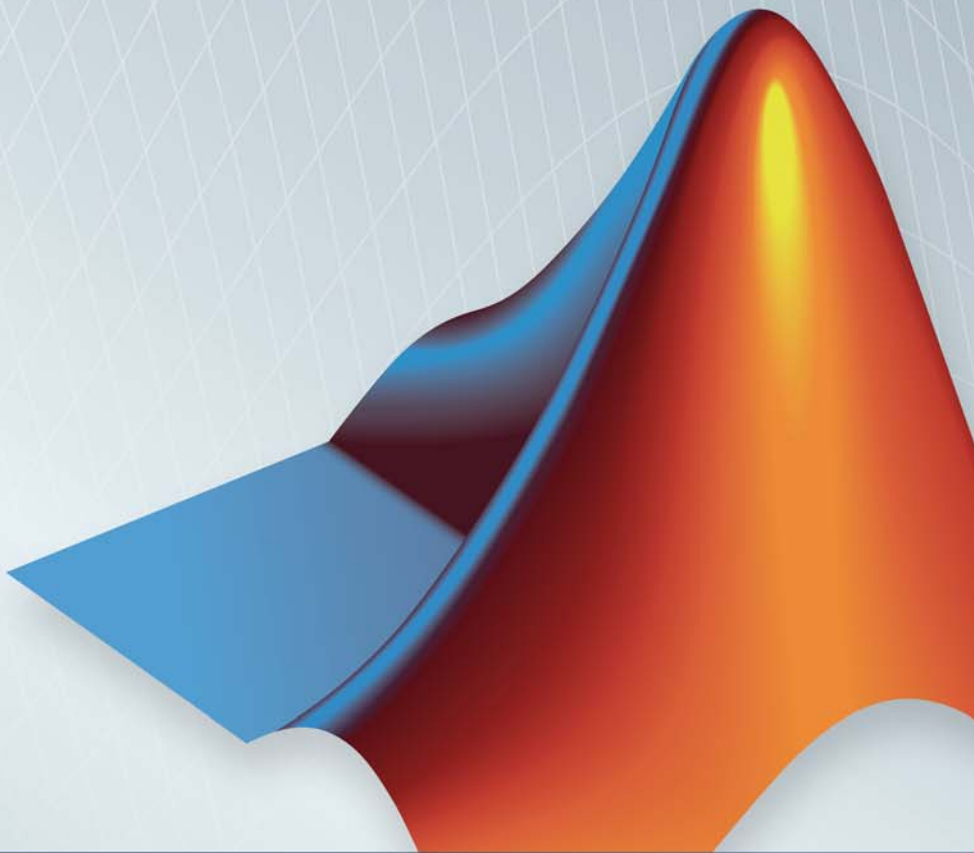


Simulink®

User's Guide

R2013b



MATLAB® & SIMULINK®



How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Simulink® User's Guide

© COPYRIGHT 1990–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)

Introduction to Simulink

Simulink Basics

1

Start the Simulink Software	1-3
Open the MATLAB Software	1-3
Open the Library Browser	1-3
Open the Simulink Editor	1-4
Open a Model	1-5
What Happens When You Open a Model	1-5
Open an Existing Model	1-5
Models with Different Character Encodings	1-6
Avoid Initial Model Open Delay	1-6
Load a Model	1-8
Load Variables When Loading a Model	1-8
Save a Model	1-10
How to Tell If a Model Needs Saving	1-10
Save a Model for the First Time	1-11
Model Names	1-11
Save a Previously Saved Model	1-11
What Happens When You Save a Model?	1-11
Saving Models in the SLX File Format	1-12
Saving Models with Different Character Encodings	1-15
Export a Model to a Previous Simulink Version	1-16
Save from One Earlier Simulink Version to Another	1-17
Simulink Editor	1-19
Editor Layout	1-19
Undoing Commands	1-23
Window Management	1-23
Zoom and Pan Block Diagrams	1-26

Preview Content of Hierarchical Items	1-28
What Is Content Preview?	1-28
Enable Content Preview	1-29
What Content Preview Displays	1-30
Update a Block Diagram	1-32
Updating the Diagram	1-32
Simulation Updates the Diagram	1-32
Update Diagram at Edit Time	1-32
Printing Capabilities	1-34
Print Interactively or Programmatically	1-34
Printing Options	1-34
Canvas Color	1-34
Print Model Reports	1-35
Basic Printing	1-36
Print the vdp Model Using Default Settings	1-36
Print a Subsystem Hierarchy	1-38
Select the Systems to Print	1-41
Print Current System	1-41
Print Subsystems	1-41
Print a Model Referencing Hierarchy	1-42
Specify the Page Layout and Print Job	1-43
Page and Print Job Setup	1-43
Two Interfaces for Page and Print Job Setup	1-43
Tiled Printing	1-44
Print Multiple Pages for Large Models	1-45
Add a Log of Printed Models	1-46
Add a Sample Time Legend	1-47
Print from the MATLAB Command Line	1-48
Printing Commands	1-48

Print Systems with Multiline Names or Names with Spaces	1-48
Set Paper Orientation and Type	1-49
Position and Size a System	1-49
Use Tiled Printing	1-50
Export Models to Third-Party Applications	1-52
Print to a PDF or Postscript File	1-53
Export Models to Image File Formats	1-54
Generate a Model Report	1-55
Model Report Options	1-56
End a Simulink Session	1-58
Keyboard and Mouse Shortcuts for Simulink	1-59
Model Viewing Shortcuts	1-59
Block Editing Shortcuts	1-59
Line Editing Shortcuts	1-61
Signal Label Editing Shortcuts	1-61
Annotation Editing Shortcuts	1-61
Simulink Demos Are Now Called Examples	1-62

Simulation Stepping

2

Simulation Stepping	2-2
How Stepping Through a Simulation Works	2-2
Simulation Stepper Interface	2-5
Simulation Stepper Limitations	2-11
Step Through a Simulation	2-14
Configure Simulation Stepping	2-14
Step Forward and Back	2-14

Set Conditional Breakpoints for Stepping a Simulation	2-17
Conditional Breakpoints Limitations	2-18
Add and Edit Conditional Breakpoints	2-18
Observe Conditional Breakpoint Values	2-19

How Simulink Works

3

How Simulink Works	3-2
Modeling Dynamic Systems	3-3
Block Diagram Semantics	3-3
Creating Models	3-4
Time	3-5
States	3-5
Block Parameters	3-9
Tunable Parameters	3-9
Block Sample Times	3-9
Custom Blocks	3-10
Systems and Subsystems	3-11
Signals	3-16
Block Methods	3-16
Model Methods	3-17
Simulating Dynamic Systems	3-18
Model Compilation	3-18
Link Phase	3-19
Simulation Loop Phase	3-19
Solvers	3-21
Zero-Crossing Detection	3-24
Algebraic Loops	3-39

Modeling Dynamic Systems

Creating a Model

4

Create an Empty Model	4-2
Create a Model Template	4-2
Populate a Model	4-4
Copy Blocks to Your Model	4-4
Browse Block Libraries	4-4
Search Block Libraries	4-5
Copy Blocks to Models	4-5
Select Modeling Objects	4-6
Select an Object	4-6
Select Multiple Objects	4-6
Specify Block Diagram Colors	4-8
Set Block Diagram Colors Interactively	4-8
Platform Differences for Custom Colors	4-9
Choose a Custom Color	4-9
Define a Custom Color	4-10
Specify Colors Programmatically	4-11
Connect Blocks	4-12
Automatically Connect Blocks	4-12
Manually Connect Blocks	4-15
Disconnect Blocks	4-21
Align, Distribute, and Resize Groups of Blocks	4-22
Annotate Diagrams	4-23
Add and Edit Annotations	4-23
Summary of Annotations Properties Dialog Box	4-28
Annotation Callback Functions	4-29
Associate Click Functions with Annotations	4-30
Annotations API	4-33
TeX Formatting Commands in Annotations	4-33
Create Annotations Programmatically	4-35

Create a Subsystem	4-37
Subsystem Advantages	4-37
Two Ways to Create a Subsystem	4-38
Create a Subsystem with a Subsystem Block	4-38
Group Existing Blocks into a Subsystem	4-38
Subsystem Execution	4-40
Navigate Model Hierarchy	4-40
Label Subsystem Ports	4-43
Control Access to Subsystems	4-43
Control Subsystem Behavior with Callbacks	4-44
Interconvert Subsystems and Block Diagrams	4-45
Empty Subsystems and Block Diagrams	4-45
Use Control Flow Logic	4-46
Equivalent C Language Statements	4-46
Conditional Control Flow Logic	4-46
While and For Loops	4-49
Callbacks for Customized Model Behavior	4-56
Model, Block, and Port Callbacks	4-56
What You Can Do with Callbacks	4-56
Avoid run Commands in Callback Code	4-57
See Also	4-57
Model Callbacks	4-58
Create Model Callbacks	4-58
View Model Callbacks	4-59
Model Callback Parameters	4-60
Block Callbacks	4-64
Create Block Callbacks	4-64
Block Callback Parameters	4-64
Port Callbacks	4-72
Callback Tracing	4-73
Model Workspaces	4-74
Model Workspace Differences from MATLAB	
Workspace	4-74
Troubleshooting Memory Issues	4-75

Simulink.ModelWorkspace Data Object Class	4-75
Change Model Workspace Data	4-76
Specify Data Sources	4-79
Symbol Resolution	4-83
Symbols	4-83
Symbol Resolution Process	4-83
Numeric Values with Symbols	4-85
Other Values with Symbols	4-85
Limit Signal Resolution	4-86
Explicit and Implicit Symbol Resolution	4-87
Consult the Model Advisor	4-88
About the Model Advisor	4-88
Start the Model Advisor	4-89
Overview of the Model Advisor Window	4-92
Overview of the Model Advisor Dashboard	4-95
Run Model Advisor Checks	4-96
Run Checks Using Model Advisor Dashboard	4-99
Highlight Model Advisor Analysis Results	4-101
Fix a Warning or Failure	4-103
Revert Changes Using Restore Points	4-107
View and Save Model Advisor Reports	4-109
Run the Model Advisor Programmatically	4-112
Check Support for Libraries	4-112
Checks Triggering an Update Diagram	4-113
Model Advisor Limitations	4-113
Consult the Upgrade Advisor	4-113
Manage Model Versions	4-115
How Simulink Helps You Manage Model Versions	4-115
Model File Change Notification	4-116
Specify the Current User	4-118
Manage Model Properties	4-118
Log Comments History	4-125
Version Information Properties	4-127
Model Discretizer	4-130
What Is the Model Discretizer?	4-130
Requirements	4-130
Discretize a Model with the Model Discretizer	4-131
View the Discretized Model	4-140

Discretize Blocks from the Simulink Model	4-143
Discretize a Model with the sldisemdl Function	4-153

Working with Sample Times

5

What Is Sample Time?	5-2
Specify Sample Time	5-3
Designate Sample Times	5-3
Specify Block-Based Sample Times Interactively	5-5
Specify Port-Based Sample Times Interactively	5-6
Specify Block-Based Sample Times Programmatically ...	5-7
Specify Port-Based Sample Times Programmatically	5-8
Access Sample Time Information Programmatically	5-8
Specify Sample Times for a Custom Block	5-8
Determining Sample Time Units	5-8
Change the Sample Time After Simulation Start Time ...	5-8
View Sample Time Information	5-9
View Sample Time Display	5-9
Sample Time Legend	5-10
Print Sample Time Information	5-14
Types of Sample Time	5-15
Discrete Sample Time	5-15
Continuous Sample Time	5-16
Fixed in Minor Step	5-16
Inherited Sample Time	5-16
Constant Sample Time	5-17
Variable Sample Time	5-18
Triggered Sample Time	5-19
Asynchronous Sample Time	5-19
Block Compiled Sample Time	5-21
Sample Times in Subsystems	5-24

Sample Times in Systems	5-26
Purely Discrete Systems	5-26
Hybrid Systems	5-29
Resolve Rate Transitions	5-32
How Propagation Affects Inherited Sample Times	5-33
Process for Sample Time Propagation	5-33
Simulink Rules for Assigning Sample Times	5-33
Backpropagation in Sample Times	5-35

Referencing a Model

6

Overview of Model Referencing	6-2
About Model Referencing	6-2
Referenced Model Advantages	6-5
Masking Model Blocks	6-6
Models That Use Model Referencing	6-7
Model Referencing Resources	6-7
Create a Model Reference	6-8
Convert a Subsystem to a Referenced Model	6-12
Conversion Process	6-12
Select Subsystems to Convert	6-12
Prepare the Model for Conversion	6-13
Run a Conversion Tool	6-16
Connect the Model Block and Perform Other Post-Conversion Tasks	6-18
Convert a Masked Subsystem	6-19
Referenced Model Simulation Modes	6-21
Simulation Modes for Referenced Models	6-21
Specify the Simulation Mode	6-23
Mixing Simulation Modes	6-23

Using Normal Mode for Multiple Instances of Referenced Models	6-25
Accelerating a Freestanding or Top Model	6-33
View a Model Reference Hierarchy	6-35
Display Version Numbers	6-35
Model Reference Simulation Targets	6-37
Simulation Targets	6-37
Build Simulation Targets	6-38
Simulation Target Output File Control	6-39
Reduce Update Time for Referenced Models	6-42
Simulink Model Referencing Requirements	6-47
About Model Referencing Requirements	6-47
Name Length Requirement	6-47
Configuration Parameter Requirements	6-47
Model Structure Requirements	6-52
Parameterize Model References	6-54
Introduction	6-54
Global Nontunable Parameters	6-55
Global Tunable Parameters	6-55
Using Model Arguments	6-55
Conditional Referenced Models	6-61
Kinds of Conditional Referenced Models	6-61
Working with Conditional Referenced Models	6-62
Create Conditional Models	6-63
Reference Conditional Models	6-65
Simulate Conditional Models	6-66
Generate Code for Conditional Models	6-66
Requirements for Conditional Models	6-67
Protected Model	6-69
Use Protected Model in Simulation	6-71
Protected Model Web View	6-72
Inherit Sample Times	6-73

How Sample-Time Inheritance Works for Model Blocks ..	6-73
Conditions for Inheriting Sample Times	6-73
Determining Sample Time of a Referenced Model	6-74
Blocks That Depend on Absolute Time	6-74
Blocks Whose Outputs Depend on Inherited Sample Time	6-76
Refresh Model Blocks	6-77
S-Functions with Model Referencing	6-78
S-Function Support for Model Referencing	6-78
Sample Times	6-78
S-Functions with Accelerator Mode Referenced Models ...	6-79
Using C S-Functions in Normal Mode Referenced Models	6-80
Protected Models	6-80
Simulink Coder Considerations	6-80
Buses in Referenced Models	6-81
Signal Logging in Referenced Models	6-82
Model Referencing Limitations	6-83
Introduction	6-83
Limitations on All Model Referencing	6-83
Limitations on Normal Mode Referenced Models	6-86
Limitations on Accelerator Mode Referenced Models	6-87
Limitations on Rapid Accelerator Mode Referenced Models	6-90
Limitations on PIL Mode Referenced Models	6-90

Create Conditional Subsystems

7

Conditional Subsystems	7-2
Export-Function Models	7-4
About Export-Function Models	7-4

Requirements for Export-Function Models	7-5
Specifying periodic sample time on function-call root-level Inport blocks	7-6
Execution Order for Function-Call Root-level Inport Blocks	7-7
Workflow for Export-Function Models	7-10
Nested Export-Function Models	7-14
Comparison between Export-Function Models and Models with Asynchronous Function-Call Inputs	7-15
Create an Enabled Subsystem	7-17
What Are Enabled Subsystems?	7-17
Creating an Enabled Subsystem	7-18
Blocks that an Enabled Subsystem Can Contain	7-24
Using Blocks with Constant Sample Times in Enabled Subsystems	7-28
Create a Triggered Subsystem	7-33
What Are Triggered Subsystems?	7-33
Using Model Referencing Instead of a Triggered Subsystem	7-35
Creating a Triggered Subsystem	7-35
Blocks That a Triggered Subsystem Can Contain	7-36
Create an Action Subsystem	7-38
What Are Action Subsystems?	7-38
Set States when an Action Subsystem Executes	7-40
Create a Triggered and Enabled Subsystem	7-42
What Are Triggered and Enabled Subsystems?	7-42
Creating a Triggered and Enabled Subsystem	7-43
A Sample Triggered and Enabled Subsystem	7-44
Creating Alternately Executing Subsystems	7-45
Create a Function-Call Subsystem	7-48
What is a Function-Call Subsystem?	7-48
Creating Function-Call Subsystems	7-48
Sample Time Propagation in Function-Call Subsystems ..	7-48
Conditional Execution Behavior	7-50
What Is Conditional Execution Behavior?	7-50

Propagating Execution Contexts	7-52
Behavior of Switch Blocks	7-53
Displaying Execution Contexts	7-54
Disabling Conditional Execution Behavior	7-55
Displaying Execution Context Bars	7-55
Conditional Subsystem Output Initialization	7-56
Why Initialize Conditional Subsystem Output with Explicit Values?	7-56
Initialization Mode	7-56
When to Use Simplified Initialization	7-57
Simplified Mode Behavior and Requirements	7-58
When to Use Classic Initialization	7-59
Specify or Inherit Conditional Subsystem Initial Values	7-61
Inherit Initial Values from the Input Signal	7-61
Explicitly Specify an Initial Value	7-62
Set Initialization Mode to Simplified or Classic	7-63
Convert from Classic to Simplified Initialization Mode	7-64
Address Classic Mode Issues by Using Simplified Mode	7-65
Classic Mode Issues	7-65
Identity Transformation Can Change Model Behavior ...	7-66
Discrete-Time Integrator or S-Function Block Can Produce Inconsistent Output	7-68
Sorted Order Can Affect Merge Block Output	7-71

Modeling Variant Systems

8

Variant Systems Overview	8-2
What Is A Variant?	8-2
Variant Controls and Active Variants	8-2

How Simulink Represents Variants	8-3
Workflow for Implementing Variants	8-3
Variant Controls	8-4
Create Variant Controls	8-4
Save Variant Controls	8-5
Reuse Simulink Variant Objects	8-5
Enumerated Types as Variant Control	8-6
Set Up Model Variants	8-8
Configure the Model Variants Block	8-9
Disable and Enable Model Variants	8-12
Parameterize Model Variants	8-13
Additional Examples	8-13
Set Up Variant Subsystems	8-14
Configure the Variant Subsystem Block	8-17
Disable or Enable Subsystem Variants	8-17
Additional Examples	8-18
Convert Subsystem Blocks to Variant Subsystem Blocks	8-19
Select the Active Variant for Simulation	8-20
Set the Active Variant	8-20
Override Variants	8-20
Ignore Variant Choices	8-21
Open the Active Variant	8-21
Variant Management	8-22
Manage Variants	8-22
Variant Manager	8-22
Considerations in Model Hierarchy Validation	8-24
Manage Variants Using Variant Manager	8-25
Open Variant Manager	8-26
Add New Variant Configurations	8-28
Validate Variant Configurations	8-30
Set Default Active Configuration	8-31
Define Constraints	8-31
Save Variant Configuration Data	8-32

Model Explorer Overview	9-2
What You Can Do Using the Model Explorer	9-2
Opening the Model Explorer	9-2
Model Explorer Components	9-3
The Main Toolbar	9-4
Adding Objects	9-5
Customizing the Model Explorer Interface	9-5
Basic Steps for Using the Model Explorer	9-6
Focusing on Specific Elements of a Model or Chart	9-7
Model Explorer: Model Hierarchy Pane	9-9
What You Can Do with the Model Hierarchy Pane	9-9
Simulink Root	9-10
Base Workspace	9-10
Configuration Preferences	9-11
Model Nodes	9-11
Displaying Partial or Whole Model Hierarchy Contents ..	9-12
Displaying Linked Library Subsystems	9-13
Displaying Masked Subsystems	9-14
Linked Library and Masked Subsystems	9-14
Displaying Node Contents	9-14
Navigating to the Block Diagram	9-15
Working with Configuration Sets	9-15
Expanding Model References	9-15
Cutting, Copying, and Pasting Objects	9-17
Model Explorer: Contents Pane	9-19
Contents Pane Tabs	9-19
Data Displayed in the Contents Pane	9-22
Link to the Currently Selected Node	9-22
Horizontal Scrolling in the Object Property Table	9-23
Working with the Contents Pane	9-24
Editing Object Properties	9-25
Control Model Explorer Contents Using Views	9-26
Using Views	9-26
Customizing Views	9-29
Managing Views	9-30

Organize Data Display in Model Explorer	9-36
Layout Options	9-36
Sorting Column Contents	9-36
Grouping by a Property	9-37
Changing the Order of Property Columns	9-41
Adding Property Columns	9-42
Hiding or Removing Property Columns	9-43
Marking Nonexistent Properties	9-45
Filter Objects in the Model Explorer	9-46
Controlling the Set of Objects to Display	9-46
Using the Row Filter Option	9-46
Filtering Contents	9-48
Workspace Variables in Model Explorer	9-52
Finding Variables That Are Used by a Model or Block	9-52
Finding Blocks That Use a Specific Variable	9-55
Finding Unused Workspace Variables	9-56
Editing Workspace Variables	9-58
Exporting Workspace Variables	9-60
Importing Workspace Variables	9-62
Search Using Model Explorer	9-63
Searching in the Model Explorer	9-63
The Search Bar	9-64
Show and Hide the Search Bar	9-64
Search Bar Controls	9-64
Search Options	9-66
Run a Search	9-68
Refine a Search	9-69
Model Explorer: Property Dialog Pane	9-70
What You Can Do with the Dialog Pane	9-70
Showing and Hiding the Dialog Pane	9-70
Editing Properties in the Dialog Pane	9-70
Finder	9-73
Find Model Objects	9-73
Filter Options	9-75
Search Criteria	9-76

Model Browser	9-80
About the Model Browser	9-80
Navigating with the Mouse	9-82
Navigating with the Keyboard	9-82
Showing Library Links	9-82
Showing Masked Subsystems	9-82
Model Dependency Viewer	9-83
About Model Dependency Views	9-83
Opening the Model Dependency Viewer	9-88
Manipulating a Dependency View	9-89
Browsing Dependencies	9-95
Saving a Dependency View	9-95
Printing a Dependency View	9-95
View Linked Requirements in Models and Blocks	9-96
Overview of Requirements Features in Simulink	9-96
Highlighting Requirements in a Model	9-97
Viewing Information About a Requirements Link	9-99
Navigating to Requirements from a Model	9-100
Filtering Requirements in a Model	9-102

Managing Model Configurations

10

About Model Configurations	10-2
Multiple Configuration Sets in a Model	10-3
Share a Configuration for Multiple Models	10-4
Share a Configuration Across Referenced Models	10-6
Manage a Configuration Set	10-12
Create a Configuration Set in a Model	10-12
Create a Configuration Set in the Base Workspace	10-12
Open a Configuration Set in the Configuration Parameters Dialog Box	10-13

Activate a Configuration Set	10-13
Set Values in a Configuration Set	10-14
Copy, Delete, and Move a Configuration Set	10-14
Save a Configuration Set	10-15
Load a Saved Configuration Set	10-16
Copy Configuration Set Components	10-16
Manage a Configuration Reference	10-18
Create and Attach a Configuration Reference	10-18
Resolve a Configuration Reference	10-19
Activate a Configuration Reference	10-21
Manage Configuration Reference Across Referenced Models	10-22
Change Parameter Values in a Referenced Configuration Set	10-23
Save a Referenced Configuration Set	10-24
Load a Saved Referenced Configuration Set	10-24
Why is the Build Button Not Available for a Configuration Reference?	10-25
About Configuration Sets	10-26
What Is a Configuration Set?	10-26
What Is a Freestanding Configuration Set?	10-27
Model Configuration Preferences	10-28
About Configuration References	10-29
What Is a Configuration Reference?	10-29
Why Use Configuration References?	10-29
Unresolved Configuration References	10-30
Configuration Reference Limitations	10-31
Configuration References for Models with Older Simulation Target Settings	10-31
Model Configuration Command Line Interface	10-33
Overview	10-33
Load and Activate a Configuration Set at the Command Line	10-34
Save a Configuration Set at the Command Line	10-35
Create a Freestanding Configuration Set at the Command Line	10-36
Create and Attach a Configuration Reference at the Command Line	10-37

Attach a Configuration Reference to Multiple Models at the Command Line	10-38
Get Values from a Referenced Configuration Set	10-39
Change Values in a Referenced Configuration Set	10-39
Obtain a Configuration Reference Handle	10-40
Use refresh When Replacing a Referenced Configuration Set	10-41

Configuring Models for Targets with Multicore Processors

11

Introduction to Concurrent Execution	11-2
About Configuring Models for Concurrent Execution	11-2
Supported Targets	11-3
Helpful Terms	11-3
Software Requirements	11-4
 Design Considerations	 11-5
Modeling for Concurrency	11-5
Simulation Limitations	11-9
 Modeling Process for Concurrent Execution	 11-11
 Configure Your Model	 11-12
Creating a Concurrent Execution Configuration Set	11-12
Creating and Propagating a Configuration Reference	11-13
 Analyze Baseline	 11-18
 Customize Concurrent Execution Settings	 11-21
Configuring Data Transfer Communications	11-21
Configuring Periodic Tasks	11-23
Configuring Aperiodic Triggers and Tasks	11-24
Map Blocks to Tasks Incrementally	11-26
 Interpret Simulation Results	 11-29
Introduction	11-29

Model with System Tasks	11-30
Sample Configured Model with Multiple Target Tasks ...	11-31
How Simulink Determines Data Transfer Requirements ..	11-34
Build and Download to a Multicore Target	11-35
Generating Code	11-35
Configuring Embedded Coder for Native Threads	
Example	11-45
Build and Download	11-46
Profile and Evaluate	11-47
Generate Profile Report	11-48
Concurrent Execution Example Model	11-52
Modeling Concurrent Execution on Multicore Targets	11-52
Command-Line Interface for Concurrent Execution ..	11-61

Modeling Best Practices

12

General Considerations when Building Simulink	
Models	12-2
Avoiding Invalid Loops	12-2
Shadowed Files	12-4
Model Building Tips	12-6
Model a Continuous System	12-8
Best-Form Mathematical Models	12-11
Series RLC Example	12-11
Solving Series RLC Using Resistor Voltage	12-12
Solving Series RLC Using Inductor Voltage	12-13
Model a Simple Equation	12-15
Componentization Guidelines	12-17
Componentization	12-17

Componentization Techniques	12-17
General Componentization Guidelines	12-18
Summary of Componentization Techniques	12-19
Subsystems Summary	12-21
Libraries Summary	12-25
Model Referencing Summary	12-29
Modeling Complex Logic	12-36
Modeling Physical Systems	12-37
Modeling Signal Processing Systems	12-38

Managing Projects

13

Organize Large Modeling Projects	13-4
What Are Simulink Projects?	13-5
Try Simulink Project Tools with the Airframe	
Project	13-7
Explore the Airframe Project	13-7
Set Up Project Files and Open Simulink Project	13-8
View, Search, and Sort Project Files	13-8
Understand Project Startup and Shutdown Tasks	13-10
Create a Startup Shortcut	13-11
Open and Run Frequently Used Files	13-12
Review Changes in Modified Files	13-12
Run Project Integrity Checks	13-14
Run Dependency Analysis	13-14
Commit Modified Files	13-18
View Source Control and Project Information	13-19
Create a New Project to Manage Existing Files	13-21
Add Files to the Project	13-25

Create a New Project from an Archived Project	13-27
Create a New Project Using Templates	13-28
Open Recent Projects	13-32
Change the Project Name, Root, Description, and Startup Folder	13-33
What Can You Do With Project Shortcuts?	13-35
Automate Startup Tasks with Shortcuts	13-36
Set Project Path at Startup and Reset at Shutdown ...	13-39
Automate Shutdown Tasks with Shortcuts	13-41
Create Shortcuts to Frequent Tasks	13-43
Create Shortcuts	13-43
Group Shortcuts	13-44
Annotate Shortcuts to Use Meaningful Names	13-45
Use Shortcuts to Find and Run Frequent Tasks	13-47
Using Templates to Create Standard Project Settings	13-49
Create a Template from the Current Project	13-50
Create a Template from a Project Under Version Control	13-51
Edit a Template	13-52
View and Validate Templates	13-53
Import Templates	13-55

