dictionary). For example, suppose the name of the dictionary is `myData.sldd` and the name of the `Simulink.ConfigSet` object is `myConfigs`.

   ```
   dictionaryObj = Simulink.data.dictionary.open('myData.sldd');
   configsSectObj = getSection(dictionaryObj,'Configurations');
   entryObj = getEntry(configsSectObj,'myConfigs');
   ```

2. Store a copy of the target `Simulink.ConfigSet` object in a temporary variable.

   ```
   temp = getValue(entryObj);
   ```

3. In the temporary variable, modify the target configuration parameter (in this case, set **Stop time** to `20`).

   ```
   set_param(temp,'StopTime','20');
   ```

4. Use the temporary variable to overwrite the configuration set in the dictionary.

   ```
   setValue(entryObj,temp);
   ```

5. Save changes made to the dictionary.

   ```
   saveChanges(dictionaryObj)
   ```

See Also

`Simulink.data.dictionary.cleanupWorkerCache` | `Simulink.data.dictionary.setupWorkerCache` | `Simulink.findVars` | `set_param`

Related Examples

- “Enumerations in Data Dictionary” on page 74-12
- “Migrate Model Reference Hierarchy to Use Dictionary” on page 74-6
- “What Is a Data Dictionary?” on page 74-2
- “Optimize, Estimate, and Sweep Block Parameter Values” on page 37-38

Managing Signals

Working with Signals

Signal Basics

In this section...

A *signal* is a time-varying quantity that has values at all points in time. You can specify a wide range of signal attributes, including:

- Signal name
- Data type (for example, 8-bit, 16-bit, or 32-bit integer)
- Numeric type (real or complex)
- Dimensionality (one-dimensional, two-dimensional, or multidimensional array)

In Simulink, signals are the outputs of dynamic systems represented by blocks in a Simulink diagram and by the diagram itself. The lines in a block diagram represent mathematical relationships among the signals defined by the block diagram. For example, a line connecting the output of block A to the input of block B indicates that the signal output of B depends on the signal output of A.

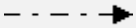
Simulink block diagrams represent signals with lines that have an arrowhead. The source of the signal corresponds to the block that writes to the signal during evaluation of its block methods (equations). The destinations of the signal are blocks that read the signal during the evaluation of the block methods (equations). The destination of signals in a model do not necessarily represent the order of simulation of blocks in a model. The simulation order is determined by Simulink automatically.

Note Simulink signals are mathematical, not physical, entities. The lines in a block diagram represent mathematical, not physical, relationships among blocks. Simulink signals do not travel along the lines that connect blocks in the same way that electrical signals travel along a wire. Block diagrams do not represent physical connections between blocks.

You can create a signal by adding a source block to your model. For example, you can create a signal that varies sinusoidally with time by adding an instance of the Sine, Cosine block from the Simulink Sources library into your model. To see a list of the blocks that create signals in a model, see “Sources”. Alternatively, you can use the “Viewers and Generators Manager” on page 28-77 to create signals in your model without using blocks.

Signal Line Styles

A Simulink model can include many different types of signals. As you construct a block diagram, all signal types appear as a thin, solid line. After you update the diagram or start simulation, the signals appear with the specified line styles. These signal types enable you to differentiate between different signal types. From all signal types, you can only customize the nonscalar signal type. To learn more, see “Signal Types” on page 75-7.

Signal Type	Line Style
Scalar and nonscalar	
Nonscalar (with the **Wide nonscalar lines** option enabled—see “Wide Nonscalar Lines” on page 75-49)	
Control signal	
Virtual bus	
Nonvirtual bus	
Array of buses	
Variable-size	

Signal Properties

You may want to specify signal properties in your model to give a name or a label to your signals, prepare data for logging, or to customize your signals in a model. Use the Property Inspector, the Model Data Editor, or the Signal Properties dialog box to specify properties for:

- Signal names and labels
- Signal logging
- Simulink Coder to use to generate code
- Documentation of the signal

To access the signal properties in the Property Inspector, first display the Property Inspector. On the **Modeling** tab, under **Design**, click **Property Inspector**. When you select a signal, the properties appear in the Property Inspector. To use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), inspect the **Signals** tab and select a signal. To use the Signal Properties dialog box, right-click a signal and select **Properties**. For information about the benefits of each approach, see “Add Blocks and Set Parameters” on page 1-13.

To specify signal properties programmatically, use a function such as `get_param` to create a variable that holds the handle to the block output port that creates the signal line. Then, use `set_param` to set the programmatic parameters of the port. For example:

```
p = get_param(gcb,'PortHandles')
l = get_param(p.Outport,'Line')
set_param(l,'Name','s9')
```

Signal Names and Labels

You can name a signal interactively or programmatically in a model. The syntactic requirements for a signal name depend on how you use the name. The most common cases are:

- Do not use a less than character (<) to start a signal name.
- The signal name can resolve to a `Simulink.Signal` object. (See `Simulink.Signal`.) The signal name must then be a legal MATLAB identifier. This identifier starts with an alphabetic character,

followed by alphanumeric or underscore characters up to the length given by the function `namelengthmax`.

- The signal has a name so the signal can be identified and referenced by name in a data log. (See “Export Signal Data Using Signal Logging” on page 72-41.) Such a signal name can contain space and newline characters. These characters can improve readability but sometimes require special handling techniques, as described in “Handling Spaces and Newlines in Logged Names” on page 72-68
- The signal name exists only to clarify the diagram and has no computational significance. Such a signal name can contain anything and does not need special handling.
- The signal is an element of a bus object. Use a valid C language identifier for the signal name.
- 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.

Making every signal name a legal MATLAB identifier handles a wide range of model configurations. Unexpected requirements can require changing signal names to follow a more restrictive syntax. You can use the function `isvarname` to determine whether a signal name is a legal MATLAB identifier.

Name a signal interactively by:

- Using the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**)
- Using the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**) **Signals** tab
- From the Signal Properties dialog box.

The signal name appears below a signal, displayed as a signal label.

To name a signal programmatically, use the `get_param` and `set_param` functions on the signal. Table below summarizes how to work with signal names and labels in the Simulink Editor.

Task	Action
Name a signal line	Double-click the signal and type its name.
Name a branch of a named signal line	Double-click the branch.
Name every branch of a signal	Right-click the signal, select **Properties**, and use the dialog box.
Delete signal label and name	Delete characters in the label or delete the name in Signal Properties dialog box.
Delete signal label only	Right-click the label and select **Delete Label**.
Open signal label text box for editing	Double-click the signal line. Click the label. Select the signal line (not the label) and use **F2**. On Macintosh platforms, select the signal line (not the label) and use **control+return**.
Move signal label	Drag the label to a new location on the same signal line.
Copy signal label	**Ctrl+**drag the signal label.

Task	Action
Change the label font	Select the signal line (not the label), and then on the **Format** tab, click the **Font Properties** button arrow, then click **Fonts for Model**.

Signal Display Options

Displaying signal attributes in the model diagram can make the model easier to read. For example, in the Simulink Editor, on the **Debug** tab, use the **Information Overlays** menu to include in the model layout information about signal attributes, such as:

- Port data types
- Design ranges
- Signal dimensions
- Signal resolution

For details, see “Display Signal Attributes” on page 75-45.

You can also highlight a signal and its source or destination blocks. For details, see “Highlight Signal Sources and Destinations” on page 75-25.

Store Design Attributes of Signals and States

You can use block parameters and signal properties to specify signal design attributes such as data type, minimum and maximum values, physical unit, and numeric complexity. To configure states, you can use block parameters. When you use these block parameters and signal properties, you store the specifications in the model file.

Alternatively, you can specify these attributes by using the properties of a `Simulink.Signal` object that you store in a workspace or data dictionary. See `Simulink.Signal` and “Data Objects” on page 67-58.

Choose which strategy to use based on your modeling goals.

- To improve model portability, readability, and ease of maintenance, store these specifications in the model file. Use the Property Inspector, the Model Data Editor, block dialog boxes, and signal properties dialog boxes to access the parameters and properties. You do not need to save and manage external `Simulink.Signal` objects. Consider setting the model configuration parameter **Signal resolution** to `None`, which disables the use of `Simulink.Signal` objects by the model.

  To configure design attributes and code generation settings for signals by using a list that you can sort, group, and filter, consider the Model Data Editor. With this tool, you store the specifications in the model file instead of using `Simulink.Signal` objects. See “Configure Data Properties by Using the Model Data Editor” on page 67-131.
- To separate these specifications from the model so that you can manage each independently, use `Simulink.Signal` objects. You can then configure the specifications in a flat list that you can sort, group, and filter with the Model Data Editor or the Model Explorer. To determine where to permanently store the objects, see “Determine Where to Store Variables and Objects for Simulink Models” on page 67-100.

Test Signals

You can perform the following kinds of tests on signals:

- “Minimum and Maximum Values” on page 75-6
- “Connection Validation” on page 75-6

Minimum and Maximum Values

For many Simulink blocks, you can specify a range of valid values for the output signals. Simulink provides a diagnostic for detecting when blocks generate signals that exceed their specified ranges during simulation. For details, see “Specify Signal Ranges” on page 75-31.

Connection Validation

Many Simulink blocks have limitations on the types of signals that they accept. Before simulating a model, Simulink checks all blocks to ensure that the blocks can accommodate the types of signals output by the ports to which the blocks connect and reports errors about incompatibilities.

To detect signal compatibility errors before running a simulation, update the diagram.

Scenarios

The Signal Editor block displays interchangeable groups of scenarios. Use the Signal Editor to display, create, edit, and switch interchangeable scenarios.

Scenarios can help with testing a model.

See Also

Related Examples

- “Control Signal Data Types” on page 67-6
- “Signal Label Propagation” on page 75-12
- “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61
- “Merging Signals”

Signal Types

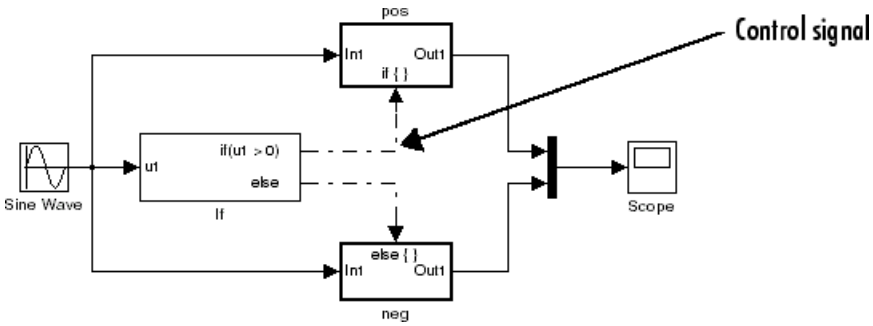
Multiple types of signals can connect the blocks in a model. For example, a model may contain a control signal to initiate the execution of another block and buses to simplify line routing.

The following table summarizes the types of Simulink signals.

Name	Description
Array	Composite signal that provides index-based signal access.
Array of Buses	Concatenated signal that contains nonvirtual buses.
Bus	Composite signal that provides name-based signal access.
Composite Signal	Signal composed of other signals. See “Composite Signals” on page 75-8.
Concatenated Signal	Nonvirtual composite signal that provides index-based signal access.
Control Signal	Signal used by a block to initiate the execution of another block. For example, a signal that executes a function-call subsystem or an action subsystem. See “Control Signals” on page 75-7.
Matrix	Two-dimensional composite signal that provides index-based signal access.
Multidimensional (*N*-D) Signal	Composite signal with more than two dimensions that provides index-based signal access.
Mux Signal	Virtual composite signal that provides index-based signal access. Also known as a virtual vector.
Nonscalar Signal	Signal with at least one dimension, such as a vector (1-D), matrix (2-D), or multidimensional array (*N*-D). Nonscalar signals are a type of composite signal.
Nonvirtual Signal	Signal that affects simulation and code generation. See “Virtual and Nonvirtual Signals” on page 75-8.
Scalar Signal	Signal that supports only one value at a time.
Variable-Size Signal	Signal whose size (the number of elements in a dimension), in addition to its values, can change during a model simulation.
Vector	One-dimensional composite signal that provides index-based signal access.
Virtual Signal	Signal that represents another signal or set of signals. A virtual signal is used for graphical purposes and has no functional effect. See “Virtual and Nonvirtual Signals” on page 75-8.

Control Signals

A *control signal* is a signal used by a block to initiate the execution of another block. For example, a signal that executes a function-call or action subsystem is a control signal. When you update or simulate a block diagram, Simulink uses a dash-dot pattern to draw lines that represent the control signals.



Composite Signals

You can group multiple signals in a composite signal, route the signal from block to block, and extract the constituent signals where needed. When you have many parallel signals, composite signals can simplify the appearance of a model and help to clarify generated code. Composite signals can be virtual or nonvirtual.

Types of composite signals include:

- Virtual buses
- Nonvirtual buses
- Mux signals
- Concatenated signals
- Arrays of buses

For more information on composite signals, see “Types of Composite Signals” on page 76-2.

Virtual and Nonvirtual Signals

A virtual signal is a signal that graphically represents other signals or parts of other signals. Virtual signals are purely graphical entities; Simulink ignores them when simulating a model, and they do not exist in generated code. Some blocks, such as the Mux block, always generate virtual signals. Others, such as the Bus Creator block, can generate either virtual or nonvirtual signals. A signal is virtual if the block that generates it is virtual.

A nonvirtual signal is a signal that affects simulation and code generation. Some blocks, such as the Vector Concatenate block, always generate nonvirtual signals. A signal is nonvirtual if the block that generates it is nonvirtual.

For information on virtual and nonvirtual blocks, see “Nonvirtual and Virtual Blocks” on page 36-2.

See Also

Related Examples

- “Display Signal Attributes” on page 75-45
- “Control Signal Data Types” on page 67-6
- “Signal Basics” on page 75-2

Investigate Signal Values

In this section...

While transmitting valuable data about a model and between the blocks, signals may take different values. In this section, you learn how to initialize, display signals as well as more information on the data types and dimensions a signal can take.

Initialize Signal Values

If a signal does not have an explicit initial value, the initial value that Simulink uses depends on the data type of the signal.

Signal Data Type	Default Initial Value
Numeric (other than fixed-point)	Zero
Fixed-point	Real-world ground value
Boolean	False
Enumerated	Default value

You can specify the non-default initial values of signals for Simulink to use at the beginning of simulation.

- For any signal, you can define a signal object (`Simulink.Signal`), and use that signal object to specify an initial value for the signal.
- For some blocks, such as Outport, Data Store Memory, and Memory, you can use either a signal object or a block parameter, or both, to specify the initial value of a block state or output.

For details, see “Initialize Signals and Discrete States” on page 75-37.

View Signal Values

You can use either blocks or the signal viewers (such as the Signal & Scope Manager) to display the values of signals during a simulation. For example, you can use either the Scope block or the Signal & Scope Manager to graph time-varying signals on an oscilloscope-like display during simulation. For general information about options for viewing signal values, see “Scope Blocks and Scope Viewer Overview” on page 28-6. For detailed information about:

- Blocks that you can use to display signals in a model, see “Sinks”
- Signal viewers, “Floating Scope and Scope Viewer Tasks” on page 28-67
- The Signal & Scope Manager, see “Viewers and Generators Manager” on page 28-77

- Test points, which are signals that Simulink guarantees to be observable when using a Floating Scope block in a model, see “Configure Signals as Test Points” on page 75-43.

Display Signal Values in Model Diagrams

To include graphical displays of signal values in a model diagram, use one of the following approaches:

Display Data Tips During Simulation

For many blocks, Simulink can display block output (port values) as data tips on the block diagram while a simulation is running.

1. In the Simulink Editor, on the **Debug** tab, select **Output Values** , go to the **Signal** tab, and toggle on the **Output Value Label > Toggle Value Displays** button.
2. To change display options, use the **Options** submenu.

For details, see “Display Port Values for Debugging” on page 36-16.

Display Signal Value After Simulation

To display, below a specific signal, the signal value after simulation:

1. Right-click the signal.
2. In the context menu, select **Show Value Label of Selected Port**.

Signal Data Types

Data type refers to the format used to represent signal values internally. By default, the data type of Simulink signals is double. You can create signals of other data types. Simulink signals support the same range of data types as MATLAB. See “About Data Types in Simulink” on page 67-2 for more information.

Complex Signals

The values of signals can be complex numbers or real numbers. A signal whose values are complex numbers is a complex signal. Create a complex-valued signal using one of the following approaches:

- Load complex-valued signal data from the MATLAB workspace into the model via a root-level Inport block.
- Create a Constant block in your model and set its value to a complex number.
- Create real signals corresponding to the real and imaginary parts of a complex signal, then combine the parts into a complex signal, using the Real-Imag to Complex conversion block.

Manipulate complex signals via blocks that accept them. If you are not sure whether a block accepts complex signals, see the documentation for the block.

Exporting Signal Data

You can save signal values to the MATLAB workspace during simulation, for later retrieval and postprocessing. For a summary of different approaches, see “Approaches for Exporting Signal Data” on page 72-2.

See Also

Related Examples

- “Control Signal Data Types” on page 67-6
- “Initialize Signals and Discrete States” on page 75-37
- “Signal Basics” on page 75-2
- “Signal Types” on page 75-7
- “Specify Signal Ranges” on page 75-31
- “Configure Signals as Test Points” on page 75-43

Signal Label Propagation

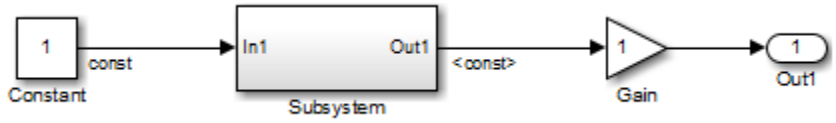
In this section...

You can give signals signal names and configure propagating the signal names in a Simulink model to track a signal label through one block or many blocks. For a list of supported blocks, see “Blocks That Support Signal Label Propagation” on page 75-12.

When you name a signal and enable the display of signal label propagation for output signals of allowed blocks:

- If there is a user-specified signal name that Simulink can propagate, the propagated signal label includes the name in angle brackets (for example, `<sig1>`).
- If there is no signal name to propagate, Simulink displays an empty set of angle brackets (`<>`) for the label.

For example, in the following model, the output signal from the Subsystem block is configured for signal label propagation. The propagated signal label (`<const>`) is based on the name of the upstream output signal of the Constant block (`const`).



For more information on how Simulink creates propagated signal labels, see “How Simulink Propagates Signal Labels” on page 75-13.

Blocks That Support Signal Label Propagation

You can use signal label propagation with output signals for several *connection* blocks, which route signals through the model without changing the data. Connection blocks perform no signal transformation.

Also, Model blocks support signal label propagation.

The connection blocks that support signal label propagation are:

- Enable
- From
- Function-Call Split
- Goto
- Inport (subsystem only; not model input ports)
- Signal Specification

- Subsystem, Atomic Subsystem, CodeReuse Subsystem (through subsystem Inport and Outport blocks)
- Trigger

The Bus Creator and Bus Selector blocks do *not* support signal label propagation. However, if you want to view the hierarchy for any bus signal, use the “Display Bus Hierarchy” on page 76-31.

The Signal Properties dialog box for a signal indicates whether that signal supports signal label propagation. The **Show propagated signals** parameter is available only for blocks that support signal label propagation. For details, see “Display Propagated Signal Labels” on page 75-16.

How Simulink Propagates Signal Labels

In general, Simulink performs signal label propagation consistently:

- For different modeling constructs (for example, non-bus and bus signals, virtual and nonvirtual buses, subsystem and model variants, model referencing, and libraries)
- In models with or without hidden blocks, which Simulink inserts in certain cases to enable simulation
- At model load, edit, update, and simulation times

For information about some special cases, see:

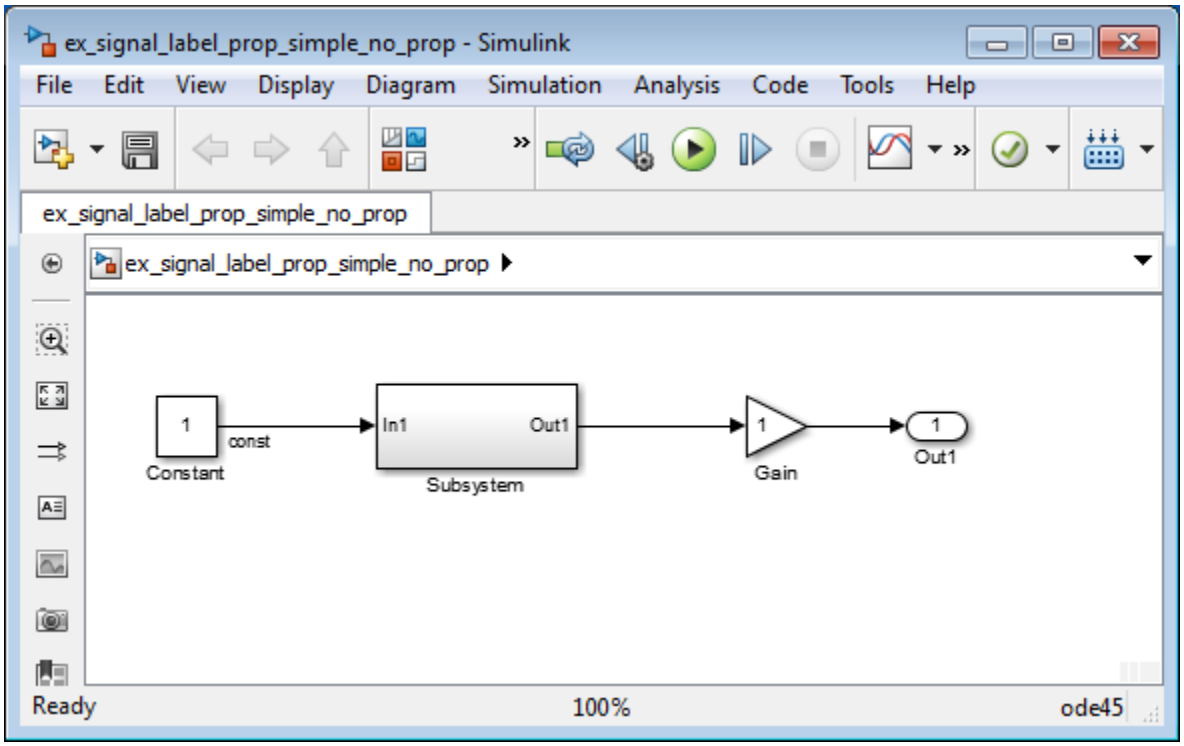
- “Processing for Referenced Models” on page 75-17
- “Processing for Variants and Configurable Subsystems” on page 75-18

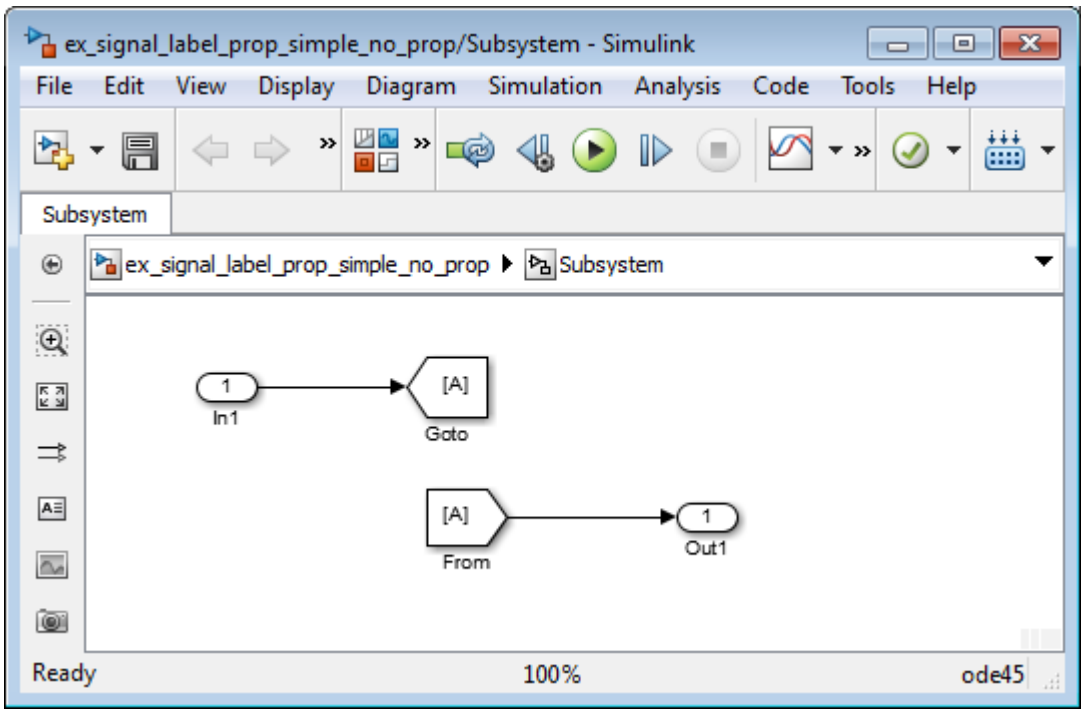
General Signal Label Propagation Processing

In general, when you enable signal label propagation for an output signal of a block (for example, BlockA), Simulink performs the following processing to find the source signal name to propagate:

1. Checks the block whose output signal connects to BlockA, and if necessary, continues checking upstream blocks, working backward from the closest block to the farthest block.
2. Stops when it encounters a block that either:
   - Supports signal label propagation and has a signal name
   - Does not support signal label propagation
3. Obtains the signal name, if any, of the output signal for the block at which Simulink stops.
4. Uses that signal name for the propagated signal label of any output signals of downstream blocks for which you enable signal label propagation.

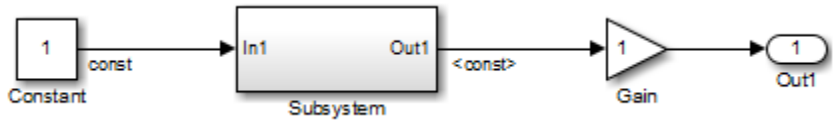
For example, in the following model, suppose that you enable signal label propagation for the output signal for the Subsystem block (that is, the signal connected to the `Out1` port).



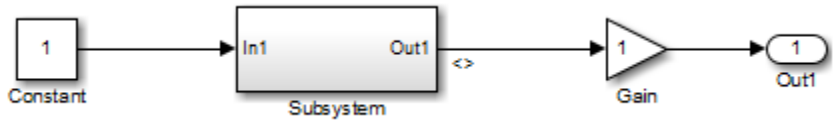


Simulink checks inside the subsystem, checks upstream from the From and GoTo blocks (which support signal label propagation and do not have a name), and then checks farther upstream, to the Constant block, which does not support signal label propagation.

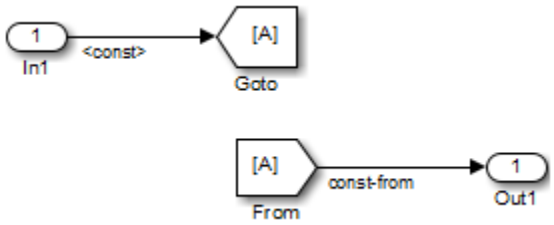
Simulink uses the signal name of the Constant block output signal, `const`. The propagated signal label for the Subsystem block output signal is `<const>`.



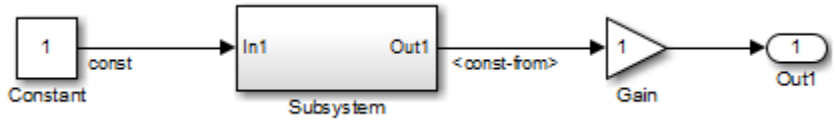
If the output signal from the Constant block did not have a signal name, then the propagated signal label would be an empty set of angle brackets (<>).



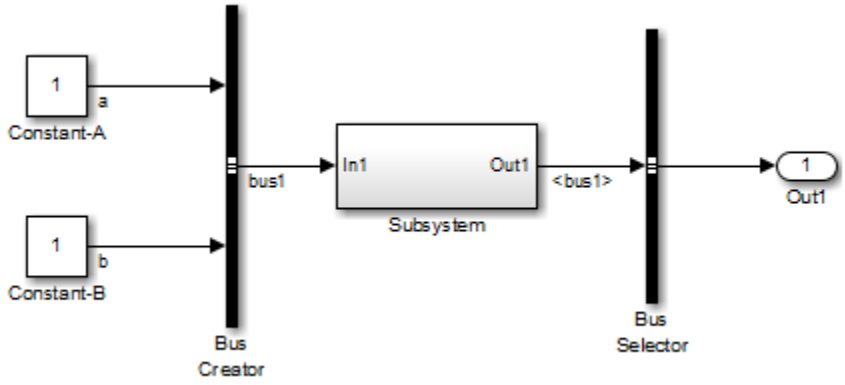
Suppose that in the Subsystem block you enable signal label propagation for the output signal from the In1 block, and you use the Signal Properties dialog box to specify the signal name `const-from` for the output signal of the From block, as shown below.



The propagated signal label for the Subsystem block output signal changes to `<const-from>`, because that is the first named signal that Simulink encounters in its signal label propagation processing.



In the following model, the signal label propagation for the output signal of the Subsystem block uses the signal name `bus1`, which is the name of the output bus signal of the Bus Creator block. The propagated signal label does not include the names of the bus element signals (`a` and `b`).

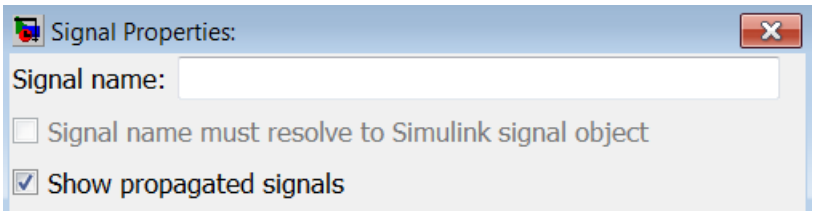


Display Propagated Signal Labels

You can display propagated signal labels for individual signals, or display the propagated signal labels for all signals in a model. To display the labels for all signals, in the Simulink Editor, on the **Debug** tab, select **Information Overlays > Propagated Signal Labels**.

To display a propagated signal label for an individual signal:

1. Right-click the signal for which you want to display a propagated signal label and select **Properties**.
2. In the Signal Properties dialog box, select **Show propagated signals**.



The **Show propagated signals** parameter is available only for output signals from blocks that support signal label propagation.

To enable this signal property programmatically, create a handle to the signal line, and specify `signalPropagation` as `'on'`. For example, you can use this code to enable or disable the property for all of the signals in a model diagram.

```
% Create an array of handles to every signal line in the diagram
signalLines = find_system(gcs,'FindAll','on','type','line');

% Enable or disable the property for each signal line
for i = 1:length(signalLines)

      % set(signalLines(i),'signalPropagation','off');
      set(signalLines(i),'signalPropagation','on');
end
```

If a signal already has a label, then an *alternative* approach for displaying a propagated signal label is:

1. In the model diagram, click the signal label.
2. Remove the label text.
3. In the signal label text box, enter an angle bracket (<).
4. Click outside the signal label.

   Simulink displays the propagated signal label.

Special Cases of Signal Propagation

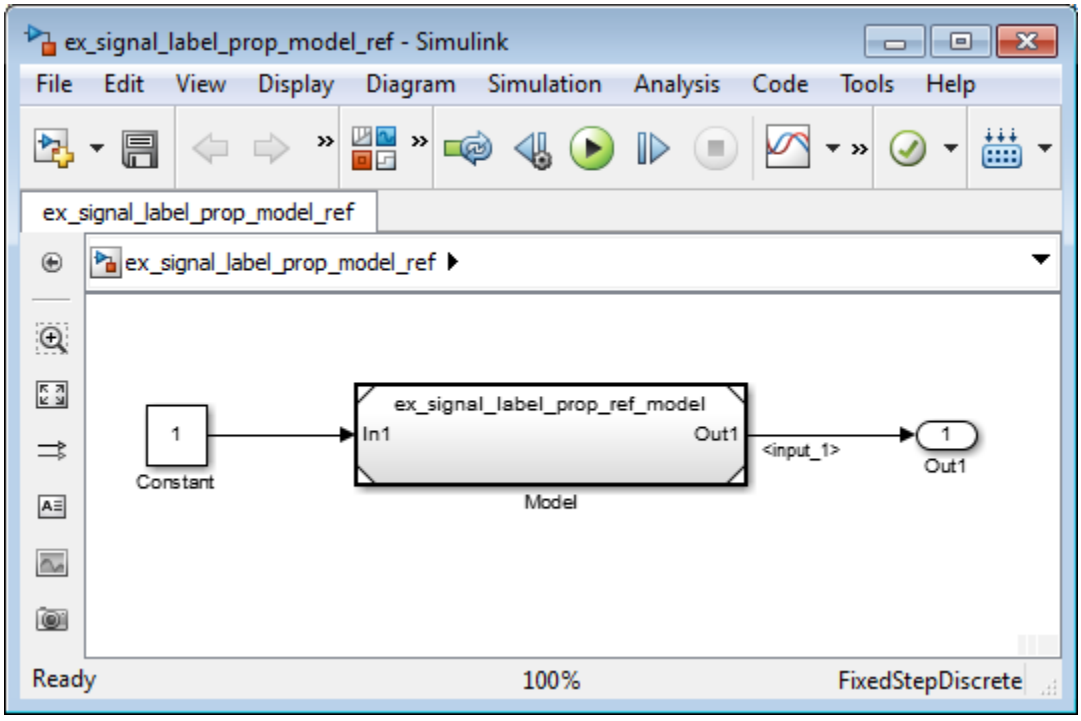
Processing for Referenced Models

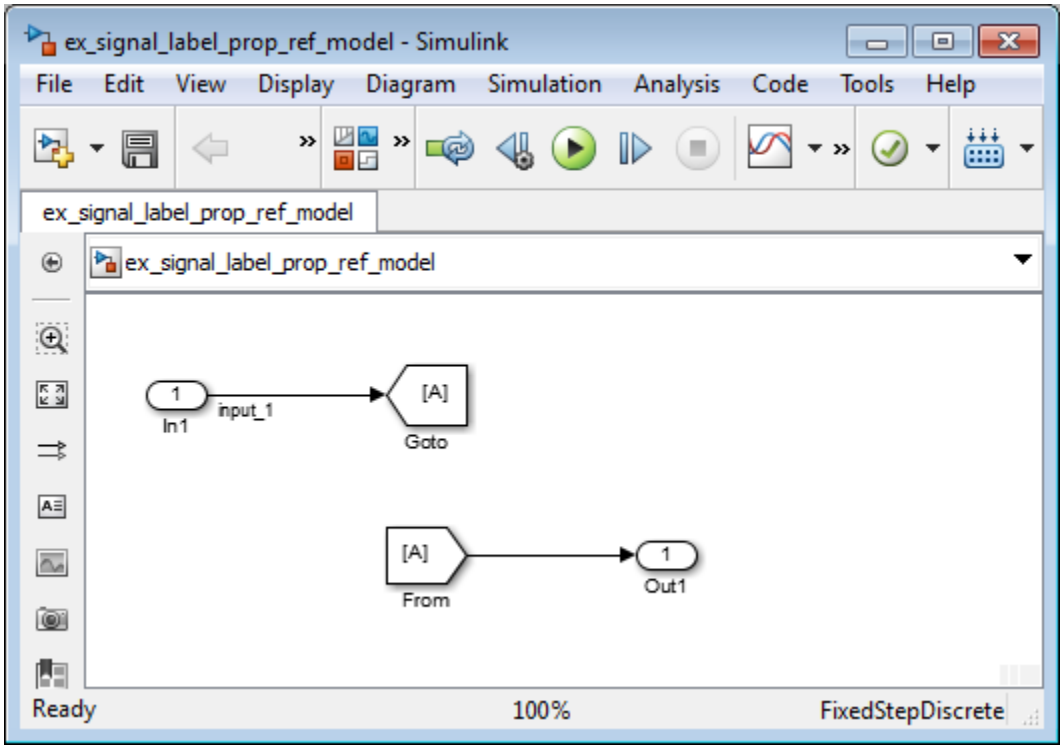
To enable signal label propagation for referenced models, in addition to the steps described in “Display Propagated Signal Labels” on page 75-16, use the default setting for the **Model Configuration Parameters > Model Referencing > Propagate all signal labels out of the model** parameter. In other words, make sure the parameter is enabled.

If you make a change inside a referenced model that affects signal label propagation, the propagated signal labels outside of the referenced model do not reflect those changes until after you update the diagram or simulate the model.

For example, the model `ex_signal_label_prop_model_ref` has a referenced model that includes an output signal from the In1 block that has a signal name of `input_1`.

If you enable signal label propagation for the signal from the Out1 port of the Model block, that signal does *not* reflect the name `input_1` until after you update the diagram or simulate the model.





Processing for Variants and Configurable Subsystems

Simulink updates the propagated signal label (if enabled) for the output signal of the Subsystem or Model block, when *both* of these conditions occur:

- The output signals for variant models have different signal names.
- You change the active variant model or variant subsystem.

For Subsystem blocks, the signal label updates at edit time. For Model blocks, the update occurs when you update diagram or simulate the model.

See Also

Related Examples

- “Display Signal Attributes” on page 75-45
- “Highlight Signal Sources and Destinations” on page 75-25
- “Signal Basics” on page 75-2
- “Signal Types” on page 75-7

Determine Signal Dimensions

In this section...

Simulink blocks can output one-dimensional, two-dimensional, or multidimensional signals. The Simulink user interface and documentation generally refer to 1-D signals as *vectors* and 2-D or multidimensional signals as *matrices*. A one-element array is frequently referred to as a *scalar*. A *row vector* is a 2-D array that has one row. A *column vector* is a 2-D array that has one column.

- A one-dimensional (1-D) signal consists of a series of one-dimensional arrays output at a frequency of one array (vector) per simulation time step.
- A two-dimensional (2-D) signal consists of a series of two-dimensional arrays output at a frequency of one 2-D array (matrix) per block sample time.
- A multidimensional signal consists of a series of multidimensional (two or more dimensions) arrays output at a frequency of one array per block sample time. You can specify multidimensional arrays with any valid MATLAB multidimensional expression, such as [4 3]. See “Multidimensional Arrays” for information on multidimensional arrays.

Simulink blocks vary in the dimensionality of the signals they can accept or output. Some blocks can accept or output signals of any dimension. Some can accept or output only scalar or vector signals.

Note Simulink does not support dynamic signal dimensions during a simulation. That is, the dimension of a signal must remain constant while a simulation is executing. However, you can change the size of a signal during a simulation. See “Variable-Size Signal Basics” on page 77-2.

If a block can emit nonscalar signals, the dimensions of the signals that the block outputs depend on the block parameters, if the block is a source block; otherwise, the output dimensions depend on the dimensions of the block input and parameters.

To determine the dimensions that a signal ultimately uses for simulation, first update the block diagram (for example, by pressing **Ctrl+D**). Then, choose one of these techniques:

- Display the dimensions directly on the block diagram. Use this technique to trace signal dimensions along a path of blocks. In the model, on the **Debug** tab, select **Information Overlays** > **Signal Dimensions**).
- Inspect the dimensions in the Model Data Editor, which shows you information in a searchable, sortable table. In the table, the right side of each cell in the **Dimensions** column shows the true dimensions of the corresponding signal line in the model. For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

Simulink Blocks that Support Multidimensional Signals

The Simulink Block Data Type Support table includes a column identifying the blocks with multi-dimension signal support.

1. In the MATLAB command line, enter `showblockdatatypetable`.

   A separate window with the Simulink Block Data Type Support table opens.
2. In the Block column, locate the name of a Simulink block. Columns to the right are data types or features. An a **X** in a column indicates support for that feature.

Simulink supports signals with up to 32 dimensions. Do not use signals with more than 32 dimensions.

Determine the Output Dimensions of Source Blocks

A *source* block is a block that has no inputs. Examples of source blocks include the Constant block and the Sine Wave block. See “Sources” for a complete listing of Simulink source blocks. The output dimensions of a source block are the same as those of its output value parameters if the block **Interpret vector parameters as 1-D** parameter is off (that is, not selected in the block parameter dialog box). If the **Interpret vector parameters as 1-D** parameter is on, the output dimensions equal the output value parameter dimensions unless the parameter dimensions are N-by-1 or 1-by-N. In the latter case, the block outputs a vector signal of width N.

As an example of how the output value parameters and **Interpret vector parameters as 1-D** parameter of a source block determine the dimensionality of its output, consider the Constant block. This block outputs a constant signal equal to its **Constant value** parameter. The following table illustrates how the dimensionality of the **Constant value** parameter and the setting of the **Interpret vector parameters as 1-D** parameter determine the dimensionality of the block output.

Constant Value	Interpret vector parameters as 1-D	Output
scalar	off	one-element array
scalar	on	one-element array
1-by-N matrix	off	1-by-N matrix
1-by-N matrix	on	N-element vector
N-by-1 matrix	off	N-by-1 matrix
N-by-1 matrix	on	N-element vector
M-by-N matrix	off	M-by-N matrix
M-by-N matrix	on	M-by-N matrix

Simulink source blocks allow you either to specify the dimensions of the signals that they output or specify values from which Simulink infers the dimensions. You can therefore use the source blocks to introduce signals of various dimensions into your model.

Determine the Output Dimensions of Nonsource Blocks

If a block has inputs, the dimensions of its outputs are, after scalar expansion, the same as those of its inputs. (All inputs must have the same dimensions, as discussed in “Signal and Parameter Dimension Rules” on page 75-21).

Signal and Parameter Dimension Rules

When creating a Simulink model, you must observe the following rules regarding signal and parameter dimensions.

Input Signal Dimension Rule

All nonscalar inputs to a block must have the same dimensions.

A block can have a mix of scalar and nonscalar inputs as long as all the nonscalar inputs have the same dimensions. Simulink expands the scalar inputs to have the same dimensions as the nonscalar inputs (see “Scalar Expansion of Inputs and Parameters” on page 75-22).

Block Parameter Dimension Rule

In general, block parameters must have the same dimensions as the dimensions of the inputs to the block. Simulink performs some processing that provides flexibility relating to that general rule.

- A block can have scalar parameters corresponding to nonscalar inputs. In this case, Simulink expands a scalar parameter to have the same dimensions as the corresponding input (see “Scalar Expansion of Inputs and Parameters” on page 75-22).
- If an input is a vector, the corresponding parameter can be either an N-by-1 or a 1-by-N matrix. In this case, Simulink applies the N matrix elements to the corresponding elements of the input vector. This exception allows use of MATLAB row or column vectors, which are actually 1-by-N or N-by-1 matrices, respectively, to specify parameters that apply to vector inputs.

Vector or Matrix Input Conversion Rules

Simulink converts vectors to row or column matrices and row or column matrices to vectors under the following circumstances:

- If a vector signal is connected to an input that requires a matrix, Simulink converts the vector to a one-row or one-column matrix.
- If a one-column or one-row matrix is connected to an input that requires a vector, Simulink converts the matrix to a vector.
- If the inputs to a block consist of a mixture of vectors and matrices and the matrix inputs all have one column or one row, Simulink converts the vectors to matrices having one column or one row, respectively.

Note You can configure Simulink to display a warning or error message if a vector or matrix conversion occurs during a simulation. See “Vector/matrix block input conversion” for more information.

Scalar Expansion of Inputs and Parameters

Scalar expansion is the conversion of a scalar value into a nonscalar array. Many Simulink blocks support scalar expansion of inputs and parameters. Block-specific descriptions indicate whether Simulink applies scalar expansion to block inputs and parameters.

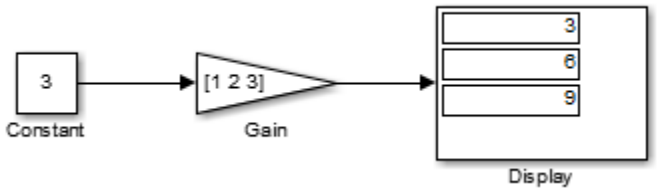
Scalar expansion of inputs refers to the expansion of scalar inputs to match the dimensions of other nonscalar inputs or nonscalar parameters. When the input to a block is a mix of scalar and nonscalar signals, Simulink expands the scalar inputs into nonscalar signals having the same dimensions as the other nonscalar inputs. For example, a scalar of 4 is expanded to the vector [4 4 4] if the associated nonscalar has a dimension of 3.

Scalar expansion of parameters refers to the expansion of scalar block parameters to match the dimensions of nonscalar inputs.

Input(s)	Associated Block Parameter	Scalar Expansion
Scalar	Nonscalar	Input expanded to match parameter dimensions. See “Scalar Input and Nonscalar Parameter” on page 75-22.
Nonscalar	Scalar	Scalar parameter expanded to match number of elements of input. See “Nonscalar Input and Scalar Parameter” on page 75-23.
Combination of scalar and nonscalar	No corresponding parameter	Scalar inputs expanded to match dimensions of largest nonscalar input. See “Scalar and Nonscalar Inputs and No Associated Parameter” on page 75-23.

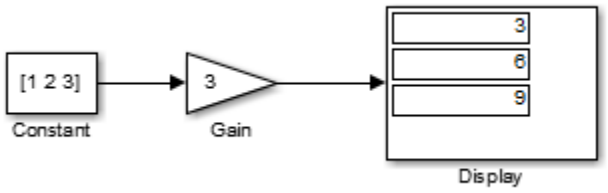
Scalar Input and Nonscalar Parameter

In this example, the Constant block input to the Gain block is scalar. The Gain block **Gain** parameter is a nonscalar. Simulink expands the scalar input to match the dimensions of a nonscalar **Gain** parameter, as reflected in the simulation results in the Display block.



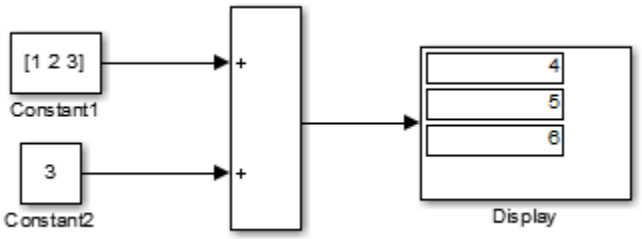
Nonscalar Input and Scalar Parameter

In this example, the Constant block input to the Gain block is nonscalar. The Gain block **Gain** parameter is a scalar. Simulink expands the scalar parameter to match the dimensions of a nonscalar input from the Constant block, as reflected in the simulation results in the Display block.



Scalar and Nonscalar Inputs and No Associated Parameter

In this example, the Constant1 block input to the Sum block is nonscalar, and the Constant2 block input is scalar. The Sum block has no associated parameter. Simulink expands the scalar input from Constant2 to match to the dimensions of the nonscalar Constant1 block input. The input is expanded to the vector `[3 3 3]`.



Get Compiled Port Dimensions

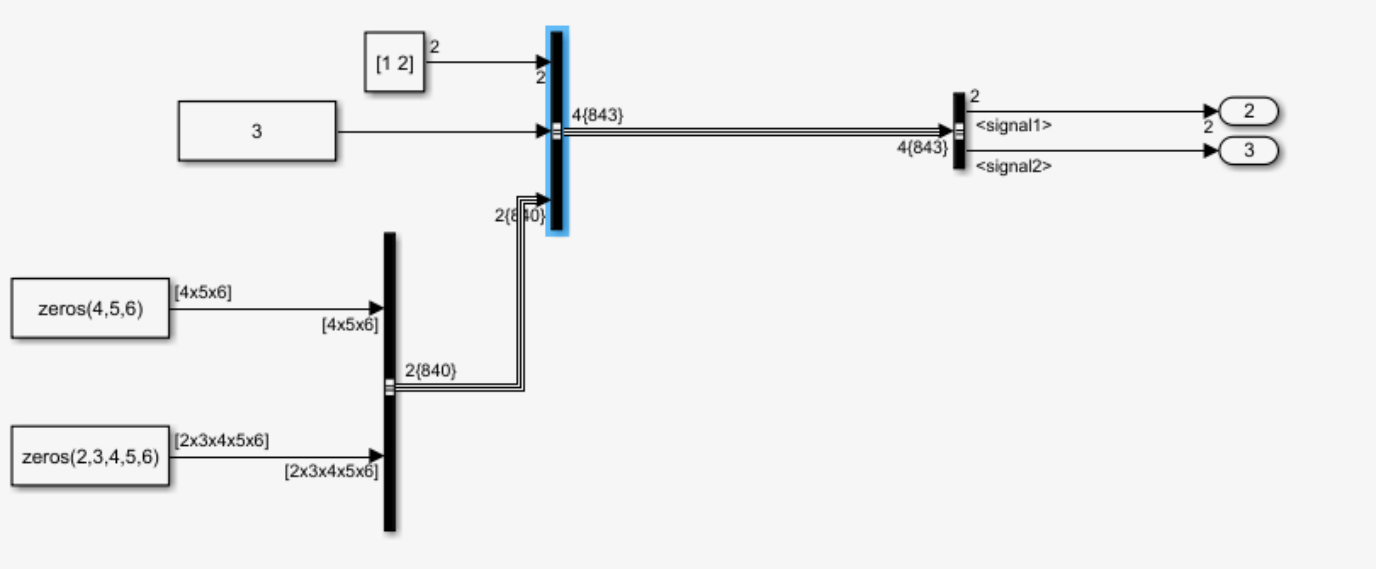
To get the dimensions of port signals, pause the simulation by using the **Step Forward** button. Select a block and use the `PortHandles` parameter. Then use the `GetCompiledPortDimensions` parameter. For example, if you step forward in this model and select the Constant block:


```
ph = get_param(gcb,'PortHandles');
dim = get_param(ph.Outport,'CompiledPortDimensions')
```

```
dim =

     2     2     3
```

For nonbus ports, the result is an array in which the first element is the number of dimensions (in this case 2), and the next two elements (corresponding to the number of dimensions) are the values of the dimensions.

For bus signals, the results include some extra elements. Suppose that you step forward in a model with buses and you select the Bus Creator block that contains a nested bus:




```
ph = get_param(gcb,'PortHandles');
dim = get_param(ph.Outport,'CompiledPortDimensions')

dim = -2 4 1 2 1 1 3 4 5 6 5 2 3 4 5 6
```

The first element is `-2`, which indicates that the signal is a bus. The second element is the number of leaf nodes. The subsequent elements follow the same pattern as for nonbus signals. In this example, the third element represents the number of dimensions for the first signal in the bus (in this case 1), and the next number is the value in that dimension (2).

See Also

Related Examples

- “Display Signal Attributes” on page 75-45
- “Determine Signal Dimensions” on page 75-19
- “Signal Basics” on page 75-2
- “Investigate Signal Values” on page 75-9

Highlight Signal Sources and Destinations

In this section...

You can highlight a signal and its source or destination blocks, then remove the highlighting once it has served its purpose. Signal highlighting crosses subsystem and model reference boundaries, allowing you to trace a signal across multiple subsystem levels. If a signal is composite, all source or destination blocks are highlighted. See “Types of Composite Signals” on page 76-2.

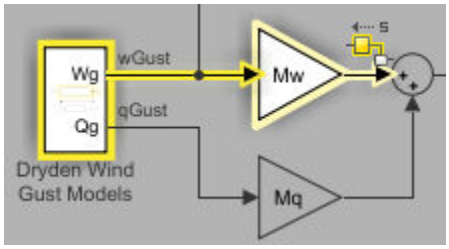
To continue the trace towards the source or destination of the signal, use the left and right arrow keys of your keyboard, respectively.

Highlight Signal Source

To begin a trace to the source blocks of a signal, select the **Highlight Signal to Source** option from the context menu for the signal. The

badge identifies the start of the trace. This option highlights:

- All branches of the signal anywhere in the model
- All virtual blocks through which the signal passes
- The nonvirtual blocks that write the value of the signal



To continue tracing towards the source of the signal, press the **Left** arrow key.

Highlight Signal Destination

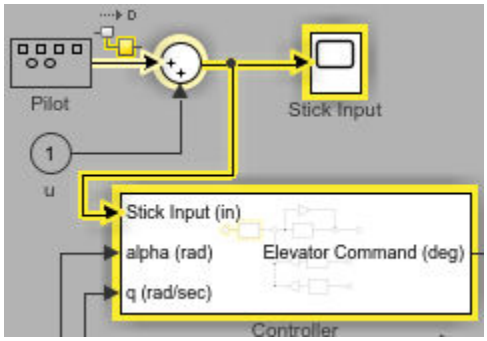
To begin a trace to the destination blocks of a signal, select the **Highlight Signal to Destination** option from the context menu for the signal. Press the **Right** arrow key to move the trace towards the

final destination. The

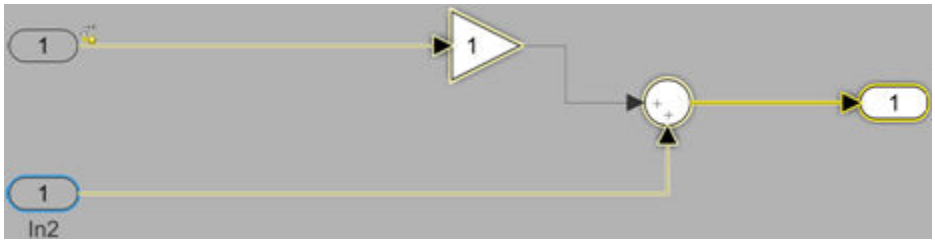
badge identifies the start of the trace. This option highlights:

- All branches of the signal anywhere in the model
- All virtual blocks through which the signal passes
- The nonvirtual blocks that read the value of the signal
- The signal and destination block for all blocks that are duplicates of the Inport block for the line that you select

In this example, the selected trace shows a path of the signal from the Stick Input to the Elevator Command in the Controller subsystem of the `f14` model.

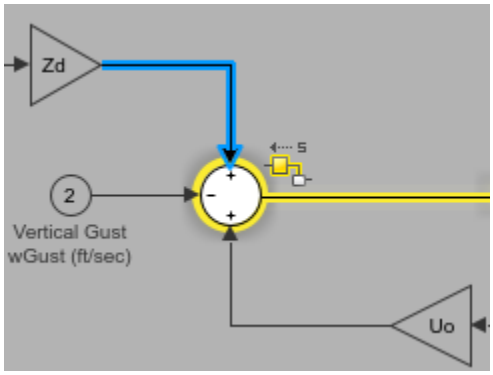


In the next example, selecting the signal from In2 and choosing the **Highlight Signal to Destination** option highlights the signal and destination block for In2 and In1, because In1 and In2 are duplicate Inport blocks.



Choose the Path of a Trace

In some situations, a signal trace can take multiple viable paths in the next segment of the trace. This can be seen when a signal is combined or split through a Mux or Demux block, or if it is branched out as an input to another block. In such cases, the options are highlighted in blue and can be cycled using the **Up** and **Down** arrow keys on the keyboard. Once you have highlighted the block to which you would like to move, proceed as normal.

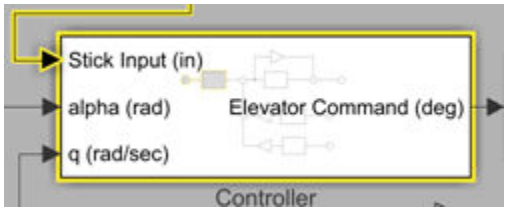


In the figure above, the trace to source has reached the Sum block where it can take one of three possible paths. The first option to Gain *Zd* is highlighted in blue.

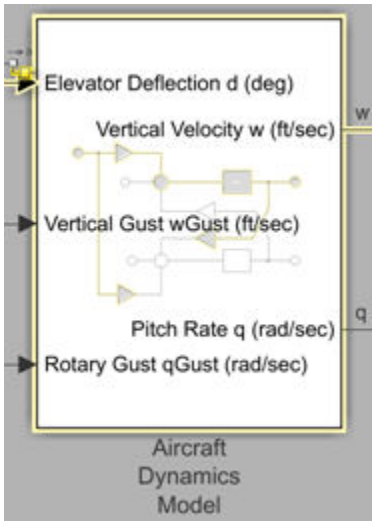
Trace a Signal To and From Subsystems

When the trace reaches a subsystem, it does not automatically trace in. Instead, the trace highlights the target subsystem and you are able to see a preview of the trace within the subsystem.

In this example, the selected signal trace to destination has reached the Controller subsystem of the `f14` model. You can see that the contents of the subsystem are visible and the next segment of the trace is highlighted within the preview.



When the trace comes out of a subsystem, the subsystem preview also displays the path of the trace inside it. The next example shows the path of a trace through the Aircraft Dynamics Model subsystem from the input `wGust` to vertical velocity `w` output.

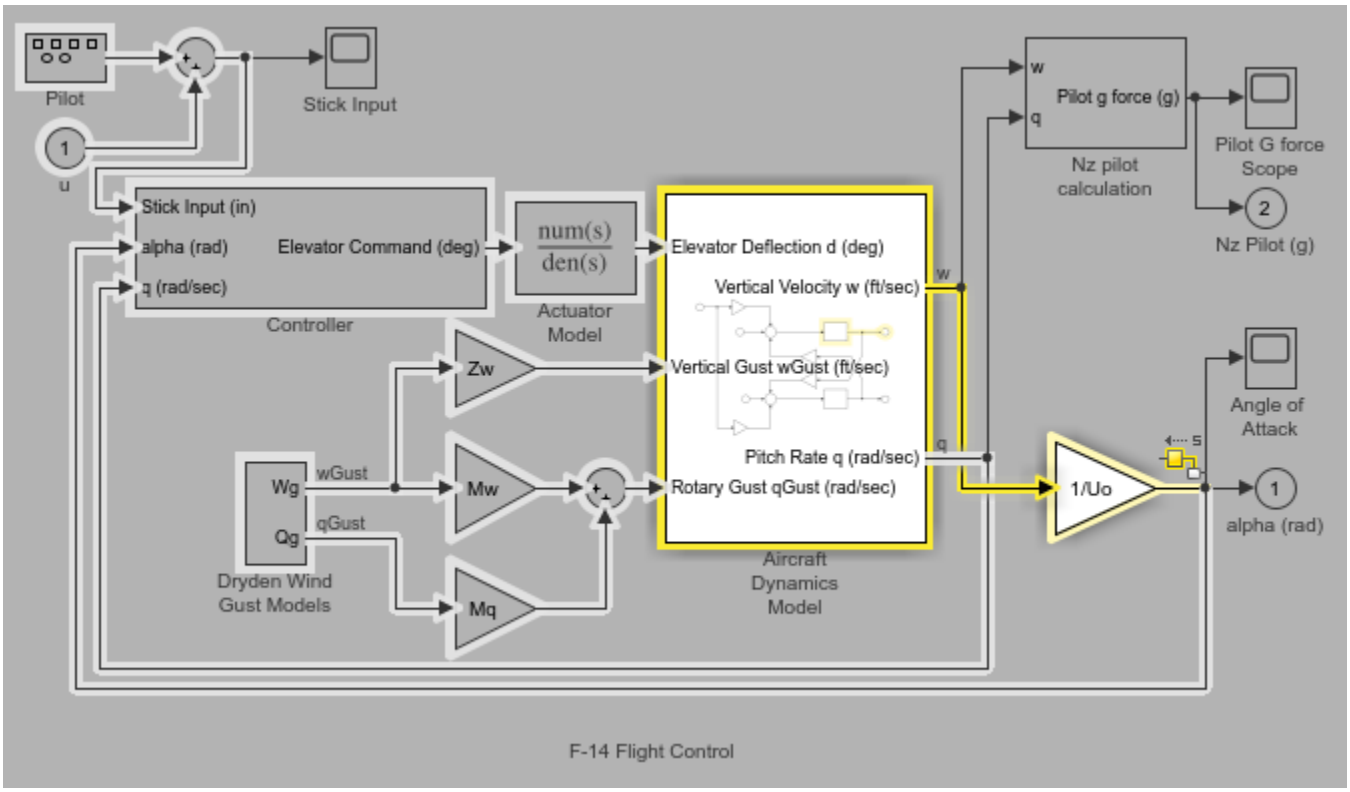


Note You cannot jump over a subsystem or a referenced model in your trace. You need to trace the signal path through them.

Show All Possible Paths of a Trace

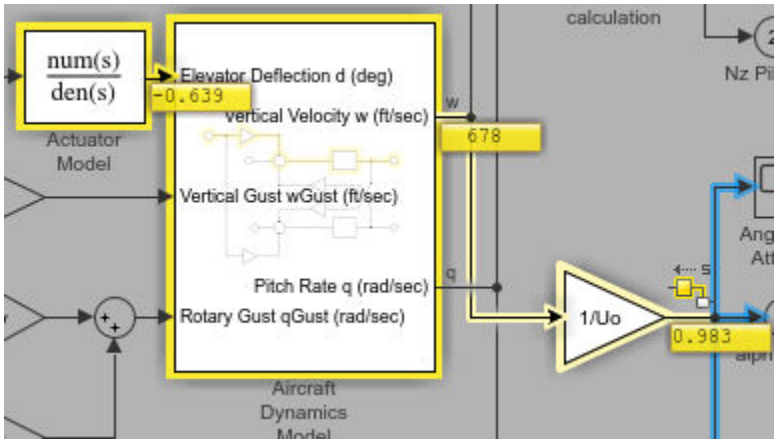
To highlight all possible paths a trace can take, press **Ctrl** key along with the **Left** (towards source) or the **Right** (towards destination) key. You can continue to trace as before.

To clear the highlighting, simultaneously press the **Ctrl** key and the arrow key in the direction opposite to the trace.

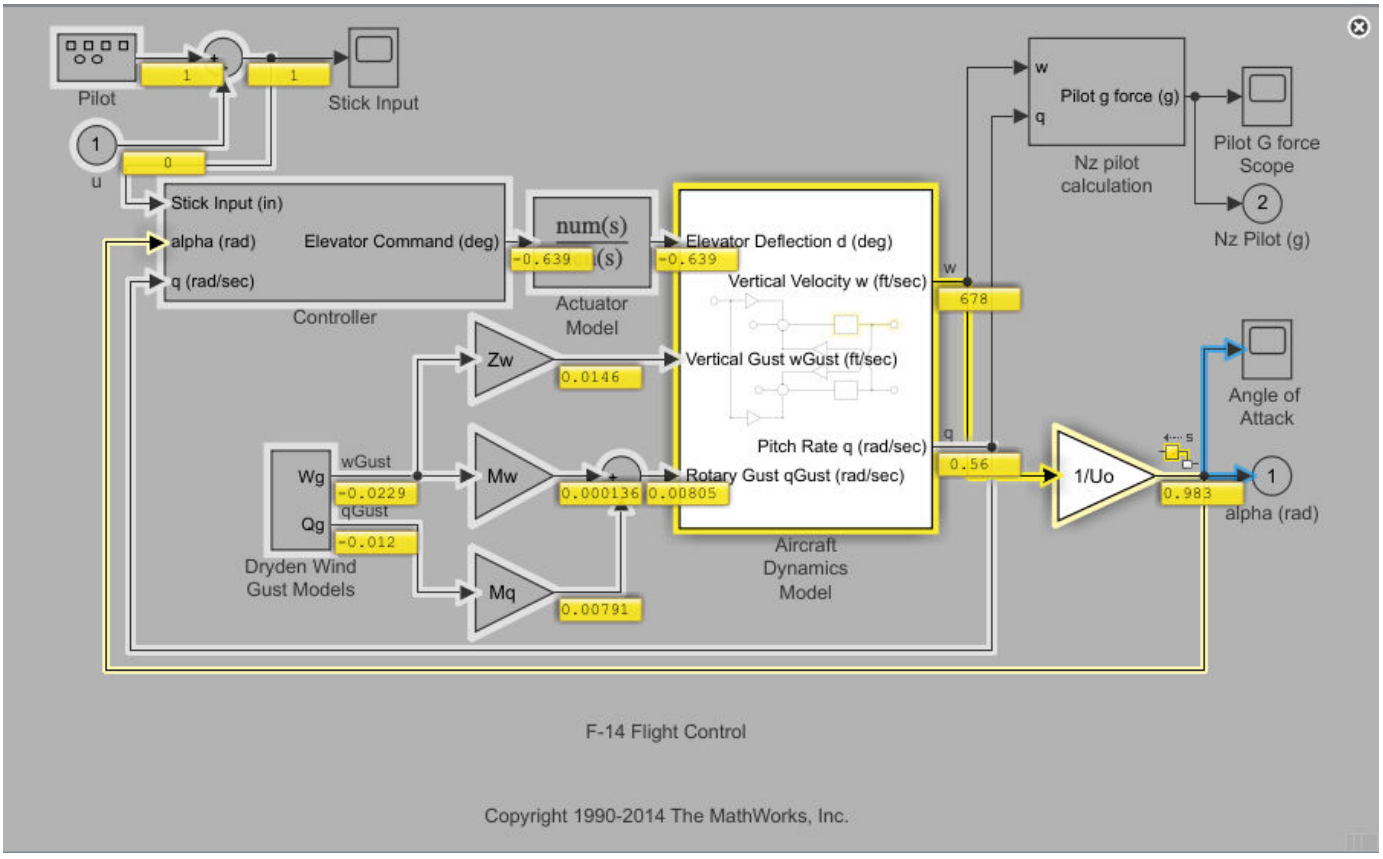


Display Port Values Along a Trace

After you trace the path of a signal, press the **L** key to display the values of a signal at the output port of each block.



If you highlight all the possible paths of the trace as described in “Show All Possible Paths of a Trace” on page 75-28, press the **Ctrl + L** to view the signal values at all the output ports.



Remove Highlighting

To remove all highlighting, right-click the model, and then select **Remove Highlighting**, or, on the **Signal** tab, click **Remove Trace**. You can also press **Ctrl+Shift+H** or click the icon at the top right corner of the editor.

To back up a trace, press the arrow key opposite to the original intent of the trace. For example, if you would like to back up a trace to the destination of the signal, press the left arrow key. To back up a trace to the source of a signal, press the **right arrow** key.

Resolve Incomplete Highlighting to Library Blocks

If the path from a source block or to a destination block includes an unresolved reference to a library block, the highlighting options highlight the path from or to the library block, respectively. To display the complete path, press **Ctrl+D** to update the diagram. The diagram update resolves all library references and displays the complete path to a destination block or from a source block.

Limitations

The signal tracing tool has the following limitations in its usage:

- The signal trace does not preserve bus information when tracing into or out of a referenced model.
- Tracing past a Goto block changes the active level to that containing the matching From block. However, this does not guarantee that the matching From block is the active block if there are other valid blocks at the same level.
- Signal tracing is unsupported by certain blocks. Unsupported blocks are blocks where:
  - You cannot trace into them to highlight their contents. Examples of such blocks include but are not limited to Stateflow charts, Simscape subsystems, and function blocks,.
  - You cannot cross them and continue tracing (e.g. Simscape blocks).
- The styling of the signal does not update with modifications to the model when signal tracing is enabled.
- The operation to “Show All Possible Paths of a Trace” on page 75-28 does not traverse through referenced models in the same way as it does subsystems. Referenced models are treated as non-virtual blocks and the impact region will not highlight their contents.

See Also

Related Examples

- “Display Signal Attributes” on page 75-45
- “Signal Label Propagation” on page 75-12
- “Signal Basics” on page 75-2

Specify Signal Ranges

In this section...

Simulink blocks allow you to specify a range of valid values for their output signals. Specifying signal ranges help you to optimize data types and improve generated code.

If you have Embedded Coder, Simulink Coder can optimize the code that you generate from the model by taking into account the minimum and maximum values that you specify for signals and parameters. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Embedded Coder).

Blocks That Allow Signal Range Specification

The following blocks allow you to specify ranges for their output signals:

- Abs
- Constant
- Data Store Memory
- Data Type Conversion
- Difference
- Discrete Derivative
- Discrete-Time Integrator
- Gain
- Inport
- Interpolation Using Prelookup
- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Math Function
- MinMax
- Multiport Switch
- Outport
- Product, Divide, Product of Elements
- Relay
- Repeating Sequence Interpolated
- Repeating Sequence Stair

- Saturation
- Saturation Dynamic
- Signal Specification
- Sum, Add, Subtract, Sum of Elements
- Switch

Work with Signal Ranges in Blocks

To specify signal ranges for most blocks, use the **Output minimum** and **Output maximum** parameters of a block to specify a range of valid values for the block output signal. Exceptions include the Data Store Memory, Inport, Outport, and Signal Specification blocks, for which you use their **Minimum** and **Maximum** parameters to specify a signal range. See “Blocks That Allow Signal Range Specification” on page 75-31 for a list of applicable blocks.

To access these parameters, use the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**), the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**), or the block dialog box.

Specify a minimum or maximum as an expression that evaluates to a scalar, real number with `double` data type. For example, you can use:

- A literal number such as `98.884`. Implicitly, the data type is `double`.
- A numeric workspace variable (see “Share and Reuse Block Parameter Values by Creating Variables” on page 37-9) whose data type is `double`. Use this technique to share a minimum or maximum value between multiple data items.

  However, you cannot use variables to set the `Min` or `Max` properties of a `Simulink.Signal` object.

The scalar value that you specify applies to each element of a composite signal (for example, when the signal is nonscalar or a bus). For information about scalar expansion, see “Scalar Expansion of Inputs and Parameters” on page 75-22.

To leave the minimum or maximum of a signal unspecified, use an empty matrix `[]`, which is the default value.

Specify Ranges for Modeling Constructs

If you use modeling constructs such as bus signals, data stores, and Stateflow charts, you can use different techniques to specify design range information. Use the information in the table.

Description of Target Signal	Technique and More Information
Numerically complex signal	When you specify an **Output minimum** or **Output maximum** for a signal that is numerically complex, the specified minimum and maximum values apply separately to the real part and to the imaginary part of the complex number. If the value of either part of the number is less than the minimum, or greater than the maximum, the complex number is outside the specified range. No range checking occurs against any combination of the real and imaginary parts, such as `(sqrt(a^2+b^2))`.

Description of Target Signal	Technique and More Information
Signal elements in a bus	If you assemble the bus by using a Bus Creator block, you can specify range information on the upstream blocks that feed the Bus Creator. Regardless of the technique you use to assemble the bus, you can create a `Simulink.Bus` object and use it as the data type of the bus signal. In this case, consider specifying range information by using the `Min` and `Max` properties of the `Simulink.BusElement` objects that reside in the bus object. For more information, see “Specify Bus Properties with Simulink.Bus Objects” on page 76-44.
Signal in a MATLAB Function block	Use the Ports and Data Manager to specify the **Minimum** and **Maximum** properties of the data. See “Setting General Properties” on page 44-35.
Signal in a Stateflow chart	Set the **Minimum** and **Maximum** properties of the corresponding Stateflow data. See “Limit Range” (Stateflow).
Signal that you associate with a signal object (such as `Simulink.Signal`)	Set the `Min` and `Max` properties of the signal object. See `Simulink.Signal`.
Data store (Data Store Memory block or `Simulink.Signal` object)	For a Data Store Memory block, set the **Minimum** and **Maximum** block parameters. For a signal object, set the `Min` and `Max` properties.

Troubleshoot Signal Range Errors

Simulink provides a diagnostic named **Simulation range checking**, which you can enable to detect when signals exceed their specified ranges during simulation. When enabled, Simulink compares the signal values that a block outputs with both the specified range (see “Work with Signal Ranges in Blocks” on page 75-32) and the block data type. That is, Simulink performs the following check:

```
DataTypeMin ≤ MinValue ≤ VALUE ≤ MaxValue ≤ DataTypeMax
```

where

- `DataTypeMin` is the minimum value representable by the block data type.
- `MinValue` is the minimum value the block should output, specified by, e.g., **Output minimum**.
- `VALUE` is the signal value that the block outputs.
- `MaxValue` is the maximum value the block should output, specified by, e.g., **Output maximum**.
- `DataTypeMax` is the maximum value representable by the block data type.

Note It is possible to overspecify how a block handles signals that exceed particular ranges. For example, you can specify values (other than the default values) for both signal range parameters and enable the **Saturate on integer overflow** parameter. In this case, Simulink displays a warning message that advises you to disable the **Saturate on integer overflow** parameter.

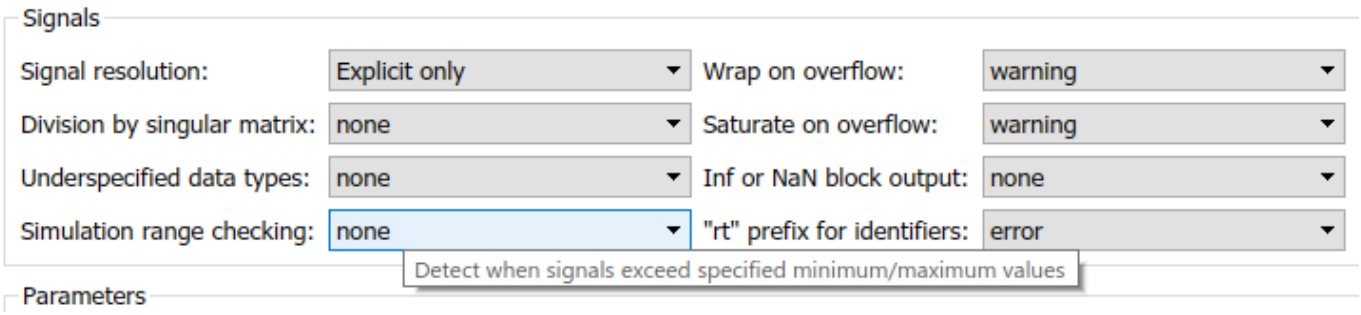
Enable Simulation Range Checking

To enable the **Simulation range checking** diagnostic:

1. In your model window, on the **Modeling** tab, click **Model Settings**.

   Simulink displays the Configuration Parameters dialog box.

2. In the **Select** tree on the left side of the Configuration Parameters dialog box, click the **Diagnostics > Data Validity** category. On the right side under **Signals**, set the **Simulation range checking** diagnostic to `error` or `warning`.



3. Click **OK** to apply your changes and close the Configuration Parameters dialog box.

See “Simulation range checking” for more information.

Simulate Models with Simulation Range Checking

To check for signal range errors or warnings:

1. Enable the **Simulation range checking** diagnostic for your model (see “Enable Simulation Range Checking” on page 75-33).
2. In your model window, click **Run** to simulate the model.

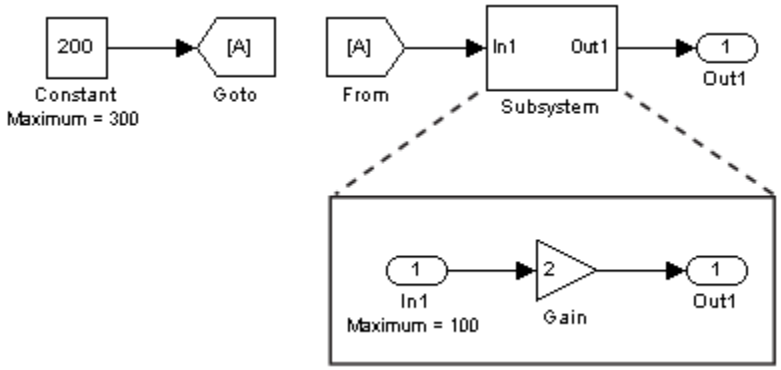
   Simulink simulates your model and performs signal range checking. If a signal exceeds its specified range when the **Simulation range checking** diagnostic specifies `error`, Simulink stops the simulation and generates an error (for example, in the Diagnostic Viewer).