Explore the Example Templates	13-56
Group and Sort File Views	13-57
Search and Filter File Views	13-59
Work with Project Files	13-60
Move Project Files	13-64
Back Out Changes	13-65
Add Labels to Files	13-66
Create Labels	13-67
View and Edit Label Data	13-68
Create a Batch Function	13-70
Create Shortcuts to Batch Job Functions	13-71
Run a Simulink Project Batch Job	13-72
Upgrade Model Files to SLX and Preserve Revision	
History	13-74
Project Tools for Migrating Model Files to SLX	13-74
Upgrade the Model and Commit the Changes	13-74
Verify Changes After Upgrade to SLX	13-77
Archive Projects in Zip Files	13-80
Automate Project Management Tasks	13-81
Manipulate a Simulink Project at the Command Line	13-81
Get Simulink Project at the Command Line	13-81
Find Project Commands	13-82
Examine Project Files	13-82

Label a Project File	13-84
Attach Data to a Label	13-84
Create New Category of Project Labels	13-85
Define a New Label	13-86
Attach New Labels and Label Data to a File	13-86
Query Shortcuts	13-87
Close Project	13-89
More Project API Examples	13-89
About Source Control with Projects	13-90
Register Model Files with Source Control Tools	13-91
Add a Project to Source Control	13-92
Add Files to a New Repository	13-98
Disable Source Control	13-99
Change Source Control	13-100
Set Up SVN Source Control	13-101
Set Up SVN Integration Provided with Simulink Project ..	13-101
Set Up SVN Integration for SVN Version Already Installed	13-102
Set Up SVN Integration for SVN Version Not Yet Provided with Simulink Project	13-103
Register Model Files with Subversion	13-104
Share a Subversion Repository	13-107
Write a Source Control Adapter with the SDK	13-109
Retrieve a Working Copy of a Project from Source Control	13-110
Tag and Retrieve Versions of Project Files	13-115
Refresh Status of Project Files	13-117

Check for Modifications	13-121
Update Revisions of Project Files	13-122
Check Out Files	13-125
View Modified Files	13-128
Project Definition Files	13-129
Review Changes	13-130
Precommit Actions	13-132
Commit Modified Files to Source Control	13-134
Revert Local Changes and Release Locks	13-136
Resolve Conflicts	13-137
Work with Derived Files in Projects	13-139
What Is Dependency Analysis?	13-140
Project Dependency Analysis	13-140
Model Dependency Analysis	13-140
Choose Files and Run Dependency Analysis	13-141
Check Dependencies Results and Resolve Problems ..	13-144
Perform Impact Analysis	13-148
About Impact Analysis	13-148
Find Impact of Selected Files	13-149
Explore Impact Graph	13-152
Export Impact Results	13-155
Save and Reload Dependency Analysis Results	13-157

Analyze Model Dependencies	13-159
What Are Model Dependencies?	13-159
Generate Manifests	13-160
Command-Line Dependency Analysis	13-167
Edit Manifests	13-169
Compare Manifests	13-172
Export Files in a Manifest	13-173
Scope of Dependency Analysis	13-175
Best Practices for Dependency Analysis	13-178
Use the Model Manifest Report	13-179

Simulating Dynamic Systems

Running Simulations

14

Simulation Basics	14-2
Control Execution of a Simulation	14-3
Start a Simulation	14-3
Pause or Stop a Simulation	14-5
Use Blocks to Stop or Pause a Simulation	14-5
Specify Simulation Start and Stop Time	14-8
Choose a Solver	14-9
What Is a Solver?	14-9
Choosing a Solver Type	14-10
Choosing a Fixed-Step Solver	14-13
Choosing a Variable-Step Solver	14-17
Choosing a Jacobian Method for an Implicit Solver	14-24
Interact with a Running Simulation	14-31
Save and Restore Simulation State as SimState	14-32
Overview of the SimState	14-32
Save the SimState	14-33
Restore the SimState	14-35

Change the States of a Block within the SimState	14-37
SimState Interface Checksum Diagnostic	14-37
Limitations of the SimState	14-38
Using SimState within S-Functions	14-38
Diagnose Simulation Errors	14-39
Response to Run-Time Errors	14-39
Simulation Diagnostics Viewer	14-39
Create Custom Simulation Error Messages	14-41

Running a Simulation Programmatically

15

About Programmatic Simulation	15-2
Run Simulation Using the sim Command	15-3
Single-Output Syntax for the sim Command	15-3
Examples of Implementing the sim Command	15-4
Calling sim from Within parfor	15-5
Backwards Compatible Syntax	15-5
Control Simulation Using the set_param Command ...	15-8
Run Simulation from an S-Function	15-9
Control Simulation Using Block Callbacks	15-9
Run Parallel Simulations	15-11
Overview of Calling sim from within parfor	15-11
Simulink and Parallel Computing Toolbox Software	15-15
Simulink and MATLAB Distributed Computing Server Software	15-16
sim in parfor with Normal Mode	15-17
sim in parfor with Normal Mode and MATLAB Distributed Computing Server Software	15-18
sim in parfor with Rapid Accelerator Mode	15-20
Workspace Access Issues	15-21
Resolving Workspace Access Issues	15-21
Data Concurrency Issues	15-23
Resolving Data Concurrency Issues	15-23

Error Handling in Simulink Using MSLException	15-26
Error Reporting in a Simulink Application	15-26
The MSLException Class	15-26
Methods of the MSLException Class	15-26
Capturing Information about the Error	15-26

Visualizing and Comparing Simulation Results

16

View Simulation Results	16-2
What Are Scope Blocks, Signal Viewers, Test Points and Data Logging?	16-2
Scope Block and Scope Signal Viewer Differences	16-3
Why Use Signal Generators and Signal Viewers Instead of Source Blocks and Scope Blocks?	16-4
Signal Viewer Tasks	16-6
About Signal Viewers	16-6
Attaching a Signal Viewer	16-6
Adding Signals to a Signal Viewer	16-8
Displaying a Signal Viewer	16-9
Saving Data to MATLAB Workspace	16-10
Plotting the Viewer Data	16-10
Scope Signal Viewer Tasks	16-11
Adding a Legend to a Scope Signal Viewer	16-11
Creating a Multiple Axes Scope Signal Viewer	16-11
Scope Signal Viewer Characteristics	16-14
Scope Signal Viewer Toolbar	16-14
Scope Signal Viewer Context Menu	16-15
Scope Signal Viewer Parameters Dialog Box	16-16
Line Styles with Scope Signal Viewer	16-19
Parameter Settings and Performance with Scope Signal Viewer	16-20
Signal Generator Tasks	16-21
Attaching a Signal Generator.	16-21
Removing a Generator	16-22

Signal and Scope Manager	16-23
About the Signal & Scope Manager	16-23
Opening the Signal and Scope Manager	16-24
Changing Generator or Viewer Parameters	16-25
Adding Signals to a Viewer	16-25
Removing a Generator or Viewer from a Simulink Model ..	16-25
Viewing Test Point Data	16-26
Signal Selector	16-27
About the Signal Selector	16-27
Select signals for object	16-28
Model Hierarchy	16-28
Inputs/Signals List	16-28
Control Time Scope Programmatically	16-31
Use Simulink.scopes.TimeScopeConfiguration	16-31
Time Scope Configuration Parameters	16-32

Inspecting and Comparing Logged Signal Data

17

Inspect Signal Data with Simulation Data Inspector ..	17-2
Requirements for Recording Data	17-4
Record Simulation Data	17-5
Configure Model for Recording Data	17-5
Configure Simulation Data Inspector	17-5
Simulate Model and Record a Run	17-6
Import Logged Signal Data	17-7
Import Signal Data from the Base Workspace	17-7
Import Signal Data from a MAT-File	17-8
Load Previously Recorded Data from a MAT-file	17-10
Inspect Signal Data	17-11

Overview	17-11
View Signal Data	17-11
Create a View of Multiple Plots	17-13
Explore Signal Data in a Multiple Plot View	17-16
Replace a Run in a View	17-17
Copy a View in the Inspect Signals Tab	17-20
Delete a View in the Inspect Signals Tab	17-21
Export a View in the Inspect Signals Tab	17-21
Import a View in the Inspect Signals Tab	17-22
Compare Signal Data	17-23
Comparison of One Signal From Multiple Simulations	17-25
Compare All Logged Signal Data From Multiple Simulations	17-28
Create Simulation Data Inspector Report	17-32
Export Results in the Simulation Data Inspector Tool	17-34
Save Data to a MAT-File	17-34
How the Simulation Data Inspector Tool Aligns Signals	17-35
Inspect Signals: Aligning Signals for Replacing a Run ...	17-35
Compare Runs: Aligning Signals for Comparing Signal Data	17-36
How the Simulation Data Inspector Tool Compares Time Series Data	17-37
How Tolerances Are Applied	17-37
Run Management Configuration	17-38
Append New Runs	17-38
Specify a Run Naming Rule	17-38
Overwrite a Run	17-39
Customize the Simulation Data Inspector Interface ...	17-41

Overview	17-41
Open the Simulation Data Inspector Tool	17-41
Why Is the Simulation Data Inspector Tool Empty?	17-42
Add/Delete a Column in the Signal Browser Table	17-42
Modify Grouping in Signal Browser Table	17-45
Rename a Run	17-47
Specify the Line Configuration	17-47
Select a Run or Signal Option in the Signal Browser Table	17-48
Display Run Properties	17-51
Modify a Plot in the Simulation Data Inspector Tool	17-51
Toolbar	17-53

Limitations of the Simulation Data Inspector	17-56
---	--------------

Record and Inspect Signal Data Programmatically ...	17-57
Overview	17-57
Run Management	17-57
Signal Management	17-58
Import/Export Data	17-59
Comparison Results	17-59
Create a Run in the Simulation Data Inspector	17-60
Compare Signal Data	17-60
Compare Runs of Simulation Data	17-61
Specify Signal Tolerances	17-62
Record Data During Parallel Simulations	17-63

Analyzing Simulation Results

18

Viewing Output Trajectories	18-2
Viewing and Exporting Simulation Data	18-2
Using the Scope Block	18-2
Using Return Variables	18-3
Using the To Workspace Block	18-3
Using the Simulation Data Inspector Tool	18-4
 Linearizing Models	 18-5
About Linearizing Models	18-5

Linearization with Referenced Models	18-7
Linearization Using the 'v5' Algorithm	18-9
Finding Steady-State Points	18-11

Improving Simulation Performance and Accuracy

19

About Improving Performance and Accuracy	19-2
Speed Up Simulation	19-3
Comparing Performance	19-5
Performance of the Simulation Modes	19-5
Measure Performance	19-7
Improve Acceleration Mode Performance	19-9
Techniques	19-9
C Compilers	19-10
Improve Simulation Accuracy	19-11

Performance Advisor

20

How Performance Advisor Improves Simulation Performance	20-2
Performance Advisor Workflow	20-3
Get Started with Performance Advisor	20-4
Prepare to Use Performance Advisor	20-4

Start Performance Advisor	20-4
Performance Advisor Window	20-5
Prepare a Model for Performance Advisor	20-9
Enable Data Logging for your Model	20-9
Create a Performance Advisor Baseline Measurement ...	20-10
Run Performance Advisor Checks	20-12
Use Performance Advisor Reports	20-14
View Performance Advisor Reports	20-14
Save Performance Advisor Reports	20-15
Operate on Performance Advisor Results	20-17
View Results	20-17
Respond to Results	20-18
Review the Actions Taken	20-19

Simulink Debugger

21

Introduction to the Debugger	21-2
Debugger Graphical User Interface	21-3
Displaying the Graphical Interface	21-3
Toolbar	21-4
Breakpoints Pane	21-5
Simulation Loop Pane	21-6
Outputs Pane	21-7
Sorted List Pane	21-8
Status Pane	21-9
Debugger Command-Line Interface	21-10
Controlling the Debugger	21-10
Method ID	21-10
Block ID	21-10

Accessing the MATLAB Workspace	21-11
Debugger Online Help	21-12
Start the Simulink Debugger	21-13
Starting from a Model Window	21-13
Starting from the Command Window	21-13
Start a Simulation	21-15
Run a Simulation Step by Step	21-17
Introduction	21-17
Block Data Output	21-18
Stepping Commands	21-19
Continuing a Simulation	21-20
Running a Simulation Nonstop	21-20
Set Breakpoints	21-22
About Breakpoints	21-22
Setting Unconditional Breakpoints	21-22
Setting Conditional Breakpoints	21-25
Display Information About the Simulation	21-28
Display Block I/O	21-28
Display Algebraic Loop Information	21-30
Display System States	21-31
Display Solver Information	21-32
Display Information About the Model	21-33
Display Model's Sorted Lists	21-33
Display a Block	21-34

Accelerating Models

22

What Is Acceleration?	22-2
-----------------------------	------

How Acceleration Modes Work	22-3
Overview	22-3
Normal Mode	22-3
Accelerator Mode	22-4
Rapid Accelerator Mode	22-5
Code Regeneration in Accelerated Models	22-7
Structural Changes That Cause Rebuilds	22-7
Determining If the Simulation Will Rebuild	22-8
Choosing a Simulation Mode	22-11
Simulation Mode Tradeoffs	22-11
Comparing Modes	22-12
Decision Tree	22-14
Design Your Model for Effective Acceleration	22-16
Select Blocks for Accelerator Mode	22-16
Select Blocks for Rapid Accelerator Mode	22-17
Control S-Function Execution	22-17
Accelerator and Rapid Accelerator Mode Data Type Considerations	22-18
Behavior of Scopes and Viewers with Rapid Accelerator Mode	22-19
Factors Inhibiting Acceleration	22-20
Perform Acceleration	22-23
Customize the Build Process	22-23
Run Acceleration Mode from the User Interface	22-24
Making Run-Time Changes	22-26
Interact with the Acceleration Modes	
Programmatically	22-27
Why Interact Programmatically?	22-27
Build Accelerator Mode MEX-files	22-27
Control Simulation	22-27
Simulate Your Model	22-28
Customize the Acceleration Build Process	22-29
Run Accelerator Mode with the Simulink Debugger ..	22-30
Advantages of Using Accelerator Mode with the Debugger	22-30

How to Run the Debugger	22-30
When to Switch Back to Normal Mode	22-31
Capture Performance Data	22-32
What Is the Profiler?	22-32
How the Profiler Works	22-32
Enabling the Profiler	22-34
How to Save Simulink Profiler Results	22-37

Managing Blocks

Working with Blocks

23

About Blocks	23-2
What Are Blocks?	23-2
Block Tool Tips	23-2
Virtual Blocks	23-2
Add Blocks	23-4
Ways to Add Blocks	23-4
Add Blocks by Browsing or Searching with the Library	
Browser	23-5
Copy Blocks from a Model	23-5
Add Frequently Used Blocks	23-6
Add Blocks Programmatically	23-8
Edit Blocks	23-9
Copy Blocks in a Model	23-9
Copy Blocks Between Windows	23-10
Move Blocks	23-11
Delete Blocks	23-13
Comment Blocks	23-14
Set Block Properties	23-15
Block Properties Dialog Box	23-15
General Block Properties	23-17
Block Annotation Properties	23-17

Block Callbacks	23-19
Create Block Annotations Programmatically	23-21
Change the Appearance of a Block	23-23
Change a Block Orientation	23-23
Resize a Block	23-25
Displaying Parameters Beneath a Block	23-26
Drop Shadows	23-26
Manipulate Block Names	23-27
Specify Block Color	23-29
Display Port Values	23-30
Port Value Data Tips	23-30
Display Value for a Port	23-31
Control the Port Value Display	23-31
Displayed Value When No Data Is Available	23-33
Port Value Display Limitations	23-34
Control and Display the Sorted Order	23-36
What Is Sorted Order?	23-36
Display the Sorted Order	23-36
Sorted Order Notation	23-37
How Simulink Determines the Sorted Order	23-47
Assign Block Priorities	23-50
Rules for Block Priorities	23-51
Block Priority Violations	23-54
Access Block Data During Simulation	23-56
About Block Run-Time Objects	23-56
Access a Run-Time Object	23-56
Listen for Method Execution Events	23-57
Synchronizing Run-Time Objects and Simulink Execution	23-58
Configure a Block for Code Generation	23-60

About Block Parameters	24-2
Set Block Parameters	24-4
Display a Block Parameter Dialog Box	24-4
Specify Parameter Values	24-6
About Parameter Values	24-6
Use Workspace Variables in Parameter Expressions	24-6
Resolve Variable References in Block Parameter Expressions	24-7
Use Parameter Objects to Specify Parameter Values	24-7
Convert Numeric Variable into Simulink.Parameter Object	24-8
Determine Parameter Data Types	24-8
Check Parameter Values	24-10
About Value Checking	24-10
Blocks That Perform Parameter Range Checking	24-10
Specify Ranges for Parameters	24-11
Perform Parameter Range Checking	24-12
Tunable Parameters	24-14
About Tunable Parameters	24-14
Tune a Block Parameter	24-14
Inline Parameters	24-15
About Inline Parameters	24-15
Specify Some Parameters as Nonlinear	24-15
Structure Parameters	24-17
About Structure Parameters	24-17
Define Structure Parameters	24-18
Referencing Structure Parameters	24-18
Structure Parameter Arguments	24-19
Tunable Structure Parameters	24-20
Parameter Structure Limitations	24-21

About Lookup Table Blocks	25-2
Anatomy of a Lookup Table	25-4
Lookup Tables Block Library	25-5
Guidelines for Choosing a Lookup Table	25-7
Data Set Dimensionality	25-7
Data Set Numeric and Data Types	25-7
Data Accuracy and Smoothness	25-8
Dynamics of Table Inputs	25-8
Efficiency of Performance	25-8
Summary of Lookup Table Block Features	25-10
Enter Breakpoints and Table Data	25-11
Entering Data in a Block Parameter Dialog Box	25-11
Entering Data in the Lookup Table Editor	25-13
Entering Data Using Inports of the Lookup Table Dynamic Block	25-16
Characteristics of Lookup Table Data	25-18
Sizes of Breakpoint Data Sets and Table Data	25-18
Monotonicity of Breakpoint Data Sets	25-19
Representation of Discontinuities in Lookup Tables	25-20
Formulation of Evenly Spaced Breakpoints	25-22
Methods for Estimating Missing Points	25-23
About Estimating Missing Points	25-23
Interpolation Methods	25-23
Extrapolation Methods	25-24
Rounding Methods	25-25
Example Output for Lookup Methods	25-26
Edit Existing Lookup Tables	25-28
When to Use the Lookup Table Editor	25-28
Layout of the Lookup Table Editor	25-28
Browsing Lookup Table Blocks	25-29

Editing Table Values	25-30
Working with Table Data of Standard Format	25-32
Working with Table Data of Nonstandard Format	25-37
Importing Data from an Excel Spreadsheet	25-54
Adding and Removing Rows and Columns in a Table	25-56
Displaying N-Dimensional Tables in the Editor	25-57
Plotting Lookup Tables	25-58
Editing Custom Lookup Table Blocks	25-62
Create a Logarithm Lookup Table	25-64
Prelookup and Interpolation Blocks	25-67
Optimize Generated Code for Lookup Table Blocks ...	25-68
Remove Code That Checks for Out-of-Range Inputs	25-68
Optimize Breakpoint Spacing in Lookup Tables	25-70
Update Lookup Table Blocks to New Versions	25-72
Comparison of Blocks with Current Versions	25-72
Compatibility of Models with Older Versions of Lookup Table Blocks	25-73
How to Update Your Model	25-74
What to Expect from the Model Advisor Check	25-75
Lookup Table Glossary	25-78

Working with Block Masks

26

Block Masks	26-3
What Are Masks?	26-3
When to Use Masks?	26-3
How Mask Parameters Work	26-5
Mask Code Execution	26-7
Mask Code Placement	26-7

Drawing Command Execution	26-7
Initialization Command Execution	26-8
Callback Code Execution	26-9
Mask Terminology	26-11
Mask a Block	26-12
Create mask	26-12
Define mask parameters	26-12
Set mask parameter values	26-13
Draw Mask Icon	26-15
Draw static icon	26-15
Draw dynamic icon	26-17
Additional examples	26-18
Create Mask Documentation	26-19
Initialize Mask	26-21
Mask Editor Initialization Pane	26-21
Dialog variables	26-22
Initialization Commands	26-23
Initialization Command Limitations	26-23
Best Practices for Masking	26-24
Use These Best Practices	26-24
Avoid These Practices	26-24
Considerations for Masking Model Blocks	26-25
Referenced Model Name	26-25
Variable Workspace	26-26
Masks on Blocks in User Libraries	26-27
About Masks and User-Defined Libraries	26-27
Masking a Block for Inclusion in a User Library	26-27
Masking a Block that Resides in a User Library	26-27
Masking a Block Copied from a User Library	26-28
Promote Underlying Block Parameters to Mask	26-29

Create Custom Interface for Simulink Blocks	26-32
Rules for Promoting Parameters	26-35
General Rules	26-35
Promotion from directly masked block	26-35
Promotion from child blocks within subsystems	26-36
Links created from masked blocks	26-36
Mask Blocks and Promote Parameters	26-37
Mask Built-In Blocks Directly and Within Subsystems ...	26-37
Create Custom Interface for Multiple Parameters in Subsystem	26-37
Operate on Existing Masks	26-41
Change a Block Mask	26-41
View Mask Parameters	26-41
Look Under Block Mask	26-41
Remove and Cache Mask	26-42
Restore Cached Mask	26-43
Permanently Delete Mask	26-43
Calculate Values Used Under the Mask	26-44
Control Masks Programmatically	26-47
Use Simulink.Mask and Simulink.MaskParameter	26-47
Use get_param and set_param	26-49
Create Dynamic Mask Dialog Boxes	26-50
About Dynamic Masked Dialog Boxes	26-50
Show parameter	26-51
Enable parameter	26-51
Setting Masked Block Dialog Box Parameters	26-51
Setting Nested Masked Block Parameters	26-53
Create Dynamic Masked Subsystems	26-54
Allow library block to modify its contents	26-54
Create Self-Modifying Masks for Library Blocks	26-54
Evaluate Blocks Under Self-Modifying Mask	26-58
Debug Masks That Use MATLAB Code	26-60

Code Written in Mask Editor	26-60
Code Written Using MATLAB Editor/Debugger	26-60
Masking Linked Blocks	26-61
Guidelines for Mask Parameters	26-62
Mask Behavior for Masked, Linked Blocks	26-63
Mask a Linked Block	26-64
Create a Custom Library With Mask on Link Block	26-64
Add a Mask to the Masked, Link Block	26-64
View Masks Below the Top Mask	26-65

Creating Custom Blocks

27

When to Create Custom Blocks	27-2
Types of Custom Blocks	27-3
MATLAB Function Blocks	27-3
Subsystem Blocks	27-4
S-Function Blocks	27-4
Comparison of Custom Block Functionality	27-7
Custom Block Considerations	27-7
Modeling Requirements	27-10
Speed and Code Generation Requirements	27-13
Expanding Custom Block Functionality	27-17
Create a Custom Block	27-18
How to Design a Custom Block	27-18
Defining Custom Block Behavior	27-20
Deciding on a Custom Block Type	27-21
Placing Custom Blocks in a Library	27-27
Adding a User Interface to a Custom Block	27-29
Adding Block Functionality Using Block Callbacks	27-38

Custom Block Examples	27-44
Creating Custom Blocks from Masked Library Blocks	27-44
Creating Custom Blocks from MATLAB Functions	27-44
Creating Custom Blocks from S-Functions	27-45

Working with Block Libraries

28

About Block Libraries and Linked Blocks	28-2
Block Libraries	28-2
Benefits of Block Libraries	28-2
Library Browser	28-2
Linked Blocks	28-2
Create and Work with Linked Blocks	28-4
About Linked Blocks	28-4
Create a Linked Block	28-4
Update a Linked Block	28-5
Modify Linked Blocks	28-6
Find a Linked Block's Prototype	28-7
Find Linked Blocks in a Model	28-7
Work with Library Links	28-8
Display Library Links	28-8
Lock Links to Blocks in a Library	28-9
Disable Links to Library Blocks	28-11
Restore Disabled or Parameterized Links	28-12
Check and Set Link Status Programmatically	28-16
Break a Link to a Library Block	28-18
Fix Unresolved Library Links	28-19
Create Block Libraries	28-20
Create a Library	28-20
Create a Sublibrary	28-21
Modify and Lock Libraries	28-21
Make Backward-Compatible Changes to Libraries	28-22
Add Libraries to the Library Browser	28-32
Display a Library in the Library Browser	28-32

Example of a Minimal sblocks.m File	28-32
Add More Descriptive Information in sblocks.m	28-33

Using the MATLAB Function Block

29

Integrate MATLAB Algorithm in Model	29-4
Defining Local Variables for Code Generation	29-4
What Is a MATLAB Function Block?	29-6
Calling Functions in MATLAB Function Blocks	29-6
Why Use MATLAB Function Blocks?	29-8
Create Model That Uses MATLAB Function Block	29-9
Adding a MATLAB Function Block to a Model	29-9
Programming the MATLAB Function Block	29-10
Building the Function and Checking for Errors	29-12
Defining Inputs and Outputs	29-14
Code Generation Readiness Tool	29-16
What Information Does the Code Generation Readiness Tool Provide?	29-16
Summary Tab	29-17
Code Structure Tab	29-19
See Also	29-22
Check Code Using the Code Generation Readiness Tool	29-23
Run Code Generation Readiness Tool at the Command Line	29-23
Run the Code Generation Readiness Tool From the Current Folder Browser	29-23
Debugging a MATLAB Function Block	29-24
How Debugging Affects Simulation Speed	29-24
Enabling and Disabling Debugging	29-24
Debugging the Function in Simulation	29-24

Watching Function Variables During Simulation	29-28
Checking for Data Range Violations	29-30
Debugging Tools	29-30
MATLAB Function Block Editor	29-34
Customizing the MATLAB Function Block Editor	29-34
MATLAB Function Block Editor Tools	29-34
Editing and Debugging MATLAB Function Block Code ...	29-35
Ports and Data Manager	29-38
MATLAB Function Reports	29-53
About MATLAB Function Reports	29-53
Location of MATLAB Function Reports	29-53
Opening MATLAB Function Reports	29-54
Description of MATLAB Function Reports	29-54
Viewing Your MATLAB Function Code	29-54
Viewing Call Stack Information	29-56
Viewing the Compilation Summary Information	29-56
Viewing Error and Warning Messages	29-56
Viewing Variables in Your MATLAB Code	29-58
Keyboard Shortcuts for the MATLAB Function Report ...	29-64
Report Limitations	29-65
Type Function Arguments	29-67
About Function Arguments	29-67
Specifying Argument Types	29-67
Inheriting Argument Data Types	29-69
Built-In Data Types for Arguments	29-70
Specifying Argument Types with Expressions	29-71
Specifying Fixed-Point Designer Data Properties	29-71
Size Function Arguments	29-75
Specifying Argument Size	29-75
Inheriting Argument Sizes from Simulink	29-75
Specifying Argument Sizes with Expressions	29-76
Add Parameter Arguments	29-77
Resolve Signal Objects for Output Data	29-78
Implicit Signal Resolution	29-78

Eliminating Warnings for Implicit Signal Resolution in the Model	29-78
Disabling Implicit Signal Resolution for a MATLAB Function Block	29-79
Forcing Explicit Signal Resolution for an Output Data Signal	29-79
Types of Structures in MATLAB Function Blocks	29-80
Attach Bus Signals to MATLAB Function Blocks	29-81
Structure Definitions in Example	29-81
Bus Objects Define Structure Inputs and Outputs	29-81
How Structure Inputs and Outputs Interface with Bus Signals	29-83
Working with Virtual and Nonvirtual Buses	29-83
Rules for Defining Structures in MATLAB Function Blocks	29-84
Index Substructures and Fields	29-85
Create Structures in MATLAB Function Blocks	29-86
Assign Values to Structures and Fields	29-88
Initialize a Matrix Using a Non-Tunable Structure Parameter	29-90
Define and Use Structure Parameters	29-93
Defining Structure Parameters	29-93
FIMATH Properties of Non-Tunable Structure Parameters	29-93
Limitations of Structures and Buses in MATLAB Function Blocks	29-95
What Is Variable-Size Data?	29-96