   Otherwise, if a signal exceeds its specified range when the **Simulation range checking** diagnostic specifies `warning`, Simulink generates a warning message in the MATLAB Command Window. Each message identifies the block whose output signal exceeds its specified range, and the time step at which this violation occurs.

Signal Range Propagation for Virtual Blocks

Some virtual blocks (see “Nonvirtual and Virtual Blocks” on page 36-2) allow you to specify ranges for their output signals, for example, the Inport and Outport blocks. When the **Simulation range checking** diagnostic is enabled for a model that contains such blocks, the signal range of the virtual block propagates backward to the first instance of a nonvirtual block whose output signal it receives. If the nonvirtual block specifies different values for its own range, Simulink performs signal range checking with the tightest range possible. That is, Simulink checks the signal using the larger minimum value and the smaller maximum value.

For example, consider the following model:



In this model, the Constant block specifies its **Output maximum** parameter as 300, and that of the Inport block is set to 100. Suppose you enable the **Simulation range checking** diagnostic and simulate the model. The Inport block back propagates its maximum value to the nonvirtual block that precedes it, i.e., the Constant block. Simulink then uses the smaller of the two maximum values to check the signal that the Constant block outputs. Because the Constant block outputs a signal whose value (200) exceeds the tightest range, Simulink generates an error.

Unexpected Errors or Warnings for Data with Greater Precision or Range than double

When a data item (signal or parameter) uses a data type other than `double`, before comparison, Simulink casts the data item and each design limit (minimum or maximum value that you specify) to the nondouble data type. This technique helps prevent the generation of unnecessary, misleading errors and warnings.

However, Simulink stores design limits as `double` before comparison. If the data type of the data item has higher precision than `double` (for example, a fixed-point data type with a 128-bit word length and a 126-bit fraction length) or greater range than `double`, and `double` cannot exactly represent the value of a design limit, Simulink can generate unexpected warnings and errors.

If the nondouble type has higher precision, consider rounding the design limit to the next number furthest from zero that `double` can represent. For example, suppose that a signal generates an error after you set the maximum value to `98.8847692348509014`. At the command prompt, calculate the next number furthest from zero that `double` can represent.

```
format long
98.8847692348509014 + eps(98.8847692348509014)

ans =

  98.884769234850921
```

Use the resulting number, `98.884769234850921`, to replace the maximum value.

See Also

Related Examples

- “Display Signal Attributes” on page 75-45

- “Control Signal Data Types” on page 67-6
- “Signal Basics” on page 75-2
- “Investigate Signal Values” on page 75-9
- “Fixed Point”

Initialize Signals and Discrete States

In this section...

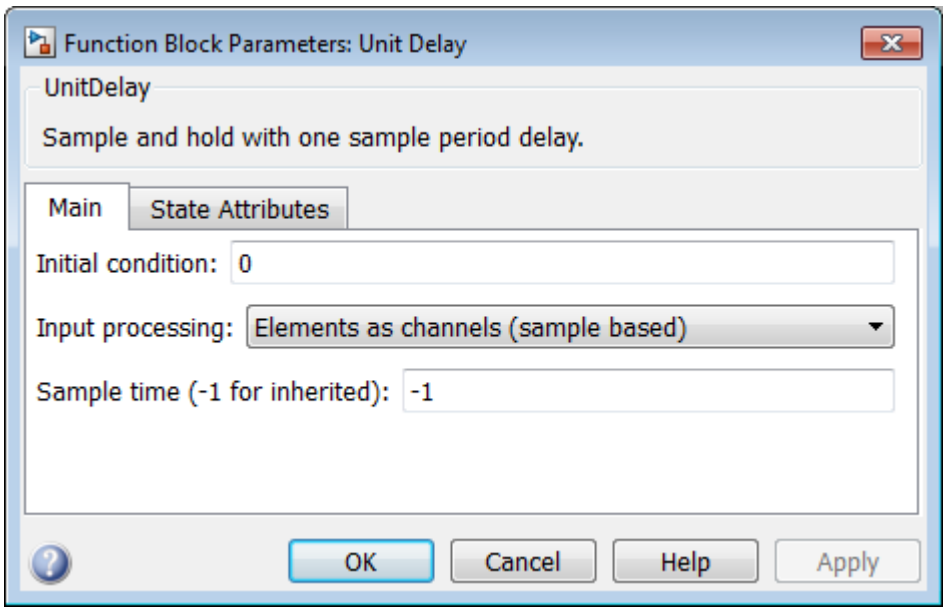
Simulink allows you to specify the initial values of signals and discrete states, i.e., the values of the signals and discrete states at the **Start time** of the simulation. You can use signal objects to specify the initial values of any signal or discrete state in a model. In addition, for some blocks, e.g., Outport, Data Store Memory, or Memory, you can use either a signal object or a block parameter or both to specify the initial value of a block state or output. In such cases, Simulink checks to ensure that the values specified by the signal object and the parameter are consistent. For information about initializing bus signals, see “Specify Initial Conditions for Bus Signals” on page 76-57

When you specify a signal object for signal or discrete state initialization, or a variable as the value of a block parameter, Simulink resolves the name that you specify to an appropriate object or variable, as described in “Symbol Resolution” on page 67-127.

A given signal can be associated with at most one signal object under any circumstances. The signal can refer to the object more than once, but every reference must resolve to exactly the same object. A different signal object that has exactly the same properties will not meet the requirement for uniqueness. A compile-time error occurs if a model associates more than one signal object with any signal. For more information, see `Simulink.Signal` and the Merge block.

Using Block Parameters to Initialize Signals and Discrete States

For blocks that have an initial value or initial condition parameter, you can use that parameter to initialize a signal. For example, the following Block Parameters dialog box initializes the signal for a Unit Delay block with an initial condition of 0.



To access these block parameters, choose one of these techniques:

- Use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). Use this technique to configure multiple signals and states with a searchable, sortable table. To initialize a block state or data store, you can use the appropriate tab (**States** or **Data Stores**). To initialize a signal, state, or data store, you can use the **Parameters** tab and find the row that corresponds to the relevant block parameter.

  For more information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.
- Use the Property Inspector (on the **Modeling** tab, under **Design**, click **Property Inspector**). Use this technique to configure one signal or state at a time. Select the block that maintains the target state or generates the target signal and find the relevant block parameter.
- Use the block parameter dialog box. Use this technique to configure one signal or state at a time or to compare the configurations of a few signals or states side by side.

For more information about techniques to access block parameters (including the parameters that control signal and state initialization), see “Set Properties and Parameters”.

Use Signal Objects to Initialize Signals and Discrete States

You can use signal objects that have a storage class other than `'auto'` or, when you set the default storage class of the corresponding data category to `Default` (the default setting) in the Code Mapping Editor, `'Model default'` to initialize:

- Discrete states with an initial condition parameter
- Signals in a model except bus signals and blocks that output constant value

To specify an initial value, use the Model Explorer or MATLAB commands to do the following:

1. Create the signal object.

   On the Model Explorer toolbar, select **Add** > **Simulink Signal**. The signal object appears in the base workspace with a default name. Rename the object as `S1`. Alternatively, use this command at the command prompt:

   ```
   S1 = Simulink.Signal;
   ```

   The name of the signal object must be the same as the name of the signal that the object is initializing. Although not required, consider setting the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. This setting makes signal objects in the MATLAB workspace consistent with signals that appear in your model.

   Consider using the Data Object Wizard to create signal objects. The Data Object Wizard searches a model for signals for which signal objects do not exist. You can then selectively create signal objects for multiple signals listed in the search results with a single operation. For more information about the Data Object Wizard, see “Create Data Objects for a Model Using Data Object Wizard” on page 67-64.
2. Set the signal object storage class to a value other than `Auto` or `Model default`. In the Model Explorer **Contents** pane, select the signal object. In the Dialog pane, set **Storage class** to `ExportedGlobal`. Alternatively, use this command at the command prompt:

   ```
   S1.CoderInfo.StorageClass = 'ExportedGlobal';
   ```

3 Set the initial value. You can specify a MATLAB expression, including the name of a workspace variable, that evaluates to a numeric scalar value or array.

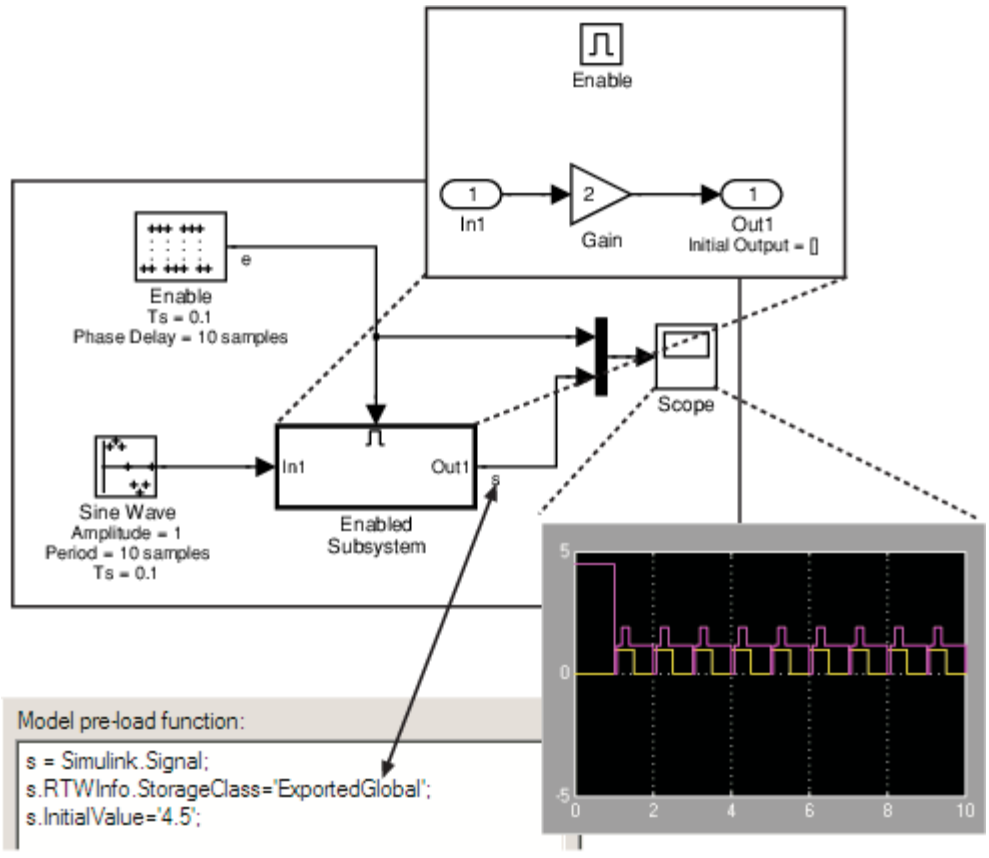
The Simulink engine converts the initial value so the type, complexity, and dimension are consistent with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model.

In the Model Explorer Dialog pane, set **Initial value** to `0.5`. Alternatively, use this command at the command prompt:

```
S1.InitialValue = '0.5'
```

If you can also use a block parameter to set the initial value of the signal or state, you should set the parameter either to empty (`[]`) or to the same value as the initial value of the signal object. If you set the parameter value to empty, Simulink uses the value specified by the signal object to initialize the signal or state. If you set the parameter to any other value, Simulink compares the parameter value to the signal object value and displays an error if they differ.

The following example shows a signal object specifying the initial output of an enabled subsystem.



Signal `s` is initialized to 4.5. To avoid a consistency error, the initial value of the enabled subsystem Outport block must be `[]` or 4.5.

If you need a signal object and its initial value setting to persist across Simulink sessions, see “Create Persistent Data Objects” on page 67-72.

Some initial value settings may depend on the initialization mode. For more information, see “Underspecified initialization detection”.

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 output of the following blocks are ignored. The initial values of the corresponding blocks are used instead.

- Output signals of conditionally executed subsystems
- Merge blocks

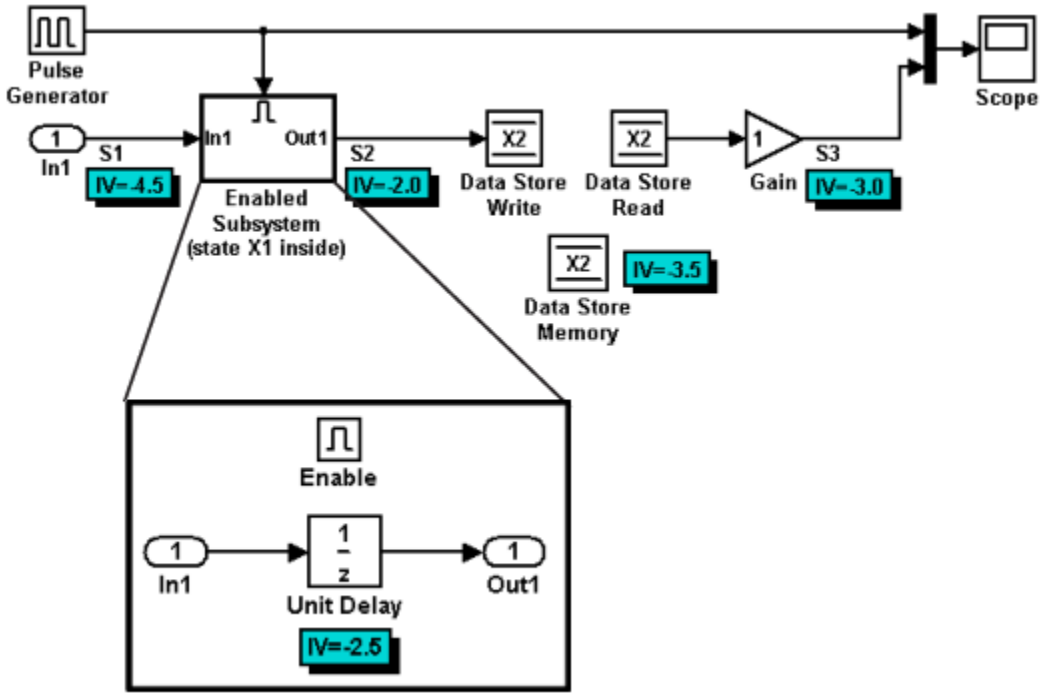
Using Signal Objects to Tune Initial Values

Simulink allows you to use signal objects as an alternative to parameter objects (see `Simulink.Parameter`) to tune the initial values of block outputs and states that can be specified via a tunable parameter. To use a signal object to tune an initial value, create a signal object with the same name as the signal or state and set the signal object initial value to an expression that includes a variable defined in the MATLAB workspace. You can then tune the initial value by changing the value of the corresponding workspace variable during the simulation.

For example, suppose you want to tune the initial value of a Memory block state named `M1`. To do this, you might create a signal object named `M1`, set its storage class to `'ExportedGlobal'`, set its initial value to `K` (`M1.InitialValue='K'`), where `K` is a workspace variable in the MATLAB workspace, and set the corresponding initial condition parameter of the Memory block to `[]` to avoid consistency errors. You could then change the initial value of the Memory block state any time during the simulation by changing the value of `K` at the MATLAB command line and updating the block diagram (e.g., by typing **Ctrl+D**).

Initialization Behavior Summary for Signal Objects

The following model and table show different types of signals and discrete states that you can initialize and the simulation behavior that results for each.



Signal or Discrete State	Description	Behavior
S1	Root input port	• Initialized to `S1.InitialValue`. • If you use the **Data Import/Export** pane of the Configuration Parameters dialog to specify values for the root inputs, the initial value is overwritten and may differ at each time step. Otherwise, the value remains constant.
X1	Unit Delay block — Block with a discrete state that has an initial condition	• Initialized to `X1.InitialValue`. • Simulink checks whether `X1.InitialValue` matches the initial condition specified for the block and displays an error if a mismatch occurs. • At first write, the output equals `X1.InitialValue` and the state equals `S1`. • At each time step after the first write, the output equals the state and the state is updated to equal `S1`. • If the block is inside an enabled subsystem, you can use the initial value as a reset value if the subsystem Enable block parameter **States when enabling** is set to `reset`.
X2	Data Store Memory block	• Data type work (DWork) vector initialized to `X2.InitialValue`. For information on work vectors, see “DWork Vector Basics”. • Simulink checks whether `X2.InitialValue` matches the initial condition specified for the block, and displays an error if a mismatch occurs. • Data Store Write blocks overwrite the value.

Signal or Discrete State	Description	Behavior
S2	Output of an enabled subsystem	• Initialized to `S2.InitialValue` or the value of the Outport block. If multiple initial values are specified for the same signal, all initial values must be the same. • The first write occurs when the subsystem is enabled. The block feeding the subsystem output sets the value. • The initial value is also used as a reset value if the subsystem Enable block parameter **States when enabling** or Outport block parameter **Output when disabled** is set to `reset`.
S3	Persistent signals	• Initialized to `S3.InitialValue`. • The output value is reset by the block at each time step. • Affects code generation only. For simulation, setting the initial value for `S3` is irrelevant because the values are overwritten at model simulation start time.

See Also

Related Examples

- “Using Initialize, Reset, and Terminate Functions” on page 10-168
- “Set Block Parameter Values” on page 37-2
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Signal Basics” on page 75-2
- “Investigate Signal Values” on page 75-9

Configure Signals as Test Points

In this section...

What Is a Test Point?

A *test point* is a signal that Simulink guarantees to be observable when using a Floating Scope block in a model. Simulink allows you to designate any signal in a model as a test point.

Designating a signal as a test point exempts the signal from model optimizations, such as signal storage reuse (see “Signal storage reuse” (Simulink Coder)) and block reduction (see “Implement logic signals as Boolean data (vs. double)”). These optimizations render signals inaccessible and hence unobservable during simulation.

Signals designated as test points will not have algebraic loops minimized, even if **Minimize algebraic loop occurrences** is selected (for more information about algebraic loops, see “Algebraic Loop Concepts” on page 3-27).

Test points are primarily intended for use when generating code from a model with Simulink Coder. For more information about test points in the context of code generation, see “Appearance of Test Points in the Generated Code” (Simulink Coder).

Marking a signal as a test point has no impact on signal logging that uses the `Dataset` logging format. For information about logging signals, see “Export Signal Data Using Signal Logging” on page 72-41.

Use one of the following ways to designate a signal as a test point:

- Open the **Signal Properties** dialog for the signal and check **Test Point** in the **Logging and accessibility** section.
- Use the Model Data Editor for batch configuration and for signals that are difficult to locate in a large model or hierarchy of subsystems. On the **Signals** tab, set the **Change view** drop-down list to `Instrumentation` and use the **Test Point** column. For information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.
- To configure Stateflow data in a chart as test points, see “Monitor Test Points in Stateflow Charts” (Stateflow).

To configure a signal as a test point programmatically:

1. Get handles to the ports of the block.

   ```
   portHandles = get_param('myModel/myBlock','portHandles');
   ```

   `portHandles` is a structure. Each field stores a handle to a block port.
2. Extract a handle to the output port that creates the target signal line.

   ```
   outportHandle = portHandles.Outport;
   ```
3. Set the port parameter `TestPoint` to `'on'`.

   ```
   set_param(outportHandle,'TestPoint','on')
   ```

Displaying Test Point Indicators

By default, Simulink displays an indicator on each signal whose **Signal Properties > Test point** option is enabled. For example, in the following model signals `s2` and `s3` are test points:

Note Simulink does not display an indicator on a signal that is specified as a test point by a `Simulink.Signal` object, because such a specification is external to the graphical model.

A signal that is a test point can also be logged. See “Export Signal Data Using Signal Logging” on page 72-41 for information about signal logging. The appearance of the indicator changes to indicate signals for which logging is also enabled.

To turn display of test point indicators on or off, in the Simulink Editor, on the **Debug** tab, select **Information Overlays > Log & Testpoint** to enable or disable the option.

See Also

Related Examples

- “Investigate Signal Values” on page 75-9
- “Signal Basics” on page 75-2

More About

- “Configure Data Properties by Using the Model Data Editor” on page 67-131
- “Export Signal Data Using Signal Logging” on page 72-41

Display Signal Attributes

In this section...

A signal line in a model has attributes such as data type, dimensions, and numeric complexity. When you display these attributes on the block diagram, you can:

- Make the model easier to understand by others.
- Determine the value of the attribute that the signal ultimately uses for simulation (for example, when a signal uses an inherited data type).
- Plan your strategy to control these attributes along a data path (a series of connected blocks).

Additionally, to inspect and specify these attributes in a searchable, sortable table, you can use the Model Data Editor (see “Configure Data Properties by Using the Model Data Editor” on page 67-131).

Ports & Signals Menu

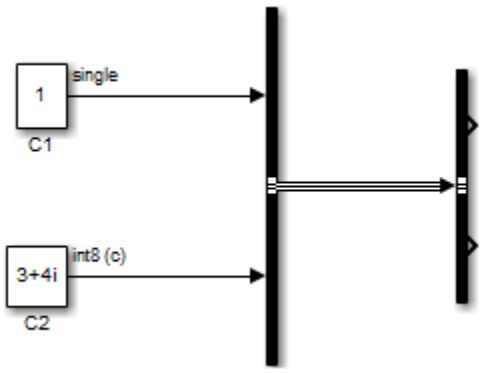
In the Simulink Editor, on the **Debug** tab, the **Information Overlays** menu offers the following options for displaying signal properties on the block diagram:

- Linearization Indicators
- Port Data Types (See “Port Data Types” on page 75-46)
- Design Ranges (See “Design Ranges” on page 75-47)
- Signal Dimensions (See “Signal Dimensions” on page 75-47)
- Storage Class
- Testpoint/Logging Indicators
- Signal Resolution Indicators (See “Signal to Object Resolution Indicator” on page 75-48)
- Viewer Indicators
- Wide Nonscalar Lines (See “Wide Nonscalar Lines” on page 75-49)

In addition, you can display sample time information. In the Simulink Editor, on the **Debug** tab, the **Information Overlays** menu provides the choices of **Colors** and **Text**. The **Colors** option allows the block diagram signal lines and blocks to be color-coded based on the sample time types and relative rates. The **Text** option provides black codes on the signal lines which indicate the type of sample time. If you select both **Colors** and **Text**, then both the colors and the annotations display. All of these options cause a Sample Time Legend to appear. The legend contains a description of the type of sample time and the sample time rate. If **Colors** is enabled, color codes also appear in the legend. The same is true if **Text** is enabled.

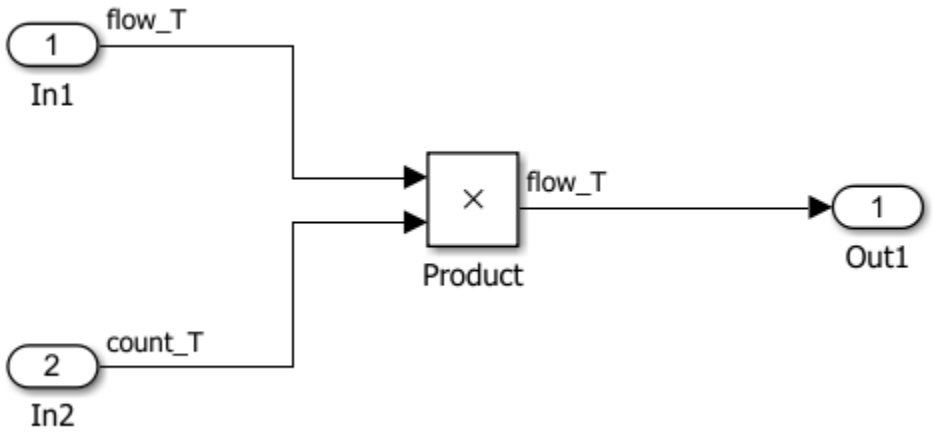
Port Data Types

Displays the data type that each signal uses for simulation and code generation. The data type appears next to the output port that emits the signal.



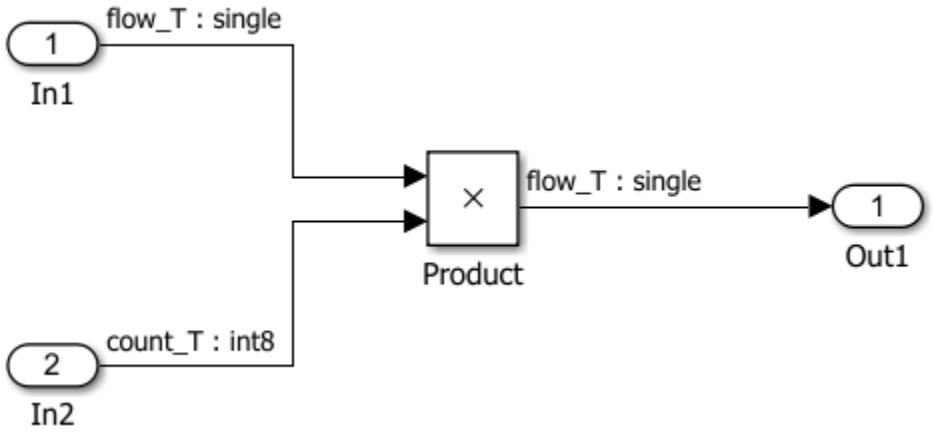
The notation `(c)` indicates that the signal is numerically complex (`i`).

If you use data type aliases (such as a `Simulink.AliasType` objects in the base workspace or a data dictionary) to set output data types in your model, by default, the diagram displays the aliases.



If you create a chain of aliases (for example, by using one `Simulink.AliasType` object as the base type of another `Simulink.AliasType` object), the diagram displays only the alias that you use to set the output data type of each signal. The diagram does not display the underlying aliases in the chain.

To display the lowest underlying base data type (such as `int8`, `single`, or `s16En14`) as well as the alias, in the **Debug** tab, open the **Information Overlays** drop-down. Under **Ports** click **Base Data Types** and **Alias Data Types**.



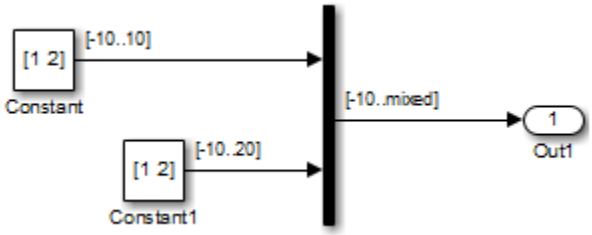
Alternatively, you can display the base type and not the alias by selecting **Base Data Types**.

When you use a fixed-point data type, the diagram displays the base type by using a standard notation that indicates the characteristics of the type (such as signedness and binary fraction length). To interpret this notation, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).

When you save a model with **Base Data Types** enabled, the next time you load the model, it displays the data type and complexity signal attributes.

Design Ranges

Displays the compiled design range of a signal next to the output port that emits the signal. The ranges are computed during an update diagram.

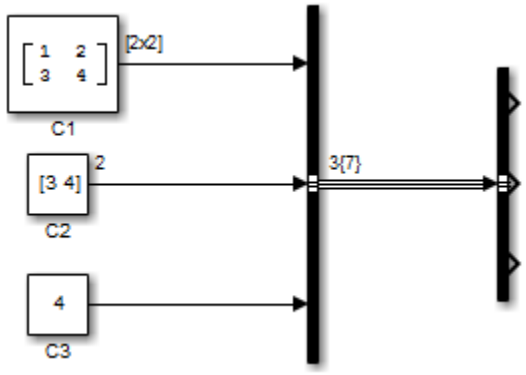


Ranges are displayed in the format `[min..max]`. In the above example, the design range at the output port of the Mux block is displayed as `[-10..mixed]`, because the two signals the Mux block combines have the same design minimum but different design maximums.

You can also use command-line parameters `CompiledPortDesignMin` and `CompiledPortDesignMax` to access the design minimum and maximum of port signals, respectively, at compile time. For more information, see “Common Block Properties”.

Signal Dimensions

Display the dimensions of nonscalar signals next to the line that carries the signal.



The format of the display depends on whether the line represents a single signal or a bus. If the line represents a single vector signal, Simulink displays the width of the signal. If the line represents a single matrix signal, Simulink displays its dimensions as `[`N_1`x`N_2`]` where N_i is the size of the `i`th dimension of the signal. If the line represents a bus carrying signals of the same data type, Simulink displays `N{M}` where `N` is the number of signals carried by the bus and `M` is the total number of signal elements carried by the bus. If the bus carries signals of different data types, Simulink displays only the total number of signal elements `{M}`.

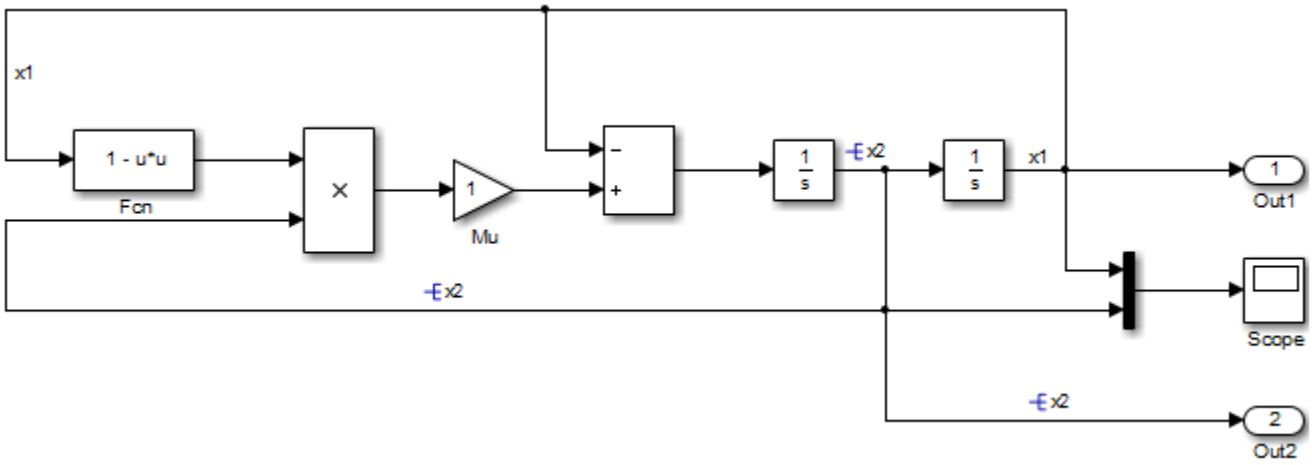
When you save a model with this option enabled, the next time you load the model, it displays signal dimensions.

Signal to Object Resolution Indicator

The Simulink Editor by default graphically indicates signals that must resolve to signal objects. For any labeled signal whose **Signal name must resolve to signal object** property is enabled, a signal resolution icon appears to the left of the signal name. The icon looks like this:

A signal resolution icon indicates only that the **Signal name must resolve to signal object** property for a signal is enabled. The icon does not indicate whether the signal is actually resolved, and does not appear on a signal that is implicitly resolved without its **Signal name must resolve to signal object** property being enabled.

Where multiple labels exist, each label displays a signal resolution icon. No icon appears on an unlabeled branch. In the next figure, signal `x2` must resolve to a signal object, so a signal resolution icon appears to the left of the signal name in each label:

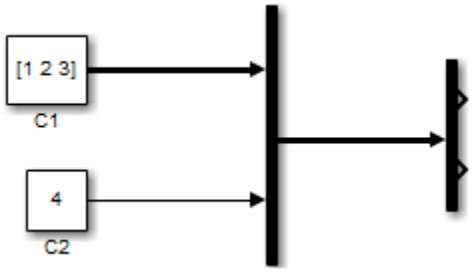


To suppress the display of signal resolution icons, in the model window, on the **Debug** tab, select **Information Overlays > Signal Resolves to Object** to disable. This option is enabled default. To restore signal resolution icons, enable **Signal Resolves to Object** . Individual signals cannot be set to show or hide signal resolution indicators independently of the setting for the whole model. For additional information, see:

- “Symbol Resolution” on page 67-127
- “Initialize Signals and Discrete States” on page 75-37
- `Simulink.Signal`

Wide Nonscalar Lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.



For more information on vector and matrix signals, see “Signal Types” on page 75-7.

See Also

Related Examples

- “Determine Signal Dimensions” on page 75-19
- “Highlight Signal Sources and Destinations” on page 75-25
- “Signal Basics” on page 75-2

Signal Groups

In this section...

About Signal Groups

To display, create, and edit interchangeable groups of signal sources and quickly switch the groups into and out of a model, use the Signal Builder block. You can define any piecewise linear signal shape (waveform).

Note The Signal Builder block is not recommended to work with signal groups. Instead, use the Signal Editor block to display, create, edit, and switch interchangeable scenarios. For more information, see “Load Data with Interchangeable Scenarios” on page 71-37.

Use signal groups when testing a model, especially when using them in conjunction with the Simulink Assertion block and the Model Coverage Tool from the Simulink Coverage.

Solver pane settings in Model Configuration Parameters can affect the Signal Builder block output. See “Simulation Phases in Dynamic Systems” on page 3-2 and “Compare Solvers” on page 3-6 for a description of how solvers affect simulation.

Note The Signal Builder block adds a port for each signal you create. The block **Position** parameter limits the number of ports the Signal Builder block can have, and therefore the number of signals you can create. For more information, see the **Position** parameter at “Common Block Properties”.

You can also use the `signalbuilder` function to populate a Signal Builder block.

Supported Waveforms

The Signal Builder block supports these waveforms.

Using the Signal Builder Block with Fast Restart

After you turn on fast restart:

- In between runs, you can change data, rename signals and signal groups, and add new groups. You cannot:
  - Import signals or signal groups
  - Change signal output settings
- You can click the **Run all** button once. To reenable the **Run all** button, toggle the fast restart button on the Simulink Editor tool bar. However, **Run all** does not use fast restart.

Editing Signal Groups

The Signal Builder window allows you to create, rename, move, then delete signal groups from the set of groups represented by a Signal Builder block.

Creating and Deleting Signal Groups

To create a signal group:

1. In Signal Builder, copy an existing signal group.
2. Modify it to suit your needs.

To copy an existing signal group:

1. In Signal Builder, select the group from the list.
2. Select **Group > Copy**.

   A new group is created.

To delete a group, select the group from the list, and select **Group > Delete**.

Renaming Signal Groups

To rename a signal group:

1. In Signal Builder, select the group from the list.
2. Select **Group > Rename**.
3. Edit the existing name in the dialog box or enter a new name. Click **OK**.

Moving Signal Groups

To reposition a group in the stack of group panes:

1. In Signal Builder, select the pane.
2. To move the group lower in the stack, select **Group > Move Down**.
3. To move the pane higher in the stack, select **Group > Move Up**.

Editing Signals

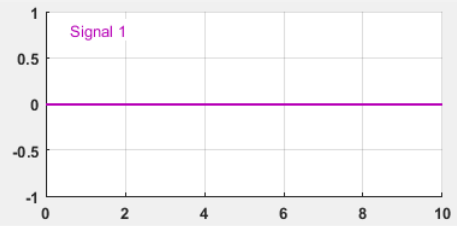
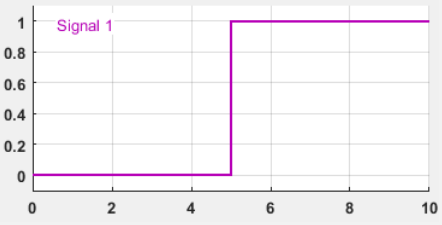
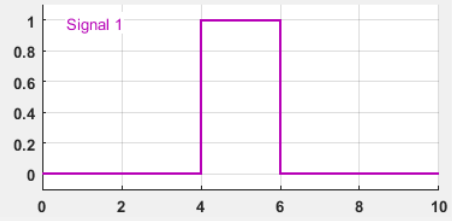
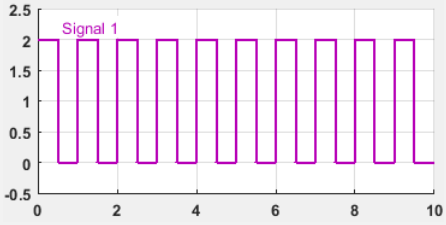
Signal Builder allows you to create, cut and paste, hide, and delete signals from signal groups.

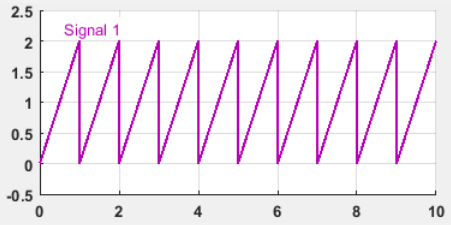
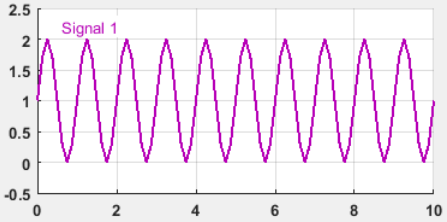
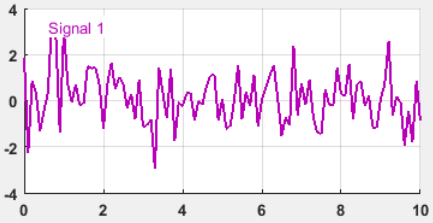
Creating Signals

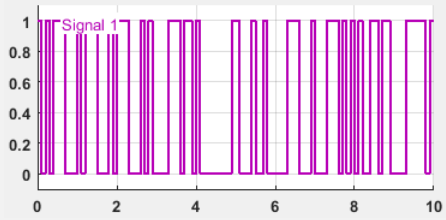
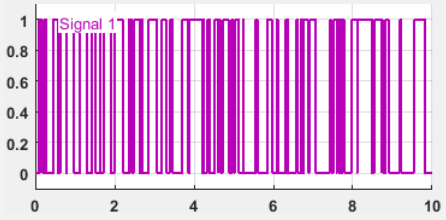
To create a signal in the currently selected signal group:

1. In Signal Builder, from the Active Group list, select the group you want to add the signal to.
2. Select **Signal > New**.

   The menu lists the waveforms you can add.

Waveform	Description	Inputs	Output
Constant	Constant waveform	None	
Step	Step waveform	None	
Pulse	Pulse waveform	None	
Square	Square waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset • **% Duty cycle** Percent of the period the signal is positive (a value between 0 and 100)	

Waveform	Description	Inputs	Output
Sawtooth	Sawtooth waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset	
Sampled Sin	Sampled sine waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset • **Samples Per Period** Number of samples per waveform period	
Sampled Gaussian Noise	Sampled Gaussian noise waveform based on a Gaussian distribution with input mean and standard deviation at input frequency	• **Frequency (Hz)** Waveform frequency, in hertz • **Mean** The mean value of the random variable output • **Standard Deviation** The standard deviation squared of the random variable output • **Seed (empty to use current state)** The initial seed value for the random number generator	

Waveform	Description	Inputs	Output
Pseudorandom Noise	Pseudorandom noise waveform based on a binomial distribution with upper and lower values at input frequency	• **Frequency (Hz)** Frequency with which waveform fluctuates between **Upper value** and **Lower value**, in hertz • **Upper value** Upper limit of signal • **Lower value** Lower limit of signal • **Seed** The initial seed value for the random number generator	
Poisson Random Noise	Poisson random noise waveform that alternates between 0 and 1	• **Avg rate (1/sec)** Average rate of transition between 0 and 1 • **Seed (empty to use current state)** The initial seed value for the random number generator	

Waveform	Description	Inputs	Output
Custom	Custom piecewise linear waveform; custom values must fit within the display area	• **Time values** Vector of two or more time coordinates • **Y values** Vector of two or more signal amplitudes that correspond to the values in **Time values** The entries in either field can be any MATLAB expression that evaluates to a vector, including the results from the evaluation of a MATLAB workspace variable. The resulting vectors must be of equal length. **Note** Signal Builder displays a warning if you add a custom waveform with a large number of data points (100,000,000 or more).	

3. Select the waveform you want to add.
4. Specify the inputs (in the prompt), and click **OK**.

If you select a standard waveform, Signal Builder adds a signal with that waveform to the group. If you select a custom waveform, you are prompted for **Time values** and **Y values**.

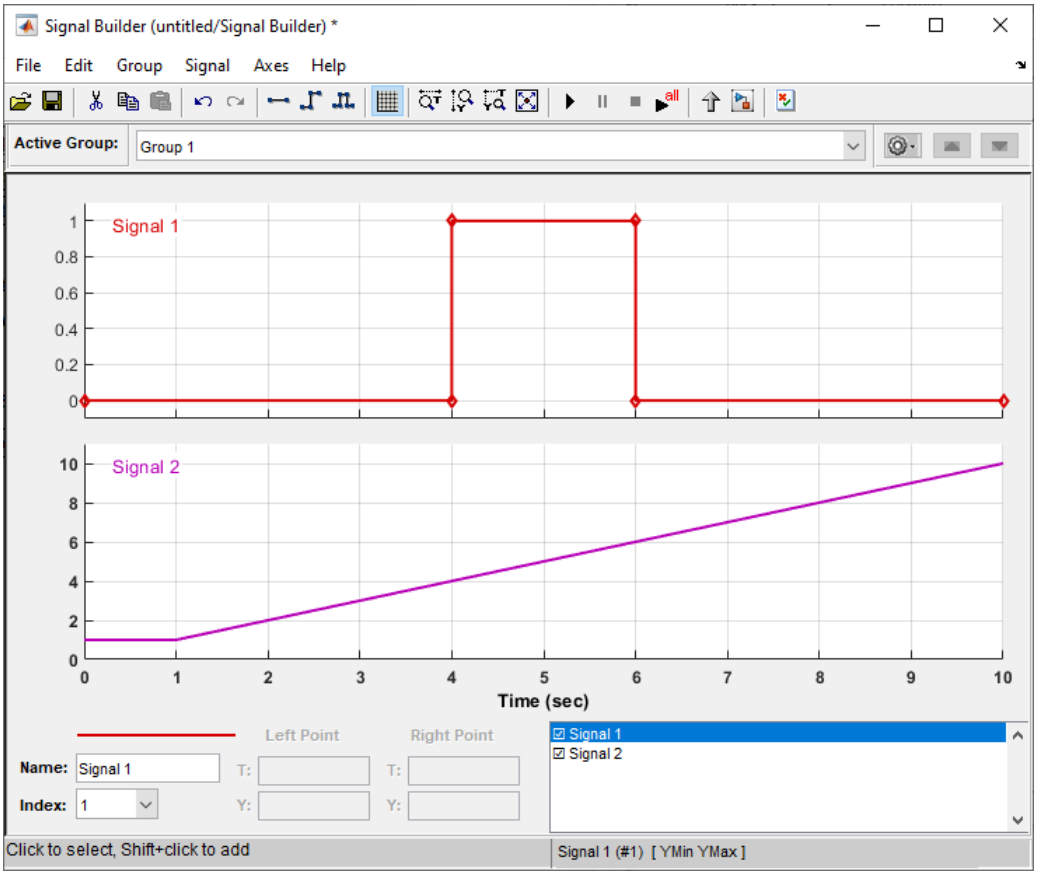
You can also use MATLAB workspace variables to create new signals.

1. In the MATLAB Command Window, create data for two variables, *t* and *y*.

   ```
   t = 1:10
   y = 1:10
   ```

   These vectors must be the same size.
2. Create a model and add a Signal Builder block.
3. Double-click the Signal Builder block.
4. Select **Signal** > **New** > **Custom**.
5. In the Custom Waveform window, enter *t* in the **Time values** field and *y* in the **Y values** field and then click **OK**.

   The Signal Builder block window displays the new signal as Signal 2.



Defining Signal Output

To specify the type of output to use for sending test signals:

1. In Signal Builder, select **Signal** > **Output**.
2. From the list, select:

   - **Ports**

     Default. Sends individual signals from the block. An output port named Signal *N* appears for each Signal *N*.

   - **Bus**

     Sends single, virtual, nonhierarchical bus of signals from the block. An output port named Bus appears.

Tip

- You cannot use the **Bus** option to create a bus of nonvirtual signals.
- The **Bus** option enables you to change your model layout without having to reroute Signal Builder block signals. Use the Bus Selector block to select the signals from this bus.

- If you create a Signal Builder block using the Signal & Scope Manager or using the **Create & Connect Generator** option from a signal line context menu, you cannot define signal output. In these cases, the block sends individual signals.

Copying and Pasting Signals

To copy a signal from one group and paste it into another group as a new signal:

1. In Signal Builder, select the signal you want to copy.
2. Select **Edit > Copy**.
3. Select the group you want to paste the signal into.
4. Select **Edit > Paste**.

To copy a signal from one axis and paste it into another axis to replace its signal:

1. Select the signal you want to copy.
2. Select **Edit > Copy**.
3. Select the signal on the axis that you want to update.
4. Select **Edit > Paste**.

Deleting Signals

To delete a signal, in Signal Builder, select the signal and choose **Delete** or **Cut** from the **Edit** menu. Signal Builder deletes the signal from the current group. Because each signal group must contain the same number of signals, Signal Builder also deletes all signals sharing the same index in the other groups.

Renaming a Signal

To rename a signal:

1. In Signal Builder, select **Signal > Rename**.

   A dialog box appears with an edit field that displays the current name of the signal.
2. Edit or replace the current name with a new name.
3. Click **OK**.

You can also edit the signal name in the **Name** field in the lower-left corner of the Signal Builder window.

Replacing a Signal

To replace a signal:

1. In Signal Builder, select the signal, then select **Signal > Replace with** .

   A menu of waveforms appears. It includes a set of standard waveforms (**Constant**, **Step**, and so on) and a **Custom** waveform option.
2. Select one of the waveforms.

If you select a standard waveform, Signal Builder replaces a signal in the currently selected group with that waveform. For other waveforms, the Signal Builder displays a dialog to allow you to provide input for the requested waveform.

Waveform	Description	Inputs
Constant	Constant waveform.	None
Step	Step waveform.	None
Pulse	Pulse waveform	None
Square	Square waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset • **% Duty cycle** Percent of the period in which the signal is positive. Enter a value between 0 and 100.
Sawtooth	Sawtooth waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset
Sampled Sin	Sampled sine waveform	• **Frequency (Hz)** Waveform frequency, in hertz • **Amplitude** Waveform amplitude • **Y Offset** Waveform vertical offset • **Samples Per Period** Number of samples per waveform period

Waveform	Description	Inputs
Sampled Gaussian Noise	Sampled Gaussian noise waveform based on a Gaussian distribution with input mean and standard deviation at input frequency.	• **Frequency (Hz)** Waveform frequency, in hertz • **Mean** The mean value of the random variable output • **Standard Deviation** The standard deviation squared of the random variable output • **Seed (empty to use current state)** The initial seed value for the random number generator
Pseudorandom Noise	Pseudorandom noise waveform based on a binomial distribution with upper and lower values at input frequency.	• **Frequency (Hz)** Frequency with which waveform fluctuates between **Upper value** and **Lower value**, in hertz • **Upper value** Upper limit of signal • **Lower value** Lower limit of signal • **Seed** The initial seed value for the random number generator
Poisson Random Noise	Poisson random noise waveform that alternates between 0 and 1	• **Avg rate (1/sec)** Average rate of transition between 0 and 1 • **Seed (empty to use current state)** The initial seed value for the random number generator
Custom	Custom piecewise linear waveform. Custom values must fit within the display area.	• **Time values** Vector of two or more time coordinates • **Y values** Vector of two or more signal amplitudes that correspond to the values in **Time values** The entries in either field can be any MATLAB expression that evaluates to a vector. The resulting vectors must be of equal length. **Note** Signal Builder returns a warning if you add a custom waveform with a large number of data points (100,000,000 or more). You can then cancel the action.

You can also edit the signal name in the **Name** field in the lower-left corner of the Signal Builder window.

Changing a Signal Index

To change a signal index:

1. In Signal Builder, select the signal, then select **Signal > Change Index**.

   A dialog box appears with a drop-down list field containing the existing index of the signal.
2. From the drop-down list, select another index and select **OK**. Or select an index from the **Index** list in the lower-left corner of the Signal Builder window.

Hiding Signals

By default, the Signal Builder window displays the group waveforms in the group pane. To hide a waveform:

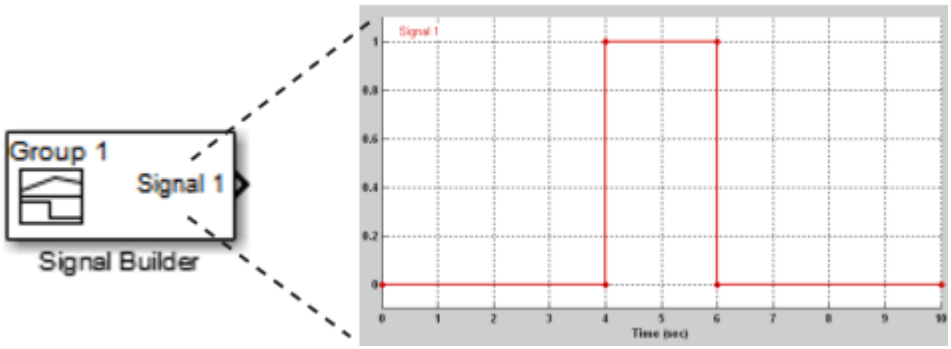
1. In Signal Builder, select the waveform, then select **Signal > Hide**.
2. To redisplay a hidden waveform, select the **Group** pane, then select **Signal > Show**.
3. Select the signal from the list. Alternatively, you can hide and redisplay a hidden waveform by double-clicking its name in the Signal Builder signal list.

Creating Signal Group Sets Manually

This topic describes how to create signal group sets manually. If you have signal data files, such as those from test cases, consider importing this data as described in “Importing Signal Group Sets” on page 75-61.

To create an interchangeable set of signal groups:

1. Drag an instance of the Signal Builder block from the Simulink Sources library and drop it into your model.



   By default, the block represents a single signal group containing a single signal source that outputs a square wave pulse.
2. Use the block signal editor to create additional signal groups, add signals to the signal groups, modify existing signals and signal groups, and select the signal group that the block outputs.

Note Each signal group must contain the same number of signals.

3 Connect the output of the block to your diagram.

   The block displays an output port for each signal that the block can output.

You can create as many Signal Builder blocks as you like in a model, each representing a distinct set of interchangeable groups of signal sources. When a group has multiple signals, the signals might have different end times. However, Signal Builder block requires the end times of signals within a group to match. If a mismatch occurs, Signal Builder block matches the end times by holding the last value of the signal with the smaller end time.

See “Simulating with Signal Groups” on page 75-85 for information on using signal groups in a model.

Importing Signal Group Sets

The topics in this section describe how to import signal data into the Signal Builder block. You should already have a signal data file whose contents you want to import. For example, you might have signal data from previously run test cases. See “Importing Signal Groups from Existing Data Sets” on page 75-61 for a description of the data formats that the Signal Builder block accepts. The procedures in the following topics use the file `3Grp_3Sig.xls` in the folder *`matlabroot`*`\help\toolbox\simulink\ug\examples\signals` (open).

Signal Builder accepts signals only of type double.

If you import a `Simulink.SimulationData.Dataset` data set, the block imports it as its own group.

Importing Signal Groups from Existing Data Sets

You might have existing signal data sets that you want to enter into the Signal Builder block. The **File** > **Import from File** command on the Signal Builder window starts the Import File dialog box. This dialog box is modal, which means that focus cannot change to another MATLAB window while the dialog box is running. If you want to see changes in the Signal Builder window after you import data, do one of the following:

- Close the Import File dialog box.
- Set up the Import File dialog box and Signal Builder window side by side.

Note You cannot undo the results of importing a signal data file. In addition, you cannot undo the last action performed before opening the Import File dialog box. When you close the Import File dialog box, the **Undo last edit** and **Redo last edit** buttons on the Signal Builder window are grayed out. These buttons are grayed out regardless of whether you imported a data file.

The Import File dialog box accepts the following appropriately formatted file types:

- Microsoft Excel (`.xls`, `.xlsx`)
- Comma-separated value (CSV) text files (`.csv`)
- MAT-files (`.mat`)

Tip To import signal data from a Microsoft Excel spreadsheet, consider using the From Spreadsheet block. The From Spreadsheet block incrementally loads data from the spreadsheet during simulation.

If you use a From Spreadsheet block, you do not need to do anything to handle changes to sheet values.

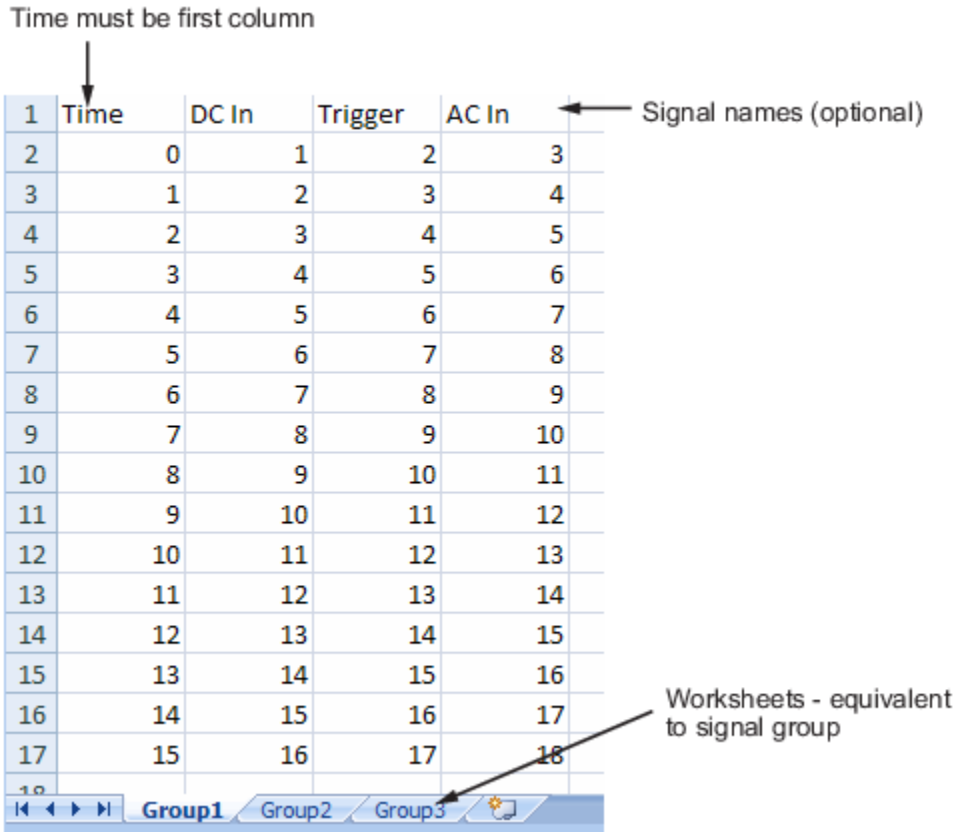
Note Signal Builder block uses the `xlsread` function. See the `xlsread` documentation for information on supported platforms.

You can import your data set file only if it is appropriately formatted.

For Microsoft Excel spreadsheets:

- The Signal Builder block interprets the first row as signal name. If you do not specify a signal name, the Signal Builder block assigns a default one with the format `Imported_Signal #`, where # increments with each additional unnamed signal.
- The Signal Builder block interprets the first column as time. In this column, the time values must increase.
- The Signal Builder block interprets the remaining columns as signals.
- If there are multiple sheets:
  - Each sheet must have the same number of signals (columns).
  - Each sheet must have the same set of signal names (if any).
  - Each column on each sheet must have the same number of rows.
- Signal Builder block interprets each worksheet as a signal group.

This example contains an acceptably formatted Microsoft Excel spreadsheet. It has three worksheets named Group1, Group2, and Group3, representing three signal groups.



	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

For CSV text files:

- Each file contains only numbers. Do not name signals in a CSV file.
- The Signal Builder block interprets the first column as time. In this column, the time values must increase.
- The Signal Builder block interprets the remaining columns as signals.
- Each column must have the same number of entries.
- The Signal Builder block interprets each file as one signal group.
- The Signal Builder block assigns a default signal name to each signal with the format `Imported_Signal #`, where # increments with each additional signal.

This example contains 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
```

For MAT-files:

- The Signal Builder block supports data store logging that the `Simulink.SimulationData.Dataset` object represents and interprets this data as a single group.
- The Signal Builder block supports Simulink output saved as a structure with time.
- The Signal Builder block supports the Signal Builder data format. This format is a group of cell arrays that must be labeled.
- Signal Builder block does not support:
  - Simulink output as only a structure
  - Simulink output as only an array

Note Signal Builder returns a warning if you import a large number of data points (100,000,000 or more). You can then cancel the action.

This example contains an acceptably logged MATLAB workspace. Use the MATLAB workspace **Save** command to save the variables to a MAT-file. Import this file to the Signal Builder block.

Replacing All Signal Data with Selected Data

Simulink software creates a default Signal Builder block with one signal. To replace this signal and all other signal data that the block might display:

1. Create a model and drag a Signal Builder block into that model.
2. Double-click the block.

   The Signal Builder window appears with its default Signal 1.
3. In Signal Builder, select **File** > **Import from File**.

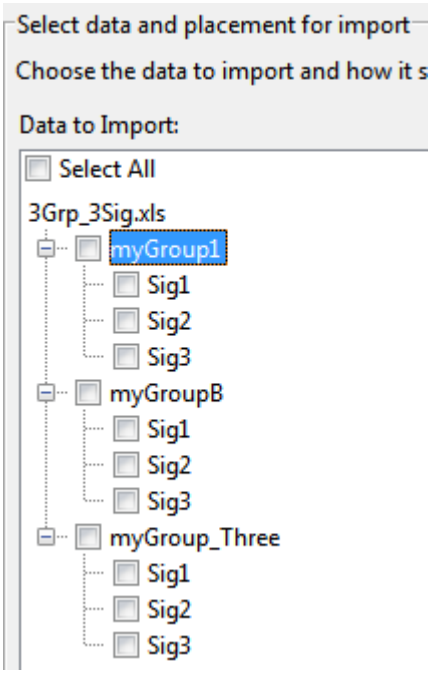
   The Import File dialog box appears.
4. In the **File to Import** field, enter a signal data file name or click **Browse**.

   The file browser appears.
5. If you select the file browser, navigate to and select a signal data file. For example, select `3Grp_3Sig.xls`.

   Note If you try to import an improperly formatted data file, an error message pops up. When you click to dismiss this window, the **Status History** pane displays a more detailed error message (if there is one). For example:

   ```
   File 'signals.mat' format does not
   comply with Signal Builder required format.
   There is no 'time' parameter defined.
   ......................
   ```

   The **Data to Import** pane contains the signal data from the file. Click the expander to display all the signals.



6 Select the signals you want to import. To import all the signals, click **Select All**.

7 From the **Placement for Selected Data** list, select the action to take on the signal data. For example, select `Replace existing dataset`.

The **Confirm Selection** button is activated. Validate your signal selection before the Signal Builder block performs the specified action. If the signal data selection is not appropriate, **Confirm Selection** remains grayed out. For example, **Confirm Selection** remains grayed out if the number of signals you select is not the same as the number of signals in the Signal Builder group that you want to replace.

8 Click the **Confirm Selection** button.

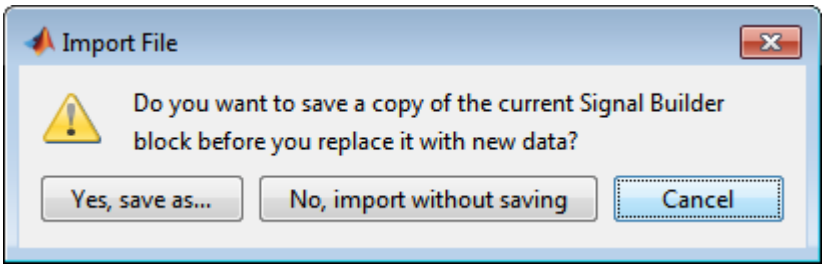
If the requested action is a valid one, the Status History pane displays messages to indicate the status. For example:

```
Current data in Signal Builder will be replaced
by new data.
Number of groups: 3
Group name(s):
  myGroup1
  myGroupB
  myGroup_Three

Number of signals per group: 3
Signal name(s):
  Sig1
  Sig2
  Sig3


 (Names may have been renamed for uniqueness.)
 ......................
```

The confirmation also enables the **OK** and **Apply** buttons.

9 If you are satisfied with the status message, click **Apply** to replace the existing signal data with the contents of this file.

When selecting `Replace existing dataset`, the software gives you the opportunity to save the existing contents of the Signal Builder block.

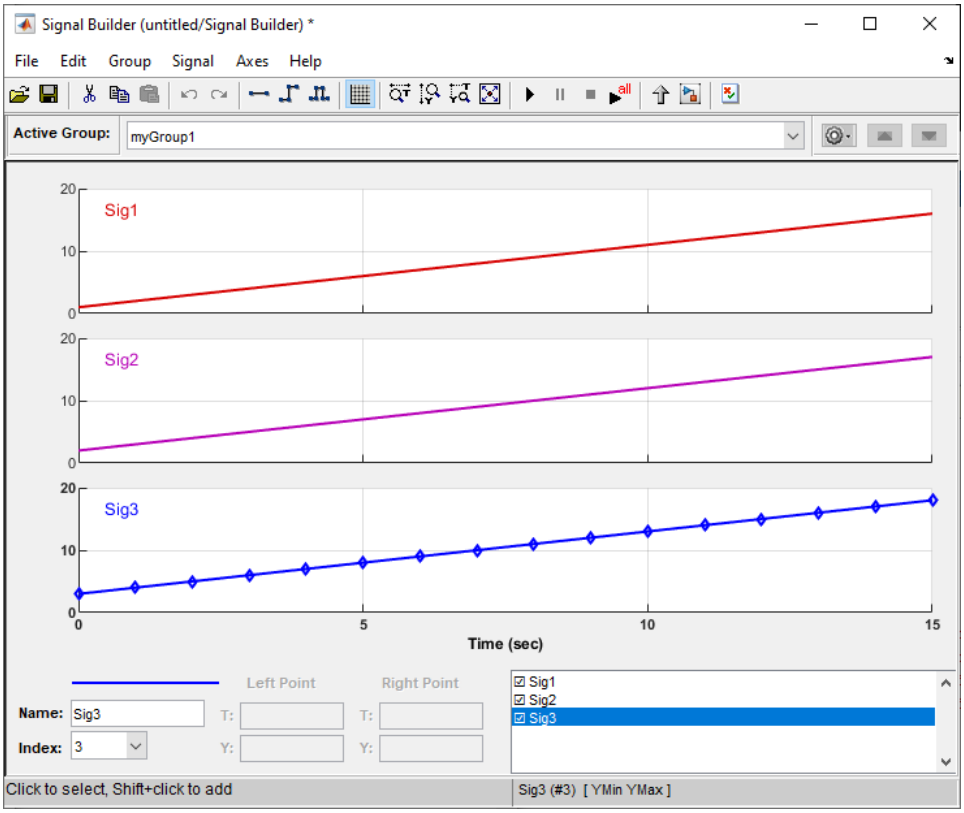


10 Click a button, as follows:

To...	Click...
Save the contents of the Signal Builder block before replacing it with the new signal data. **Note** This selection prompts you to save the Signal Builder block in a model name of your choice. The software saves only the Signal Builder block and no other model content.	**Yes, save as**
Replace the contents of the Signal Builder block without saving them first.	**No, import without saving**
Stop the replacement process.	**Cancel**

For this example, select **No, import without saving** to replace the contents of the Signal Builder block.

11 The Signal Builder block updates with the new signal data. Click **OK** to close the Import File dialog box and inspect the Signal Builder block.



12 Click **OK**.

13 Inspect the updated Signal Builder window to confirm that your signal data is intact.

14 Close the Signal Builder window and save and close the model. For example, save the model as `signalbuilder1`.

Appending Selected Signals to All Existing Signal Groups

You can import signals from a signal data file and append selected signals to the end of all existing signal groups. If the signal names to be appended are not unique, the software renames them by incrementing each name by 1 or higher until it is a unique signal name. For example, if the block and data file contain signals named `thermostat`, the software renames the imported signal to `thermostat1` upon appending. If you add another signal named `thermostat`, the software names that latest version `thermostat2`.

This topic uses `signalbuilder1` from the procedure in “Replacing All Signal Data with Selected Data” on page 75-64.

1 In the MATLAB Command Window, type `signalbuilder1`.

2 Double-click the Signal Builder block.

The Signal Builder window appears.

3 In the Signal Builder window, select **File > Import from File**.

The Import File dialog box appears.

4 In the **File to Import** field, enter a signal data file name or click **Browse**.

The file browser is displayed.

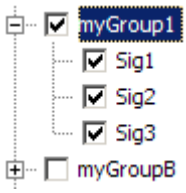
5 If you select the file browser, navigate to and select a signal data file. For example, select `3Grp_3Sig.xls`.

Note If you try to import an improperly formatted signal data file, an error message pops up. When you click to dismiss this window, the **Status History** pane displays an error message. For example:

```
File 'signals.mat' format does not
comply with Signal Builder required format.
There is no 'time' parameter defined.
.....................
```

The **Data to Import** pane contains the signal data from the file. Click the expander to display all the signals.

6 Select the signals you want to import. In this example, there are three groups, myGroup1, myGroupB, and myGroup_Three. Select all the signals in myGroup1.

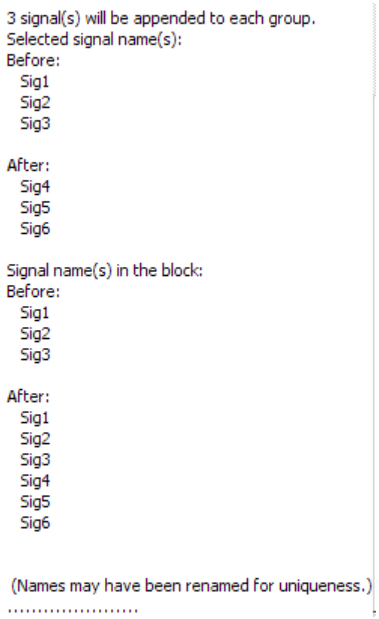


7 From the **Placement for Selected Data** list, select the action to take on the signal data. For example, select `Append selected signals to all groups`.

The **Confirm Selection** button is activated. Validate your signal selection before the Signal Builder block performs the specified action. If the signal data selection is not appropriate, **Confirm Selection** remains grayed out. For example, **Confirm Selection** remains grayed out if the number of signals you select is not the same as the number of signals in the Signal Builder group that you want to replace.

8 Click the **Confirm Selection** button.

If the requested action is a valid one, the Status History pane displays messages to indicate the state. For example:

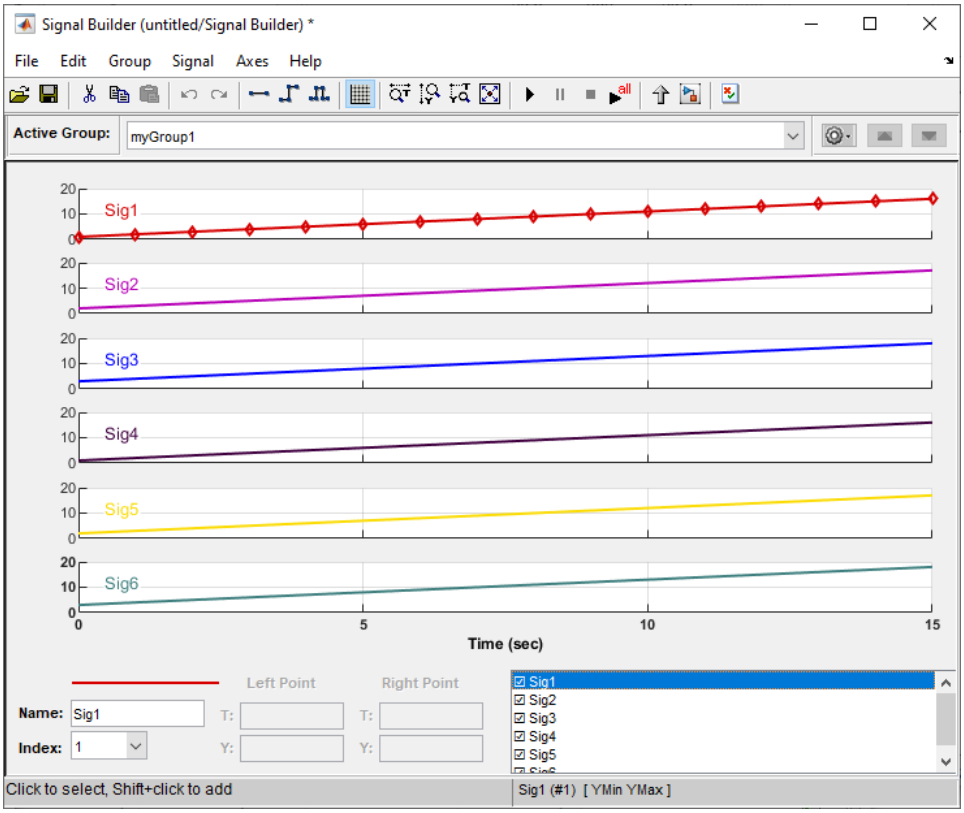


The confirmation also enables the **OK** and **Apply** buttons.

Observe the **Before** and **After** headings for the signals. These sections indicate the names of the block and imported data signals before and after the append action.

9 If you are satisfied with the status message, click **Apply** to append the selected signals to all the signal groups in the Signal Builder block.

10 The Signal Builder block updates with the new signal data. Click **OK** to close the Import File dialog box and inspect the Signal Builder block.



11 Click **OK**.

12 Inspect the updated Signal Builder window to confirm that your signal data is intact. Notice that the software has renamed the signals Sig1, Sig2, and Sig3 from the signal data file to Sig4, Sig5, and Sig6 in the Signal Builder block.

13 Close the Signal Builder window and save and close the model. For example, save the model as `signalbuilder2`.

Appending Selected Signals to Sequential Existing Signal Groups

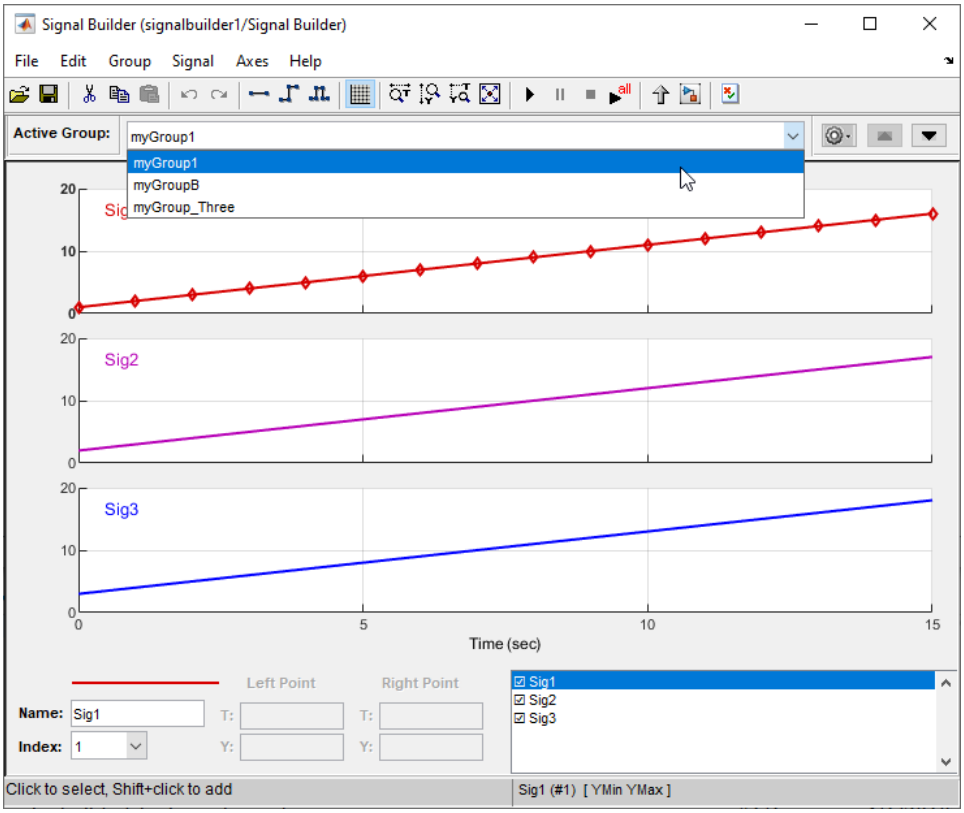
You can append signals, in the order in which they are selected, to the end of sequential signal groups. This statement means that you select the same number of signals as there are signal groups, and sequentially append each signal to a different group. The software renames each appended signal to the name of the last appended signal.

This topic uses `signalbuilder1` from the procedure in “Replacing All Signal Data with Selected Data” on page 75-64.

1 In the MATLAB Command Window, type `signalbuilder1`.

2 Double-click the Signal Builder block.

3 Note how many groups exist in the Signal Builder block. For example, this Signal Builder block has three groups, myGroup1, myGroupB, and myGroup_Three.



4 In the Signal Builder window, select **File > Import from File**.

5 In the **File to Import** field, enter a signal data file name or click **Browse**.

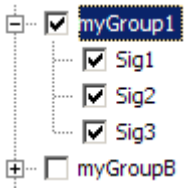
6 If you select the file browser, navigate to and select a signal data file. For example, select `3Grp_3Sig.xls`.

Note If you try to import an improperly formatted signal data file, an error message pop-up window appears. When you click to dismiss this window, the **Status History** pane displays an error message. For example:

```
File 'signals.mat' format does not
comply with Signal Builder required format.
There is no 'time' parameter defined.
```

The **Data to Import** pane contains the signal data from the file. Click the expander to display all the signals.

7 Select the signals you want to import. In this example, there are three groups, myGroup1, myGroupB, and myGroup_Three. Select all the signals in myGroup1.



8 From the **Placement for Selected Data** list, select the action to take on the signal data. For example, select `Append selected signals to different groups (in order)`.

The **Confirm Selection** button is activated. Validate your signal selection before the Signal Builder block performs the specified action.

9 Click the **Confirm Selection** button.

If the requested action is a valid one, the Status History pane displays messages to indicate the state. For example:

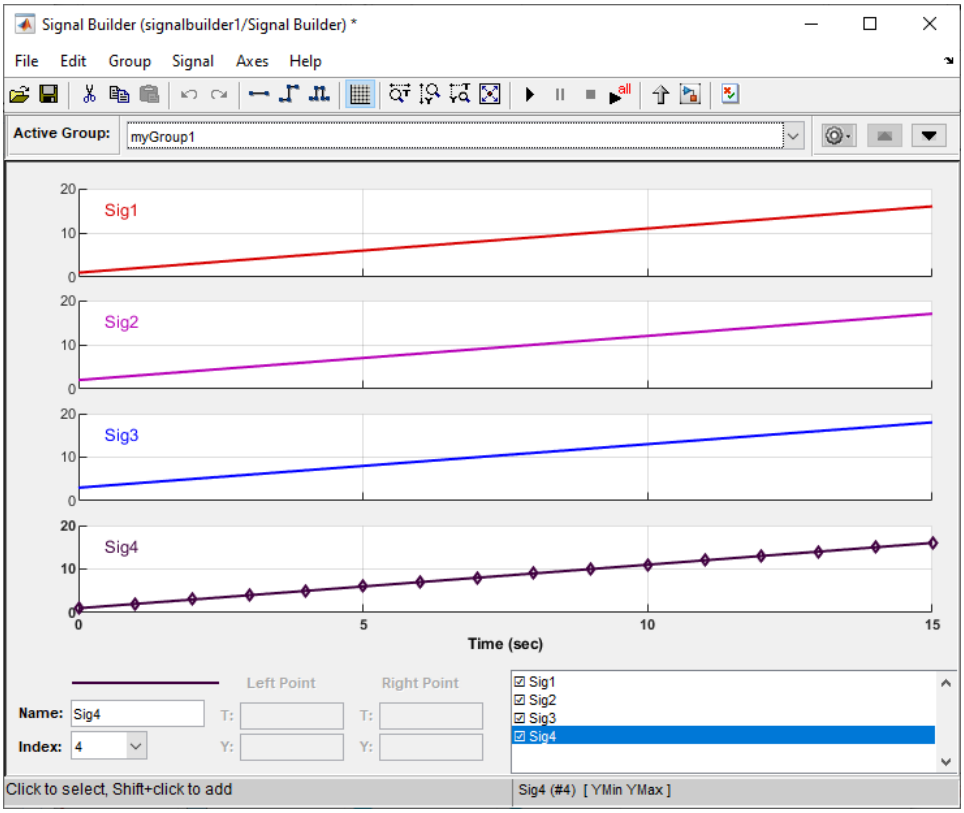
```
 1 signal will be appended to each group.
Selected signal names:
Before:
  Sig1
  Sig2
  Sig3

Selected unique signal name:
After:
  Sig4
```

The confirmation also enables the **OK** and **Apply** buttons.

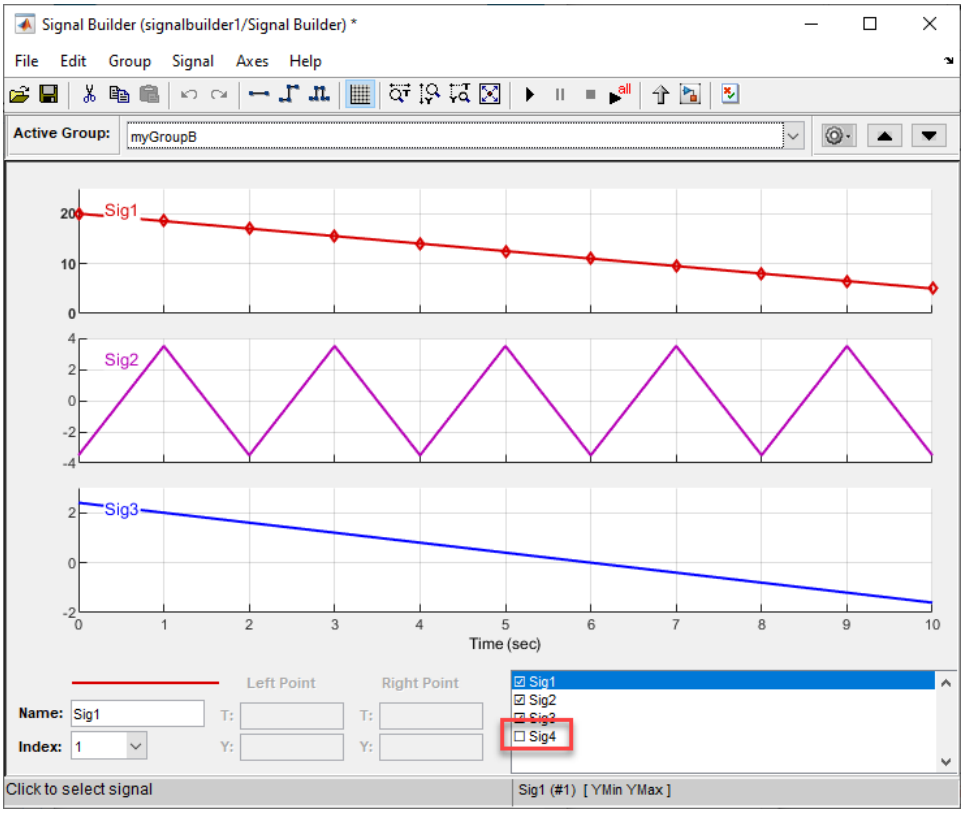
10 If you are satisfied with the status message, click **Apply** to append the signals.

The Signal Builder block updates with the new signal data. Click **OK** to close the Import File dialog box and inspect the three groups of the Signal Builder block.

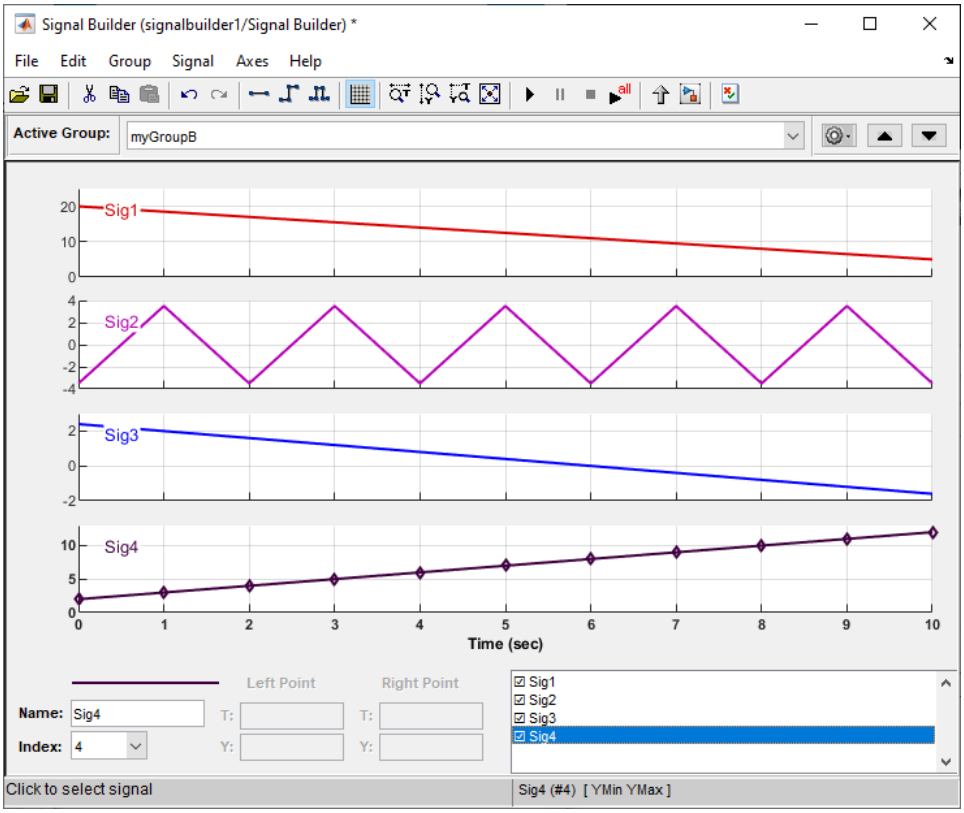
The topmost signal group, myGroup1, shows all signals by default, including the new Sig4.



11 Click another group name, for example, myGroupB. Notice that Sig4 exists for the group, hidden by default.



12 To show Sig4 on this pane, double-click Sig4 in the Selection Status area of the pane. The graph is updated to reflect Sig4.



13 Close the Signal Builder window and save and close the model. For example, save the model as `signalbuilder3`.

Appending Signal Groups to Existing Groups

You can append one or more signal groups to the end of the list of existing signal groups. If the block already has a signal group with the same name as the one you are adding, the software increments the group name by 1 or higher until it is unique before adding it. For example, if the block and data file contain groups named `MyGroup1`, the software renames the imported group to `MyGroup2` upon appending. If you add another group named `MyGroup1`, the software names that latest version `MyGroup3`.

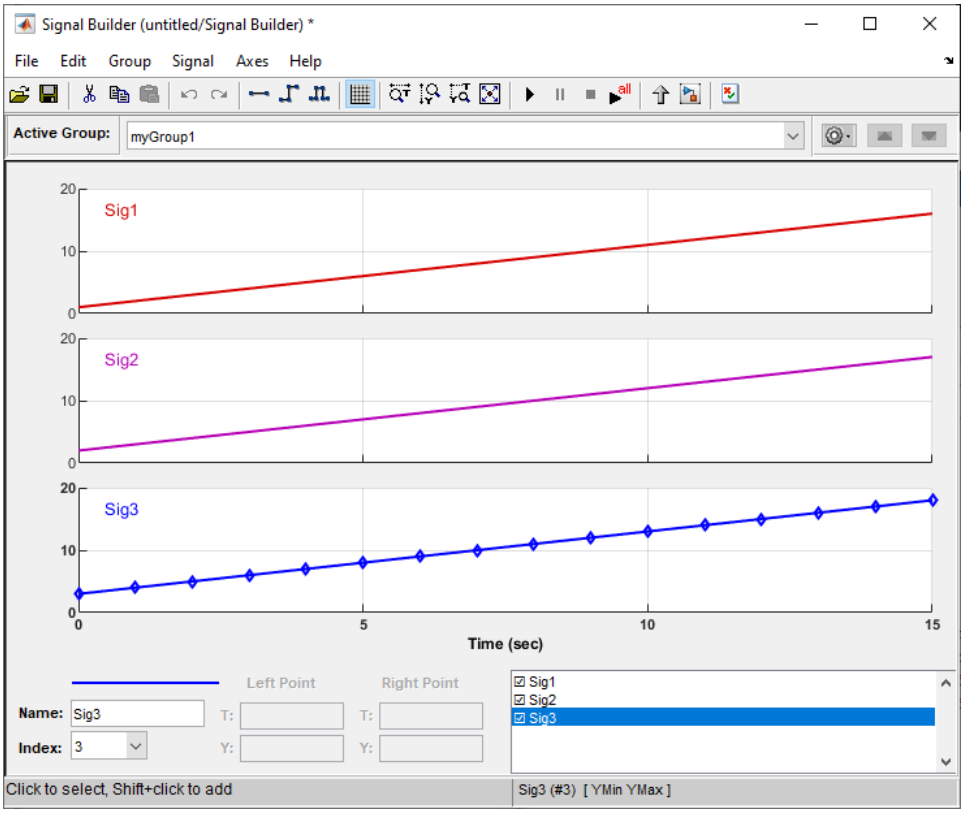
This topic uses `signalbuilder1` from the procedure in “Replacing All Signal Data with Selected Data” on page 75-64.

1 In the MATLAB Command Window, type `signalbuilder1`.

2 Double-click the Signal Builder block.

The Signal Builder window appears.

3 Note how many groups exist in the Signal Builder block, and how many signals exist in each group. The Signal Builder block requires that all groups have the same number of signals. For example, this Signal Builder block has three groups, myGroup1, myGroupB, and myGroup_Three. Three signals exist in each group.



4 Double-click the block.

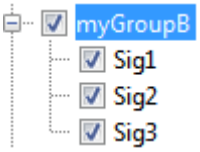
5 In the **File to Import** text field, enter a signal data file name or click **Browse**.

6 If you select the file browser, navigate to and select a signal data file. For example, select `3Grp_3Sig.xls`.

The **Data to Import** pane contains the signal data from the file. Click the expander to display all the signals.

7 Evaluate the number of signals in the groups of this data file. If the number of signals in each group equals the number of signals in the groups that exist in the block, you can append one of these groups to the block.

8 Select the group you want to import. In this example, there are three groups, myGroup1, myGroupB, and myGroup_Three. Select myGroupB.



9 From the **Placement for Selected Data** list, select the action to take on the signal group. For example, select `Append groups`.

The **Confirm Selection** button is activated. Validate your signal selection before the Signal Builder block performs the specified action.

10 Click the **Confirm Selection** button.

If the requested action is a valid one, the Status History pane displays messages to indicate the state. For example:

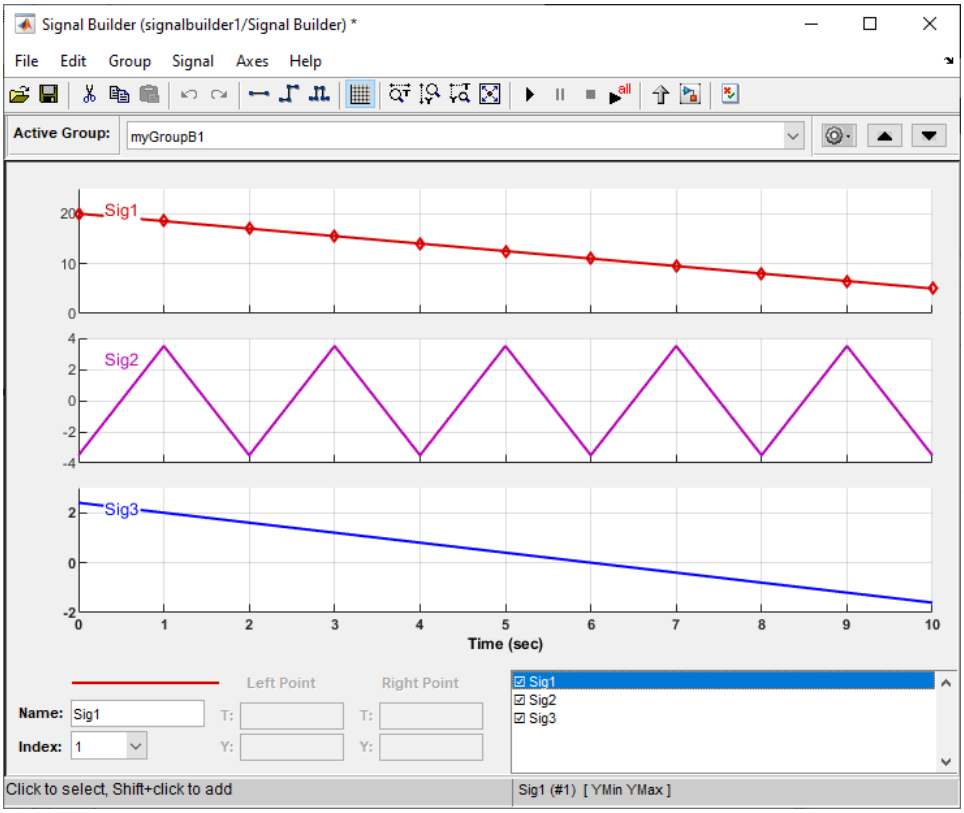
```
1 group(s) (each with 3 signal(s)),
will be appended to the existing block.
Selected group name(s):
Before:
  myGroupB

After:
  myGroupB1


Signal name(s) in selected group(s):
Before:
  Sig1
  Sig2
  Sig3

Signal name(s) in the block:
Before:
  Sig1
  Sig2
  Sig3

After:
  Sig1
  Sig2
  Sig3
```

The confirmation also enables the **OK** and **Apply** buttons.

11 If you are satisfied with the status message, click **Apply** to append the signals.

The Signal Builder block updates with the new signal data. Click **OK** to close the Import File dialog box and inspect the groups of the Signal Builder block.

Notice the addition of the new signal group as the last pane. Because there is already a signal group named myGroupB, the software automatically increments the new signal group name by 1. Select myGroupB.



12 Close the Signal Builder window and save and close the model. For example, save the model as `signalbuilder4`.

Appending Signals with the Same Name to Existing Signal Groups

If you append a signal whose name is the same as a signal that exists in the Signal Builder block, the software increments the name of the appended signal by 1. The software repeats incrementing until the appended signal name is unique. For example:

1 Assume your Signal Builder block has a signal group, `myGroup1`, with the signals Sig1, Sig2, and Sig3.

2 Append a signal named Sig1 to `myGroup1`.

3 Observe that the software increments Sig1 to Sig4 before appending it to `myGroup1`.

Appending a Group of Signals with Different Signal Names

If you append a signal group whose signal names differ from those that exist in the Signal Builder block, the software changes the names of the existing signals to be the same as the appended signals. For example,

1 Assume your Signal Builder block has a signal group, `myGroup1`, with the signals Sig1, Sig2, and Sig3.

2 Append a signal group named `myGroup2` whose signal names are SigA, SigB, and SigC.

3 Observe that the software:

- Appends `myGroup2`.

- Renames the signals in `myGroup1` to be the same as those in `myGroup2`.

Importing Data with Custom Formats

This topic describes how to import signal data formatted in a custom format. You should already have the signal data from a file whose contents you want to import. See “Importing Signal Groups from Existing Data Sets” on page 75-61 for a description of the data formats that the Signal Builder block accepts. If your data is not formatted using one of these data formats, use the following workflow to import the custom formatted data. This workflow uses the following files, located in the folder *`matlabroot`*`\help\toolbox\simulink\examples` (open), as examples:

- `SigBldCustomFile.xls` — Signal data Microsoft Excel file using a format that Signal Builder block does not accept, for example:

Group1	Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Signal1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Signal2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Signal3	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Signal4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Group2	Signal1	1.6	2.6	3.6	4.6	5.6	6.6	7.6	8.6	9.6	10.6	11.6	12.6	13.6	14.6	15.6	16.6
	Signal2	1.8	2.8	3.8	4.8	5.8	6.8	7.8	8.8	9.8	10.8	11.8	12.8	13.8	14.8	15.8	16.8
	Signal3	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	Signal4	2.2	3.2	4.2	5.2	6.2	7.2	8.2	9.2	10.2	11.2	12.2	13.2	14.2	15.2	16.2	17.2

- `createSignalBuilderSupportedFormat.m` — Custom MATLAB function that uses `xlsread` to read Microsoft Excel spreadsheets. This example function reformats the custom data, in a format that the Signal Builder block supports, as follows:

  - *`grpNames`* — Cell array that contains group name character vectors with number of rows = 1, number of columns = number of groups.
  - *`sigNames`* — Cell array that contains signal name character vectors with number of rows = 1, columns = number of signals.
  - *`time`* — Cell array that contains time data with number of rows = number of signals, columns = number of groups.
  - *`data`* — Cell array that contains signal data with number of rows = number of signals, columns = number of groups.

  Signal Builder has the following requirements for this custom function:

  - Number of signals in each group must be the same.
  - Signal names in each group must be the same.
  - Number of data points in each signal must be the same.
  - Each element in the *time* and *data* cell array holds a matrix of real numbers. This matrix can be [1x*N*] or [*N*x1], where *N* is the number of data points in every signal.

1 Identify the format of your custom signal data, for example:

```
SigBldCustomFile.xls
```

2 Create a custom MATLAB function that:

a Uses a MATLAB I/O function, such as `xlsread`, to read your custom formatted signal data. For example, `createSignalBuilderSupportedFormat.m`.

b Formats the custom formatted signal data to one that the Signal Builder block accepts, for example, a MAT-file.

3 Use your custom MATLAB function to write your custom formatted signal data to a file that Signal Builder block accepts. For example:

```
createSignalBuilderSupportedFormat('SigBldCustomFile.xls', 'OutputData.mat')
```

4 Import the reformatted signal data file, `OutputData.mat`, into the Signal Builder block (see “Importing Signal Group Sets” on page 75-61).

Editing Waveforms

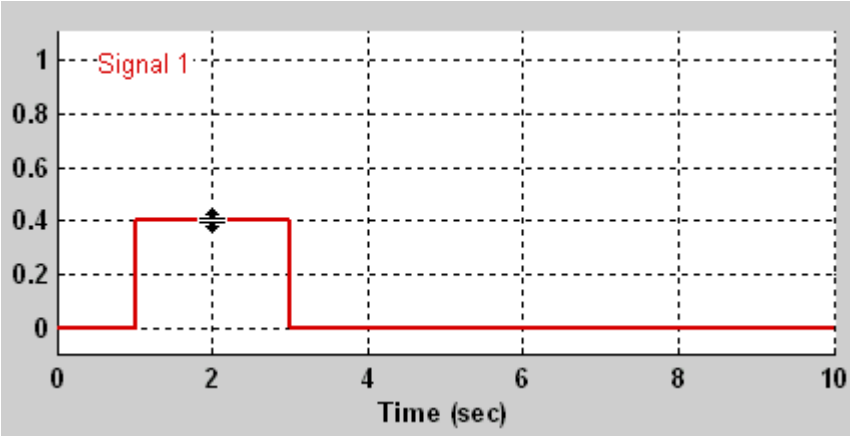
Signal Builder allows you to change the shape, color, and line style and thickness of the waveforms output by a group.

Reshaping a Waveform

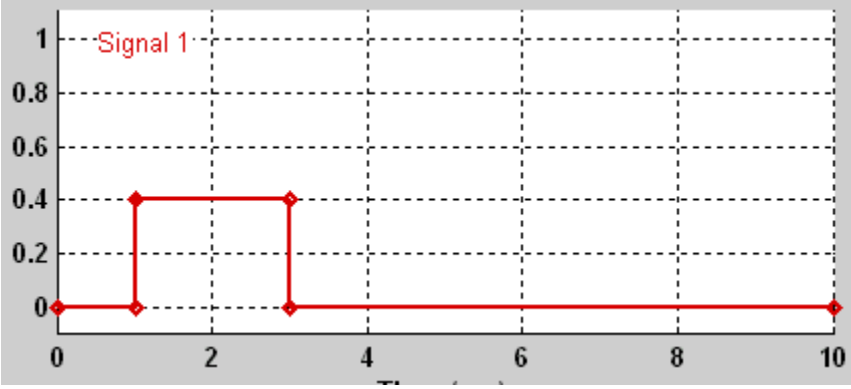
Signal Builder allows you to change the waveform by selecting and dragging its line segments and points with the mouse or arrow keys or by editing the coordinates of segments or points.

Selecting a Waveform

To select a waveform, left-click the mouse on any point on the waveform.



The Signal Builder displays the waveform points to indicate that the waveform is selected.

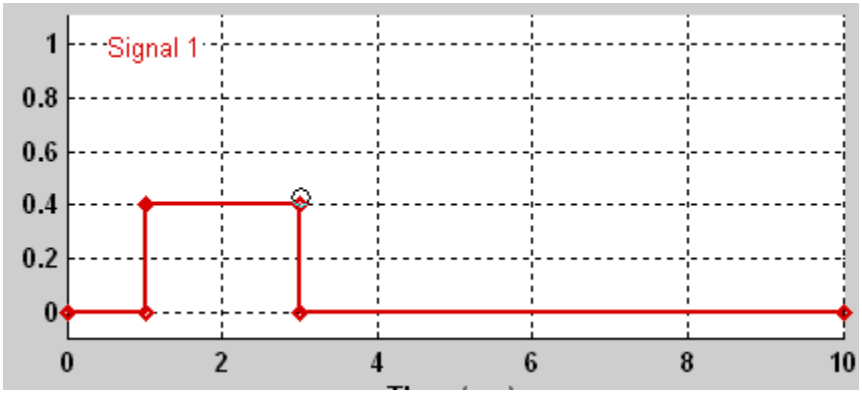


To deselect a waveform, left-click any point on the waveform axis that is not on the waveform itself or press the **Esc** key.

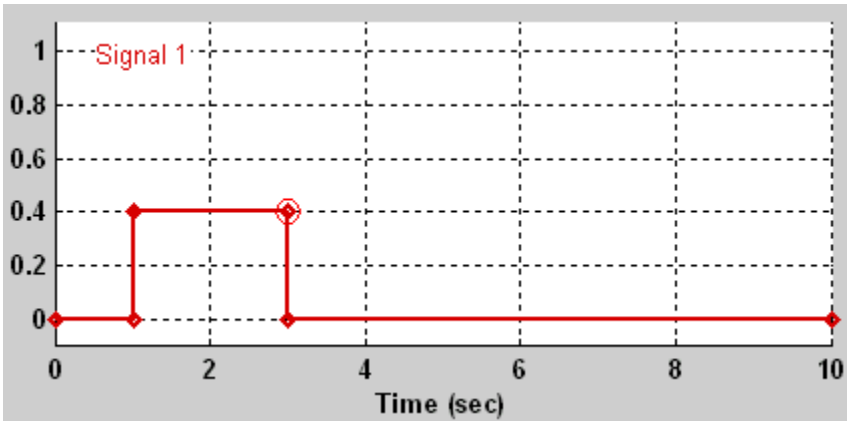
Working with Points

You can work with points in a waveform:

- To select a point in a waveform, first select the waveform. Then, position the mouse cursor over the point. The cursor changes shape to indicate that it is over a point.



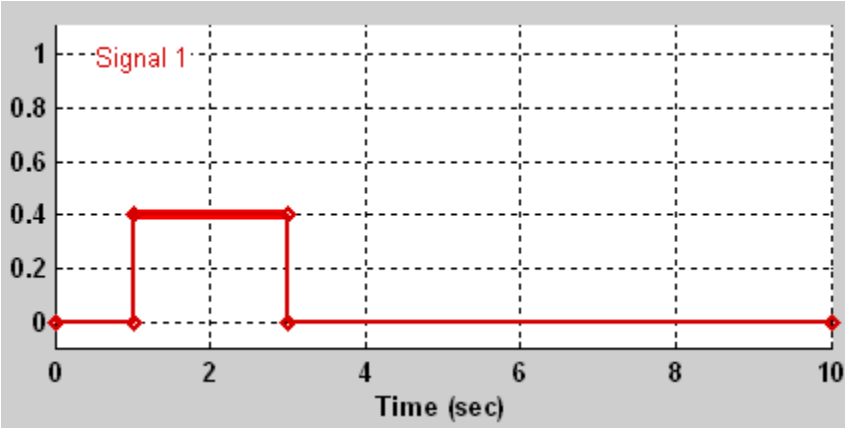
  Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate your selection.



- To insert a point, select the waveform and **Shift+**click the section for the point.
- To deselect the point, press the **Esc** key.
- To delete a point, select the point and press the **Backspace** or **Delete** keys.
- To edit a point with the `signalbuilder` function, use the `signalbuilder set` function to replace the waveform. You cannot programmatically remove a point.

Selecting Segments

To select a line segment, first select the waveform that contains it. Then, left-click the segment. The Signal Builder thickens the segment to indicate that it is selected.



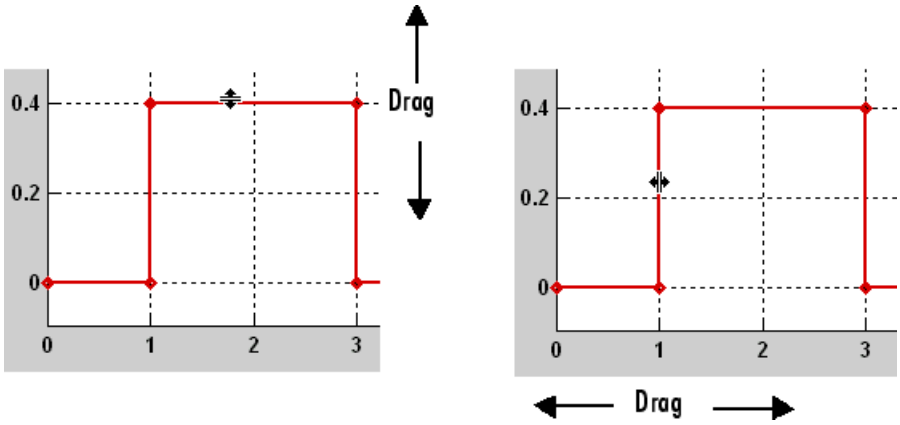
To deselect the segment, press the **Esc** key.

Moving Waveforms

To move a waveform, select it and use the arrow keys on your keyboard to move the waveform in the desired direction. Each key stroke moves the waveform to the next location on the snap grid (see “Snap Grid” on page 75-83) or by `0.1` inches if the snap grid is not enabled.

Dragging Segments

To drag a line segment to a new location, position the mouse cursor over the line segment. The mouse cursor changes shape to show the direction in which you can drag the segment.



Press the left mouse button and drag the segment in the direction indicated to the desired location. You can also use the arrow keys on your keyboard to move the selected line segment.

Dragging points

To drag a point along the signal amplitude (vertical) axis, move the mouse cursor over the point. The cursor changes shape to a circle to indicate that you can drag the point. Drag the point parallel to the *y*-axis to the desired location. To drag the point along the time (horizontal) axis, press the **Shift** key while dragging the point. You can also use the arrow keys on your keyboard to move the selected point.

Snap Grid

Each waveform axis contains an invisible snap grid that facilitates precise positioning of waveform points. The origin of the snap grid coincides with the origin of the waveform axis. When you drop a point or segment that you have been dragging, the Signal Builder moves the point or the segment points to the nearest point or points on the grid, respectively. The Signal Builder **Axes** menu allows you to specify the grid horizontal (time) axis and vertical (amplitude) axis spacing independently. The finer the spacing, the more freedom you have in placing points but the harder it is to position points precisely. By default, the grid spacing is 0, which means that you can place points anywhere on the grid; i.e., the grid is effectively off. Use the **Axes** menu to select the spacing that you prefer.

Inserting and Deleting points

To insert a point, first select the waveform. Then hold down the **Shift** key and left-click the waveform at the point where you want to insert the point. To delete a point, select the point and press the **Del** key.

Editing Point Coordinates

To change the coordinates of a point, first select the point. The Signal Builder displays the current coordinates of the point in the **Left Point** edit fields at the bottom of the Signal Builder window. To change the amplitude of the selected point, edit or replace the value in the **Y** field with the new value and press **Enter**. The Signal Builder moves the point to its new location. Similarly edit the value in the **T** field to change the time of the selected point.

Editing Segment Coordinates

To change the coordinates of a segment, first select the segment. The Signal Builder displays the current coordinates of the endpoints of the segment in the **Left Point** and **Right Point** edit fields at the bottom of the Signal Builder window. To change a coordinate, edit the value in its corresponding edit field and press **Enter**.

Changing the Color of a Waveform

To change the color of a waveform, select the waveform and then select **Color** from the Signal Builder **Signal** menu. The Signal Builder displays the MATLAB color chooser. Choose a new color for the waveform. Click **OK**.

Changing a Waveform Line Style and Thickness

The Signal Builder can display a waveform as a solid, dashed, or dotted line. It uses a solid line by default. To change the line style of a waveform, select the waveform, then select **Line Style** from the Signal Builder **Signal** menu. Select a line style from the menu.

To change the line thickness of a waveform, select the waveform, then select **Line Width** from the **Signal** menu. Edit the thickness value and click **OK**.

Signal Builder Time Range

The Signal Builder time range determines the span of time over which its output is explicitly defined. By default, the time range runs from 0 to 10 seconds. You can change both the beginning and ending times of a block time range (see “Changing a Signal Builder Time Range” on page 75-84).

If the simulation starts before the start time of a block time range, the block extrapolates its initial output from its first two defined outputs. If the simulation runs beyond the block time range, the

block by default outputs values extrapolated from the last defined signal values for the remainder of the simulation. The Signal Builder **Simulation Options** dialog box allows you to specify other final output options (see “Signal values after final time” on page 75-86 for more information).

Note When you click the **Start simulation** button on the Signal Builder block toolbar, the simulation uses the stop time of the model. The end of the time range specified in the waveform is not the stop time for the model.

Changing a Signal Builder Time Range

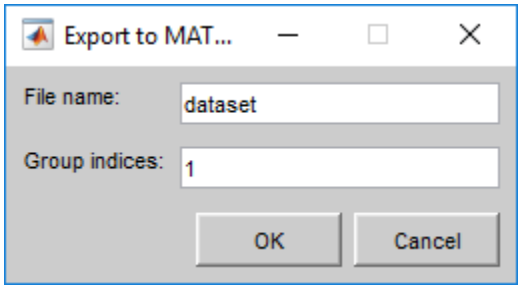
To change the time range, select **Change Time Range** from the Signal Builder **Axes** menu.

In the dialog box, edit the **Min time** and **Max time** fields as necessary to reflect the beginning and ending times of the new time range, respectively.

Exporting Signal Group Data

You can export data that defines Signal Builder block signals groups to a MAT-file or the MATLAB Workspace.

To export Signal Builder signal data, formatted as `Simulink.SimulationData.Dataset`, to a MAT-file, select **File > Export Data > To MAT-file**.



- **File name** — Enter a name for the MAT-file to contain the data.
- **Group indices** — Enter one or signal group numbers for which you want to export the data, specified as a scalar or vector. Numbers must correspond to an existing group in the block.

Alternatively, you can use the `signalbuilder get` function to return one or more data sets. For example:

```
[ds1 ds2]=signalbuilder(block,'get',[group1 group2])
```

To export signal data to the MATLAB workspace, select **File > Export Data > To Workspace**.

The Signal Builder exports the data by default to a workspace variable named `channels`. To export to a differently named variable, enter the variable name in the **Variable name** field. The Signal Builder exports the data to the workspace as the value of the specified variable.

The exported data is an array of structures. The structure `xData` and `yData` fields contain the coordinate points defining signals in the currently selected signal group.

To access all the data in the signal groups of a Signal Builder block, use the `signalbuilder get` function:

```
[time, data]=signalbuilder(block,'get',signal,group)
```

For example:

```
% For time 0 to 5, create three signal groups.
block = signalbuilder([], 'create', [0 5], {[2 2] [4 4] [7 8];[0 2] [0 4] [7 10]});
% Get the signals for all three groups.
[time, data]=signalbuilder(block,'get',[1 2],[1:3])

time =

  2×3 cell array

    [1×2 double]    [1×2 double]    [1×2 double]
    [1×2 double]    [1×2 double]    [1×2 double]


data =

  2×3 cell array

    [1×2 double]    [1×2 double]    [1×2 double]
    [1×2 double]    [1×2 double]    [1×2 double]
```

Simulating with Signal Groups

You can use standard simulation commands to run models containing Signal Builder blocks or you can use the **Run** or **Run all** button in the Signal Builder window (see “Running All Signal Groups” on page 75-85).

If you want to capture inputs and outputs that the **Run all** button generates, consider using the SystemTest™ software.

Activating a Signal Group

During a simulation, a Signal Builder block always outputs the active signal group. The active signal group is the group selected in the Signal Builder window for that block, if the dialog box is open. Otherwise, the active group is the group that was selected when the dialog box was last closed. To activate a group, open the group Signal Builder window and select the group.

Running Different Signal Groups in Succession

The Signal Builder toolbar includes the standard Simulink buttons for running a simulation. This facilitates running several different signal groups in succession. For example, you can open the dialog box, select a group, run a simulation, select another group, run a simulation, and so on, all from the Signal Builder window.

Running All Signal Groups

To run all the signal groups defined by a Signal Builder block, open the block dialog box and click the 

button from the Signal Builder toolbar. The **Run all** button runs a series of simulations, one for each signal group defined by the block. If you installed Simulink Coverage on your system and are using the Model Coverage Tool, the **Run all** button configures the tool to collect and save coverage data for each simulation in the MATLAB workspace and display a report of the combined coverage

results at the end of the last simulation. This allows you to quickly determine how well a set of signal groups tests your model.

Note To stop a series of simulations started by the **Run all** command, enter **Ctrl+C** at the MATLAB command line.

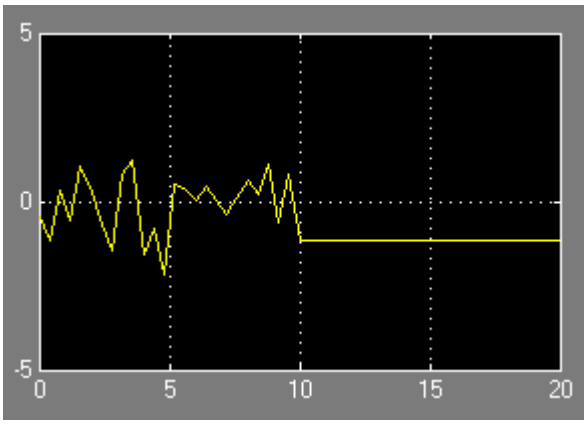
Simulation from Signal Builder Block

To control simulations from the Signal Builder block, select **File** > **Simulation Options**.

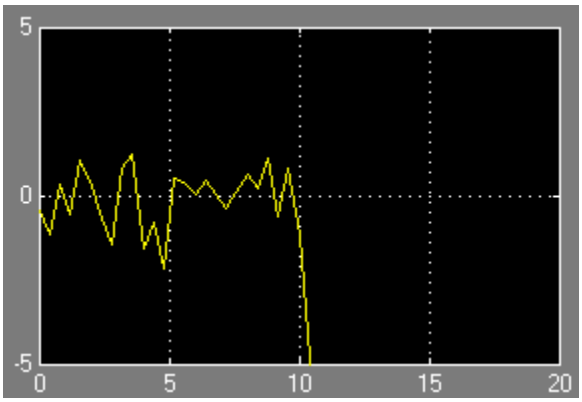
Signal values after final time

The setting of this control determines the output of the Signal Builder block if a simulation runs longer than the period defined by the block.

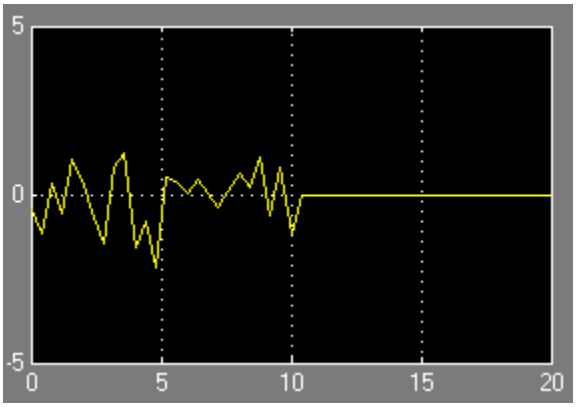
- To output the last defined value of each signal in the currently active group for the remainder of the simulation, select `Hold final value`. For example:



- To output values extrapolated from the last defined value of each signal in the currently active group for the remainder of the simulation, select `Extrapolate`. For example:

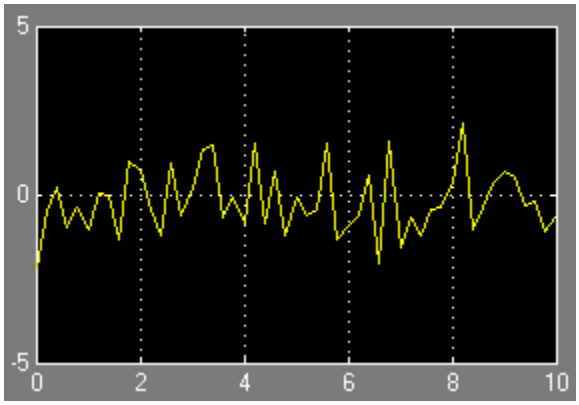


- To output zero for the remainder of the simulation, select `Set to zero`. For example:

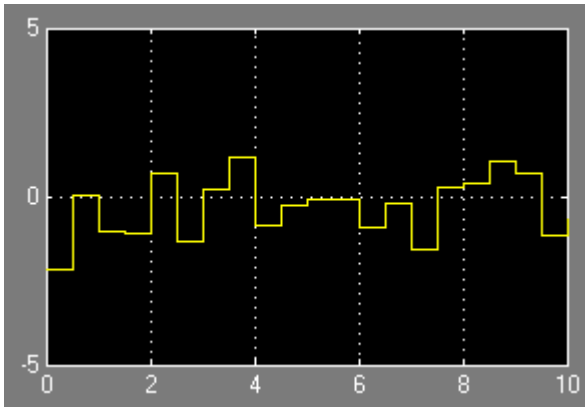


Sample time

To output a continuous signal, enter 0 in the **Sample time** parameter. For example, the following display shows the output of a Signal Builder block set to output a continuous Gaussian waveform over a period of 10 seconds.



To output a discrete signal, enter the sample time of the signal in this parameter. The following example shows the output of a Signal Builder block set to emit a discrete Gaussian waveform having a `0.5` second sample time.



Enable zero crossing

To have the Signal Builder block detect zero-crossing events, set **Enable zero crossing** `On` (default). The Signal Builder block sets the zero-crossing detection on the From Workspace block that you use to create the Signal Builder signal groups. For more information, see “Zero-Crossing Detection” on page 3-10.

See Also

Signal Builder | Signal Editor | `signalbuilder`

Related Examples

- “Load Data with Interchangeable Scenarios” on page 71-37
- “Export Simulation Data” on page 72-2
- “Initialize Signals and Discrete States” on page 75-37
- “Signal Basics” on page 75-2
- “Investigate Signal Values” on page 75-9

Using Composite Signals

Types of Composite Signals

To reduce visual complexity in a model, you can combine signals into composite signals. The signals that a composite signal contains are called elements. Elements retain their separate identities, which let you extract them from the composite signal.

You can access composite signal elements by name or index, depending on the composite signal type.

- Name-based composite signals allow for signal hierarchy. They are generically called buses.
- Index-based composite signals are flat, regardless of whether you create them in stages. They require that all input signals have the same data type.

When you group signals into a composite signal, you can decide whether they affect simulation and code generation.

- A virtual composite signal simplifies the visual appearance of a model by combining two or more signal lines into one line. It does not group the signals in any functional sense and, therefore, does not affect simulation or code generation. By accessing elements directly, virtual composite signals execute faster than nonvirtual composite signals in simulations and generated code.
- A nonvirtual composite signal visually and functionally groups signals, affecting both simulation and code generation.

Models can use a combination of these composite signal types.

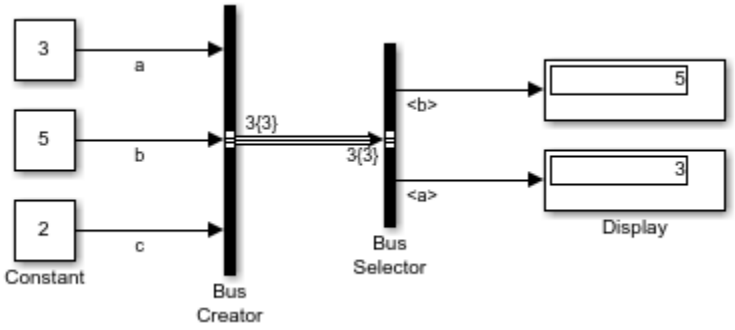
Composite Signal Feature	Name-Based Access	Index-Based Access
Visual Grouping	“Virtual Bus” on page 76-2	“Mux Signal” on page 76-6
Functional Grouping	“Nonvirtual Bus” on page 76-3	“Concatenated Signal” on page 76-5

You can identify composite signal types by their line style after compiling or simulating the model.

Line Style	Composite Signal Type
	Virtual bus
	Nonvirtual bus
	Nonscalar signal when the **Nonscalar Signals** information overlay is enabled (includes index-based composite signals)
	Index-based composite signal that contains nonvirtual buses

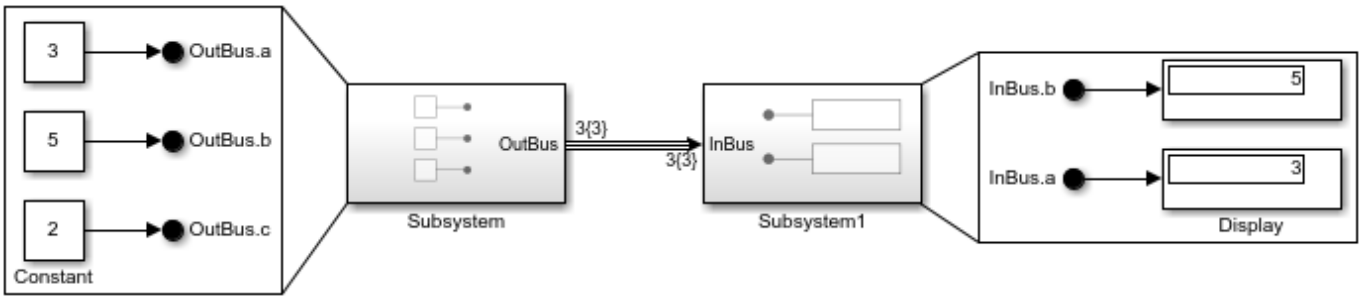
Virtual Bus

This model shows a virtual bus that contains signals `a`, `b`, and `c`.



Bus Creator blocks create buses within a subsystem or model. Bus Selector blocks extract specified elements of the bus.

This model shows an equivalent virtual bus passing through a subsystem boundary.



Out Bus Element blocks create a bus at a subsystem or model interface. In Bus Element blocks extract specified elements of a bus at a subsystem or model interface.

You can use virtual buses to:

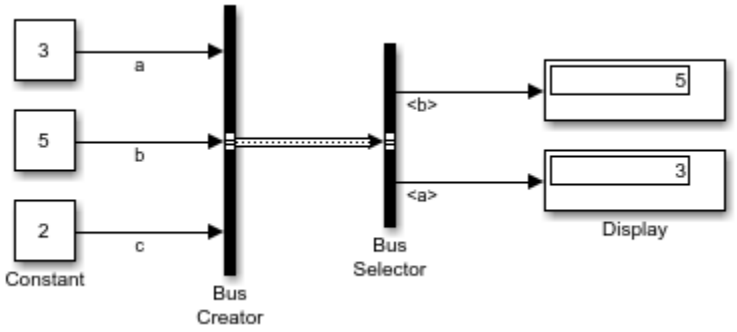
- Contain bus elements that have different sample rates.
- Cross model reference boundaries.

To specify and validate the properties of a virtual bus, you can specify a `Simulink.Bus` object.

To create a virtual bus, see “Group Signal Lines into Virtual Buses” on page 76-8.

Nonvirtual Bus

This model shows a nonvirtual bus that contains signals `a`, `b`, and `c`.



Bus Creator blocks create buses within a subsystem or model. Bus Selector blocks extract specified elements of the bus.

You can use nonvirtual buses to:

- Package bus data as structures in the generated C code.
- Construct an array of buses.
- Interface with external code through an S-function.
- Have bus data cross MATLAB Function block or Stateflow chart boundaries.
- Display and log buses with a Scope block.

All elements of a nonvirtual bus must use 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.

A `Simulink.Bus` object must define the bus you want to make nonvirtual. A bus becomes nonvirtual when you select the **Output as nonvirtual bus** parameter. Selecting this parameter causes simulation and code generation to apply the structure defined by the `Bus` object. When this parameter is cleared, the `Bus` object only validates the properties of the bus.

The type of bus can make a significant difference in the efficiency, size, and readability of the generated code. For a bus to appear in the generated code, it must be nonvirtual. Only the elements of a virtual bus appear in the generated code.

For example, suppose a bus passes through a Unit Delay block. For simplicity, the bus contains only three elements: `a`, `b`, and `c`. This table shows the effect of the **Output as nonvirtual bus** parameter on the generated code.

Generated Code	Virtual Bus	Nonvirtual Bus

model`_types.h` file	Virtual buses do not require type definitions.	`Bus` objects appear in the generated code as structures.
		
		`typedef struct {`
		`  real_T a;`
		`  real_T b;`
		`  real_T c;`
		`} BusObject;`

Generated Code	Virtual Bus	Nonvirtual Bus

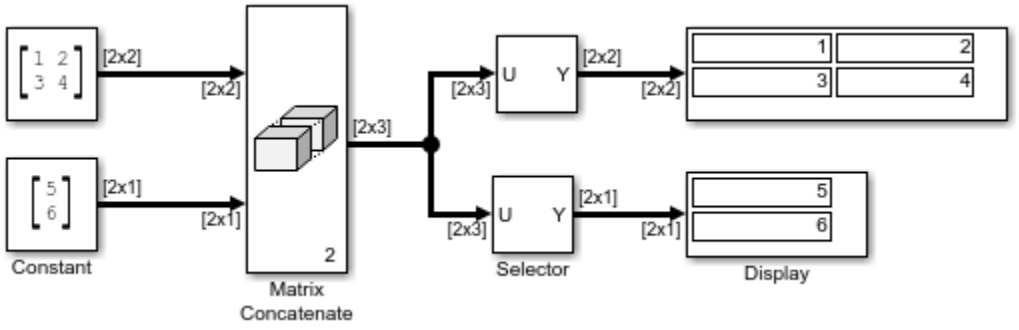
model.h file	The generated code defines one Unit Delay block for each element of the virtual bus.
	
	`typedef struct {`
	`  real_T UnitDelay_1_DSTATE;  /* '<Root>/Unit Delay' */`
	`  real_T UnitDelay_2_DSTATE;  /* '<Root>/Unit Delay' */`
	`  real_T UnitDelay_3_DSTATE;  /* '<Root>/Unit Delay' */`
	`} DW_model_T;`
		The generated code defines one Unit Delay block for the nonvirtual bus, using the `BusObject` structure.
		
		`typedef struct {`
		`  BusObject UnitDelay_DSTATE;  /* '<Root>/Unit Delay' */`
		`} DW_model_T;`

To create a nonvirtual bus, see “Create Nonvirtual Buses” on page 76-19.

If you intend to generate code for a model that uses buses, see “Generate Efficient Code for Bus Signals” (Simulink Coder). Generating code for nonvirtual buses can result in multiple copies of some buses.

Concatenated Signal

This model shows a concatenated signal that places the input matrices side by side.



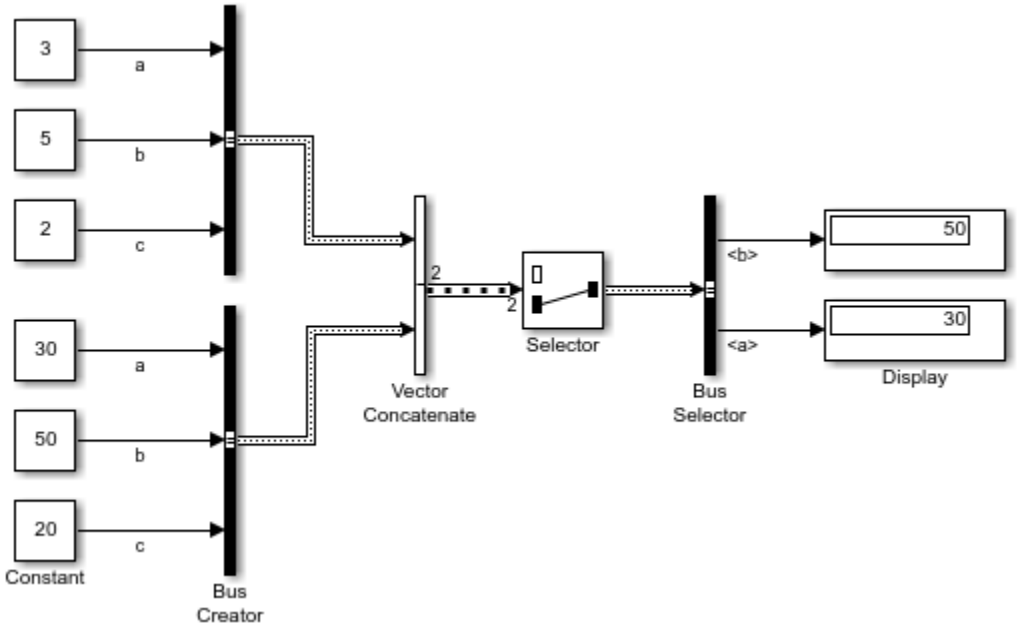
The Matrix Concatenate block creates concatenated signals. Elements can be either vectors or matrices, depending on how you configure this block. The Selector block extracts signals based on specified indices. The extracted signals can be grouped differently than the input signals.

You can use concatenated signals in mathematical operations.

To group signals with a Vector Concatenate or Matrix Concatenate block, the signals must have the same data type. When the data type is a `Bus` object, the inputs must be nonvirtual buses.

Concatenated nonvirtual buses are also known as an array of buses. In an array of buses, all elements are nonvirtual buses that use the same `Bus` object to specify properties. An array of buses is equivalent to an array of structures in MATLAB. You can use an array of buses to model a multichannel system. While all the channels have the same properties, each of the channels may have a different value.

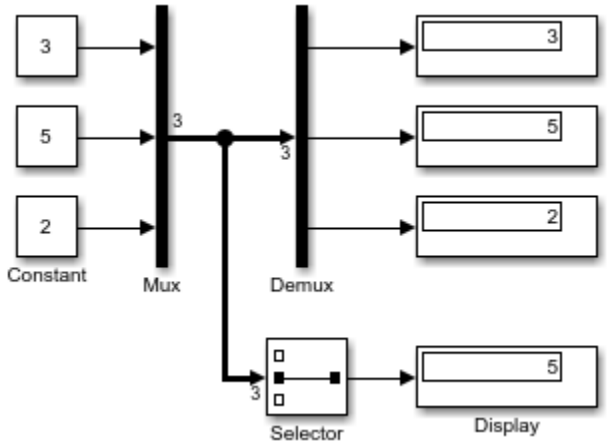
In this model, a Vector Concatenate block creates an array of buses.



For more information on arrays of buses, see “Combine Buses into an Array of Buses” on page 76-64.

Mux Signal

This model shows a mux signal that places the three input signals side by side.



The Mux block creates mux signals. The Demux block extracts all signals, which may be grouped differently than the input signals. The Selector block extracts signals based on specified indices. The extracted signals can be grouped differently than the input signals.

You can use a mux signal to perform computations on multiple vectors. You can also use a Mux block to create a vector of function calls.

Input signals for a Mux block can be any combination of scalars, vectors, and mux signals, but they must have the same data type and numeric type. The signals in the output mux signal appear in the

same order as the input signals for the Mux block. You can use multiple Mux blocks to create a mux signal in stages, but the result is flat as if you used a single Mux block.

See Also

Related Examples

- “Group Signal Lines into Virtual Buses” on page 76-8
- “Display Bus Information” on page 76-31
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Bus-Capable Blocks” on page 76-36
- “Signal Types” on page 75-7

Group Signal Lines into Virtual Buses

You can combine signals into a bus and then access the bus as a whole or select specific signals from the bus. A virtual Simulink bus is analogous to a bundle of wires held together by tie wraps. For comparison, a nonvirtual Simulink bus is analogous to a struct in C code.

Not all blocks can accept buses and some blocks implicitly convert buses to vectors. To learn which blocks support which types of buses, see “Bus-Capable Blocks” on page 76-36. To identify bus conversions, see “Manage Bus-to-Vector Conversions”.

How you create virtual buses differs based on the location of the signals that you want to group.

- “Group Signal Lines Within a Component” on page 76-8
- “Connect Multiple Output Signals to a Port” on page 76-10
- “Combine Multiple Subsystem Ports into One Port” on page 76-14

To focus on fundamental steps, these examples are simple, however, buses are most useful when you have many signals to combine.

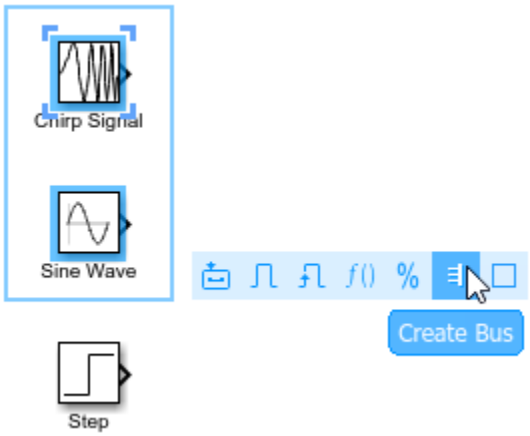
Tip When you create a bus, the line style updates when you simulate the model or, on the **Modeling** tab, click **Update Model**.

Group Signal Lines Within a Component

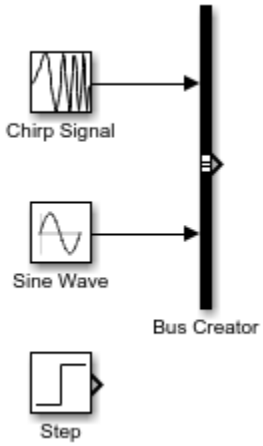
This example shows how to group signals into a virtual bus using Bus Creator blocks.

Open the example model, which contains three blocks.

To create a bus that contains the signals from multiple blocks, drag to select the blocks. For this example, select the Chirp Signal and Sine Wave blocks. In the action bar that appears, click **Create Bus**.

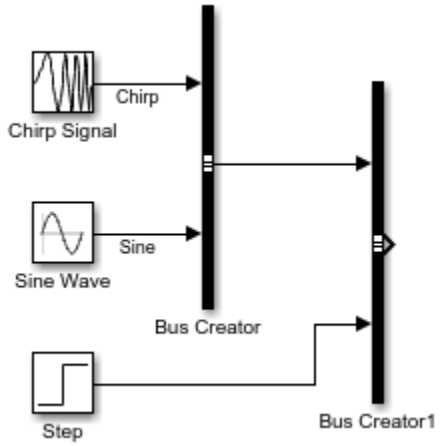


Simulink adds a Bus Creator block and connects the input signals to that block. The output of the Bus Creator block is a virtual bus.



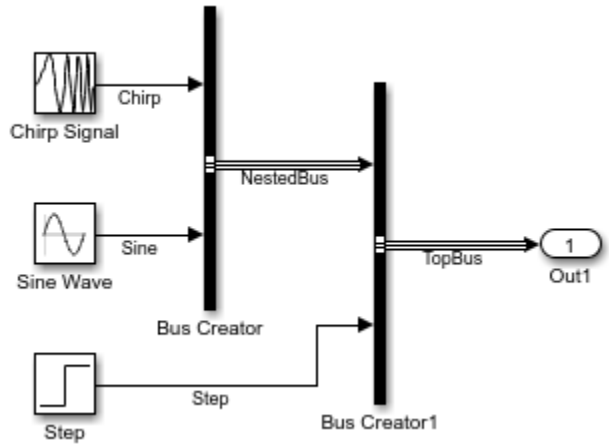
To make it easier to identify the elements of the bus, label the input signals to the Bus Creator block. Label the output signal of the Chirp Signal block by double-clicking the signal line and entering `Chirp`. Similarly, label the output signal of the Sine Wave block `Sine`.

To create a second bus that contains the first bus and the output signal of the Step block, drag to select the Bus Creator and Step blocks. In the action bar that appears, click **Create Bus**. Since the `Sine` and `Chirp` signals are elements of the input bus, Simulink creates the same bus regardless of whether your selection includes the Sine Wave and Chirp Signal blocks.



Simulink adds another Bus Creator block and connects the input signals to that block. Label the output signal of the Step block `Step` and the output signal of the first Bus Creator block `NestedBus`. You can nest buses to any depth. If one of the inputs to a Bus Creator block is a bus, then its output is a bus hierarchy that contains at least one nested bus.

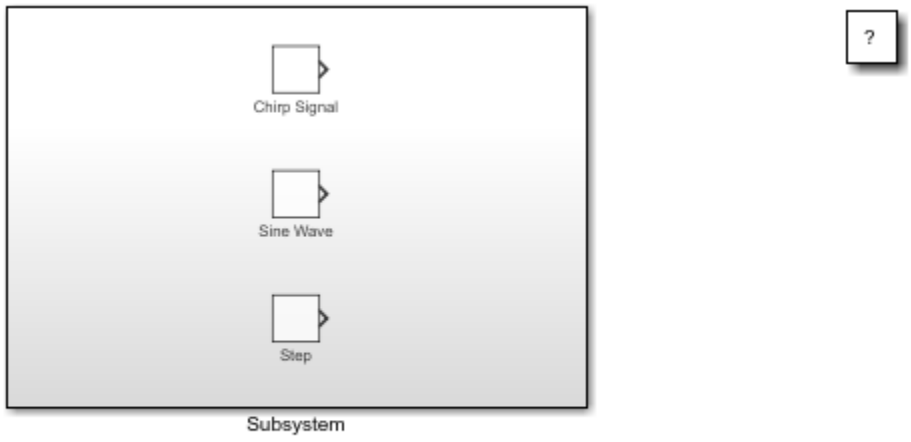
Connect the output signal of the second Bus Creator block to an Outport block and label the signal `TopBus`. Now that all signal lines are connected, simulate the model.



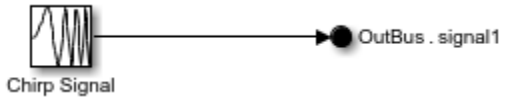
Connect Multiple Output Signals to a Port

This example shows how to combine the output signals of a subsystem or model into a virtual bus using Out Bus Element blocks.

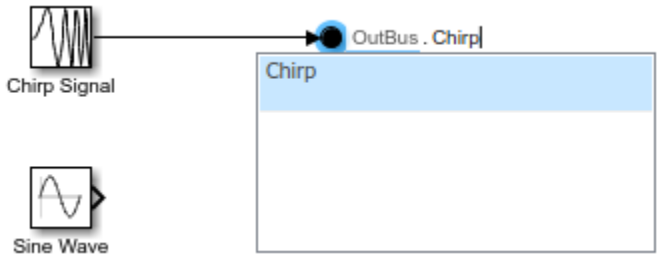
Open the example model, which contains a subsystem with three source blocks.



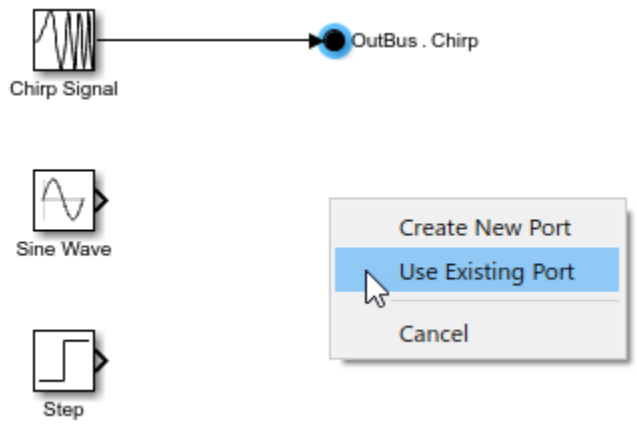
Open the subsystem and add an Out Bus Element block to it. An Out Bus Element block is similar to a Bus Creator block connected to an Outport block. Connect the output of the Chirp Signal block to the Out Bus Element block.



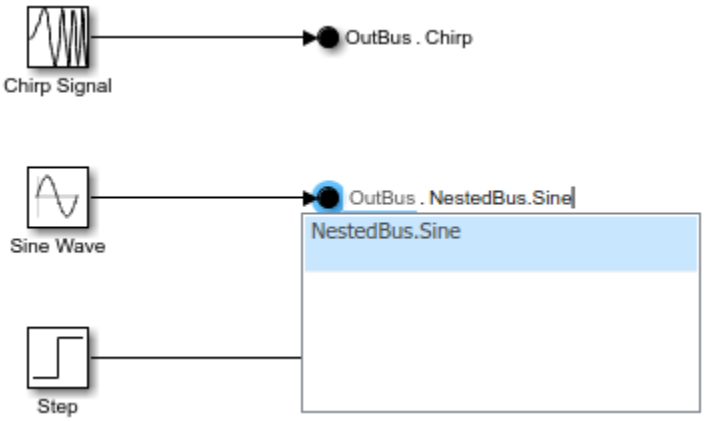
The label next to the Out Bus Element block has two parts. The first part of the label describes the bus (`OutBus`) and the second part of the label describes the bus element (`signal1`). To make identifying elements of the bus easier, rename the element by double-clicking `signal1` and entering `Chirp`.



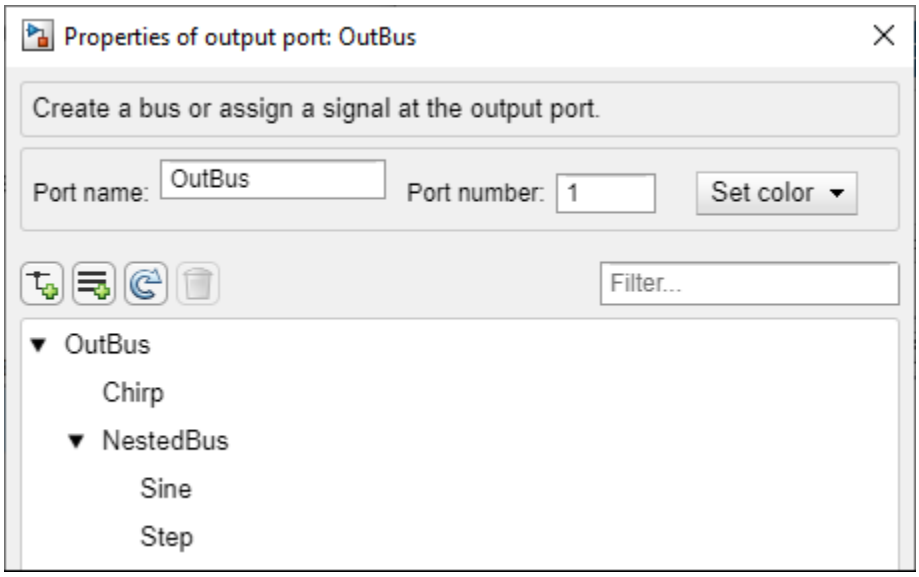
Use **Ctrl**+drag on the Out Bus Element block to make two copies of the block. When copying the block this way, you must specify whether to create a new port or use the existing port. To create one output bus that contains all of the signals, choose **Use Existing Port** each time you copy the block, then connect the signals.



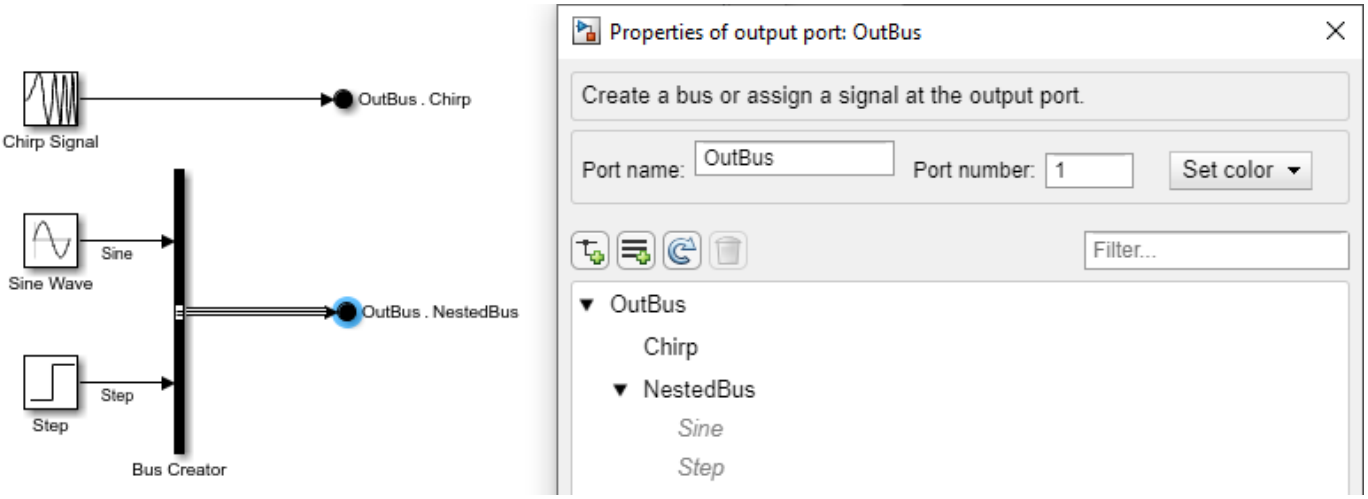
To create hierarchy in the output bus, use dots in the part of the label that describes the bus element. Each dot creates a new level of hierarchy. Create a nested bus named `NestedBus` by defining the bus elements as `NestedBus.Sine` and `NestedBus.Step` respectively.



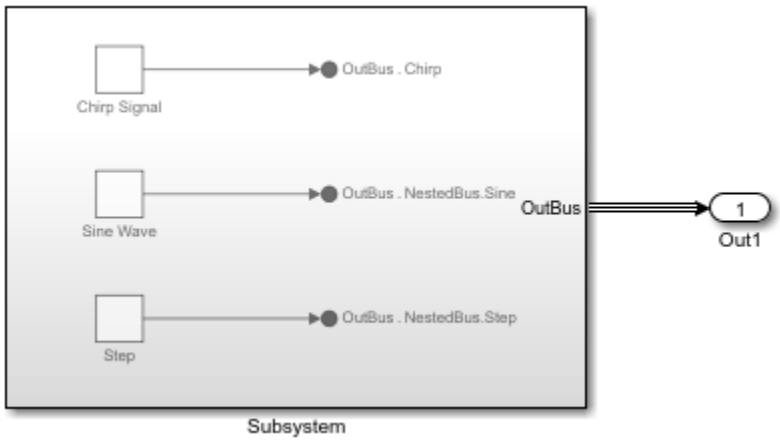
Inspect the hierarchy in the block parameters dialog box for the port by double-clicking on any of the Out Bus Element blocks.



Alternatively, if you grouped the `Sine` and `Step` signals into a virtual bus using a Bus Creator block, you could connect that bus to an Out Bus Element block. The label specifies the name of the element connected to the block, which is the `NestedBus` virtual bus.



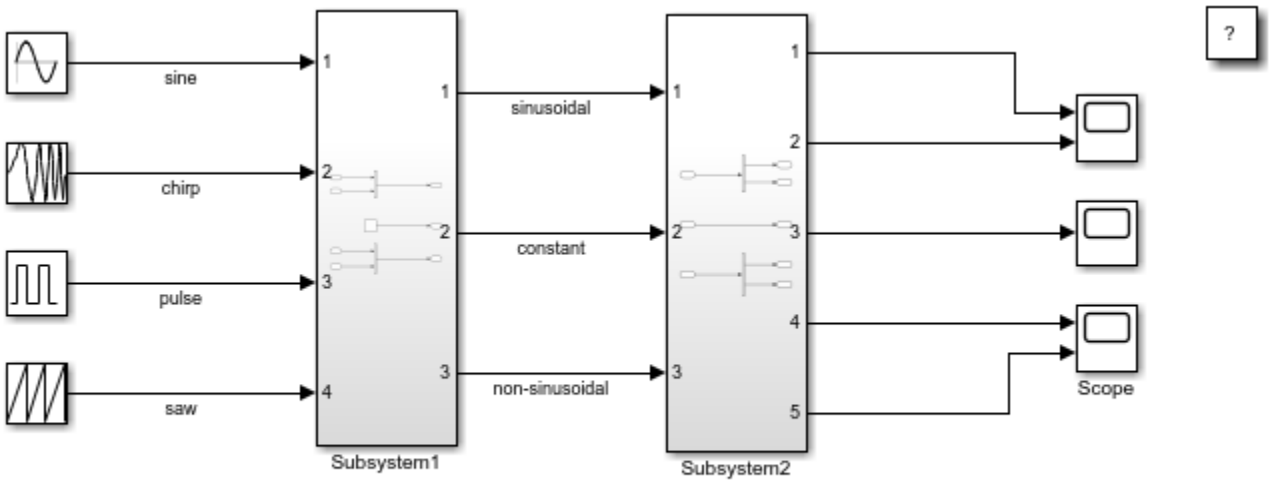
Return to the top model. The three Out Bus Element blocks correspond with one port. Connect the output of this port to an Outport block and simulate the model.



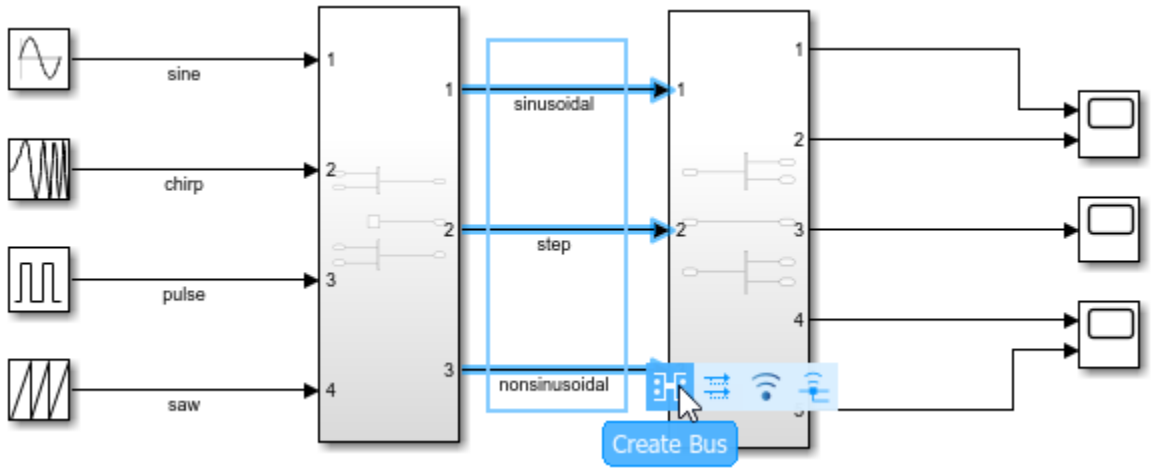
Combine Multiple Subsystem Ports into One Port

This example shows three ways to simplify a subsystem interface by converting multiple ports and their connected signals into one port and a bus.

Open the example model, which contains two subsystems with multiple input and output ports.

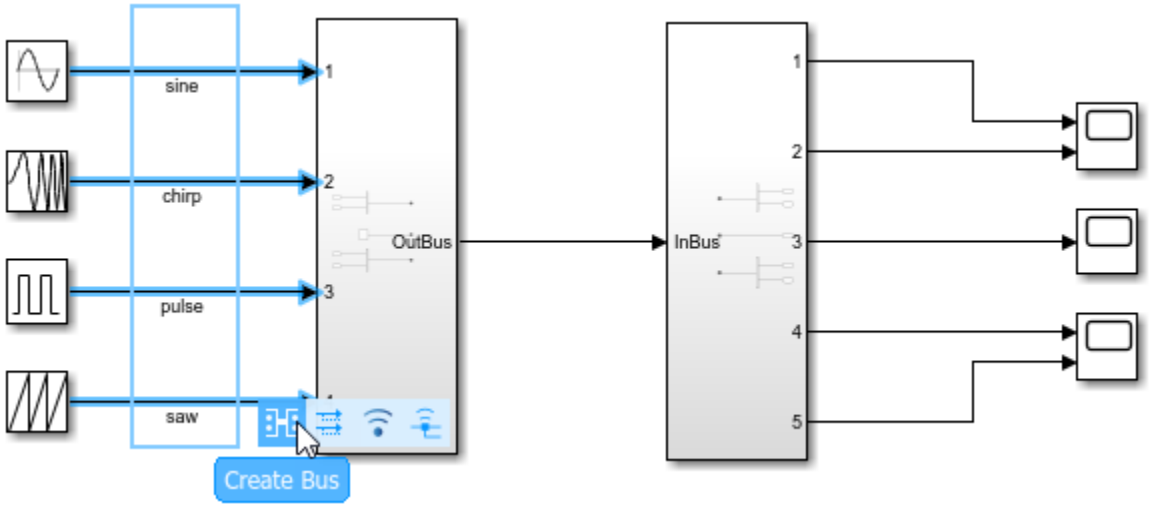


Drag a selection box around the signal lines between the two subsystems. From the action bar that appears, click **Create Bus**.



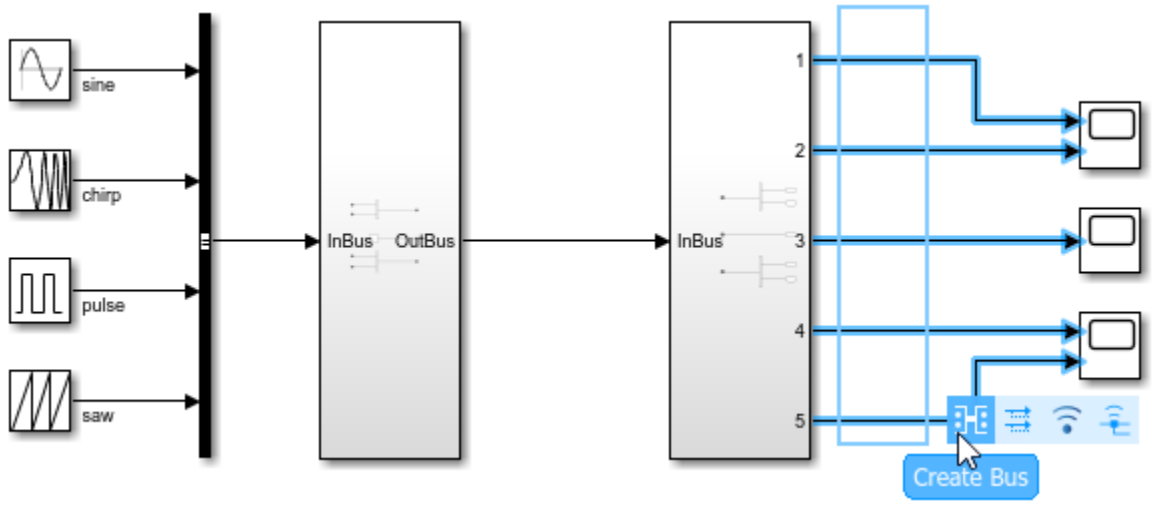
Simulink replaces the Inport and Outport blocks in the source and destination subsystems with In Bus Element and Out Bus Element blocks.

Drag a selection box around the signal lines between the source blocks and first subsystem. From the action bar that appears, click **Create Bus**.



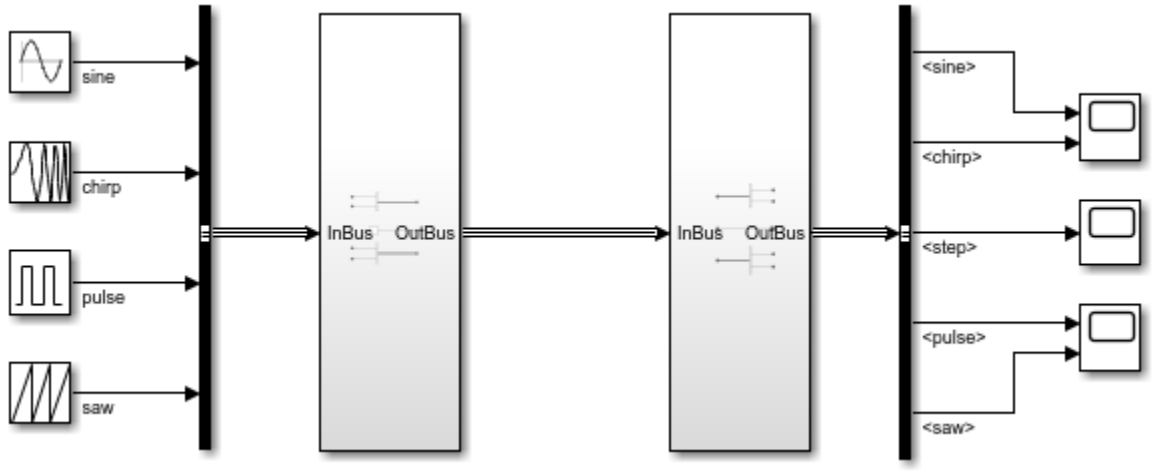
Simulink adds a Bus Creator block before the first subsystem and replaces the Inport blocks in the first subsystem with In Bus Element blocks.

Drag a selection box around the signal lines between the second subsystem and Scope blocks. From the action bar that appears, click **Create Bus**.



Simulink replaces the Outport blocks in the second subsystem with Out Bus Element blocks and adds a Bus Selector block after the second subsystem.

The resulting model uses virtual buses at the subsystem interfaces.



See Also

Bus Creator | Bus Selector | In Bus Element | Out Bus Element

Related Examples

- “Types of Composite Signals” on page 76-2
- “Bus-Capable Blocks” on page 76-36
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Specify Initial Conditions for Bus Signals” on page 76-57

Modify Bus Hierarchy

To view the hierarchy of a bus, right-click on the associated signal line and click **Signal Hierarchy** in the context menu. You can also see the signal hierarchy in the block parameter dialog box for the Bus Creator, Bus Assignment, Bus Selector, In Bus Element, and Out Bus Element blocks.

To change the hierarchy of elements in a bus, you can:

- Separate a bus into its constituent signals with a Bus Selector block, then reassemble the signals with one or more Bus Creator blocks.
- For an Out Bus Element block, edit the second part of the label, which corresponds to the bus element. Suppose you have a bus element named`signal1` and you want to move `signal1` into a nested bus named `NestedBus`. You would change the part of the label that corresponds to the bus element from `signal1` to `NestedBus.signal1`.
- In the Out Bus Element block parameter dialog box, click and drag signals to different hierarchy levels.

To change the order of elements in a bus:

- In the Bus Creator block parameter dialog box, select one signal or adjacent signals, then click the **Up** or **Down** button.
- In the Out Bus Element block parameter dialog box, click and drag signals within their existing hierarchy level.

When you change the bus hierarchy, Simulink automatically handles most of the complexities involved. For example, Simulink repairs broken selections in the Bus Selector and Bus Assignment block parameter dialog boxes due to upstream bus hierarchy changes. By default, the related **Repair bus selections** configuration parameter is set to `Warn and repair`. The repairs occur when you update a model.

Tip To change the value of elements in a bus, use a Bus Assignment block. For details, see “Replace Values of Bus Elements” on page 76-38.

Resolve Circular Dependencies in Buses

Nesting buses can produce a loop of blocks where a bus is an element of itself. This circular definition cannot be resolved and therefore causes an error. To trace the loop, you can use the location cited in the error message.

1. Select a signal line associated with the location cited in the error message.
2. Right-click a signal and choose **Highlight Signal to Source** or **Highlight Signal to Destination**. See “Highlight Signal Sources and Destinations” on page 75-25 for more information.
3. Continue choosing signals and highlighting their sources and destinations until the loop becomes clear.
4. Restructure the model to eliminate the circular bus definition.

Because the problem is a circular definition, rather than a circular computation, the cycle cannot be broken by inserting additional blocks. For example, you cannot fix a circular definition the way that

you break an algebraic loop by inserting a Unit Delay block. You must restructure the model to eliminate the circular bus definition.

See Also

Bus Creator | Bus Selector | Out Bus Element

Related Examples

- “Types of Composite Signals” on page 76-2
- “Bus-Capable Blocks” on page 76-36
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Specify Initial Conditions for Bus Signals” on page 76-57

Create Nonvirtual Buses

A nonvirtual Simulink bus is analogous to a struct in C code. To package bus data as structures in generated C code, use nonvirtual buses.

You must also use nonvirtual buses to:

- Construct an array of buses.
- Interface with external code through an S-function.
- Have bus data cross MATLAB Function block or Stateflow chart boundaries.
- Display and log buses with a Scope block.

To create a nonvirtual bus, you must:

- Specify a `Simulink.Bus` object as the data type for the bus. Set **Data type** to `Bus: <object name>`, where `<object name>` is the `Bus` object name.
- Specify that the output of the block is a nonvirtual bus. Select **Output as nonvirtual bus** or, for Outport blocks, **Output as nonvirtual bus in parent model**.

All signals in a nonvirtual bus must have the same sample time. For more information, see “Modify Sample Times for Nonvirtual Buses” on page 76-42.

To simulate a model that contains nonvirtual buses, the referenced `Bus` objects must be in the base workspace or a data dictionary. You must define the `Bus` object or use an already defined `Bus` object. A model callback can load the necessary `Bus` objects.

The way to create nonvirtual buses differs based on the location of the bus:

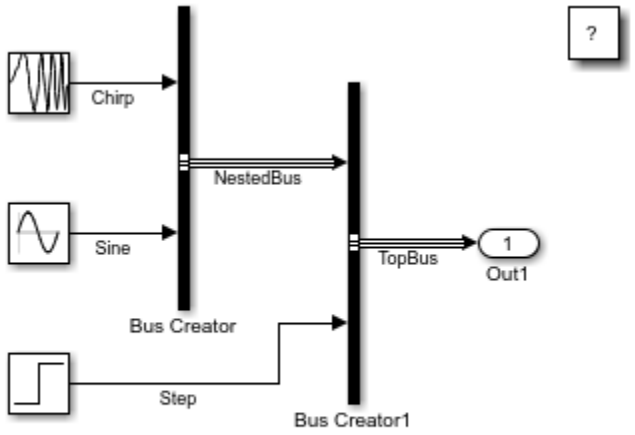
- “Create Nonvirtual Buses with Bus Creator Blocks” on page 76-19
- “Create Nonvirtual Bus Output for Referenced Models” on page 76-21
- “Convert Virtual Bus to Nonvirtual Bus” on page 76-22

To focus on fundamental steps, these examples are simple. However, buses are most useful when you have many signals to combine.

Create Nonvirtual Buses with Bus Creator Blocks

This example shows how to configure a Bus Creator block to output a nonvirtual bus.

Open and simulate the example model, which contains a virtual bus hierarchy.



Create Simulink.Bus Objects

Since the virtual buses in this model are not defined by `Bus` objects, you must create `Bus` objects that match the bus hierarchy. If the virtual buses were defined by `Bus` objects, you would not need to create `Bus` objects.

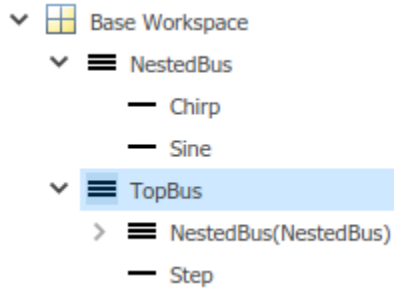
To create and save `Bus` objects for the buses in the model, use the `Simulink.Bus.createObject` function. When you specify a Bus Creator block that creates a bus that contains other buses, this function creates `Bus` objects for the bus created by the block and all nested buses. When you specify a file name, it saves the `Bus` objects in a function with that name.