How MATLAB Function Blocks Implement	
Variable-Size Data	29-97
Enable Support for Variable-Size Data	29-98
Declare Variable-Size Inputs and Outputs	29-99
Filter a Variable-Size Signal	29-100
About the Example	29-100
Simulink Model	29-100
Source Signal	29-101
MATLAB Function Block: uniquify	29-101
MATLAB Function Block: avg	29-103
Variable-Size Results	29-104
Enumerated Types Supported in MATLAB Function	
Blocks	29-107
Define Enumerated Data Types for MATLAB Function	
Blocks	29-109
Add Inputs, Outputs, and Parameters as Enumerated	
Data	29-110
Basic Approach for Adding Enumerated Data to	
MATLAB Function Blocks	29-112
Instantiate Enumerated Data in MATLAB Function	
Blocks	29-113
Control an LED Display	29-114
About the Example	29-114
Class Definition: switchmode	29-114
Class Definition: led	29-115
Simulink Model	29-115
MATLAB Function Block: checkState	29-116
How the Model Displays Enumerated Data	29-117
Operations on Enumerated Data	29-118

Using Enumerated Data in MATLAB Function	
Blocks	29-119
When to Use Enumerated Data	29-119
Limitations of Enumerated Types	29-119
Share Data Globally	29-120
When Do You Need to Use Global Data?	29-120
Using Global Data with the MATLAB Function Block	29-120
Choosing How to Store Global Data	29-121
How to Use Data Store Memory Blocks	29-123
How to Use Simulink.Signal Objects	29-125
Using Data Store Diagnostics to Detect Memory Access	
Issues	29-127
Limitations of Using Shared Data in MATLAB Function	
Blocks	29-128
Add Frame-Based Signals	29-129
About Frame-Based Signals	29-129
Supported Types for Frame-Based Data	29-129
Adding Frame-Based Data in MATLAB Function Blocks ..	29-130
Examples of Frame-Based Signals in MATLAB Function	
Blocks	29-131
Create Custom Block Libraries	29-135
When to Use MATLAB Function Block Libraries	29-135
How to Create Custom MATLAB Function Block	
Libraries	29-135
Example: Creating a Custom Signal Processing Filter Block	
Library	29-136
Code Reuse with Library Blocks	29-149
Debugging MATLAB Function Library Blocks	29-154
Properties You Can Specialize Across Instances of Library	
Blocks	29-154
Use Traceability in MATLAB Function Blocks	29-156
Extent of Traceability in MATLAB Function Blocks	29-156
Traceability Requirements	29-156
Basic Workflow for Using Traceability	29-157
Tutorial: Using Traceability in a MATLAB Function	
Block	29-158

Include MATLAB Code as Comments in Generated Code	
Code	29-161
How to Include MATLAB Code as Comments in the Generated Code	29-161
Location of Comments in Generated Code	29-162
Including MATLAB Function Help Text in the Function Banner	29-165
Limitations of MATLAB Source Code as Comments	29-165
Enhance Code Readability for MATLAB Function Blocks	29-167
Requirements for Using Readability Optimizations	29-167
Converting If-Elseif-Else Code to Switch-Case Statements	29-167
Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements	29-170
Control Run-Time Checks	29-176
Types of Run-Time Checks	29-176
When to Disable Run-Time Checks	29-176
How to Disable Run-Time Checks	29-177
Track Object Using MATLAB Code	29-178
Learning Objectives	29-178
Tutorial Prerequisites	29-179
Example: The Kalman Filter	29-179
Files for the Tutorial	29-182
Tutorial Steps	29-184
Best Practices Used in This Tutorial	29-204
Key Points to Remember	29-204
Where to Learn More	29-204
Filter Audio Signal Using MATLAB Code	29-206
Learning Objectives	29-206
Tutorial Prerequisites	29-206
Example: The LMS Filter	29-207
Files for the Tutorial	29-210
Tutorial Steps	29-211
Encapsulating the Interface to External Code	29-239

Encapsulate Interface to an External C Library	29-240
Best Practices for Using <code>coder.ExternalDependency</code> ..	29-243
Terminate Code Generation for Unsupported External Dependency	29-243
Parameterize Methods for MATLAB and Generated Code	29-243
Parameterize <code>updateBuildInfo</code> for Multiple Platforms	29-244
Update Build Information from MATLAB code	29-245

System Objects

30

What Are System Objects?	30-2
System Design and Simulation in Simulink	30-4
System Objects in MATLAB Code Generation	30-5
System Objects in Generated Code	30-5
System Objects in <code>codegen</code>	30-9
System Objects in the MATLAB Function Block	30-9
System Objects in the MATLAB System Block	30-10
System Objects and MATLAB Compiler Software	30-10
System Objects in Simulink	30-11
System Objects in the MATLAB Function Block	30-11
System Objects in the MATLAB System Block	30-11
System Object Methods	30-12
What Are System Object Methods?	30-12
The <code>Step</code> Method	30-12
Common Methods	30-13
System Design in Simulink Using System Objects	30-15
Define New Kinds of System Objects for Use in Simulink ..	30-15
Test New System Objects in MATLAB	30-21

Add System Objects to Your Simulink Model	30-21
---	-------

Define New System Objects

31

Define Basic System Objects	31-3
Change Number of Step Inputs or Outputs	31-5
Specify System Block Input and Output Names	31-8
Validate Property and Input Values	31-10
Initialize Properties and Setup One-Time Calculations	31-13
Set Property Values at Construction Time	31-16
Reset Algorithm State	31-19
Define Property Attributes	31-21
Hide Inactive Properties	31-25
Limit Property Values to Finite String Set	31-27
Process Tuned Properties	31-30
Release System Object Resources	31-32
Define Composite System Objects	31-34
Define Finite Source Objects	31-38

Save System Object	31-40
Load System Object	31-43
Clone System Object	31-46
Define System Block Icon	31-47
Add Header to System Block Dialog	31-49
Add Property Groups to System Object and Block Dialog	31-51
Set Output Size	31-55
Set Output Data Type	31-58
Set Output Complexity	31-61
Specify Whether Output Is Fixed- or Variable-Size	31-64
Specify Discrete State Output Specification	31-67
Use Update and Output for Nondirect Feedthrough ..	31-70
Methods Timing	31-73
Setup Method Call Sequence	31-73
Step Method Call Sequence	31-74
Reset Method Call Sequence	31-74
Release Method Call Sequence	31-75
System Object Input Arguments and ~ in Code Examples	31-76
What Are Mixin Classes?	31-77

What Is the MATLAB System Block?	32-2
Why Use the MATLAB System Block?	32-2
Choosing the Right Block Type	32-2
System Objects	32-3
Interpreted Execution or Code Generation	32-3
MATLAB System Block Limitations	32-4
MATLAB System and System Objects Examples	32-5
Implement a MATLAB System Block	32-7
Understanding the MATLAB System Block	32-8
Change Blocks Implemented with System Objects	32-10
Change Block Icon and Port Labels	32-11
Modify MATLAB System Block Dialog	32-11
Use System Objects in Feedback Loops	32-13
Simulation Modes	32-15
Interpreted Execution vs. Code Generation	32-15
Simulation Using Code Generation	32-16
Mapping System Objects to Block Dialog Box	32-17
System Object to Block Dialog Box Default Mapping	32-17
System Object to Block Dialog Box Custom Mapping	32-19
Considerations for Using System Objects in	
Simulink	32-22
System Objects in Simulink	32-22
Simulink Engine Interaction with System Object	
Methods	32-24
Simulink Engine Phases Mapped to System Object	
Methods	32-24
Add and Implement Propagation Methods	32-27

When to Use Propagation Methods	32-27
Add Propagation Methods to System Objects	32-27
Implement Propagation Methods	32-28
Troubleshoot System Objects in Simulink	32-30
Class Not Found	32-30
Error Invoking Object Method	32-30
Performance	32-31

Design Considerations for C/C++ Code Generation

33

When to Generate Code from MATLAB Algorithms ...	33-2
When Not to Generate Code from MATLAB Algorithms ..	33-2
Which Code Generation Feature to Use	33-4
Prerequisites for C/C++ Code Generation from MATLAB	33-6
MATLAB Code Design Considerations for Code Generation	33-7
See Also	33-8
Expected Differences in Behavior After Compiling	
MATLAB Code	33-9
Why Are There Differences?	33-9
Character Size	33-9
Order of Evaluation in Expressions	33-9
Termination Behavior	33-10
Size of Variable-Size N-D Arrays	33-10
Size of Empty Arrays	33-11
Floating-Point Numerical Results	33-11
NaN and Infinity Patterns	33-12
Code Generation Target	33-12
MATLAB Class Initial Values	33-12
Variable-Size Support for Code Generation	33-12

MATLAB Language Features Supported for C/C++ Code	
Generation	33-13
MATLAB Language Features Not Supported for C/C++	
Code Generation	33-14

Functions Supported for Code Generation

34

Functions Supported for C/C++ Code Generation —	
Alphabetical List	34-2

Functions Supported for C/C++ Code Generation —	
Categorical List	34-80
Aerospace Toolbox Functions	34-81
Arithmetic Operator Functions	34-82
Bit-Wise Operation Functions	34-82
Casting Functions	34-83
Communications System Toolbox Functions	34-83
Complex Number Functions	34-83
Computer Vision System Toolbox Functions	34-84
Data and File Management Functions	34-86
Data Type Functions	34-86
Derivative and Integral Functions	34-87
Discrete Math Functions	34-87
Error Handling Functions	34-88
Exponential Functions	34-88
Filtering and Convolution Functions	34-89
Fixed-Point Designer Functions	34-89
Histogram Functions	34-98
Image Processing Toolbox Functions	34-98
Input and Output Functions	34-100
Interpolation and Computational Geometry Functions ...	34-101
Linear Algebra	34-101
Logical Operator Functions	34-101
MATLAB Compiler Functions	34-102
MATLAB Desktop Environment Functions	34-102
Matrix and Array Functions	34-102
Nonlinear Numerical Methods	34-106
Phased Array System Toolbox Functions	34-107
Polynomial Functions	34-110

Programming Utilities	34-110
Relational Operator Functions	34-110
Rounding and Remainder Functions	34-111
Set Functions	34-111
Signal Processing Functions in MATLAB	34-112
Signal Processing Toolbox Functions	34-112
Special Values	34-117
Specialized Math	34-117
Statistical Functions	34-118
Statistics Toolbox Functions	34-118
String Functions	34-124
Structure Functions	34-125
Trigonometric Functions	34-125

System Objects Supported for Code Generation

35

System Objects Supported for C/C++ Code

Generation	35-2
Code Generation for System Objects	35-2
Computer Vision System Toolbox System Objects	35-2
Communications System Toolbox System Objects	35-7
DSP System Toolbox System Objects	35-13
Phased Array System Toolbox System Objects	35-19
Image Acquisition Toolbox System Objects	35-23

Defining MATLAB Variables for C/C++ Code Generation

36

Variables Definition for Code Generation	36-2
--	------

Best Practices for Defining Variables for C/C++ Code

Generation	36-3
Define Variables By Assignment Before Using Them	36-3
Use Caution When Reassigning Variables	36-6

Use Type Cast Operators in Variable Definitions	36-6
Define Matrices Before Assigning Indexed Variables	36-6
Eliminate Redundant Copies of Variables in Generated Code	36-7
When Redundant Copies Occur	36-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables	36-7
Defining Uninitialized Variables	36-8
Reassignment of Variable Properties	36-9
Define and Initialize Persistent Variables	36-10
Reuse the Same Variable with Different Properties ...	36-11
When You Can Reuse the Same Variable with Different Properties	36-11
When You Cannot Reuse Variables	36-12
Limitations of Variable Reuse	36-14
Avoid Overflows in for-Loops	36-16
Supported Variable Types	36-18

Defining Data for Code Generation

37

Data Definition for Code Generation	37-2
Code Generation for Complex Data	37-4
Restrictions When Defining Complex Variables	37-4
Expressions With Complex Operands Yield Complex Results	37-5
Code Generation for Characters	37-6

What Is Variable-Size Data?	38-2
Variable-Size Data Definition for Code Generation ...	38-3
Bounded Versus Unbounded Variable-Size Data	38-4
Control Memory Allocation of Variable-Size Data	38-5
Specify Variable-Size Data Without Dynamic Memory	
Allocation	38-6
Fixing Upper Bounds Errors	38-6
Specifying Upper Bounds for Variable-Size Data	38-6
Variable-Size Data in Code Generation Reports	38-9
What Reports Tell You About Size	38-9
How Size Appears in Code Generation Reports	38-10
How to Generate a Code Generation Report	38-10
Define Variable-Size Data for Code Generation	38-11
When to Define Variable-Size Data Explicitly	38-11
Using a Matrix Constructor with Nonconstant	
Dimensions	38-12
Inferring Variable Size from Multiple Assignments	38-12
Defining Variable-Size Data Explicitly Using	
coder.varsize	38-13
C Code Interface for Arrays	38-17
C Code Interface for Statically Allocated Arrays	38-17
Diagnose and Fix Variable-Size Data Errors	38-18
Diagnosing and Fixing Size Mismatch Errors	38-18
Diagnosing and Fixing Errors in Detecting Upper	
Bounds	38-20
Incompatibilities with MATLAB in Variable-Size	
Support for Code Generation	38-22

Incompatibility with MATLAB for Scalar Expansion	38-22
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	38-24
Incompatibility with MATLAB in Determining Size of Empty Arrays	38-25
Incompatibility with MATLAB in Determining Class of Empty Arrays	38-26
Incompatibility with MATLAB in Vector-Vector Indexing	38-27
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation	38-28
Incompatibility with MATLAB in Concatenating Variable-Size Matrices	38-29
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks	38-29
 Restrictions on Variable Sizing in Toolbox Functions	
Supported for Code Generation	38-30
Common Restrictions	38-30
Toolbox Functions with Variable Sizing Restrictions	38-31

Code Generation for MATLAB Structures

39

Structure Definition for Code Generation	39-2
Structure Operations Allowed for Code Generation	39-3
Define Scalar Structures for Code Generation	39-4
Restriction When Using struct	39-4
Restrictions When Defining Scalar Structures by Assignment	39-4
Adding Fields in Consistent Order on Each Control Flow Path	39-4
Restriction on Adding New Fields After First Use	39-5
 Define Arrays of Structures for Code Generation	39-7
Ensuring Consistency of Fields	39-7

Using repmat to Define an Array of Structures with Consistent Field Properties	39-7
Defining an Array of Structures Using Concatenation	39-8
Make Structures Persistent	39-9
Index Substructures and Fields	39-10
Assign Values to Structures and Fields	39-12
Pass Large Structures as Input Parameters	39-14

Code Generation for Enumerated Data

40

Enumerated Data Definition for Code Generation	40-2
Enumerated Types Supported for Code Generation ...	40-3
Enumerated Type Based on Simulink.IntEnumType	40-3
When to Use Enumerated Data for Code Generation ..	40-4
Generate Code for Enumerated Data from MATLAB Function Blocks	40-5
Define Enumerated Data for Code Generation	40-6
Naming Enumerated Types for Code Generation	40-7
Instantiate Enumerated Types for Code Generation ..	40-8
Operations on Enumerated Data Allowed for Code Generation	40-9
Assignment Operator, =	40-9
Relational Operators, <> <= >= == ~=	40-9
Cast Operation	40-10
Indexing Operation	40-10

Control Flow Statements: if, switch, while	40-11
Include Enumerated Data in Control Flow	
Statements	40-12
if Statement with Enumerated Data Types	40-12
switch Statement with Enumerated Data Types	40-13
while Statement with Enumerated Data Types	40-16
Customize Enumerated Types Based on	
Simulink.IntEnumType	40-18
Control Names of Enumerated Type Values in	
Generated Code	40-19
Change and Reload Enumerated Data Types	40-21
Restrictions on Use of Enumerated Data in	
for-Loops	40-22
Toolbox Functions That Support Enumerated Types for	
Code Generation	40-23

Code Generation for MATLAB Classes

41

MATLAB Classes Definition for Code Generation	41-2
Language Limitations	41-2
Code Generation Features Not Compatible with Classes ..	41-3
Defining Class Properties for Code Generation	41-4
Calls to Base Class Constructor	41-6
Classes That Support Code Generation	41-8
Generate Code for MATLAB Value Classes	41-9

Generate Code for MATLAB Handle Classes and System Objects	41-15
MATLAB Classes in Code Generation Reports	41-18
What Reports Tell You About Classes	41-18
How Classes Appear in Code Generation Reports	41-18
How to Generate a Code Generation Report	41-20
Troubleshooting Issues with MATLAB Classes	41-21
Class <i>class</i> does not have a property with name <i>name</i> ...	41-21

Code Generation for Function Handles

42

Function Handle Definition for Code Generation	42-2
Define and Pass Function Handles for Code Generation	42-3
Function Handle Limitations for Code Generation ...	42-5

Defining Functions for Code Generation

43

Specify Variable Numbers of Arguments	43-2
Supported Index Expressions	43-3
Apply Operations to a Variable Number of Arguments	43-4
When to Force Loop Unrolling	43-4
Using Variable Numbers of Arguments in a for-Loop	43-5
Implement Wrapper Functions	43-7

Passing Variable Numbers of Arguments from One Function to Another	43-7
Pass Property/Value Pairs	43-8
Variable Length Argument Lists for Code Generation	43-10

Calling Functions for Code Generation

44

Resolution of Function Calls in MATLAB Generated Code	44-2
Key Points About Resolving Function Calls	44-4
Compile Path Search Order	44-4
When to Use the Code Generation Path	44-5
Resolution of File Types on Code Generation Path ...	44-6
Compilation Directive %#codegen	44-8
Call Local Functions	44-9
Call Supported Toolbox Functions	44-10
Call MATLAB Functions	44-11
Declaring MATLAB Functions as Extrinsic Functions ...	44-12
Calling MATLAB Functions Using feval	44-16
How MATLAB Resolves Extrinsic Functions During Simulation	44-16
Working with mxArray	44-17
Restrictions on Extrinsic Functions for Code Generation ..	44-19
Limit on Function Arguments	44-19

Optimization Strategies	45-3
Modularize MATLAB Code	45-6
Eliminate Redundant Copies of Function Inputs	45-7
Inline Code	45-9
Prevent Function Inlining	45-9
Use Inlining in Control Flow Statements	45-9
Control Inlining Using Configuration Object	45-11
Control Size of Functions Inlined	45-11
Control Size of Functions After Inlining	45-12
Control Stack Size Limit on Inlined Functions	45-12
Fold Function Calls into Constants	45-14
Control Stack Space Usage	45-16
Stack Allocation and Performance	45-17
Rewrite Logical Array Indexing as a Loop	45-18
Dynamic Memory Allocation and Performance	45-19
When Dynamic Memory Allocation Occurs	45-19
Minimize Dynamic Memory Allocation	45-21
Provide Maximum Size for Variable-size Arrays	45-22
Disable Dynamic Memory Allocation During Code Generation	45-29
Set Dynamic Memory Allocation Threshold	45-30

Set Dynamic Memory Allocation Threshold Using Project Interface	45-30
Set Dynamic Memory Allocation Threshold from Command Line	45-32
Excluding Unused Paths from Generated Code	45-33
Prevent Code Generation for Unused Execution Paths	45-34
Prevent Code Generation When Local Variable Controls Flow	45-34
Prevent Code Generation When Input Variable Controls Flow	45-35
Generate Code with Parallel for-loops (parfor)	45-37
Minimize Redundant Operations in Loops	45-39
Unroll for-Loops	45-41
Limit Copying the for-loop Body in Generated Code	45-41
Support for Integer Overflow and Non-Finites	45-44
Disable Support for Integer Overflow	45-44
Disable Support for Non-Finites	45-45
Integrate Custom Code	45-46
MATLAB Coder Optimizations in Generated Code	45-52
Constant Folding	45-52
Loop Fusion	45-53
Successive Matrix Operations Combined	45-54
Unreachable Code Elimination	45-54
Generate Reusable Code	45-56

Data Types	46-2
About Data Types	46-2
Data Types Supported by Simulink	46-3
Fixed-Point Data	46-4
Enumerations	46-6
Bus Objects	46-6
Block Support for Data and Signal Types	46-7
Create Signals of a Specific Data Type	46-7
Specify Block Output Data Types	46-8
Specify Data Types Using Data Type Assistant	46-16
Display Port Data Types	46-28
Data Type Propagation	46-29
Data Typing Rules	46-29
Typecast Signals	46-30
Validate a Floating-Point Embedded Model	46-30
Validate a Single-Precision Model	46-31
Data Objects	46-35
About Data Object Classes	46-35
About Data Object Methods	46-36
Create Data Objects Using Model Explorer	46-38
Using the Model Explorer to Create Data Objects	46-39
About Object Properties	46-40
Changing Object Properties	46-41
Handle Versus Value Classes	46-42
Comparing Data Objects	46-44
Saving and Loading Data Objects	46-44
Using Data Objects in Simulink Models	46-45
Creating Persistent Data Objects	46-45
Data Object Wizard	46-45
Define Data Classes	46-51
Supported Property Types	46-56
Upgrade Level-1 Data Classes	46-57

Associating User Data with Blocks	46-59
Design Minimum and Maximum	46-60
Use of Design Minimum and Maximum	46-60
Valid Values for Design Minimum and Maximum	46-60

Enumerations and Modeling

47

About Simulink Enumerations	47-2
Define Simulink Enumerations	47-3
Workflow to Define a Simulink Enumeration	47-3
Create Simulink Enumeration Class	47-3
Customize Simulink Enumeration	47-4
Save Enumeration in a MATLAB File	47-7
Change and Reload Enumerations	47-7
Import Enumerations Defined Externally to MATLAB ...	47-8
Use Enumerated Data in Simulink Models	47-10
Simulate with Enumerations	47-10
Specify Enumerations as Data Types	47-12
Get Information About Enumerations	47-13
Enumeration Value Display	47-13
Instantiate Enumerations	47-15
Enumerated Values in Computation	47-19
Simulink Constructs that Support Enumerations	47-22
Overview	47-22
Block Support	47-22
Class Support	47-24
Logging Enumerated Data	47-24
Importing Enumerated Data	47-24
Simulink Enumeration Limitations	47-25
Enumerations and Scopes	47-25
Enumerated Types for Switch Blocks	47-25
Nonsupport of Enumerations	47-25

Using Simulation Data	48-3
Working with Simulation Data	48-3
Export Simulation Data	48-4
Simulation Data	48-4
Approaches for Exporting Signal Data	48-4
Enable Simulation Data Export	48-6
View Logged Simulation Data With the Simulation Data Inspector	48-7
Memory Performance	48-7
Data Format for Exported Simulation Data	48-9
Data Format for Block-Based Exported Data	48-9
Data Format for Model-Based Exported Data	48-9
Signal Logging Format	48-9
Logged Data Store Format	48-10
State and Output Data Format	48-10
Limit Amount of Exported Data	48-15
Decimation	48-15
Limit Data Points to Last	48-15
Samples to Export for Variable-Step Solvers	48-17
Output Options	48-17
Refine Output	48-17
Produce Additional Output	48-18
Produce Specified Output Only	48-19
Export Signal Data Using Signal Logging	48-21
Signal Logging	48-21
Signal Logging Workflow	48-21
Signal Logging in Rapid Accelerator Mode	48-22
Signal Logging for Array of Buses Signals	48-22
Signal Logging Limitations	48-23
Configure a Signal for Signal Logging	48-24
Enable Logging for a Signal	48-24

Specify Logging Name	48-27
Limit the Data Logged for a Signal	48-29
View Signal Logging Configuration	48-31
Approaches for Viewing the Signal Logging	
Configuration	48-31
Use Simulink Editor to View Signal Logging	
Configuration	48-32
Use Signal Logging Selector to View Signal Logging	
Configuration	48-34
Use Model Explorer to View Signal Logging	
Configuration	48-36
Enable Signal Logging for a Model	48-38
Enable and Disable Logging Globally for a Model	48-38
Specify the Signal Logging Data Format	48-39
Specify a Name for the Signal Logging Data	48-44
Override Signal Logging Settings	48-45
Benefits of Overriding Signal Logging Settings	48-45
Two Interfaces for Overriding Signal Logging Settings ...	48-45
Scope of Signal Logging Setting Overrides	48-46
Signal Logging Selector	48-47
Command-Line Interface for Overriding Signal Logging	
Settings	48-53
Access Signal Logging Data	48-59
View Signal Logging Data	48-59
Signal Logging Object	48-60
View Logged Signal Data with the Simulation Data	
Inspector	48-60
Programmatically Access Logged Signal Data Saved in	
Dataset Format	48-61
Programmatically Access Logged Signal Data Saved in	
ModelDataLogs Format	48-67
Techniques for Importing Signal Data	48-72
Signal Data Import Techniques Summary	48-72
Comparison of Techniques	48-73
Time and Signal Values for Imported Data	48-75

Import Data to Model a Continuous Plant	48-78
Share Simulation Data Across Models	48-78
Example of Importing Data to Model a Continuous Plant	48-78
Import Data to Test a Discrete Algorithm	48-81
Specify a Signal-Only Structure	48-81
Example of Importing Data to Test a Discrete Algorithm ..	48-81
Import Data for an Input Test Case	48-82
Guidelines for Importing a Test Case	48-82
Example of Test Case Data	48-82
Use From Workspace Block to Import an Input Test Case	48-83
Use Signal Builder Block to Import an Input Test Case ..	48-84
Import Signal Logging Data	48-86
Import Data to Root-Level Input Ports	48-88
Root-Level Input Ports	48-88
Enable Data Import	48-88
Input Data	48-89
Import Bus Data	48-91
Import and Map Root-Level Inport Data	48-92
Importing and Mapping Workflow	48-94
MAT-File for Import and Mapping	48-95
Import and Map Signal Data	48-97
Import and Map Bus Data	48-98
Understanding Mapping Results	48-102
Convert Test Harness Model to Harness-Free Mode	48-107
Converting Test Harness Model to Harness-Free Model ..	48-107
Alternative Workflows to Load Data	48-116
Create Custom Mapping File Function	48-117
Import MATLAB timeseries Data	48-121
Specify Time Dimension	48-121
Models with Multiple Root Inport Blocks	48-122
Import Structures of timeseries Objects for Buses	48-123
Imported Bus Data Requirements	48-123

Convert Simulink.TsArray Objects	48-123
Import Bus Data	48-124
Import Array of Buses Data	48-126
Import Simulink.Timeseries and Simulink.TsArray	
Data	48-133
Use MATLAB Timeseries for New Models	48-133
Simulink.TsArray Data	48-133
Import Data Arrays	48-134
Data Array Format	48-134
Specify the Input Expression	48-134
Import MATLAB Time Expression Data	48-136
Specify the Input Expression	48-136
Import Data Structures	48-137
Data Structures	48-137
One Structure for All Ports or a Structure for Each Port ..	48-138
Specify Signal Data	48-138
Specify Time Data	48-139
Examples of Specifying Signal and Time Data	48-140
Import and Export States	48-142
State Information	48-142
Save State Information	48-142
Import Initial States	48-146
Import and Export State Information for Referenced	
Models	48-149

Working with Data Stores

49

About Data Stores	49-2
Local and Global Data Stores	49-2
When to Use a Data Store	49-3
Create Data Stores	49-3
Access Data Stores	49-4

Configure Data Stores	49-5
Data Stores with Buses and Arrays of Buses	49-5
Data Stores with Data Store Memory Blocks	49-8
Creating the Data Store	49-8
Specifying Data Store Memory Block Attributes	49-8
Data Stores with Signal Objects	49-12
Creating the Data Store	49-12
Local and Global Data Stores	49-12
Signal Object Attributes for Data Stores	49-12
Access Data Stores with Simulink Blocks	49-15
Writing to a Data Store	49-15
Reading from a Data Store	49-15
Accessing a Global Data Store	49-16
Accessing Specific Bus and Matrix Elements	49-17
Data Store Examples	49-24
Overview	49-24
Local Data Store Example	49-24
Global Data Store Example	49-25
Log Data Stores	49-27
Logging Local and Global Data Store Values	49-27
Supported Data Types, Dimensions, and Complexity for Logging Data Stores	49-27
Data Store Logging Limitations	49-28
Logging Data Stores Created with a Data Store Memory Block	49-28
Logging Icon for the Data Store Memory Block	49-29
Logging Data Stores Created with a Simulink.Signal Object	49-29
Accessing Data Store Logging Data	49-30
Order Data Store Access	49-32
About Data Store Access Order	49-32
Ordering Access Using Function Call Subsystems	49-32
Ordering Access Using Block Priorities	49-36
Data Store Diagnostics	49-39

About Data Store Diagnostics	49-39
Detecting Access Order Errors	49-39
Detecting Multitasking Access Errors	49-42
Detecting Duplicate Name Errors	49-44
Data Store Diagnostics in the Model Advisor	49-46
Data Stores and Software Verification	49-47

Managing Signals

Working with Signals

50

Signal Basics	50-2
About Signals	50-2
Creating Signals	50-3
Signal Line Styles	50-4
Signal Properties	50-5
Testing Signals	50-6
Signal Types	50-8
Summary of Signal Types	50-8
Control Signals	50-9
Composite (Bus) Signals	50-9
Virtual Signals	50-11
About Virtual Signals	50-11
Mux Signals	50-11
Signal Values	50-15
Signal Data Types	50-15
Signal Dimensions, Size, and Width	50-15
Complex Signals	50-16
Initializing Signal Values	50-16
Viewing Signal Values	50-17
Displaying Signal Values in Model Diagrams	50-17
Exporting Signal Data	50-18

Signal Names and Labels	50-19
Signal Names	50-19
Signal Labels	50-22
Signal Label Propagation	50-24
Propagated Signal Labels	50-24
Blocks That Support Signal Label Propagation	50-25
Display Propagated Signal Labels	50-25
How Simulink Propagates Signal Labels	50-26
Signal Dimensions	50-34
About Signal Dimensions	50-34
Simulink Blocks that Support Multidimensional Signals ..	50-35
Determine Output Signal Dimensions	50-36
About Signal Dimensions	50-36
Determining the Output Dimensions of Source Blocks ...	50-36
Determining the Output Dimensions of Nonsource Blocks	50-37
Signal and Parameter Dimension Rules	50-37
Scalar Expansion of Inputs and Parameters	50-38
Display Signal Sources and Destinations	50-42
About Signal Highlighting	50-42
Highlighting Signal Sources	50-42
Highlighting Signal Destinations	50-43
Removing Highlighting	50-44
Resolving Incomplete Highlighting to Library Blocks	50-44
Signal Ranges	50-45
About Signal Ranges	50-45
Blocks That Allow Signal Range Specification	50-45
Specifying Ranges for Signals	50-46
Checking for Signal Range Errors	50-47
Initialize Signals and Discrete States	50-52
About Initialization	50-52
Using Block Parameters to Initialize Signals and Discrete States	50-53
Using Signal Objects to Initialize Signals and Discrete States	50-53

Using Signal Objects to Tune Initial Values	50-54
Example: Using a Signal Object to Initialize a Subsystem	
Output	50-56
Initialization Behavior Summary for Signal Objects	50-57
Test Points	50-59
What Is a Test Point?	50-59
Designating a Signal as a Test Point	50-59
Displaying Test Point Indicators	50-60
Display Signal Attributes	50-62
Ports & Signals Menu	50-62
Port Data Types	50-63
Design Ranges	50-63
Signal Dimensions	50-64
Signal to Object Resolution Indicator	50-65
Wide Nonscalar Lines	50-66
Display Port Numbers When Addressing Errors	50-67
Signal Groups	50-69
About Signal Groups	50-69
Signal Builder Window	50-69
Creating Signal Group Sets	50-83
Editing Waveforms	50-113
Signal Builder Time Range	50-119
Exporting Signal Group Data	50-120
Printing, Exporting, and Copying Waveforms	50-120
Simulating with Signal Groups	50-121
Simulation Options Dialog Box	50-122

Using Composite Signals

51

Composite Signals	51-3
What is a Composite Signal?	51-3
Techniques for Combining Signals	51-3

Buses	51-6
What is a Bus?	51-6
Types of Simulink Buses	51-7
Bus Objects	51-7
View Information about Buses	51-7
Virtual and Nonvirtual Buses	51-13
About Virtual and Nonvirtual Buses	51-13
Choosing Between Virtual and Nonvirtual Buses	51-14
Creating Nonvirtual Buses	51-15
Nonvirtual Bus Sample Times	51-16
Automatic Bus Conversion	51-16
Explicit Bus Conversion	51-17
Create and Access a Bus	51-18
Nest Buses	51-20
Circular Bus Definitions	51-21
Bus-Capable Blocks	51-22
Bus Objects	51-24
About Bus Objects	51-24
Bus Object Capabilities	51-25
Associating Bus Objects with Simulink Blocks	51-25
Bus Object API	51-27
Manage Bus Objects with the Bus Editor	51-28
Introduction	51-28
Open the Bus Editor	51-29
Display Bus Objects	51-30
Create Bus Objects	51-33
Create Bus Elements	51-36
Nest Bus Definitions	51-39
Change Bus Entities	51-42
Export Bus Objects	51-46
Import Bus Objects	51-48
Close the Bus Editor	51-48

Store and Load Bus Objects	51-50
MATLAB Code Files	51-50
MATLAB Data Files (MAT-Files)	51-51
Database or Other External Source Files	51-51
Map Bus Objects to Models	51-52
Use a Rigorous Naming Convention	51-52
Filter Displayed Bus Objects	51-53
Filter by Name	51-54
Filter by Relationship	51-55
Change Filtered Objects	51-57
Clear the Filter	51-58
Customize Bus Object Import and Export	51-59
Prerequisites for Customization	51-60
Writing a Bus Object Import Function	51-60
Writing a Bus Object Export Function	51-61
Registering Customizations	51-62
Changing Customizations	51-63
Connect Buses to Imports and Outputs	51-65
Connect Buses to Root Level Imports	51-65
Connect Buses to Root Level Outputs	51-65
Connect Buses to Nonvirtual Imports	51-65
Connect Buses to Model, Stateflow, and MATLAB Function Blocks	51-68
Connect Multi-Rate Buses to Referenced Models	51-68
Specify Initial Conditions for Bus Signals	51-69
Bus Signal Initialization	51-69
Create Initial Condition (IC) Structures	51-71
Three Ways to Initialize Bus Signals Using Block Parameters	51-76
Setting Diagnostics to Support Bus Signal Initialization ..	51-80
Combine Buses into an Array of Buses	51-81
What Is an Array of Buses?	51-81
Benefits of an Array of Buses	51-82
Array of Buses Limitations	51-83
Define an Array of Buses	51-85

See Also	51-86
Arrays of Buses in Models	51-87
Blocks That Support Arrays of Buses	51-87
Arrays of Buses with Bus-Related Blocks	51-88
Set Up a Model to Use Arrays of Buses	51-89
Set Diagnostic	51-92
Signal Line Style	51-93
Convert Models to Use Arrays of Buses	51-94
General Conversion Approach	51-94
Code Generation for Arrays of Buses	51-97
Bus Data Crossing Model Reference Boundaries	51-98
Buses and Libraries	51-99
Prevent Bus and Mux Mixtures	51-100
What Are Bus and Mux Signal Mixtures?	51-100
Why Avoid Mixing Bus and Mux Signals?	51-101
When to Configure a Model to Prevent Bus and Mux Mixtures?	51-102
Two Upgrade Procedures	51-103
Correct Mux Blocks That Create Bus Signals	51-104
Choose the Appropriate Procedure	51-104
Models Without Model Referencing	51-104
Models With Model Referencing	51-105
Address Compatibility Issues After Running Upgrade Advisor	51-106
Correct Buses Used as Muxes	51-109
Three Approaches	51-109
Use the Model Advisor	51-109
Explicitly Add Bus to Vector Blocks	51-110
Reorganize the Model	51-111
Bus to Vector Block Compatibility Issues	51-112
Buses in Generated Code	51-114

Working with Variable-Size Signals

52

Variable-Size Signal Basics	52-2
About Variable-Size Signals	52-2
Creating Variable-Size Signals	52-2
How Variable-Size Signals Propagate	52-3
Empty Signals	52-4
Subsystem Initialization of Variable-Size Signals	52-4
See Also	52-5
Simulink Models Using Variable-Size Signals	52-6
Variable-Size Signal Generation and Operations	52-6
Variable-Size Signal Length Adaptation	52-10
Mode-Dependent Variable-Size Signals	52-14
See Also	52-19
S-Functions Using Variable-Size Signals	52-21
Level-2 MATLAB S-Function with Variable-Size Signals	52-21
C S-Function with Variable-Size Signals	52-22
See Also	52-23
Simulink Block Support for Variable-Size Signals	52-24
Simulink Block Data Type Support	52-24
Conditionally Executed Subsystem Blocks	52-24
Switching Blocks	52-25
See Also	52-27
Variable-Size Signal Limitations	52-28
See Also	52-29

Customizing Simulink Environment and Printed Models

Customizing the Simulink User Interface

53

Add Items to Model Editor Menus	53-2
About Adding Items	53-2
Code for Adding Menu Items	53-2
Define Menu Items	53-4
Register Menu Customizations	53-10
Callback Info Object	53-11
Debugging Custom Menu Callbacks	53-11
Menu Tags	53-11
Disable and Hide Model Editor Menu Items	53-16
About Disabling and Hiding Model Editor Menu Items ...	53-16
Example: Disabling the New Model Command on the Simulink Editor's File Menu	53-16
Creating a Filter Function	53-16
Registering a Filter Function	53-17
Disable and Hide Dialog Box Controls	53-18
About Disabling and Hiding Controls	53-18
Disable a Button on a Dialog Box	53-19
Write Control Customization Callback Functions	53-20
Dialog Box Methods	53-20
Dialog Box and Widget IDs	53-21
Register Control Customization Callback Functions	53-22
Customize the Library Browser	53-24
Reorder Libraries	53-24
Disable and Hide Libraries	53-24
Customize the Library Browser Menu	53-25
Registering Customizations	53-27
About Registering User Interface Customizations	53-27
Customization Manager	53-27

Print Frames	54-2
What are Print Frames?	54-2
PrintFrame Editor	54-3
Single Use or Multiple Use Print Frames	54-5
Text and Variable Content	54-5
Create a Print Frame	54-6
Add Rows and Cells to Print Frames	54-7
Add and Remove Rows	54-7
Add and Remove Cells	54-7
Resize Rows and Cells	54-8
Add Content to Print Frame Cells	54-9
Types of Content	54-9
Add Content to Cells	54-9
Block Diagram	54-10
Variables	54-11
Text	54-12
Format Content in Cells	54-12
Print Using Print Frames	54-13

Running Models on Target Hardware

About Run on Target Hardware Feature

What is “Run on Target Hardware”?	55-2
Open Examples for This Support Package	55-4
Using the Help Browser	55-4
Using the Block Library	55-6
Using Support Package Installer	55-7

Support Packages and Support Package Installer	55-10
What Is a Support Package?	55-10
What Is Support Package Installer?	55-10
Install This Support Package on Other Computers . . .	55-12
Tune and Monitor Models Running on Target	
Hardware	55-14
Overview of Using External Mode	55-14
Run Your Simulink Model in External Mode	55-15
Stop External Mode	55-16
External Mode Control Panel	55-17
Troubleshoot Running Models on Target Hardware . .	55-19
Block Produces Zeros or Does Nothing in Simulation	55-19

Arduino Hardware

56

Install Support for Arduino Hardware	56-2
Open Block Libraries for Arduino Hardware	56-3
From the Command Line	56-3
From the Simulink Library Browser	56-4
Connect Arduino Ethernet Shield to Arduino	
Hardware	56-8
Configure Network Settings for Arduino Ethernet	
Shield	56-10
Run Model on Arduino Hardware	56-12
Stop or Restart a Model on the Arduino Hardware	56-13

Tune and Monitor Model Running on Arduino Mega	
2560 Hardware	56-14
About External Mode	56-14
Run Your Model in External Mode	56-15
Stop External Mode	56-17
Use Serial Communications with Arduino Hardware ..	56-18
Hardware	56-18
Transmit Serial Data	56-19
Receive Serial Data	56-20
Detect and Fix Task Overruns on Arduino Hardware ..	56-21
Troubleshoot Running Models on Arduino	
Hardware	56-23
“Could not automatically set host COM port”	56-23
“Invalid port name. COM port must start with”	56-24
Configure Host COM Port Manually	56-26
Configure COM Port on Windows	56-26
Configure COM Port on Mac OS X	56-28

BeagleBoard Hardware

57

Install Support for BeagleBoard Hardware	57-2
Replace Firmware on BeagleBoard Hardware	57-3
Choose the Type of Serial Cable	57-19
Connect to Serial Port on BeagleBoard Hardware	57-20
See Also	57-24
Configure Network Connection with BeagleBoard	
Hardware	57-25

Get IP Address of BeagleBoard Hardware	57-29
Open Block Library for BeagleBoard Hardware	57-31
Run Model on BeagleBoard Hardware	57-34
See Also	57-35
Tune and Monitor Model Running on BeagleBoard	
Hardware	57-36
About External Mode	57-36
Run Your Simulink Model in External Mode	57-37
Stop External Mode	57-38
Detect and Fix Task Overruns on BeagleBoard	
Hardware	57-40

Gumstix Overo Hardware

58

Install Support for Gumstix Overo Hardware	58-2
Replace Firmware on Gumstix Overo Hardware	58-3
Open a Serial Command-line Session with Gumstix	
Overo Hardware	58-20
See Also	58-24
Configure Network Settings on Gumstix Overo	
Hardware	58-25
Get IP Address of Gumstix Overo Hardware	58-29
Open Block Library for Gumstix Overo Hardware	58-31
Run Model on Gumstix Overo Hardware	58-33
See Also	58-34

Stop or Restart a Model Running on Gumstix Overo	
Hardware	58-35
Tune and Monitor Model Running on Gumstix Overo	
Hardware	58-36
Overview of Using External Mode	58-36
Run Your Simulink Model in External Mode	58-37
Stop External Mode	58-38
Detect and Fix Task Overruns on Gumstix Overo	
Hardware	58-39
Access the Linux Desktop Directly Using Desktop	
Computer Peripherals	58-40
Access the Linux Desktop Remotely Using VNC	58-41

LEGO MINDSTORMS NXT Hardware

59

Install Support for LEGO MINDSTORMS NXT	
Hardware	59-2
Replace Firmware on NXT Brick	59-3
Open Block Library for LEGO MINDSTORMS NXT	
Hardware	59-7
Run Model on NXT Brick	59-11
Tune Parameters and Monitor Data in a Model Running	
on NXT Brick	59-13
About External Mode	59-13
Run Your Simulink Model in External Mode	59-14
Stop External Mode	59-15
Set COM Port Number Manually	59-15

Detect and Fix Task Overruns on NXT Brick	59-19
Exchange Data Between Two NXT Bricks	59-20
Tips for Use Bluetooth Blocks	59-20
Add Bluetooth Blocks to your Models	59-20
Configure Bluetooth Slave	59-24
Configure Bluetooth Master	59-26
Set Up A Bluetooth Connection	59-29

PandaBoard Hardware

60

Install Support for PandaBoard Hardware	60-2
Replace Firmware on PandaBoard Hardware	60-3
Choose the Type of Serial Cable	60-18
Connect to Serial Port on PandaBoard Hardware	60-19
See Also	60-23
Configure Network Connection with PandaBoard Hardware	60-24
Get IP Address of PandaBoard Hardware	60-28
Open Block Library for PandaBoard Hardware	60-31
Run Model on PandaBoard Hardware	60-34
See Also	60-35
Tune and Monitor Model Running on PandaBoard Hardware	60-37
About External Mode	60-37
Run Your Simulink Model in External Mode	60-38

Stop External Mode	60-39
Detect and Fix Task Overruns on PandaBoard	
Hardware	60-40
Access the Linux Desktop Directly Using Desktop	
Computer Peripherals	60-41
Access the Linux Desktop Remotely Using VNC	60-43
Configure Wi-Fi on PandaBoard Hardware	60-46

Raspberry Pi Hardware

61

Install Support for Raspberry Pi Hardware	61-2
Replace Firmware on Raspberry Pi Hardware	61-3
Open a Command-Line Session with Raspberry Pi	
Hardware	61-16
See Also	61-17
Get IP Address of Raspberry Pi Hardware	61-18
Configure Raspberry Pi Hardware to Mail IP Address	
Changes	61-20
Configure Network Settings on Raspberry Pi	
Hardware	61-23
Open Block Library for Raspberry Pi Hardware	61-26
Run Model on Raspberry Pi Hardware	61-28
See Also	61-29

Stop or Restart a Model Running on Raspberry Pi	
Hardware	61-30
Tune and Monitor Model Running on Raspberry Pi	
Hardware	61-31
Overview of Using External Mode	61-31
Run Your Simulink Model in External Mode	61-32
Stop External Mode	61-33
Detect and Fix Task Overruns on Raspberry Pi	
Hardware	61-34
Access the Linux Desktop Using Computer	
Peripherals	61-35
Extend the Linux Partition on the SD Card	61-36
Shut Down Raspberry Pi Hardware	61-38

Index

Introduction to Simulink

- Chapter 1, “Simulink Basics”
- Chapter 2, “Simulation Stepping”
- Chapter 3, “How Simulink Works”

Simulink Basics

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

- “Start the Simulink Software” on page 1-3
- “Open a Model” on page 1-5
- “Load a Model” on page 1-8
- “Save a Model” on page 1-10
- “Simulink Editor” on page 1-19
- “Zoom and Pan Block Diagrams” on page 1-26
- “Preview Content of Hierarchical Items” on page 1-28
- “Update a Block Diagram” on page 1-32
- “Printing Capabilities” on page 1-34
- “Basic Printing” on page 1-36
- “Select the Systems to Print” on page 1-41
- “Specify the Page Layout and Print Job” on page 1-43
- “Tiled Printing” on page 1-44
- “Print Multiple Pages for Large Models” on page 1-45
- “Add a Log of Printed Models” on page 1-46
- “Add a Sample Time Legend” on page 1-47
- “Print from the MATLAB Command Line” on page 1-48
- “Export Models to Third-Party Applications” on page 1-52
- “Print to a PDF or Postscript File” on page 1-53

- “Export Models to Image File Formats” on page 1-54
- “Generate a Model Report” on page 1-55
- “End a Simulink Session” on page 1-58
- “Keyboard and Mouse Shortcuts for Simulink” on page 1-59
- “Simulink Demos Are Now Called Examples” on page 1-62


Start the Simulink Software

Open the MATLAB Software

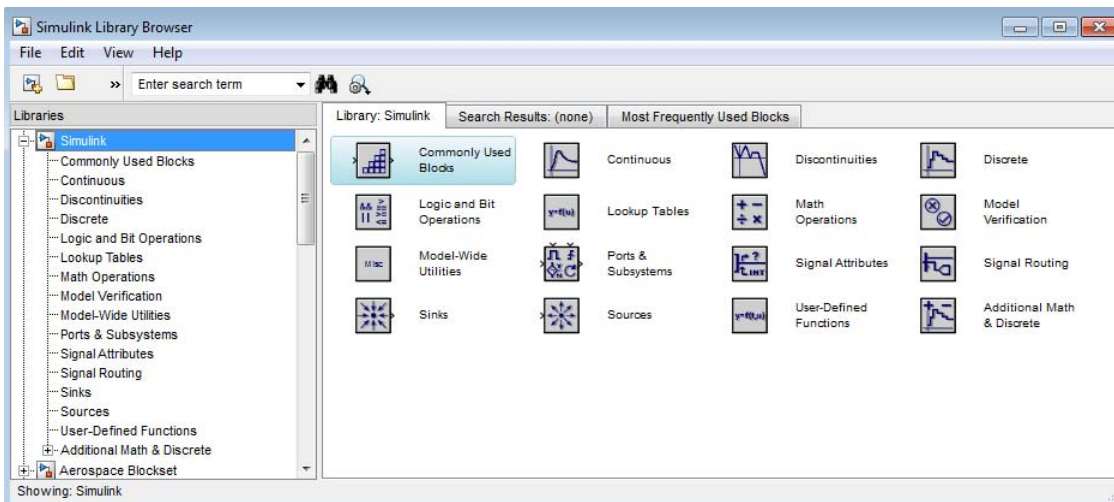
To start the Simulink software, first start the MATLAB® technical computing environment. For details about starting MATLAB, see “Startup and Shutdown”.

Open the Library Browser

To start Simulink Library Browser from MATLAB, use one of these approaches:

- On the MATLAB toolbar, click the Simulink button (.
- At the MATLAB prompt, enter the `simulink` command.

The Library Browser opens. It displays a tree-structured view of the Simulink block libraries installed on your system. The Simulink library window displays icons representing the pre-installed block libraries.



Note On computers running the Windows® operating system, you can display the Simulink library window by right-clicking the Simulink node in the Library Browser window.

To create models, copy blocks from the Library Browser into a model window in the Simulink Editor.

Open the Simulink Editor

Use *one* of the following approaches to open the Simulink Editor:

- Open an existing model. For example, at the MATLAB prompt, enter the name of a model. For details, see “Open an Existing Model” on page 1-5.
- In the Library Browser, select **File > Open**. Choose a model or enter the file name of the model.
- At the MATLAB prompt, use the `open_system` command. For example, to open a model named `vdp`:

```
open_system('vdp')
```

Note To have the Simulink Editor display properly, use 32-bit color mode.

Typically, remote desktop connections (for example, a VNC connection) use a default color mode of 16-bits or less.

Open a Model

In this section...

“What Happens When You Open a Model” on page 1-5

“Open an Existing Model” on page 1-5

“Models with Different Character Encodings” on page 1-6

“Avoid Initial Model Open Delay” on page 1-6

What Happens When You Open a Model

Opening a model brings the model into memory and displays the model graphically in the Simulink Editor.

You can also bring a model into memory without displaying it, as described in “Load a Model” on page 1-8.

Open an Existing Model

To open an existing model, use *one* of these approaches:

- (Windows operating systems only) On the Library Browser toolbar, click the **Open** button.
- In the Library Browser or the Simulink Editor, select **File > Open**. Choose a model or enter the file name of the model.
- At the MATLAB command prompt, enter the name of the model, without the file extension (e.g., `.slx`). The model must be in the current folder or on the path.

Note If you have an earlier version of the Simulink software, and you want to open a model that was created in a later version, first use the later version to save the model in a format compatible with the earlier version. Then open the model in the earlier version. For details, see “Export a Model to a Previous Simulink Version” on page 1-16.

Models with Different Character Encodings

If you open a model created in a MATLAB software session configured to support one character set encoding (for example, Shift_JIS), in a session configured to support another character encoding (for example, US_ASCII), Simulink displays a warning for SLX files. For MDL files, you might see a warning or an error message, depending on whether it can encode the model, using the current character encoding. The warning or error message specifies the encoding of the current session and the encoding used to create the model. To display the model's text correctly:

- 1 Close all models open in the current session.
- 2 Use the `slCharacterEncoding` command to change the character encoding of the current MATLAB software session to that of the model as specified in the warning message.
- 3 Reopen the model.

You can now edit and save the model.

Simulink can check if models contain characters unsupported in the current locale. For more details, see “Check file for foreign characters” and “Saving Models with Different Character Encodings” on page 1-15.

Avoid Initial Model Open Delay

The first model that you open in a MATLAB session takes longer to open than do subsequent models. This is because MATLAB does not load the Simulink product into memory until the first time that you open a Simulink model. This just-in-time loading of the Simulink product reduces the MATLAB startup time and avoids unnecessary consumption of system memory.

To avoid the initial model opening delay, you can have MATLAB load the Simulink software when the MATLAB product starts up. You can issue the command to load Simulink at MATLAB startup from one of two places:

- The `-r` command line option
- The MATLAB `startup.m` file

Use one of these commands:

- `load_simulink` — loads the Simulink product
- `simulink` — loads the Simulink product and opens the Simulink Library Browser

For example, to load the Simulink product when the MATLAB software starts up on a computer running the Microsoft® Windows operating system, create a desktop shortcut with the following target:

```
matlabroot\bin\win32\matlab.exe -r load_simulink
```

Similarly, the following command loads the Simulink software when the MATLAB software starts up on Macintosh and Linux® computers:

```
matlab -r load_simulink
```

Load a Model

Loading a model brings it into memory but does not display it graphically.

Tip To both bring a model into memory and display it graphically, open the model as described in “Open a Model” on page 1-5.

After you load a model (as distinct from opening it), you can work with the model programmatically as if it were visible. However, you cannot use the Simulink Editor to edit the model unless you open the model.

You cannot load a model using the Simulink Editor or Library Browser.

To load a model programmatically, at the MATLAB command prompt, enter the `load_system` command. Specify the model to be loaded. For example, to load the `vdp` model:

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

Load Variables When Loading a Model

You can use model callbacks to load variables and perform other actions when a model is loaded. For details, see “Callbacks for Customized Model Behavior” on page 4-56.

You can use the `PreloadFcn` callback to automatically preload variables into the MATLAB workspace when you open a model.

Parameters in different parts of the Simulink model might require some variables. For example, if you have a model that contains a Gain block and the gain is specified as `K`, Simulink looks for the variable `K` to be defined. You can automatically define `K` every time the model loads.

You can define variables, such as `K`, in a MATLAB script. You can use the `PreLoadFcn` callback to execute the MATLAB script.

To create model callbacks interactively, in the Simulink Editor, select **File > Model Properties > Model Properties** and use the **Callbacks** tab to edit callbacks.

To create a callback programmatically, at the MATLAB command prompt, enter the following :

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

where `expression` is a valid MATLAB command or a MATLAB script that exists in your MATLAB search path.

For example, if your model is called `modelName.slx` and your variables are defined in a MATLAB script called `loadvar.m`, you would type the following:

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

Now save the model. Every time you subsequently open this model, the `loadvar` function executes. You can see the variables from the `loadvar.m` declared in the MATLAB workspace.

For example, if you have a model that contains a Gain block with a gain specified as `K`, Simulink looks in the MATLAB base workspace for the variable `K` to be defined. You can use the `PreLoadFcn` callback, to automatically define `K` every time you open the model.

Assuming that you have several variables (such as `K`) predefined in a MATLAB file called `loadvar.m`, to be used in a model called `modelName.slx`, you can use the `PreLoadFcn` callback to execute the MATLAB file `loadvar.m`. For example, type the following at MATLAB command prompt:

```
set_param('modelName','PreLoadFcn','loadvar')
```

After saving the model, every subsequent time you open this model, the `loadvar` function executes. After opening `modelName.slx`, type:

```
whos
```

The variables from the `loadvar.m` file are in the MATLAB workspace.

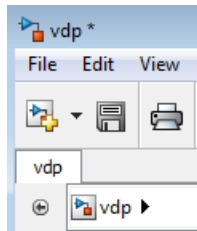
Save a Model

In this section...

- “How to Tell If a Model Needs Saving” on page 1-10
- “Save a Model for the First Time” on page 1-11
- “Model Names” on page 1-11
- “Save a Previously Saved Model” on page 1-11
- “What Happens When You Save a Model?” on page 1-11
- “Saving Models in the SLX File Format” on page 1-12
- “Saving Models with Different Character Encodings” on page 1-15
- “Export a Model to a Previous Simulink Version” on page 1-16
- “Save from One Earlier Simulink Version to Another” on page 1-17

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 the model needs saving, an asterisk appears next to the model name in the title bar.



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 for the First Time

To save a model for the first time, in the Simulink Editor, select **File > Save**. Provide a location and name for the model file.

Model Names

Model file names must start with a letter and can contain letters, numbers, and underscores. The file name must not be the same as that of a MATLAB software command.

The total number of characters 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.

Save a Previously Saved Model

If you are saving a model whose model file was previously saved:

- To *replace* the file contents, in the Simulink Editor, select **File > Save**.
- To save the model with a new name or location, in the Simulink Editor, select **File > Save As**.

Note See also “Upgrade Models to SLX” on page 1-12.

What Happens When You Save a Model?

Simulink saves the model by generating a specially formatted file called the *model file* that contains the model (block diagram) and block properties. The Simulink software follows this process while saving a model:

- 1** If the model file already exists, Simulink renames the model file as a temporary file.
- 2** All block `PreSaveFcn` callback routines execute first, then the model `PreSaveFcn` callback routine executes.
- 3** Simulink writes the model file to a new file using the same name and, by default, an extension of `.slx`.

4 All block PostSaveFcn callback routines execute, then the model

5 Simulink deletes the temporary file.

If an error occurs during this process, Simulink:

- Renames the temporary file to the name of the original model file
- Writes the current version of the model to a file with an .err extension
- Issues an error message

If an error occurs in Step 2 of the save process, then Step 3 is omitted and Steps 4 and 5 are performed.

Saving Models in the SLX File Format

Save New Models as SLX

You save 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.

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 (if writable) and Simulink does not remove the MDL file.

If you save an existing MDL file using **File > Save**, 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 Select **File > 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 “Upgrade Model Files to SLX and Preserve Revision History” on page 13-74.

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
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 13-91.
Changing character encodings.	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 1-16.