```
busInfo = Simulink.Bus.createObject...
  ('NonvirtualBusCreationModel',...
  'NonvirtualBusCreationModel/Bus Creator1',...
  'NonvirtualBusCreationFunction');
```

In the base workspace, Simulink creates two `Bus` objects named after the corresponding buses, `TopBus` and `NestedBus`. In the current folder, Simulink creates a function named `NonvirtualBusCreationFunction.m`.

To see the created `Bus` objects, open the Bus Editor by entering:

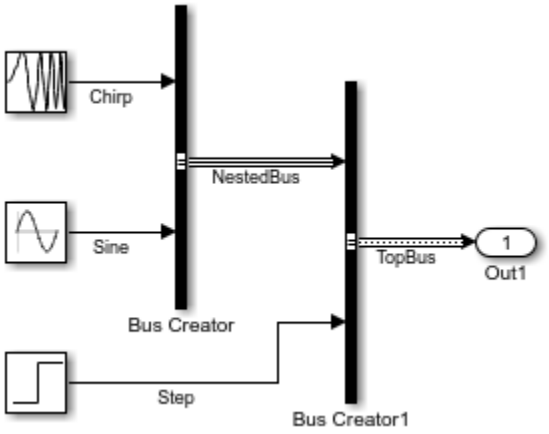
```
buseditor
```



Specify Nonvirtual Bus Outputs

In the model, double-click the Bus Creator1 block. In the dialog box, set the **Output data type** to `Bus: TopBus` and select the **Output as nonvirtual bus** check box.

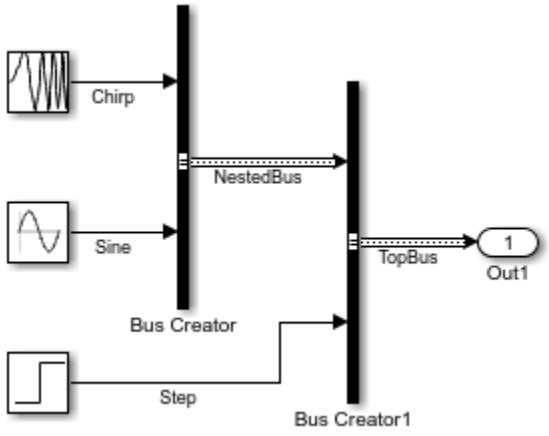
To identify the nonvirtual bus by line style, simulate the model.



The Bus Creator1 block output is a nonvirtual bus, while the Bus Creator block output remains a virtual bus.

Double-click the Bus Creator block. In the dialog box, set **Output data type** to `Bus: NestedBus` and select the **Output as nonvirtual bus** check box.

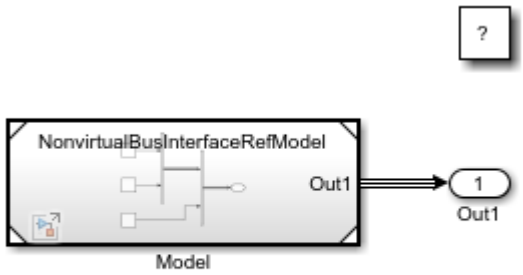
To identify the new nonvirtual bus by line style, simulate the model.



Create Nonvirtual Bus Output for Referenced Models

This example shows how to convert a virtual bus to a nonvirtual bus at the output port of a referenced model.

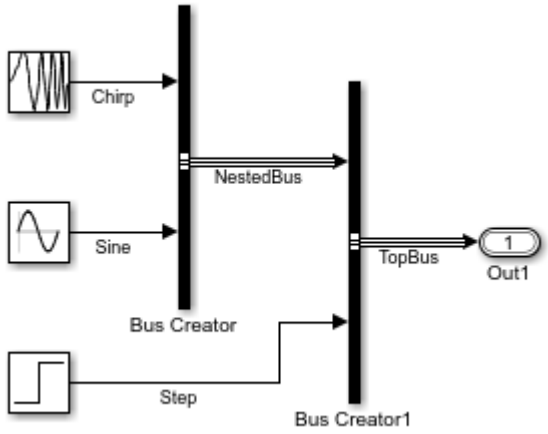
Open and simulate the example model, which references a model with a virtual bus output.



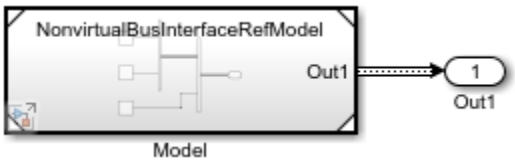
To define the model interface, the output port of the referenced model has a `Simulink.Bus` object as its data type. To create the `Bus` object in the base workspace when the referenced model is loaded, the referenced model uses a callback.

In the referenced model, double-click the Outport block. In the dialog box, on the **Signal Attributes** tab, select **Output as nonvirtual bus in parent model**, then click **OK**.

To update line styles, simulate the model again.



The input to the Outport block remains a virtual bus.

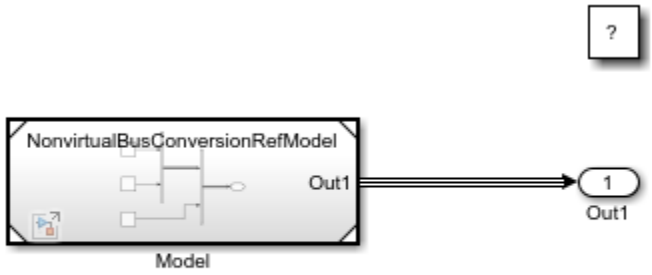


The output of the Model block is now a nonvirtual bus.

Convert Virtual Bus to Nonvirtual Bus

This example shows how to convert a virtual bus to a nonvirtual bus within a model.

Open and simulate the example model, which references a model with a virtual bus output.



To define the model interface, the output port of the referenced model has a `Simulink.Bus` object as its data type. To create the `Bus` object in the base workspace when the referenced model is loaded, the referenced model uses a callback.

To convert the virtual bus output to a nonvirtual bus, add a To Nonvirtual Bus block between the Model block and Outport block. This block is a preconfigured version of the Signal Conversion block that has the **Output** parameter set to `Nonvirtual bus`.

To identify the nonvirtual bus by line style, simulate the model.

See Also

Blocks
Bus Creator

Objects
`Simulink.Bus`

Related Examples

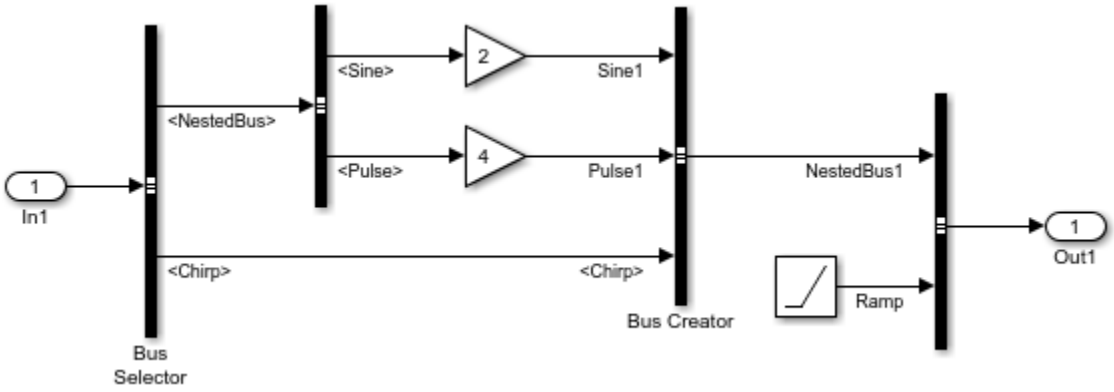
- “Types of Composite Signals” on page 76-2
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Inspect Generated Code for Nonvirtual Buses” on page 76-108
- “Create Structures in MATLAB Function Blocks” on page 44-63

Simplify Subsystem and Model Interfaces with Buses

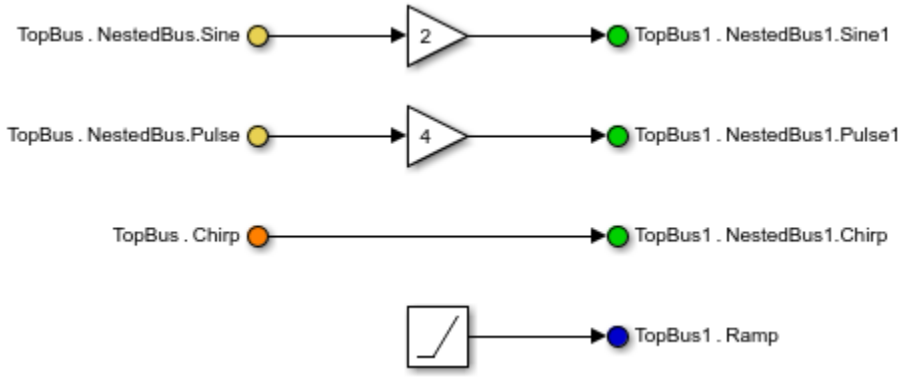
Buses allow you to simplify subsystem and model interfaces with In Bus Element and Out Bus Element blocks. These blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to elements closer to their point of usage.

You can use these blocks instead of Inport and Bus Selector blocks for inputs, and Outport and Bus Creator blocks for outputs. For example, this model uses Inport, Bus Selector, Bus Creator, and Outport blocks.



This equivalent model uses In Bus Element and Out Bus Element blocks.



To support In Bus Element and Out Bus Element blocks, parent subsystems and models must:

- Convert nonvirtual buses to virtual buses before the related input ports.
- Support virtual bus output from the related output ports.

Consider using In Bus Element and Out Bus Element blocks for models with buses that you anticipate changing frequently during the model development process.

Tip Simulink can help you update a subsystem interface to use In Bus Element and Out Bus Element blocks. From the action bar, you can:

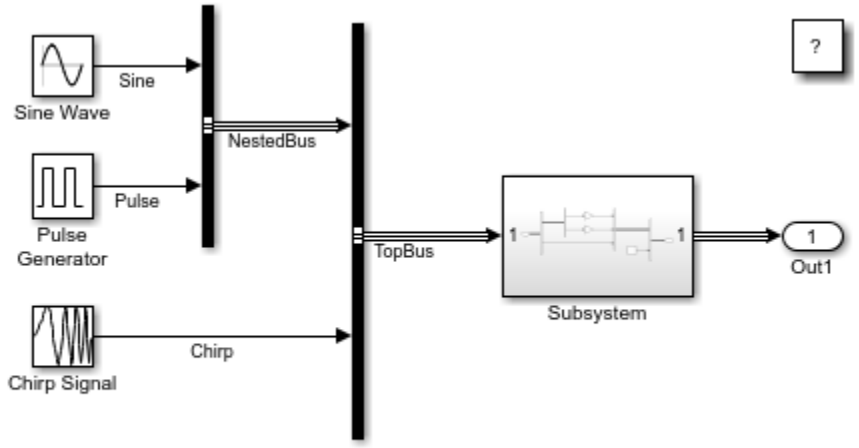
These actions are not available for model interfaces or for signal lines and blocks that have extra specifications, including signal names and logging. Extra specifications create potential conflicts.

The following examples illustrate how to use the action bar to update subsystem interfaces. The example models are simple, however, buses are most useful when you have many signals to combine.

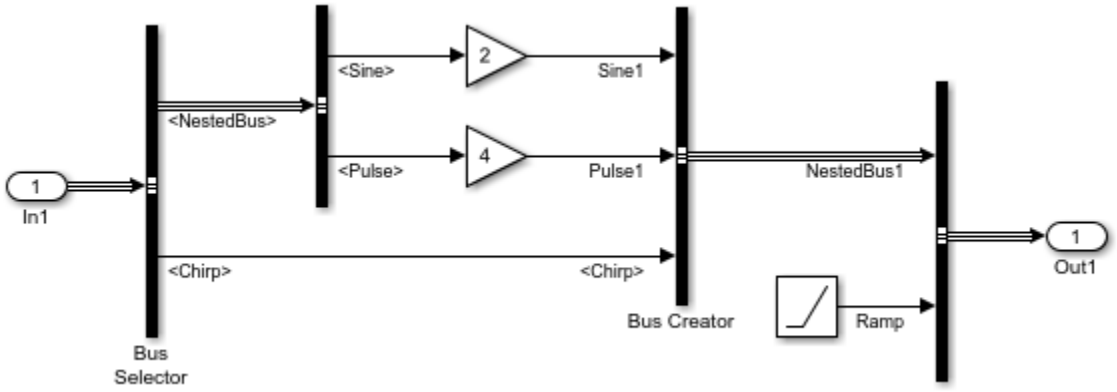
Simplify Bus Interfaces in Subsystems

This example shows how to convert an interface that uses Inport, Bus Selector, Bus Creator, and Outport blocks to use In Bus Element and Out Bus Element blocks.

Open and simulate the example model, which contains a subsystem that modifies an input bus hierarchy using Bus Selector and Bus Creator blocks. The subsystem uses Inport and Outport blocks for input and output.

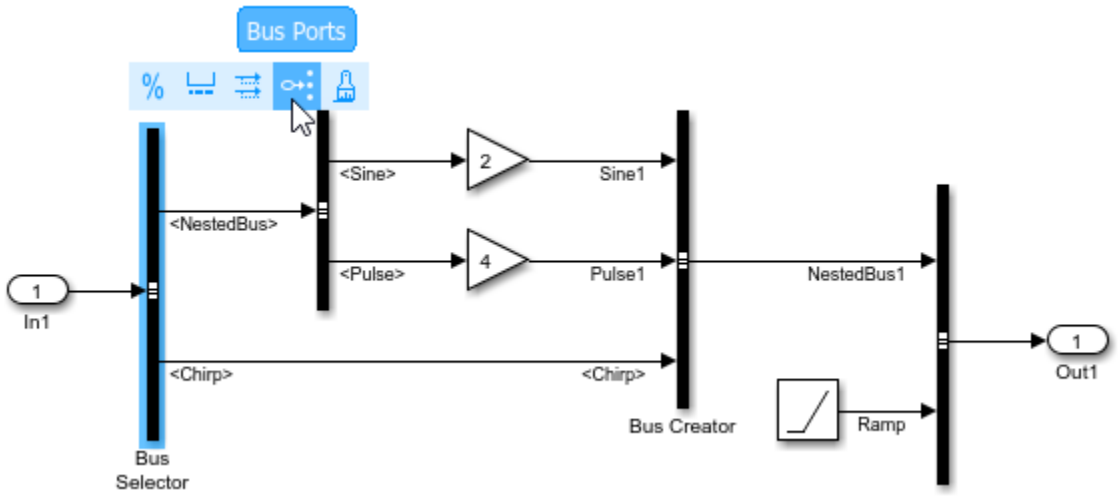


Open the subsystem.

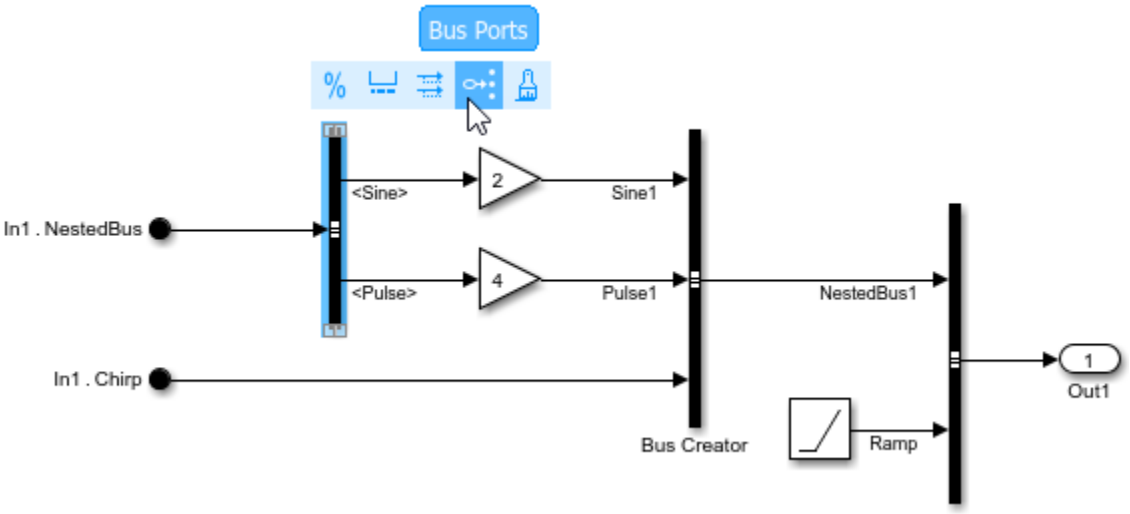


To convert Inport and Bus Selector blocks to In Bus Element blocks:

1. Click a Bus Selector block that directly connects to an Inport block.
2. In the action bar that appears when you pause over the ellipsis, click **Bus Ports**.

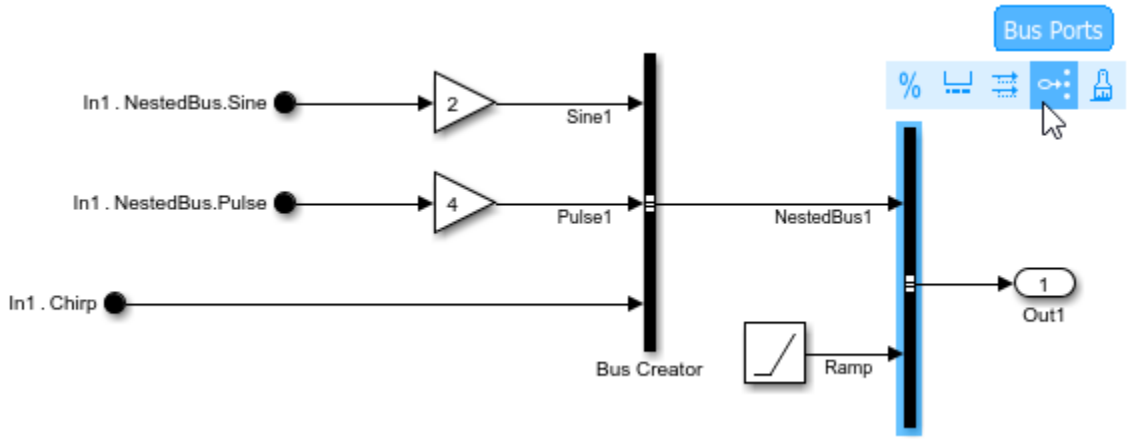


You can similarly convert an In Bus Element and Bus Selector block.

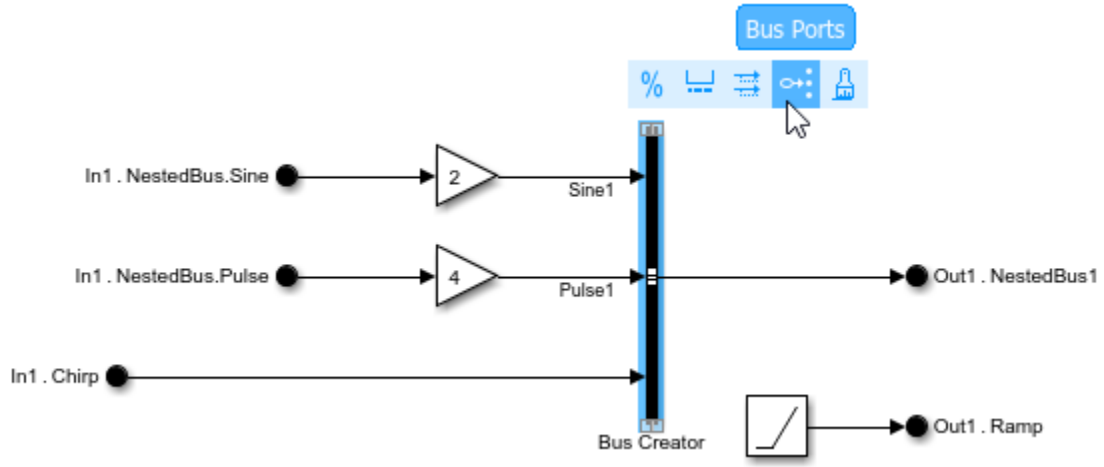


To convert Outport and Bus Creator blocks to Out Bus Element blocks:

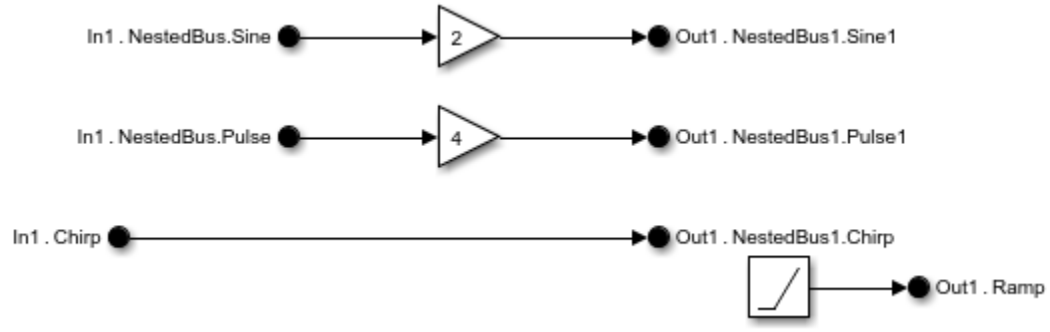
1. Click a Bus Creator block that directly connects to an Outport block without branching.
2. In the action bar that appears when you pause over the ellipsis, click **Bus Ports**.



You can similarly convert Out Bus Element and Bus Creator blocks.



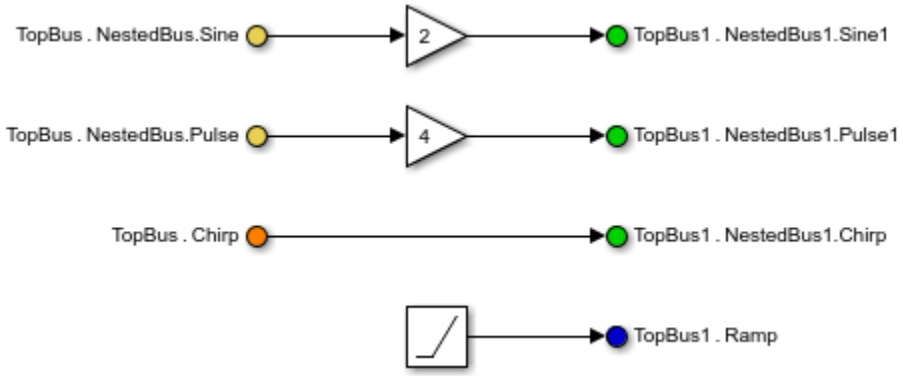
The resulting model simplifies line routing, makes it easier to incrementally change the interface, and lets you access elements closer to their point of usage.



You can change the name of a bus and its elements by double-clicking the block labels and editing them.

To easily identify elements of the same nested bus or bus port, specify block colors.

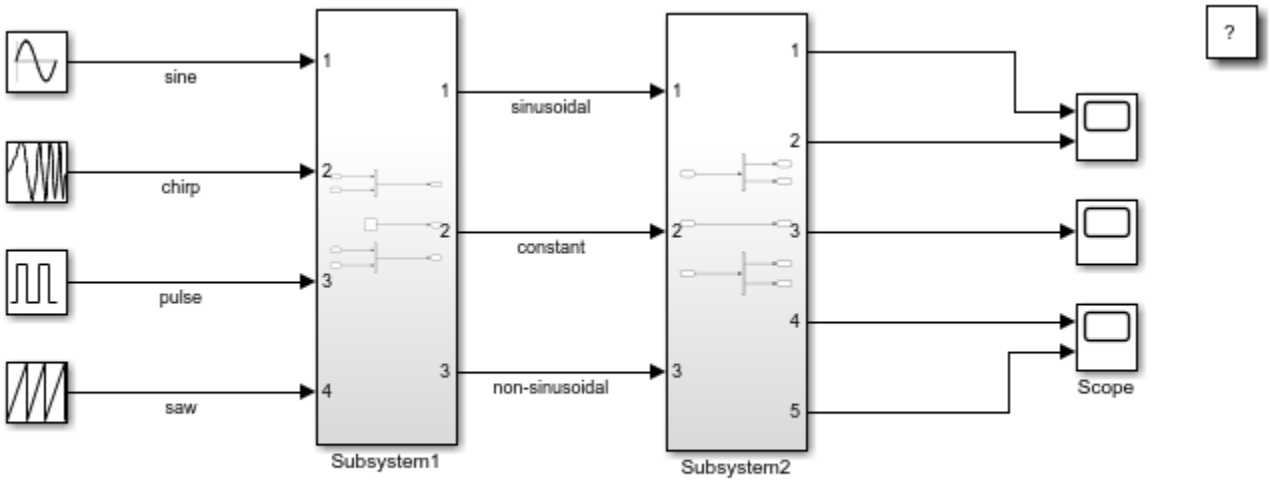
1. Double-click an In Bus Element or Out Bus Element block to open the dialog box for the related port.
2. Select an element or the top bus.
3. Specify the background color with the **Set color** dropdown menu.



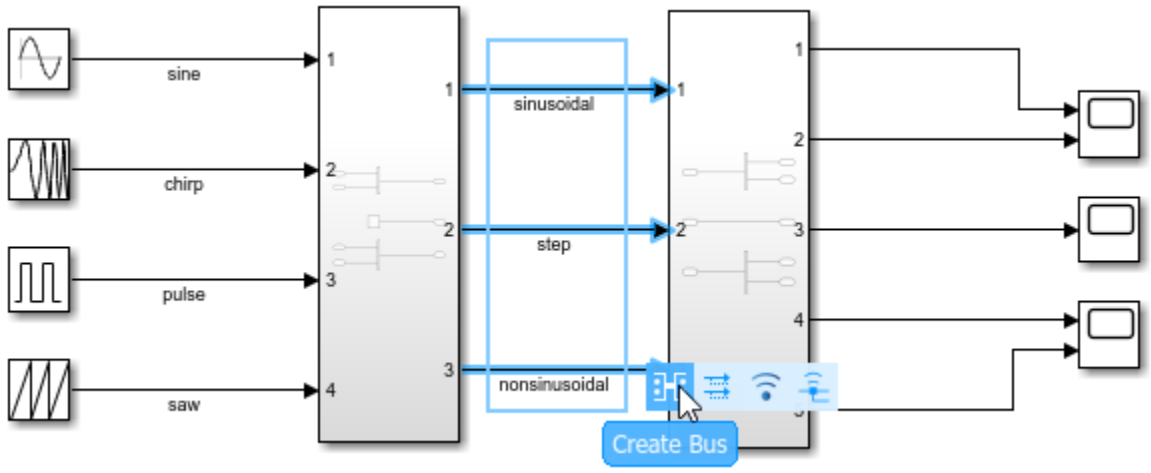
Combine Multiple Subsystem Ports into One Port

This example shows three ways to simplify a subsystem interface by converting multiple ports and their connected signals into one port and a bus.

Open the example model, which contains two subsystems with multiple input and output ports.

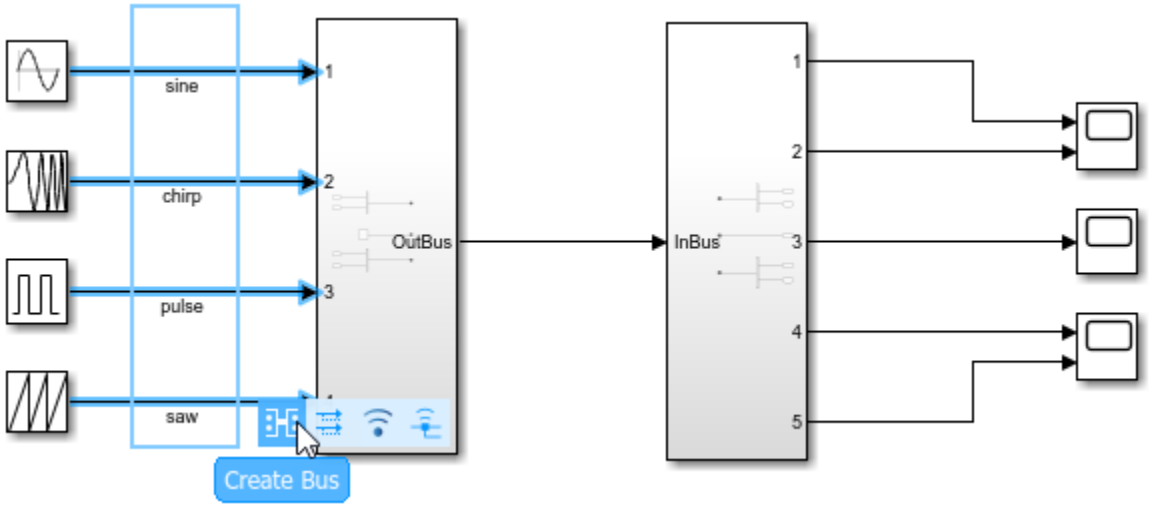


Drag a selection box around the signal lines between the two subsystems. From the action bar that appears, click **Create Bus**.



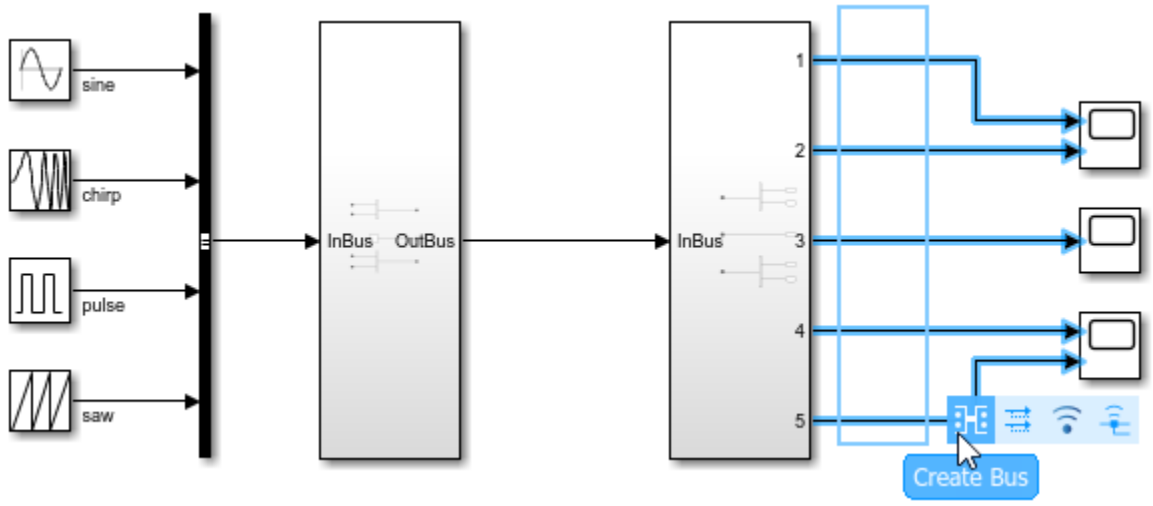
Simulink replaces the Inport and Outport blocks in the source and destination subsystems with In Bus Element and Out Bus Element blocks.

Drag a selection box around the signal lines between the source blocks and first subsystem. From the action bar that appears, click **Create Bus**.



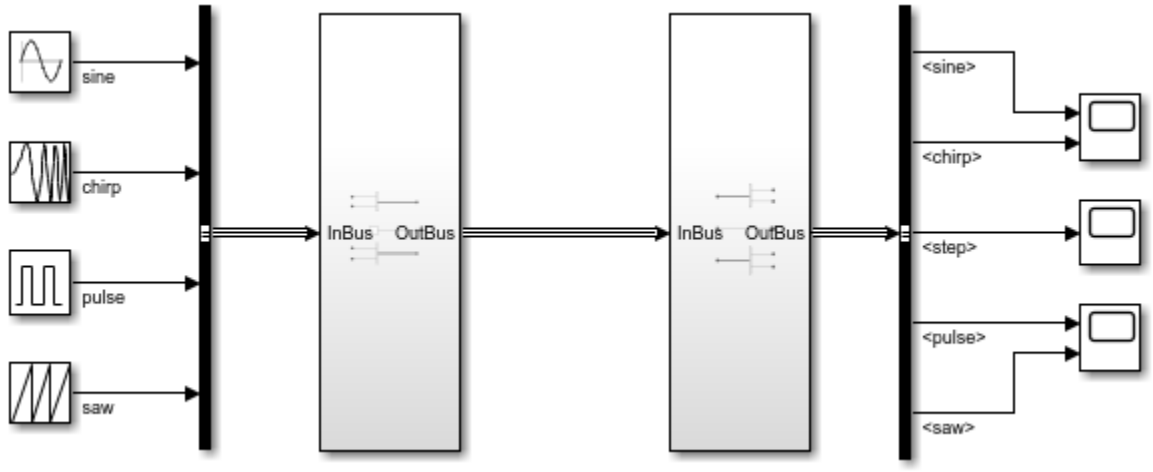
Simulink adds a Bus Creator block before the first subsystem and replaces the Inport blocks in the first subsystem with In Bus Element blocks.

Drag a selection box around the signal lines between the second subsystem and Scope blocks. From the action bar that appears, click **Create Bus**.



Simulink replaces the Outport blocks in the second subsystem with Out Bus Element blocks and adds a Bus Selector block after the second subsystem.

The resulting model uses virtual buses at the subsystem interfaces.



See Also

In Bus Element | Out Bus Element

Related Examples

- “Types of Composite Signals” on page 76-2
- “Group Signal Lines into Virtual Buses” on page 76-8
- “Create Nonvirtual Buses” on page 76-19

Display Bus Information

You can display bus information using multiple approaches.

- To display whether a bus is virtual or nonvirtual, update or simulate the model. A virtual bus appears as three solid lines () and a nonvirtual bus appears as two solid lines on either side of a dashed line ().
- To interactively display the hierarchy of a bus, see “Display Bus Hierarchy” on page 76-31.
- To interactively display the value of bus elements at a port, see “Display Value of Bus Elements” on page 76-32.
- To programmatically display the virtuality and hierarchy of a bus, see “Programmatically Get Bus Hierarchy and Virtuality” on page 76-33.

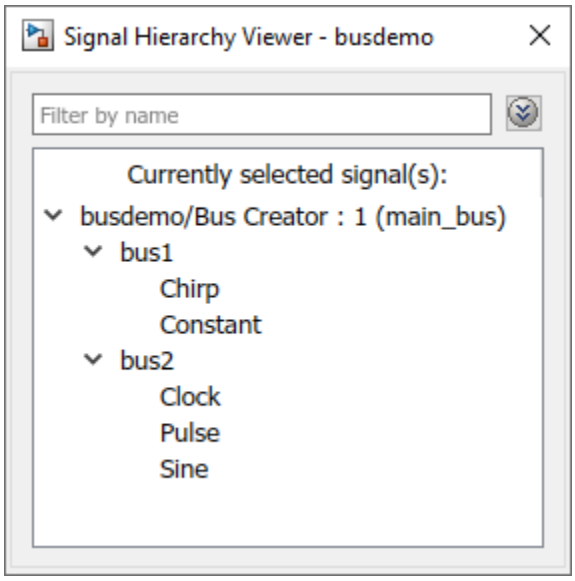
Display Bus Hierarchy

To interactively display bus hierarchy:

1. Click a signal line.
2. On the **Signal** tab, select **Signal Hierarchy**.

   A Signal Hierarchy Viewer opens, showing the signal hierarchy for the selected signal.

For example, this Signal Hierarchy Viewer shows the signal hierarchy for a bus named `main_bus`.

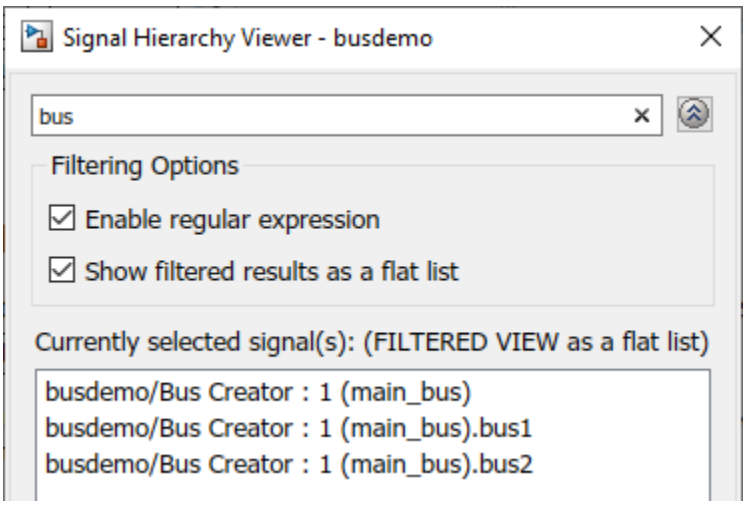


Each Signal Hierarchy Viewer is associated with a specific model. If you edit a model while the associated Signal Hierarchy Viewer is open, the Signal Hierarchy Viewer reflects those updates.

Note To produce accurate results at edit time in the Signal Hierarchy Viewer, your model must compile successfully.

To filter the displayed signals, click the **Options** button on the right side of the **Filter by name** edit box.

- To use MATLAB regular expressions for filtering signal names, select **Enable regular expression**. For example, to display all signals whose names end with a lowercase `r` (and their immediate parents), enter `r$` in the **Filter by name** edit box. For more information, see “Regular Expressions”.
- To display a flat list of the filtered results, select **Show filtered results as a flat list**. The flat list uses dot notation to indicate the hierarchy of buses. This example shows a filtered set of nested buses.

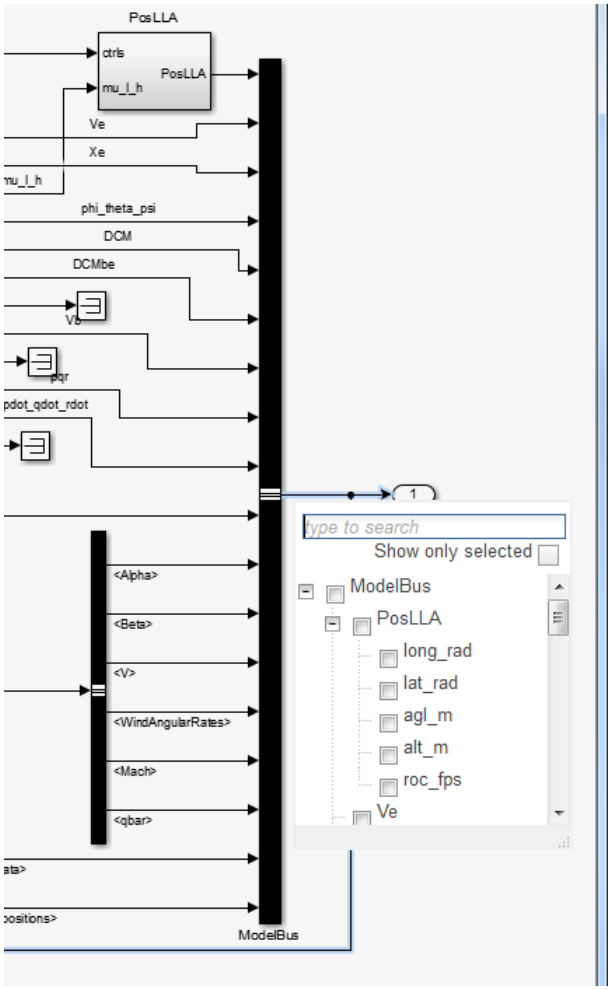


Display Value of Bus Elements

To interactively display the values of bus elements at a port:

1. Click a signal line.
2. On the **Signal** tab, select **Output Value Label**.
3. Click the port value label, and select the signals you want to display.

   For example, in this model, you can select which signals to display from the signals that are contained in `ModelBus`.



For more information, see “Display Value for a Specific Port” on page 36-19.

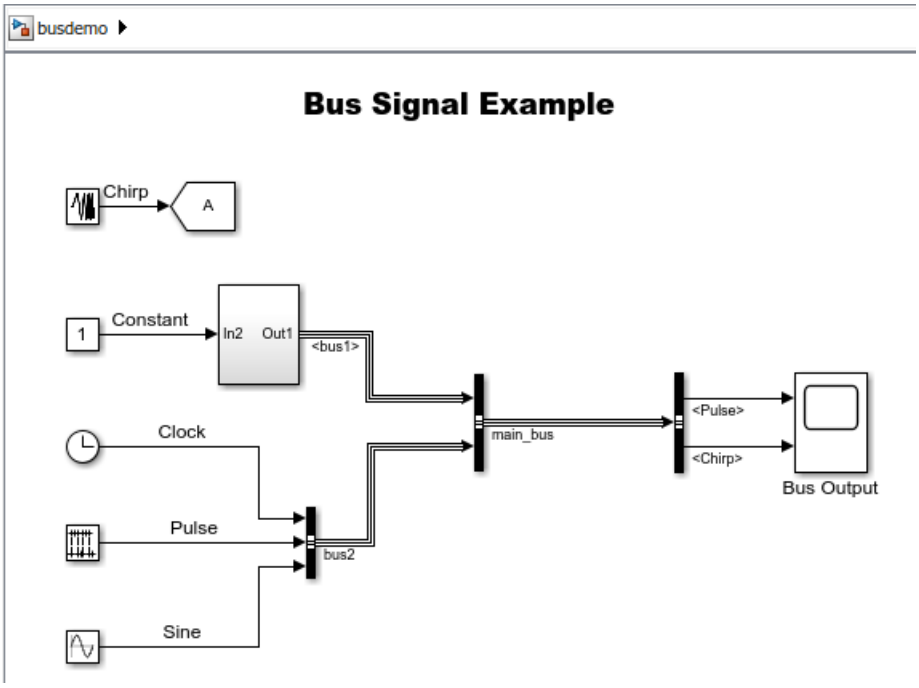
Programmatically Get Bus Hierarchy and Virtuality

To programmatically get the hierarchy and virtuality of a bus in a compiled model, query these parameters with the `get_param` command:

- `'SignalHierarchy'` — If the signal is a bus, returns the name and hierarchy of the signals in the bus.
- `'CompiledBusType'` — For a model that is in the `'compile'` phase, returns information about whether the signal connected to a port is a bus and whether the signal is a virtual or nonvirtual bus. Before you query the `CompiledBusType` parameter value, use the `model` function to put the model in the `'compile'` phase.

For example, open and simulate the `busdemo` model.

```
open_system('busdemo')
sim('busdemo');
```



Obtain the handle of the port for which you want bus information.

```
ph = get_param(['busdemo/Bus Creator'], 'PortHandles');
```

Get the signal hierarchy at the port.

```
sh = get_param(ph.Outport, 'SignalHierarchy')
```

```
sh =

  struct with fields:

    SignalName: 'main_bus'
     BusObject: ''
      Children: [2×1 struct]
```

Get the compiled bus type at the port while the model is compiling.

```
busdemo([],[],[],'compile');
bt = get_param(ph.Outport, 'CompiledBusType')
```

```
bt =

    'VIRTUAL_BUS'
```

Terminate compilation.

```
busdemo([],[],[],'term');
```

See Also

Related Examples

- “Display Value for a Specific Port” on page 36-19
- “Group Signal Lines into Virtual Buses” on page 76-8

Bus-Capable Blocks

Bus-capable blocks can accept buses as input, produce buses as output, or both. Some bus-capable blocks work with nonvirtual buses, but not with virtual buses. Some bus-capable blocks have additional requirements for buses; see the block documentation for details.

Block	Input
All virtual blocks	✓
Assignment(nonvirtual buses)	✓
Bus Assignment	✓
Bus Creator	✓
Bus Selector	✓
Constant (nonvirtual buses)	
Data Store Memory (nonvirtual buses)	Has no input port, can store buses
Data Store Read (nonvirtual buses)	
Data Store Write (nonvirtual buses)	✓
Delay	✓ (special requirements)
From File (nonvirtual buses)	
From Workspace (nonvirtual buses)	
Signal Editor (nonvirtual buses)	
Interpolation Using Prelookup	✓
Manual Switch	✓
Memory	✓ (special requirements)
Merge	✓ (special requirements))
Multiport Switch	✓ (special requirements)
Prelookup	
Rate Transition	✓ (special requirements)
Signal Conversion	✓
Signal Specification	✓
Switch	✓ (special requirements)
To File (nonvirtual buses)	✓ (special requirements)
To Workspace (nonvirtual buses)	✓
Unit Delay	✓ (special requirements)

Block	Input
Vector Concatenate (nonvirtual buses)	✓
Zero-Order Hold	✓

These modeling patterns support the use of buses.

- Subsystems
- Model referencing
- S-Functions
- Stateflow charts
- MATLAB Function blocks
- MATLAB System blocks

Buses that contain signals of enumerated data types do not pass through a block that requires a nonzero scalar initial value (such as a Unit Delay block). Use a structure value to initialize signals with enumerated types.

Root level bus outputs are not logged when you select the **Output** configuration parameter. Use standard signal logging instead, as described in “Export Signal Data Using Signal Logging” on page 72-41.

Do not use signal logging for bus or arrays of buses directly from within a For Each subsystem. Either use a Bus Selector block to select the bus element signals to log or add an Outport block outside of the subsystem and then log that signal. For details, see “Log Signals in For Each Subsystems” on page 72-71.

See Also

Related Examples

- “Identify Automatic Bus Conversions” on page 76-40
- “S-Function Limitations”
- “Nonvirtual Buses at Model Interfaces” on page 76-55
- “Nonvirtual Buses and MATLAB System Block” on page 45-15

Replace Values of Bus Elements

To assign the value of an input to a bus element, you can use a Bus Assignment block. Use a Bus Assignment block to change bus element values without adding Bus Selector and Bus Creator blocks that select bus elements and reassemble them into a bus.

Connect to the Bus Assignment block ports:

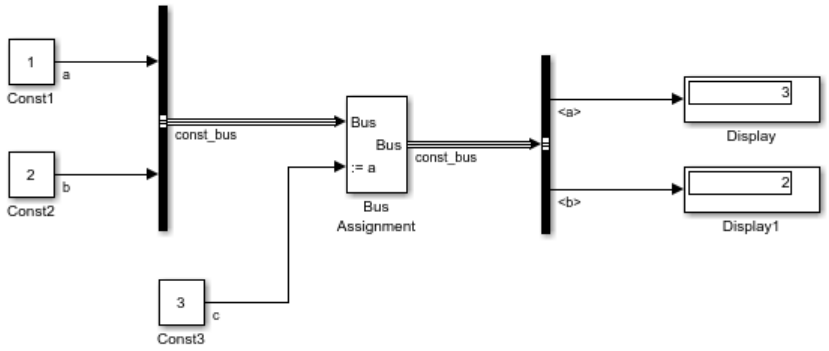
- The bus to which you want to assign the values
- The lines whose values you want to assign to specified bus elements

Connect the bus to the first input port of the Bus Assignment block, and one or more lines with values to be assigned to the other ports. The Block Parameters dialog box lists the elements available for assignment in the bus. The bus can be virtual or nonvirtual. Select the elements to which you want to assign values. If you specify more than one element to assign values to, the Bus Assignment block adds ports.

The elements that you assign values to can be nonbus or bus signals. The new values must match the attributes of the elements in the original bus.

Update a Bus Element

This simple example illustrates the mechanics of using the Bus Assignment block. In more complex models, using a Bus Assignment block simplifies updating a bus to reflect the processing that occurs in a separate component, such as a subsystem or referenced model. Here is the model after you simulate it.



Some key steps in constructing this model are:

1. Connect two Constant blocks to a Bus Creator block. The value of signal `a` is 1, and the value of signal `b` is 2.
2. Connect the Bus Creator output bus `const_bus` to the first port of a Bus Assignment block. The bus elements `a` and `b` are available to assign new values to them.
3. Connect the Constant block output signal `c` to the second port of the Bus Assignment block.
4. For the Bus Assignment block, in the Block Parameters dialog box **Elements in the bus** list, select the `a` signal and click **Select**.
5. Use a Bus Selector to select signals `a` and `b` from the `const_bus` bus and connect those signals to Display blocks.

6 Simulate the model. The Display blocks show that the value of signal `a`, which was 1 when the `const_bus` bus was created, is now 3, reflecting the assignment of the `c` signal from the `Const3` block.

See Also

Blocks
Bus Assignment

Related Examples

- “Group Signal Lines into Virtual Buses” on page 76-8

Identify Automatic Bus Conversions

To comply with composite signal requirements and limitations, Simulink may add hidden Bus to Vector and Signal Conversion blocks to your model. These additions help you avoid manually converting or refactoring your models, which can be time consuming and error prone.

Bus-to-Vector Conversions

When a block requires a vector but receives a virtual bus, a hidden Bus to Vector block may convert the bus to a vector. Bus to Vector blocks are virtual and do not affect simulation results, code generation, or performance. However, when a bus is treated as a vector, the elements of the bus become inaccessible.

To receive warnings or errors when a bus is treated as a vector, set the **Bus signal treated as vector** configuration parameter to `warning` or `error`, respectively. These settings allow you to identify potential problems at the source, instead of at downstream blocks that expect a bus.

To correct buses used as vectors:

- In the Model Advisor for the top model, run the “Check bus signals treated as vectors” check and perform the recommended actions.
- Use the `Simulink.BlockDiagram.addBusToVector` function to add Bus To Vector blocks where Simulink would implicitly convert buses to vectors. For an example, see “Manage Bus-to-Vector Conversions”.
- Replace the related Bus Creator block with a Mux block, which creates a vector.

Virtual and Nonvirtual Bus Conversions

When updating a diagram before simulation or code generation, Simulink might automatically convert a virtual bus to a nonvirtual bus or a nonvirtual bus to a virtual bus. For example, Simulink implicitly converts a bus when:

- A block, such as an S-Function block or a Stateflow chart, receives a virtual bus, but requires a nonvirtual bus.
- A referenced model receives a nonvirtual bus, but the corresponding input port specifies a virtual bus.
- A root-level output port receives a nonvirtual bus, but specifies a virtual bus.

Simulink inserts hidden Signal Conversion blocks into the model where needed. If a `Simulink.Bus` object is not specified for a virtual bus, the conversion to a nonvirtual bus fails and you receive an error.

Unlike Bus to Vector blocks, Signal Conversion blocks do not affect the structure of the output bus.

See Also

Blocks
Bus to Vector | Signal Conversion

Related Examples

- “Manage Bus-to-Vector Conversions”

More About

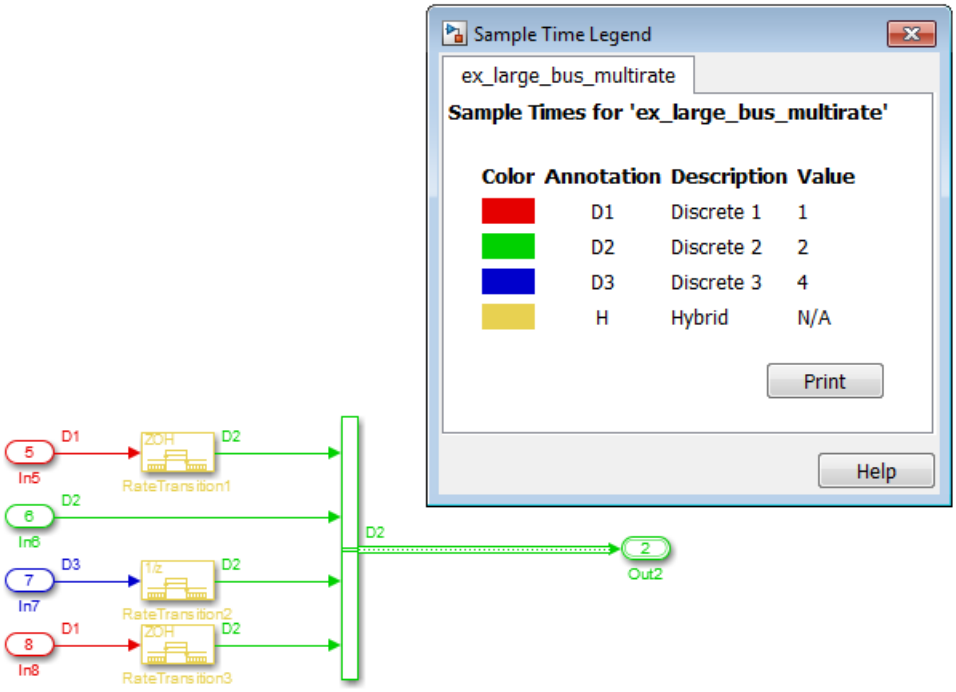
- “Types of Composite Signals” on page 76-2
- “Check Your Model Using the Model Advisor” on page 5-2
- “Nonvirtual Buses at Model Interfaces” on page 76-55

Modify Sample Times for Nonvirtual Buses

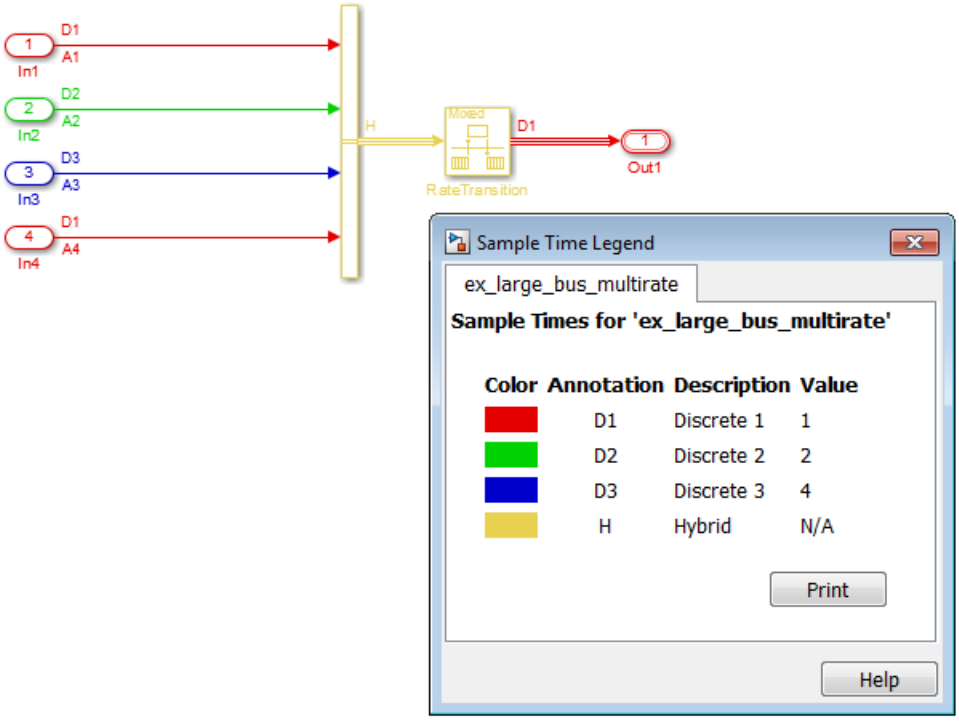
Unlike virtual buses, which can combine signals that have different sample times, all of the signals in a nonvirtual bus must have the same sample time.

To group signals with different sample times in a nonvirtual bus, make the sample times of the input signals the same by inserting Rate Transition blocks.

Suppose a model creates a nonvirtual bus using a Bus Creator block. To standardize the sample time of the input signals, the model uses three Rate Transition blocks before the Bus Creator block.



Suppose another model converts a virtual bus to a nonvirtual bus using a root-level Outport block. To standardize the sample time of the elements of the virtual bus, the model uses a single Rate Transition block before the Outport block.



Alternatively, specify the same sample time at each block that generates an element of the bus.

See Also

Blocks
Rate Transition

More About

- “Types of Composite Signals” on page 76-2
- “Identify Automatic Bus Conversions” on page 76-40
- “Sample Time”

Specify Bus Properties with Simulink.Bus Objects

A bus can be associated with a `Simulink.Bus` object, which specifies properties that Simulink uses to validate the bus. `Bus` objects are optional for virtual buses, but required for nonvirtual buses.

A `Bus` object specifies only the architectural properties of a bus, as distinct from the values of the signals it contains. For example, a `Bus` object can specify the number of elements in a bus, the order of those elements, whether and how elements are nested, and the data types of constituent signals; but not the signal values.

A `Bus` object is analogous to a structure definition in C: it defines the members of the bus, but does not create the bus. A `Bus` object is also similar to a cable connector. The connector defines all the pins and their configuration and controls what types of wires can be connected to it. Similarly, a `Bus` object defines the configuration and properties of the signals that the associated bus must have.

Bus Object Workflow

Using `Bus` objects in a model involves performing these tasks, in many cases iteratively.

1. “Determine Whether to Use Simulink.Bus Objects” on page 76-44
2. “Determine How to Manage Simulink.Bus Objects” on page 76-45
3. “Create and Specify Simulink.Bus Objects” on page 76-46
4. “Save Simulink.Bus Objects” on page 76-47
5. “Map Simulink.Bus Objects to Models” on page 76-47

Determine Whether to Use Simulink.Bus Objects

Required Uses of Simulink.Bus Objects

You must use `Bus` objects for these modeling configurations:

- Nonvirtual buses that cross model reference boundaries
- Stateflow charts with bus input or output
- S-function or Legacy Code Tool interface with external code

You can associate a `Bus` object with multiple blocks. Some blocks require that you specify a `Bus` object if the block has a bus input or output. When a `Bus` object governs a signal input or output for a block, the signal must be a bus that has the properties specified by the object. Any variance causes an error.

These blocks require a `Bus` object for bus input and output.

- Constant
- Data Store Memory
- Data Store Read
- Data Store Write
- From File
- From Workspace

- Function Caller
- Inport (top-level)
- Interpolation Using Prelookup
- MATLAB Function
- MATLAB System
- Outport (top-level)
- Prelookup
- S-Function
- State Reader

Optional Uses of Simulink.Bus Objects

If you use Bus Creator block parameters to specify bus properties, all blocks downstream from the bus inherit the same properties.

You can use Bus Creator block parameters to define virtual buses and perform limited error checking. To perform thorough error checking on a bus, associate a `Bus` object with that bus. Using `Bus` objects to check buses for errors is important when you want to create reusable and shareable model components.

To make tracing the correspondence between the model and the generated code for a bus easier, use a nonvirtual bus. The generated code for a nonvirtual bus produces a structure. Nonvirtual buses can result in multiple copies of some buses.

These blocks can specify a `Bus` object for bus input and output.

- Argument Inport
- Argument Outport
- Bus Creator
- In Bus Element
- Out Bus Element
- Permute Dimensions
- Probe
- Reshape
- Signal Conversion
- Signal Specification
- Unit Delay

Determine How to Manage Simulink.Bus Objects

You can save `Bus` objects to these locations:

- Data dictionary
- Function
- MAT-file
- Database or other external files

If you do not save Bus objects, then when you reopen a model that uses the Bus objects, you need to recreate the Bus objects.

Different Bus object storage locations provide different advantages.

Location	Usage Considerations
Data dictionary	Use for large model componentization. When you save to a data dictionary from the base workspace, you get all the variables used by the model, not just the Bus objects. Before you save to a data dictionary, read “Considerations before Migrating to Data Dictionary” on page 74-7.
Function	Use for when you want to use MATLAB for traceability and model differencing.
MAT-file	Use for faster Bus object saving and loading.
Database or other external files	Use for comparing bus interface information with design documents stored in an external data source.

Create and Specify Simulink.Bus Objects

To create or edit Bus objects interactively, use the **Bus Editor** or **Model Explorer**. Bus objects created with these tools are initially stored in the base workspace or data dictionary. To visualize bus hierarchy and access capabilities such as import and export, use the **Bus Editor**. When you have many Bus objects or the Bus objects are stored in multiple locations, use the **Model Explorer**. The **Model Explorer** provides quick startup regardless of the number of Bus objects and allows you to easily switch between editing Bus objects in the base workspace and data dictionaries.

To create and edit Bus objects programmatically, see “Create Bus Objects Programmatically” on page 76-49. Bus objects are initially stored in either the base workspace or a function.

After you create a Bus object and specify its attributes, you can associate it with any block that needs to use the bus definition that the object provides. To associate a block with a bus, in the Block Parameters dialog box, set **Data type** to `Bus: <object name>` and replace `<object name>` with the Bus object name.

You can specify the Bus object as the data type of a block either before or after defining the Bus object. However, before you simulate the model, the Bus object and the corresponding bus must have the same number of bus elements, in the same order. Also, each element in the Bus object and in the corresponding bus must have the same data type and dimensions.

During model development, you can modify buses to match Bus objects or modify Bus objects to match buses.

If you do not want to change the Bus object, you can:

- Create a Bus object that matches the changes to the bus and use the new Bus object for the blocks that the changed bus connects to.
- Revert the bus changes so that the bus continues to match the associated Bus object.

Save Simulink.Bus Objects

To save `Bus` objects stored in the base workspace, you can use any MATLAB technique that saves the contents of the base workspace. However, the resulting file contains everything in the base workspace, not just `Bus` objects.

Location	File Creation Method	File Contents
Data dictionary	See "Migrate Models to Use Simulink Data Dictionary" on page 74-6.	`Bus` objects and other base workspace variables used by a model
Function	Use the Bus Editor or `Simulink.Bus.save` function.	`Bus` objects
MAT-file	Use the Bus Editor.	`Bus` objects
Database or other external files	Use the `Simulink.importExternalCTypes` function, scripts, or Database Toolbox™ functionality on C code structure (`struct`) definitions. In preparation for integrating existing algorithmic C code for simulation (for example, by using the Legacy Code Tool), you can package signal or parameter data in the definitions according to structure type.	`Bus` objects

You can customize `Bus` object export by providing a custom function that writes to a location outside MATLAB. For example, exported `Bus` objects can be saved as records in a database. See "Customize Bus Object Import and Export" on page 76-51 for details.

When you modify saved `Bus` objects, you must resave them to keep the changes.

Map Simulink.Bus Objects to Models

Before you simulate a model, all the `Bus` objects it uses must be loaded into the base workspace. For automation and consistency across models, mapping `Bus` objects to models is important.

- By identifying all of the `Bus` objects that a model requires, you can ensure that those objects are loaded before model execution.
- By identifying all models that use a `Bus` object, you can ensure that changes to a `Bus` object do not cause unexpected changes in any of the models that use the `Bus` object.

To ensure the necessary `Bus` objects load before model execution, consider:

- Projects — Automatically load or run files that define `Bus` objects by configuring the files to run when you open a project. For details, see "Project Management".
- Data dictionaries — Store `Bus` objects with variables and other objects for one or more models.

  To share a `Bus` object among models, you can link each model to a dictionary and create a common referenced dictionary to store the object. For an example, see "Partition Dictionary Data Using Referenced Dictionaries" on page 74-25.

- Databases — Capture mapping information in an external data source, such as a database.

  You can customize `Bus` object import by providing a custom function that reads from a location outside MATLAB. See “Customize Bus Object Import and Export” on page 76-51 for details.
- Model callbacks — Automatically load or run files that define `Bus` objects by using the `load` function in a model callback.

  If a model uses only a few `Bus` objects, consider copying the `Bus` object code directly into the callback, instead of loading a file. For an example, open model and examine the callback.

To find where a `Bus` object is used in an open model, see “Finding Blocks That Use a Specific Variable” on page 67-111.

Tip Using a rigorous and standard naming convention is very helpful for mapping `Bus` object usage. For example, consider the model and data required for an actuator control function. Naming the model `Actuator` and the input and output ports `Actuator_bus_in` and `Actuator_bus_out`, respectively, makes the connection between the `Bus` objects and the model clear.

Note that this approach can cause issues if the output from one model is fed directly to another model. In this case, the naming mismatch results in an error.

See Also

Classes
`Simulink.Bus` | `Simulink.BusElement`

Related Examples

- Bus Editor
- “Create Bus Objects Programmatically” on page 76-49
- “Customize Bus Object Import and Export” on page 76-51
- “Modify Sample Times for Nonvirtual Buses” on page 76-42

Create Bus Objects Programmatically

You can programmatically create a `Simulink.Bus` object and its `Simulink.BusElement` objects from arrays, blocks, cell arrays, structures, or C code.

As you create `Bus` objects programmatically, you can store them in the MATLAB workspace or a data dictionary or save their definitions in a function. For `Bus` objects in the base workspace, you can programmatically save their definitions in a function using the `Simulink.Bus.save` function.

To simulate a block that uses a `Bus` object, that `Bus` object must be in the base workspace or in a data dictionary.

Create Bus Objects from Arrays

Create a hierarchy of `Bus` objects using arrays. Array indexing lets you create and access multiple elements in an array. Dot notation lets you access property values.

Create two `BusElement` objects, named `Chirp` and `Sine`, in the base workspace.

```
elems(1) = Simulink.BusElement;
elems(1).Name = 'Chirp';

elems(2) = Simulink.BusElement;
elems(2).Name = 'Sine';
```

Create a `Bus` object, named `NestedBus`, that uses the elements defined in the `elems` array.

```
NestedBus = Simulink.Bus;
NestedBus.Elements = elems;
```

Create two more `BusElement` objects, named `NestedBus` and `Step`. To have `NestedBus` represent a `Bus` object, specify a `Bus` object data type.

```
clear elems

elems(1) = Simulink.BusElement;
elems(1).Name = 'NestedBus';
elems(1).DataType = 'Bus: NestedBus';

elems(2) = Simulink.BusElement;
elems(2).Name = 'Step';
```

Create the bus at the top of the bus hierarchy that uses the elements defined in the `elems` array.

```
TopBus = Simulink.Bus;
TopBus.Elements = elems;
```

You can view the created objects in the **Bus Editor**.

```
buseditor
```

Create Bus Objects from Blocks

To programmatically create a `Bus` object based on a block in a model, use the `Simulink.Bus.createObject` function.

If you specify a Bus Creator block that is at the highest level of a bus hierarchy, the function creates `Bus` objects for all of the buses in the hierarchy, including nested buses.

Create Bus Objects from MATLAB Data

To create a `Bus` object from a cell array, use the `Simulink.Bus.cellToObject` function. Each subordinate cell array represents a `Bus` object

To create a `Bus` object from a MATLAB structure, use the `Simulink.Bus.createObject` function. The structure can contain MATLAB `timeseries`, MATLAB `timetable`, and `matlab.io.datastore.SimulationDatastore` objects or be a numeric structure.

Create Bus Objects from External C Code

You can create a `Bus` object that corresponds to a structure type (`struct`) that your existing C code defines. Then, in preparation for integrating existing algorithmic C code for simulation (for example, by using the Legacy Code Tool), you can use the `Bus` object to package signal or parameter data according to the structure type. To create the object, use the `Simulink.importExternalCTypes` function.

See Also

Functions
`Simulink.Bus.cellToObject` | `Simulink.Bus.createObject`

Classes
`Simulink.Bus` | `Simulink.BusElement`

Related Examples

- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44

Customize Bus Object Import and Export

You can use the Bus Editor to import `Bus` objects to the base workspace and to export `Bus` objects from the base workspace, as described in “Save Simulink.Bus Objects” on page 76-47. By default, the Bus Editor can save bus objects to, and import `Bus` objects from, a function or MAT-file. The files must be in a location that is accessible using an ordinary **Open** or **Save** dialog box.

You can write customized MATLAB functions that provide alternative import or export (or both) functionality. For example, you can write a customized function that stores the objects as records in a database, in a format that your organization uses.

After you design and implement a custom `Bus` object import or export function, use the Simulink Customization Manager to register the function. The registration process establishes custom import and export functions as callbacks for the Bus Editor **Import to Base Workspace** and **Export to File** commands. The callbacks replace the default capabilities of the Bus Editor. Customizing the Bus Editor import and export capabilities has no effect on other MATLAB or Simulink functions. Canceling import or export customization restores the default Bus Editor capabilities for that operation without affecting the other.

To create `Bus` objects from external C code, you do not need to make customizations. See “Create Bus Objects from External C Code” on page 76-50.

Required Background Knowledge

Customizing `Bus` object import or export requires that you understand:

- MATLAB language and programming techniques
- Simulink `Bus` object syntax
- The proprietary format into which you translate `Bus` objects, and the techniques necessary to access the facility that stores the objects.
- Any platform-specific techniques for obtaining data from the user, such as the name of the location in which to store or access `Bus` objects.

Write a Bus Object Export Function

A custom `Bus` object export function requires at least one argument. You can use additional arguments to handle special actions by the function. The value of the first argument is a cell array containing the names of all `Bus` objects that the Bus Editor has selected. You can use functions, global variables, or any other MATLAB technique, to provide values for any additional arguments. The general algorithm of a customized export function is:

1. Iterate over the list of object names in the first argument.
2. Obtain the `Bus` object corresponding to each name.
3. Translate the `Bus` object to the proprietary syntax.
4. Save the translated `Bus` object in the local repository.

This example shows the syntactic shell of such an export callback function is:

```
function myExportCallBack(selectedBusObjects)
disp('Custom export was called!');
for idx = 1:length(selectedBusObjects)
```

```
    disp([selectedBusObjects{idx} ' was selected for export.']);
end
```

Although this function does not export any `Bus` objects, it is syntactically valid and can be registered. It accepts a cell array of `Bus` object names, iterates over them, and prints each name. An operational export function:

- Uses each name to retrieve the corresponding `Bus` object from the base workspace
- Converts the object to proprietary format
- Stores the converted object

The additional logic is enterprise-specific.

Write a Bus Object Import Function

A custom `Bus` object import function can take zero or more arguments to perform its task. You can use functions, global variables, or any other MATLAB technique to provide argument values. Also, the function can poll the user for information, such as a designation of where to obtain `Bus` object information. The general algorithm of a custom `Bus` object import function is:

1. Obtain `Bus` object information from the local repository.
2. Translate each `Bus` object definition to a `Simulink.Bus` object.
3. Save each `Bus` object to the MATLAB base workspace.

This example shows the syntactic shell of an import callback function is:

```
function myImportCallBack
disp('Custom import was called!');
```

Although this function does not import any `Bus` objects, it is syntactically valid and can be registered with the Simulink Customization Manager. An operational import function:

- Gets a designation of where to obtain the `Bus` objects to import
- Converts each `Bus` object to a `Simulink.Bus` object
- Stores the object in the base workspace

The additional logic is enterprise-specific.

Register Customizations

To customize `Bus` object import or export, provide a customization registration function that inputs and configures the Customization Manager whenever you start Simulink software or refresh Simulink customizations. The steps for using a customization registration function are:

1. Create a file named `sl_customization.m` to contain the customization registration function. Alternatively, you can use an existing customization file.
2. At the top of the file, create a function named `sl_customization` that takes a single argument (or use the customization function in an existing file). When the function is invoked, the value of this argument is the Customization Manager.
3. Configure the `sl_customization` function to set `importCallbackFcn` and `exportCallbackFcn` to be function handles that specify your customized `Bus` object import and export functions.

4 If `sl_customization.m` is a new customization file, put it anywhere on the MATLAB search path. Two frequently used locations are `matlabroot` and the current working folder. Alternatively, you can extend the search path.

Here is a simple example of a customization registration function:

```
function sl_customization(cm)
disp('My customization file was loaded.');
cm.BusEditorCustomizer.importCallbackFcn = @myImportCallBack;
cm.BusEditorCustomizer.exportCallbackFcn = @(x)myExportCallBack(x);
```

When Simulink starts up, it traverses the MATLAB search path looking for files named `sl_customization.m`. Simulink loads each such file that it finds (not just the first file) and executes the `sl_customization` function at its top, establishing the customizations that the function specifies.

Executing the example customization function displays a message (which an actual function probably would not). The function establishes that the Bus Editor uses a function named `myImportCallBack()` to import `Bus` objects, and a function named `myExportCallBack(x)` to export `Bus` objects.

The function corresponding to a handle that appears in a callback registration can be undefined when the registration occurs. However, it must be defined when the Bus Editor calls the function. The same latitude and requirement applies to any functions or global variables used to provide the values of additional arguments.

Other functions can also exist in the `sl_customization.m` file. However, the Simulink software ignores files named `sl_customization.m`, except when it starts up or refreshes customizations. Any changes to functions in the customization file are ignored until one of those events occurs. By contrast, changes to other functions on the MATLAB path take effect immediately.

For more information, see “Registering Customizations” on page 78-23.

Change Customizations

You can change the handles established in the `sl_customization` function by:

- Changing the function to specify the changed handles
- Saving the function
- Refreshing customizations by executing `sl_refresh_customizations`

Simulink traverses the MATLAB path and reloads all `sl_customization.m` files that it finds, executing the first function in each one, just as it did on Simulink startup.

You can revert to default import or export behavior by setting in the `sl_customization` function the appropriate `BusEditorCustomizer` element to `[]` and then refreshing customizations. Alternatively, you can eliminate both customizations in one operation by executing:

```
cm.BusEditorCustomizer.clear
```

where `cm` was previously set to a customization manager object (see “Register Customizations” on page 76-52).

Changes to the import and export callback functions themselves, as distinct from changes to the handles that register them as customizations, take effect immediately, unless they are in the

`sl_customization.m` file itself. If the callback functions are in the `sl_customization.m` file, they take effect next time you refresh customizations. Keeping the callback functions in separate files usually provides more flexible and modular results.

See Also

Blocks
Bus Creator

Functions
`Simulink.BlockDiagram.addBusToVector` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

Classes
`Simulink.Bus` | `Simulink.BusElement`

Related Examples

- “Simulink Environment Customization”
- “Register Customizations” on page 76-52
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44

Nonvirtual Buses at Model Interfaces

A model reference boundary is the boundary between a model that contains a Model block and the referenced model. If bus data passes to a Model block, then that data crosses the boundary to the referenced model. Bus data input for the Model block must be consistent with the bus data that the referenced model requires.

For bus data that crosses model reference boundaries, decide whether to use a virtual or nonvirtual bus. Using a virtual bus can be simpler than using a nonvirtual bus. However, using a nonvirtual bus provides a well-defined data interface for code generation. To learn the differences between virtual and nonvirtual buses, see “Types of Composite Signals” on page 76-2.

Tip For virtual buses at interfaces, use In Bus Element and Out Bus Element blocks. These blocks support multirate virtual buses and do not require `Simulink.Bus` objects.

For nonvirtual buses at interfaces, use Inport and Outport blocks.

Model Reference Requirements for Nonvirtual Buses

If you use a bus as an input to or an output from a referenced model:

- Only a nonvirtual bus can contain variable-size signal elements.
- For code generation, you can only configure the `I/O arguments step method` style of C++ class interface for the referenced model when using a nonvirtual bus or when using the `Default` style of C++ class interface.
- For code generation, you can only configure function prototype control for the referenced model when using a nonvirtual bus.

Nonvirtual Buses with Root-Level Inport Blocks

To pass a bus from a root-level Inport block into a referenced model, in the Block Parameters dialog box of the Inport block:

- Set the **Data type** parameter to `Bus: <object name>`
- Replace `<object name>` with the name of the `Bus` object that defines the bus the Inport block produces.

To output a nonvirtual bus from the root-level Inport block, select **Output as nonvirtual bus** in the Block Parameters dialog box of the Inport block.

All signals in a nonvirtual bus must have the same sample time. For details, see “Modify Sample Times for Nonvirtual Buses” on page 76-42.

For information about importing data to root-level Inport blocks, see “Load Bus Data to Root-Level Input Ports” on page 70-46.

Nonvirtual Buses with Root-Level Outport Blocks

To pass a bus through a root-level Outport block of a referenced model, in the Block Parameters dialog box of the Outport block:

- Set **Data type** to `Bus: <object name>`.
- Replace `<object name>` with the name of the `Bus` object that defines the bus the Outport block produces.

To output a nonvirtual bus from the port that corresponds to the root-level Outport block, select **Output as nonvirtual bus in parent model** in the Block Parameters dialog box of the Outport block.

All signals in a nonvirtual bus must have the same sample time. For details, see “Modify Sample Times for Nonvirtual Buses” on page 76-42.

Rate Transitions for Nonvirtual Buses

To pass a multirate bus as a nonvirtual bus into a referenced model, use an Inport block. Add blocks in the parent and referenced model as follows:

1. In the parent model, convert the multirate bus to a single-rate bus by inserting a Rate Transition block. Inport blocks can only pass single-rate nonvirtual buses into referenced models. The Rate Transition block must specify a rate in its **Block Parameters > Output port sample time** field unless one of the following is true:
   - The **Configuration Parameters > Solver** pane specifies a rate with these settings:
     - The **Periodic sample time constraint** parameter is set to `Specified`.
     - The **Sample time properties** parameter contains the specified rate.
   - The Inport block that accepts the nonvirtual bus in the referenced model specifies a rate in its **Block Properties > Signal Attributes > Sample time** field.
2. In the referenced model, use a Bus Selector block to pick out signals of interest, and use Rate Transition blocks to convert the signals to the desired rates.

See Also

Blocks

Bus Creator | Bus Selector | In Bus Element | Inport | Out Bus Element | Outport | Rate Transition | Signal Conversion

More About

- “Types of Composite Signals” on page 76-2
- “Create Nonvirtual Buses” on page 76-19
- “Load Bus Data to Root-Level Input Ports” on page 70-46
- “Load Input Data for a Bus Using In Bus Element Blocks” on page 70-55
- “Modify Sample Times for Nonvirtual Buses” on page 76-42

Specify Initial Conditions for Bus Signals

Bus signal initialization is a special form of signal initialization. For general information about initializing signals, see “Initialize Signals and Discrete States” on page 75-37. For details about initializing arrays of buses, see “Initialize Arrays of Buses” on page 76-76.

Bus signal initialization specifies the bus element values that Simulink uses for the first execution of a block that uses that bus. By default, the initial value for a bus element is the ground value (represented by `0`). Bus initialization involves specifying nonzero initial conditions.

You can use bus initialization features to:

- Specify initial conditions for signals that have different data types.
- Apply a different initial condition for each signal in the bus.
- Specify initial conditions for a subset of signals in a bus without specifying initial conditions for all the signals.
- Use the same initial conditions for multiple blocks, signals, or models.

Blocks That Support Bus Signal Initialization

You can initialize bus values that input to a block if that block meets both of these conditions:

- Has an initial value or initial condition block parameter
- Supports buses

These blocks support bus initialization:

- Data Store Memory
- IC
- Memory
- Merge
- Outport (when the block is inside a conditionally executed context)
- Receive
- Rate Transition
- Unit Delay

For example, the Unit Delay block is a bus-capable block. Its Block Parameters dialog box has an **Initial conditions** parameter.

You cannot initialize a bus that has:

- Variable-size signals
- Frame-based signals

Set Diagnostics to Support Bus Initialization

To enable bus initialization, before you start a simulation, set the “Underspecified initialization detection” configuration parameter to `simplified`.

Create Initial Condition Structures

You can create partial or full initial condition structures to represent initial values for a bus. To create an initial condition structure, use one of these approaches:

- Define a MATLAB structure in the MATLAB base or Simulink model workspace. You can manually define the structure, or alternatively for full structures, you can use the `Simulink.Bus.createMATLABstruct` function.
- In the Block Parameters dialog box for a block that supports bus initialization, for the initial condition parameter specify an expression that evaluates to a structure.

For information about defining MATLAB structures, see “Create Structure Array”.

The field that you specify in an initial condition structure must match these data attributes of the bus element exactly:

- Name
- Dimension
- Complexity

For example, if you define a bus element to be a real [2x2] double array, then in the initial condition structure, define the value to initialize that bus element to be a real [2x2] double array.

Explicitly specify fields in the initial condition structure for every bus element that has an enumerated (`enum`) data type.

Control Data Types of Structure Fields

If any of the signal elements of the target bus use a data type other than `double`, you can use different techniques to control the data types of the fields of initial condition structures. The technique that you choose can influence the efficiency and readability of the generated code. See “Control Data Types of Initial Condition Structure Fields” on page 76-101.

Create Full Structures for Initialization

A full initial condition structure provides an initial value for every element of a bus. The initial condition structure mirrors the bus hierarchy and reflects the attributes of the bus elements.

Specifying full structures during code generation offers these advantages:

- Generates more readable code
- Supports a modeling style that explicitly initializes all signals

Use the `Simulink.Bus.createMATLABStruct` function to streamline the creation of a full MATLAB initial condition structure with the same hierarchy, names, and data attributes as a bus. This function fills all of the elements that you do not specify with ground values for those elements.

You can use several different kinds of input with the function, including:

- A `Bus` object name
- An array of port handles

You can invoke the function from the Bus Editor. Select the `Bus` object for which you want to create a full MATLAB structure, and then select the **File > Create a MATLAB structure** menu item.

To detect when structure parameters are not consistent in shape (hierarchy and names) with the associated bus, use the Model Advisor.

1. On the **Modeling** tab, click **Model Advisor**.
2. Click **OK**.
3. Select **By Task > Modeling Signals and Parameters using Buses > "Check structure parameter usage with bus signals"**.
4. Click the **Run This Check** button.

   The Model Advisor identifies partial initial condition structures.

After you create the structure, you can edit it in the MATLAB Editor.

Create Partial Structures for Initialization

A partial initial condition structure provides initial values for a subset of the elements of a bus. If you use a partial initial condition structure, during simulation, Simulink creates a full initial condition structure to represent all the bus elements. Simulink assigns the respective ground value to each element for which the partial initial condition structure does not explicitly assign a value.

Specifying partial structures for block parameter values can be useful during the iterative process of creating a model. Partial structures enable you to focus on a subset of signals in a bus. When you use partial structures, Simulink initializes unspecified signals implicitly.

When you define a partial initial condition structure:

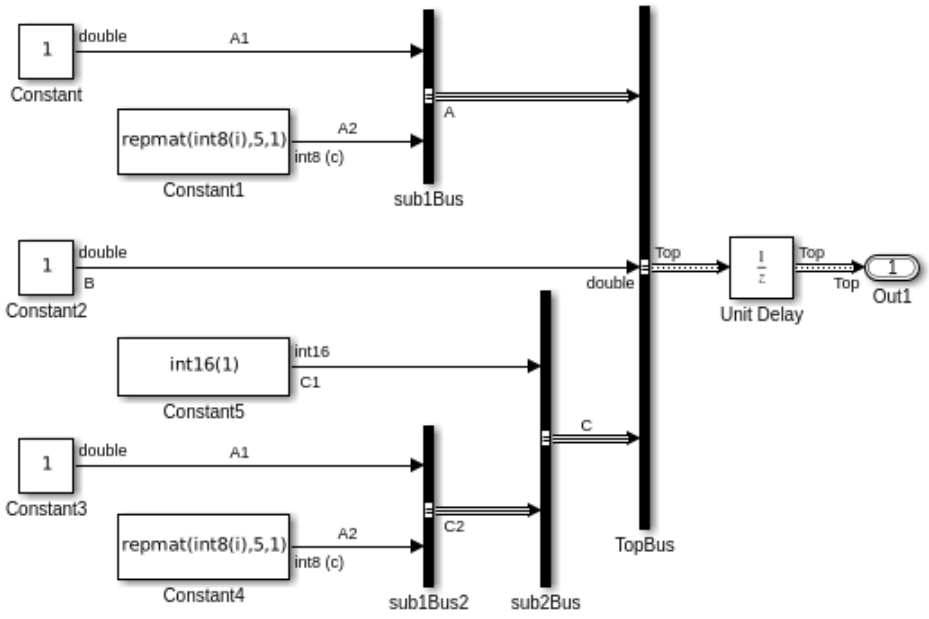
- Include only fields that are in the bus.
- Omit one or more fields that are in the bus.
- Make the field in the initial condition structure correspond to the nesting level of the bus element.
- Within the same nesting level in both the structure and the bus, optionally specify the structure fields in a different order than the bus elements.

Note The value of an initial condition structure must lie within the design minimum and maximum range of the corresponding bus element. Simulink performs this range checking when you do an update diagram or simulate the model.

Suppose that you have a bus, `Top`, composed of three elements: `A`, `B`, and `C`, with these characteristics:

- `A` is a nested bus, with two signal elements.
- `B` is a single signal.
- `C` is a nested bus that includes bus `A` as a nested bus.

The `ex_bus_initial_conditions` model includes the nested `Top` bus. This is how the model appears after it has been updated.



Here is a summary of the `Top` bus hierarchy and the data type, dimension, and complexity of the bus elements.

```
Top
   A (sub1)
      A1 (double)
      A2 (int8, 5x1, complex)
   B (double)
   C (sub2)
      C1 (int16)
      C2 (sub1)
         A1 (double)
         A2 (int8, 5x1, complex)
```

In these examples, `K` is an initial condition structure specified for the initial value of the Unit Delay block. The initial condition structure corresponds to the `Top` bus in the `ex_bus_initial_conditions` model. Here are some valid initial condition specifications.

Valid Syntax	Description
`K.A.A1 = 3`	Initialize the bus element `Top.A.A1` using the value `3`.
`K = struct('C',struct('C1',int16(4)))`	The bus element `Top.C.C1` is `int16`. The corresponding structure field explicitly specifies `int16(4)`. Alternatively, you could specify the field value as `4` without specifying an explicit data type.
`K = struct('B',3,'A',struct('A1',4))`	Bus elements `Top.B` and `Top.A` are at the same nesting level in the bus. For bus elements at the same nesting level, the order of corresponding structure fields does not matter.

Invalid Partial Initial Condition Structures

In the following examples, `K` is an initial condition structure specified for the initial value of the Unit Delay block. The initial condition structure corresponds to the `Top` bus in the `ex_bus_initial_conditions` model.

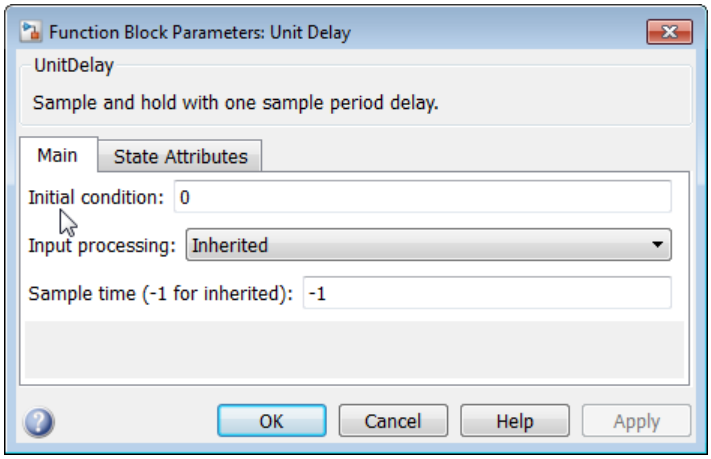
These three initial condition specifications are *not* valid:

Invalid Syntax	Reason the Syntax Is Invalid
`K.A.A2 = 3`	Value dimension and complexity do not match. The bus element `Top.A.A2` is `5x1`, but `K.A.A2` is `1x1`; `Top.A.A2` is complex, but `K.A.A2` is real.
`K.C.C2 = 3`	You cannot use a scalar value to initialize initial condition substructures.
`K = struct('B',3,'X',4)`	You cannot specify fields that are not in the bus (`X` does not exist in the bus).

Initialize Bus Signals Using Block Parameters

Initialize a bus by setting the initial condition parameter for a block that receives a bus as input and that supports bus initialization (see “Blocks That Support Bus Signal Initialization” on page 76-57).

For example, the Block Parameters dialog box for the Unit Delay block has an **Initial conditions** parameter.



For a block that supports bus initialization, you can replace the default value of `0` using one of these approaches:

- “MATLAB Structure for Initialization” on page 76-62
- “MATLAB Variable for Initialization” on page 76-62
- “Simulink.Parameter For Initialization” on page 76-63

All three approaches require that you define an initial condition structure (see “Create Initial Condition Structures” on page 76-58). You cannot specify a nonzero scalar value or any other type of value other than `0`, an initial condition structure, or `Simulink.Parameter` object to initialize a bus.

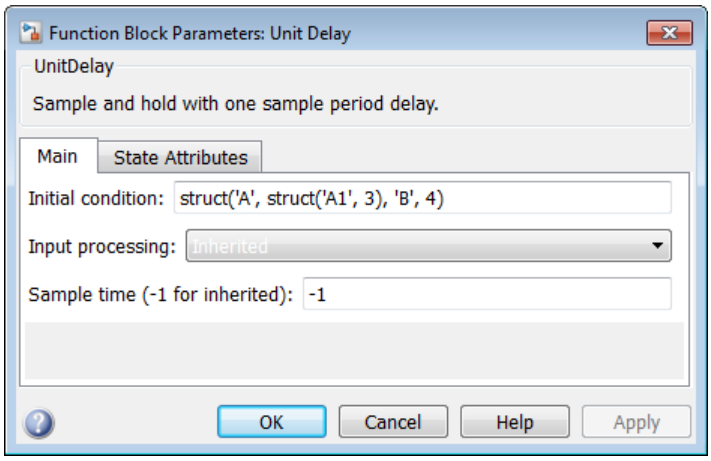
Defining an initial condition structure as a MATLAB variable, rather than specifying the initial condition structure directly in the Block Parameters dialog box offers several advantages, including:

- Reuse of the initial condition structure for multiple blocks
- Using the initial condition structure as a tunable parameter in the generated code

MATLAB Structure for Initialization

You can initialize a bus using a MATLAB structure that explicitly defines the initial conditions for the bus.

For example, in the **Initial conditions** parameter of the Unit Delay block, you could type in a structure.



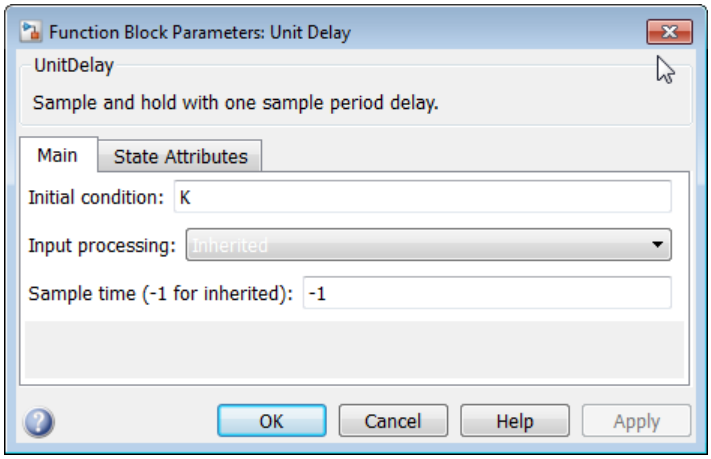
MATLAB Variable for Initialization

You can initialize a bus using a MATLAB variable that you define as an initial condition structure with the appropriate values.

For example, you could define the following partial structure in the base workspace:

```
K = struct('A', struct('A1', 3), 'B', 4);
```

You can then specify the K structure as the **Initial conditions** parameter of the Unit Delay block:



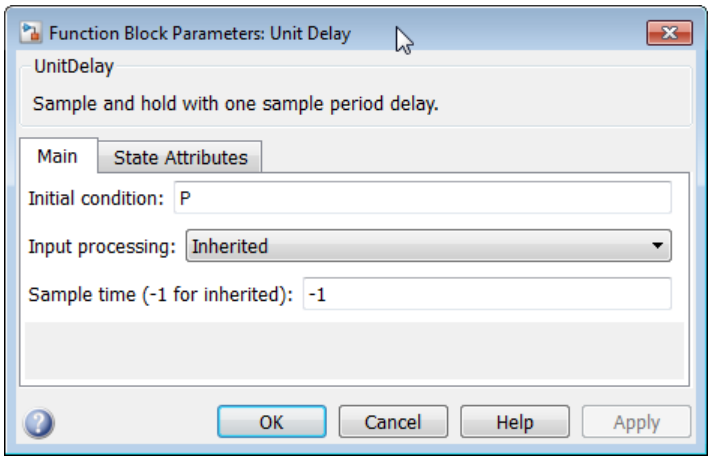
Simulink.Parameter For Initialization

You can initialize a bus using a `Simulink.Parameter` object that uses an initial condition structure for the `Value` property.

For example, you could define the partial structure `P` in the base workspace (reflecting the `ex_bus_initial_conditions` model discussed in the previous section):

```
P = Simulink.Parameter;
P.DataType = 'Bus: Top';
P.Value = Simulink.Bus.createMATLABStruct('Top');
P.Value.A.A1 = 3;
P.Value.B = 5;
```

You can then specify the `P` structure as the **Initial conditions** parameter of the Unit Delay block:



See Also

Blocks
Bus Creator | Bus Selector | Unit Delay

Functions
`Simulink.BlockDiagram.addBusToVector` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

Classes
`Simulink.Bus` | `Simulink.BusElement`

Related Examples

- “Load Bus Data to Root-Level Input Ports” on page 70-46
- “Nonvirtual Buses at Model Interfaces” on page 76-55
- “Initialize Arrays of Buses” on page 76-76
- “Virtual Bus” on page 76-2

Combine Buses into an Array of Buses

Tip Simulink provides several techniques for combining signals into a composite signal. For a comparison of techniques, see “Types of Composite Signals” on page 76-2.

What Is an Array of Buses?

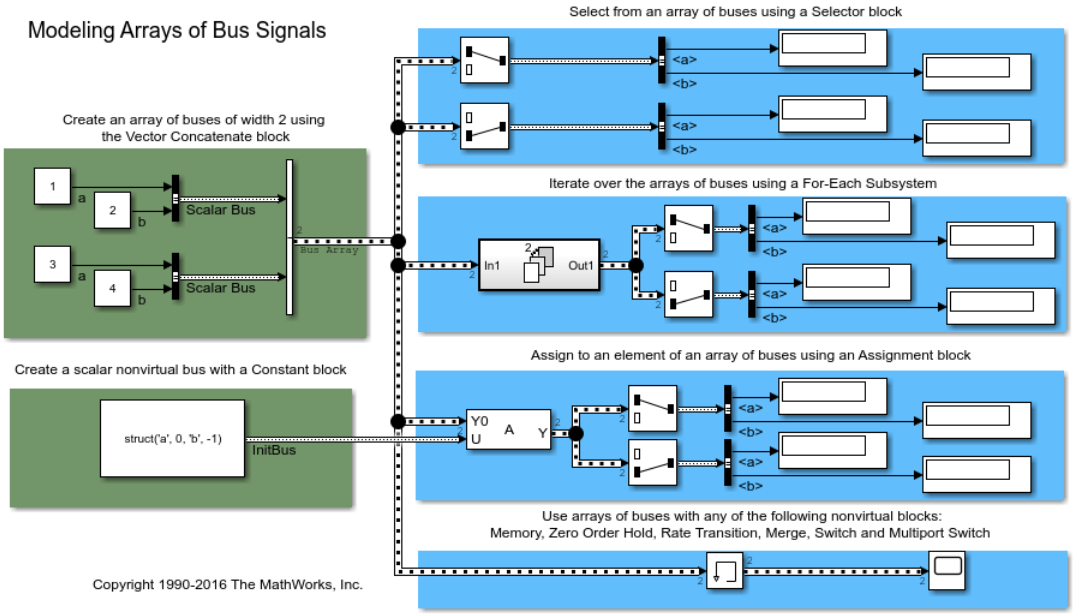
An array of nonvirtual buses is an array whose elements are buses. Each `Bus` object has the same signal name, hierarchy, and attributes for its bus elements.

An example of using an array of buses is to model a multi-channel system, such as a communications system. You can model all the channels using the same `Bus` object, although each of the channels could have a different value.

To use an array of buses:

- Use a `Bus` object as a data type (see “Specify a Bus Object Data Type” on page 67-38).
- Specify dimensions for the bus and bus elements.

For an example of a model that uses an array of buses, open the `sldemo_bus_arrays` model. In this example, the nonvirtual bus input signals connect to a Vector Concatenate or Matrix Concatenate block that creates an array of buses. Here is the diagram after you update it:



The model uses the array of buses with:

- An Assignment block, to assign a bus in the array
- A For Each Subsystem block, to perform iterative processing over each bus in the array
- A Memory block, to output the array of buses input from the previous time step

Benefits of an Array of Buses

Use an array of buses to:

- Represent structured data compactly.
  - Reduce model complexity.
  - Reduce maintenance by centralizing algorithms used for processing multiple buses.
- Streamline iterative processing of multiple buses of the same type, for example, by using a For Each Subsystem with the array of buses.
- Simplify changing the number of buses, without your having to restructure the rest of the model or make updates in multiple places in the model.
- Use built-in blocks, such as the Assignment or Selector blocks, to manipulate arrays of buses just like arrays of any other type. Using an array of buses avoids the need for you to create custom S-functions to manage packing and unpacking structure signals.
- Use the combined bus data across subsystem boundaries, model reference boundaries, and into or out of a MATLAB Function block.
- Keep all the logic in the Simulink model, rather than splitting the logic between C code and the Simulink model. This approach supports integrated consistency and correctness checking, maintaining metadata in the model, and avoids the need to manage model components in two different environments.
- Generate code that has an array of C structures, which you can integrate with legacy C code that uses arrays of structures. This approach simplifies indexing into an array for Simulink computations, using a `for` loop on indexed structures.

Define an Array of Buses

For information about the kinds of buses that you can combine into an array of buses, see “Bus Requirements” on page 76-70.

To define an array of buses, use a Concatenate block. The table describes the array of buses input requirements and output for each of the Vector Concatenate and the Matrix Concatenate versions of the Concatenate block.

Block	Bus Signal Input Requirement	Output
Vector Concatenate	Vectors, row vectors, or columns vectors	If any of the inputs are row or column vectors, output is a row or column vector.
Matrix Concatenate	Signals of any dimensionality (scalars, vectors, and matrices)	Trailing dimensions are assumed to be 1 for lower dimensionality inputs. Concatenation is on the dimension that you specify with the **Concatenate dimension** parameter.

Note Do not use a Mux block or a Bus Creator block to define an array of buses. Instead, use a Bus Creator block to create scalar buses.

1. Define *one* `Bus` object for *all* of the buses that you want to combine into an array of buses. For information about defining `Bus` objects, see “Specify Bus Properties with Simulink.Bus Objects” on page 76-44.

   The `sldemo_bus_arrays` model defines an `sldemo_bus_arrays_busobject` `Bus` object, which both of the Bus Creator blocks use for the input buses (`Scalar Bus`) for the array of buses.

2. Add a Vector Concatenate or Matrix Concatenate block to the model and open the Block Parameters dialog box.

   The `sldemo_bus_arrays_busobject` model uses a Vector Concatenate block, because the inputs are scalars.

3. Set the **Number of inputs** parameter to be the number of buses that you want to be in the array of buses.

   The block icon displays the number of input ports that you specify.

4. Set the **Mode** parameter to match the type of the input bus data.

   In the `sldemo_bus_arrays` model, the input bus data is scalar, so the **Mode** setting is `Vector`.

5. If you use a Matrix Concatenation block, set the **Concatenate dimension** parameter to specify the output dimension along which to concatenate the input arrays. Enter one of the following values:

   - 1 — concatenate input arrays vertically
   - 2 — concatenate input arrays horizontally
   - A higher dimension than 2 — perform multidimensional concatenation on the inputs

6. Connect to the Concatenate block all the buses that you want to be in the array of buses.

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 buses, 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.

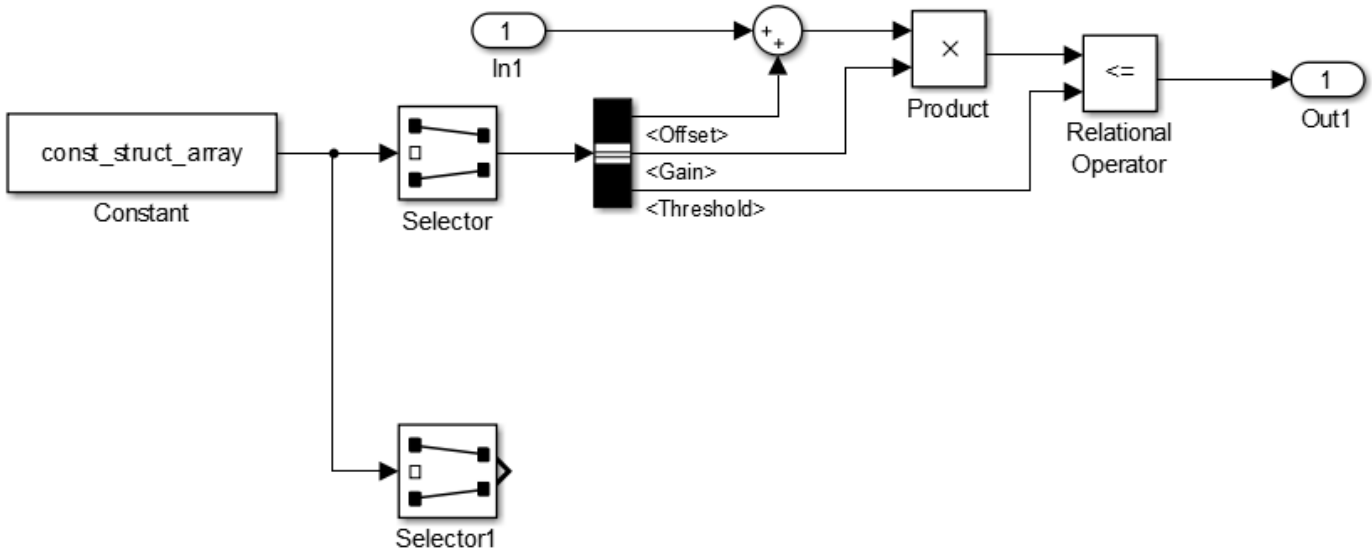
   The output of the Constant block is a single scalar bus of type `ParamBus`. The structure variable `const_param_struct` specifies the constant value in the block.

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`.

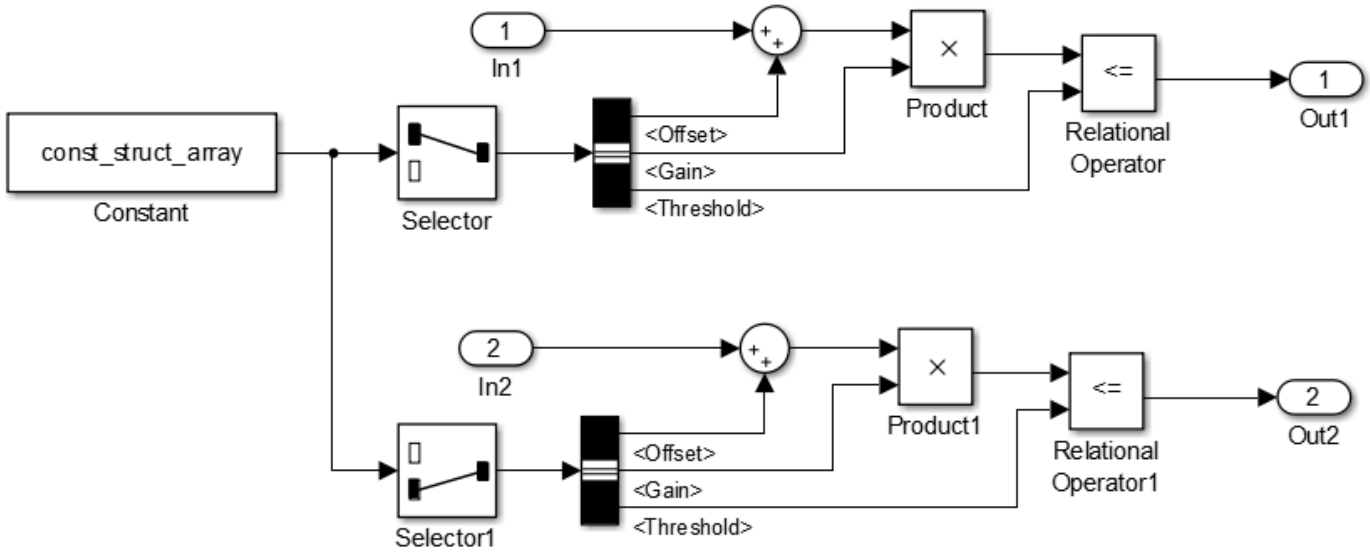
5 Add two Selector blocks to the model, and connect the Constant block as shown.



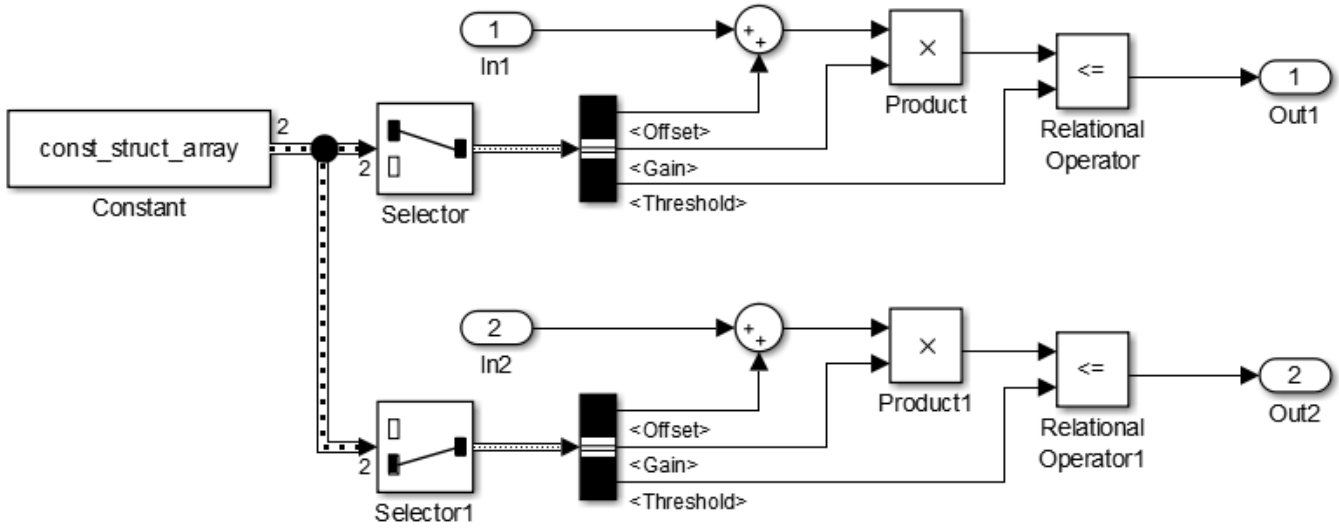
6 In the Selector block dialog box, set **Index** to 1 and **Input port size** to 2.

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.

See Also

Blocks
Matrix Concatenate | Vector Concatenate

Related Examples

- “Use Arrays of Buses in Models” on page 76-70
- “Work with Arrays of Buses” on page 76-73
- “Convert Models to Use Arrays of Buses” on page 76-79
- “Repeat an Algorithm Using a For Each Subsystem” on page 76-81
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Generate Code for Nonvirtual Buses” on page 76-101
- “Types of Composite Signals” on page 76-2
- “Blocks That Support Arrays of Buses” on page 76-70

Use Arrays of Buses in Models

Array of Buses Requirements and Limitations

Bus Requirements

All buses combined into an array of buses must:

- Be nonvirtual
- Have the same bus type (that is, same name, hierarchies, and attributes for the bus elements)
- Have no variable-size signals or frame-based signals

Blocks That Support Arrays of Buses

These blocks support arrays of buses:

- Virtual blocks (see “Nonvirtual and Virtual Blocks” on page 36-2)
- These nonvirtual blocks:
  - Data Store Memory
  - Data Store Read
  - Data Store Write
  - Memory
  - Merge
  - Multiport Switch
  - Rate Transition
  - Switch
  - Unit Delay
  - Zero-Order Hold
- Assignment
- MATLAB Function
- Matrix Concatenate
- Selector
- Signal Conversion
- Vector Concatenate
- Width

Note You can use an array of buses as an input to an In Bus Element block, but you cannot use that block to select individual buses. The block passes through the whole array of buses.

Block Parameter Settings

Using an array of buses with some blocks requires specific block parameter settings.

This information is also in the reference pages for each of these blocks. For usage information for bus-related blocks, see “Work with Arrays of Buses” on page 76-73.

Block	Block Parameters Settings
Memory	**Initial condition** — You can specify this parameter with: • The value `0`. In this case, all the individual signals in the array of buses use the initial value `0`. • An array of structures that specifies an initial condition for each of the individual signals in the array of buses. • A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

Merge	• **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 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.

Multiport Switch	**Number of data ports** — Set to a value of 2 or greater.
Signal Conversion	**Output** — Set to `Signal copy`.
Switch	**Threshold** — Specify a scalar threshold.

Structure Parameter Requirements

To initialize an array of buses with structure parameters, you can use:

- The number `0`. In this case, all the individual signals in the array of buses use the initial value `0`.
- A scalar `struct` that represents the same hierarchy of fields and field names as the bus type. In this case, the scalar structure expands to initialize each of the individual signals in the array of buses.
- An array of structures that specifies an initial value for each of the individual signals in the array of buses.

  If you use an array of structures, all the structures in the array must have the same hierarchy of fields. Each field in the hierarchy must have the same characteristics across the array:

  - Field name
  - Numeric data type, such as `single` or `int32`
  - Complexity
  - Dimensions

You cannot use partial structures.

For more information about specifying initial conditions for buses, see “Initialize Arrays of Buses” on page 76-76.

Signal Logging Limitation

Simulink software does not log arrays of buses inside referenced models in rapid accelerator mode.

Stateflow Limitations

Stateflow action language does not support arrays of buses.

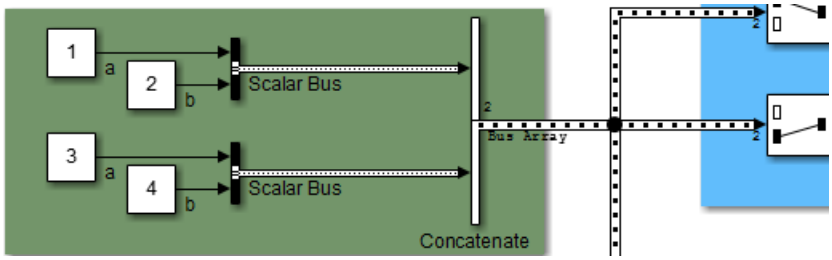
Bus Creator Blocks

A Bus Creator block can accept an array of buses as input, but cannot have an array of buses as output.

Signal Line Style

After model simulation, the line style for the array of buses is a thicker version of the signal line style for a nonvirtual bus.

For example, in the `sldemo_bus_arrays` model, the `Scalar Bus` signal is a nonvirtual bus, and the `Bus Array` output signal of the Concatenate block is an array of buses.



See Also

Related Examples

- “Combine Buses into an Array of Buses” on page 76-64
- “Work with Arrays of Buses” on page 76-73
- “Convert Models to Use Arrays of Buses” on page 76-79
- “Repeat an Algorithm Using a For Each Subsystem” on page 76-81
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Access Array of Buses Signal Logging Data” on page 72-19
- “Generate Code for Nonvirtual Buses” on page 76-101

Work with Arrays of Buses

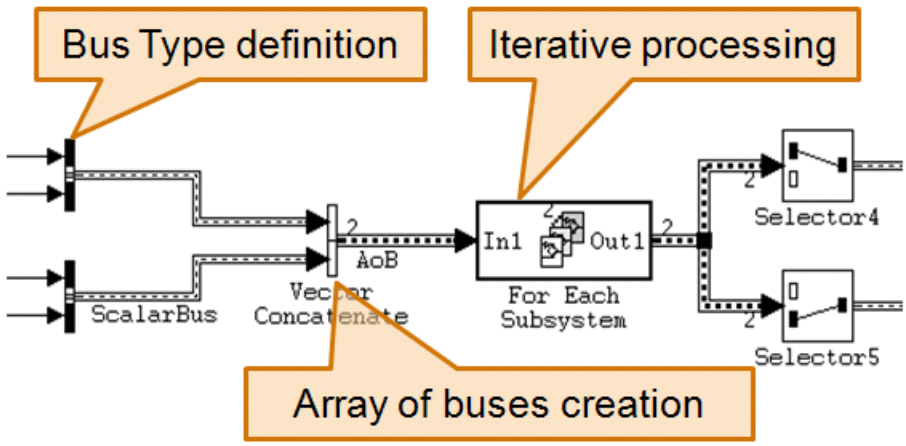
Set Up Model for Arrays of Buses

Setting up a model to use an array of buses usually involves these basic tasks:

1. Define the array of buses (see “Define an Array of Buses” on page 76-65).
2. Add a subsystem for performing iterative processing on each element of the array of buses. For example, use a For Each Subsystem block or an Iterator block. Connect the array of buses from the Concatenate block to the iterative processing subsystem. See “Perform Iterative Processing” on page 76-73.
3. Model your scalar algorithm within the iterative processing subsystem (for example, a For Each subsystem).
   a. Operate on the array of buses (using Selector and Assignment blocks).
   b. Use the Bus Selector and Bus Assignment blocks to select elements from, or assign elements to, a scalar bus within the subsystem.

   See “Assign Values into an Array of Buses” on page 76-74 and “Select Bus Elements from an Array of Buses” on page 76-75.
4. Optionally, import or log array of buses data. See “Import Array of Buses Data” on page 76-76 and “Log Arrays of Buses” on page 76-76

The resulting model includes these components.



Perform Iterative Processing

You can perform iterative processing on the bus data of an array of buses using blocks such as a For Each Subsystem block, a While Iterator Subsystem block, or a For Iterator Subsystem block. You can use one of these blocks to perform the same kind of processing on:

- Each bus in the array of buses
- A selected subset of buses in the array of buses

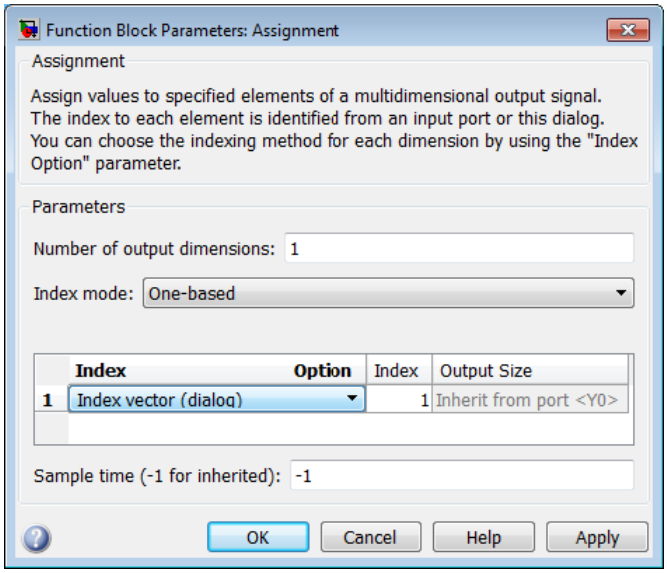
Assign Values into an Array of Buses

To assign a value to a signal within an array of buses, use:

1. A Bus Assignment block to assign a value to a bus element
2. An Assignment block to assign the bus to the array of buses

Use an Assignment block to assign values to specified elements in a bus array.

For example, in the `sldemo_bus_arrays` model, the Assignment block assigns the value to the first element of the array of buses.



To assign bus elements within a bus, use the Bus Assignment block. The input for the Bus Assignment block must be a scalar bus.

Assign into Arrays of Buses

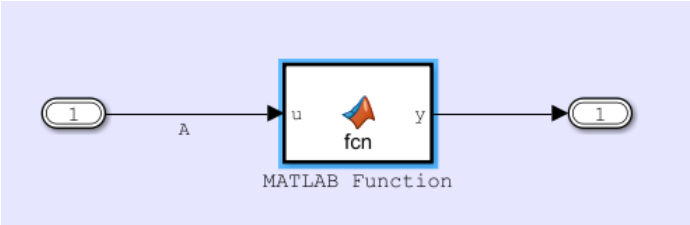
You can use a Bus Assignment block to assign or fully replace a nested bus that is an array of buses. To assign the data for a nested bus inside an array of buses or to make a partial assignment to certain elements with the array of buses, you can use a MATLAB Function block.

For example, suppose that you have this bus structure:

Variable	Type	element	Modification
A	Parent	1	
c	boolean	1	0
children	Child	10	
a	int32	5	0
b	single	1	

The bus has a `children` element, which is a sub-bus array. This example shows how to assign to element `c` and to element `a`. The Inport and Outport blocks use the `Parent Bus` object. To define the

assignments, this example uses a MATLAB Function block, because you cannot assign into element `a` using a Bus Assignment or Assignment block.



The MATLAB Function block uses this function code for making the assignments:

```
function y = fcn(u)

y = u;
y.c = false;
for idx = 1:length(y.children)
    y.children(idx).a = int32(zeros(5, 1));
end
```

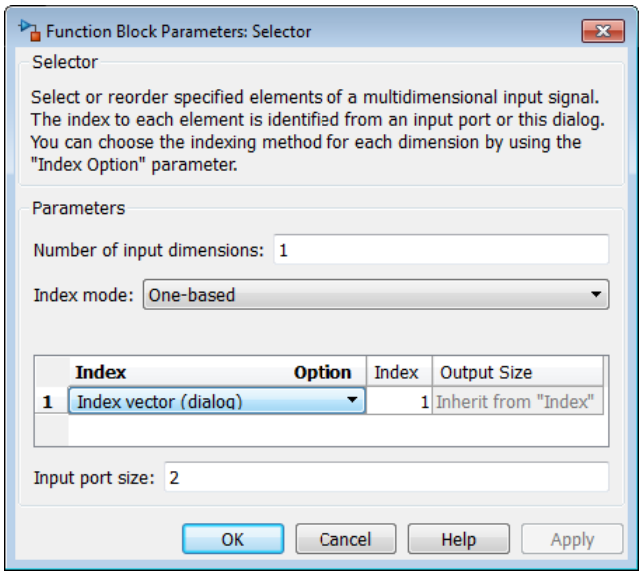
Select Bus Elements from an Array of Buses

To select a signal within an array of buses, use a:

1. Selector block to find the appropriate bus within the array of buses.
2. Bus Selector block to select the signal.

Use a Selector block to select elements of an array of buses. The input array of buses can have any dimension. The output bus of the Selector block is a selected or reordered set of elements from the input array of buses.

For example, the `sldemo_bus_arrays` model uses Selector blocks to select elements from the array of buses that the Assignment and For Each Subsystem blocks outputs. In this example, here is the Block Parameters dialog box for the Selector block that selects the first element:



To select bus elements within a bus, use the Bus Selector block. The input for the Bus Selector block must be a scalar bus.

Import Array of Buses Data

Use a root Inport block to import (load) an array of structures of MATLAB `timeseries` objects for an array of buses. You can import partial data into the array of buses.

For details, see “Import Array of Buses Data” on page 70-49.

You cannot use a From Workspace or From File block to import data for an array of buses.

Log Arrays of Buses

To export an array of buses, mark the signal for signal logging. For more information, see “Save Run-Time Data from Simulation”.

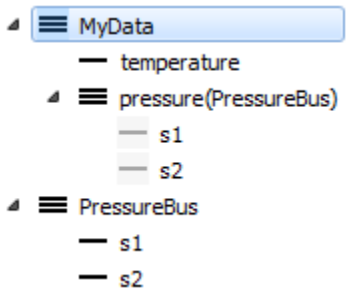
Note Simulink does not log signals inside referenced models in rapid accelerator mode.

To access the signal logging data for a specific signal in an array of buses, navigate through the structure hierarchy and specify the index to the specific signal. For details, see “Access Array of Buses Signal Logging Data” on page 72-19.

Initialize Arrays of Buses

To specify a unique initial value for each of the individual signals in an array of buses, you can use an array of initial condition structures. Each structure in the array initializes one of the buses.

Here is an example that shows how to initialize an array of buses. Suppose that you define the bus types `MyData` and `PressureBus`.



Suppose that you set the data type of the signal element `temperature` to `int16`, and the data type of the elements `s1` and `s2` to `double`.

To specify initial conditions for an array of buses, you can create a variable whose value is an array of initial condition structures.

```
initValues(1).temperature = int16(5);
initValues(1).pressure.s1 = 9.87;
initValues(1).pressure.s2 = 8.71;
```

```
initValues(2).temperature = int16(20);
initValues(2).pressure.s1 = 10.21;
initValues(2).pressure.s2 = 9.56;

initValues(3).temperature = int16(35);
initValues(3).pressure.s1 = 8.98;
initValues(3).pressure.s2 = 9.17;
```

The variable `initValues` provides initial conditions for a signal that is an array of three buses. You can use `initValues` to specify the **Initial condition** parameter of a block such as Unit Delay.

Alternatively, you can use a single scalar structure to specify the same initial conditions for all the buses in the array.

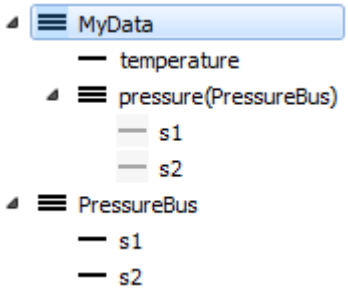
```
initStruct.temperature = int16(15);
initStruct.pressure.s1 = 10.32;
initStruct.pressure.s2 = 9.46;
```

If you specify `initStruct` in the **Initial condition** parameter of a block, each bus in the array uses the same initial value, `15`, for the signal element `temperature`. Similarly, the buses use the initial value `10.32` for the element `pressure.s1` and the value `9.46` for the element `pressure.s2`.

To create an array of structures for a bus that uses a large hierarchy of signal elements, consider using the function `Simulink.Bus.createMATLABStruct`.

This example shows how to initialize a nested array of buses. Create an initial condition structure for a complicated signal hierarchy that includes nested arrays of buses.

1. In the Bus Editor, create the `Bus` objects `MyData` and `PressureBus`.



2. In the hierarchy pane, select the bus element `pressure`. Set the **Dimensions** property to `[1 3]`.
3. Create an array of four initialization structures by using the function `Simulink.Bus.createMATLABStruct`. Store the array in the variable `initStruct`. Initialize all the individual signals to the ground value, `0`.

   ```
   initStruct=Simulink.Bus.createMATLABStruct('MyData',[],[1 4]);
   ```
4. In the base workspace, double-click the variable `initStruct` to view it in the variable editor.

   The four structures in the array each have the fields `temperature` and `pressure`.
5. To inspect a `pressure`, double-click one of the fields.

   The value of each of the four `pressure` fields is an array of three substructures. Each substructure has the fields `s1` and `s2`.

6 To provide unique initialization values for the signals in an array of buses, you can specify the values manually using the variable editor.

Alternatively, you can write a script. For example, to access the field `s1` of the second substructure `pressure` in the third structure of `initStruct`, use this code:

```
initStruct(3).pressure(2).s1 = 15.35;
```

Code Generation

Code generation for arrays of buses produces structures with a specific format. See “Code Generation for Arrays of Buses” on page 76-106.

See Also

Related Examples

- “Combine Buses into an Array of Buses” on page 76-64
- “Use Arrays of Buses in Models” on page 76-70
- “Convert Models to Use Arrays of Buses” on page 76-79
- “Repeat an Algorithm Using a For Each Subsystem” on page 76-81
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Access Array of Buses Signal Logging Data” on page 72-19
- “Generate Code for Nonvirtual Buses” on page 76-101

Convert Models to Use Arrays of Buses

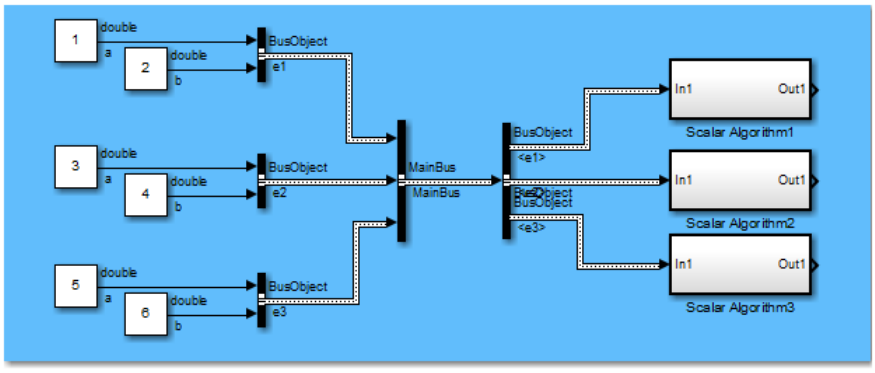
There are several reasons to convert a model to use an array of buses (see “Benefits of an Array of Buses” on page 76-65). For example:

- The model was developed before Simulink supported arrays of buses (introduced in R2010b), and the model contains many subsystems that perform the same kind of processing.
- The model has grown in complexity.

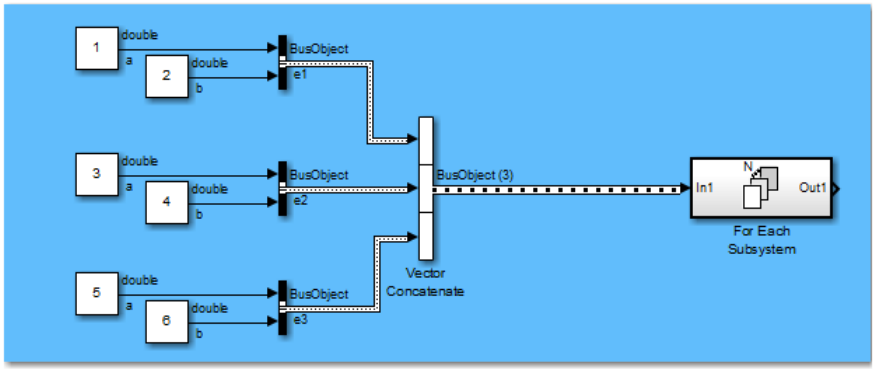
General Conversion Approach

Here is a general approach for converting a model that contains buses to a model that uses an array of buses. The method that you use depends on your model. For details about these techniques, see “Combine Buses into an Array of Buses” on page 76-64 and “Use Arrays of Buses in Models” on page 76-70.

This workflow refers to a stylized example model. The example shows the original modeling pattern and a new modeling pattern that uses an array of buses.



Original modeling pattern



New modeling pattern

In the original modeling pattern:

- The target bus to be converted is named `MainBus`, and it has three elements, each of type `BusObject`.

- The `ScalarAlgorithm1`, `ScalarAlgorithm2`, and `ScalarAlgorithm3` subsystems encapsulate the algorithms that operate on each of the bus elements. The subsystems all have the same content.
- A Bus Selector block picks out each element of `MainBus` to drive the subsystems.

The construction in the original modeling pattern is inefficient for two reasons:

- A copy of the subsystem that encapsulates the algorithm is made for each element of the bus that is to be processed.
- Adding another element to `MainBus` involves changing the `Bus` object definition and the Bus Selector block, and adding a subsystem. Each of these changes is a potential source of error.

To convert the original modeling pattern to use an array of buses:

1. Identify the target bus and associated algorithm that you want to convert. Typically, the target bus is a bus of buses, where each element bus is of the same type.

   - The bus that you convert must be a nonvirtual bus. If all elements of the target bus have the same sample time (or if the sample time is inherited), you can convert a virtual bus to a nonvirtual bus.
   - The target bus cannot have variable-dimensioned and frame-based elements.
2. Use a Concatenate block to convert the original bus of buses to an array of buses.

   In the example, the new modeling pattern uses a Vector Concatenate block to replace the Bus Creator block that creates the `MainBus` signal. The output of the Vector Concatenate block is an array of buses, where the type of the bus is `BusObject`. The new model eliminates the wrapper bus (`MainBus`).
3. Replace all identical copies of the algorithm subsystem with a single For Each subsystem that encapsulates the scalar algorithm. Connect the array of buses to the For Each subsystem.

   The new model eliminates the Bus Selector blocks that separate out the elements of the `MainBus` signal in the original model.
4. Configure the For Each Subsystem block to iterate over the input array of buses and concatenate the output bus.

   The scalar algorithm within the For Each subsystem cannot have continuous states. For additional limitations, see the For Each Subsystem block documentation.

See Also

Related Examples

- “Combine Buses into an Array of Buses” on page 76-64
- “Use Arrays of Buses in Models” on page 76-70
- “Work with Arrays of Buses” on page 76-73
- “Repeat an Algorithm Using a For Each Subsystem” on page 76-81

Repeat an Algorithm Using a For Each Subsystem

If you repeat algorithms in a diagram by copying and pasting blocks and subsystems, maintaining the model can become difficult. Individual signal lines and subsystems can crowd the diagram, reducing readability and making simple changes difficult. Variables can also crowd workspaces, reducing model portability. A model can develop these efficiency issues as you add to the design over time.

To repeat an algorithm, you can iterate the algorithm over signals, subsystems, and parameters that are grouped into arrays and structures. This example shows how to convert an inefficiently complex repetitive algorithm into a compact form that is easier to manage.

Explore Example Model

1. Open the example model `ex_repeat_algorithm`. The model creates about 30 variables in the base workspace.
2. Inspect the subsystem Burner_1_Analysis. This subsystem executes an algorithm by using the base workspace variables as parameters in blocks such as Constant and Discrete-Time Integrator.
3. Inspect the subsystems Burner_2_Analysis and Burner_3_Analysis. All three subsystems execute the same algorithm but use different workspace variables to parameterize the blocks.
4. Inspect the three Analysis_Delay subsystems. These subsystems repeat a different algorithm from the one in the Analysis subsystems.
5. Return to the top level of the model. The Memory blocks delay the input signals before they enter the Analysis_Delay subsystems.
6. Look at the **Data Import/Export** pane of the Configuration Parameters dialog box. The model uses the variables `SensorsInput` and `t` as simulation inputs.

   During simulation, each of the nine columns in the matrix variable `SensorsInput` provides input data for an Inport block at the top level of the model.

Reduce Signal Line Density with Buses

You can use buses to group related signals into a single structured signal, reducing line density and improving model readability.

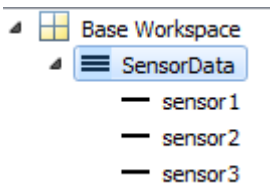
Each subsystem in the example model requires three signal inputs. You can combine each group of three signals into a single bus.

You could modify all the subsystems in the example model to use buses. However, because some of the subsystems are identical, you can delete them and later replace them with For Each Subsystem blocks.

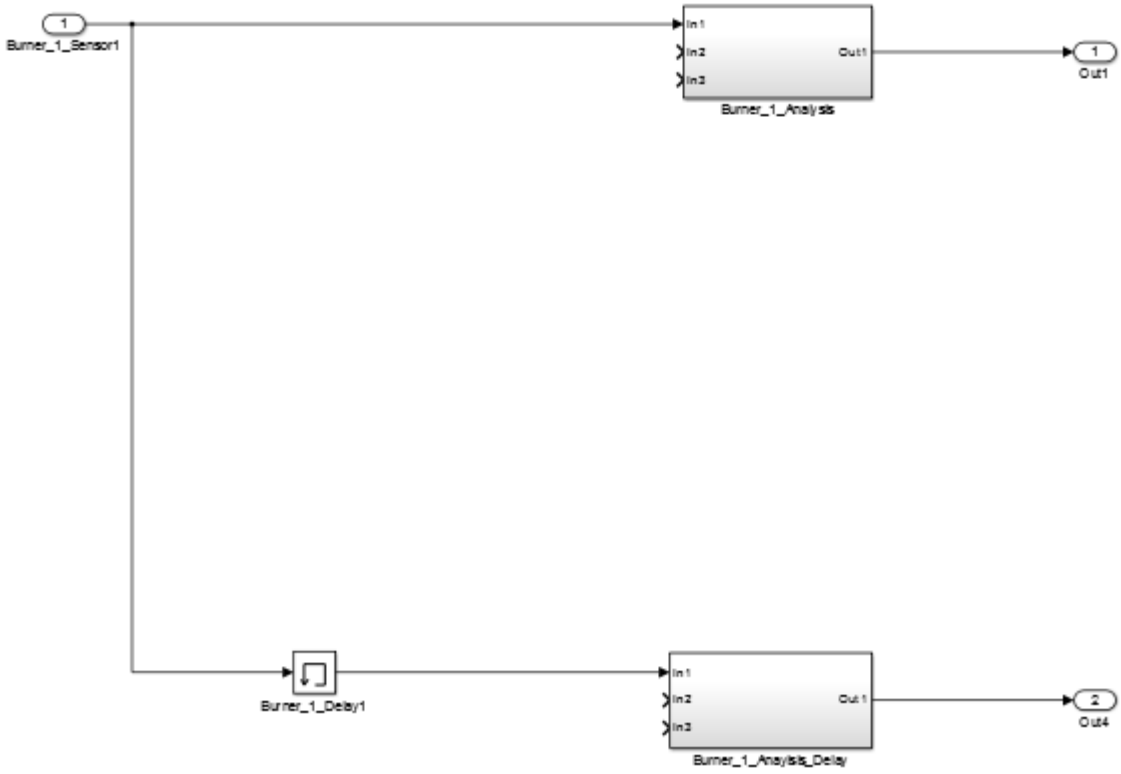
1. Open the Bus Editor.

   ```
   buseditor
   ```

2. Create a bus type `SensorData` with three signal elements: `sensor1`, `sensor2`, and `sensor3`.



3 Delete the blocks as shown in the figure, leaving only the Burner_1_Sensor1 and Burner_1_Delay1 blocks as inputs to the two remaining subsystems.



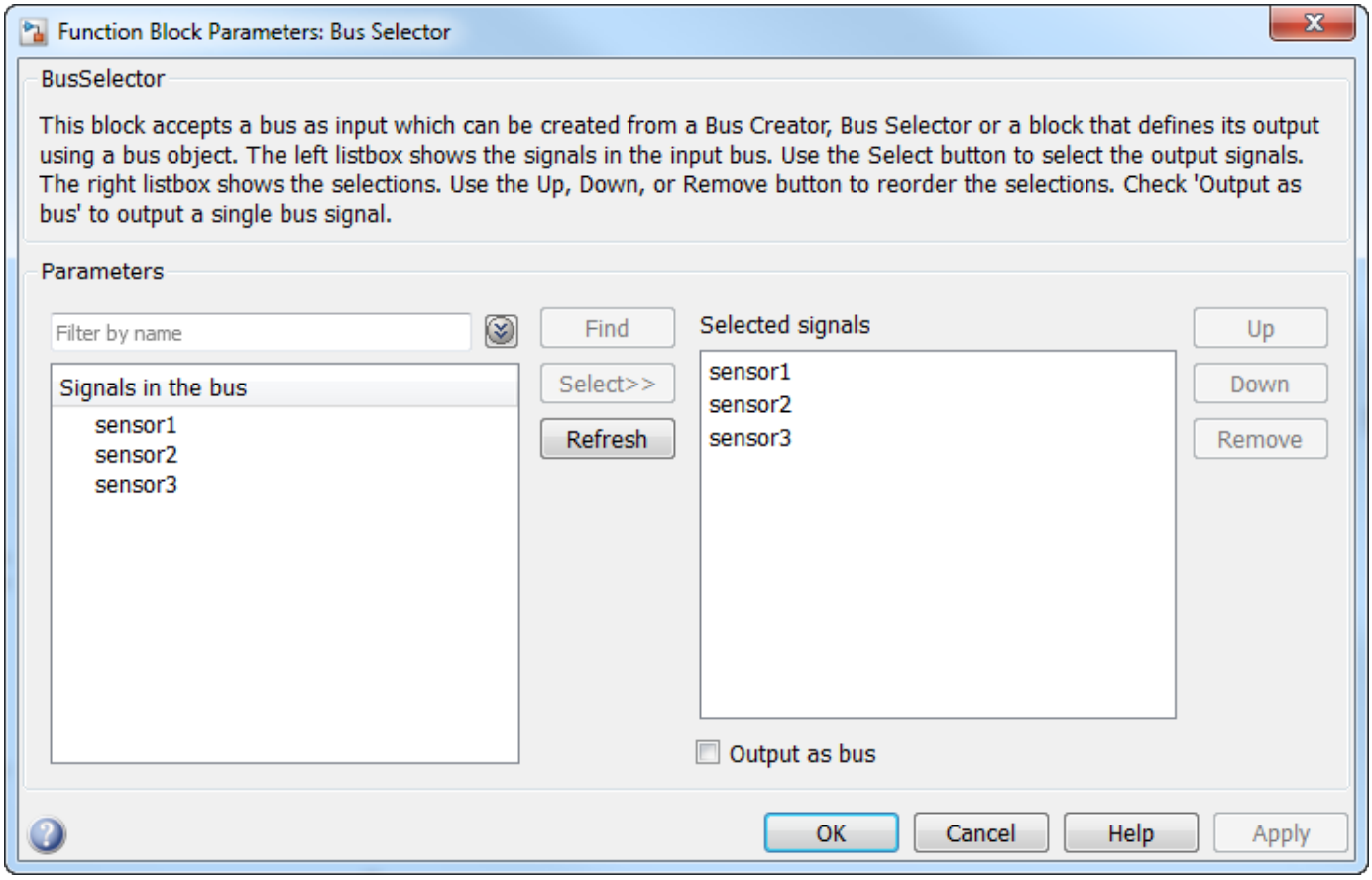
4 On the **Signal Attributes** tab of the Burner_1_Sensor1 Inport block dialog box, set **Data type** to `Bus: SensorData`.

The output of the block is a bus that contains the three signal elements `sensor1`, `sensor2`, and `sensor3`.

5 Open the subsystem Burner_1_Analysis. Delete the signal output lines of the three Inport blocks. Delete the In2 and In3 Inport blocks.

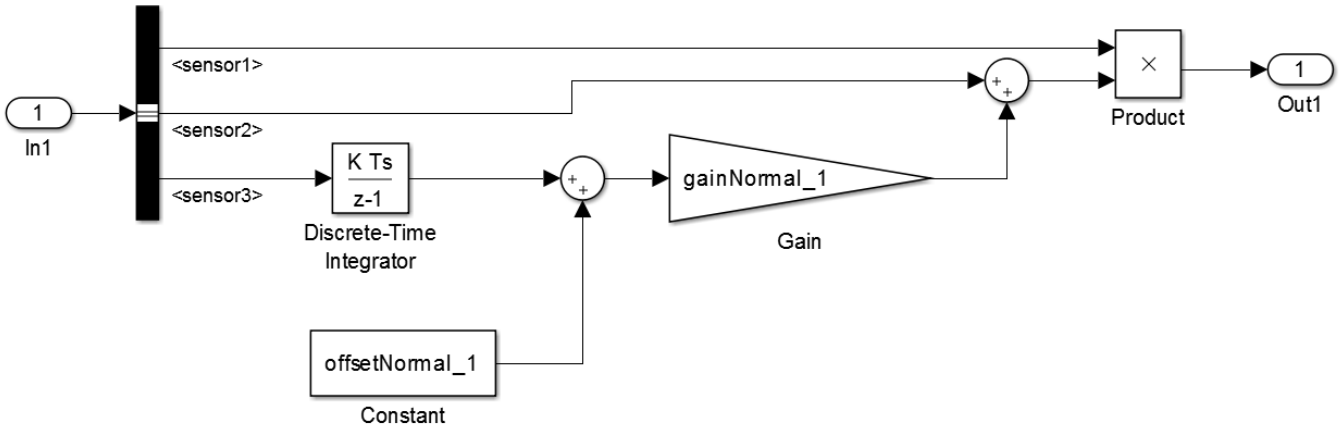
6 Add a Bus Selector block to the right of the In1 Inport block. Connect the Inport block output to the Bus Selector block.

7 In the Bus Selector block dialog box, select the signals `sensor1`, `sensor2`, and `sensor3`.



The Bus Selector block extracts the three signal elements from the input bus. Other blocks in the model can use the extracted signal elements.

8 In the subsystem, connect the blocks as shown.



9 In the subsystem Burner_1_Analysis_Delay, use a Bus Selector block to extract the signals in the bus. Use the same technique as you did in the subsystem Burner_1_Analysis.

Repeat an Algorithm

A For Each Subsystem block partitions an input signal and sequentially executes an algorithm on each partition. For example, if the input to the subsystem is an array of six signals, you can configure the subsystem to execute the same algorithm on each of the six signals.

You can use For Each subsystems to repeat an algorithm in an iterative fashion. This approach improves model readability and makes it easy to change the repeated algorithm .

1. Add two For Each Subsystem blocks to the model. Name one of the subsystems Burner_Analysis. Name the other subsystem Burner_Analysis_Delay.
2. Copy the contents of the subsystem Burner_1_Analysis into the subsystem Burner_Analysis. Before you paste the blocks, delete the Inport and Outport blocks in the For Each subsystem.
3. In the For Each block dialog box in the Burner_Analysis subsystem, select the check box to partition the input `In1`.
4. Copy the contents of the subsystem Burner_1_Analysis_Delay into the subsystem Burner_Analysis_Delay.
5. In the For Each block dialog box in the Burner_Analysis_Delay subsystem, select the check box to partition the input `In1`.
6. At the top level of the model, delete the subsystems Burner_1_Analysis and Burner_1_Analysis_Delay. Connect the new For Each Subsystem blocks in their place.
7. On the **Signal Attributes** tab of the Burner_1_Sensor1 Inport block dialog box, set **Port dimensions** to 3.

   The block output is a three-element array of buses. The For Each subsystems in the model repeat an algorithm for each of the three buses in the array.
8. Create a `Simulink.SimulationData.Dataset` object that the Inport block can use to import the simulation data. You can use this code to create the object and store it in the variable `SensorsInput`.