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`).

Saving Models with Different Character Encodings

- “MDL Files and Character Encodings” on page 1-15
- “SLX Files and Character Encodings” on page 1-16

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 “Saving Models in the SLX File Format” on page 1-12).
- Use the following procedure to 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 that you’ve 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, the software

displays a warning message 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 file 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, such as Simulink 7.0 (R2007b). For example, you might want to perform such an export to make a model available to 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, select **File > Save**. This saves a copy in the latest version of Simulink. This step avoids compatibility problems.
- 2** Simulink Editor, select **File > Export Model to > Previous Version**.

The Export to Previous Version dialog box appears.

- 3** In the dialog box, from the **Save as type** list, select the previous version to which to export the model.
- 4** Click the **Save** button.

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 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.

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 selecting **File > Save**.

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 by selecting **File > Export Model to > 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 by selecting **File > Save**.

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”

Simulink Editor

In this section...

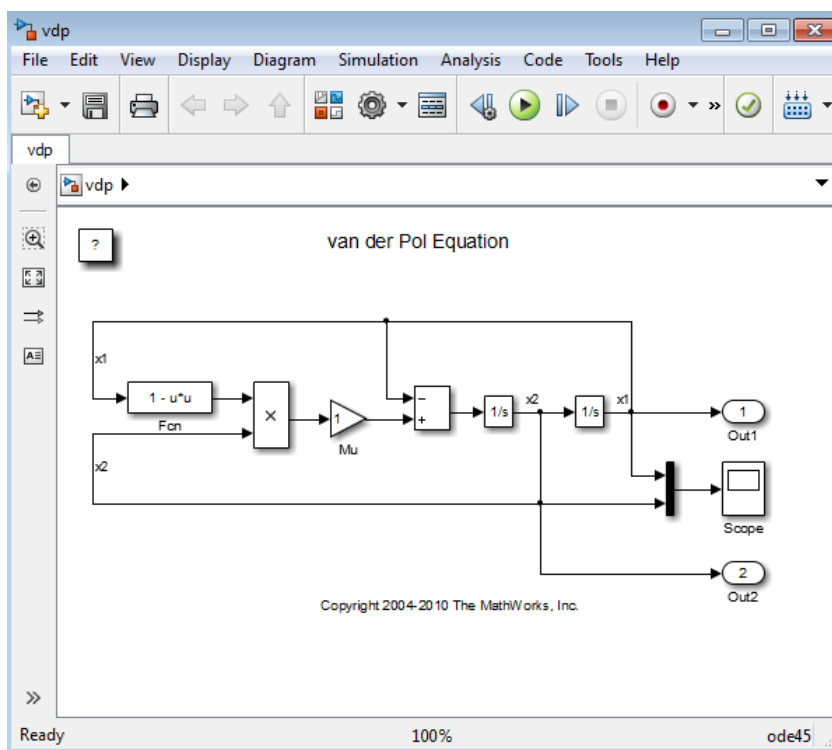
“Editor Layout” on page 1-19

“Undoing Commands” on page 1-23

“Window Management” on page 1-23

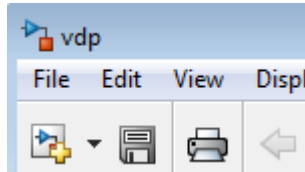
Editor Layout

Opening a Simulink model or library displays the model or library in the Simulink Editor. For more information, see “Open the Simulink Editor” on page 1-4.



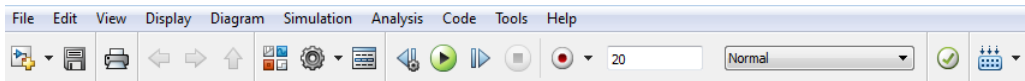
The Simulink Editor has the following major sections:

- **Title bar**



In the top left corner, the title bar displays the name of the model or subsystem that is open in the model window.

- **Menu bar and toolbar**



At the top of the Simulink Editor, you can access commands to work with models. Several buttons in the toolbar provide quick access to commonly used Simulink Editor menu options. Other buttons in the toolbar open other Simulink tools, such as the Model Explorer (for details, see “Open Simulink Tools from the Toolbar” on page 1-22).

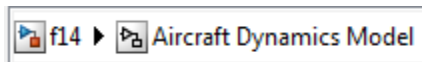
- **Palette**



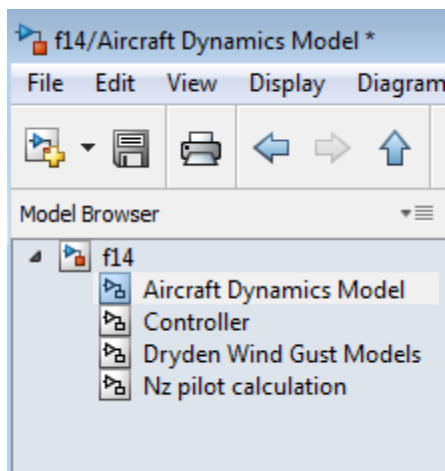
The icons in the vertical bar on the right side of the Simulink Editor perform very common tasks, such as adding an annotation.


- **Explorer bar**

The breadcrumb shows the systems that you have open in the editor. Select a system in the breadcrumb to open that system in the model window. If you click in the Explorer bar whitespace, you can edit the hierarchy. Also, the down arrow at the right side of the Explorer bar provides a history.



- **Model Browser**



Click the double arrows  in the bottom left corner of the Simulink Editor to open or close a tree-structured view of the model in the editor.

- **Canvas** — The canvas is the area where you edit the block diagram. You can use the mouse and keyboard to create and connect blocks, select and move blocks, edit block labels, display block dialog boxes, and so on.





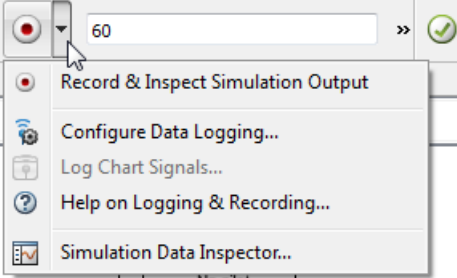
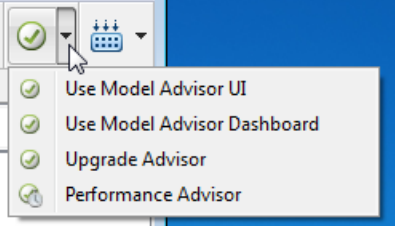
To display *context menus* specific to a particular model object in the canvas, such as a block, right-click the object.

- **Status information** — Near the top of the editor, you can see (and reset) the simulation time and the simulation mode. The bottom status line shows the status of Simulink processing, the zoom factor, and the solver.

Open Simulink Tools from the Toolbar

The Simulink Editor toolbar provides several buttons for quick access to other Simulink tools.

For buttons that have associated menu options, clicking the button invokes the first menu option.

Tools	Buttons and Associated Menus
Library Browser	
Model Explorer	
Simulation Stepper	Next Step  Previous Step 
Simulation Data Inspector	
Advisor Tools	

Undoing Commands

You can cancel the effects of up to 101 consecutive operations. To undo commands, select **Edit > Undo**. Repeat until you get to the command that you want to undo. You can undo operations such as:

- Adding, deleting, or moving a block
- Adding, deleting, or moving a line
- Adding, deleting, or moving a model annotation
- Editing a block name
- Creating a subsystem (see “Undoing Subsystem Creation” on page 4-40 for more information)

To reverse the effects of an **Undo** command, select **Edit > Redo**.

Window Management

One Simulink Editor Per Model

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

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

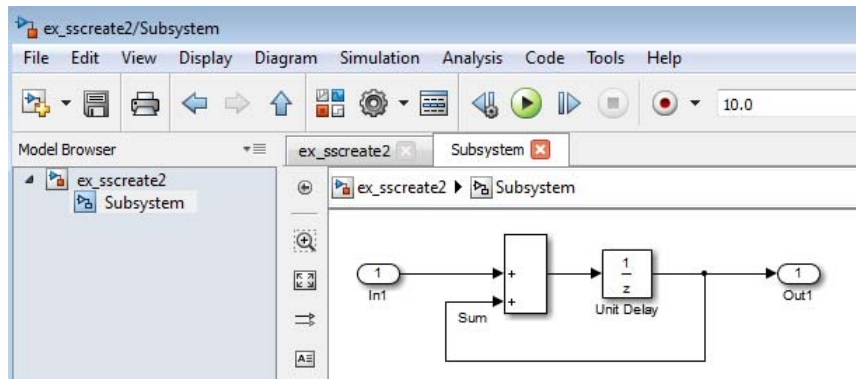
```
open_system ('vdp', 'window')
```

Open a Subsystem

The Simulink Editor displays only one active window at a time. By default, if you open a Subsystem block, the opened subsystem replaces (in terms of being displayed in the model window) the model from which you opened the subsystem.

You can use tabs in the Simulink Editor to make it easier to navigate between model windows for the top model and subsystems. To open a subsystem in a separate tab:

- 1 Right-click the Subsystem block.
- 2 From the context-menu, select **Open In New Tab**. In the example below, there are separate tabs for the `ex_sscreate2` model and for the Subsystem.



Navigate between the top model and subsystems by clicking the appropriate tab. Or, choose an option following options from the **View > Navigate** menu, such as **Back**, **Back to Parent**, or **Previous Tab**.

For more information about opening subsystems, see “Navigate Model Hierarchy” on page 4-40.

To rearrange the order of tabs within a Simulink Editor window:

- 1 Select the tab that you want to move.
- 2 Drag the tab to where you want it to appear.
- 3 Release the mouse button.

Open a Referenced Model


If you open a Model block, the referenced model opens in a separate Simulink Editor.

Bring the MATLAB Desktop Forward

Simulink Editor windows open on top of the MATLAB desktop. To bring the MATLAB desktop back to the top of your screen, in the Simulink Editor, select **View > MATLAB Desktop**.

Zoom and Pan Block Diagrams

You can enlarge or shrink the view of the block diagram in the current Simulink Editor window.

Goal of Zoom or Pan	What You Do
Enlarge the model diagram incrementally	Select View > Zoom > Zoom In or press Ctrl++ .
Enlarge the model diagram dynamically	Roll the mouse scroll wheel forward.
Zoom by a factor of two, displaying the selected model object	<ol style="list-style-type: none"> 1 In the palette, drag the zoom button () to the object that you want to include in the enlarged view. 2 Press the left mouse button.
Zoom until the whole model diagram just fits in the current window	Select View > Zoom > Fit to View or press Spacebar .
Zoom and pan	<ol style="list-style-type: none"> 1 Roll the mouse scroll wheel forward. 2 Release the scroll wheel. 3 Hold the scroll wheel down and pan to the desired positioning by moving the mouse. 4 Release the scroll wheel. <p>To disable the zoom and pan behavior for the scroll wheel, clear the File > Simulink Preferences > Editor Defaults > Scroll wheel controls zooming preference.</p>
Pan with the mouse	Hold down Spacebar and drag mouse.

Goal of Zoom or Pan	What You Do
Pan incrementally with arrow keys	If scroll bars are visible, then with nothing selected, use Shift+Left Arrow (or the right, up, or down arrow key) or for finer panning, just the arrow key.
Shrink the model diagram incrementally	Select View > Zoom > Zoom Out .
Shrink the model diagram dynamically	Roll the mouse scroll wheel backward.
Restore the model diagram to 100%	Select View > Zoom > Normal View (100%) or press Alt+1 .

To disable the zoom and pan behavior for the scroll wheel, clear the **File > Simulink Preferences > Editor Defaults > Scroll wheel controls zooming** preference. If you hold the **Ctrl** key and use the scroll wheel, the scroll wheel behavior is the opposite of how the preference is set.

Preview Content of Hierarchical Items

In this section...
“What Is Content Preview?” on page 1-28
“Enable Content Preview” on page 1-29
“What Content Preview Displays” on page 1-30

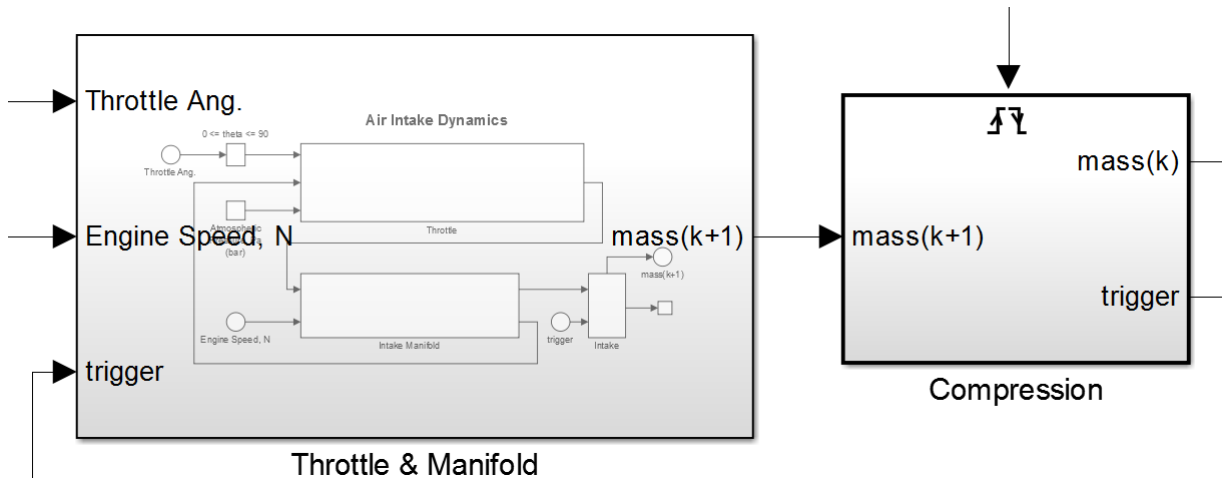
What Is Content Preview?

Content preview displays a representation of the contents of a hierarchical item, such as a subsystem, on the block. Content preview helps you to understand at a glance the kind of processing performed by the hierarchical item.

Hierarchical items that support content preview are:

- Subsystem blocks (but not masked subsystems)
- Model blocks (for referenced models)
- Stateflow® charts, subcharts, and graphical functions

By default, the Simulink Editor displays models without content preview. Enable or disable content preview for individual items. For example, in the following model, the Throttle & Manifold subsystem has content preview enabled, and the Compression subsystem does not.



Content preview displays a representation of the contents of a hierarchical item. For details, see “What Content Preview Displays” on page 1-30.

A slight delay can occur in drawing models that contain many hierarchical items that enable content preview if those items contain many blocks.

Enable Content Preview

Content preview settings apply across Simulink sessions.

Enable for Individual Hierarchical Items

- 1 In the Simulink or Stateflow Editor, select one or more Subsystem or Model blocks or charts.
- 2 Right-click and select **Format > Content Preview**.

Enable for the Currently Displayed System

- 1 In the Simulink Editor, select **Edit > Select All**.
- 2 Select **Diagram > Format > Content Preview**.

Enable for Model Blocks

Enabling content preview for Model blocks requires opening the referenced model.

- 1 In the Simulink Editor, right-click the Model block.
- 2 Select **Format > Content Preview**.
- 3 Open the Model block.

Note To enable content preview for multiple instances of the same referenced model, enable content preview for each Model block that references an instance.

Scope of Content Preview Settings

Content preview settings apply across Simulink sessions.

What Content Preview Displays

Simulink scales the content preview to fit the size of the block. To improve the readability of the content preview, you can:

- Zoom the hierarchical block.
- Resize the hierarchical block icon.

As you edit a model and apply updates to the appearance of the model (for example, move blocks or change background color), the content preview reflects those changes.

In addition to block icons and signals, content preview displays the following, if present in the system:

- Block labels
- Signal labels
- Highlighted blocks for signals with **Highlight Signal to Source** or **Highlight Signal to Destination** enabled

- Sample time color coding
- Stateflow animation

Simulink does not display content preview for:

- Masked blocks
- Hierarchical block icons when they are smaller than their default size in the Library Browser
- Subsystem blocks when they have the **Read/Write permissions** block parameter set to `NoReadOrWrite`.
- Protected Model blocks
- Model blocks whose referenced models are not loaded
- Models that have **Simulink Preferences > Editor Defaults > Use classic diagram theme** enabled.

The content preview image Simulink does not show:

- Block icon graphics, including masked block images
- The content of other hierarchical items contained in the content preview
- Signal line styles (for example, for buses)
- Port information such as port values

Update a Block Diagram

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 dialog box.

Simulation Updates the Diagram

Simulink updates the block 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 at Edit Time

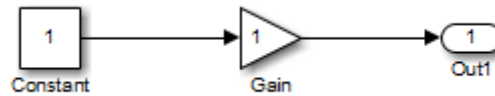
As you create a model, at any point you can have Simulink update the diagram. Updating the diagram periodically can help you to identify and fix potential simulation issues as you develop the model. This approach can make it easier to identify the sources of problems by focusing on a set of recent changes.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

To update the diagram at edit time, use one of these approaches:

- In the Simulink Editor, select **Simulation > Update Diagram**.
- Press **Ctrl+D**.

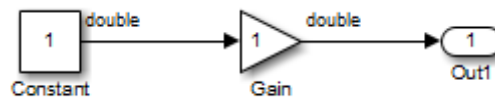
For example:

- 1** Create the following model.



- 2** In the Simulink Editor, select **Display > Signals & Ports > Port 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 parameters dialog box, set **Output data type** to `single`.

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

- 4** In the Simulink Editor, select **Simulation > Update Diagram**.

The updated block diagram reflects the change that you made previously.



In this example, Simulink infers a data type for the output of the Gain block. This is because you did not specify a data type for the block. The inferred data type inferred is `single`, because single precision is all that is necessary to simulate the model accurately, given that the precision of the block input is `single`.

Printing Capabilities

In this section...

“Print Interactively or Programmatically” on page 1-34

“Printing Options” on page 1-34

“Canvas Color” on page 1-34

“Print Model Reports” on page 1-35

Print Interactively or Programmatically

You can print a block diagram:

- Interactively in the Simulink Editor, by selecting **File > Print**
- Programmatically, by using 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 1-41.
- “Specify the Page Layout and Print Job” on page 1-43
- “Print Multiple Pages for Large Models” on page 1-45
- “Print Using Print Frames” on page 54-13
- “Add a Log of Printed Models” on page 1-46
- “Add a Sample Time Legend” on page 1-47

Canvas Color

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

Print Model Reports

In the Simulink Editor, you can generate a model report, which 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. To print the report, click **File > Print > Print Details** and specify report options. For more information, see “Generate a Model Report” on page 1-55.

If you have the Simulink Report Generator™ installed, you can generate a detailed report about a system and print it. To do so, in the Simulink Editor, select **File > Reports > System Design Description**. For more information, see “System Design Description”.

Basic Printing

In this section...

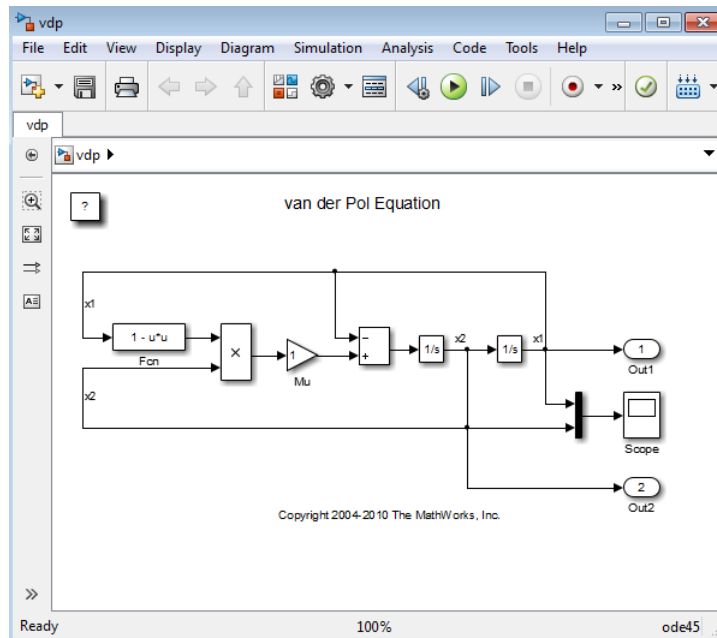
“Print the vdp Model Using Default Settings” on page 1-36

“Print a Subsystem Hierarchy” on page 1-38

Print the vdp Model Using Default Settings

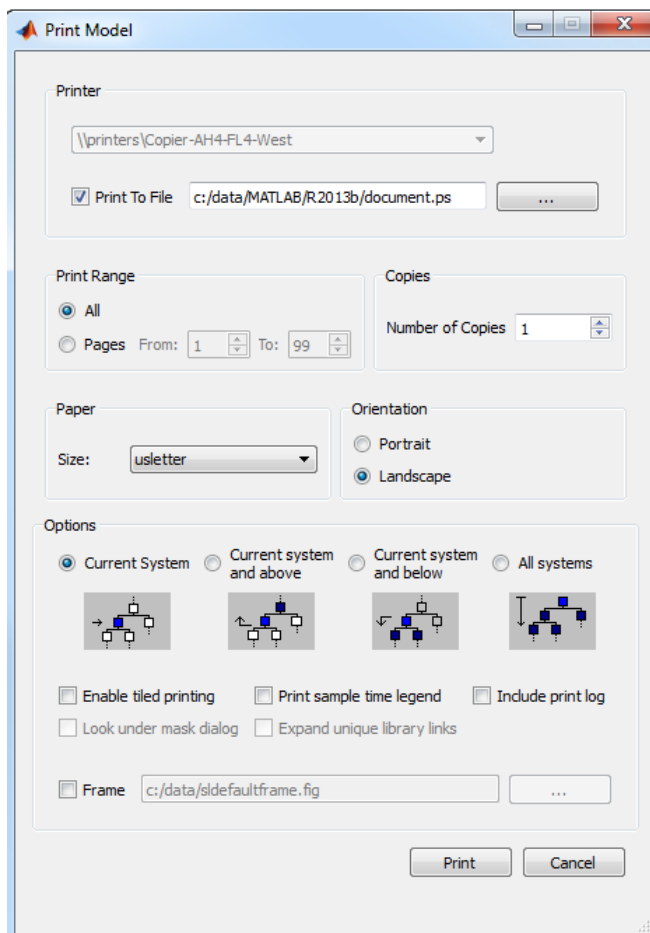
The default print settings produce good quality printed output for quickly capturing a model in printed form.

1 Open the vdp model.

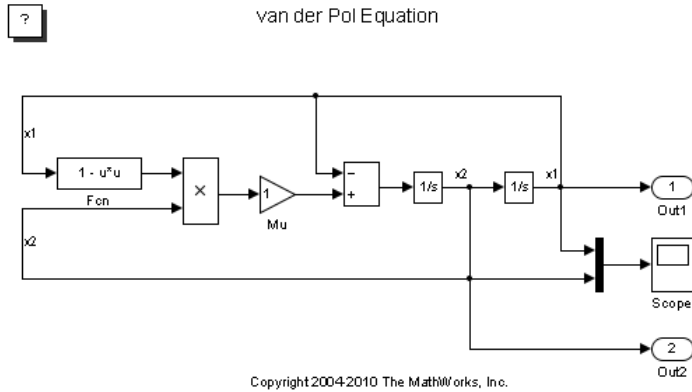


2 In the Simulink Editor, select **File > Print > Print**.

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.

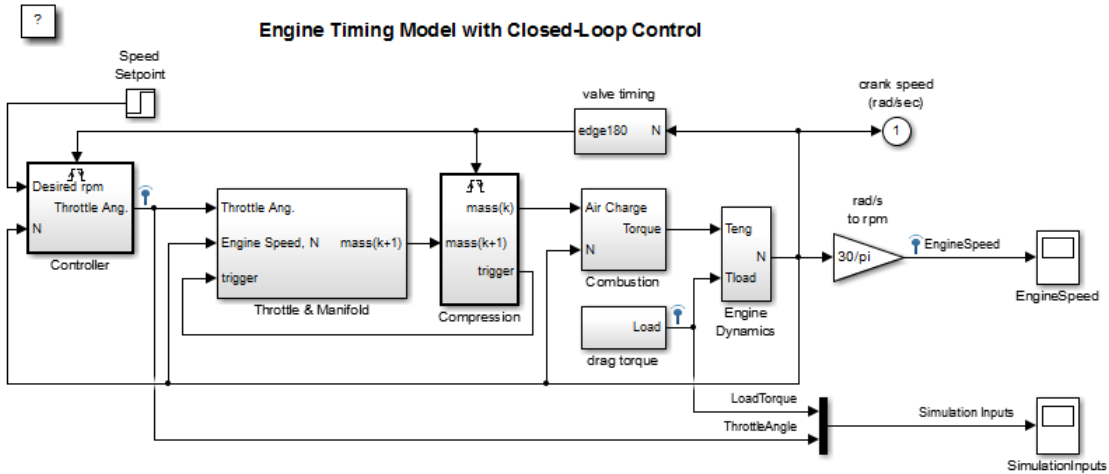


Print a Subsystem Hierarchy

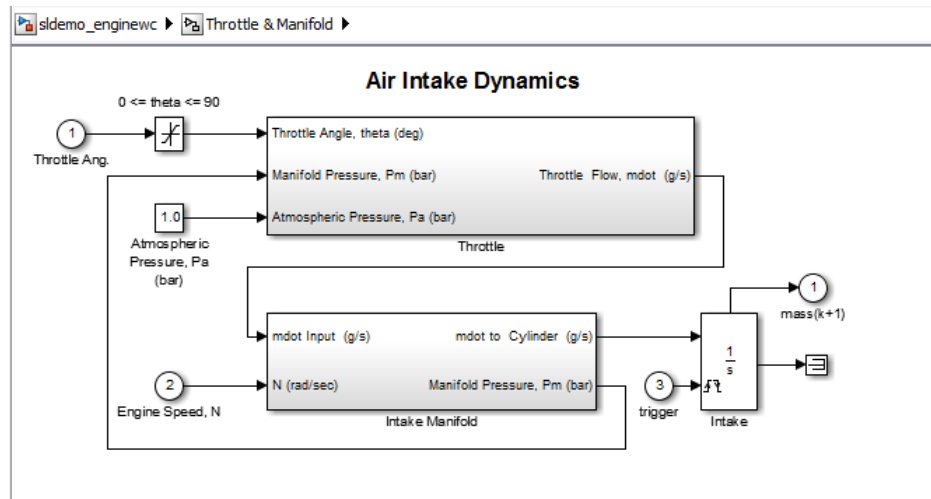
You can print levels in nested subsystems.

- 1 Open the sldemo_enginewc model.

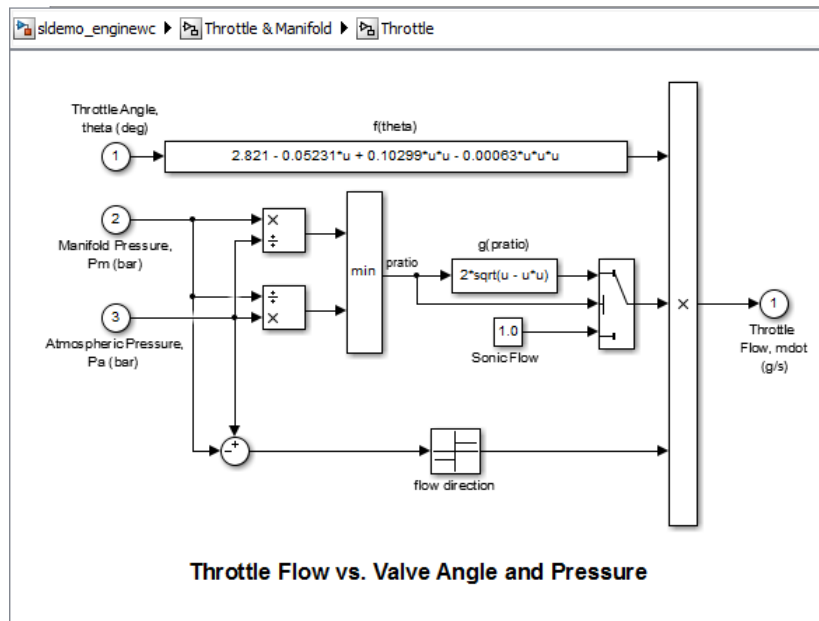
sldemo_enginewc ▶



- 2 Open the Throttle & Manifold subsystem.



3 Open the Throttle subsystem.



4 In the Simulink Editor, select **File > Print > Print**.

5 In the Print Model dialog box, click **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.

For details, see “Print Subsystems” on page 1-41.