```
% First, create an array of structures whose field values are
% timeseries objects

for i = 1:3 % Burner number

    % Sensor 1
    eval(['tempInput(1,' num2str(i) ').sensor1 = ' ...
        'timeseries(SensorsInput(:,' num2str(3*(i-1)+1) '),t);'])

    % Sensor 2
    eval(['tempInput(1,' num2str(i) ').sensor2 = ' ...
        'timeseries(SensorsInput(:,' num2str(3*(i-1)+2) '),t);'])

    % Sensor 3
    eval(['tempInput(1,' num2str(i) ').sensor3 = ' ...
        'timeseries(SensorsInput(:,' num2str(3*(i-1)+3) '),t);'])

end

% Create the Dataset object.

SensorsInput = Simulink.SimulationData.Dataset;
SensorsInput = addElement(SensorsInput,tempInput,'element1');
```

```
clear tempInput t i
```

The code first creates a variable `tempInput` that contains an array of three structures. Each structure has three fields that correspond to the signal elements in the bus type `SensorData`, and each field stores a MATLAB `timeseries` object. Each `timeseries` object stores one of the nine columns of data from the variable `SensorsInput`, which stores the simulation input data for each of the sensors.

The code then overwrites `SensorsInput` with a new `Simulink.SimulationData.Dataset` object and adds `tempInput` as an element of the object.

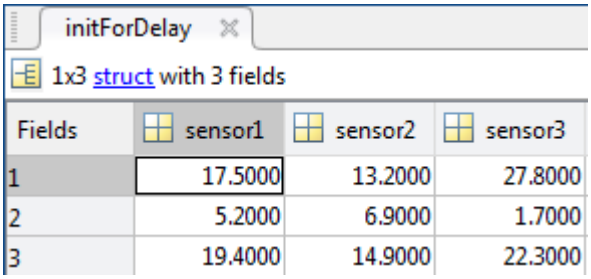
9 Set the **Input** configuration parameter to `SensorsInput`.

Since `SensorsInput` provides simulation input data in the form of `timeseries` objects, you do not need to specify a variable that contains time data.

10 Create an array of structures that initializes the remaining Memory block, and store the array in the variable `initForDelay`. Specify the structure fields with the values of the existing initialization variables such as `initDelay_1_sensor1`.

```
for i = 1:3 % Burner number

    % Sensor 1
    eval(['initForDelay(' num2str(i) ').sensor1 = ' ...
        'initDelay_' num2str(i) '_sensor1;'])

    % Sensor 2
    eval(['initForDelay(' num2str(i) ').sensor2 = ' ...
        'initDelay_' num2str(i) '_sensor2;'])

    % Sensor 3
    eval(['initForDelay(' num2str(i) ').sensor3 = ' ...
        'initDelay_' num2str(i) '_sensor3;'])
end
```

To view the contents of the new variable `initForDelay`, double-click the variable name in the base workspace. The variable contains an array of three structures that each has three fields: `sensor1`, `sensor2`, and `sensor3`.



11 In the Memory block dialog box, set **Initial condition** to `initForDelay`.

The Memory block output is an array of buses that requires initialization. Each signal element in the array of buses acquires an initial value from the corresponding field in the array of structures.

Organize Parameters into Arrays of Structures

The base workspace contains many variables that the example model uses for block parameters. To reduce the number of workspace variables, package them into arrays of structures, and use the individual structure fields to specify block parameters.

A For Each Subsystem block can partition an array of values that you specify as a mask parameter. Each iteration of the subsystem uses a single partition of the array to specify block parameters. If you specify the parameter as an array of structures, each iteration of the subsystem can use one of the structures in the array.

1. Create an array of structures that parameterizes the Burner_Analysis For Each subsystem, and store the array in the variable `paramsNormal`. Specify the structure fields by using the values of existing parameter variables such as `gainNormal_1`, `offsetNormal_1`, and `initDelayed_1`.

   ```
   for i = 1:3
       eval(['paramsNormal(' num2str(i) ').gain = gainNormal_' num2str(i) ';'])
       eval(['paramsNormal(' num2str(i) ').offset = offsetNormal_' num2str(i) ';'])
       eval(['paramsNormal(' num2str(i) ').init = initNormal_' num2str(i) ';'])
   end
   ```

   The variable contains an array of three structures that each has three fields: `gain`, `offset`, and `init`.

2. In the model, right-click the Burner_Analysis For Each subsystem and select **Mask > Create Mask**.
3. On the **Parameters & Dialog** pane of the dialog box, under **Parameter**, click **Edit**. For the new mask parameter, set **Prompt** to `Parameter structure` and **Name** to `paramStruct`. Click **OK**.
4. In the mask for the Burner_Analysis subsystem, set **Parameter structure** to `paramsNormal`.
5. Open the subsystem. In the For Each block dialog box, on the **Parameter Partition** pane, select the check box to partition the parameter `paramStruct`. Set **Partition dimension** to 2.
6. For the blocks in the subsystem, set these parameters.

Block	Parameter Name	Parameter Value
Gain	**Gain**	`paramStruct.gain`
Discrete-Time Integrator	**Initial condition**	`paramStruct.init`
Constant	**Constant value**	`paramStruct.offset`

7. Create an array of structures that parameterizes the Burner_Analysis_Delay For Each subsystem, and store the array in the variable `paramsForDelay`.

   ```
   for i = 1:3
       eval(['paramsForDelay(' num2str(i) ').gain = gainDelayed_' num2str(i) ';'])
       eval(['paramsForDelay(' num2str(i) ').offset = offsetDelayed_' num2str(i) ';'])
       eval(['paramsForDelay(' num2str(i) ').init = initDelayed_' num2str(i) ';'])
   end
   ```

8. At the top level of the model, right-click the Burner_Analysis_Delay For Each subsystem and select **Mask > Create Mask**.
9. On the **Parameters & Dialog** pane of the dialog box, under **Parameter**, click **Edit**. For the new mask parameter, set **Prompt** to `Parameter structure` and **Name** to `paramStruct`. Click **OK**.
10. In the mask for the For Each Subsystem block, set **Parameter structure** to `paramsForDelay`.
11. Open the subsystem. In the For Each block dialog box, on the **Parameter Partition** pane, select the check box to partition the parameter `paramStruct`. Set **Partition dimension** to 2.

12 For the blocks in the subsystem, set these parameters.

Block	Parameter Name	Parameter Value
Gain	**Gain**	`paramStruct.gain`
Discrete-Time Integrator	**Initial condition**	`paramStruct.init`
Constant	**Constant value**	`paramStruct.offset`

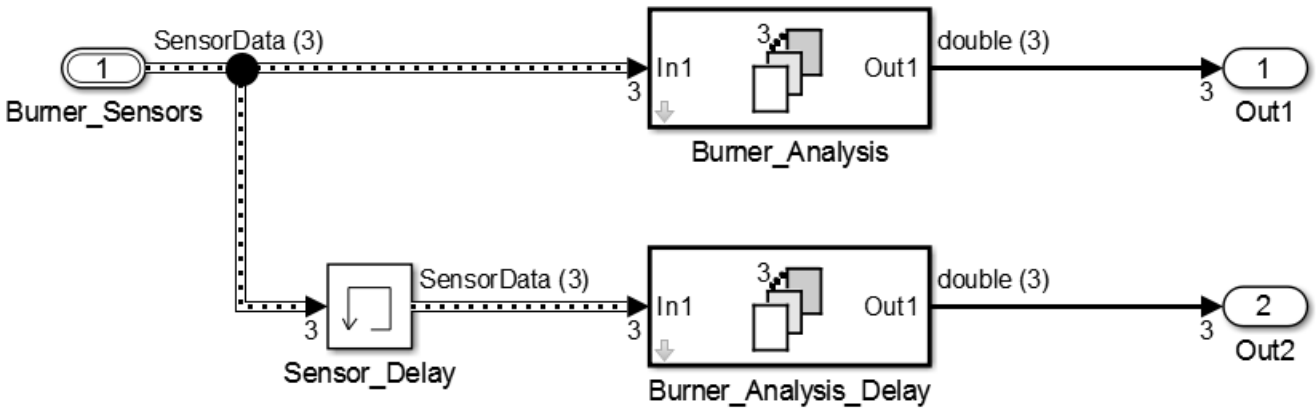
13 Clear the unnecessary variables from the base workspace.

```
% Clear the old parameter variables that you replaced
% with arrays of structures
clear -regexp _

% Clear the iteration variables
clear i
```

The model requires few variables in the base workspace.

Inspect the Converted Model

To view the new signal and subsystem organization, update the diagram.



The model input is an array of three buses. The model uses two For Each subsystems to execute the two algorithms on each of the three buses in the input array.

In the base workspace, arrays of structures replace the many variables that the model used. Mathematically, the modified model behaves the same way it did when you started because the arrays of structures contain the values of all the old variables.

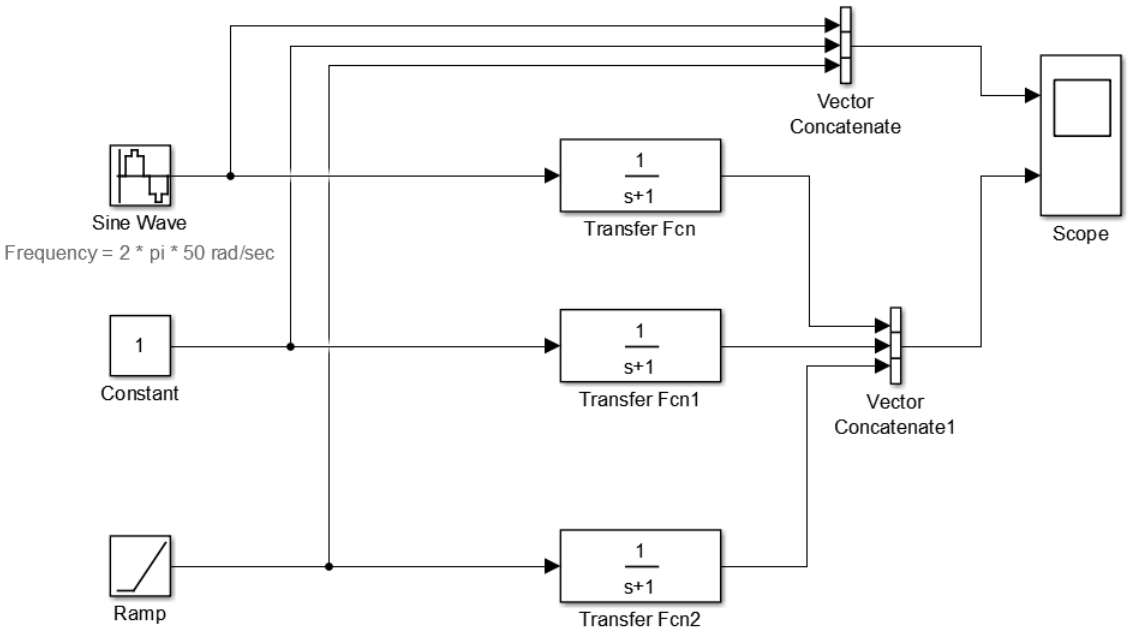
Tip You can log nonbus signals in a For Each subsystem. However, you cannot use signal logging for buses or arrays of buses from within a For Each subsystem. Either use a Bus Selector block to select the bus element signals that you want to log or add an Outport block outside of the subsystem and then log that signal. For details, see “Log Signals in For Each Subsystems” on page 72-71.

Examples of Working with For Each Subsystems

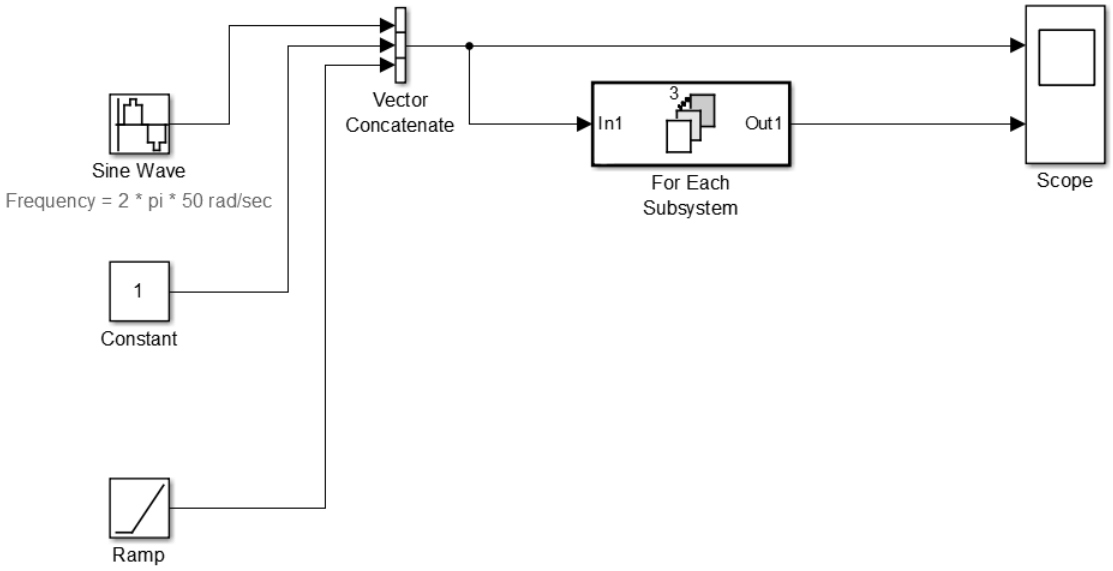
Vectorize Algorithms Using For Each Subsystems

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.

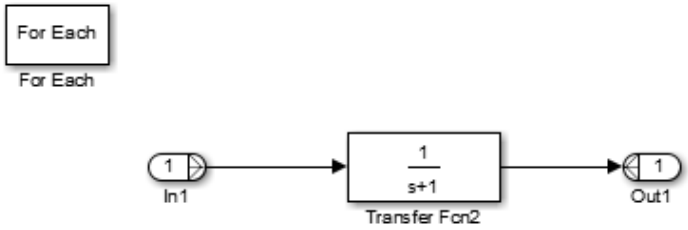
This model uses identical Transfer Fcn blocks to independently process each element of the input sine wave signal. A Vector Concatenate block concatenates the resulting output signals. This repetitive process is graphically complex and difficult to maintain. Adding another element to the signal also requires significant reworking of the model.



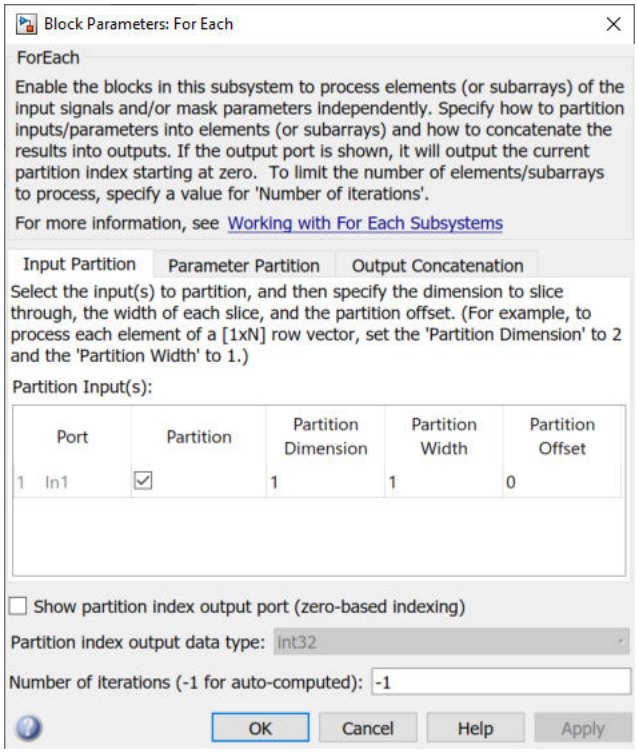
You can simplify this model by replacing the repetitive operations with a single For Each Subsystem block.



The For Each subsystem block 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 **Partition Width** parameters on the For Each block are both set to 1 for the input.

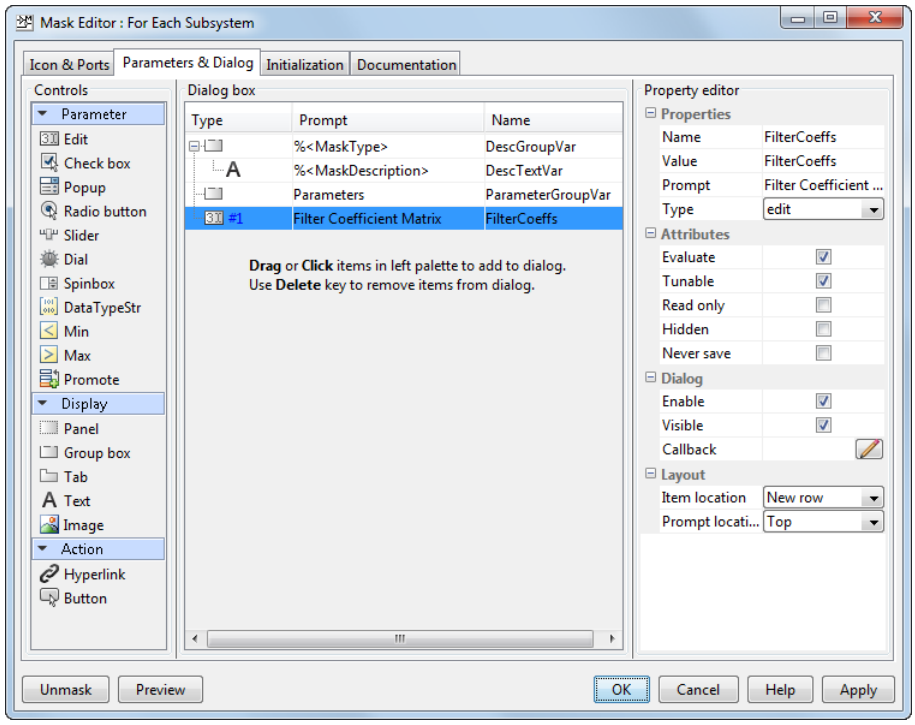


You can scale up this approach to add more signals without having to change the model significantly. This approach is easily scalable and graphically simpler.

Model Parameter Variation Without Changing Model Structure

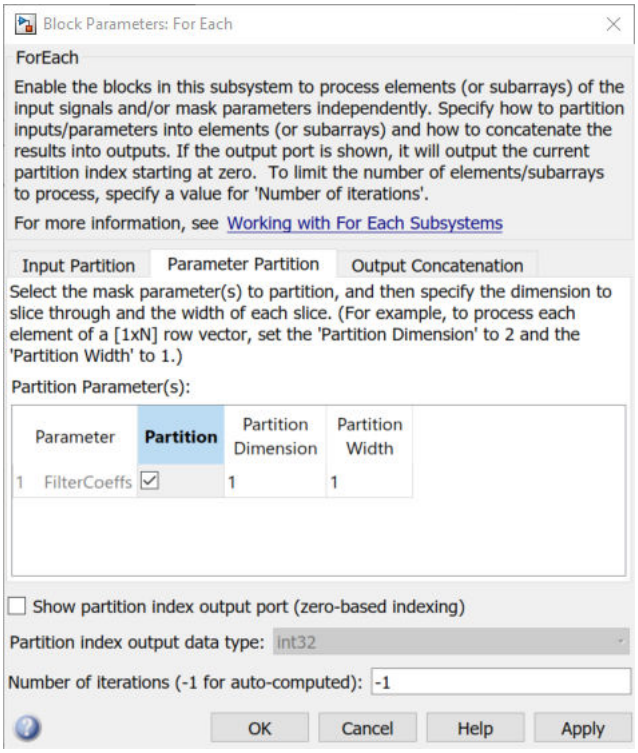
This example shows how to model parameter variation in an algorithm. It uses the For Each Subsystem partitioning model from the above example and creates different filters for each input signal while retaining model simplicity. An array of filter coefficients is fed to the For Each subsystem block as a mask parameter marked for partitioning. In each iteration of the For Each subsystem block, a partition of the filter coefficient array 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” on page 39-6.



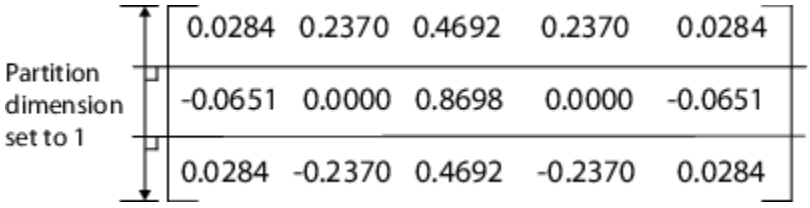
2 Open the For Each subsystem block. Inside the subsystem, open the For Each block dialog box.

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 block, 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 block 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

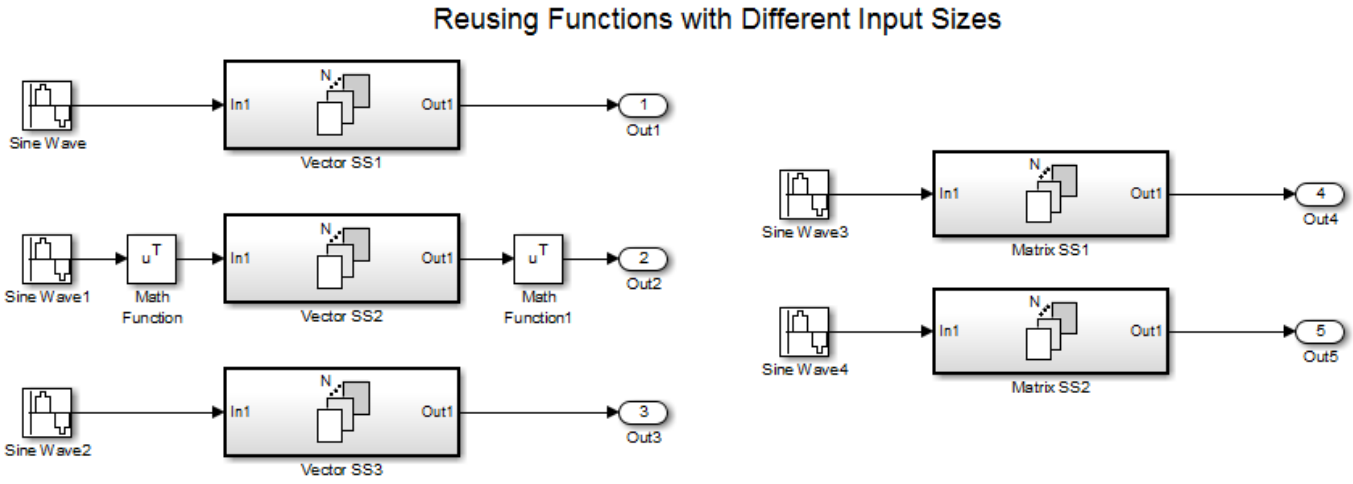


into three rows of parameters:

$$\begin{bmatrix} 0.0284 & 0.2370 & 0.4692 & 0.2370 & 0.0284 \end{bmatrix}$$
$$\begin{bmatrix} -0.0651 & 0.0000 & 0.8698 & 0.0000 & -0.0651 \end{bmatrix}$$
$$\begin{bmatrix} 0.0284 & -0.2370 & 0.4692 & -0.2370 & 0.0284 \end{bmatrix}$$

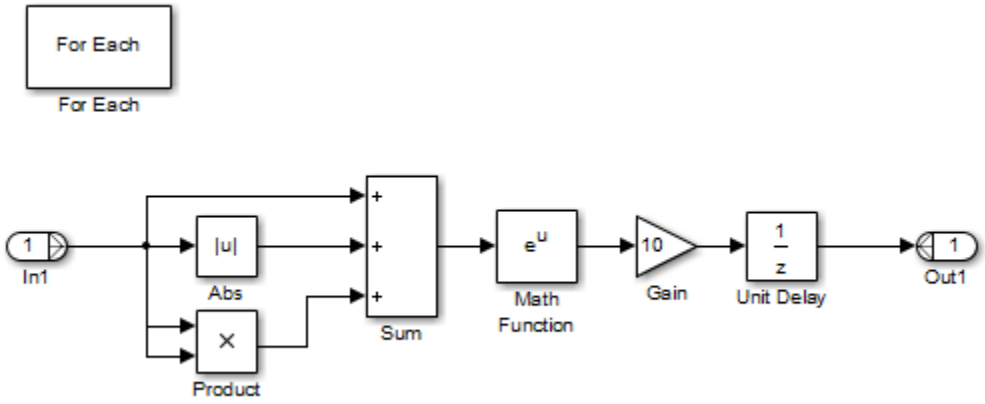
Improved Code Reuse Using For Each Subsystems

This example shows how you can improve code reuse when you have two or more identical For Each Subsystem blocks. 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 these 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.

See Also

Objects
`Simulink.Bus`

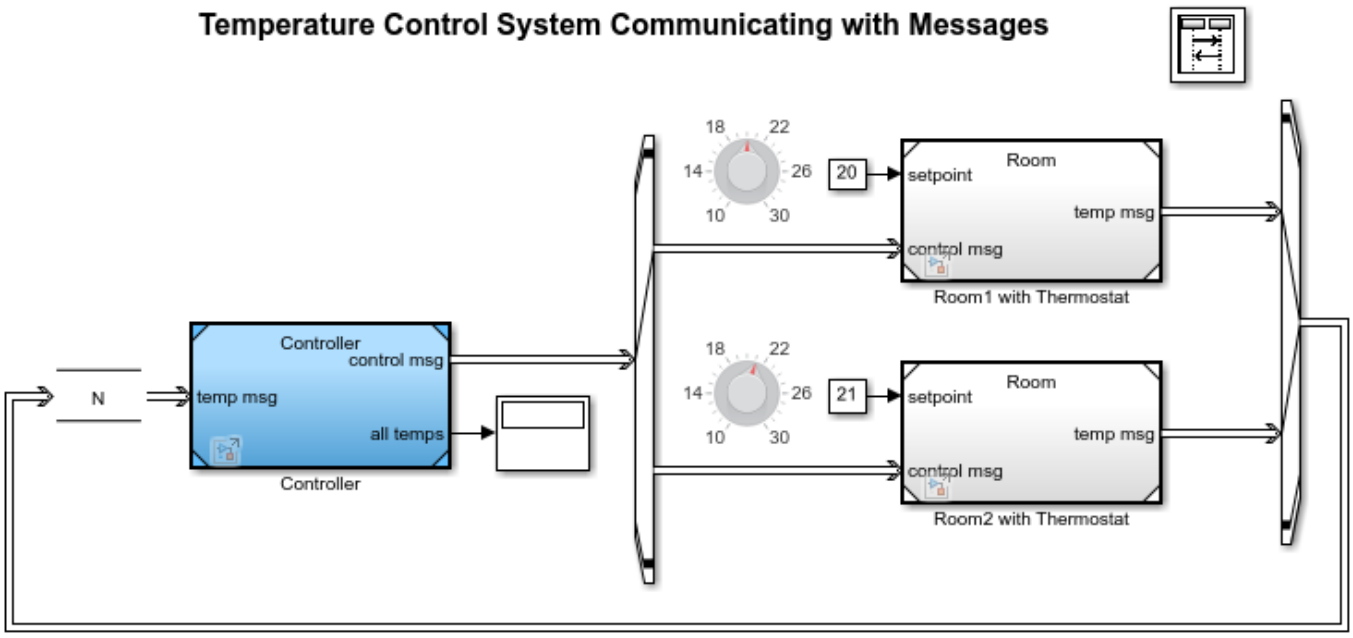
Blocks
For Each Subsystem

Related Examples

- “Convert Models to Use Arrays of Buses” on page 76-79
- “Load Bus Data to Root-Level Input Ports” on page 70-46
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44
- “Specify Initial Conditions for Bus Signals” on page 76-57
- “Organize Related Block Parameter Definitions in Structures” on page 37-19
- “Log Signals in For Each Subsystems” on page 72-71
- “Create a Custom Library” on page 41-2

Temperature Control System Communicating with Messages

This example shows how to use message communication within a distributed system where the controller manages multiple incoming messages from different senders in an iterative manner and sends messages to communicate commands to the different receivers. The example uses a model of a control system managing temperatures in two different rooms with separate thermostats. The algorithmic modeling of the components basically follows the Stateflow example “Model Bang-Bang Temperature Control System” (Stateflow), while the communication between the components is modeled using Simulink® messages and SimEvents® blocks. The referenced models Controller and Thermometer, colored in blue, are software components expected to generate standalone code, while the other components model the environment.



Model Overview

The model contains `N` identical rooms with thermostats (modeled by multi-instanced model blocks), where `N = 2` is a Simulink parameter defined in the Simulink Data Dictionary file `slddMsg.sldd`, which is linked to the top model and referenced models. Each room can set the setpoint temperature separately. The thermostats for rooms are remotely controlled by a controller, using the same control algorithm for all thermostats.

The thermostats send temperature messages to the controller every `0.2` seconds, while the controller sends the command messages to the thermostats to command heating on or off every `1` second. An Entity Output Switch (SimEvents) block routes the controller's messages to one of the thermostats according to the message bus data field `deviceID` (the bus is also defined in `slddMsg.sldd` and shared across all models). An Entity Input Switch (SimEvents) block routes the messages from different thermostats to the controller.

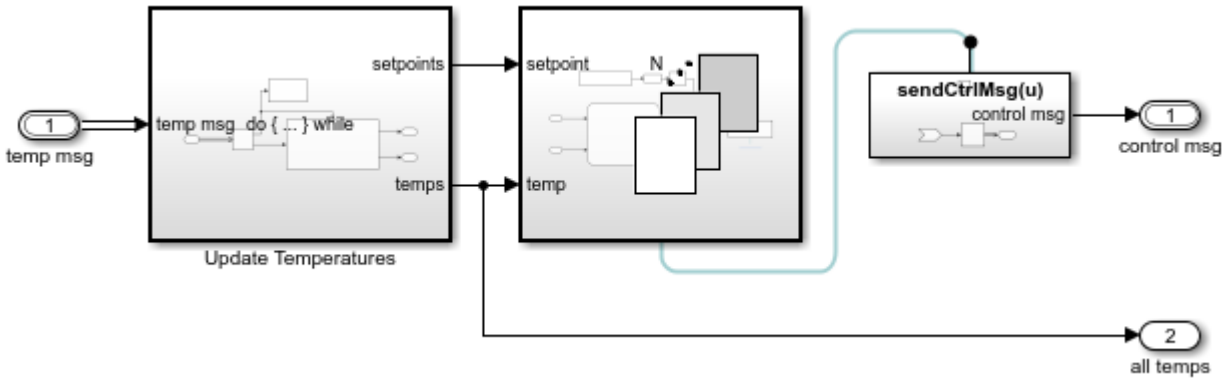
The model is easily scalable by changing the value of `N`, adding more instances of the model block, and increasing the port number of the Entity Output Switch and Entity Input Switch blocks. Each

Thermometer model inside the Room model has an `ID` argument, which must be set with a value that matches the output port index of the Entity Output Switch.

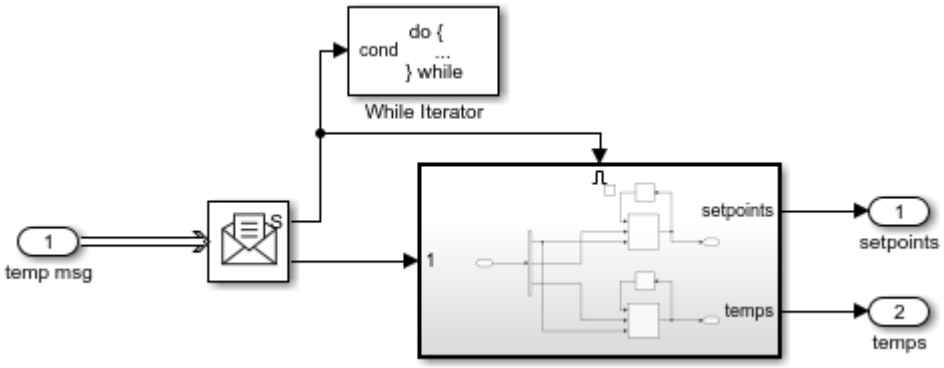
A Queue block (FIFO, overwriting type queue) in front of the controller model buffers the message, which models the queue inside the message middleware of the controller. Here, a capacity of `N` is good enough for the queue as long as no messages from the thermostats are dropped in the transport. A capacity of `5*N` is needed for the worst scenario with message loss, where `5` is the sample time of the controller divided by the sample time of the thermostats. In addition, the queue in front of each thermostat with capacity 1 is automatically inserted and shows a badge icon of sandwiched "1", because a capacity-1 queue is automatically inserted if you do not intentionally place a Queue block. See “Use a Queue Block to Manage Messages” on page 11-10.

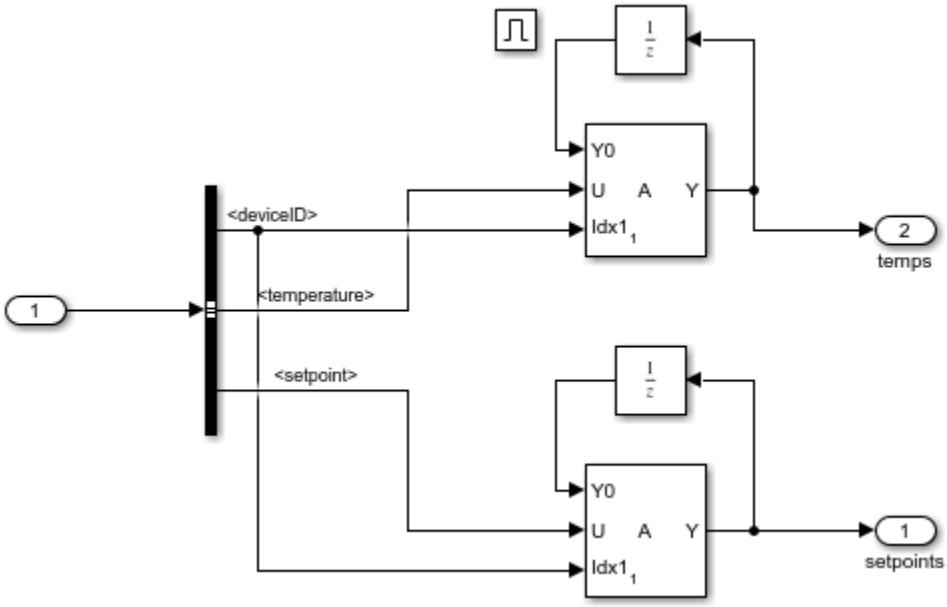
Double click the Sequence Viewer block to view the sequences of messages and events.

Controller Model

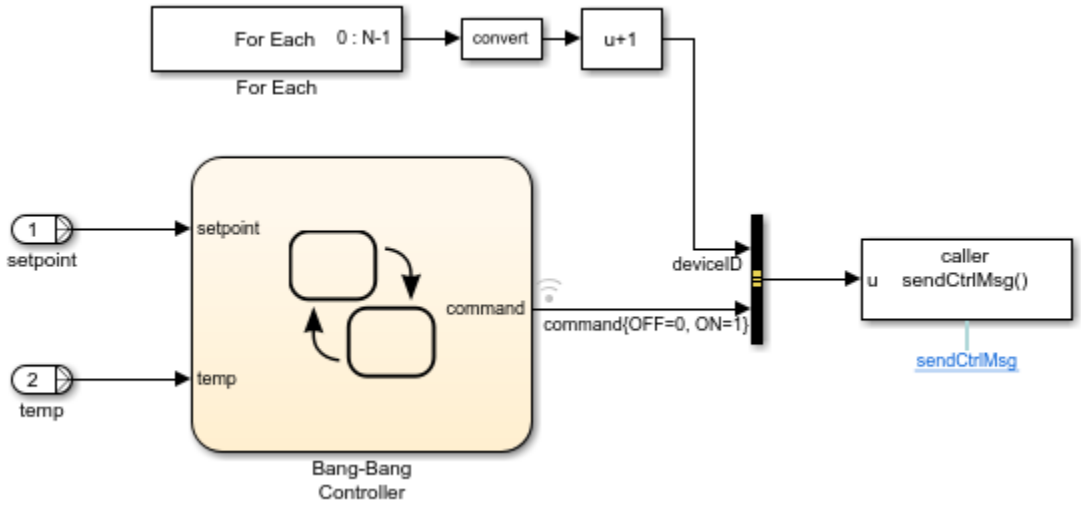


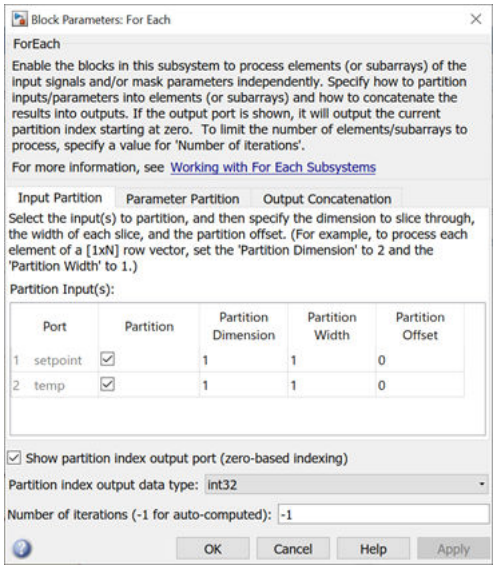
In the Controller model, the Update Temperature subsystem connected with the Inport block first receives all messages containing the temperature information from rooms. The subsystem stores that information in two vectors of temperature setpoint and current temperature. Then, the For Each subsystem reads the vectors, processes the signals, and sends out the control message via the Simulink Function `sendCtrlMsg`.





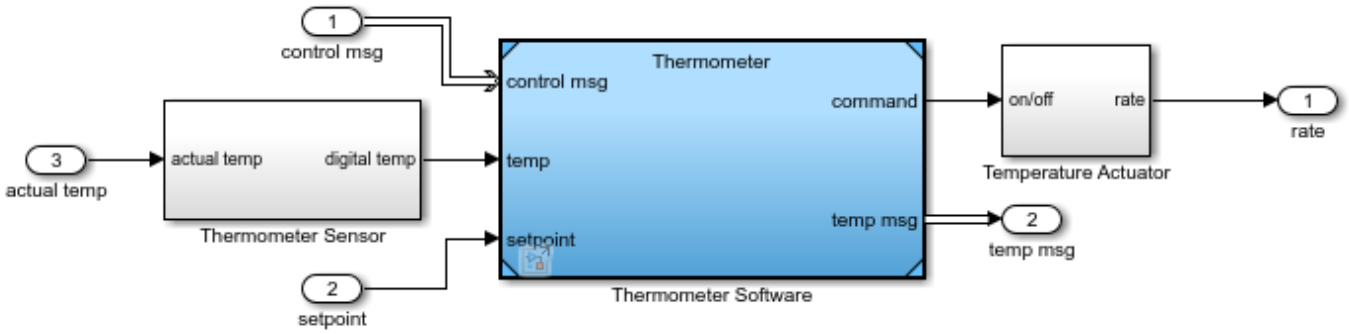
The Update Temperature subsystem is a do-while subsystem whose termination condition port is fed by the Receive block's status port, meaning it runs until cannot receive any more messages from the external queue (in the top model). The message data is of `DeviceMsg` bus type (defined in `slddMsg.sldd`), which has two fields: `temperature` and `deviceID`. Thus, when the output signal of the Receive block propagates to the Enable subsystem whose enable port is connected to the Receive block's status port, the Bus Selector block decomposes the signal into `deviceID`, `temperature`, and `setpoint` signals. The `setpoint` and `temperature` signals are then assigned to the respective vector elements associated with the `deviceID`. Finally, the vectors maintained by the Unit Delay blocks are output as signals by the Enable subsystem and Update Temperature subsystem to the For Each subsystem.





The For Each subsystem, whose block settings are shown above, is set to have `N` iterations, and both of its input ports are partitioned. A Stateflow chart models the Bang-Bang Controller, which resembles the one explained in “Model Bang-Bang Temperature Control System” (Stateflow). Its output port outputs a Boolean signal indicating whether or not to turn on heating. This signal is packed into a nonvirtual signal at the Bus Creator block with the deviceID (one-based) from the iteration number (zero-based). The signal is given to the Function Caller block, which calls the Simulink Function `SendCtrlMsg` (placed outside the For Each Subsystem) to send the message out from the model.

Room Model

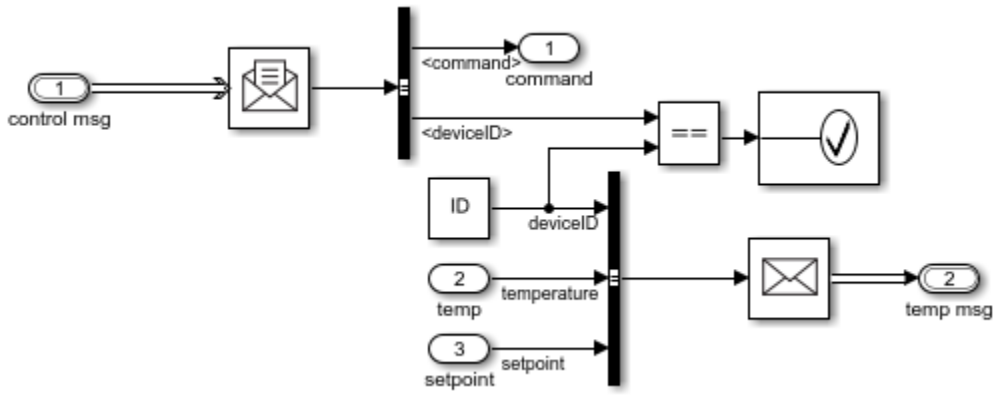


In the Room model, the Thermostat subsystem interacts with the environment. The Thermostat has two inputs, the control message and setpoint temperature signal, and two outputs, the heating rate and the temperature message to the controller. The Gain and Integrator blocks simulate the physics of the room heating or cooling with respect to the heating rate and room size.

The Thermostat subsystem is composed of a Thermometer Sensor subsystem, a Thermometer Software model block, and a Temperature Actuator subsystem. The Thermometer Software model block periodically receives a control message from the controller and unpacks it into a Boolean command (on/off) to the Temperature Actuator subsystem, which determines the heating rate. The Thermometer Software also inputs a temperature signal from Thermometer Sensor subsystem, which

detects the analog temperature, converts it to a digital signal, and sends the message back to the controller.

Thermometer Model



In the Thermometer model, a Receive block connects with the Inport block to receive a control message from the external queue at each time step. The message data is decomposed into a command signal, which is an output, and a deviceID signal, which must match the ID argument of the model. The ID argument should be set in the model block in the top model. The Receive block's initial value is set to a MATLAB structure with the `deviceID` field equal to the model argument `ID` and the `command` field taking a value of `false`. Meanwhile, the signals of digital temperature, setpoint, and deviceID are packed into a nonvirtual bus signal and sent as a message to the Outport block.

Code Generation

For code generation and deployment, the referenced models Controller and Thermometer (colored blue) can generate standalone embedded-target C++ code and can be deployed separately on embedded devices with message middleware. For more information, see “Generate C++ Messages to Communicate Between Simulink and an Operating System or Middleware” (Embedded Coder); see also “Use Handwritten Code to Integrate C++ Messages with POSIX” (Embedded Coder).

Message root-level Inport/Outport does not support C code generation and code customization. If you need to generate C code and call into a message middleware API for sending messages, consider moving the Simulink Function `sendCtrlMsg` to the top model and customizing the name properly so that the referenced model generates a customizable call site of an external function. Similarly, for the receive side, consider using a Simulink Function containing a Receive block in the top model and using a Function Caller block in the referenced model to replace the Receive block.

See Also

Integrator | Gain | For Each Subsystem | Function Caller | Receive | Simulink Function

Share and Reuse Bus-Routing Blocks

With custom libraries, you can share and reuse blocks that input or output buses. For a block linked to a parent library block that expects a bus input, you must provide the appropriate input bus. For example, the input bus for a Bus Selector block that is linked to a library must contain signals that have the same name as the signals that the parent library block selects.

To modify the parent library block, perform these steps. For details about modifying library links, see “Linked Blocks” on page 41-10.

1. Copy the parent library block to a model.
2. Connect a bus to the input port of the block.
3. Disable the link to the parent library block.
4. Edit the block within the context of the model.
5. Resolve the link to the library.
6. In the Link Tool, in **Push/Restore Mode**, select `Push` to place the edited content in the library.
7. Save the library.

To lock the interface of a parent library block, set its data type to a `Simulink.Bus` object.

With subsystem references, you can share and reuse a subsystem containing blocks that input or output buses and their associated signal lines. For more information, see “Subsystem Reference” on page 4-23.

See Also

Blocks
Bus Assignment | Bus Creator | Bus Selector | In Bus Element | Out Bus Element

More About

- “Types of Composite Signals” on page 76-2
- “Group Signal Lines into Virtual Buses” on page 76-8
- “Create a Custom Library” on page 41-2
- “Linked Blocks” on page 41-10

Generate Code for Nonvirtual Buses

If you have Simulink Coder, whether you use a virtual or nonvirtual bus can make a significant difference in the efficiency, size, and readability of generated code. For example, a nonvirtual bus appears as a structure in generated code, and only one copy exists of any algorithm that uses the bus. The use of a structure in the generated code can be helpful when tracing the correspondence between the model and the code. For example, here is the generated code for the Bus Creator block in the `ex_bus_logging` model.

```
50
51      /* BusCreator: '<Root>/COUNTERBUSCreator' incorporates:
52       *  BusCreator: '<Root>/LIMITBUSCreator'
53       *  Constant: '<Root>/lower_ saturation_ limit'
54       *  Constant: '<Root>/upper_saturation_limit'
55       */
56      ex_bus_logging_B.COUNTERBUS_n.data = rtb_data;
57      ex_bus_logging_B.COUNTERBUS_n.limits.upper_saturation_limit = 40;
58      ex_bus_logging_B.COUNTERBUS_n.limits.lower_saturation_limit = 0;
59
```

A virtual bus does not appear as a structure or any other coherent unit in generated code. A separate copy of any algorithm that manipulates the bus exists for each element. In general, virtual buses do not affect the generated code.

To group signals into structures in the generated code, use nonvirtual buses. See “Organize Data into Structures in Generated Code” (Simulink Coder).

When you create a MATLAB structure to initialize a bus that contains non-double signal elements, you need to set the values of structure fields. The technique that you choose to set the values can influence the efficiency and readability of the generated code. See “Control Data Types of Initial Condition Structure Fields” on page 76-101.

When you generate code for a bus that is input to or output from a referenced model, there are some code generation limitations. See “Limitations for Virtual Buses Crossing Model Reference Boundaries” on page 76-106.

Code generation for arrays of buses produces structures with a specific format. See “Code Generation for Arrays of Buses” on page 76-106.

Control Data Types of Initial Condition Structure Fields

You can use a MATLAB structure to initialize the signal elements in a bus. See “Specify Initial Conditions for Bus Signals” on page 76-57.

If the signal elements of the target bus use numeric data types other than `double`, in general:

- To avoid manually matching the field data types with the data types of the signal elements, use untyped expressions to set the field values. As you develop and rapidly prototype a model, use this technique for convenience.
- To generate more efficient production code and to avoid floating-point storage in the code, match the data types of the structure fields with the data types of the corresponding signal elements.

The technique that you choose can influence the efficiency and readability of the generated code.

For examples and more information about tunable initial conditions in the generated code, see "Control Signal and State Initialization in the Generated Code" (Simulink Coder).

Inline Numeric Values of Structure Fields in the Generated Code

If you set the **Default parameter behavior** configuration parameter to `Inlined`, by default, the field values of the initial condition structure appear in the generated code as inlined numbers (non-tunable). For these structures, use untyped expressions to set the field values in Simulink. The field values do not require data types because the structure is not tunable in the generated code.

However, if you later set **Default parameter behavior** to `Tunable` or apply a storage class to the structure by using a `Simulink.Parameter` object, the code can contain floating-point storage and inefficient explicit typecasts and bit shifts. To avoid these issues, consider matching the data types of the structure fields with the data types of the corresponding signal elements.

Generate Tunable Structure Specified Directly in a Block Dialog Box

Suppose that you specify an initial condition structure directly in a block dialog box, or in a `Simulink.Signal` object, with an expression such as `struct('signal1',5,'signal2',7.2)` (instead of storing the structure in a variable or `Simulink.Parameter` object). In this case, to generate a tunable structure in the code, you set **Default parameter behavior** to `Tunable`.

Use the table to decide how to control the data types of the fields in these initial condition structures.

Goal		Technique
Use a nonvirtual bus.		Use untyped expressions to set the field values.
Use a virtual bus.	Avoid manually matching the field data types with those of the signal elements.	Use untyped expressions to set the field values.
	Generate more efficient code and avoid floating-point storage.	Match the structure field data types with the signal element types. Store the data type information in the `struct` by using typed expressions to set the field values.

Generate Tunable Structure Stored in a Variable or Parameter Object

Suppose that you store an initial condition structure in a variable or `Simulink.Parameter` object that you create in the base workspace or a data dictionary. For example, you use this technique to share the structure between multiple blocks, or to generate a tunable structure when you set **Default parameter behavior** to `Inlined`. In this case, use the table to decide how to control the data types of the fields in the initial condition structure.

Goal	Technique
Avoid manually matching the field data types with those of the signal elements.	Use untyped expressions to set the field values. In the generated code, the structure fields use the data type `double`. The generated algorithm uses explicit typecasts to reconcile the data type mismatches.

Goal	Technique
Generate more efficient code and avoid floating-point storage.	Match the structure field data types with the signal element types. Store the data type information in the structure fields or use a `Simulink.Bus` object to control the data types of the fields and the signal elements simultaneously. To use the Model Advisor to check your model for potentially expensive data type mismatches, see “Check structure parameter usage with bus signals”.
Initialize an array of buses in a referenced model by using an array of structures. Pass the array of structures to the referenced model as the value of a model argument in the Model block.	Match the structure field data types with the signal element types. Store the data type information in the structure fields or use a `Simulink.Bus` object to control the data types of the structure fields and the signal elements simultaneously. If you do not pass the structure to the referenced model as a model argument, follow the other guidelines for nonvirtual buses to decide how to control the data types.

Use Untyped Expressions to Set Field Values

You can use untyped expressions to set the structure field values. The fields implicitly use the data type `double`. Set the field values to represent the ideal, real-world initialization values.

You avoid manually matching the field data types with the data types of the corresponding signal elements. However, depending on the virtuality of the bus, the method that you use to apply the initial condition, and other factors, you can introduce floating-point storage and potentially inefficient typecasts in the generated code.

Suppose that you create a bus `myBusSig` with these signal elements. Each element uses a specific data type.

```
myBusSig
   signalElement1 (int32)
   signalElement2 (boolean)
   signalElement3 (single)
```

Create an initial condition structure `initStruct`. Use untyped expressions to specify the field values. Optionally, to enhance readability of the Boolean field `signalElement2`, use the value `false` instead of `0`.

```
initStruct.signalElement1 = 3;
initStruct.signalElement2 = false;
initStruct.signalElement3 = 17.35;
```

If you use the function `Simulink.Bus.createMATLABStruct` to create the structure, the function stores data type information in the structure fields. After you create the structure, you can optionally use untyped expressions to change the field values. See “Use Simulink.Bus.createMATLABStruct to Create Structure” on page 76-104.

Store Data Type Information in Structure Fields

To store data type information in the structure fields, use typed expressions to set the field values, or use the function `Simulink.Bus.createMATLABStruct` to create the structure. Use these

techniques to generate efficient code by eliminating floating-point storage and potentially inefficient explicit typecasts.

To avoid manually applying new data types to the structure fields when you change the data types of the corresponding signal elements, consider using a `Simulink.Bus` object to control the data types in the structure and the bus simultaneously.

Use Typed Expressions to Set Field Values

Suppose that you create a bus `myBusSig` with this hierarchy of signal elements. Each element uses a specific data type.

```
myBusSig
   signalElement1 (int32)
   signalElement2 (boolean)
   signalElement3 (single)
```

Create an initial condition structure `initStruct` by using typed expressions to set the field values. Match the data types of the fields with the data types of the corresponding signal elements.

```
initStruct.signalElement1 = int32(3);
initStruct.signalElement2 = false;
initStruct.signalElement3 = single(17.35);
```

The structure fields store data type information. If you later change the data type of a signal element, manually apply the new data type to the corresponding structure field.

To match a fixed-point data type, set the field value by using a `fi` object.

Change Field Value by Preserving Data Type Information

Suppose that you change the value of a field in an existing initial condition structure. To preserve the data type information in the field you can use subscripted assignment, with the syntax `(:)`.

```
initStruct.signalElement3(:) = 16.93;
```

If you do not use subscripted assignment, you must remember to preserve the data type by using a typed expression.

```
initStruct.signalElement3 = single(16.93);
```

If you do not use either of these techniques, the field loses the data type information.

```
initStruct.signalElement3 = 16.93; % Field data type is now 'double'.
```

Use Simulink.Bus.createMATLABStruct to Create Structure

You can use the function `Simulink.Bus.createMATLABStruct` to create a structure whose fields all have ground values, typically `0`. If you configure the data types of the signal elements before using the function, for example by setting the output data types of the blocks that generate the signal elements, each field in the output structure uses the same data type as the corresponding signal element. The fields store the data type information as if you use typed expressions to set the values.

You can initialize some of the signal elements with a value other than ground by passing a partial structure to the function. When you create this partial structure, match the data type of each field with the data type of the corresponding signal element by using typed expressions. For more information and examples, see `Simulink.Bus.createMATLABStruct`.

When you later change the value of a field in the structure, choose one of these techniques to set the new value:

- Untyped expression. The field value no longer stores the data type information.
- Typed expression or subscripted assignment. The field value continues to store the data type information.

Use Bus Object as Data Type of Initial Condition Structure

Whether you store data type information in the structure fields or use untyped expressions to set the field values, you can use a `Simulink.Bus` object as the data type of the entire initial condition structure. You can then manage the field values and data types independently.

If you use this technique, consider using untyped expressions to set the field values. Then, you do not need to match the field data types manually when you change the data types of the signal elements. To control the data types of the fields and the signal elements, use the `DataType` property of the elements in the `Bus` object.

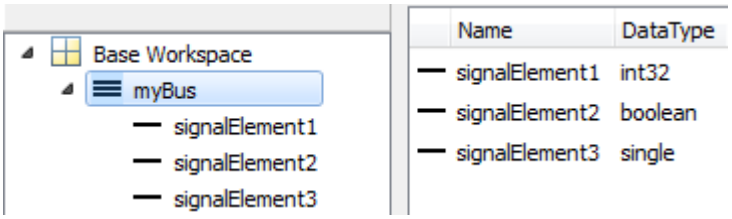
Suppose that you use a Bus Creator block to create a bus `myBusSig` with these signal elements.

```
myBusSig
   signalElement1 (int32)
   signalElement2 (boolean)
   signalElement3 (single)
```

1. Open the Bus Editor.

   ```
   buseditor
   ```

2. Create a `Bus` object, `myBus`, that corresponds to the bus.



3. Create an initial condition structure `initStruct`. Used untyped expressions to set the field values. To enhance readability of the field `signalElement2`, use the Boolean value `false` instead of `0`.

   ```
   initStruct.signalElement1 = 3;
   initStruct.signalElement2 = false;
   initStruct.signalElement3 = 17.35;
   ```

4. To represent the structure, create a `Simulink.Parameter` object.

   ```
   initStruct = Simulink.Parameter(initStruct);
   ```

5. Use the parameter object to specify an initial condition for the bus. For example, in a Unit Delay block dialog box, set **Initial condition** to `initStruct`.

6. Use the `Bus` object to specify the data type of the parameter object.

   ```
   initStruct.DataType = 'Bus: myBus';
   ```

7 Use the `Bus` object to specify the data type of the bus. For example, in the Bus Creator block dialog box, set **Output data type** to `Bus: myBus`.

During simulation and in the generated code, the structure fields and the signal elements use the data types that you specify in the `Bus` object. Before simulation and code generation, the parameter object casts the structure fields to the data types that you specify in the `Bus` object.

For basic information about `Bus` objects, see “Specify Bus Properties with Simulink.Bus Objects” on page 76-44.

Configure Data Types for Existing Structure

To remove data type information from all the fields of a structure, you can write a custom function that replaces the field values with `double` numbers. Use the example function `castStructToDbl` as a template.

To convert a structure that uses doubles to one that stores data type information, you can create a reference structure using the function `Simulink.Bus.createMATLABStruct`. You can then write a custom function to cast the field values to the data types in the reference structure. Use the example function `castStructFromDbl` as a template.

Check for Mismatched Data Types with Model Advisor

To detect when the data types of structure fields are not consistent with associated bus elements, use the Model Advisor.

1 On the **Modeling** tab, click **Model Advisor**.

2 Click **OK**.

3 Select **By Task > Modeling Signals and Parameters using Buses > “Check structure parameter usage with bus signals”**.

4 Click the **Run This Check** button.

Limitations for Virtual Buses Crossing Model Reference Boundaries

If you use a bus as an input to or an output from a referenced model (Model block):

- You cannot configure the `I/O arguments step method` style of C++ class interface for the referenced model.

  As a workaround, use a nonvirtual bus instead. Alternatively, use the `Default` style of C++ class interface.
- You cannot configure function prototype control for the referenced model.

  As a workaround, use a nonvirtual bus instead.

For more information about using buses as inputs to or outputs from a referenced model, see “Nonvirtual Buses at Model Interfaces” on page 76-55. For more information about bus virtuality, see “Types of Composite Signals” on page 76-2.

Code Generation for Arrays of Buses

When you generate code for a model that includes an array of buses, a `typedef` that represents the underlying bus type appears in the `*_types.h` file.

Code generation produces an array of C structures that you can integrate with legacy C code that uses arrays of structures. As necessary, code for bus variables (arrays) is generated in the following structures:

- Block IO
- States
- External inputs
- External outputs

Here is a simplified example of some generated code for an array of buses.

```
typedef struct {
  real_T a;
  real_T b;
} BusObject;
```

```
/* Block signals (auto storage) */
typedef struct {
  BusObject ForEachSubsystem_IterInp_0[2];
} BlockIO_aob1;
```

For basic information about code generation for nonvirtual buses, which appear in the code as structures, see “Organize Data into Structures in Generated Code” (Simulink Coder).

See Also

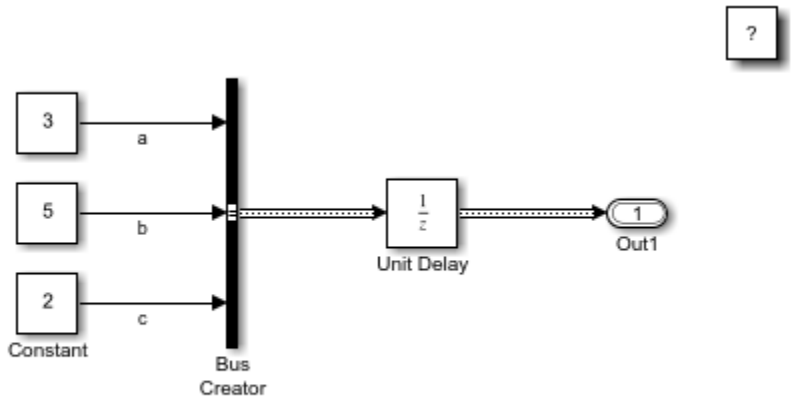
Related Examples

- “Organize Data into Structures in Generated Code” (Simulink Coder)
- “Nonvirtual Buses at Model Interfaces” on page 76-55
- “Inspect Generated Code for Nonvirtual Buses” on page 76-108

Inspect Generated Code for Nonvirtual Buses

This example shows how nonvirtual buses appear in the generated code for a model. The steps require a Simulink® Coder™ license.

Open and simulate the example model, which contains a nonvirtual bus.



Open the **Simulink Coder** app. On the **Apps** tab, click the arrow on the far right of the **Apps** section. Under **Code Generation**, click **Simulink Coder**.

To generate code for the model, on the **C Code** tab, click **Generate Code**.

To see the generated files, open the `NonvirtualBusCodeGenModel_grt_rtw` folder.

`NonvirtualBusCodeGenModel_types.h` defines the `Simulink.Bus` object as a struct.

```
typedef struct {
  real_T a;
  real_T b;
  real_T c;
} BusObject;
```

`NonvirtualBusCodeGenModel.h` defines the Unit Delay block using the `BusObject` struct.

```
typedef struct {
  BusObject UnitDelay_DSTATE;          /* '<Root>/Unit Delay' */
} DW_NonvirtualBusCodeGenModel_T;
```

`NonvirtualBusCodeGenModel.c` implements the Unit Delay block, which passes the nonvirtual bus to the Outport block.

```
NonvirtualBusCodeGenModel_Y.Out1 =
NonvirtualBusCodeGenModel_DW.UnitDelay_DSTATE;
```

See Also

Blocks
Bus Creator

Objects
`Simulink.Bus`

Related Examples

- “Types of Composite Signals” on page 76-2
- “Specify Bus Properties with Simulink.Bus Objects” on page 76-44

Trace Connections Using Interface Display

How Interface Display Works

In the Simulink Editor, you can turn on and off the display of interfaces in a model. When you are building large, complex models, you can connect or add signal lines between blocks or buses that are at different levels. The interface view allows you to trace signals through the nested levels. This capability helps you to:

- Identify inputs and outputs.
- Trace signal lines and bus elements to sources and terminations.
- Annotate signal characteristics such as data type, dimensions, and sample time.
- View units associated with signals, where applicable.

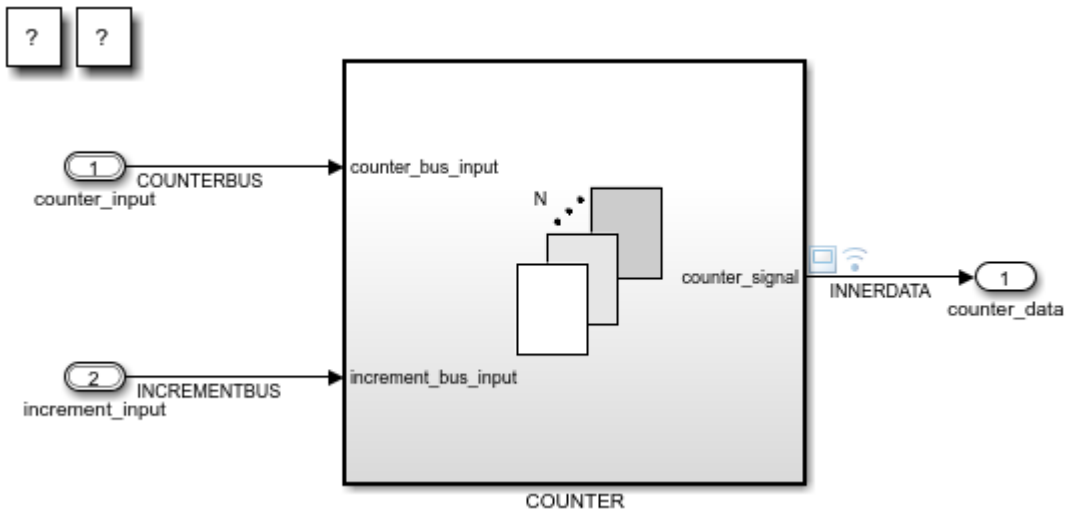
When you build a model, transition block pairs such as Inport and Outport and From and Goto help you to simplify connections of the crossovers among many signal lines. The interface view enables you to trace the hand-off and receipt between such blocks by way of colored highlights.

Trace Connections in a Subsystem

This example shows how to use the display of model interfaces to examine, trace, and understand the flow of signals and buses. This model propagates buses into referenced models.

1 Open the model `sldemo_mdlref_counter_bus`.

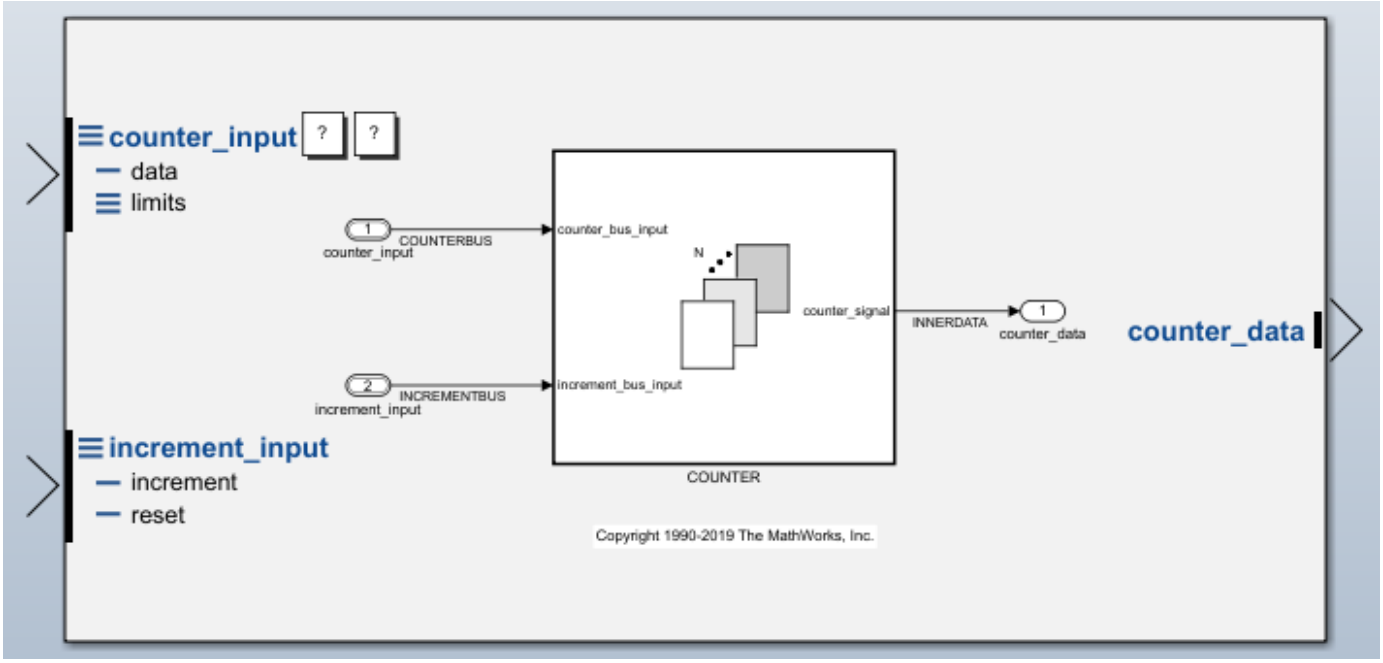
The counter_bus_input port channels the data and saturation limits of the counter to count and sets the upper and lower limit values. The increment_bus_input port channels a bus to change the increment and reset the counter.



2 On the **Modeling** tab, under **Design**, click **Model Interface** to enable the interface view.

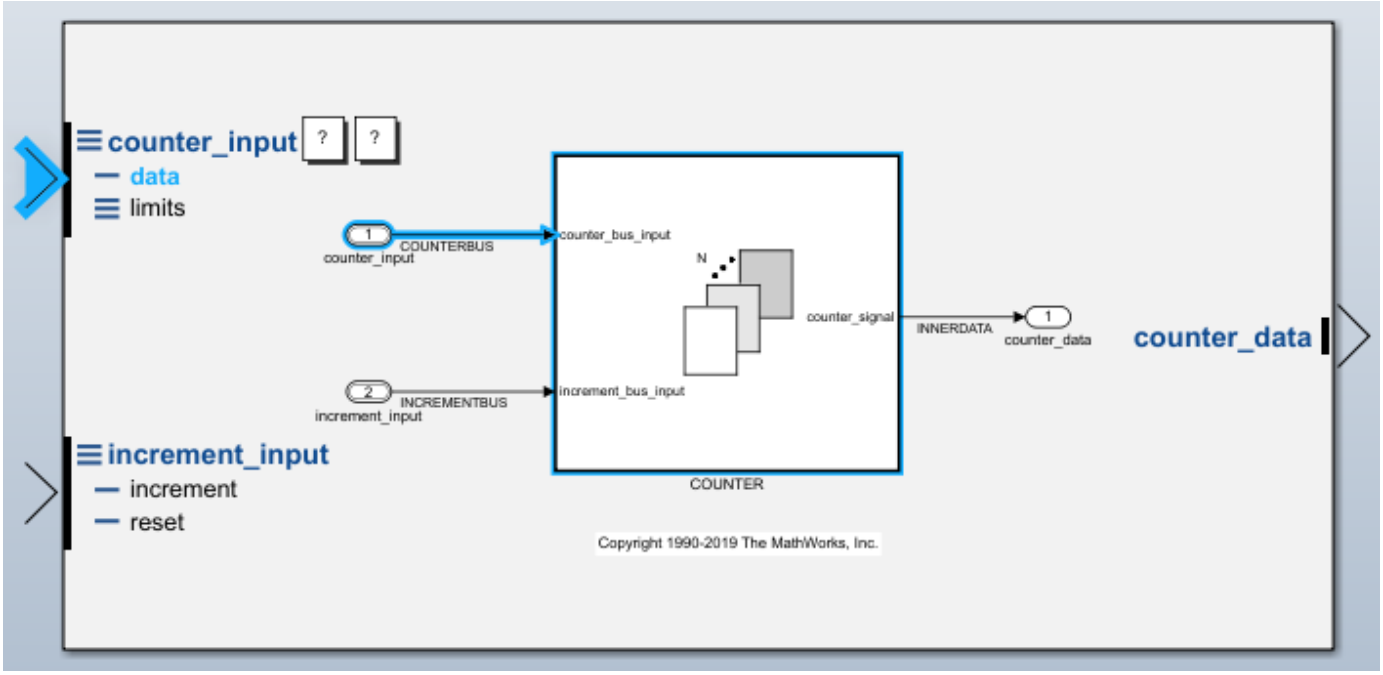
Tip Use the perspectives control in the lower-right corner of the model to toggle the display of interfaces.

The three bars next to the `counter_input` and `increment_input` interfaces indicate the bus input signals. The single bars indicate data lines, such as counts per second, or command lines, such as reset, to start a new counting sequence. The three bars next to `limits` indicate that buses are nested inside the COUNTER subsystem.



3 Under `counter_input`, click `data`.

The path for the data appears in blue. The COUNTER subsystem is highlighted, indicating the path continues within it.



4 Double-click the COUNTER subsystem.

The continuation of the path for the data signal appears in blue.

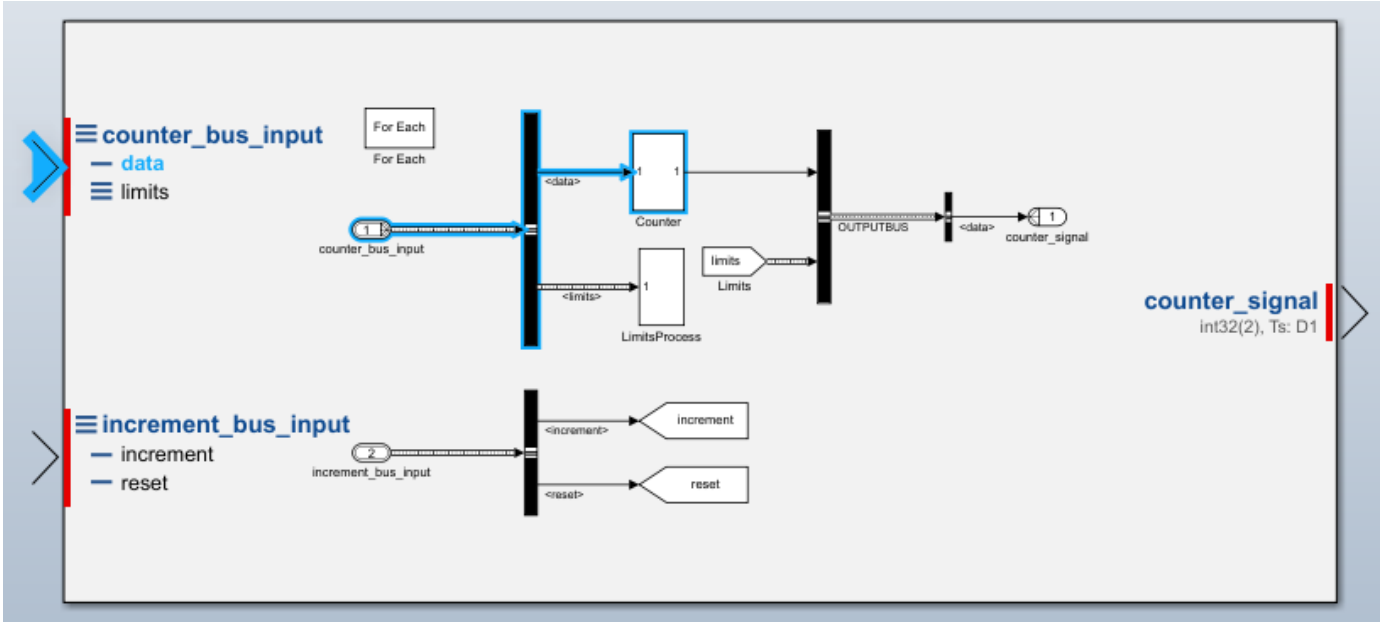
5 On the **Modeling** tab, click **Update Model**.

Note This model requires values from a parent model to simulate completely.

The `counter_signal` interface displays these signal attributes, which help you to synchronize signals between blocks during simulation:

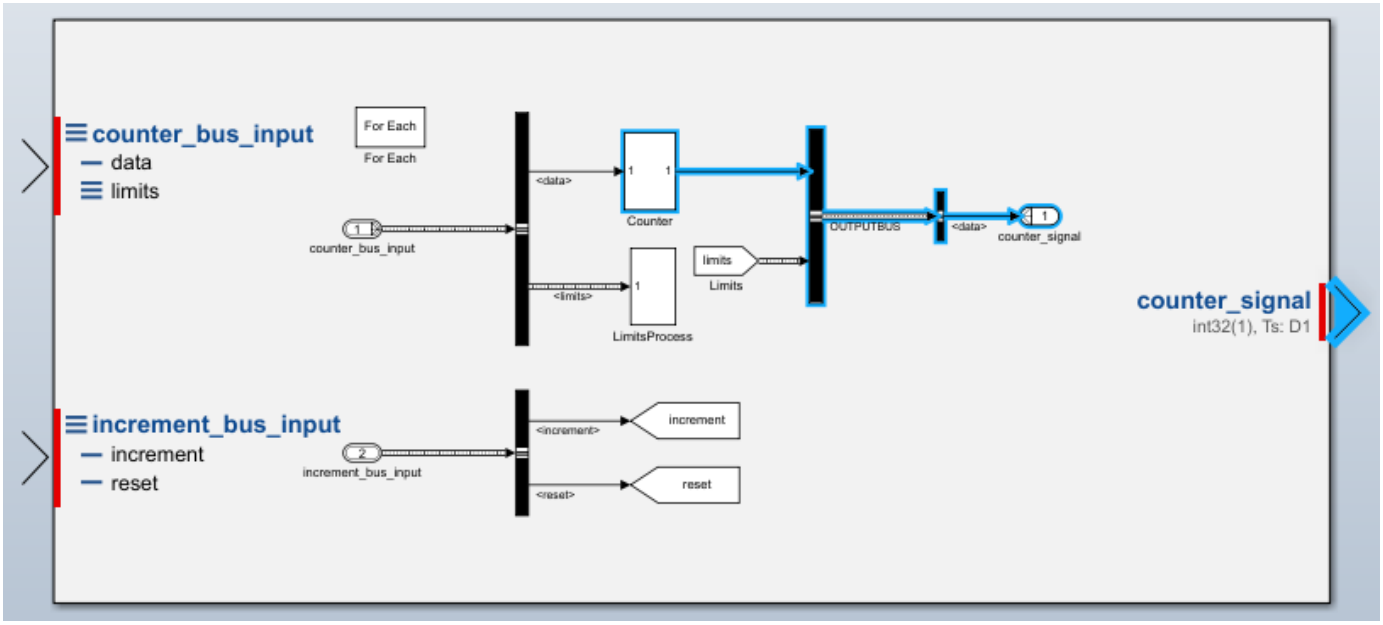
- Data type: `int32` (signed 32-bit integer)
- Dimensions: `(2)` (a 1-D Simulink representation of a scalar)
- Sample time: `Ts:D1` (a discrete sample time of D1, which is the highest speed)

In addition, when you update the diagram with interfaces displayed, the model displays the color code for sample time at each interface. For example, this model displays a red bar at each interface to indicate a sample time of D1.



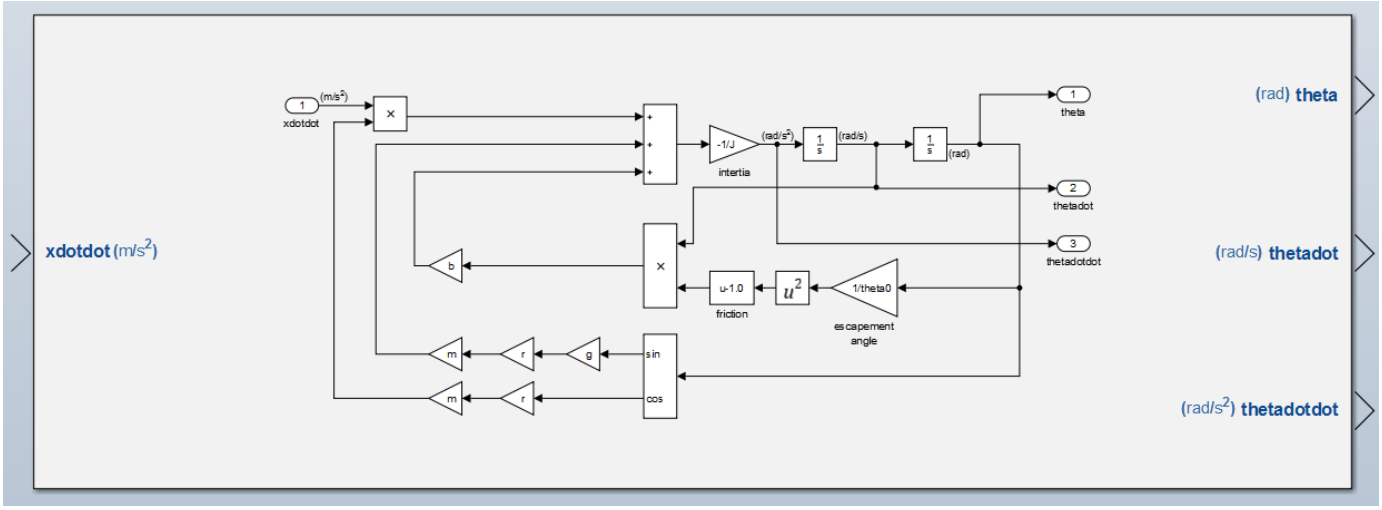
Tip To display a legend of the meaning of sample time colors, on the **Debug** tab, select **Information Overlays > Colors**.

6 Click the `counter_signal` output interface to see the output of the bus outlined in blue, where the path ends.



7 If you want to print this diagram with the interfaces displayed, in the **Simulation** tab, click **Print**.

When signals in your model have units associated with them, you see the units in the interface view. For example, in the model `sldemo_metro`, the Metronome1 subsystem shows units for inputs and outputs of the subsystem in the interface view.



To modify the attributes of the existing interface (such as signal names, data types, and dimensions), consider using the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). For information about the Model Data Editor, see “Configure Data Properties by Using the Model Data Editor” on page 67-131.

See Also

More About

- “Define Interfaces of Model Components” on page 22-17
- “What Is Sample Time?” on page 7-2
- “Virtual Bus” on page 76-2

Working with Variable-Size Signals

Variable-Size Signal Basics

In this section...

A Simulink signal can be a scalar, vector (1-D), matrix (2-D), or N-D. A Simulink variable-size signal is a signal whose size (the number of elements in a dimension), in addition to its values, can change during a model simulation. However, during a simulation, the number of dimensions cannot change. This capability allows you to model systems with varying resources, constraints, and environments.

You can create variable-size signals in your Simulink model by using:

- Switch or Multiport Switch blocks with different input ports having fixed-size signals with different sizes. The output is a variable-size signal.
- A selector block and the `Starting and ending indices (port)` indexing option. The index port signal can specify different subregions of the input data signal which produce an output signal of variable size as the simulation progresses.
- The S-function block with the output port configured for a variable-size signal. The output includes not only the values but also the dimension of the signal.

How Variable-Size Signals Propagate

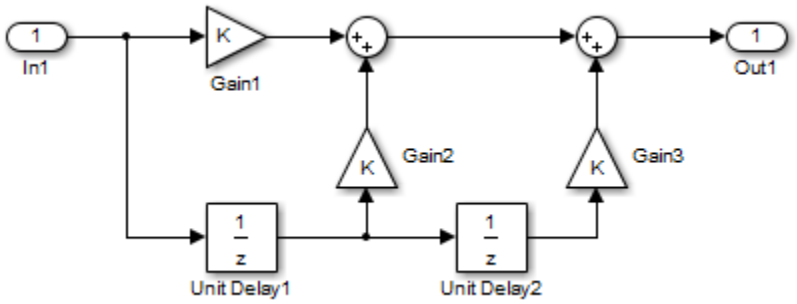
In the Simulink environment, variable-size signals can change their size during model execution in one of two ways:

- **At every step of model execution.**

  Various blocks in the model modify the sizes of the signals during execution of the output method.
- **Only during initialization of conditionally executed subsystems.**

  Size changes occur during distinct mode-switching events in subsystems such as Action, Enable, and Function-Call subsystems.

You can see the key difference by considering a Discrete 2-Tap Filter block with states.



Discrete 2-Tap Filter

Assume that the input signal dimension to this filter changes from 4 to 1 during simulation. It is ambiguous when and how the states of the Unit Delay blocks should adapt from 4 to 1 to continue processing the input. To ensure consistency, both Unit Delay blocks must change their state behavior synchronously. To prevent ambiguity, Simulink generally disallows blocks whose number of states depends on input signal sizes in contexts where signal sizes change at any point during execution.

In contrast, consider the same Discrete 2-Tap Filter block in a Function-Call subsystem. Assume that this subsystem is using the second way to propagate variable-size signals. In this case, the size of the input signal changes from 4 to 1 only at the initialization of the subsystem. At initialization, the subsystem resets all of its states (including the states of the two Unit Delay blocks) to their initial values. Resetting the subsystem ensures no ambiguity on the assignment of states to the input signal of the filter.

“Mode-Dependent Variable-Size Signals” on page 77-14 shows how you can use the two ways of propagating variable-size signals in a complementary fashion to model complex systems.

Determine Whether Signal Line Has Variable Size

The following example demonstrates how to use commands at the command prompt or in a script to determine whether a signal line has a variable size. In a large model or hierarchy of subsystems or referenced models, use this technique to determine whether a signal has a variable size due to an upstream block.

The example model `sldemo_varsize_basic` contains a signal `a` that is downstream of a Switch block.

1. Open the example model.
2. Select the Sum block whose output signal is labeled `a`.
3. Type the following at the command window to set the model to a compiled state (similar to a diagram update).

   ```
   sldemo_varsize_basic([],[],[],'compile')
   ```
4. Get a handle to the block output port.

   ```
   portHandles = get_param(gcb,'portHandles');
   outPortHandle = portHandles.Outport;
   ```
5. Query the programmatic parameter `CompiledPortDimensionsMode` of the output port.

   ```
   varSize = get_param(outPortHandle,'CompiledPortDimensionsMode')
   ```

```
varSize =

     1
```

The value of the variable `varSize` is `1`, which indicates that the signal `a` has variable size.

The value `0` indicates that a signal does not have variable size.

6 Terminate the model compilation.