Select the Systems to Print

In this section...
“Print Current System” on page 1-41
“Print Subsystems” on page 1-41
“Print a Model Referencing Hierarchy” on page 1-42

Print Current System

To select a specific system in a model to print, display that system in the currently open Simulink Editor tab and select **File > Print > 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 1-42.

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, select **File > Print > 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.

Specify the Page Layout and Print Job

In this section...
“Page and Print Job Setup” on page 1-43
“Two Interfaces for Page and Print Job Setup” on page 1-43

Page and Print Job Setup

Use the Print Model dialog box to specify these page layout and print job options. These settings apply only for this specific printing session:

- Page orientation (portrait or landscape)
- Paper size
- Range of pages to print
- Number of copies to print

Two Interfaces for Page and Print Job Setup

In general, to set up the page for printing a model, you can use the Print Model dialog box.

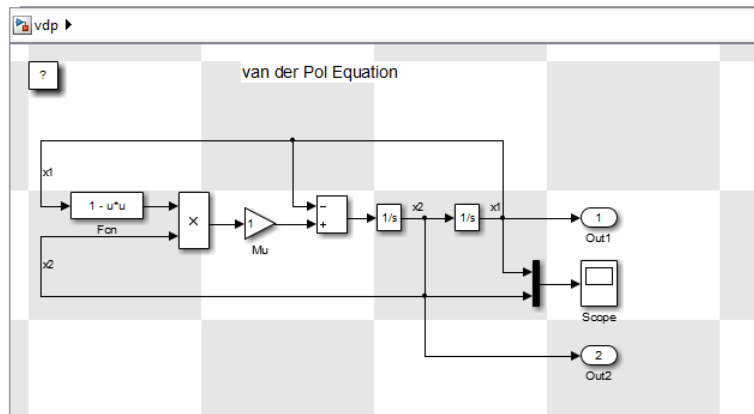
However, on Microsoft Windows platforms, you can also use the Print Setup dialog box (**File > Print > Printer Setup**). The settings in the Print Setup dialog box persist across Simulink sessions.

Settings for paper size and orientation in the Print Setup dialog box override the corresponding Print Model dialog box settings.

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 below, 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.

For details, see “Print Multiple Pages for Large Models” on page 1-45

Print Multiple Pages for Large Models

- 1 In the Simulink Editor, open the model in the current tab.
- 2 Select **File > Print > Print**.
- 3 In the Print Model dialog box, click **Enable tile printing**.

The default Enable tile printing setting in the Print Model dialog box is the same as the **File > Print > Enable Tile Printing** setting. If you change the Print Model dialog box **Enable tile 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, select **File > 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 1-50.
- 6 Optionally, specify a subset of pages to print. In the Print Model dialog box, specify the **Page Range**.
- 7 Click **Print**.

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 for which you want a log.
- 2 Select **File > Print > Print**.
- 3 In the Print Model dialog box, click **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 **Include all systems** enabled and **Tiled printing** cleared.

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





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, select **Simulation > Update diagram**.
- 2** Select **File > Print > Print**.
- 3** In the Print Model dialog box, click **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 **Include all systems** enabled.

Sample Times for 'sldemo_enginewc'

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

For more information about sample time legends, see “View Sample Time Information” on page 5-9.

Print from the MATLAB Command Line

In this section...

“Printing Commands” on page 1-48

“Print Systems with Multiline Names or Names with Spaces” on page 1-48

“Set Paper Orientation and Type” on page 1-49

“Position and Size a System” on page 1-49

“Use Tiled Printing” on page 1-50

Printing Commands

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

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

Tip When you use the `print` command, you can print only one specific 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 1-41.

You can use `set_param` to specify printing options for models. For details, see “Model Parameters”.

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 string. 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 with `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 is 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]')
```

Export Models to Third-Party Applications

On Microsoft Windows platforms, you can copy a model in either bitmap or metafile format. You can then paste the clipboard model to an application that accepts figures in bitmap or metafile format. For a description of how to set up the figure settings and save a figure to the clipboard on a Windows platform, see “Exporting to the Windows or Macintosh Clipboard”.

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** Copy a Simulink product model to the Windows operating system clipboard.
- 2** Paste the model from the clipboard to a third-party application, such as word processing software.

Print to a PDF or Postscript File

You can print a model to a .pdf or .ps (Postscript) file.

- 1** In the Simulink Editor, select **File > Print > Print**.
- 2** Specify or navigate to the existing .pdf or .ps file that you want to print to.
- 3** Click **Print**.

If you print directly to a Postscript file, Simulink creates a separate file for each system it prints.

If you print to a PDF file directly or through a print driver such as doPDF or Adobe PDF Writer, Simulink creates one file for all of the systems.

Export Models to Image File Formats

To export a model to another file format, such as .png or .jpeg, use the `-device` argument of 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.

Generate a Model Report

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, select **File > Reports > System Design Description**. For more information, see “System Design Description”.

To generate a model report for the current model:

1 In the Simulink Editor, select **File > Print > Print Details**.

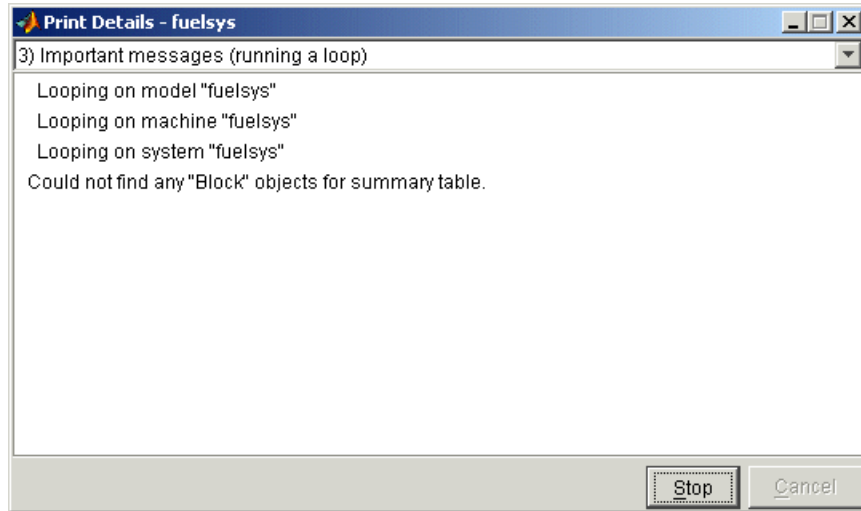
The Print Details dialog box appears.

2 Select the desired report options. For details, see “Model Report Options” on page 1-56.

3 Select **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 terminate the report generation, press **Stop**. When the report generation process finishes, the **Stop** button changes to an **Options** button. Clicking this button redisplay 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.

End a Simulink Session

To terminate a Simulink software session, close all Simulink windows.

To terminate a MATLAB software session, in the Simulink Editor, select **File > Exit MATLAB**.

Keyboard and Mouse Shortcuts for Simulink

In this section...
“Model Viewing Shortcuts” on page 1-59
“Block Editing Shortcuts” on page 1-59
“Line Editing Shortcuts” on page 1-61
“Signal Label Editing Shortcuts” on page 1-61
“Annotation Editing Shortcuts” on page 1-61

Note On Macintosh platforms, use the **command** key instead of **Ctrl**.

Model Viewing Shortcuts

The following table lists keyboard shortcuts for viewing models.

Task	Shortcut
Zoom in	Ctrl++
Zoom out	Ctrl+-
Zoom to normal (100%)	Alt+1
Pan	Hold the mouse scroll wheel down and drag the mouse
Fit diagram to screen	Spacebar
Pan with mouse	Hold down Spacebar and drag mouse
Delete selection	Delete or Backspace
Move selection	Make selection and use arrow keys

Block Editing Shortcuts

The following table lists mouse and keyboard actions that apply to blocks.

Task	Shortcut
Select one block	Left mouse button (LMB)
Select multiple blocks	Shift + LMB
Copy block from another window	Drag block
Move block	Drag block
Resize block, keeping same ratio of width and height	Select block, press Shift , and resize block
Resize block from the center	Select block, press Ctrl , and resize block
Rotate block clockwise	Ctrl+R
Rotate block counterclockwise	Ctrl+Shift+R
Flip block	Ctrl+I
Duplicate block	Ctrl+C , then Ctrl+V
Connect blocks	Select output port, hover over input port, and press Ctrl + LMB
Disconnect block	Shift + drag block
Create subsystem from selected blocks	Ctrl+G
Open selected subsystem	Enter
Go to parent of selected subsystem	Esc
Look under a block mask	Ctrl + U
Comment out a block	Select block and press Ctrl+Shift+X
Uncomment a block	Select commented block and press Ctrl+Shift+X
Apply block name edit	Esc

Line Editing Shortcuts

The following table lists mouse and keyboard actions that apply to lines.

Task	Shortcut
Select one line	Left mouse button (LMB)
Select multiple lines	Shift + LMB
Draw branch line	Ctrl + drag line; or RMB and drag line
Route lines around blocks	Shift + draw line segments
Move line segment	Drag segment
Move vertex	Drag vertex

Signal Label Editing Shortcuts

The next table lists mouse and keyboard actions that apply to signal labels.

Action	Shortcut
Create signal label	Double-click line, then enter label
Copy signal label	Ctrl + drag label
Move signal label	LMB + drag label
Edit signal label	Click in label, then edit

Annotation Editing Shortcuts

The next table lists mouse and keyboard actions that apply to annotations.


Action	Shortcut
Create annotation	Double-click in diagram, then enter text
Copy annotation	Ctrl+C then Ctrl+V
Move annotation	Drag annotation
Edit annotation	Click in text, then edit

Simulink Demos Are Now Called Examples

Starting in R2012b, Simulink models and videos that were previously called *demos* are now called *examples*. There are two ways to access these examples from the Help browser:

- At the top of the product landing page, click **Examples**.



- On any documentation page, click the Table of Contents button , and then select **Examples**.



You can filter documentation search results to display only examples.

The screenshot shows the MATLAB Help browser interface. At the top, a search bar contains the text 'fft' and a magnifying glass icon. Below the search bar, the text 'Results 1 through 10 of 12' is displayed. On the left side, there are three filter panels: 'Refine by Product' (showing 'MATLAB'), 'Refine by Category' (showing 'Graphics', 'Mathematics', 'Data and File Management', 'Advanced Software Development', and 'Language Fundamentals'), and 'Refine by Type' (showing 'Examples and How To' with a lightbulb icon). The 'Refine by Type' panel is circled in orange, and a mouse cursor is pointing at the 'Examples and How To' option. The search results on the right are as follows:

- Using FFT**
This example shows how to use the FFT function to analyze the variations in sunspot activity over the last 300 years.
[MATLAB > MATLAB Examples](#)
- FFT for Spectral Analysis**
This example shows the use of the FFT function for spectral analysis.
[MATLAB > MATLAB Examples](#)
- Fast Fourier Transform (FFT)**
Efficient computation of the DFT
[MATLAB > Mathematics > Fourier Analysis and Filtering](#)
- Discrete Fourier Transform (DFT)**
Introduction to the DFT
[MATLAB > Mathematics > Fourier Analysis and Filtering](#)

For more information about changes to the Help browser, see the R2012b MATLAB Release Notes.

Simulation Stepping

- “Simulation Stepping” on page 2-2
- “Step Through a Simulation” on page 2-14
- “Set Conditional Breakpoints for Stepping a Simulation” on page 2-17

Simulation Stepping

In this section...

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

“Simulation Stepper Interface” on page 2-5

“Simulation Stepper Limitations” on page 2-11

How Stepping Through a Simulation Works

The Simulation Stepper lets you step through the major time steps of a simulation. Access the Simulation Stepper from the Simulink Editor toolbar or from the **Simulation** menu:



The stepping capabilities enable you to analyze your model in various ways. For example, you can analyze plotted data at a particular moment in simulation time. (Stepping does not modify the model or change the course of the simulation.) You can:

- Play a complete simulation of Simulink models
- Start and stop a simulation
- Step forward and back through a simulation
- Change from running to paused (**Pause** button) or step back (if backward stepping is enabled).
- Set conditions to pause a simulation, before and during simulation.
- Add breakpoints at simulation time

You can use the stepping capabilities to perform actions such as analyze plotted data at a particular moment in simulation time.

To step back through a model, the software stores simulation states, or snapshots. A snapshot captures all the information required to continue a

simulation from that point in the simulation. A simulation snapshot contains simulation state (SimState) and information related to logged data and visualization blocks. For more information on snapshots, see “Simulation Snapshots” on page 2-7.

In conjunction with enabling the step back capability, you must also simulate the model or step it forward to save snapshots for the step back capability.

Tip 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.

The Simulink software stores simulation state when it steps forward through a simulation. Then, when you step back, the software uses the simulation snapshot, stored as SimState, to display the previous state of the simulation. (The model does not simulate in reverse when stepping back.)

How Simulation Stepper Differs from Simulink Debugger

The Simulation Stepper and the Simulink Debugger both enable you to start, stop, and step through a model simulation. Both allow you to use breakpoints as part of a debugging session. However, Simulation Stepper and Simulink Debugger are different in their intended use. Simulation Stepper lets you see all the data between major steps.

Use Simulink Debugger for advanced debugging scenarios that involve custom blocks (such as S-functions). The Simulink Debugger can help you analyze the issues with a particular block method. Simulink Debugger also gives you finer control on breakpoints. It also gives you finer control to step from one method to another block method within a time step. For more information, see “Introduction to the Debugger” on page 21-2.






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.		✓



Action	Simulation Stepper	Simulink Debugger
Step through a simulation block by block.		✓
Access via a command-line interface.		✓

To better understand the difference between Simulation Stepper and Simulink Debugger, you need to understand the simulation process. (For an overview of model simulation stages, see “Simulating Dynamic Systems” on page 3-18.)


Simulation Stepper Interface

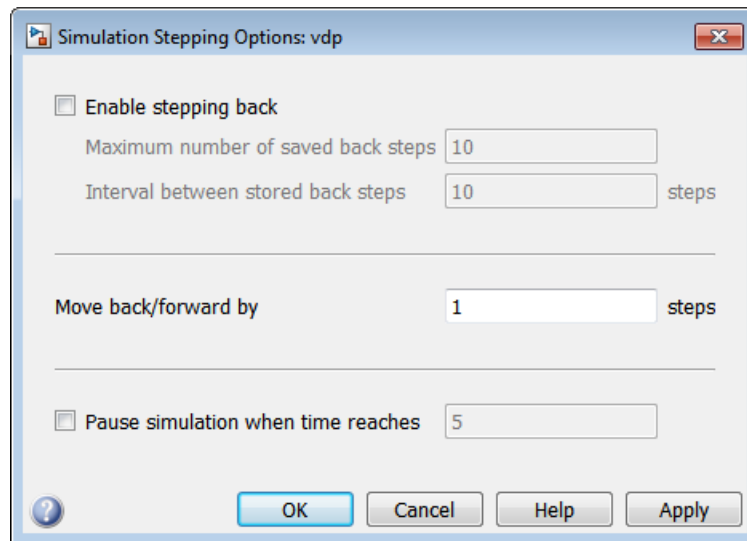
Simulation Stepper Toolbar Buttons

Button	Action
	Open the Simulation > Simulation Stepping Options dialog box to configure stepping options, such as pausing after a specific simulation time and enabling stepping back. This button is not available after you enable stepping back.
	Step the simulation back to a previously captured simulation state. Clicking this button to the final step of the simulation ends the simulation.
	Stepping back is not available because there are no saved states to step back to.
	Start/continue simulation.
	Pause simulation.


Button	Action
	Step the simulation forward.
	Stop the simulation.

Simulation Stepping Options Dialog Box

Click the Stepping Options button  or **Simulation > Stepping Options** to open the Simulation Stepping Options dialog box.



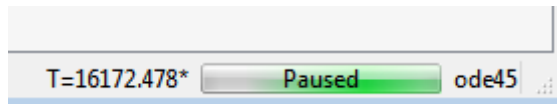
You use this dialog box to enable stepping back through a simulation. When stepping back is enabled, after you start the simulation, the button changes

to , which you use to step back. For more information, see “Simulation Stepping Options”.

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 is not updated 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.



When you use Simulation Stepper to step through a simulation, the time on the status bar updates at every major time step of the simulation. The value that is displayed (the time of the last completed step) is not always the same as the time of the solver. This event occurs because of the difference in the way that different solvers propagate the simulation time in a single iteration of the simulation loop. Simulation Stepper pauses at a single position within the simulation loop. However, some solvers perform their time advance before the stepper pauses, and others perform their time advance after the stepper pauses, which 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).

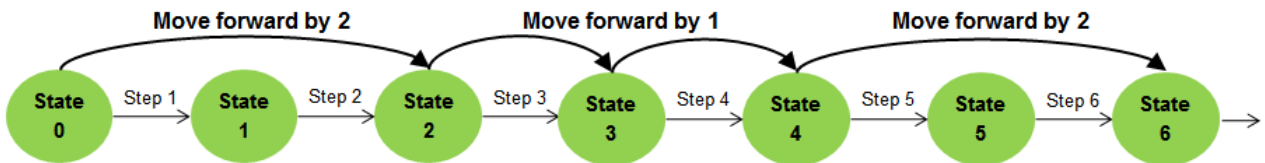
Simulation Snapshots

When you set up simulation stepping, you specify:

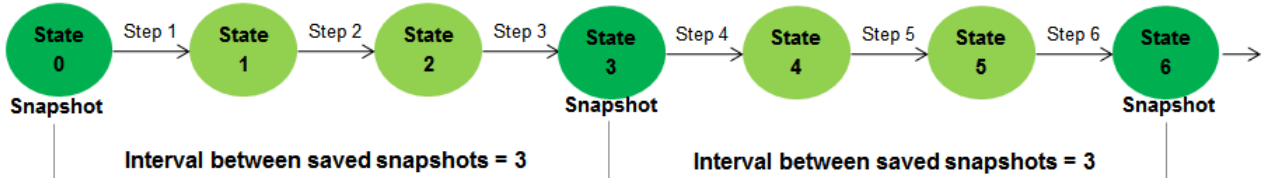
- The maximum number of steps, or snapshots, to capture while simulating forward. The greater the number, the more memory the simulation occupies and the longer the simulation takes to run.
- The number of steps to skip between snapshots. This parameter enables you to save snapshots of simulation state for stepping backward at periodic intervals, such as every three steps forward. This interval is independent

of the number of steps taken in either the forward or backward direction. Because taking simulation snapshots affects simulation speed, saving snapshots less often can improve simulation speed.

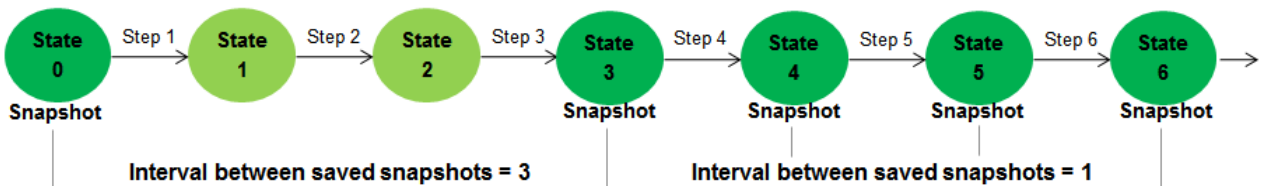
This figure shows how you can step through the simulation using the parameters in the Simulation Stepping Options dialog box. Because you can change the stepping parameters as you step through the simulation, you can step through a simulation as shown in this figure: sometimes by single steps, sometimes by two or more steps.



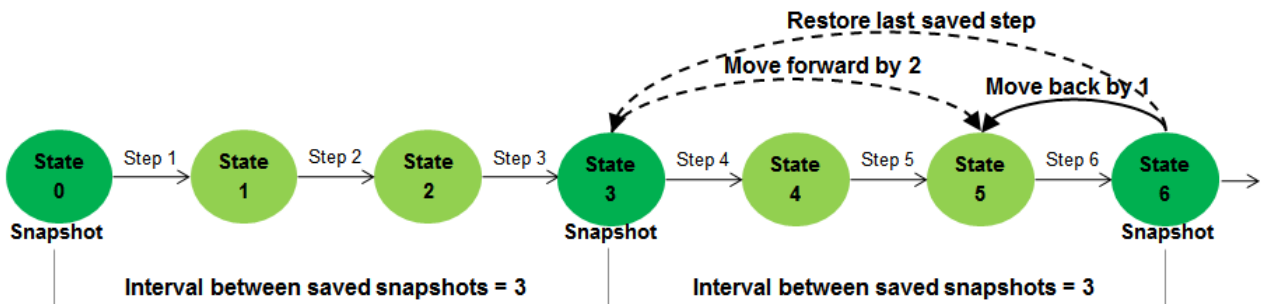
This figure shows snapshot captures in which the interval for stored snapshots is three.



This example shows the flexibility of the stepper, which allows you to change the stepping options while stepping forward. At the fourth step, the interval between stored back steps changed the snapshot steps from 3 to 1. This capability is useful when you want to capture more snapshots around a simulation time of interest.



This figure shows how the snapshots settings of the stepper can change what happens when stepping back. Suppose that the interval between snapshots is set to 3, and starting at state 6, the stepper **Move back/forward by** setting is set to 1. The stepper first restores the simulation state to the last saved snapshot (state 3), and then simulates two major times steps to arrive at the desired state (5). This capability is helpful for memory usage and simulation performance.



Stepping to the final step ends the simulation. You cannot, then, step back after reaching the end of simulation. You must restart the simulation to step backward.

Tune Parameters

While using the stepper, when the simulation is paused, you can change tunable parameters, including some solver settings.

Although you can change tunable parameters when you have paused a simulation, 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.

The 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, the 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. Values for tunable parameters are applied 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. For example, if you double-click the Manual Switch block, you change its input direction and thus affect simulation results.

For example, assume a snapshot save frequency of 3 and a step size of 1. 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 the stepper and Model block, the referenced model shares the stepping options of the top model throughout a simulation. As a result, changing the stepper settings for the referenced model during simulation changes the stepper settings of the top model. When the simulation ends, the stepper 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 change back to the settings before simulation started; the top model keeps the new values set during simulation.

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 Editor toolbar, see “Control Chart Execution in the Debugger”.

Simulation Stepper Limitations

Interface

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

Model Configuration

- Simulation stepping (forward and backward) is available only for Normal and Accelerator modes.
- The step back capability relies on SimState (“Save and Restore Simulation State as SimState” on page 14-32) 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.
- 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. In some cases, stepping back capability is not supported for that simulation. The step back capability is then disabled until the end of that simulation (at which point the setting resets to the originally requested value).

Blocks

- Some blocks do not support stepping back for reasons other than SimState support. Overall, this capability is not available in a model that uses any of these blocks:
 - 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 SimState”)
 - SimMechanics™ First Generation blocks

- Model blocks configured for Accelerator mode
- SimEvents® blocks
- MATLAB Function blocks generally support stepping back. However, use of certain constructs in the MATLAB code of this block can prevent the block from supporting stepping back. These scenarios prevent the MATLAB Function blocks from being fully supported or 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 on a model that contains these blocks:
 - XY Graph
 - Floating Scope
 - Signal Viewer
 - 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 compliant with 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 of the simulation.

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

Step Through a Simulation


In this section...

“Configure Simulation Stepping” on page 2-14

“Step Forward and Back” on page 2-14

Configure Simulation Stepping

To control aspects of the simulation stepping operations, such as to enable stepping back or set how many simulation snapshots to capture, use the Simulation Stepping Options dialog box. To open this dialog box:

- In the Simulation toolbar, click the Stepping Options button .
- Select **Simulation > Stepping Options**. This method is useful when the toolbar icon is unavailable.

For more information on simulation snapshots, see “Simulation Snapshots” on page 2-7.

For a description of the parameters, see “Simulation Stepping Options Dialog Box” on page 2-6.


Step Forward and Back


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

- 1 In the MATLAB Command Window, type