```
sldemo_varsize_basic([],[],[],'term')
```

Empty Signals

An empty signal is a signal with a length of `0`. For example, signals with size `[0]`, `[0x3]`, `[2x0]`, and `[2x0x3]` are all empty signals. Simulink allows empty signals with variable-size signals and supports most element-wise operations. However, Simulink does not support empty signals for blocks that modify signal dimensions. Unsupported blocks include Reshape, Permute, and Sum along a specified dimension.

Simulink Block Support for Variable-Size Signals

The Simulink Block Data Type Support table includes a complete list of blocks that support variable-size signals.

To view the table:

1 Open a Simulink model.

2 In the MATLAB command line, enter `showblockdatatypetable`.

A separate window with the Simulink Block Data Type Support table opens.

An X in the **Variable-Size Support** column indicates support for that block.

Tip You can also view the table by entering `showblockdatatypetable` at the command prompt.

Subsystem Initialization of Variable-Size Signals

The initial signal size from an Outport block in a conditionally executed subsystem varies depending on the parameters you select.

If you set the **Propagate sizes of variable-size signals** parameter in the parent subsystem to `During execution`, the **Initial output** parameter for the Output block must not exceed the maximum size of the input port. If the **Initial output** parameter value is:

Initial output parameter	Initial output signal size
A nonscalar matrix	The initial output signal size is the size of the **Initial output** parameter.
A scalar	The initial output signal size is a scalar.

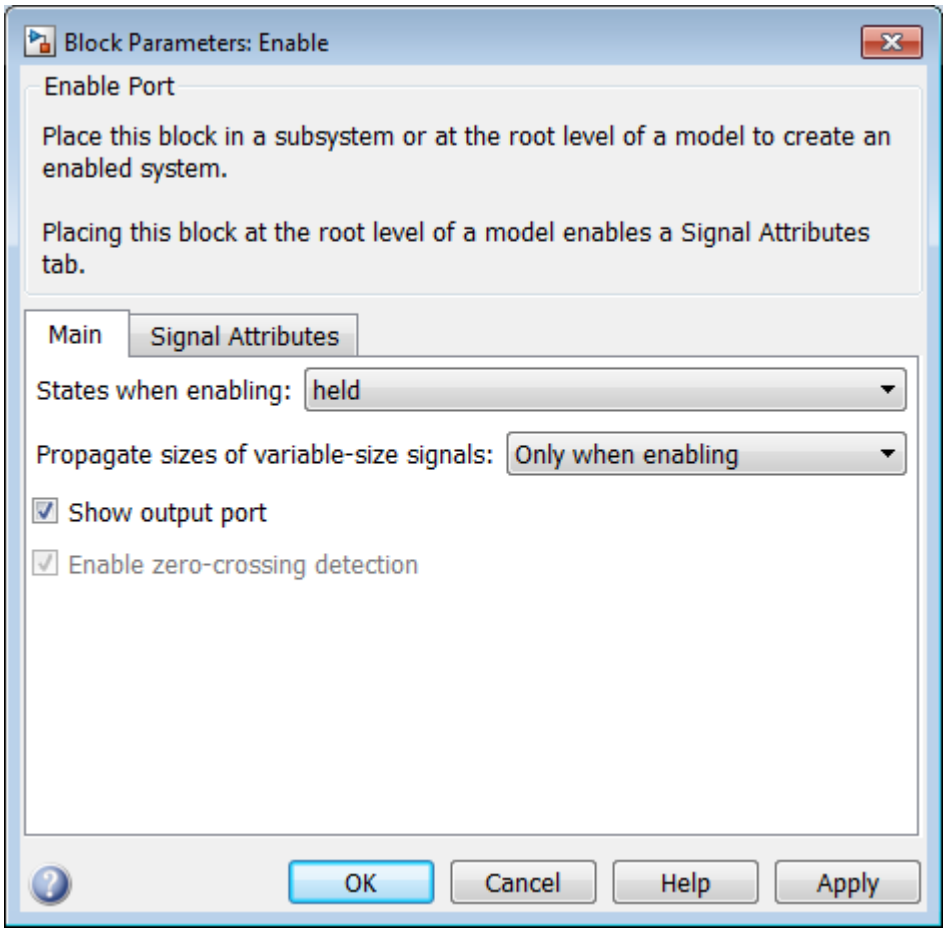
Initial output parameter	Initial output signal size
The default []	The initial output size is an empty signal (dimensions are all zeros).

If you set the **Propagate sizes of variable-size signals** parameter in the parent subsystem to `Only when enabling`, the **Initial output** parameter for the Output block must be a scalar value.

- When size is repropagated for the input of the Outport block, the initial output value is set using scalar expansion from the scalar parameter value.
- If the **Initial output** parameter is the default value `[]`, Simulink treats the initial output as a grounded value.
- If the model does not activate the parent subsystem at start time (`t = 0`), the current size of the subsystem output corresponding to the Outport block is set to maximum size.
- When its parent subsystem repropagates signal sizes, the values of the subsystem variable-size output signals are also reset to their initial output parameter values.

Conditionally Executed Subsystem Blocks

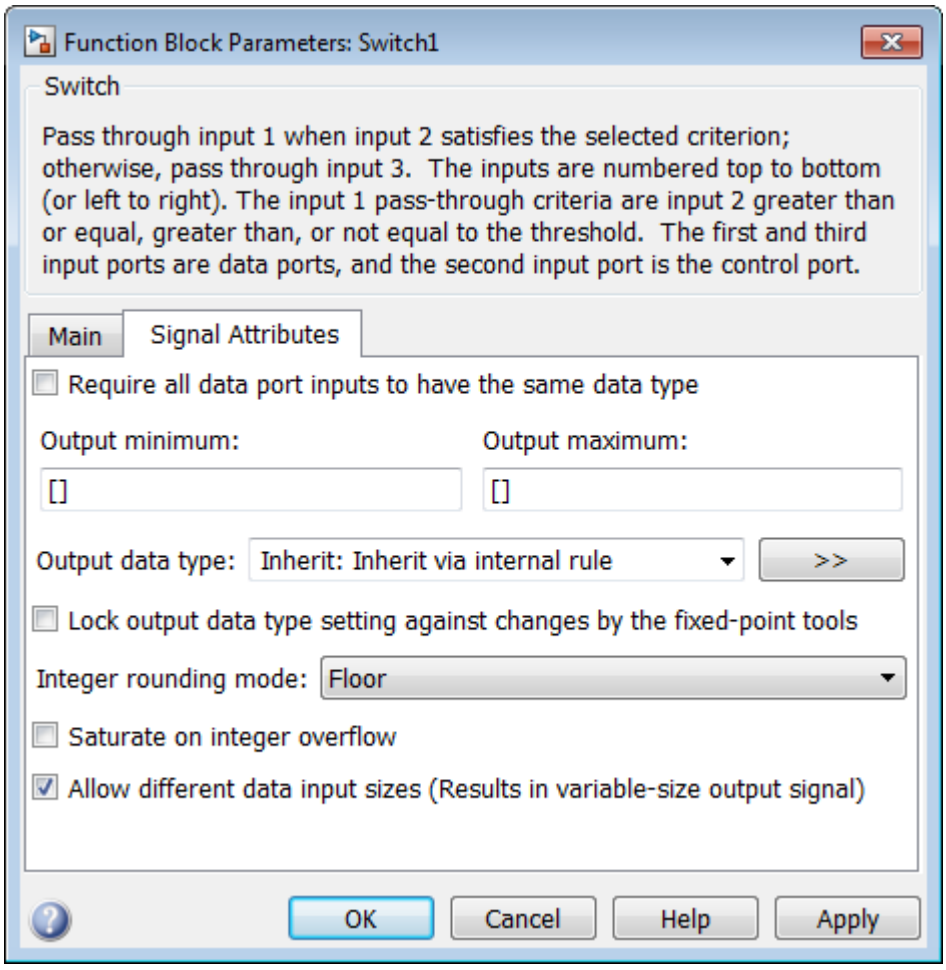
Control port blocks are in conditionally executed subsystems. You can set the **Propagate sizes of variable-size signals** parameter for these blocks to `During execution`, `Only when execution is resumed` (Action Port), and `Only when enabling` (Enable and Trigger or Function-Call).



- Action Port
- Enable
- Trigger — **Trigger type** set to `function-call`

Switching Blocks

Switching blocks support variable-size signals by allowing input signals with different sizes and propagating the size of the input signal to the output signal. You can set the **Allow different data input sizes** parameter for these blocks on the Signal Attributes pane to either on or off.



- Switch
- Multiport Switch
- Manual Switch

Variable-Size Signal Limitations

The following table describes variable-size signal limitations.

Limitation	Workaround
Array format logging does not support variable-size signals.	Use a `Structure` or `Structure With Time` format for logging variable-size signals.
Right-click signal logging does not support variable-size signals.	Use a To Workspace block (with `Structure` or `Structure With Time` format) or a root Outport block for logging variable-size signals.
A frame-based variable-size signal cannot change the frame length (first dimension size), but it can change the second dimension size (number of channels). Using frame-based signals requires DSP System Toolbox software.	Use the Frame Conversion block to convert a signal into sample-based signal.
Variable-size signals must have a discrete sample time.	—
Embedded Coder does not support variable-size signals with ERT S-functions, custom storage classes, function prototype control, the AUTOSAR, C++ interface, and the ERT reusable code interface.	—
Simulink does not support variable-size parameter or DWork vectors.	—
Rapid accelerator mode does not support models having root-level input ports with variable-size signals.	—
Virtual buses that you use as inputs to or outputs from a referenced model (Model block) do not support variable-size signals.	Configure the bus signal as nonvirtual. For more information about using buses as inputs to or outputs from a referenced model, see “Nonvirtual Buses at Model Interfaces” on page 76-55. For more information about controlling bus virtuality, see “Types of Composite Signals” on page 76-2.
Variable-size signals are not supported for: • Array of buses signals • Blocks that specify an initial condition as a MATLAB structure	—
You cannot apply a storage class to a root-level Outport block (see “C Code Generation Configuration for Model Interface Elements” (Simulink Coder)) if the signal that enters the block has a variable size.	Apply the storage class to the signal line instead of the Outport block.

See Also

Related Examples

- “Signal Basics” on page 75-2
- “Inspect Variable-Size Signals on Simulink Models” on page 77-9

Inspect Variable-Size Signals on Simulink Models

In this section...

Variable-Size Signal Generation and Operations

This example model shows how to create a variable-size signal from multiple fixed-size signals and from a single data signal. It also shows some of the operations you can apply to variable-size signals.

For a complete list of blocks that support variable-size signals, see “Simulink Block Support for Variable-Size Signals” on page 77-4.

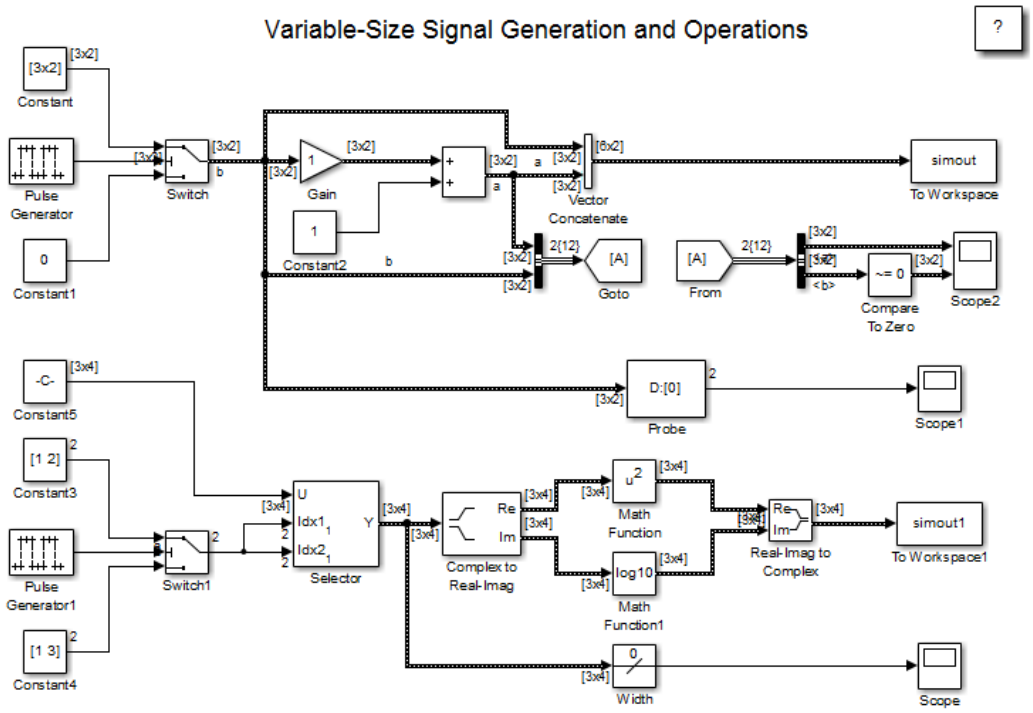
1. In the MATLAB Command Window, type

   ```
   sldemo_varsize_basic
   ```

2. In the Simulink Editor, on the **Debug** tab, select **Information Overlays > Signal Dimensions**. Run a simulation or press **Ctrl-D**.

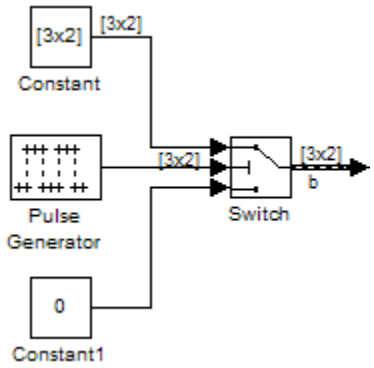
   The Simulink Editor displays the signal dimensions and line styles. See “Signal Basics” on page 75-2 for an interpretation of signal line styles.

3. So that you can see the names of the blocks in the model, on the **Format** tab, clear **Auto > Hide Automatic Block Names**.



Create a Variable-Size Signal from Fixed-Size Signals

One way to create a variable-size signal is to use the Switch block. The input signals to the Switch block can differ in their number of dimensions and in their size.



Output from the Switch block is a 2-D variable-size signal with a maximum size of `3x2`. When you select the **Allow different data input sizes** parameter on the Switch block, Simulink does not expand the scalar value from the Constant1 block.

Save Variable-Size Signal Data

You could add a To Workspace block to the output from the Switch block. Since the model already has a To Workspace block, the second To Workspace block would save data to a signal array named `simout2` The `values` field logs the actual signal values. If logged signal data is smaller than the maximum size, values are padded with NaNs or appropriate values. To obtain these signal values, type:

```
simout2.signals.values

ans(:,:,1) =

     1    -1
    -2     2
    -3     3


ans(:,:,2) =

     1    -1
    -2     2
    -3     3


ans(:,:,3) =

     0   NaN
   NaN   NaN
   NaN   NaN
```

The `valueDimensions` field logs the dimensions of a variable-size signal. To obtain the dimensions, type:

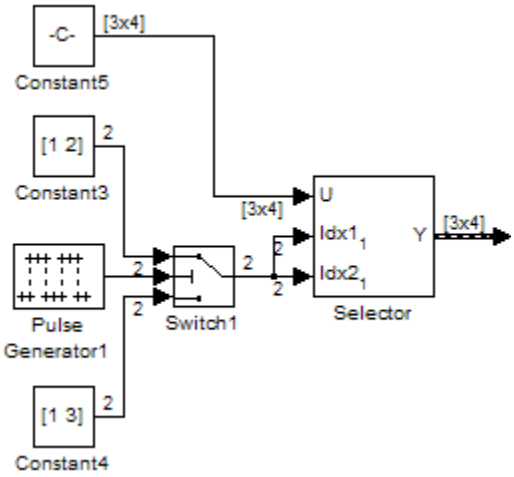
```
simout2.signals.valueDimensions
```

The signal dimensions for the first three time steps are shown.

```
ans =

     3     2
     3     2
     1     1
```

Create a Variable-Size Signal from a Single Data Signal

The data signal (Constant5) is a 3x4 matrix. The Pulse Generator represents a control signal that selects a starting and ending index value (`[1 2]` or `[1 3]`). The Selector block then uses the index values to select different parts of the data signal at each time step and output a variable-size signal.



View Changes in Signal Size

The output from the Selector block is either a 2x2 or 3x3 matrix. Because the maximum dimension for a variable-size signal is the 3x4 matrix from the data signal, the logged output signals are padded with NaNs.

Use the Probe or Width blocks to inspect the current dimensions and width of a variable-size signal. In addition, you can display variable-size signals on Scope blocks and save variable-size signals to the workspace using the To Workspace block.

Process Variable-Size Signals

The remainder of the model shows various operations that are possible with variable-size signals. Operations include using the Gain, Sum, Math Function, and Matrix Concatenate blocks. You can connect variable-size signals with the From, Goto, Bus Assignment, Bus Creator, and Bus Selector blocks.

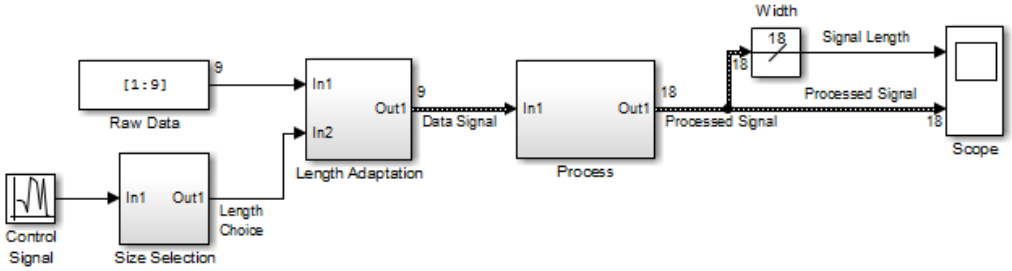
Variable-Size Signal Length Adaptation

This example model corresponds to a hypothetical system where the model adapts the length of a signal over time. Length adaptation is based on the value of a control signal. When the control signal falls within one of three predefined ranges, the fixed-size raw data signal changes to a variable-size data signal.

The variable-size signal connects to a processing block, where blocks that support variable-size signals operate on it. A MATLAB Function block with both input and output signals of variable size allow more flexibility than other blocks supporting variable-size signals. See “Simulink Block Support for Variable-Size Signals” on page 77-4.

To open the example model, in the MATLAB Command Window, type:

```
sldemo_varsize_dataLengthAdapt
```



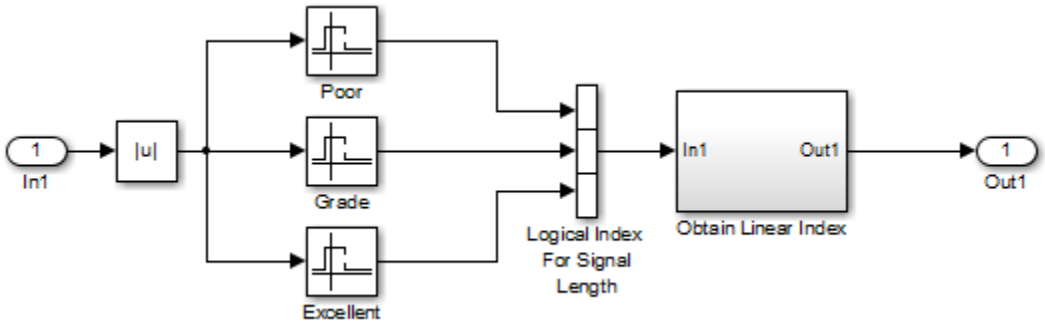
So that you can see the names of the blocks, in the model, on the **Format** tab, clear **Auto > Hide Automatic Block Names**.

Creating a Variable-Size Signal by Adapting the Length of a Data Signal

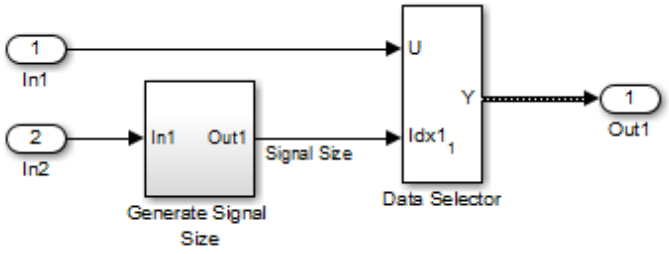
This model generates a data signal and converts the signal to a variable-size signal. The size of the signal depends on the value of a control signal. The raw data signal is a column vector with values from 1 to 9.

```
[1:9].'

ans =
     1
     2
     3
     4
     5
     6
     7
     8
     9
```

The Size Selection subsystem determines the quality of the data signal and outputs a quality value (1, 2, or 3). This value helps to select the length of the data signal in the Length Adaptation subsystem.

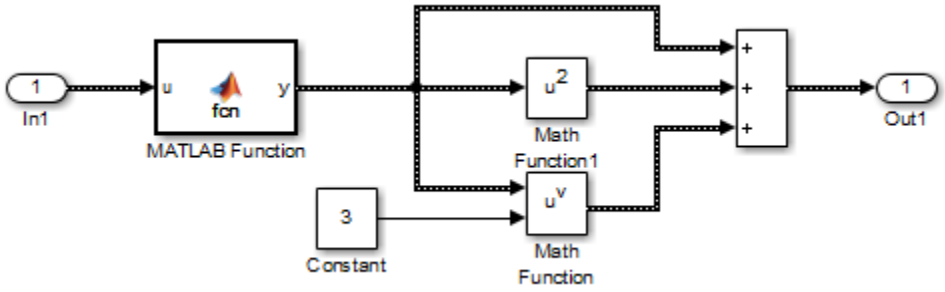


In the Length Adaptation subsystem, the Signal Size subsystem generates an index based on the quality value from the Size Selection subsystem (In2). The Data Selector block uses the starting and ending indices to adapt the length of the data signal (In1) and output a variable-size signal.



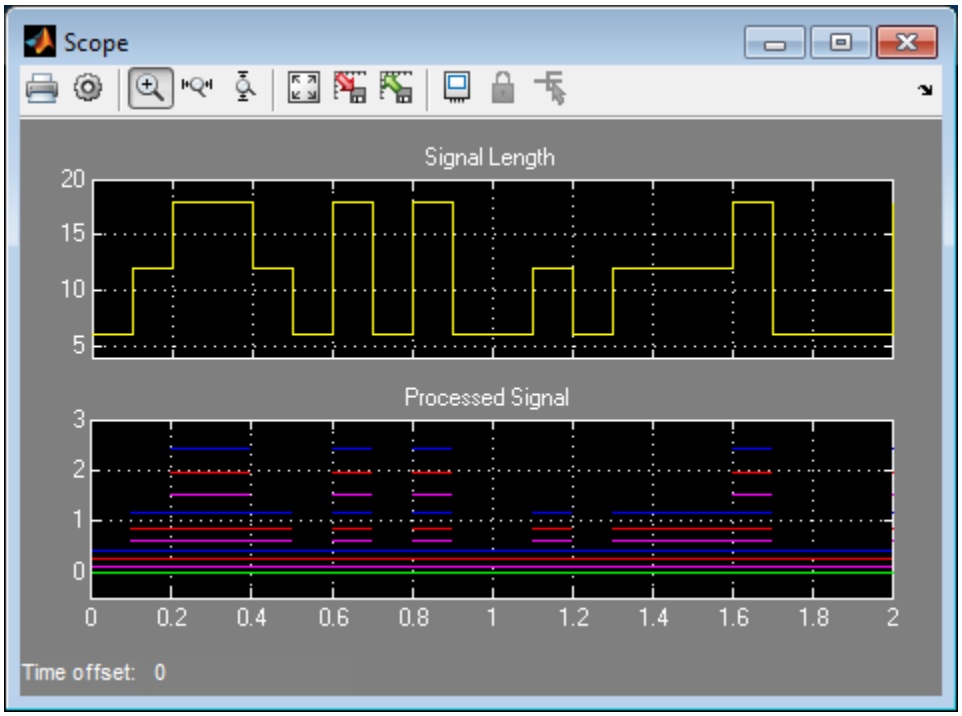
Processing a Variable-Size Signal

The center section of the model processes the variable-size signal. The MATLAB Function block adds zeros between the data values in a way that is similar to upsampling a signal. The dimension of the signal changes from 9 to 18. The Math Function blocks shows various manipulations you can do with variable-size signals.



Visualizing a Variable-Size Signal

The right section of the model determines the signal width (size) and uses a scope to visualize the width and the processed data signal.



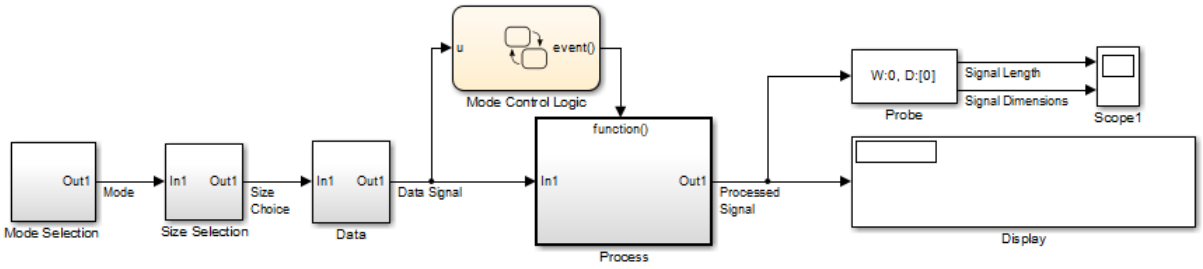
Mode-Dependent Variable-Size Signals

This example model represents a system that has three operation modes. For each mode, the data signal to process has a different size.

The Process subsystem in this model receives a variable-size signal where the size of the signal depends on the operation mode of the system. For each mode change, the Stateflow chart, Mode Control Logic, detects when the data signal size changes. It then generates a function call to reset the blocks in the Process subsystem.

To open the model, In the MATLAB Command Window, type:

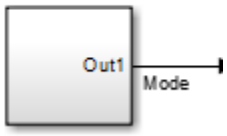
```
sldemo_varsize_multimode
```



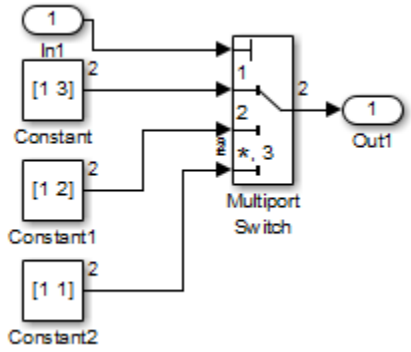
So that you can see the names of the blocks, in the model, on the **Format** tab, clear **Auto > Hide Automatic Block Names**.

Creating a Variable-Size Signal Based on Mode

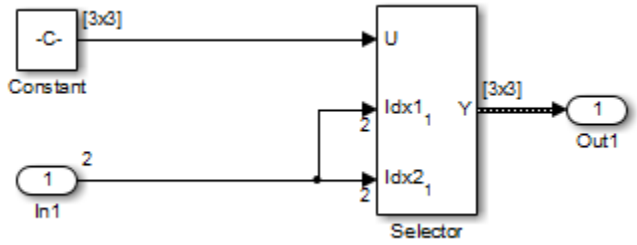
The Mode Selection subsystem determines the mode for processing a data signal and outputs a mode value (1, 2, or 3). This value helps to select the length of the data signal using the Size Selection and Data subsystems.



The Size Selection subsystem creates an index value from the mode value. In this example, the index values are `[1 3]`, `[1 2]`, and `[1 1]`.



The Data subsystem takes a data signal (Constant block) and selects part of the data signal dependent on the mode. The output is a variable-size signal with a matrix size of 3x3, 2x2, and 1x1.



The dimensions of the raw data signal (Constant block) is a 3x3. After connecting a To Workspace block to a signal line, you can view the signal in the MATLAB Command Window by typing:

```
simout.signals.values

ans(:,:,1) =

     1     4     7
     2     5     8
     3     6     9
```

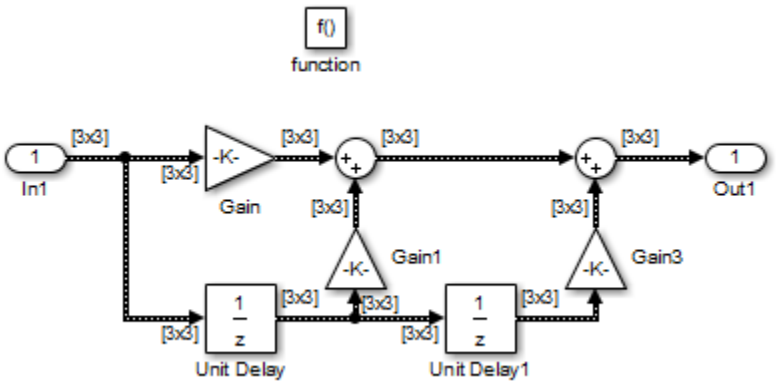
The variable-size signal generated from the Data subsystem is also a 3x3 matrix. For shorter signals, the matrix is padded with NaNs.

```
simout.signals.values

ans(:,:,1) =

     1   NaN   NaN
   NaN   NaN   NaN
   NaN   NaN   NaN


ans(:,:,2) =

     1     4   NaN
     2     5   NaN
   NaN   NaN   NaN


ans(:,:,3) =

     1     4     7
     2     5     8
     3     6     9
```

Processing a Variable-Size Signal with a Conditionally Executed Subsystem

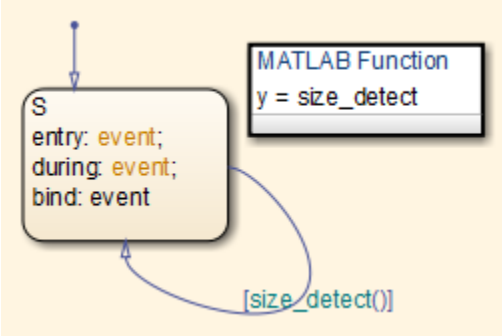
Because the Process subsystem contains a Delay block, the subsystem resets and repropagates the signal at each time step. This model uses a Stateflow chart to detect a signal size change and reset the Process subsystem.

In the function block dialog, and from the **Propagate sizes of variable-size signals** list, choose `Only when enabling`. When the model enables this subsystem, selecting this option directs the Simulink software to propagate sizes for variable-size signals inside the conditionally executed subsystem. Signal sizes can change only when they transition from disabled to enabled. For an explanation of handling signal-size changes with blocks containing states, see “How Variable-Size Signals Propagate” on page 77-2.



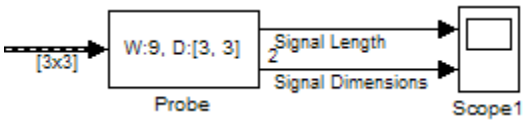
The Stateflow chart determines if there is a change in the size of the signal. The function `size_detect` calculates the width of the variable-size signal at each time step, and compares the

current width to the previous width. If there is a change in signal size, the chart outputs a function-call output event that resets and repropagates the signal sizes within the Process subsystem.

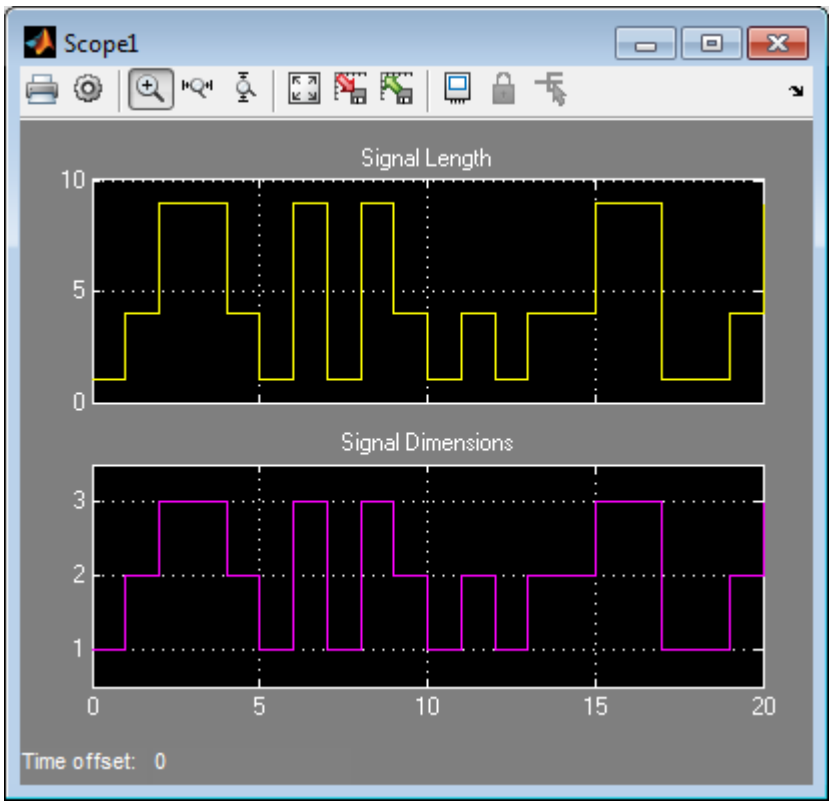


Visualizing Data

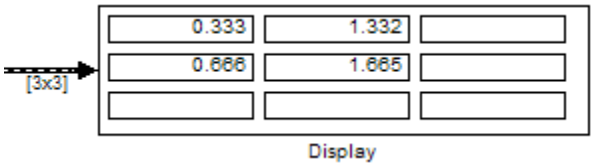
Use the Probe block to visualize signal size and signal dimension.



Since the signals are `n x n` matrices, the signal dimension lines overlap in the Scope display.



You can use a Display block and the Simulink Debugger to visualize signal values at each time step.



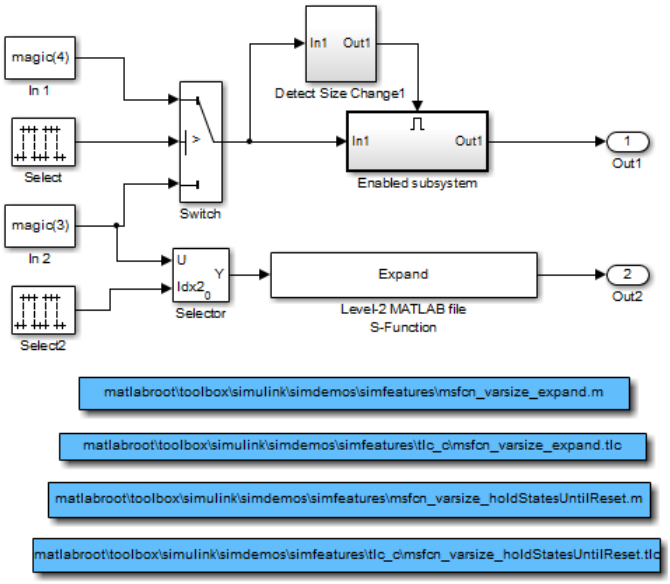
S-Functions Using Variable-Size Signals

Level-2 MATLAB S-Function with Variable-Size Signals

Both Level-2 MATLAB S-Functions and C S-Functions support variable-size signals when you set the **DimensionMode** for the output port to `Variable`. You also need to consider the current dimension of the input and output signals in the input and output update methods.

To open this example model, in the MATLAB Command Window, type:

```
msfcndemo_varsize
```



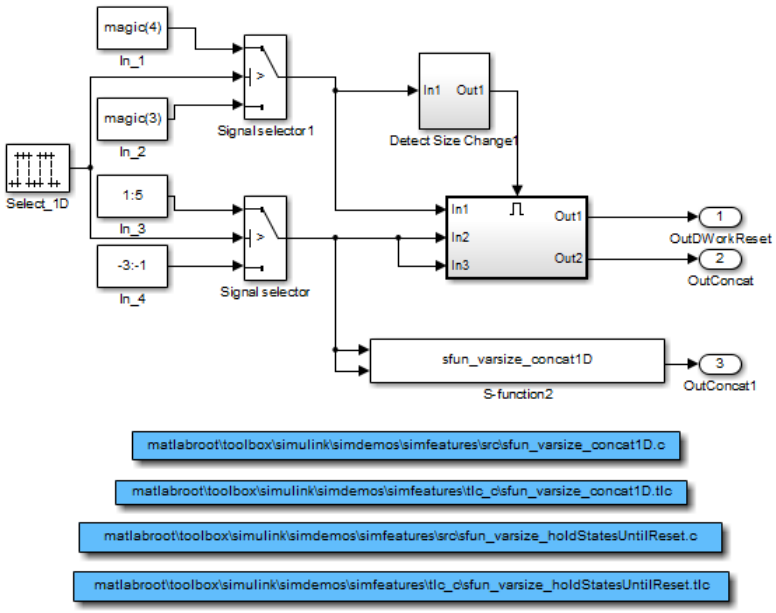
The Enabled subsystem includes a Level-2 MATLAB S-Function which shows how to implement a block that holds its states until reset. Because this block contains states and delays the input signal, the input size can change only when a reset occurs.

The Expand block is a Level-2 MATLAB S-Function that takes a scalar input and outputs a vector of length indicated by its input value. The output is by `1:n` where `n` is the input value.

C S-Function with Variable-Size Signals

To open this example model, in the MATLAB Command Window, type:

```
sfcndemo_varsize
```



The enabled subsystems have two S-Functions:

- sfun_varsize_holdStatesUntilReset is a C S-Function that has states and requires its DWorks vector to reset whenever the sizes of the input signal changes.
- sfun_varsize_concat1D is a C S-function that implements the concatenation of two unoriented vectors. You can use this function within an enabled subsystem by itself.

See Also

Related Examples

- “Parallel Channel Power Allocation”
- “Variable-Size Signal Basics” on page 77-2
- “S-Functions Using Variable-Size Signals” on page 77-18
- “Simulink Block Support for Variable-Size Signals” on page 77-4
- “Variable-Size Signal Limitations” on page 77-6

Customizing Simulink Environment and Printed Models

Customizing the Simulink User Interface

Access Frequently Used Features and Commands in Simulink

The Simulink quick access toolbar provides access to frequently used operations and favorite commands. This toolbar is always visible, even when you navigate between different Simulink Toolstrip tabs.

To customize the quick access toolbar, you can:

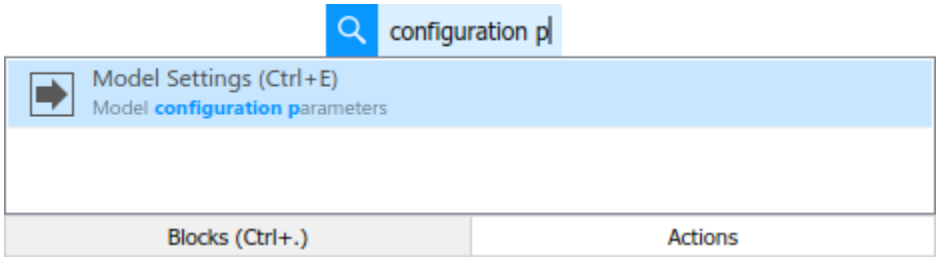
- Right-click an item in the toolstrip and select **Add to Quick Access Toolbar**. To remove a toolstrip item you have added, right-click the item and select **Remove from Quick Access Toolbar**.
- Drag items to change their position.
- Show item labels by right-clicking the item icon and selecting **Show Label**.
- Show or hide common items by clicking the button and selecting or clearing the corresponding check boxes.

To restore default settings for the quick access toolbar, use one of these options:

- Right-click an item and select **Restore Defaults** .
- Click the button and select **Restore Defaults**.

Search for Simulink Toolstrip Actions

To search for Simulink Toolstrip actions, click the **Search Toolstrip** button . In the search box that appears, start typing the name or description of the toolstrip item, then select it from the menu.

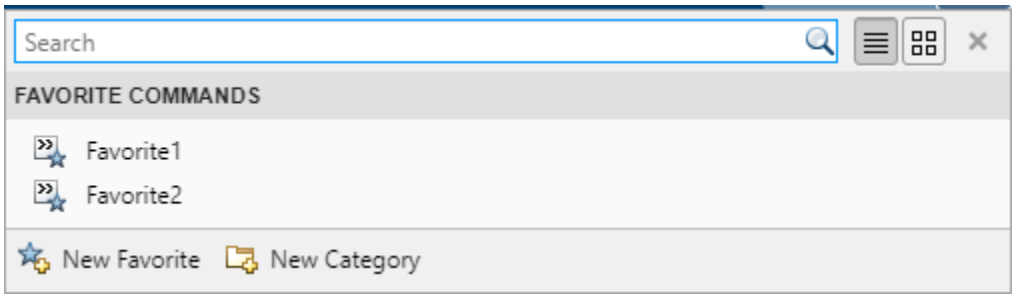


If the **Search Toolstrip** button is not in the quick access toolbar, click the button and select the **Search Toolstrip** check box.

Alternatively, you can open the action search menu by double-clicking in the Simulink Editor canvas and selecting the **Actions** tab or by using the **Ctrl + . (period)** keyboard shortcut. For more information, see “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61.

Rerun Favorite Commands for Simulink

The Simulink quick access toolbar provides you with a place to add, edit, organize, run, and delete your favorite MATLAB commands. Click the **Favorites** button  to open the **Favorite Commands** dialog box.



The list of favorite commands is independent of the list of favorite commands in the MATLAB quick access toolbar.

To create a favorite command:

1. Click **New Favorite**. The Favorite Command Editor dialog box opens.
2. In the **Label** box, enter a name for the favorite command.
3. In the **Code** box, type the statements you want the favorite command to run.
4. Set **Category** to an existing category from the dropdown list.
5. Click **OK**.
6. To run the statements in the **Code** box and ensure that they perform the desired actions, click the button for the new command. All the statements in the **Code** box of the Favorite Command Editor execute as if you ran those statements from the Command Window, although they do not appear in the Command History window.

To edit or delete a favorite command, right-click the command and choose the corresponding option.

To create a new category for your favorite commands:

1. Click **New Category**. The Favorite Category Editor dialog box opens.
2. In the **Label** box, enter a name for the category.
3. Click **OK**.

To edit or delete a category, right-click the category and choose the corresponding option.

See Also

More About

- “Keyboard Shortcuts and Mouse Actions for Simulink Modeling” on page 1-61
- “Set Simulink Preferences”
- “Improve Quick Block Insert Results” on page 78-22

Add Items to Model Editor Menus

In this section...

You can add commands and submenus to a menu bar and context menus for the Simulink Editor and Stateflow Editor.

To add an item to a menu:

- For each item, create a function, called a schema function, that defines the item (see “Define Menu Items” on page 78-5).
- Register the menu customizations with the Simulink customization manager at startup, e.g., in an `sl_customization.m` file on the MATLAB path (see “Register Menu Customizations” on page 78-9).
- Create callback functions that implement the commands triggered by the items that you add to the menus.

Code for Adding Menu Items

The following `sl_customization.m` file adds four items to the Simulink Editor **Tools** menu.

```
function sl_customization(cm)

  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyMenuItems);
end

%% Define the custom menu function.
function schemaFcns = getMyMenuItems(callbackInfo)
  schemaFcns = {@getItem1, ...
                @getItem2, ...
                {@getItem3,3}, ... %% Pass 3 as user data to getItem3.
                @getItem4};
end

%% Define the schema function for first menu item.
function schema = getItem1(callbackInfo)
  schema = sl_action_schema;
  schema.label = 'Item One';
  schema.userdata = 'item one';
  schema.callback = @myCallback1;
end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata ' was called']);
end

function schema = getItem2(callbackInfo)
  % Make a submenu label 'Item Two' with
  % the menu item above three times.
  schema = sl_container_schema;
```

```
  schema.label = 'Item Two';
    schema.childrenFcns = {@getItem1, @getItem1, @getItem1};
end

function schema = getItem3(callbackInfo)
  % Create a menu item whose label is
  % 'Item Three: 3', with the 3 being passed
  % from getMyItems above.

  schema = sl_action_schema;
  schema.label = ['Item Three: ' num2str(callbackInfo.userdata)];
end

function myToggleCallback(callbackInfo)
    if strcmp(get_param(gcs, 'ScreenColor'), 'red') == 0
        set_param(gcs, 'ScreenColor', 'red');
    else
        set_param(gcs, 'ScreenColor', 'white');
    end
end

%% Define the schema function for a toggle menu item.
function schema = getItem4(callbackInfo)
  schema = sl_toggle_schema;
  schema.label = 'Red Screen';
  if strcmp(get_param(gcs, 'ScreenColor'), 'red') == 1
    schema.checked = 'checked';
  else
    schema.checked = 'unchecked';
  end
  schema.callback = @myToggleCallback;
end
```

Define Menu Items

You define a menu item by creating a function that returns an object, called a *schema* object, that specifies the information needed to create the menu item. The menu item that you define may trigger a custom action or display a custom submenu. See the following sections for more information.

- “Defining Menu Items That Trigger Custom Commands” on page 78-5
- “Defining Custom Submenus” on page 78-8

Defining Menu Items That Trigger Custom Commands

To define an item that triggers a custom command, your schema function must accept a callback info object (see “Callback Info Object” on page 78-10) and create and return an action schema object (see “Action Schema Object” on page 78-6) that specifies the item label and a function, called a *callback*, to be invoked when the user selects the item. For example, the following schema function defines a menu item that displays a message when selected by the user.

```
function schema = getItem1(callbackInfo)

  %% Create an instance of an action schema.
  schema = sl_action_schema;


%% Specify the menu item label.
  schema.label = 'My Item 1';
  schema.userdata = 'item1';
  %% Specify the menu item callback function.
  schema.callback = @myCallback1;
```

```
end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata
        ' was called']);
end
```

Action Schema Object

This object specifies information about menu items that trigger commands that you define, including the label that appears on the menu item and the function to be invoked when the user selects the menu item. Use the function `sl_action_schema` to create instances of this object in your schema functions. Its properties include:

- `tag`

  Optional character vector that identifies this action, for example, so that it can be referenced by a filter function.

- `label`

  Character vector specifying the label that appears on a menu item that triggers this action.

- `state`

  Property that specifies the state of this action. Valid values are `'Enabled'` (the default), `'Disabled'`, and `'Hidden'`.

- `statustip`

  Character vector specifying text to appear in the editor status bar when the user selects the menu item that triggers this action.

- `userdata`

  Data that you specify. May be of any type.

- `accelerator`

  Character vector specifying a **Ctrl** and key combination to use to trigger this action. You can add a keyboard shortcut only for custom menu items that appear on menu bar menus and cannot redefine accelerators that come with the Simulink Editor. For example, **Ctrl+D** is an accelerator for **Update Diagram** in the Simulink Editor, so you cannot redefine it.

  To specify this value, use the form `'Ctrl+`*K*`'`, where *K* is the shortcut key. For example, use `'Ctrl+Alt+T'` for an accelerator invoked by holding down **Ctrl** and **Alt** and pressing **T**.

- `callback`

  Character vector specifying a MATLAB expression to be evaluated or a handle to a function to be invoked when a user selects the menu item that triggers this action. This function must accept one argument: a callback info object.

- `autoDisableWhen`

  Property that controls when a menu item is automatically disabled.

Setting	When Menu Items Are Disabled
`'Locked'`	(default) When the active editor is locked or when the model is busy
`'Busy'`	Only if the model is busy
`'Never'`	Never

Toggle Schema Object

This object specifies information about a menu item that toggles some object on or off. Use the function `sl_toggle_schema` to create instances of this object in your schema functions. Its properties include:

- `tag`

  Optional character vector that identifies this toggle action, for example, so that it can be referenced by a filter function.

- `label`

  Character vector specifying the label that appears on a menu item that triggers this toggle action.

- `checked`

  Specify whether the menu item displays a check mark. Valid values are `'unchecked'` (default) and `'checked'`

- `state`

  State of this toggle action, specified as `'Enabled'` (default), `'Disabled'`, and `'Hidden'`.

- `statustip`

  Character vector specifying text to appear in the editor status bar when the user selects the menu item that triggers this toggle action.

- `userdata`

  Data that you specify. May be of any type.

- `accelerator`

  Character vector specifying a **Ctrl** and key combination to use to trigger this action. You can add a keyboard shortcut only for custom menu items that appear on menu bar menus and cannot redefine accelerators that come with the Simulink Editor. For example, **Ctrl+D** is an accelerator for **Update Diagram** in the Simulink Editor, so you cannot redefine it.

  To specify this value, use the form `'Ctrl+`*K*`'`, where *K* is the shortcut key. For example, use `'Ctrl+Alt+T'` for an accelerator invoked by holding down **Ctrl** and **Alt** and pressing **T**.

- `callback`

  Character vector specifying a MATLAB expression to be evaluated or a handle to a function to be invoked when a user selects the menu item that triggers this action. This function must accept one argument: a callback info object.

- `autoDisableWhen`

  Property that controls when a menu item is automatically disabled.

Setting	When Menu Items Are Disabled
`'Locked'`	(default) When the active editor is locked or when the model is busy
`'Busy'`	Only if the model is busy
`'Never'`	Never

Defining Custom Submenus

To define a submenu, create a schema function that accepts a callback info object and returns a container schema object (see “Container Schema Object” on page 78-8) that specifies the schemas that define the items on the submenu. For example, the following schema function defines a submenu that contains three instances of the menu item defined in the example in “Defining Menu Items That Trigger Custom Commands” on page 78-5.

```
function schema = getItem2( callbackInfo )
    schema = sl_container_schema;
    schema.label = 'Item Two';
    schema.childrenFcns = {@getItem1, @getItem1, @getItem1};
end
```

Container Schema Object

A container schema object specifies a submenu label and its contents. Use the function `sl_container_schema` to create instances of this object in your schema functions. Properties of the object include

- `tag`

  Optional character vector that identifies this submenu.
- `label`

  Character vector specifying the submenu label.
- `state`

  Character vector that specifies the state of this submenu. Valid values are `'Enabled'` (the default), `'Disabled'`, and `'Hidden'`.
- `statustip`

  Character vector specifying text to appear in the editor status bar when the user selects this submenu.
- `childrenFcns`

  Cell array that specifies the contents of the submenu. Each entry in the cell array can be

  - A pointer to a schema function that defines an item on the submenu (see “Define Menu Items” on page 78-5)
  - A two-element cell array whose first element is a pointer to a schema function that defines an item entry and whose second element is data to be inserted as user data in the callback info object (see “Callback Info Object” on page 78-10) passed to the schema function
  - `'separator'`, which causes a separator to appear between the item defined by the preceding entry in the cell array and the item defined in the following entry. The case is ignored for this entry (for example, `'SEPARATOR'` and `'Separator'` are both valid entries). A separator is

also suppressed if it appears at the beginning or end of the submenu and separators that would appear successively are combined into a single separator (for example, as a result of an item being hidden).

For example, the following cell array specifies two submenu entries:

```
{@getItem1, 'separator', {@getItem2, 1}}
```

In this example, a 1 is passed to `getItem2` via a callback info object.

- `generateFcn`

  Pointer to a function that returns a cell array defining the contents of the submenu. The cell array must have the same format as that specified for the container schema objects `childrenFcns` property.

  Note The `generateFcn` property takes precedence over the `childrenFcns` property. If you set both, the `childrenFcns` property is ignored and the cell array returned by the `generateFcn` is used to create the submenu.

- `userdata`

  Data of any type that is passed to `generateFcn`.

- `autoDisableWhen`

  Property that controls when a menu item is automatically disabled.

Setting	When Menu Items Are Disabled
`'Locked'`	(default) When the active editor is locked or when the model is busy
`'Busy'`	Only if the model is busy
`'Never'`	Never

Register Menu Customizations

You must register custom items to be included on a Simulink menu with the customization manager. Use the `sl_customization.m` file for a Simulink installation (see “Registering Customizations” on page 78-23) to perform this task. In particular, for each menu that you want to customize, your system `sl_customization` function must invoke the customization manager `addCustomMenuFcn` method. Each invocation should pass the tag of the menu (see “Menu Tags” on page 78-10) to be customized and a custom menu function that specifies the items to be added to the menu (see “Creating the Custom Menu Function” on page 78-9). For example, the following `sl_customization` function adds custom items to the Simulink Tools menu.

```
function sl_customization(cm)
  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyItems);
```

Creating the Custom Menu Function

The custom menu function returns a cell array of schema functions that define custom items that you want to appear on the model editor menus (see “Define Menu Items” on page 78-5). The custom menu function returns a cell array similar to that returned by the `generateFcn` function.

Your custom menu function should accept a callback info object (see “Callback Info Object” on page 78-10) and return a cell array that lists the schema functions. Each element of the cell array can be either a handle to a schema function or a two-element cell array whose first element is a handle to a schema function and whose second element is user-defined data to be passed to the schema function. For example, the following custom menu function returns a cell array that lists three schema functions.

```
function schemas = getMyItems(callbackInfo)
  schemas = {@getItem1, ...
             @getItem2, ...
             {@getItem3,3} }; % Pass 3 as userdata to getItem3.
end
```

Callback Info Object

Instances of these objects are passed to menu customization functions. Methods and properties of these objects include:

- `uiObject`

  Method to get the handle to the owner of the menu for which this is the callback. The owner can be the Simulink Editor or the Stateflow Editor.

- `model`

  Method to get the handle to the model being displayed in the editor window.

- `userdata`

  User data property. The value of this property can be any type of data.

Debugging Custom Menu Callbacks

On systems using the Microsoft Windows operating system, selecting a custom menu item whose callback contains a breakpoint can cause the mouse to become unresponsive or the menu to remain open and on top of other windows. To fix these problems, use the MATLAB code debugger keyboard commands to continue execution of the callback.

Menu Tags

A menu tag identifies a Simulink Editor or the Stateflow Editor menu bar or menu. You need to know the tag for a menu to add custom items to it (see “Register Menu Customizations” on page 78-9).

Tag	What It Adds
Simulink tags	
`Simulink:MenuBar`	Menu to the Simulink Editor menu bar
`Simulink:PreContextMenu`	Item to the beginning of a Simulink Editor context menu
`Simulink:ContextMenu`	Item to the end of a Simulink Editor context menu
`Simulink:FileMenu`	Item to the end of a Simulink Editor **File** menu
`Simulink:EditMenu`	Item to the end of a Simulink Editor **Edit** menu

Tag	What It Adds
`Simulink:ViewMenu`	Item to the end of a Simulink Editor **View** menu
`Simulink:DisplayMenu`	Item to the end of a Simulink Editor **Display** menu
`Simulink:DiagramMenu`	Item to the end of a Simulink Editor **Diagram** menu
`Simulink:SimulationMenu`	Item to the end of a Simulink Editor **Simulation** menu
`Simulink:AnalysisMenu`	Item to the end of a Simulink Editor **Analysis** menu
`Simulink:CodeMenu`	Item to the end of a Simulink Editor **Code** menu
`Simulink:ToolsMenu`	Item to the end of a Simulink Editor **Tools** menu
`Simulink:HelpMenu`	Item to the end of a Simulink Editor **Help** menu
Stateflow tags	
`Stateflow:MenuBar`	Menu to the Stateflow Editor menu bar
`Stateflow:PreContextMenu`	Item to the beginning of a Stateflow Editor context menu.
`Stateflow:ContextMenu`	Items to the end of a Stateflow Editor context menu.
`Stateflow:FileMenu`	Item to the end of a Stateflow Editor **File** menu
`Stateflow:EditMenu`	Item to the end of a Stateflow Editor **Edit** menu
`Stateflow:ViewMenu`	Item to the end of a Stateflow Editor **View** menu
`Stateflow:DisplayMenu`	Item to the end of a Stateflow Editor **Display** menu
`Stateflow:ChartMenu`	Item to the end of a Stateflow Editor **Chart** menu
`Stateflow:SimulationMenu`	Item to the end of a Stateflow Editor **Simulation** menu
`Stateflow:AnalysisMenu`	Item to the end of a Stateflow Editor **Analysis** menu
`Stateflow:CodeMenu`	Item to the end of a Stateflow Editor **Code** menu
`Stateflow:ToolsMenu`	Item to the end of a Stateflow Editor **Tools** menu
`Stateflow:HelpMenu`	Item to the end of a Stateflow Editor **Help** menu

Simulink and Stateflow Editor Menu Customization

Use the same general procedures to customize Stateflow Editor menus as you use for Simulink Editor. The addition of custom menu functions to the ends of top-level menus depends on the active editor:

- Menus bound to `Simulink:FileMenu` only appear when the Simulink Editor is active.
- Menus bound to `Stateflow:FileMenu` only appear when the Stateflow Editor is active.
- To have a menu to appear in both of the editors, call `addCustomMenuFcn` twice, once for each tag. Check that the code works in both editors.

See Also

Related Examples

- “Disable and Hide Model Editor Items” on page 78-13
- “Disable and Hide Dialog Box Controls” on page 78-15
- “Customize Library Browser Appearance” on page 78-19

- “Registering Customizations” on page 78-23

Disable and Hide Model Editor Items

You can disable items that appear on the Simulink Toolstrip and context menus. You can also hide items that appear on context menus. To disable or hide an item, you must:

- Create a filter function that disables or hides the item (see “Create a Filter Function” on page 78-13).
- Register the filter function with the customization manager (see “Register a Filter Function” on page 78-13).

Example: Disable the New Model Command in the Simulink Toolstrip

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end

function state = myFilter(callbackInfo)
  state = 'Disabled';
end
```

Create a Filter Function

Your filter function must accept a callback info object and return the state that you want to assign to the item. Valid states are:

- `'Hidden'`
- `'Disabled'`
- `'Enabled'`

Your filter function may have to compete with other filter functions and with Simulink itself to assign a state to an item. Who succeeds depends on the strength of the state that each assigns to the item.

- `'Hidden'` is the strongest state. If any filter function or Simulink assigns `'Hidden'` to a menu item, it is hidden. For Simulink Toolstrip items, specifying `'Hidden'` disables the item instead of hiding it.
- `'Disabled'` overrides `'Enabled'`, but is itself overridden by `'Hidden'`.
- `'Enabled'` is the weakest state. For an item to be enabled, all filter functions and the Simulink or Stateflow products must assign `'Enabled'` to the item.

Register a Filter Function

Use the customization manager `addCustomFilterFcn` method to register a filter function. The `addCustomFilterFcn` method takes two arguments: a tag that identifies the menu or item to be filtered and a pointer to the filter function itself. For example, the following code registers a filter function for the **New Model** item on the Simulink Toolstrip.

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end
```

See Also

Related Examples

- “Add Items to Model Editor Menus” on page 78-4
- “Disable and Hide Dialog Box Controls” on page 78-15
- “Customize Library Browser Appearance” on page 78-19
- “Registering Customizations” on page 78-23

Disable and Hide Dialog Box Controls

In this section...

About Disabling and Hiding Controls

Simulink includes a customization API that allows you to disable and hide controls (also referred to as widgets), such as text fields and buttons, on most dialog boxes. The customization API allows you to disable or hide controls on an entire class of dialog boxes, for example, parameter dialog boxes, by way of a single method call.

Before you customize a Simulink dialog box or class of dialog boxes, first make sure that the dialog box or class of dialog boxes is customizable. Any dialog box that appears in the dialog pane of Model Explorer is customizable. In addition, any dialog box that has dialog and widget IDs is customizable. To determine whether a dialog box is customizable, open the dialog box, enable dialog and widget ID display (see “Widget IDs” on page 78-16), and hover over a widget. If a widget ID appears, you can customize the dialog box.

Once you have determined that a dialog box or class of dialog boxes is customizable, write MATLAB code to customize the dialog boxes. This entails writing callback functions that disable or hide controls for a specific dialog box or class of dialog boxes (see “Write Control Customization Callback Functions” on page 78-16) and registering the callback functions using the customization manager (see “Register Control Customization Callback Functions” on page 78-17). Simulink invokes the callback functions to disable or hide the controls whenever you open the dialog boxes.

Disable a Button on a Dialog Box

This `sl_customization.m` file disables the **Browse** button on the **Code Generation** pane of the Configuration Parameters dialog box for any model whose name contains `engine`.

```
function sl_customization(cm)

% Disable for standalone Configuration Parameters dialog box
configset.dialog.Customizer.addCustomization(@disableRTWBrowseButton,cm);

end

function disableRTWBrowseButton(dialogH)
  hSrc   = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName   = get_param(hModel,'Name');

  if ~isempty(strfind(modelName,'engine'))
    % Takes a cell array of widget Factory ID.
    dialogH.disableWidgets({'STF_Browser'})
  end

end
```

To test this customization:

1. Save the `sl_customization.m` file on the MATLAB path.
2. Refresh the customizations by entering `sl_refresh_customizations` at the command line or by restarting MATLAB (see “Registering Customizations” on page 78-23).
3. Open the `sldemo_engine` model, for example, by entering the command `sldemo_engine` at the command prompt.
4. Open the Configuration Parameters dialog box and look at the **Code Generation** pane to see if the **Browse** button is disabled.

Write Control Customization Callback Functions

A callback function for disabling or hiding controls on a dialog box accepts one argument: a handle to the dialog box object that contains the controls you want to disable or hide. The dialog box object provides methods that the callback function can use to disable or hide the controls that the dialog box contains.

The dialog box object also provides access to objects containing information about the current model. Your callback function can use these objects to determine whether to disable or hide controls. For example, this callback function uses these objects to disable the **Browse** button on the **Code Generation** pane of the Configuration Parameters dialog box for any model whose name contains `engine`.

```
function disableRTWBrowseButton(dialogH)

  hSrc = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName = get_param(hModel,'Name');

  if ~isempty(strfind(modelName,'engine'))
     % Takes a cell array of widget Factory ID.
     dialogH.disableWidgets({'STF_Browser'})
  end
```

Dialog Box Methods

Dialog box objects provide these methods for enabling, disabling, and hiding controls:

- `disableWidgets(widgetIDs)`
- `hideWidgets(widgetIDs)`

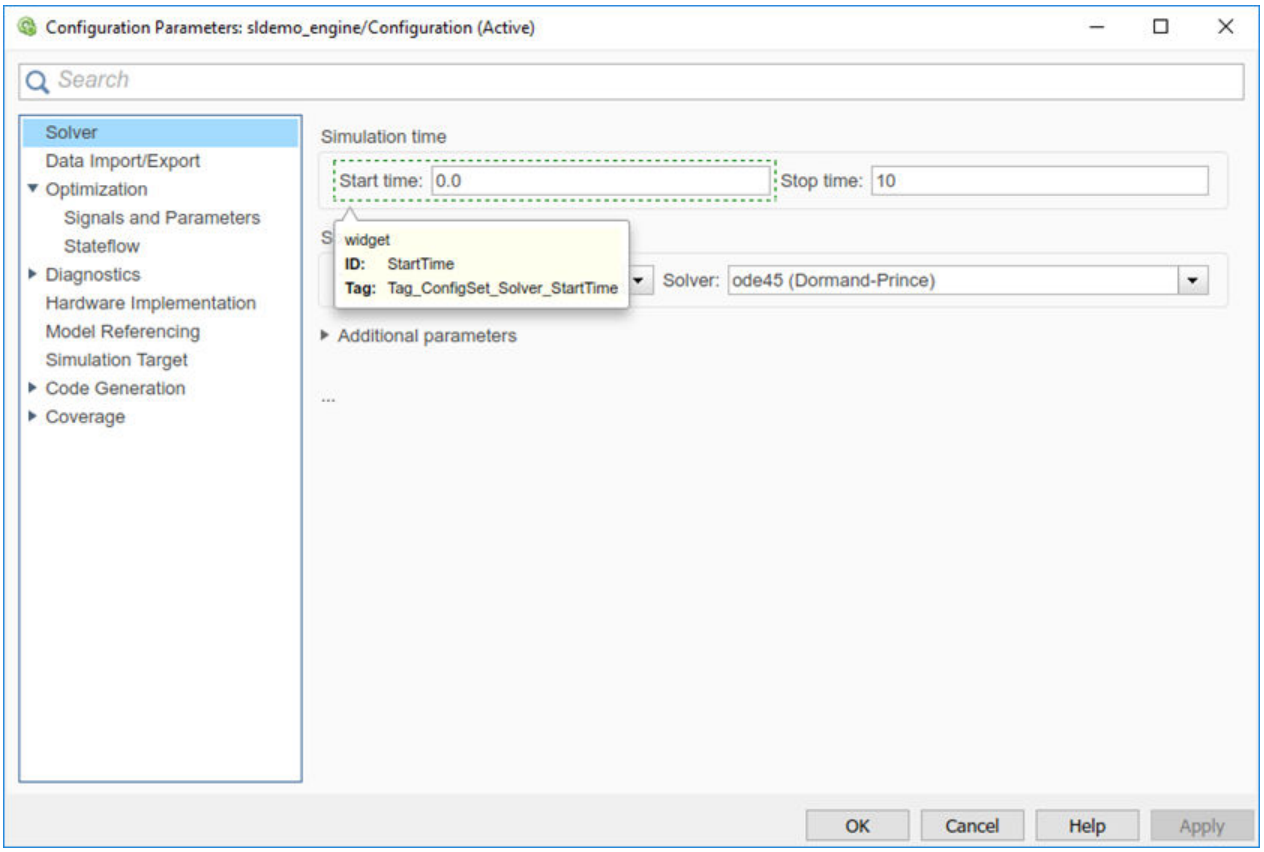
`widgetIDs` is a cell array of widget identifiers (see “Widget IDs” on page 78-16) that specify the widgets to disable or hide.

Widget IDs

Widget IDs identify a control on a Simulink dialog box. To determine the widget ID for a particular control, execute the following code at the command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip = true
```

Open the dialog box that contains the control and hover over the control to display a tooltip listing the widget ID. For example, hovering over the **Start time** field on the **Solver** pane of the Configuration Parameters dialog box shows that the widget ID for the **Start time** field is `StartTime`.



Note The tooltip displays `not customizable` for controls that are not customizable.

Register Control Customization Callback Functions

To register control customization callback functions for an installation of Simulink, include code in the installation `sl_customization.m` file (see “Registering Customizations” on page 78-23) that invokes the `configset.dialog.Customizer.addCustomization` method on the callbacks.

This method takes as an argument a pointer to the callback function to register. Invoking this method causes the registered function to be invoked before the dialog box is opened.

This example registers a callback that disables the **Browse** button on the **Code Generation** pane of the Configuration Parameters dialog box (see “Write Control Customization Callback Functions” on page 78-16).

```
function sl_customization(cm)

% Disable for standalone Configuration Parameters dialog box
configset.dialog.Customizer.addCustomization(@disableRTWBrowseButton,cm);

end
```

Note Registering a customization callback causes Simulink to invoke the callback for every instance of the class of dialog boxes specified by the method dialog box ID argument. You can therefore use a single callback to disable or hide a control for an entire class of dialog boxes. In particular, you can use a single callback to disable or hide the control for a parameter that is common to most built-in blocks. Most built-in block dialog boxes are instances of the same dialog box super class.

See Also

Related Examples

- “Add Items to Model Editor Menus” on page 78-4
- “Disable and Hide Model Editor Items” on page 78-13
- “Customize Library Browser Appearance” on page 78-19
- “Registering Customizations” on page 78-23

Customize Library Browser Appearance

In this section...

Reorder Libraries

The library name and sort priority determines its order in the tree view of the Library Browser. Libraries appear in ascending order of priority. Libraries that have the same priority are sorted alphabetically.

The Simulink library has a sort priority of `-1` by default. All other libraries have a sort priority of `0` by default. These sort priorities cause the Simulink library to display first in the Library Browser by default.

You can reorder libraries by changing their sort priorities. To change library sort priorities, add code in this form to an `sl_customization.m` file on the MATLAB path:

```
cm.LibraryBrowserCustomizer.applyOrder({'LIBNAME1',PRIORITY1, ...
                                        'LIBNAME2',PRIORITY2, ...
                                         .
                                         .
                                        'LIBNAMEN',PRIORITYN});
```

`LIBNAMEn` is the name of the library (or its model file) and `PRIORITYn` is an integer indicating the library sort priority. For example, this code moves the Simulink Extras library to the top of the Library Browser tree view.

```
cm.LibraryBrowserCustomizer.applyOrder({'Simulink Extras',-2});
```

After adding or modifying the `sl_customization.m` file, enter `sl_refresh_customizations` at the MATLAB command prompt to see the customizations take effect.

For more information on the customization functions, see “Registering Customizations” on page 78-23.

Disable and Hide Libraries

To disable or hide libraries, sublibraries, or library blocks, insert code in this form in an `sl_customization.m` file (see “Registering Customizations” on page 78-23) on the MATLAB path. Blocks that you disable or hide in a library also do not appear on the quick insert menu that you invoke in the model.

```
cm.LibraryBrowserCustomizer.applyFilter({'Item1','State', ...
                                         'Item2','State', ...
                                         .
                                         .
                                         'ItemN','State'});
```

- `ItemN` is the library, sublibrary, or block to disable or hide. Specify the item in the form `'LibraryName/Sublibrary/Block'`.

  - `LibraryName` is the library name as it appears in the browser. For a custom library, you set this value in the `slblocks.m` file with the `Browser.Name` property.
  - `Sublibrary` is the name of the sublibrary or, for a custom library, a `Subsystem` block. You can specify a block inside the subsystem in your library or in a library that you open by way of the subsystem `OpenFcn` callback. See “Create a Custom Library” on page 41-2.
  - `Block` is the block name.
- `'State'` is `'Disabled'` or `'Hidden'`.

For example, this code hides the `Sources` sublibrary of the `Simulink` library and disables the `Sinks` sublibrary.

```
cm.LibraryBrowserCustomizer.applyFilter({'Simulink/Sources','Hidden'});
cm.LibraryBrowserCustomizer.applyFilter({'Simulink/Sinks','Disabled'});
```

This code disables the Sqrt block in the sublibrary opened by way of the Subsystem2 block in the custom library `'My Library'`.

```
cm.LibraryBrowserCustomizer.applyFilter(...
{'My Library/Subsystem2/Sqrt','Disabled'});
```

After adding or modifying the `sl_customization.m` file, enter `sl_refresh_customizations` at the MATLAB command prompt to see the customizations take effect.

Expand or Collapse Library in Browser Tree

You can add a customization to expand or collapse any library in the Library Browser tree by default. For example, the `Simulink` library is expanded by default. You can specify to instead collapse it by default. Add code in this form to your `sl_customization.m` file:

```
cm.LibraryBrowserCustomizer.applyNodePreference(...
{'libraryName',logical});
```

Use `true` to expand the library and `false` to collapse it.

For example, this code collapses the `Simulink` library and expands the `Simscape` library:

```
function sl_customization(cm)
  cm.LibraryBrowserCustomizer.applyNodePreference(...
{'Simulink',false,'Simscape',true});
end
```

This code collapses a custom library named `'My Library'`.

```
function sl_customization(cm)
     cm.LibraryBrowserCustomizer.applyNodePreference(...
{'My Library',false});
end
```

After adding or modifying the `sl_customization.m` file, enter `sl_refresh_customizations` at the MATLAB command prompt to see the customizations take effect.

See Also

Related Examples

- “Add Items to Model Editor Menus” on page 78-4
- “Disable and Hide Model Editor Items” on page 78-13
- “Disable and Hide Dialog Box Controls” on page 78-15
- “Registering Customizations” on page 78-23
- “Add Libraries to the Library Browser” on page 41-7

Improve Quick Block Insert Results

You can double-click the end of a signal line that you draw from an existing block to add a block. A menu of possible blocks to insert appears. For example, at the end of a signal line out of a Gain block, the menu suggests other math blocks. Simulink comes with default suggestions.

You can improve the suggestions on this menu by updating the engine with information about your models. To do so, run a command on a single model or on all the models in a folder. Information about your model designs updates the engine. You can also remove this customization using a command.

Goal	Command
Improve results based on one model	`slblocksearchdb.trainfrommodel`
Improve results based on all the models in a folder	`slblocksearchdb.trainfrommodelsindir`
Remove the effects of a single model from the results	`slblocksearchdb.untrainmodel`
Remove the effects of all the models in a folder from the results	`slblocksearchdb.untrainmodelsindir`
Revert to the default results that come with Simulink	`slblocksearchdb.untrainall`

See Also

Related Examples

- “Build and Edit a Model Interactively” on page 1-8

Registering Customizations

About Registering Customizations in Simulink

Register your customizations using the MATLAB function `sl_customization.m`. Place the function on the MATLAB path of the Simulink installation that you want to customize or in the current folder.

You can have more than one `sl_customization.m` file. The customizations in each file takes effect, with conflicts handled by each customization. For example, if you specify priorities for libraries in multiple `sl_customization` files, only one takes effect. If you add the same menu item twice, it appears twice. To ensure that customizations are loading as expected, refresh the customizations, as described in “Reading and Refreshing the Customization File” on page 78-23.

The `sl_customization` function accepts one argument: a handle to the customization manager object, that is, `cm`. For example:

```
function sl_customization(cm)
```

In your `sl_customization` function, use customization manager object properties and methods specific to your application to register customizations. You can use customization properties and methods to:

- “Add Items to Model Editor Menus” on page 78-4
- “Disable and Hide Model Editor Items” on page 78-13
- “Disable and Hide Dialog Box Controls” on page 78-15
- “Add Libraries to the Library Browser” on page 41-7
- “Customize Library Browser Appearance” on page 78-19
- “Customize Bus Object Import and Export” on page 76-51
- “Import Lookup Table Data from MATLAB” on page 38-24

Additional MathWorks products use the customization manager object and the `sl_customization.m` file. Refer to your product documentation to learn about the methods and properties that apply to your product.

Reading and Refreshing the Customization File

The `sl_customization.m` file is read when Simulink starts. If you change the `sl_customization.m` file, either restart Simulink or enter this command to see the changes:

```
sl_refresh_customizations
```

This command runs all `sl_customization.m` files on the MATLAB path and in the current folder. Some side-effects of running `sl_refresh_customizations` include:

- Rebuilding the Simulink Toolstrip
- Rebuilding all Simulink Editor menus
- Rebuilding the Library Browser menus and toolbars
- Clearing the Library Browser cache and refreshing the Library Browser
- Reloading the Viewers and Generators Manager data

See Also

Related Examples

- “Add Items to Model Editor Menus” on page 78-4
- “Disable and Hide Model Editor Items” on page 78-13
- “Disable and Hide Dialog Box Controls” on page 78-15
- “Customize Library Browser Appearance” on page 78-19
- “Add Libraries to the Library Browser” on page 41-7

Frames for Printed Models

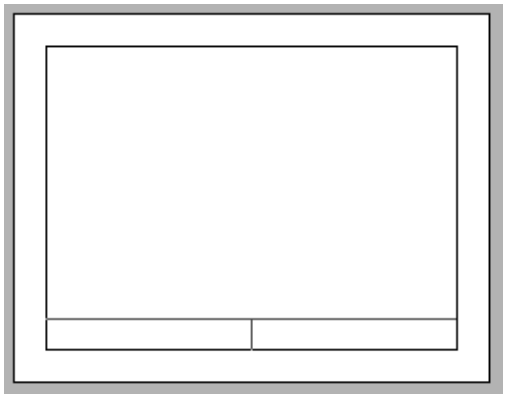
Print Frames

In this section...

What Are Print Frames?

Print frames are borders of a printed page that contain information about a block diagram, such as the model name or the date of printing. After you create a print frame, use the Simulink or Stateflow Editor to print a block diagram or chart with that print frame.

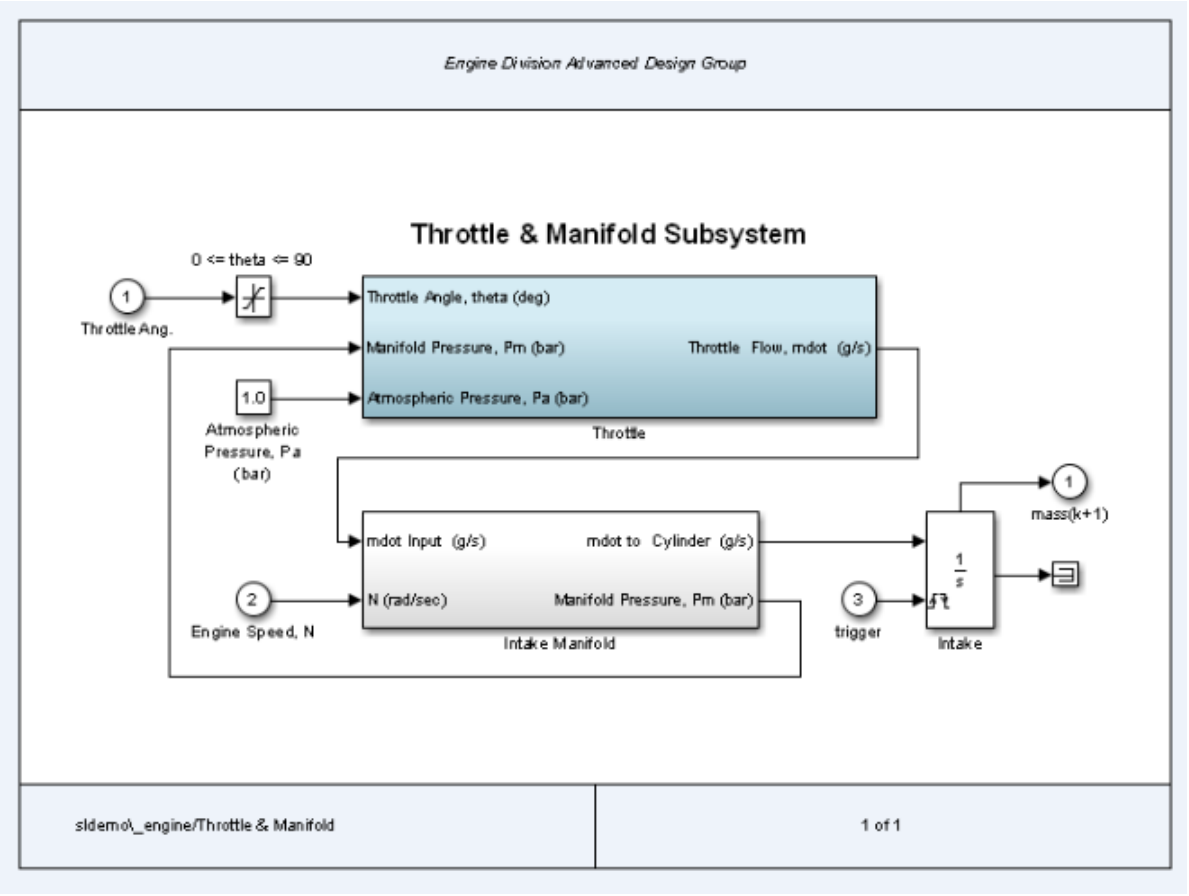
The default print frame has two rows:



Rows contain one or more cells. You can add content entries to cells. You can also add new rows and cells.

For example, the print frame below includes:

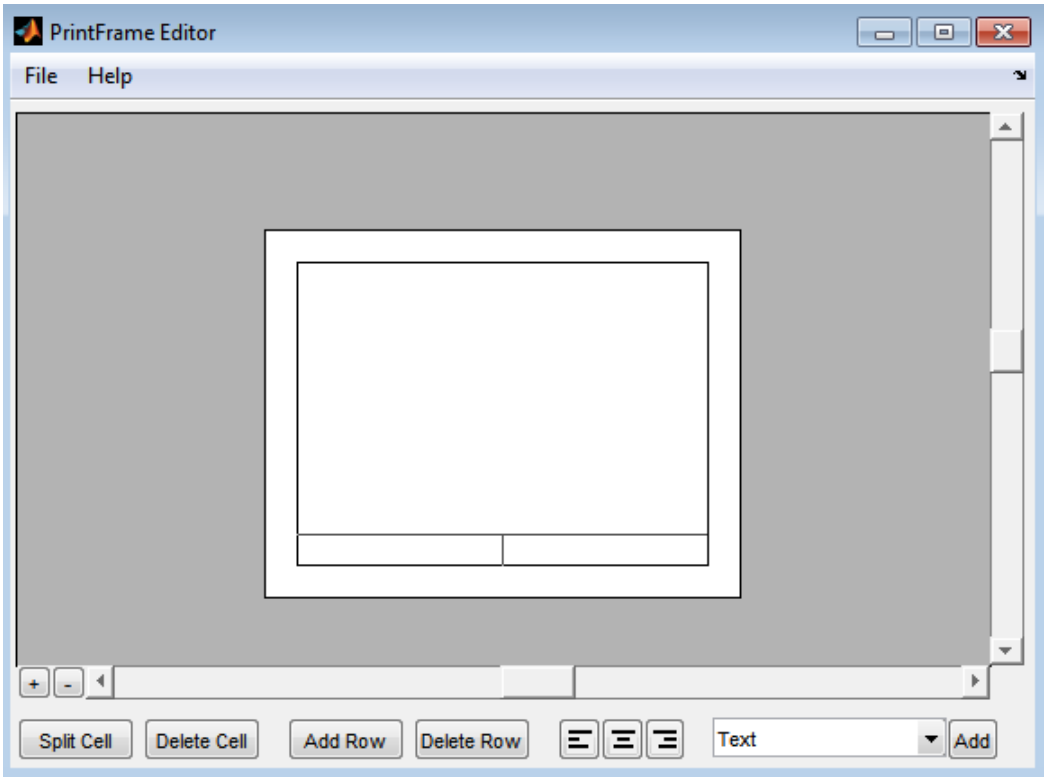
- An additional row at the top of the frame for a title
- A middle row, which includes the block diagram
- A bottom row, in which one cell has the path to the subsystem and another cell has the page number



PrintFrame Editor

Use the **PrintFrame Editor** to create and edit print frames.

To open the **PrintFrame Editor**, at the MATLAB command line, enter the `frameedit` command.



Use the **PrintFrame Editor** to:

- Set up the printed page
- Add or remove rows and cells in the print frame
- Add content to cells, such as text, the date, and page numbers
- Format cell content

To open an existing print frame, use `frameedit` command with the `filename` parameter, where `filename` is an existing print frame (a `.fig` file).

Single Use or Multiple Use Print Frames

You can design a print frame for one particular block diagram, or you can design a more generic print frame for printing multiple block diagrams.

Text and Variable Content

In cells, you can include text (such as the name and address of your organization) and variable content (such as the current date).

See Also

`frameedit`

Related Examples

- “Create a Print Frame” on page 79-6
- “Print Using Print Frames” on page 79-12

Create a Print Frame

1. At the MATLAB prompt, type `frameedit` to open the **PrintFrame Editor**.
2. In the **PrintFrame Editor**, select **File > Page Setup**.

   If necessary, change default page setup for the print frame, which is:

   - Paper type — `usletter`
   - Orientation — **landscape**

     Note The paper orientation you specify does not control the paper orientation used for printing. For example, assume you specify a landscape-oriented print frame in the **PrintFrame Editor**. If you want the printed page to have a landscape orientation, you must specify that using the Print Model dialog box.

   - Margins — .75 inches on all sides
3. Set up the layout of the print frame and add content. See:

   - “Add Rows and Cells to Print Frames” on page 79-7
   - “Add Content to Print Frame Cells” on page 79-9
4. Save the print frame as a `.fig` file. Select **File > Save As**.

See Also

Related Examples

- “Add Rows and Cells to Print Frames” on page 79-7
- “Add Content to Print Frame Cells” on page 79-9

More About

- “Print Frames” on page 79-2

Add Rows and Cells to Print Frames

In this section...

Tip Specify the print frame page setup before you create rows and cells or add content (see “Create a Print Frame” on page 79-6).

Add and Remove Rows

You can add a row above the row that you select.

1. Click in a cell to select a row.

   When you select a row, handles appear on all four corners. If you select only a line, handles appear on two corners.
2. Click **Add Row**.

   The new row appears above the row that you selected.

To remove a row, select the row and click **Delete Row**.

Add and Remove Cells

You can add cells within a row.

1. Select the cell that you want to split.
2. Click **Split Cell**.

   The cell splits into two cells.

To remove a cell, select the cell and click **Delete Cell**.

Resize Rows and Cells

You can change the dimensions of a row or cell by selecting the bordering line.

1. Click the line you want to move.

   A handle appears on both ends of the line.
2. Drag the line to resize the row or cell.

For example, to make a row taller, click on the top line that forms the row. Then drag the line up and the height of the row increases.

To change the overall dimensions of the print frame, see “Create a Print Frame” on page 79-6.

See Also

Related Examples

- “Create a Print Frame” on page 79-6
- “Add Content to Print Frame Cells” on page 79-9
- “Print Using Print Frames” on page 79-12

More About

- “Print Frames” on page 79-2

Add Content to Print Frame Cells

In this section...

Types of Content

You can add text or variables, or both, to a cell.

You must add a `Block Diagram` variable to one of the cells.

For details about the types of content, see:

- “Block Diagram” on page 79-10
- “Variables” on page 79-10
- “Text” on page 79-10

Add Content to Cells

1. Select the cell that you want to add content to.
2. From the list, select the type of content that you want to add.
3. Click **Add**.

   The type of content that you added appears in the cell.

   Tip If you click **Add** and nothing happens, it might be because you did not select a cell first.
4. If you add text, select the edit box and type in the text. For details, see “Text” on page 79-10.

   Tip To make it easier to read and edit the content that you add, you can click the **Zoom in** + button.

Include Multiple Entries in a Cell

1. Select a cell that has a content entry.
2. Add another content type item from the list.

   The new entry is added after the last entry in that cell.

You can also add descriptive text to any of the variable entries without using the **Text** item.

1. Double-click in the cell.

2 Type text in the edit box before or after the entry.

3 To end editing mode, click outside of the cell.

Note You cannot include multiple entries or text in the cell that contains the `Block Diagram` variable. `%<blockdiagram>` must be the only content in that cell.

Block Diagram

Use the `Block Diagram` variable to indicate the cell in which to print the block diagram. Every print frame must include one `Block Diagram` variable. If you do not specify a `Block Diagram` in one of the cells, you cannot save the print frame and cannot print a block diagram with it.

Do not add any other content in a cell that contains a `Block Diagram` variable.

Variables

In addition to the `Block Diagram` variable, you can add other variables, such as the current date, to cells. Simulink supplies variable content at the time of printing.

Variable entries include:

- `Block Diagram` — Add this variable in the cell in which you want the block diagram to print. For details, see “Block Diagram” on page 79-10.
- `Date` — The date that the block diagram and print frame are printed, in `dd-mmm-yyyy` format.
- `Time` — The time that the block diagram and print frame are printed, in `hh:mm` format.
- `Page Number` — The page of the block diagram being printed.
- `Total Pages` — The total number of pages being printed for the block diagram, which depends on the printing options specified.
- `System Name` — The name of the block diagram being printed.
- `Full System Name` — The name of the block diagram being printed, including its position from the root system through the current system, for example, `engine/Throttle & Manifold`.
- `File Name` — The file name of the block diagram, for example, `sldemo_engine.mdl`.
- `Full File Name` — The full path and file name for the block diagram, for example, `\\matlab \toolbox\simulink\simdemos\automotive\sldemo_engine.mdl`.

When you enter a variable, the cell displays the type of content in brackets, `<>`, preceded by a percent sign, `%`. For example, if you add a `Page Number` variable, it appears as `%<page>`.

Note Do not edit the text of a variable entry, because then the variable content does not print. For example, if you accidentally remove the `%` from the `%<page>` entry, the text `<page>` prints, instead of the actual page number.

Text

For `Text` content, type the text that you want to include in that cell (for example, the name of your organization). To type additional text on a new line, press the **Enter** key. When you are finished editing, click outside of the edit box.

You can copy and paste text from another document into a cell. Any formatting of the copied text is lost.

To type special characters (for example, superscripts and subscripts, Greek letters, and mathematical symbols), use embedded TeX sequences. For a list of allowable sequences, see the `text` command `String` property (in Text).

Format Content in Cells

You can align cell contents using the left, center, and right alignment buttons. (Block diagrams are always center aligned.)

You can change font properties, such as size or style (for example, italics or bold). To change font properties, select the cell, then right-click the contents and use the context menu to format the text.

See Also

Related Examples

- “Create a Print Frame” on page 79-6
- “Add Rows and Cells to Print Frames” on page 79-7
- “Print Using Print Frames” on page 79-12

More About

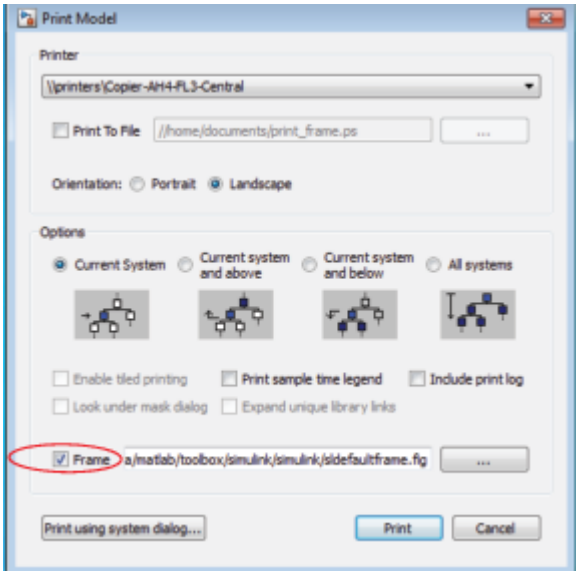
- “Print Frames” on page 79-2

Print Using Print Frames

To print using a print frame, you specify an existing print frame. If you want to build a print frame, see “Create a Print Frame” on page 79-6.

Note If you enable the print frame option, then Simulink does not use tiled printing.

1. In the Simulink Editor or Stateflow Editor, on the **Simulation** tab, click **Print**.
2. In the Print Model dialog box, select the **Frame** check box.



3. Supply the file name for the print frame you want to use. Either type the path and file name directly in the edit box, or click the **...** button and select a print frame file you saved using the **PrintFrame Editor**. The default print frame file name, `sldefaultframe.fig`, appears in the file name edit box until you specify a different file name.
4. Specify other printing options in the Print Model dialog box.

   Note The paper orientation you specify with the **PrintFrame Editor** does not control the paper orientation used for printing. For example, assume you specify a landscape-oriented print frame in the **PrintFrame Editor**. If you want the printed page to have a landscape orientation, you must specify that using the Print Model dialog box.
5. Click **OK**.

   The block diagram prints with the print frame that you specify.

See Also

Related Examples

- “Create a Print Frame” on page 79-6

- “Add Rows and Cells to Print Frames” on page 79-7
- “Add Content to Print Frame Cells” on page 79-9

More About

- “Print Frames” on page 79-2
- “Tiled Printing” on page 1-48

Running Models on Target Hardware

About Run on Target Hardware Feature

Simulink Supported Hardware

As of this release, Simulink supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
Android Devices	Android	R2014a	Current
Apple iOS Devices	Apple	R2015a	Current
Arduino Hardware	Arduino	R2013a	Current
BeagleBoard Hardware	BeagleBoard	R2012a	R2016a
LEGO MINDSTORMS EV3 Hardware	LEGO	R2014a	Current
Raspberry Pi Hardware	Raspberry Pi	R2013a	Current
Parrot Minidrones	Parrot®	R2017a	Current

For a complete list of supported hardware, see Hardware Support.

Block Produces Zeros or Does Nothing in Simulation

If you simulate a model on your host computer without running it on your target hardware:

- Input blocks produce zeros.
- Output blocks do nothing.

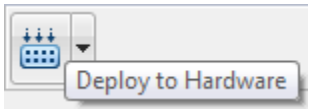
This is the expected behavior.

For example, in a model, on the **Simulation** tab, if you select `Normal` in the **Simulate** section, and then click **Run**, the following happens:

- The sensor block and Digital Input block send zeros to the model.
- The Digital Output block does nothing.

To see the blocks work normally, run your model on target hardware or use External mode.

To run the model on target hardware, on the **Modeling** tab, click **Model Settings**. In the Configuration Parameters dialog box, select **Hardware Implementation**. Then, in the Simulink Editor, on the **Hardware** tab, select **Build Stand-Alone > Build, Deploy & Start**.



To use External mode, in the **Hardware Implementation** pane of the Configuration Parameters dialog box, select your hardware board from the drop-down list. Then expand **Target hardware resources**, and under **Groups** select **External mode**. Then click **Run**.

Running Simulations in Fast Restart

How Fast Restart Improves Iterative Simulations

In the classic Simulink workflow, when you simulate a model, Simulink:

1. Compiles the model
2. Simulates the model
3. Terminates the simulation

While developing a model, you typically simulate a model repeatedly as you iterate the design. For example, you might calibrate input values or block parameters for a particular response. Changing these values or parameters does not always require compiling the model before simulating again. However, in the classic workflow, each simulation compiles the model, even if the changes do not alter the model structurally. Each compile slows down the process and increases overall simulation time.

Fast restart allows you to perform iterative simulations without compiling a model or terminating the simulation each time. Using fast restart, you compile a model only once. You can then tune parameters and root inputs and simulate the model again without spending time on compiling. Fast restart associates multiple simulation phases to a single compile phase to make iterative simulations more efficient.

Use fast restart when your workflow does not require structural changes to the model. Also, fast restart is better suited if the workflow involves any of these factors:

- The model requires multiple simulations in which simulation inputs or parameters change in every iteration.
- The compile time of the model is several seconds or longer.

You can use fast restart in normal and accelerator simulation modes. When you use fast restart in accelerator mode, you reduce simulation time and perform only a single compilation.

Limitations

These are the limitations to simulating in fast restart.

- Fast restart does not support these modes:
  - Rapid Accelerator
  - External
- When a model is in the reinitialized state, you cannot:
  - Make structural changes.
  - Make changes to nontunable parameters such as sample time.
  - Save changes to the model. You must turn off fast restart to save any changes to the model.
- You cannot turn on fast restart in a model if it contains blocks that do not support `ModelOperatingPoint` object. These blocks include:
  - Legacy (pre-R2016a) SimEvents blocks
  - Simscape Multibody First Generation blocks
  - MATLAB function blocks that contain system objects

  - S-functions that do not implement the model operating point `get` and `set` methods but have `Pwork` vectors declared
  - From Multimedia File
  - To Multimedia File
  - From Audio Device
  - To Audio Device
  - Multipath Rician Fading Channel
  - Multipath Rayleigh Fading Channel
  - Derepeat
  - DC Blocker
  - Stack
  - Queue
  - Read Binary File
  - Write Binary File
  - Video Viewer
  - Frame Rate Display
  - Video From Workspace
  - Video To Workspace
- Between simulations, fast restart does not handle changes to design data, such as bus properties.
- Parameter tunability limitations apply. See “Tunability Considerations and Limitations for Other Modeling Goals” on page 37-36.
- The Fixed-Point Tool provides limited support when a model is simulated in fast restart. You must exit fast restart to collect simulation and derived ranges, and propose data types.
- When fast restart is on, you cannot change the variant that a variant subsystem or variant model uses. This is because the inactive subsystems are not compiled in the first simulation.
- When there are multiple model references to the same referenced model, you cannot change the model visibility when the model is in the reinitialized state.
- Fast restart is not compatible with these tools:
  - Simulink Profiler
  - Simulink Debugger
- When simulating a model in fast restart, you cannot run checks using Model Advisor.
- When you enable fast restart, the `sim` command supports only the single output `Simulink.SimulationOutput` form, regardless of the syntax you use in the command.
- When you enable fast restart, you cannot pass non-tunable parameters as arguments to `sim`.

See Also

Related Examples

- “Get Started with Fast Restart” on page 81-5

More About

- “Simulation Phases in Dynamic Systems” on page 3-2
- “Tune and Experiment with Block Parameter Values” on page 37-31
- “Choosing a Simulation Mode” on page 35-10
- “Using Operating Points in Stateflow” (Stateflow)
- “Save and Restore Simulation Operating Point” on page 25-41

Get Started with Fast Restart

In this section...

When you need to simulate a model iteratively to tune parameters, achieve a desired response, or automate testing, use fast restart to avoid compiling again. Fast restart allows you to perform iterative simulations without compiling a model or terminating the simulation each time. A generic workflow using fast restart involves the following steps.

1. Turn on fast restart using the **Fast Restart** button on the Simulink Toolstrip or from the command line.
2. Simulate the model. The first simulation requires the model to compile, initialize and save a `ModelOperatingPoint` object. Once the simulation is complete, it does not terminate. Instead, the model is initialized again in fast restart.
3. Perform any of these actions:

   - Change tunable parameters.
   - Tune root-level inputs.
   - Modify base workspace, model workspace variables and data dictionary entries that are referenced by tunable parameters.
   - Change inputs to From File and From Workspace blocks.
   - Change the **Initial state** parameter for the next simulation.
   - Using the Signal Builder block, change data, rename signals and signal groups, and add new groups.
   - Change the signal logging override values for the model programmatically by using the `set_param` command and the `DataLoggingOverride` parameter. See “Override Signal Logging Settings from MATLAB” on page 72-62.

   Once you have initialized a model in fast restart, you cannot

   - Change the dimension, type, or complexity of a signal or variable.
   - Make changes to a nontunable parameter such as sample time.
   - Make structural changes such as adding or deleting blocks or connections.

   These changes require you to compile the model again. To make changes like these, turn off fast restart, make your changes, and repeat this procedure.
4. Simulate the model. The model uses the new values of parameters and inputs that you provided but does not compile again.
5. Once you have achieved the desired response, turn off fast restart.

Note When you turn off fast restart, Simulink does not store any compile information for the model. The model compiles when you next simulate the model.

Prepare a Model to Use Fast Restart

Before you simulate a model in fast restart, ensure that the model meets these requirements:

- If you have enabled callbacks in the model, make sure they do not attempt to make structural changes when the model is reinitialized. For example, callbacks such as mask initialization commands get called at the beginning of each simulation. Therefore, avoid using mask initialization code that makes structural changes to the model.
- All blocks in the model must support `ModelOperatingPoint` object.
- The simulation mode is Normal or Accelerator mode.

Note When fast restart is on, you cannot save changes to the model after it compiles. Saving changes requires Simulink to discard information about the compiled state. To save any changes to the model, turn off fast restart first.

Enable Fast Restart

Use one of these methods to enable fast restart:

- Click the **Fast restart** button  on the Simulink Editor toolbar.
- At the MATLAB Command prompt, use `set_param` to enable fast restart. Type

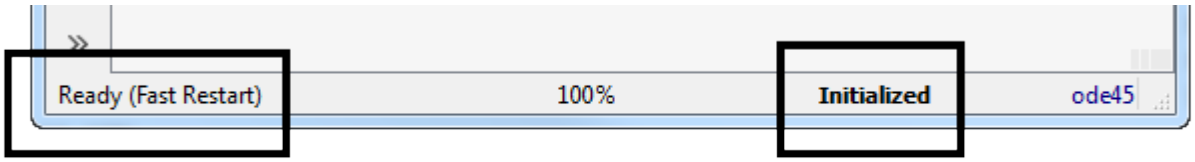
  ```
  set_param(model,'FastRestart','on')
  ```

Simulate a Model using Fast Restart

After you load your model and turn on fast restart, simulate the model.

1. Simulate the model by calling `sim` or clicking the **Run** button  in the Simulink Editor toolstrip. The first simulation in fast restart requires the model to compile and save a `ModelOperatingPoint` object.

   Once the simulation is complete, the status bar shows that the model is initialized in fast restart.



2. Adjust tunable parameters in the model, such as the gain value of a Gain block, or tune root-level input values. You can also make changes to base workspace variables. You cannot adjust nontunable parameters such as sample time, because doing so requires the model to compile once more.
3. Simulate the model again. This time, the model does not compile. When you click the **Play** button or step forward, Simulink updates blocks that have new values as well as blocks that reference workspace variables.
4. When you are satisfied with your results, turn off fast restart by clicking the **Fast restart** button off.

5 To keep your changes, save the model.

Note After a model is initialized in fast restart, Simulink displays a warning if you attempt to make a structural change to the model. To make such changes, you must turn off fast restart.

Stop a Simulation

When you click **Stop** in the middle of a fast restart simulation:

- Simulation does not terminate.
- The model is in the initialized state.
- You can now change tunable parameters in the model
- You can terminate the simulation and exit fast restart by clicking the **Fast restart** button off.

Exit Fast Restart

You can exit fast restart only when the model is in the initialized state. After simulating, click the **Fast restart** button off. To do this programmatically, type:

```
set_param(model,'FastRestart','off')
```

- Simulink terminates simulation.
- Simulink discards any compiled information about the model.
- The model must compile again the next time you simulate.

Fast Restart Methodology

Tuning Parameters Between Simulations

- When a model is initialized in fast restart, in addition to block values and base workspace variables, you can tune parameters in the **Data Import/Export** and **Solver** panes in the Configuration Parameters dialog box (on the **Simulation** tab, under **Prepare**, click **Model Settings**).
- Certain parameters are tunable between simulations only when the model is initialized in fast restart. They include:
  - **Initial Value** parameter of the IC block
  - **Initial Output** parameter of the Merge block
  - **Data** parameter of the From Workspace block
  - **Signal** parameter and signal groups of the Signal Builder block.

Model Methods and Callbacks in Fast Restart

When fast restart is on, Simulink calls model and block methods and callbacks as follows:

1 Call model `InitFcn` callback.

2 Call model `SetupRuntimeResources` method.

   a Call `mdlSetupRuntimeResources` S-function method.

3. Call model `Start` method.
   a. Call `mdlStart` S-function method.
4. Call model `Initialize` method.
   a. Call `mdlInitializeConditions` S-function method.

   Note Use the `ssIsFirstInitCond` flag to guard code that should run only during the initialization phase of any simulation, including the first and subsequent initializations in fast restart.
5. Call model and block `StartFcn` callbacks.

Note Steps 1–5 apply to all simulations in Simulink (with or without fast restart).

6. For the first simulation in fast restart, capture a simulation snapshot. A simulation snapshot contains simulation state (`ModelOperatingPoint`) and information related to logged data and visualization blocks. As part of the snapshot capture, call `ModelOperatingPoint` S-function method.
7. This is a standard execution phase of any simulation, with or without fast restart.
   - Call model `Outputs`.
   - Call model `Update`.
   - Call model `Derivatives`.
   - Repeat these steps in a loop until stop time or a stop is requested.
8. Call model `Terminate` method.
   a. Call `mdlTerminate` S-function method.
9. After simulation ends, call model and block `StopFcn` callbacks. This is a standard phase of any simulation, with or without fast restart.
10. Restore the simulation snapshot taken for fast restart. As part of the restore, call `set` S-function method.
11. Wait in a reinitialized state until one of these actions:
    - To run another simulation (programmatically or using the Simulink Editor) in fast restart, return to step 3.
    - To end Fast Restart mode and uncompile the model:
      a. Call the model method `CleanupRuntimeResources` and the `mdlCleanupRuntimeResources` S-function method.
      b. Do not call `StopFcn` callbacks again at this point.

In some cases. the `Start` and `Terminate` methods are only called once and not for each successive Fast Restart simulation. In these cases, these method calls are combined with calls to `SetupRuntimeResources` and `CleanupRuntimeResources`, respectively. These cases are as follows:

- When an S-function contains custom `ModelOperatingPoint` save and restore methods.
- When an S-function sets the flag `SS_OPTION_CALL_TERMINATE_ON_EXIT`.

- When an S-function is placed inside the accelerated mode of a referenced model.

For more information on model callbacks, see “Callbacks for Customized Model Behavior” on page 4-44.

Operating Point and Initial State Values

You can change initial state values, including `ModelOperatingPoint`, in between fast restart simulations.

When a `ModelOperatingPoint` object for initial state is used in fast restart, every new simulation resets to the start time of the model and not the snapshot time of each `ModelOperatingPoint` object. Thereafter, on the first step forward, Simulink checks to see if a `ModelOperatingPoint` has been specified. If yes, Simulink restores it before computing the next step. Thus, the first simulation step effectively fast forwards to the snapshot time of the specified `ModelOperatingPoint` object.

Analyze Data Using the Simulation Data Inspector

Fast restart supports data logging using the Simulation Data Inspector. Every simulation in fast restart creates an SDI object with the name **<modelname> fast restart run <number>**. The value of `number` increments for each simulation.

Custom Code in the Initialize Function

When you place custom code in the **Configuration Parameters** > **Simulation Target** > **Custom Code** > **Initialize function** pane in the **Model Configuration Parameters** dialog box, this gets called only during the first simulation in fast restart.

See Also

More About

- “How Fast Restart Improves Iterative Simulations” on page 81-2
- “Save and Restore Simulation Operating Point” on page 25-41

Model Component Testing

Component Verification

Component Verification

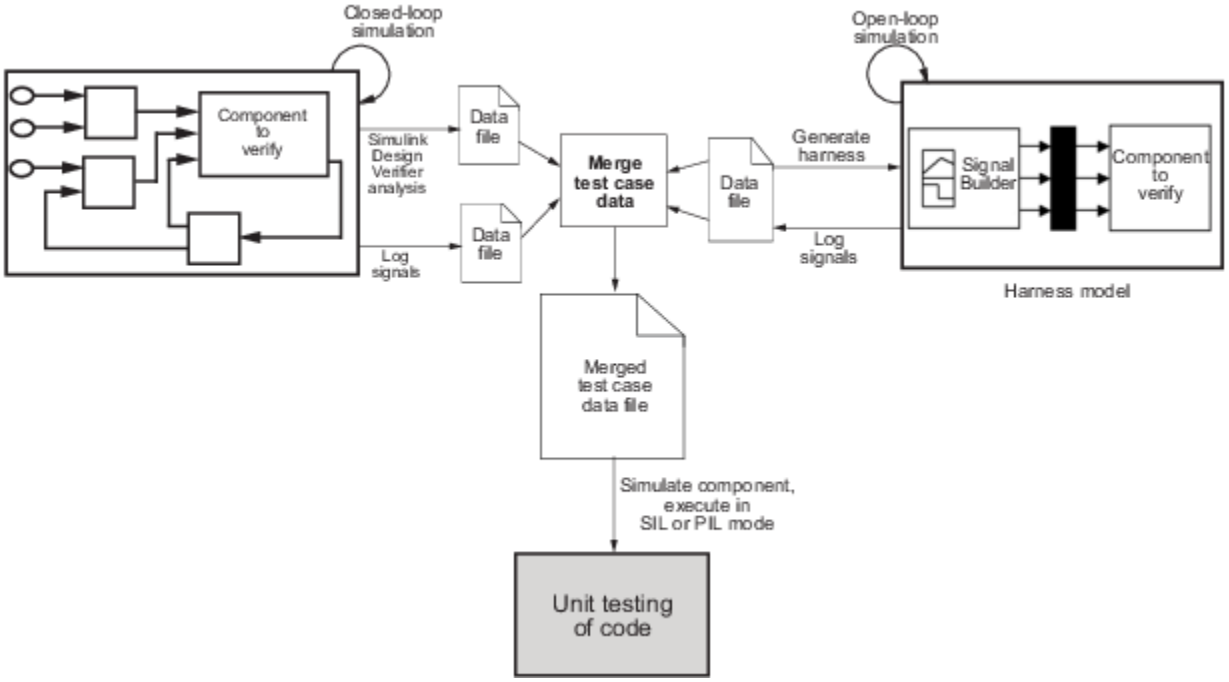
In this section...

You can test a component of your model in isolation, or as part of a larger model. Testing in isolation is useful for debugging the component algorithm and ensuring readiness for component reuse. Testing as part of a larger model considers component behavior in response to particular application inputs and outputs.

This topic is a broad overview of verification activities, including tools in additional products you can use in your verification workflow.

Workflow for Component Verification

This graphic illustrates component verification testing in closed- and open-loop configurations.



1 Choose your approach for component verification:

- For closed-loop simulations, verify a component within the context of its container model by logging the signals to that component and storing them in a data file. If those signals do not constitute a complete test suite, generate a harness model and add or modify the test cases in the Signal Builder.
- For open-loop simulations, verify a component independently of the container model by extracting the component from its container model and creating a harness model for the

extracted component. Add or modify test cases in the Signal Builder and log the signals to the component in the harness model.

2. Prepare component for verification.
3. Create and log test cases. You can also merge the test case data into a single data file.

   The data file contains the test case data for simulating the component. If you cannot achieve the expected results with a certain set of test cases, add new test cases or modify existing test cases in the data file, and merge them into a single data file.

   Continue adding or modifying test cases until you achieve a test suite that satisfies the goals of your analysis.
4. Execute the test cases in software-in-the-loop or processor-in-the-loop mode.
5. After you have a complete test suite, you can:

   - Simulate the model and execute the test cases to:
     - Record coverage using Simulink Coverage.
     - Record output values to make sure that you get the expected results.
   - Invoke the Code Generation Verification (CGV) API to execute the generated code for the model that contains the component in simulation, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode.

   Note To execute a model in different modes of execution, you use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see "Programmatic Code Generation Verification" (Embedded Coder).

Test a Component in Isolation

This workflow illustrates common steps to test reusable components such as:

- Model blocks
- Atomic subsystems
- Stateflow atomic subcharts

1. Depending on the type of component, take one of the following actions:

   - Model blocks — Open the referenced model.
   - Atomic subsystems — Extract the contents of the subsystem into its own Simulink model.
   - Atomic subcharts — Extract the contents of the Stateflow atomic subchart into its own Simulink model.
2. Create a harness model for:

   - The referenced model
   - The extracted model that contains the contents of the atomic subsystem or atomic subchart
3. Add or modify test cases in the Signal Builder in the harness model.
4. Log the input signals from the Signal Builder to the test unit.
5. Repeat steps 3 and 4 until you are satisfied with the test suite.

6. Merge the test case data into a single file.
7. Depending on your goals, take one of the following actions:
   - Execute the test cases to:
     - Record coverage.
     - Record output values and make sure that they equal the expected values.
   - Invoke the Code Generation Verification (CGV) API to execute the test cases in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode on the generated code for the model that contains the component.

If the test cases do not achieve the expected results, repeat steps 3 through 5.

Test a Model Block Included in a Larger Model

Use system analysis to:

- Verify a Model block in the context of the block's container model.
- Analyze a closed-loop controller.

1. Log the input signals to the component by simulating the container model or analyze the model using the Simulink Design Verifier software.
2. If you want to add test cases to your test suite or modify existing test cases, create a harness model with the logged signals.
3. Add or modify test cases in the Signal Builder in the harness model.
4. Log the input signals from the Signal Builder to the test unit.
5. Repeat steps 3 and 4 until you are satisfied with the test suite.
6. Merge the test case data into a single file.
7. Depending on your goals, do one of the following:
   - Execute the test cases to:
     - Record coverage.
     - Record output values and make sure that they equal the expected values.
   - Invoke the Code Generation Verification (CGV) API to execute the test cases in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode on the generated code for the model.

If the test cases do not achieve the expected results, repeat steps 3 through 5.

Run Polyspace Analysis on Generated Code by Using Packaged Options Files

When you start a Polyspace analysis directly from the Simulink toolstrip, the analysis takes the model-specific context, such a design ranges, into consideration. When running a Polyspace analysis without access to Simulink, you must specify the model-specific information by using options files. Instead of authoring these options files, use the options files generated and packaged by the function `polyspacePackNGo`.

Preserving the Simulink model context information when running a Polyspace analysis can be useful in various situations. For instance:

- Distributed workflow: A Simulink user generates code from a model and sends the code to another development environment. In this environment, a Polyspace user, who might not have Simulink, runs a separate analysis of the generated code. By using the packaged options files, the design ranges and other model-specific information is preserved in the Polyspace analysis.
- Analysis options not available in Simulink: Some Polyspace analysis options are available only when the Polyspace analysis is run separately from Simulink. Use packaged options files to run a separate Polyspace analysis while preserving the model-specific information. For instance, analyze concurrent threads in generated code by running a Polyspace analysis in the generated code by using the packaged options files.

You must have Simulink to run the function `polyspacePackNGo`. You do not need Polyspace to generate the options files from a Simulink model. The `polyspacePackNGo` function supports code generated by Embedded Coder and TargetLink®.

Generate and Package Polyspace Options Files

To generate and package Polyspace options file for analyzing code generated from a Simulink model, use `polyspacePackNGo`.

1. In the Simulink Editor, open the Configuration Parameters dialog box and configure the model for code generation.
2. To configure the model for compatibility with Polyspace, select `ert.tlc` as the **System target file**
3. To enable generating a code archive, select the option **Package code and artifacts**. Optionally, provide a name for the options package in the field **Zip file name**. If your code contains a custom code block, select **Use the same custom code settings as Simulation target** in the **Code Generation> Custom Code** pane.

   Alternatively, in the MATLAB Command Window, enter:

   ```
   % Configure the Simulink model mdlName for code generation
   configSet = getActiveConfigSet(mdlName);
   set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
   set_param(configSet, 'PackageName', 'CodeArchive.zip');
   set_param(configSet, 'SystemTargetFile', 'ert.tlc');
   set_param(configSet,'RTWUseSimCustomCode','on');
   ```
4. Generate the code archive.

   - To generate an archive of standalone generated code from the top model, use the function `rtwbuild`.

- To generate code as a model reference, use the function `slbuild`. After generating code as model reference, create the code archive by using the function `packNGo`.
- Alternatively, you can use TargetLink to generate the code. Create the code archive by archiving the generated code into a zip file.

5 To generate and package the Polyspace option files, in the MATLAB Command Window ,use the `polyspacePackNGo` function :

```
zipFile = polyspacePackNGo(mdlName);
```

See “Generate and Package Polyspace Options Files”.

If you use TargetLink to generate code, then use the TargetLink subsystem name as the input argument to `polyspacepacknGo`.

6 Optionally, you can use a `pslinkoptions` object as a second argument to modify the default options for the Polyspace analysis. Create a `pslinkoptions` object containing the additional options and specify the object when creating the archive:

```
psOpt = pslinkoptions(mdlName);
psOpt.InputRangeMode = 'FullRange';
psOpt.ParamRangeMode = 'DesignMinMax';
zipFile = polyspacePackNGo(mdlName,psOpt);
```

See “Package Polyspace Options Files That Have Specific Polyspace Analysis Options”.

7 Use the optional third argument to specify whether to generate and package Polyspace options files for code generated as a model reference. Suppose you generated code as a model reference by using the `slbuild` function. To generate and package Polyspace options for the code, at the MATLAB Command Window, enter:

```
zipFile = polyspacePackNGo(mdlName,psOpt,true);
```

See “Package Polyspace Options Files for Code Generated as a Model Reference”.

The function `polyspacepackNGo` returns the full path to the archive containing the options files. The files are located in the `polyspace` folder within the archived folder hierarchy. The content of the `polyspace` folder depends on the inputs of `polyspacePackNGo` function.

- If you do not specify the optional second and third arguments, then the folder `polyspace` contains these options files in a flat hierarchy:
  - `optionsFile.txt`: This file specifies the source files, the include files, data range specifications, and analysis options required for analyzing the generated code by using Polyspace. If your code contains custom C code, then this file specifies the relative paths of the custom source and header files.
  - *`modelname`*`_drs.xml`: This file specifies the design range specification of the model.
  - `linkdata.xml`: This file links the generated code to the components of the model.
- If you specify `psOpts.ModelbyModelRef = true`, then corresponding options files are generated for all referenced models. These options files are stored in separate folders named `polyspace_<`*`referenced model name`*`>` within the code archive. The folder `polyspace` contains the options files for the top model.

Run Polyspace Analysis by Using the Packaged Options Files

Once the code archive and the Polyspace option files are generated, you can use the archive to run a Polyspace analysis on the generated code in a different development environment without Simulink.

1. Unzip the code archive and locate the `polyspace` folder.
2. On a Windows or Linux command line, run: *`productname`* `-options-file optionsFile.txt -results-dir` *`resultdir`*.

   - *`productname`* corresponds to one of: polyspace-bug-finder, polyspace-code-prover, polyspace-bug-finder-server, or polyspace-code-prover-server.
   - *`resultdir`* corresponds to the location of the Polyspace results. This argument is optional.

   If the file `linkdata.xml` is not there, use the option **Code Generator Support** in Polyspace User Interface to specify which comments in the code act as links to the Simulink model. In the Polyspace User Interface, select **Tools > Preferences** and locate the **Miscellaneous** tab. From the context menu **Code comments that act as code-to-model-link**, select the code generator that you used. If you select **User defined**, then specify the comments that act as a code-to-model link by specifying their prefix in the field **Comments beginning with**. For instance, if you specify the prefix as `//Link_to_model`, then Polyspace interprets comments starting with `//Link_to_model` as links to model.
3. To review the result, upload it to Polyspace Access and view the results in a web browser. Alternatively, view the result by using the user interface of the Polyspace desktop products.

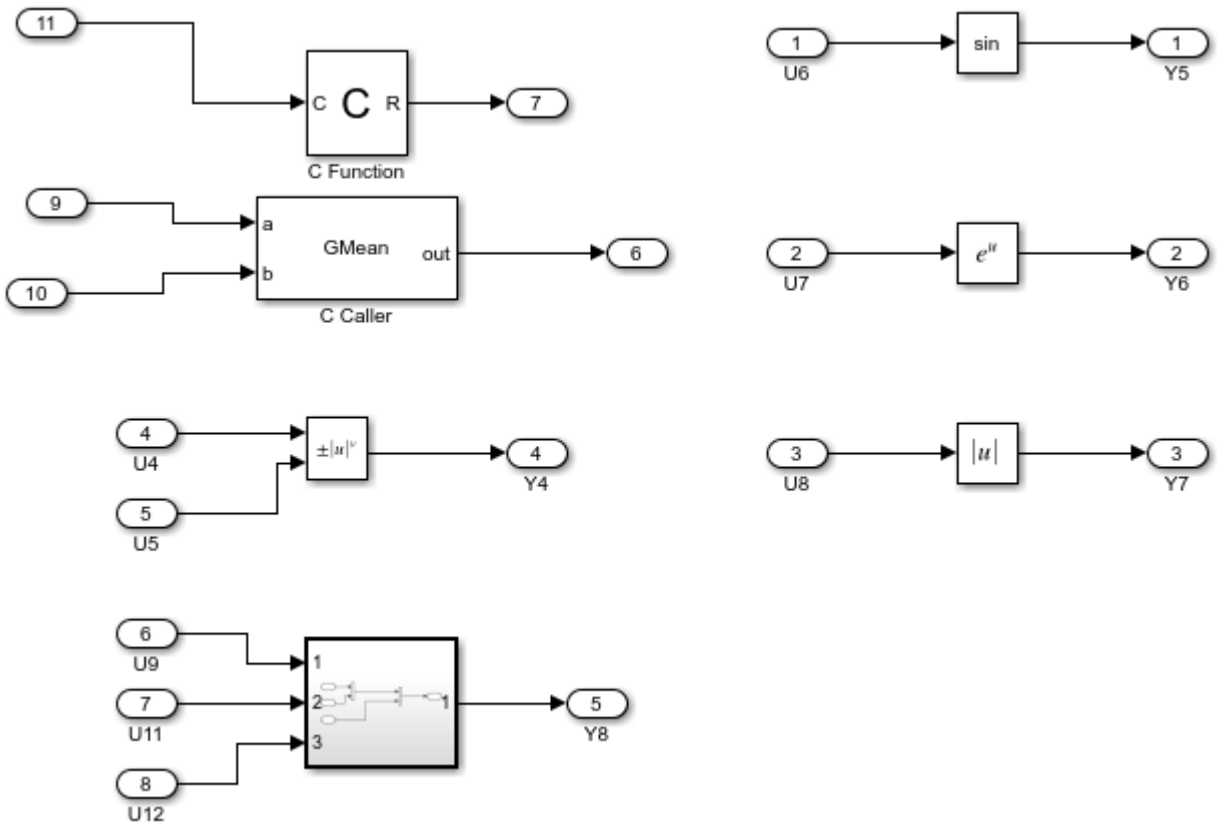
Analyze Code Generated as Standalone Code in a Distributed Workflow

Generate and package Polyspace options files from a Simulink model by using the function `polyspacepackNGo`. Use these options files to run a Polyspace analysis on the generated code that uses model-specific information, such as design range specifications, without requiring Simulink.

Open Model

The model `demo_math_operations` performs various mathematical operations on the model inputs. The model has a C Function block that executes a custom C code. The model also has a C Caller block that calls the C function `GMean`, which is implemented in the source file GMean.c. To open the model for code generation and packaging Polyspace options file, search for the current topic in the MATLAB help browser and click the **Open Model** button. Alternatively, in the MATLAB Command Window, paste and run the following code.

```
open_system('demo_math_operations');
```



Configure Model

To configure the model for generating code and packaging Polyspace options files, specify these configuration parameters:

- To create an archive containing the generated code, set `'PackageGeneratedCodeAndArtifacts'` to `true`.
- Specify a name for the code archive. For instance, set the name to `genCodeArchive.zip`.
- To use the custom code setting specified in **Simulation Target** during code generation, set `'RTWUseSimCustomCode'` to `'on'`.
- To make the model and the generated code compatible with Polyspace, set `ert.tlc` as the system target file. See “Recommended Model Configuration Parameters for Polyspace Analysis” (Polyspace Bug Finder).

In Command Window or Editor, enter these parameter configurations:

```
configSet = getActiveConfigSet('demo_math_operations');
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
set_param(configSet,'RTWUseSimCustomCode','on')
```

Generate Code Archive

Specify a folder for storing the generated code. To start code generation, in the Command Window or in the Editor, enter:

```
codegenFolder = 'demo_math_operations_ert_rtw';
if exist(fullfile(pwd,codegenFolder), 'dir') == 0
    rtwbuild('demo_math_operations')
end
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

Generate and Package Polyspace Options File

To generate Polyspace options files for the generated code, in the Command Window or in the Editor, enter:

```
zipFile = polyspacePackNGo('demo_math_operations');
```

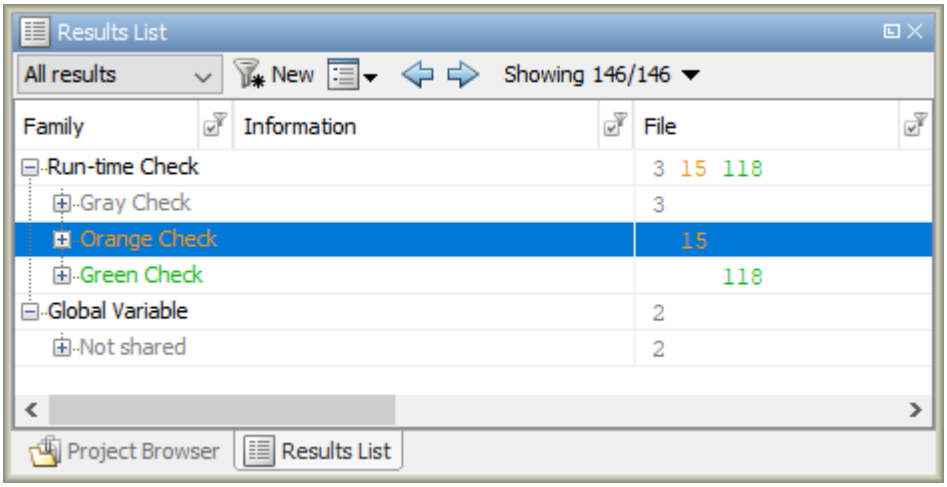
In the archive `genCodeArchive.zip`, find the options files in the folder `<current folder>/polyspace`.

Run Polyspace Analysis by Using the Packaged Options Files

1. Unzip the code archive `genCodeArchive.zip` and locate the `<current folder>/polyspace` folder.
2. Open a command-line terminal and change your working folder to the `polyspace` subfolder of the unzipped folder by using the `cd` command.
3. Start a Polyspace analysis.

   - To run a desktop Polyspace analysis, use either `polyspace-code-prover` or `polyspace-bug-finder`. To run the Polyspace analysis in a server, use either `polyspace-bug-finder-server` or `polyspace-code-prover-server`. Polyspace Bug Finder and Code Prover analyze the code differently. See “Choose Between Polyspace Bug Finder and Polyspace Code Prover” (Polyspace Bug Finder).
   - Specify the file `optionsFile.txt` as the argument to `-options-file`.

   To run a Code prover analysis, run this command: `polyspace-code-prover -options-file optionsFile.txt -results-dir Results`.
4. Follow the progress of the analysis in the log file that is generated in the `Results` folder.
5. To view the results in the desktop user interface, in the command-line interface, enter: `polyspace Results\ps_results.pscp`. The extension of the `ps_results` file changes depending on whether you run a Code Prover analysis or a Bug Finder analysis. The result contains several orange checks.



Alternatively, upload the result to Polyspace Access. See “Upload Results to Polyspace Access” (Polyspace Bug Finder Access)

6 Address the results. For more information, see “Address Results Through Bug Fix or Comments” (Polyspace Bug Finder).

See Also

`packNGo` | `polyspacePackNGo` | `rtwbuild` | `slbuild`

Simulation Testing Using Model Verification Blocks

Construct Simulation Tests by Using the Verification Manager

In this section...

Simulink Model Verification library blocks assess time-domain signals in your model, according to the specifications that you assign to the blocks. Model verification blocks return an assertion when signals fall outside the specified limit or range. During simulation, when the signal crosses the limit, the verification block can:

- Stop the simulation and bring immediate focus to that part of the model.
- Report the failure with a logical signal. If the simulation does not fail, the signal output is `true`. If the simulation fails, the signal output is `false`.

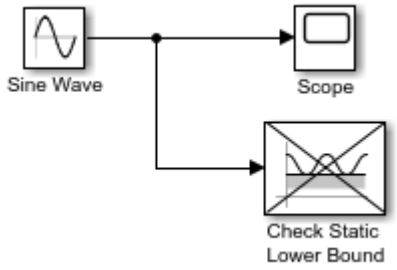
For reference information on individual model verification blocks, see “Model Verification”.

If you use a Signal Builder block to provide test signals for your model, you can enable and disable Model Verification blocks through the Verification Manager graphical interface. To open the Verification Manager, on the Signal Builder dialog box toolbar, select the **Show Verification Settings** icon  .

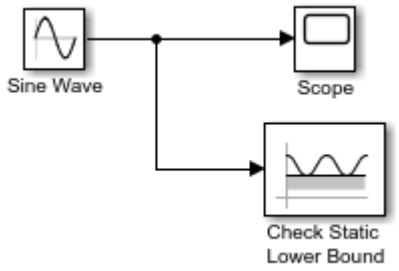
Use Model Verification Block to Check for Out-of-Bounds Signal

This example uses a Check Static Lower Bound block to stop the model simulation when a signal from a Sine Wave block crosses its lower bound limit.

In the model, the Check Static Lower Bound block has a **Lower bound** parameter of `-0.8`. The assertion is disabled, so the block appears crossed out.



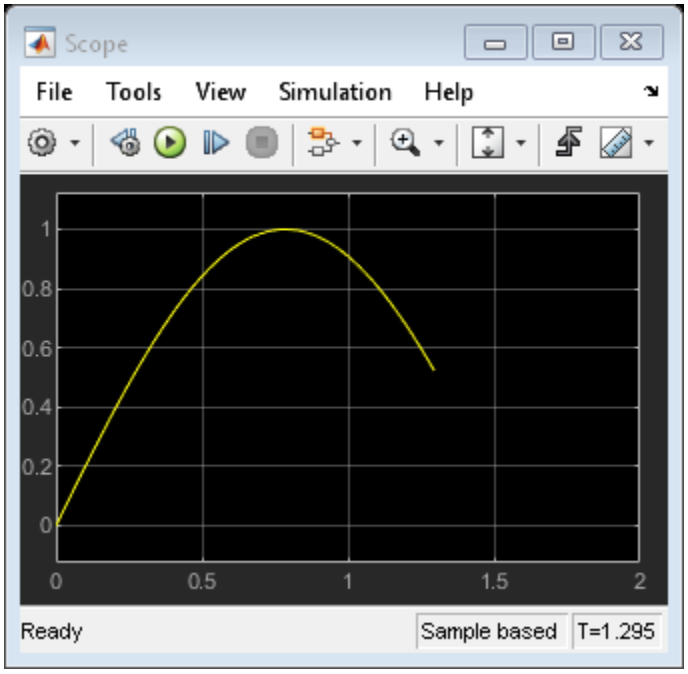
1. Double-click the Check Static Lower Bound block and select the **Enable assertion** check box. This parameter enables the assertion of the verification block. In the model, the block is no longer crossed out.



2. Run the simulation. After 1.29517 seconds, when the signal from the Sine Wave block reaches the lower bound of -0.8, the verification block stops the simulation with this diagnostic message:

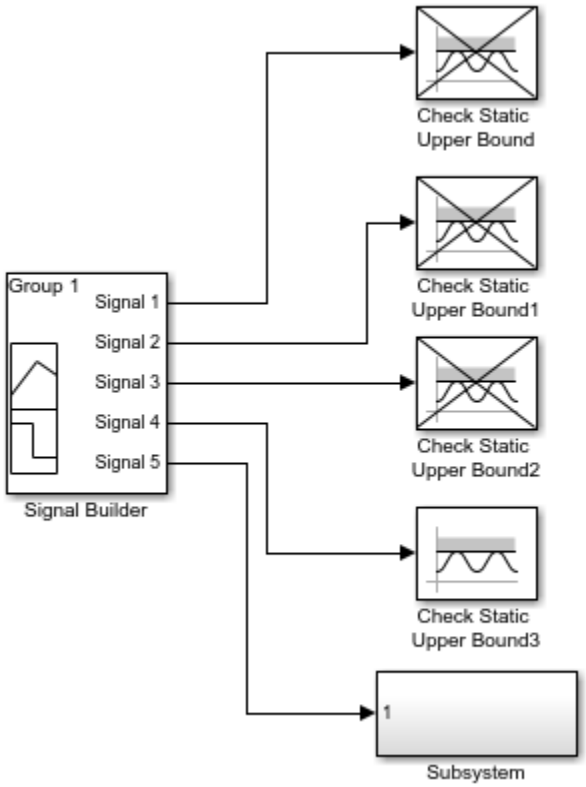
```
An error occurred while running the simulation and the simulation was terminated
Caused by:
Assertion detected in 'ex_model_verif_block_check_static_lower_errwarn/Check Static  Lower Bou
```

3. To verify the signal value, double-click the Scope block.

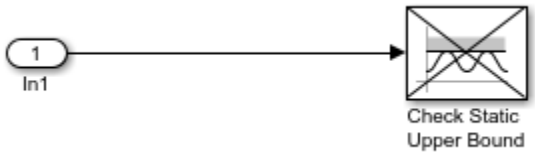


View Model Verification Blocks in Verification Manager

This model contains a Signal Builder block that feeds five test signals to Model Verification blocks. The first four signals connect directly to Check Static Upper Bound blocks.

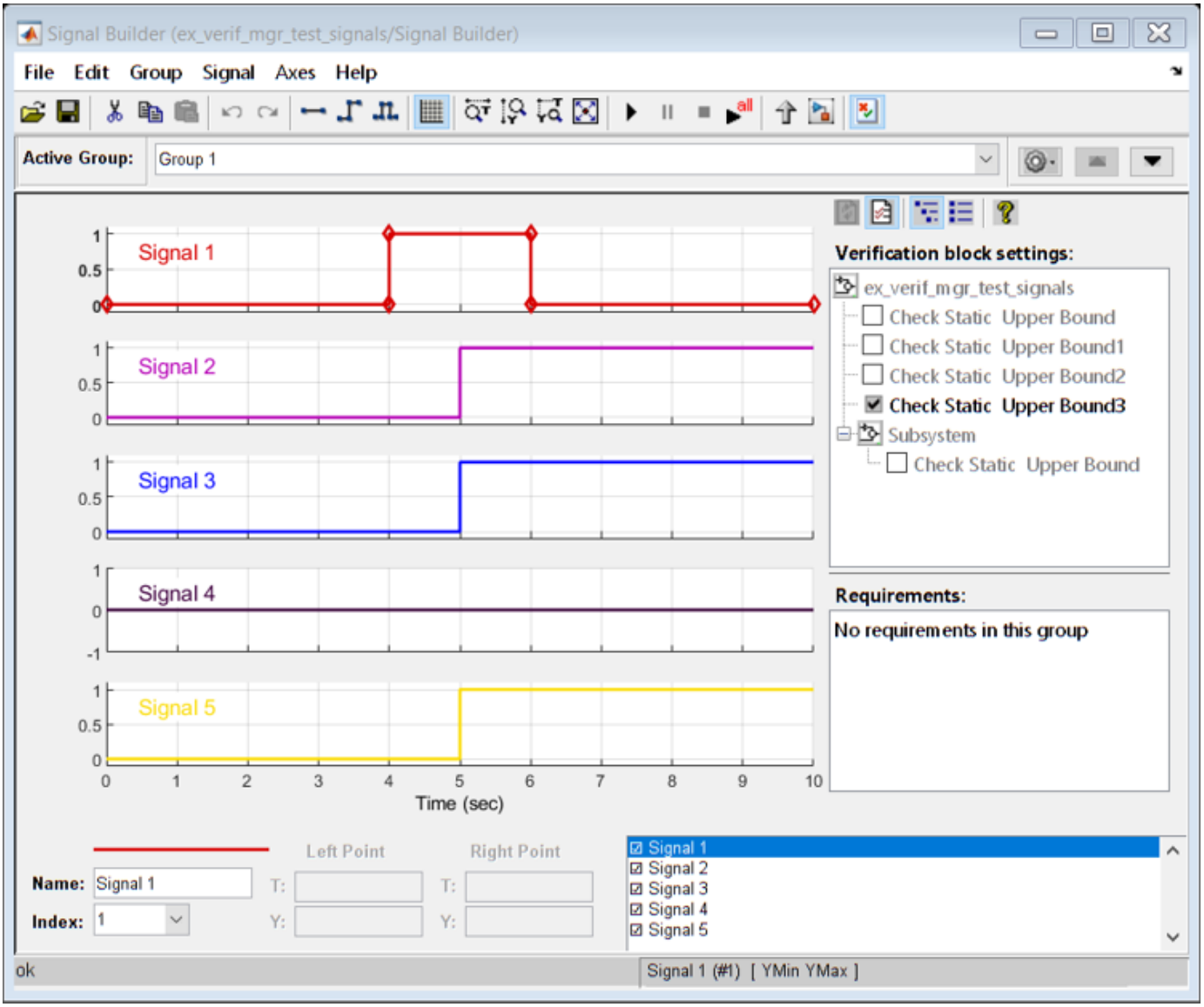


The fifth signal connects to a subsystem that contains a Check Static Upper Bound block.



Only the assertion for the Check Static Upper Bound3 block is enabled. The other Model Verification blocks appear crossed out in the model because their assertions are disabled.

To open the Verification Manager, double-click the Signal Builder block and select the **Show Verification Settings** icon.



Manage Verification Blocks and Requirement Links

The Verification Manager consists of the **Verification block settings** pane and the **Requirements** pane.

The **Verification block settings** pane lists all Model Verification blocks in the model, grouped by subsystem. For example, in the `ex_verif_mgr_test_signals` model, the **Verification block settings** pane displays five Check Static Upper Bound blocks. Four are in the top level of the model, and one is in a subsystem.

- To display all of the Model Verification blocks, click the **Show verification block hierarchy** icon  .

- To display only the blocks that are enabled for the current signal group, click the **List enabled verifications** icon  .

You can select additional options for viewing Model Verification blocks by right-clicking in the **Verification block settings** pane:

- **Display > Tree format** — List the blocks as they appear in the model hierarchy.
- **Display > Overridden blocks only** — List only the blocks that are not enabled for all test groups.
- **Display > Active blocks only** — List only the blocks that are enabled for the current signal group.

The **Requirements** pane lists the requirements document links for the current signal group. If you have Simulink Requirements, you can link requirements documents to test cases and their corresponding Model Verification blocks through this pane.

- To open or close the **Requirements** pane, click the **Requirements display** icon  .
- To link a requirements document to a test case, in the **Requirements** pane, right-click and select **Open Outgoing Links Dialog** from the context menu. In the Outgoing Links dialog box, you can browse and select a requirements document. For more information, see “Link Test Cases to Requirements Documents” (Simulink Requirements).

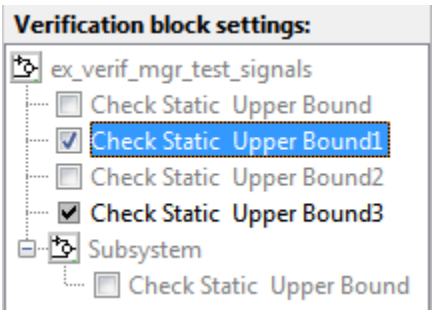
Enable and Disable Individual Model Verification Blocks

In the Verification Manager, each verification block has a status node that indicates whether its assertion is enabled or disabled. The status node also indicates whether the enabled setting applies universally or only to the current active group. This table describes the different types of status nodes and the context menu options that are available when you right-click a node.

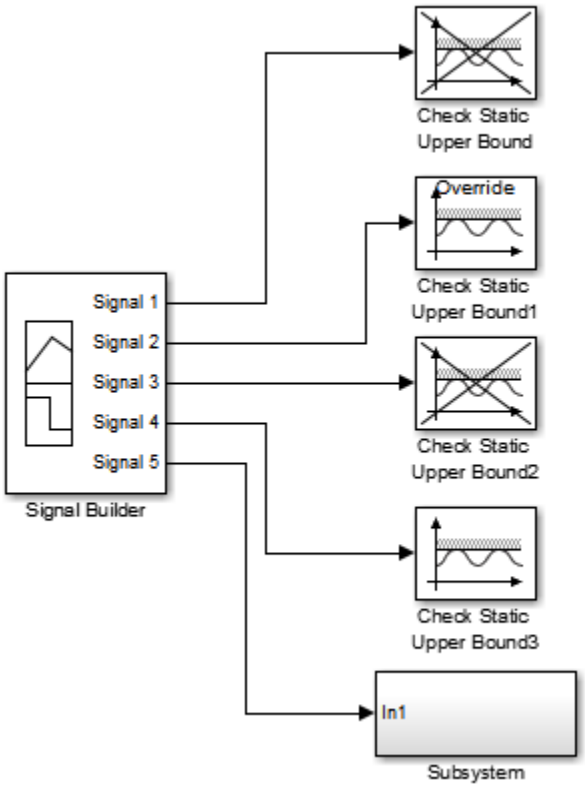
Node	Status	Context Menu Options
☐	Verification block is disabled for the current active group. Click to enable for the current active group.	**Block enable for all groups** — Enable the verification block for all test groups. The node type changes to enabled for all groups ☑.
		Block group enable — Enable the verification block for the current active group. The node type changes to enabled ☑.
☑	Verification block is enabled for the current active group. Click to disable for the current active group.	**Block enable for all groups** — Enable the verification block for all test groups. The node type changes to enabled for all groups ☑.
		Block group disable — Disable the verification block for the current active group. The node type changes to disabled ☐.
☑	Verification block is enabled for all test groups.	**Block enable by group** — Restore the individually enabled/disabled settings to this block for all test groups. Depending on your previous selection, the node type changes to enabled ☑ or disabled ☐

When you use the Verification Manager to enable a model verification block for the current active group, in the model, the block displays an **Override** label. For example, in the

`ex_verif_mgr_test_signals` model, when you select `Group 2` from the **Active Group** list, the Verification Manager shows that the Check Static Upper Bound1 block is enabled.



In the model, the Check Static Upper Bound1 block is not crossed out, but displays an **Override** label.



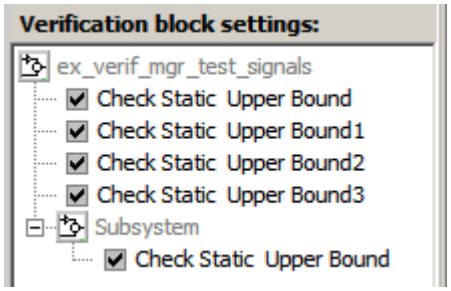
Enable and Disable Model Verification Blocks by Subsystem

If your model contains many verification blocks, it is tedious to enable and disable blocks individually. Using the Verification Manager, you can enable and disable all blocks in a subsystem. In the **Verification block settings** pane, right-click the subsystem node and select from these context menu options:

- **Contents enable for all groups** — Enable all verification blocks in this subsystem for all test groups.

- **Contents enable by group** — Restore the individually enabled/disabled settings to each verification block in this subsystem for all test groups.
- **Contents group enable** — Individually enable all verification blocks in this subsystem for the current active group.
- **Contents group disable** — Individually disable all verification blocks in this subsystem for the current active group.

For example, in the `ex_verif_mgr_test_signals` model, you can enable all of the verification blocks for all test groups by double-clicking the `ex_verif_mgr_test_signals` node and selecting **Contents enable for all groups**. In the Verification Manager, all nodes change to enabled for all groups ☑.



To restore the individually enabled/disabled settings for each verification block in each group, double-click the `ex_verif_mgr_test_signals` node and select **Contents enable by group**.

Linear System Modeling Blocks in Simulink Control Design

If you have Simulink Control Design, you can:

- Monitor time-domain and frequency-domain characteristics.
- Specify bounds on linear system characteristics.
- Check that the bounds are satisfied during simulation.

For reference information on individual blocks, see “Model Verification” (Simulink Control Design).

See Also

Check Static Lower Bound | Check Static Upper Bound | Scope | Signal Builder | Sine Wave

More About

- “Model Verification”
- “Link Test Cases to Requirements Documents” (Simulink Requirements)
- “Model Verification” (Simulink Control Design)