```
vdp
```

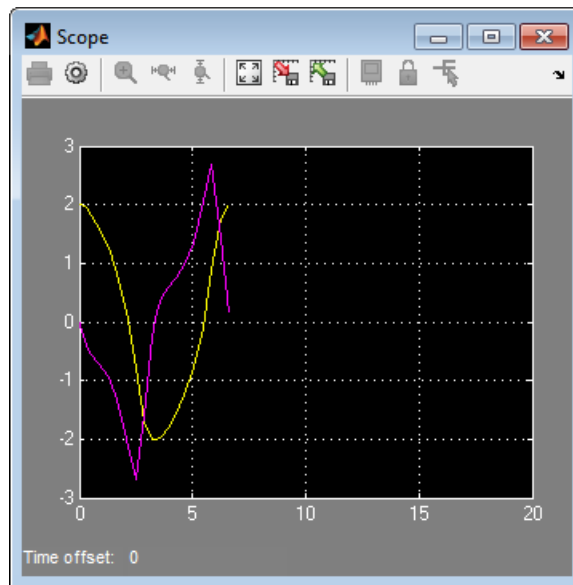
A sample model and its scope appear in the Simulink Editor.

- 2 In the Simulation toolbar, click  to open the Simulation Stepping options dialog box.
- 3 In the dialog box, select the **Enable previous stepping** check box, and then click **OK**.


- 4 In the Simulation toolbar, click .

The simulation simulates one step, and the software stores a simulation snapshot for that step.

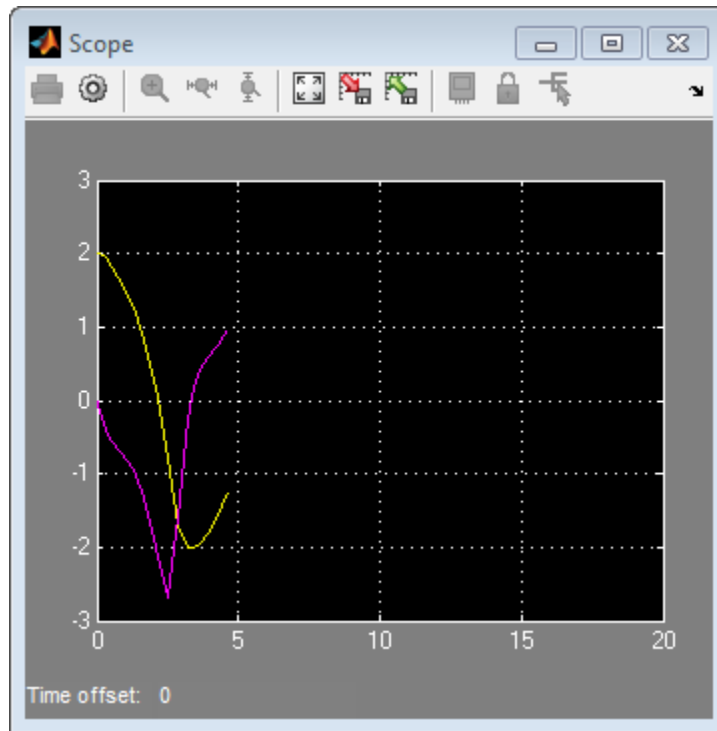
- 5 Click the Step Forward button again to step forward again and store simulation data. When you view the results of the simulation, 25 clicks plots this data:



Note You must step forward before you can step backward to create the simulation state that the step backward operation requires.





- 6 In the Simulation toolbar, click  one or more times to step backward to the previously stepped simulation snapshot.

The scope updates to reflect the step back.



Set Conditional Breakpoints for Stepping a Simulation

Set conditional breakpoints to cause the stepper to stop when a specified condition is met. One example of a good use for conditional breakpoints is when you want to examine results after a certain number of iterations in a loop. The Simulation Stepper allows you to set conditional breakpoints for scalar signals. A conditional breakpoint is a breakpoint that is triggered based on a certain expression evaluated on a signal. When the breakpoint is triggered, the corresponding simulation pauses. Breakpoints appear for signals as follows:

Icon	Description
	Enabled breakpoint. Appears when you add the conditional breakpoint.
	Enabled breakpoint hit. Appears when the simulation reaches the condition specified for the breakpoint and triggers the breakpoint.
	Disabled breakpoint. Appears when you disable a conditional breakpoint (see “Add and Edit Conditional Breakpoints” on page 2-18).
	Invalid breakpoint. Appears when the software determines that a breakpoint is invalid for the signal. An enabled breakpoint icon changes to this icon when, during simulation, the software determines that the conditional breakpoint is invalid.

Notes:

- 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.

For more information on working with conditional breakpoints, see:

- “Add and Edit Conditional Breakpoints” on page 2-18
- “Observe Conditional Breakpoint Values” on page 2-19

Conditional Breakpoints Limitations

You can set conditional breakpoints on real scalar signals of only the following data types:

- double
- single
- int
- bool

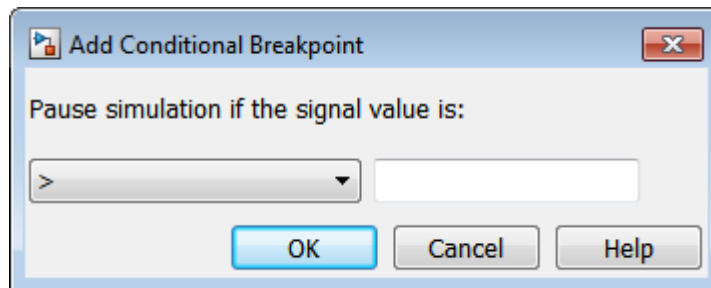
Conditional breakpoints also have the same limitations as those for port value displays (“Port Value Display Limitations” on page 23-34).

Add and Edit Conditional Breakpoints


This tutorial describes how to add conditional breakpoints to a model. For a description of how to see the values of a signal for which you have set a conditional breakpoint, see “Observe Conditional Breakpoint Values” on page 2-19.

- 1 Right-click the signal and select **Add Conditional Breakpoint**.

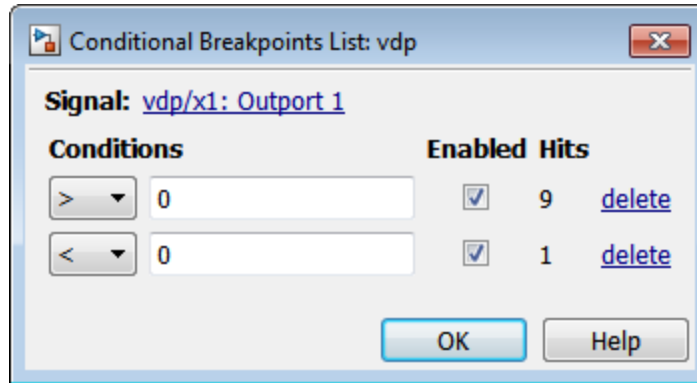
The Add Conditional Breakpoint dialog box appears.



- 2 From the drop-down list, select the condition for the signal, for example, greater than or less than:
- 3 Enter the signal value where you want simulation to pause and click **OK**.

The affected signal line displays a conditional breakpoint icon, such as .

- 4 Click the icon to view and edit the conditions set for the signal.



Condition values:

- Must be numeric. They cannot be expressions.
- Cannot be NaN.

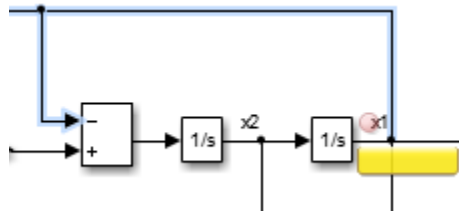
- 5 Simulate the model and notice that the model pauses as simulation steps through the conditional breakpoints.

Observe Conditional Breakpoint Values

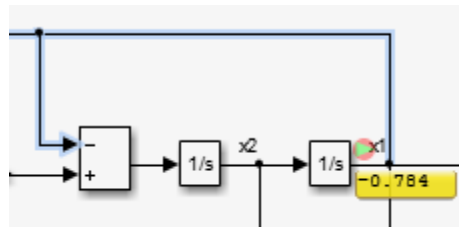
To observe a conditional breakpoint value of a block signal, use data tips to display block port values. You can add data tips before or after you add conditional breakpoints. This tutorial describes how to enable a display a port value after you add a conditional breakpoint. For a more detailed description of how to add conditional breakpoints, see “Add and Edit Conditional Breakpoints” on page 2-18.

- 1 To enable the value display for the signal, right-click the signal line and select **Show Value Label of Selected Port**.

The data tip for the value display is added.



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



How Simulink Works

- “How Simulink Works” on page 3-2
- “Modeling Dynamic Systems” on page 3-3
- “Simulating Dynamic Systems” on page 3-18

How Simulink Works

Simulink is a software package that enables you to model, simulate, and analyze systems whose outputs change over time. Such systems are often referred to as dynamic systems. The Simulink software can be used to explore the behavior of a wide range of real-world dynamic systems, including electrical circuits, shock absorbers, braking systems, and many other electrical, mechanical, and thermodynamic systems. This section explains how Simulink works.

Simulating a dynamic system is a two-step process. First, a user creates a block diagram, using the Simulink model editor, that graphically depicts time-dependent mathematical relationships among the system's inputs, states, and outputs. The user then commands the Simulink software to simulate the system represented by the model from a specified start time to a specified stop time.

For more information on this process, see:

- “Modeling Dynamic Systems” on page 3-3
- “Simulating Dynamic Systems” on page 3-18

Modeling Dynamic Systems

In this section...

“Block Diagram Semantics” on page 3-3

“Creating Models” on page 3-4

“Time” on page 3-5

“States” on page 3-5

“Block Parameters” on page 3-9

“Tunable Parameters” on page 3-9

“Block Sample Times” on page 3-9

“Custom Blocks” on page 3-10

“Systems and Subsystems” on page 3-11

“Signals” on page 3-16

“Block Methods” on page 3-16

“Model Methods” on page 3-17

Block Diagram Semantics

A classic block diagram model of a dynamic system graphically consists of blocks and lines (signals). The history of these block diagram models is derived from engineering areas such as Feedback Control Theory and Signal Processing. A block within a block diagram defines a dynamic system in itself. The relationships between each elementary dynamic system in a block diagram are illustrated by the use of signals connecting the blocks. Collectively the blocks and lines in a block diagram describe an overall dynamic system.

The Simulink product extends these classic block diagram models by introducing the notion of two classes of blocks, nonvirtual blocks and virtual blocks. Nonvirtual blocks represent elementary systems. Virtual blocks exist for graphical and organizational convenience only: they have no effect on the system of equations described by the block diagram model. You can use virtual blocks to improve the readability of your models.

In general, blocks and lines can be used to describe many “models of computations.” One example would be a flow chart. A flow chart consists of blocks and lines, but one cannot describe general dynamic systems using flow chart semantics.

The term “time-based block diagram” is used to distinguish block diagrams that describe dynamic systems from that of other forms of block diagrams, and the term block diagram (or model) is used to refer to a time-based block diagram unless the context requires explicit distinction.

To summarize the meaning of time-based block diagrams:

- Simulink block diagrams define time-based relationships between signals and state variables. The solution of a block diagram is obtained by evaluating these relationships over time, where time starts at a user specified “start time” and ends at a user specified “stop time.” Each evaluation of these relationships is referred to as a time step.
- Signals represent quantities that change over time and are defined for all points in time between the block diagram’s start and stop time.
- The relationships between signals and state variables are defined by a set of equations represented by blocks. Each block consists of a set of equations (block methods). These equations define a relationship between the input signals, output signals and the state variables. Inherent in the definition of an equation is the notion of parameters, which are the coefficients found within the equation.

Creating Models

The Simulink product provides a graphical editor that allows you to create and connect instances of block types (see “Connect Blocks” on page 4-12) selected from libraries of block types (see “Block Libraries”) via a library browser. Libraries of blocks are provided representing elementary systems that can be used as building blocks. The blocks supplied with Simulink are called built-in blocks. Users can also create their own block types and use the Simulink editor to create instances of them in a diagram. User-defined blocks are called custom blocks.

Time

Time is an inherent component of block diagrams in that the results of a block diagram simulation change with time. Put another way, a block diagram represents the instantaneous behavior of a dynamic system. Determining a system's behavior over time thus entails repeatedly solving the model at intervals, called time steps, from the start of the time span to the end of the time span. The process of solving a model at successive time steps is referred to as *simulating* the system that the model represents.

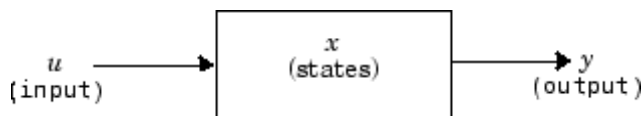
States

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.

Working with States

The following facilities are provided for determining, initializing, and logging a model's states during simulation:

- The `model` command displays information about the states defined by a model, including the total number of states defined by the model, the block that defines each state, and the initial value of each state.
- The Simulink debugger displays the value of a state at each time step during a simulation, and the Simulink debugger's `states` command displays information about the model's current states (see "Debugging").
- The **Data Import/Export** pane of a model's Configuration Parameters dialog box (see "Import and Export States" on page 48-142) allows you to specify initial values for a model's states, and to record the values of the states at each time step during simulation as an array or structure variable in the MATLAB workspace.
- The Block Parameters dialog box (and the `ContinuousStateAttributes` parameter) allows you to give names to states for those blocks (such as the Integrator) that employ continuous states. This can simplify analyzing data logged for states, especially when a block has multiple states.

The Two Cylinder Model with Load Constraints model illustrates the logging of continuous states.

Continuous States

Computing a continuous state entails knowing its rate of change, or derivative. Since the rate of change of a continuous state typically itself changes continuously (i.e., is itself a state), computing the value of a continuous state at the current time step entails integration of its derivative from the start of a simulation. Thus modeling a continuous state entails representing the operation of integration and the process of computing the state's derivative at each point in time. Simulink block diagrams use Integrator blocks to indicate integration and a chain of blocks connected to an integrator block's input to represent the method for computing the state's derivative. The chain of blocks connected to the integrator block's input is the graphical counterpart to an ordinary differential equation (ODE).

In general, excluding simple dynamic systems, analytical methods do not exist for integrating the states of real-world dynamic systems represented by ordinary differential equations. Integrating the states requires the use of numerical methods called ODE solvers. These various methods trade computational accuracy for computational workload. The Simulink product comes with computerized implementations of the most common ODE integration methods and allows a user to determine which it uses to integrate states represented by Integrator blocks when simulating a system.

Computing the value of a continuous state at the current time step entails integrating its values from the start of the simulation. The accuracy of numerical integration in turn depends on the size of the intervals between time steps. In general, the smaller the time step, the more accurate the simulation. Some ODE solvers, called variable time step solvers, can automatically vary the size of the time step, based on the rate of change of the state, to achieve a specified level of accuracy over the course of a simulation. The user can specify the size of the time step in the case of fixed-step solvers, or the solver can automatically determine the step size in the case of variable-step solvers. To minimize the computation workload, the variable-step solver chooses the largest step size consistent with achieving an overall level of precision specified by the user for the most rapidly changing model state. This ensures that all model states are computed to the accuracy specified by the user.

Discrete States

Computing a discrete state requires knowing the relationship between its value at the current time step and its value at the previous time step. This is referred to this relationship as the state's update function. A discrete state depends not only on its value at the previous time step but also on the values of a model's inputs. Modeling a discrete state thus entails modeling the state's dependency on the systems' inputs at the previous time step. Simulink block diagrams use specific types of blocks, called discrete blocks, to specify update functions and chains of blocks connected to the inputs of discrete blocks to model the dependency of a system's discrete states on its inputs.

As with continuous states, discrete states set a constraint on the simulation time step size. Specifically, the step size must ensure that all the sample times of the model's states are hit. This task is assigned to a component of the Simulink system called a discrete solver. Two discrete solvers are provided: a fixed-step discrete solver and a variable-step discrete solver. The fixed-step discrete solver determines a fixed step size that hits all the sample times of all the model's discrete states, regardless of whether the states actually change value at the sample time hits. By contrast, the variable-step discrete solver varies the step size to ensure that sample time hits occur only at times when the states change value.

Modeling Hybrid Systems

A hybrid system is a system that has both discrete and continuous states. Strictly speaking, any model that has both continuous and discrete sample times is treated as a hybrid model, presuming that the model has both continuous and discrete states. Solving such a model entails choosing a step size that satisfies both the precision constraint on the continuous state integration and the sample time hit constraint on the discrete states. The Simulink software meets this requirement by passing the next sample time hit, as determined by the discrete solver, as an additional constraint on the continuous solver. The continuous solver must choose a step size that advances the simulation up to but not beyond the time of the next sample time hit. The continuous solver can take a time step short of the next sample time hit to meet its accuracy constraint but it cannot take a step beyond the next sample time hit even if its accuracy constraint allows it to.

You can simulate hybrid systems using any one of the integration methods, but certain methods are more effective than others. For most hybrid systems,

ode23 and ode45 are superior to the other solvers in terms of efficiency. Because of discontinuities associated with the sample and hold of the discrete blocks, do not use the ode15s and ode113 solvers for hybrid systems.

Block Parameters

Key properties of many standard blocks are parameterized. For example, the Constant value of the Simulink Constant block is a parameter. Each parameterized block has a block dialog that lets you set the values of the parameters. You can use MATLAB expressions to specify parameter values. Simulink evaluates the expressions before running a simulation. You can change the values of parameters during a simulation. This allows you to determine interactively the most suitable value for a parameter.

A parameterized block effectively represents a family of similar blocks. For example, when creating a model, you can set the Constant value parameter of each instance of the Constant block separately so that each instance behaves differently. Because it allows each standard block to represent a family of blocks, block parameterization greatly increases the modeling power of the standard Simulink libraries. See “Block Parameters” on page 3-9 and “Block Libraries” for more information.

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 3-18 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. See “Block Parameters” on page 3-9 and “Tunable Parameters” on page 24-14 for more information.

Block Sample Times

Every Simulink block has a sample time which defines when the block will execute. Most blocks allow you to specify the sample time via a `SampleTime`

parameter. Common choices include discrete, continuous, and inherited sample times.

Common Sample Time Types	Sample Time	Examples
Discrete	$[T_s, T_o]$	Unit Delay, Digital Filter
Continuous	$[0, 0]$	Integrator, Derivative
Inherited	$[-1, 0]$	Gain, Sum

For discrete blocks, the sample time is a vector $[T_s, T_o]$ where T_s is the time interval or period between consecutive sample times and T_o is an initial offset to the sample time. In contrast, the sample times for nondiscrete blocks are represented by ordered pairs that use zero, a negative integer, or infinity to represent a specific type of sample time (see “View Sample Time Information” on page 5-9). For example, continuous blocks have a nominal sample time of $[0, 0]$ and are used to model systems in which the states change continuously (e.g., a car accelerating). Whereas you indicate the sample time type of an inherited block symbolically as $[-1, 0]$ and Simulink then determines the actual value based upon the context of the inherited block within the model.

Note that not all blocks accept all types of sample times. For example, a discrete block cannot accept a continuous sample time.

For a visual aid, Simulink allows the optional color-coding and annotation of any block diagram to indicate the type and speed of the block sample times. You can capture all of the colors and the annotations within a legend (see “View Sample Time Information” on page 5-9).

For a more detailed discussion of sample times, see “Sample Time”

Custom Blocks

You can create libraries of custom blocks that you can then use in your models. You can create a custom block either graphically or programmatically. To create a custom block graphically, you draw a block diagram representing the block’s behavior, wrap this diagram in an instance of the Simulink Subsystem block, and provide the block with a parameter dialog, using the

Simulink block mask facility. To create a block programmatically, you create a MATLAB file or a MEX-file that contains the block's system functions (see "S-Function Basics"). The resulting file is called an S-function. You then associate the S-function with instances of the Simulink S-Function block in your model. You can add a parameter dialog to your S-Function block by wrapping it in a Subsystem block and adding the parameter dialog to the Subsystem block. See "Block Creation" for more information.

Systems and Subsystems

A Simulink block diagram can consist of layers. Each layer is defined by a subsystem. A subsystem is part of the overall block diagram and ideally has no impact on the meaning of the block diagram. Subsystems are provided primarily to help with the organizational aspects of a block diagram. Subsystems do not define a separate block diagram.

The Simulink software differentiates between two different types of subsystems: virtual and nonvirtual. The primary difference is that nonvirtual subsystems provide the ability to control when the contents of the subsystem are evaluated.

Virtual Subsystems

Virtual subsystems provide graphical hierarchy in models. Virtual subsystems do not impact execution. During model execution, the Simulink engine flattens all virtual subsystems, i.e., Simulink expands the subsystem in place before execution. This expansion is very similar to the way macros work in a programming language such as C or C++. Roughly speaking, there will be one system for the top-level block diagram which is referred to as the root system, and several lower-level systems derived from nonvirtual subsystems and other elements in the block diagram. You will see these systems in the Simulink Debugger. The act of creating these internal systems is often referred to as *flattening the model hierarchy*.

Nonvirtual Subsystems

Nonvirtual subsystems, which are drawn with a bold border, provide execution and graphical hierarchy in models. Nonvirtual subsystems are executed as a single unit (atomic execution) by the Simulink engine. You can create conditionally executed subsystems that are executed only when

a precondition—such as a trigger, an enable, a function-call, or an action—occurs (see “Conditional Subsystems”). Simulink always computes all inputs used during the execution of a nonvirtual subsystem before executing the subsystem. Simulink defines the following nonvirtual subsystems.

Atomic subsystems. The primary characteristic of an atomic subsystem is that blocks in an atomic subsystem execute as a single unit. This provides the advantage of grouping functional aspects of models at the execution level. Any Simulink block can be placed in an atomic subsystem, including blocks with different execution rates. You can create an atomic subsystem by selecting the **Treat as atomic unit** option on a virtual subsystem (see the Atomic Subsystem block for more information).

Enabled subsystems. An enabled subsystem behaves similarly to an atomic subsystem, except that it executes only when the signal driving the subsystem enable port is greater than zero. To create an enabled subsystem, place an Enable Port block within a Subsystem block. You can configure an enabled subsystem to hold or reset the states of blocks within the enabled subsystem prior to a subsystem enabling action. Simply select the **States when enabling** parameter of the Enable Port block. Similarly, you can configure each output port of an enabled subsystem to hold or reset its output prior to the subsystem disabling action. Select the **Output when disabled** parameter in the Output block.

Triggered subsystems. You create a triggered subsystem by placing a trigger port block within a subsystem. The resulting subsystem executes when a rising or falling edge with respect to zero is seen on the signal driving the subsystem trigger port. The direction of the triggering edge is defined by the **Trigger type** parameter on the trigger port block. Simulink limits the type of blocks placed in a triggered subsystem to blocks that do not have explicit sample times (i.e., blocks within the subsystem must have a sample time of -1) because the contents of a triggered subsystem execute in an aperiodic fashion. A Stateflow chart can also have a trigger port which is defined by using the Stateflow editor. Simulink does not distinguish between a triggered subsystem and a triggered chart.

Function-call subsystems. A function-call subsystem is a subsystem that another block can invoke directly during a simulation. It is analogous to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output and update methods of the blocks that the subsystem contains in sorted order. The block that invokes a function-call subsystem is called the function-call initiator. Stateflow, Function-Call Generator, and S-function blocks can all serve as function-call initiators. To create a function-call subsystem, drag a Function-Call Subsystem block from the Ports & Subsystems library into your model and connect a function-call initiator to the function-call port displayed on top of the subsystem. You can also create a function-call subsystem from scratch by first creating a Subsystem block in your model and then creating a Trigger block in the subsystem and setting the Trigger block `Trigger type` to `function-call`.

You can configure a function-call subsystem to be triggered (the default) or periodic by setting its `Sample time type` to be `triggered` or `periodic`, respectively. A function-call initiator can invoke a triggered function-call subsystem zero, once, or multiple times per time step. The sample times of all the blocks in a triggered function-call subsystem must be set to `inherited (-1)`.

A function-call initiator can invoke a periodic function-call subsystem only once per time step and must invoke the subsystem periodically. If the initiator invokes a periodic function-call subsystem aperiodically, Simulink halts the simulation and displays an error message. The blocks in a periodic function-call subsystem can specify a noninherited sample time or `inherited (-1)` sample time. All blocks that specify a noninherited sample time must specify the same sample time, that is, if one block specifies `.1` as its sample time, all other blocks must specify a sample time of `.1` or `-1`. If a function-call initiator invokes a periodic function-call subsystem at a rate that differs from the sample time specified by the blocks in the subsystem, Simulink halts the simulation and displays an error message.

Enabled and triggered subsystems. You can create an enabled and triggered subsystem by placing a Trigger Port block and an Enable Port block within a Subsystem block. The resulting subsystem is essentially a triggered subsystem that executes when the subsystem is enabled and a rising or falling edge with respect to zero is seen on the signal driving the subsystem trigger port. The direction of the triggering edge is defined by the `Trigger` type parameter on the trigger port block. Because the contents of a triggered subsystem execute in an aperiodic fashion, Simulink limits the types of blocks placed in an enabled and triggered subsystem to blocks that do not have explicit sample times. In other words, blocks within the subsystem must have a sample time of `-1`).

Action subsystems. Action subsystems can be thought of as an intersection of the properties of enabled subsystems and function-call subsystems. Action subsystems are restricted to a single sample time (e.g., a continuous, discrete, or inherited sample time). Action subsystems must be executed by an action subsystem initiator. This is either an If block or a Switch Case block. All action subsystems connected to a given action subsystem initiator must have the same sample time. An action subsystem is created by placing an Action Port block within a Subsystem block. The subsystem icon will automatically adapt to the type of block (i.e., If or Switch Case block) that is executing the action subsystem.

Action subsystems can be executed at most once by the action subsystem initiator. Action subsystems give you control over when the states reset via the `States when execution is resumed` parameter on the Action Port block. Action subsystems also give you control over whether or not to hold the output values via the `Output when disabled` parameter on the output block. This is analogous to enabled subsystems.

Action subsystems behave very similarly to function-call subsystems because they must be executed by an initiator block. *Function-call subsystems can be executed more than once at any given time step whereas action subsystems can be executed at most once.* This restriction means that a larger set of blocks (e.g., periodic blocks) can be placed in action subsystems as compared to function-call subsystems. This restriction also means that you can control how the states and outputs behave.

While iterator subsystems. A while iterator subsystem will run multiple iterations on each model time step. The number of iterations is controlled by the While Iterator block condition. A while iterator subsystem is created by placing a While Iterator block within a subsystem block.

A while iterator subsystem is very similar to a function-call subsystem in that it can run for any number of iterations at a given time step. The while iterator subsystem differs from a function-call subsystem in that there is no separate initiator (e.g., a Stateflow Chart). In addition, a while iterator subsystem has access to the current iteration number optionally produced by the While Iterator block. A while iterator subsystem also gives you control over whether or not to reset states when starting via the `States when starting` parameter on the While Iterator block.

For iterator subsystems. A for iterator subsystem will run a fixed number of iterations at each model time step. The number of iterations can be an external input to the for iterator subsystem or specified internally on the For Iterator block. A for iterator subsystem is created by placing a For Iterator block within a subsystem block.

A for iterator subsystem has access to the current iteration number that is optionally produced by the For Iterator block. A for iterator subsystem also gives you control over whether or not to reset states when starting via the `States when starting` parameter on the For Iterator block. A for iterator subsystem is very similar to a while iterator subsystem with the restriction that the number of iterations during any given time step is fixed.

For each subsystems. The for each subsystem allows you to repeat an algorithm for individual elements (or subarrays) of an input signal. Here, the algorithm is represented by the set of blocks in the subsystem and is applied to a single element (or subarray) of the signal. You can configure the decomposition of the subsystem inputs into elements (or subarrays) using the For Each block, which resides in the subsystem. The For Each block also allows you to configure the concatenation of individual results into output signals. An advantage of this subsystem is that it maintains separate sets of states for each element or subarray that it processes. In addition, for certain models, the for each subsystem improves the code reuse of the code generated by Simulink Coder™.

Signals

The term *signal* refers to 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 (e.g., 8-bit, 16-bit, or 32-bit integer), numeric type (real or complex), and dimensionality (one-dimensional, two-dimensional, or multidimensional array). Many blocks can accept or output signals of any data or numeric type and dimensionality. Others impose restrictions on the attributes of the signals they can handle.

On the block diagram, signals are represented 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's methods (equations).

A good way to understand the definition of a signal is to consider a classroom. The teacher is the one responsible for writing on the white board and the students read what is written on the white board when they choose to. This is also true of Simulink signals: a reader of the signal (a block method) can choose to read the signal as frequently or infrequently as so desired.

For more information about signals, see “Signals”.

Block Methods

Blocks represent multiple equations. These equations are represented as block methods. These block methods are evaluated (executed) during the execution of a block diagram. The evaluation of these block methods is performed within a simulation loop, where each cycle through the simulation loop represents the evaluation of the block diagram at a given point in time.

Method Types

Names are assigned to the types of functions performed by block methods. Common method types include:

- Outputs

Computes the outputs of a block given its inputs at the current time step and its states at the previous time step.

- Update
Computes the value of the block's discrete states at the current time step, given its inputs at the current time step and its discrete states at the previous time step.
- Derivatives
Computes the derivatives of the block's continuous states at the current time step, given the block's inputs and the values of the states at the previous time step.

Method Naming Convention

Block methods perform the same types of operations in different ways for different types of blocks. The Simulink user interface and documentation uses dot notation to indicate the specific function performed by a block method:

`BlockType.MethodType`

For example, the method that computes the outputs of a Gain block is referred to as

`Gain.Outputs`

The Simulink debugger takes the naming convention one step further and uses the instance name of a block to specify both the method type and the block instance on which the method is being invoked during simulation, e.g.,

`g1.Outputs`

Model Methods

In addition to block methods, a set of methods is provided that compute the model's properties and its outputs. The Simulink software similarly invokes these methods during simulation to determine a model's properties and its outputs. The model methods generally perform their tasks by invoking block methods of the same type. For example, the model Outputs method invokes the Outputs methods of the blocks that it contains in the order specified by the model to compute its outputs. The model Derivatives method similarly invokes the Derivatives methods of the blocks that it contains to determine the derivatives of its states.

Simulating Dynamic Systems

In this section...

“Model Compilation” on page 3-18

“Link Phase” on page 3-19

“Simulation Loop Phase” on page 3-19

“Solvers” on page 3-21

“Zero-Crossing Detection” on page 3-24

“Algebraic Loops” on page 3-39

Model Compilation

The first phase of simulation occurs when the system’s model is open and you simulate the model. In the Simulink Editor, select **Simulation > 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.
- A process called attribute propagation is used to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives.
- Performs block reduction optimizations.
- Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain (see “Solvers” on page 3-21).
- Determines the block sorted order (see “Control and Display the Sorted Order” on page 23-36 for more information).

- 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 5-33).

These events are essentially the same as what occurs when you update a diagram (“Update a Block Diagram” on page 1-32). 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 3-19. 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 sorted order lists generated during the model compilation phase is used to construct the method execution lists.

Block Priorities

You can assign update priorities to blocks (see “Assign Block Priorities” on page 23-50). The output methods of higher priority blocks are executed before those of lower priority blocks. The priorities are honored only if they are consistent with its block sorting rules.

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 “Solvers” on page 3-21) used to compute the system’s continuous states, the system’s fundamental sample time (see “Sample Times in Systems” on page 5-26), and whether the system’s continuous states have discontinuities (see “Zero-Crossing Detection” on page 3-24).

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.

Loop Iteration

At each time step, the Simulink engine:

1 Computes the model’s 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 “Solvers” on page 3-21).

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 3-24 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.

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, a set of programs, known as *solvers*, are provided

that each embody a particular approach to solving a model. The Configuration Parameters dialog box allows you to choose the solver most suitable for your model (see “Choosing a Solver Type” on page 14-10).

Fixed-Step Solvers Versus Variable-Step Solvers

The solvers provided in the Simulink software fall into two basic categories: 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, reducing the step size to increase accuracy when a model’s states are changing rapidly and increasing 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 simulation time, required to maintain a specified level of accuracy for models with rapidly changing or piecewise continuous states.

Continuous Versus Discrete Solvers

The Simulink product provides both continuous and discrete solvers.

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.

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 “Choosing a Solver Type” on page 14-10).

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.

Note You must use a continuous solver to solve a model that contains both continuous and discrete states. You cannot use a discrete solver 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.

Two discrete solvers are provided: A fixed-step discrete solver and a variable-step discrete solver. The fixed-step solver by default chooses a step size and hence 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 can avoid unnecessary steps and hence shorten simulation time for multirate models (see “Sample Times in Systems” on page 5-26 for more information).

Minor Time Steps

Some continuous solvers subdivide the simulation time span into major and minor time steps, where a minor time step represents a subdivision of the major time step. 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.

Shape Preservation

Usually the integration step size is only related to the current step size and the current integration error. However, for signals whose derivative changes rapidly, you can obtain a more accurate integration results by including the derivative input information at each time step. To do so, enable the **Model Configuration Parameters > Solver > Shape Preservation** option.

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.

The Simulink software 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.

Two algorithms are provided in the Simulink software: Nonadaptive and Adaptive. For information about these techniques, see “Zero-Crossing Algorithms” on page 3-33.

Demonstrating Effects of Excessive Zero-Crossing Detection

The Simulink software comes with three models that illustrate zero-crossing behavior: `sldemo_bounce_two_integrators`, `sldemo_doublebounce`, and `sldemo_bounce`.

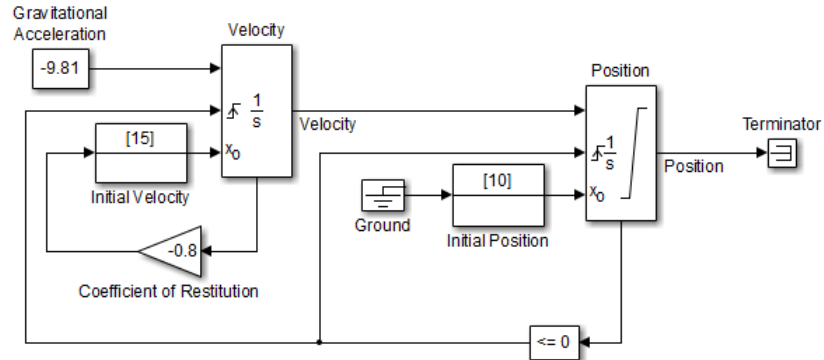
- The `sldemo_bounce_two_integrators` model demonstrates how excessive zero crossings can cause a simulation to halt before the intended completion time unless you use the adaptive algorithm.
- The `sldemo_bounce` model uses a better model design than `sldemo_bounce_two_integrators`.
- The `sldemo_doublebounce` model demonstrates how the adaptive algorithm successfully solves a complex system with two distinct zero-crossing requirements.

The Bounce Model with Two Integrators.



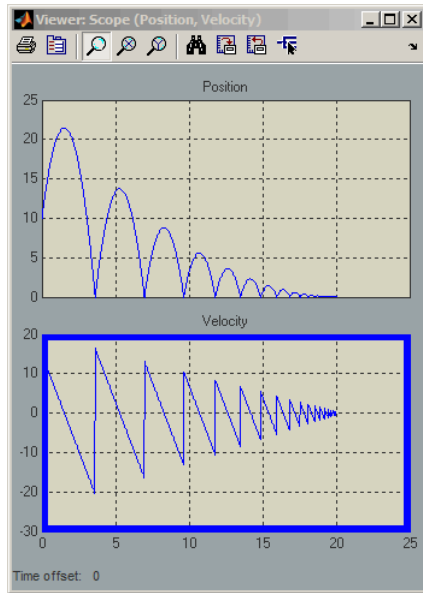
Bouncing Ball Model

Two separate Integrators are less efficient than a single Second-Order Integrator for simulating a bouncing ball. [Click here to see sldemo_bounce for the recommended modeling approach.](#)

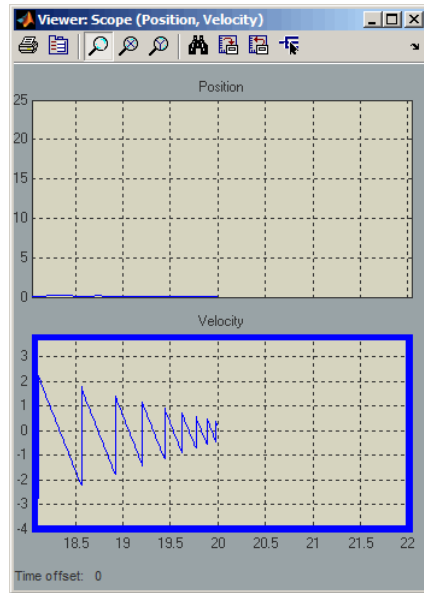


- 1** At the MATLAB command prompt, type `sldemo_bounce_two_integrators` to load the example.
- 2** Once the block diagram appears, set the **Model Configuration Parameters** > **Solver** > **Algorithm** parameter to Nonadaptive.
- 3** Also in the **Solver** pane, set the **Stop time** parameter to 20 s.
- 4** Run the model. In the Simulink Editor, select **Simulation** > **Run**.
- 5** After the simulation completes, click the Scope block window to see the results.

You may need to click on **Autoscale** to view the results in their entirety.



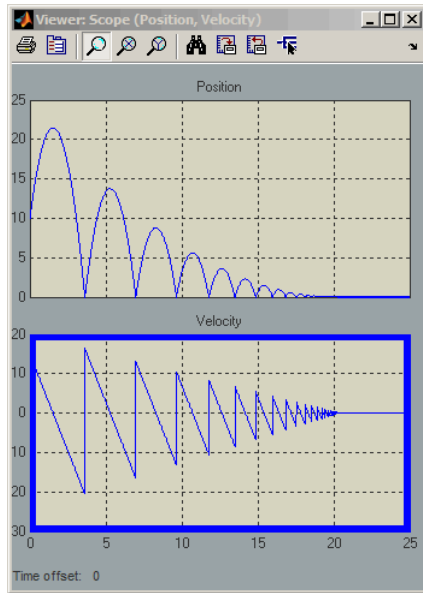
- 6 Use the scope zoom controls to closely examine the last portion of the simulation. You can see that the velocity is hovering just above zero at the last time point.



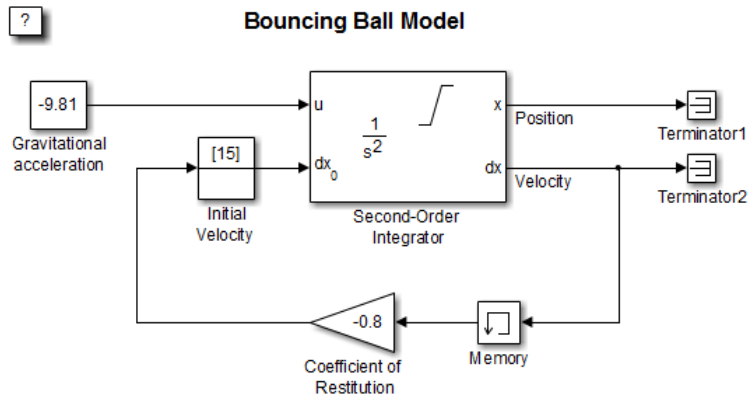
- 7 Change the simulation **Stop time** edit box in the Simulink Editor toolbar to 25 seconds, and run the simulation again.
- 8 This time the simulation halts with an error shortly after it passes the simulated 20 second time point.

Excessive chattering as the ball repeatedly approaches zero velocity has caused the simulation to exceed the default limit of 1000 for the number of consecutive zero crossings allowed. Although you can increase this limit by adjusting the **Model Configuration Parameters > Solver > Number of consecutive zero crossings** parameter. In this case, making that change does not allow the simulation to simulate for 25 seconds.

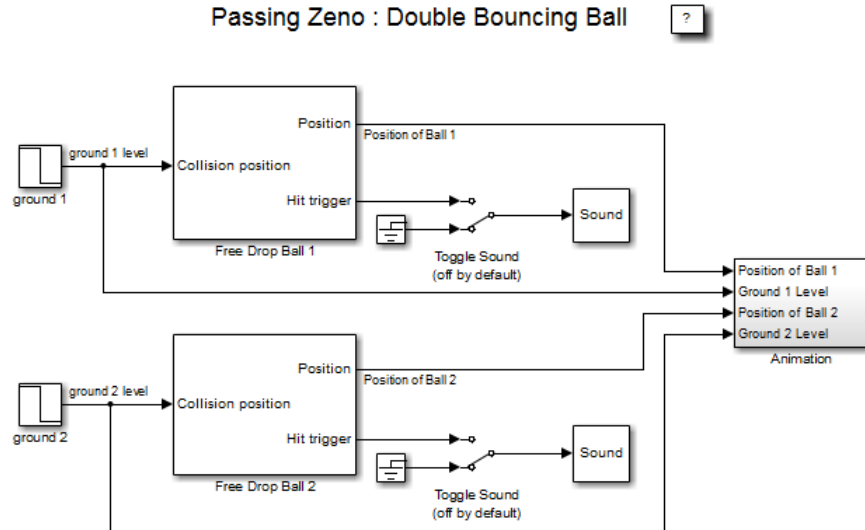
- 9 Also in the **Solver** pane, from the **Algorithm** pull down menu, select the **Adaptive** algorithm.
- 10 Run the simulation again.
- 11 This time the simulation runs to completion because the adaptive algorithm prevented an excessive number of zero crossings from occurring.



Bounce Model with a Second-Order Integrator.



The Double-Bounce Model.



- 1 At the MATLAB command prompt, type `sldemo_doublebounce` to load the example. The model and an animation window open. In the animation window, two balls are resting on two platforms.
- 2 In the animation window, click the **Nonadaptive** button to run the example using the nonadaptive algorithm. This is the default setting used by the Simulink software for all models.
- 3 The ball on the right is given a larger initial velocity and has Consequently, the two balls hit the ground and recoil at different times.
- 4 The simulation halts after 14 seconds because the ball on the left exceeded the number of zero crossings limit. The ball on the right is left hanging in mid air.
- 5 An error message dialog opens. Click **OK** to close it.
- 6 Click on the **Adaptive** button to run the simulation with the adaptive algorithm.

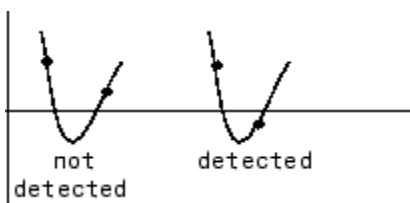
- 7 Notice that this time the simulation runs to completion, even after the ground shifts out from underneath the ball on the left at 20 seconds.

How the Simulator Can Miss Zero-Crossing Events

The bounce and double-bounce models 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 will then hunt 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 change in sign and so detects the zero-crossing event.



Preventing Excessive Zero Crossings

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

Make this change...	How to make this change...	Rationale for making this change...
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.
Use the Adaptive Algorithm	Select Adaptive from the Algorithm pull 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 .

Make this change...	How to make this change...	Rationale for making this change...
<p>Disable zero-crossing detection for a specific block</p>	<ol style="list-style-type: none"> 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. 	<p>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.</p>
<p>Disable zero-crossing detection for the entire model</p>	<p>Select Disable all from the Zero-crossing control pull down on the Solver pane of the Configuration Parameters dialog box.</p>	<p>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.</p>
<p>If using the ode15s solver, consider adjusting the order of the numerical differentiation formulas</p>	<p>Select a value from the Maximum order pull down on the Solver pane of the Configuration Parameters dialog box.</p>	<p>For more information, see "Maximum order".</p>
<p>Reduce the maximum step size</p>	<p>Enter a value for the Max step size option on the Solver pane of the Configuration Parameters dialog box.</p>	<p>This can insure the solver takes steps small enough to resolve the zero crossing. However, reducing the step size can increase simulation time, and</p>

Make this change...	How to make this change...	Rationale for making this change...
		is seldom necessary when using the Adaptive algorithm.

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:

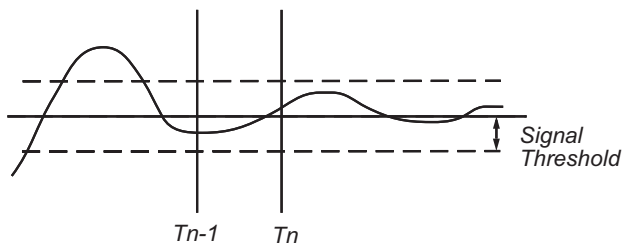
- The zero crossing error is exceeded. This is determined by the value specified in the **Signal threshold** option in the Solver pane of the Configuration Parameters dialog box. This can also be set with the `ZCThreshold` command. The default is Auto, but you can enter any real number greater than zero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

Understanding Signal Threshold

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.

How Blocks Work with Zero-Crossing Detection

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	Description of Zero Crossing
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).
Enable	One: If an Enable port is inside of a Subsystem block, it provides the capability to detect zero crossings. See the Enable Subsystem block for details “Create an Enabled Subsystem” on page 7-17.

Block	Description of Zero Crossing
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 Builder	One: to detect when the input signal has a discontinuity in either the rising or falling direction

Block	Description of Zero Crossing
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. See the Triggered Subsystem block for details: “Create a Triggered Subsystem” on page 7-33.
Enabled and Triggered Subsystem	Two: one for the enable port and one for the trigger port. See the Triggered and Enabled Subsystem block for details: “Create a Triggered and Enabled Subsystem” on page 7-42

Note Zero-crossing detection is also available for a Stateflow chart that uses continuous-time mode. See “Configure a Stateflow Chart to Update in Continuous Time” in the Stateflow documentation 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 3-35 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's 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.

Algebraic Loops

- “What Is an Algebraic Loop?” on page 3-39
- “Problems Caused by Algebraic Loops” on page 3-43
- “Identifying Algebraic Loops in Your Model” on page 3-43
- “What If I Have an Algebraic Loop in My Model?” on page 3-47
- “Simulink Algebraic Loop Solver” on page 3-49
- “Removing Algebraic Loops” on page 3-51
- “Additional Techniques to Help the Algebraic Loop Solver” on page 3-54
- “Changing Block Priorities Does Not Remove Algebraic Loops” on page 3-55
- “Artificial Algebraic Loops” on page 3-55

What Is an Algebraic Loop?

- “Algebraic Loops in Simulink” on page 3-39
- “Mathematical Definition of an Algebraic Loop” on page 3-41
- “Meaning of Algebraic Loops in Physical Systems” on page 3-42

Algebraic Loops in Simulink. An *algebraic loop* in a Simulink model occurs when a signal loop exists with only direct feedthrough blocks within the loop. *Direct feedthrough* means that the block output depends on the value of an input port; the value of the input directly controls the value of the output. *Non-direct-feedthrough* blocks maintain a State variable. Two examples are the Integrator or Unit Delay block.

Some Simulink blocks have input ports with direct feedthrough. The software cannot compute the output of these blocks without knowing the values of the signals entering the blocks at these input ports at the current time step.

Some examples of blocks with direct feedthrough inputs are:

- Math Function block
- Gain block

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

Tip To determine if a block has direct feedthrough:

1 Double-click the block.

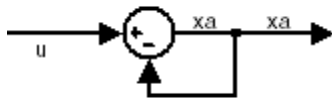
The block parameter dialog box opens.

2 Click the **Help** button in the block parameter dialog box.

The block reference page opens.

3 Scroll to the **Characteristics** section of the block reference page, which lists whether or not that block has direct feedthrough.

An example of an algebraic loop is the following simple loop. Note that this is *not* a recommended modeling pattern.



Mathematically, this loop implies that the output of 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 Definition of an Algebraic Loop. Simulink contains a suite of numerical solvers for simulating *ordinary differential equations (ODEs)*, which are systems of equations that you can write as:

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

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 *differential algebraic equations (DAEs)*, not ODEs.

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

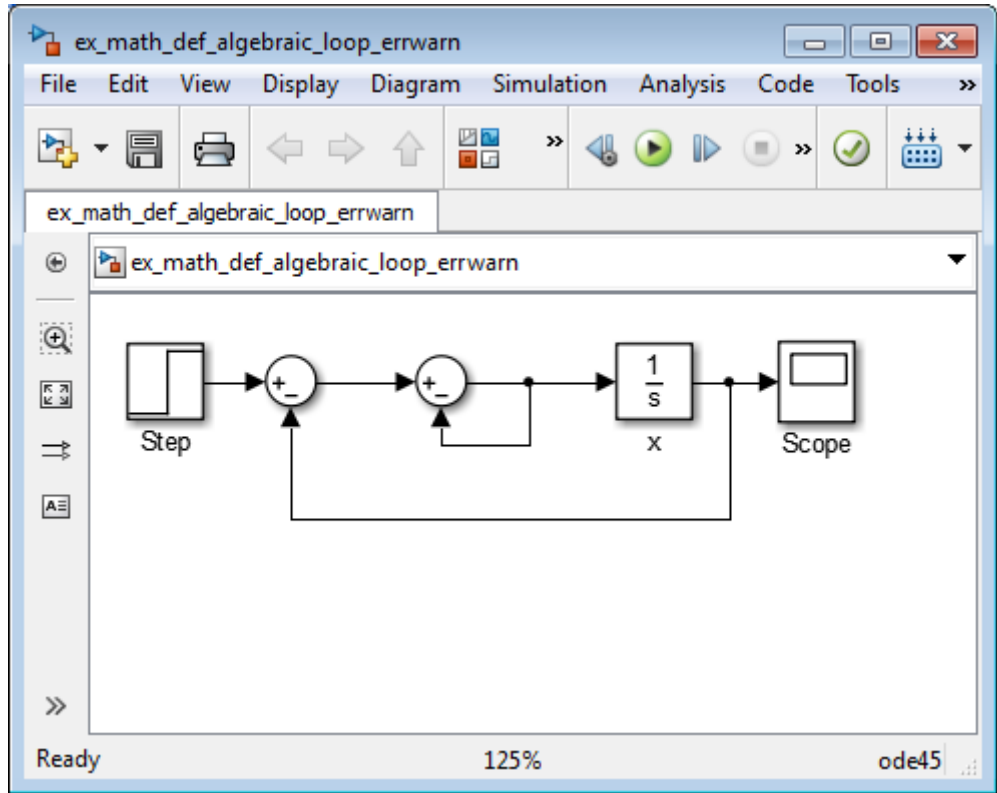
$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{x}_a, t) \\ 0 &= \mathbf{g}(\mathbf{x}, \mathbf{x}_a, t),\end{aligned}$$

- \mathbf{f} and \mathbf{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 \mathbf{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 does not solve DAEs directly. Simulink solves the algebraic equations (the algebraic loop) numerically for x_a at each step of the ODE solver.

The following Simulink model is equivalent to this system of equations in semi-explicit form:

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



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} .

Meaning of Algebraic Loops in Physical Systems. Algebraic constraints can occur when modeling physical systems, often due to conservation laws, such as conservation of mass and energy. You can also use algebraic constraints to impose design constraints on system responses in a dynamic system.

Choosing a particular coordinate system for a model can also result in an algebraic constraint. In most cases, you can eliminate algebraic loops, as described in “Removing Algebraic Loops” on page 3-51, to produce an ordinary differential equation (ODE). However, this technique may be too time consuming if you have a large, complex model.

MathWorks® offers the Simscape™ software that extends Simulink by providing tools to model systems that span mechanical, electrical, hydraulic, and other physical domains as physical networks.

Simscape software automatically constructs the differential algebraic equations (DAEs) that characterize the behavior of a Simulink model. These equations are integrated with the rest of the model, and the Simscape software solves the DAEs directly. The variables for the components in the different physical domains are solved simultaneously, thereby avoiding problems with algebraic loops.

Problems Caused by Algebraic Loops

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 might 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.

Identifying Algebraic Loops in Your Model

- “Algebraic Loop Diagnostic” on page 3-43
- “Highlighting Algebraic Loops Using the Algebraic Loop Diagnostic” on page 3-44
- “Highlighting Algebraic Loops Using the `ashow` Debugger Command” on page 3-45

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 Configuration Parameters dialog box, on the main **Diagnostics** pane, set the **Algebraic loop** parameter as follows.

Setting	Simulation Response
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.

If you want Simulink to try to solve the loop, select **none** or **warning**. If you want to review the loop before Simulink tries to solve the loop, select **error**.

Highlighting Algebraic Loops Using the Algebraic Loop Diagnostic. To highlight algebraic loops in the Simulink Editor when updating or simulating a model:

- 1 Open the `sldemo_hydcyl` model:

```
sldemo_hydcyl
```

- 2 Open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters**.

- 3 On the **Diagnostics** pane, set the **Algebraic loop** parameter to **error**.

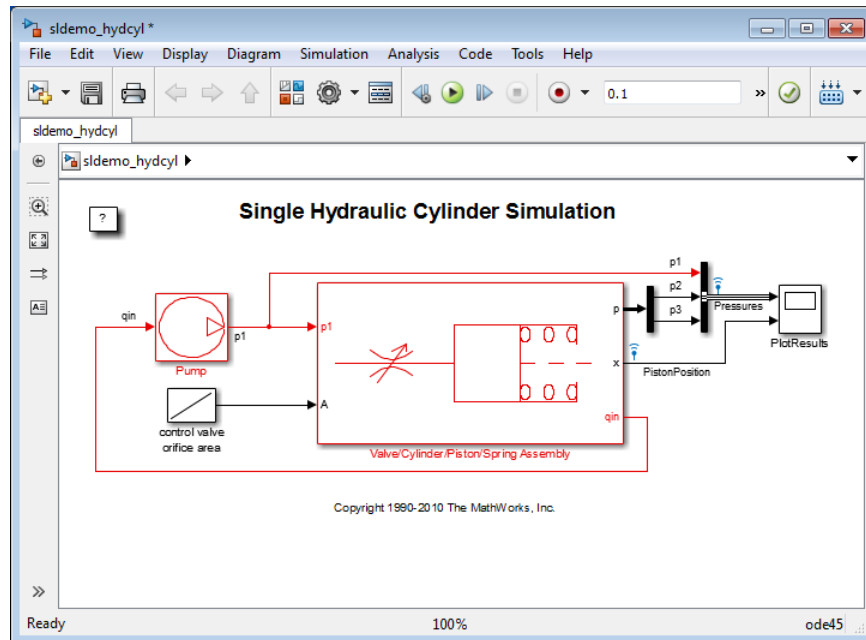
If the Simulink software finds an algebraic loop in the model, it should stop the simulation and report an error.

- 4 Click **OK** to save the setting.

- 5 Click the **Start simulation** button.

When Simulink detects an algebraic loop during initialization, the simulation stops. The Diagnostics Viewer displays an error message and lists all the blocks in the model that are part of that algebraic loop.

In the Simulink Editor, the software highlights the blocks and signals that constitute the loop in red.



6 To restore the diagram to its original colors, close the Diagnostics Viewer.

7 Close the `sldemo_hydcyl` model without saving the changes.

In the next section, the `ashow` debugger command shows that this model actually has two algebraic loops.

Highlighting Algebraic Loops Using the `ashow` Debugger Command.

The Simulink debugger allows you to step through a model simulation. To use the `ashow` command to highlight algebraic loops:

1 Open the `sldemo_hydcyl` model.

By default, the **Algebraic loop** parameter for this model is set to none.

2 Start the Simulink debugger. In the Simulink Editor, select **Simulation > Debug > Debug Model**.

3 Click the **Start/Continue** button to start the debugger.

4 In the MATLAB Command Window, type:

```
ashow
```

The software lists the two algebraic loops in the `sldemo_hydcyl` model and the number of blocks in each algebraic loop.

```
Found 2 Algebraic loop(s):  
System number#Algebraic loop id, number of blocks in loop  
- 0#1, 9 blocks in loop  
- 0#2, 4 blocks in loop
```

5 To list the blocks in the first algebraic loop, in the MATLAB Command Window, enter the following command:

```
ashow 0#1
```

The software opens the Control Valve Flow subsystem in the Valve/Cylinder/Piston/Spring Assembly subsystem, highlights that algebraic loop in the model, and lists the nine blocks in the algebraic loop:

```
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/IC  
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/signed sqrt  
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/Product  
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/laminar flow pressure drop  
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Sum7  
- sldemo_hydcyl/Pump/IC  
- sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/  
Sum1 (algebraic variable)  
- sldemo_hydcyl/Pump/Sum1  
- sldemo_hydcyl/Pump/leakage (algebraic variable)
```

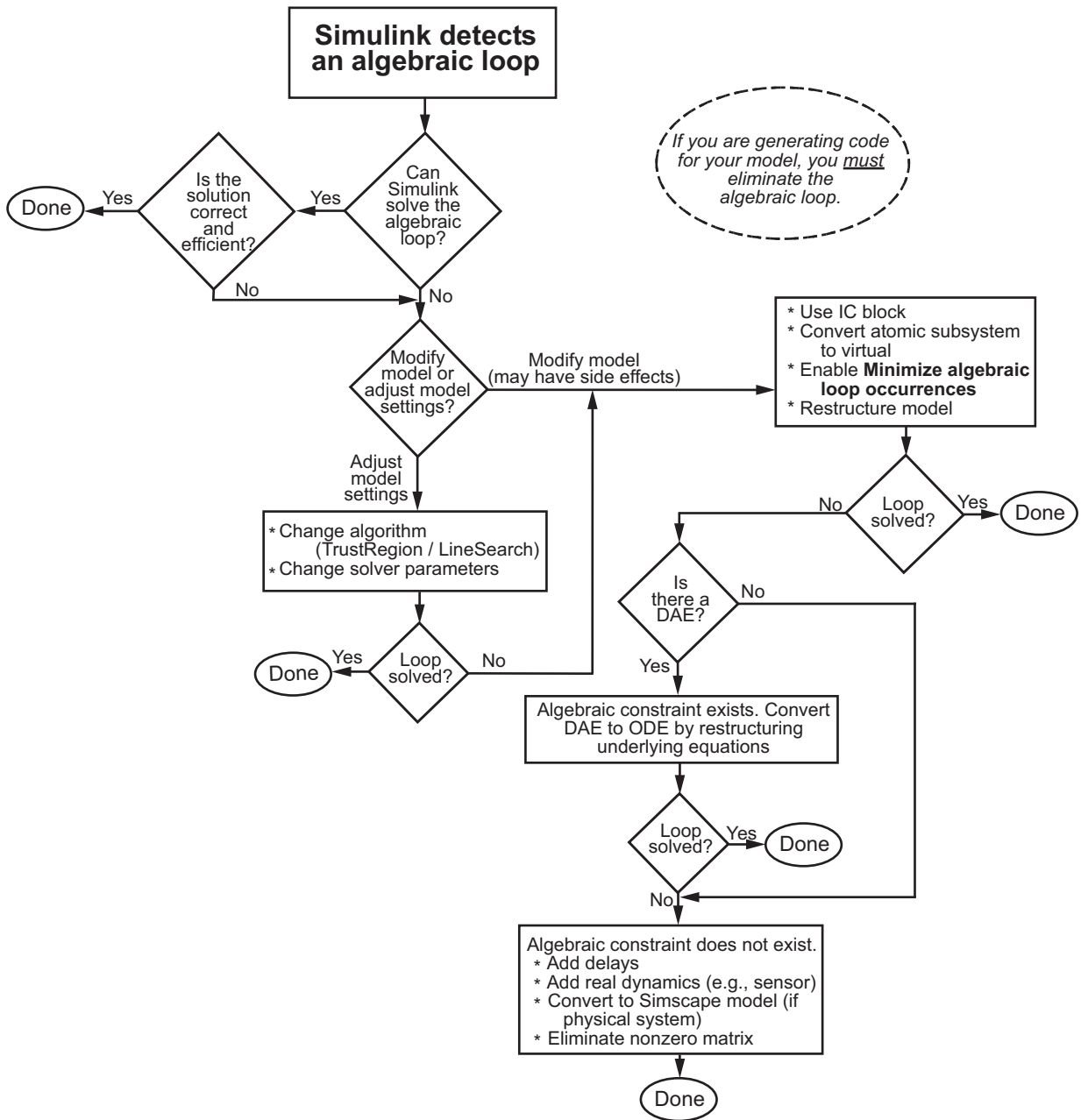
6 In the Simulink debugger, click **Close**.

7 At the MATLAB command prompt, press **Enter** to restore the default MATLAB command prompt.

What If I Have an Algebraic Loop in My Model?

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

The following workflow helps you evaluate what techniques to try to eliminate an algebraic loop. Some of those techniques are described in the following sections.



Simulink Algebraic Loop Solver

- “How the Algebraic Loop Solver Works” on page 3-49
- “Trust-Region and Line-Search Algorithms in the Algebraic Loop Solver” on page 3-50
- “Limitations of the Algebraic Loop Solver” on page 3-51

How the Algebraic Loop Solver Works. When a model contains an algebraic loop, the Simulink software 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 possible). 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).

The algebraic loop solver requires:

- One block where the loop solver can break the loop and attempt to solve the loop
- 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 as follows:

$$x_a(t) = u(t) / 2.$$

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 might 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 and line search. 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

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.
2. More, J.J., B.S.Garbow, and K.E.Hillstom. *User guide for MINPACK-1*. Argonne, IL: Argonne National Laboratory, 1980.

- 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” in the Optimization Toolbox™ documentation.
- “Line Search” in the Optimization Toolbox documentation.

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

Removing Algebraic Loops

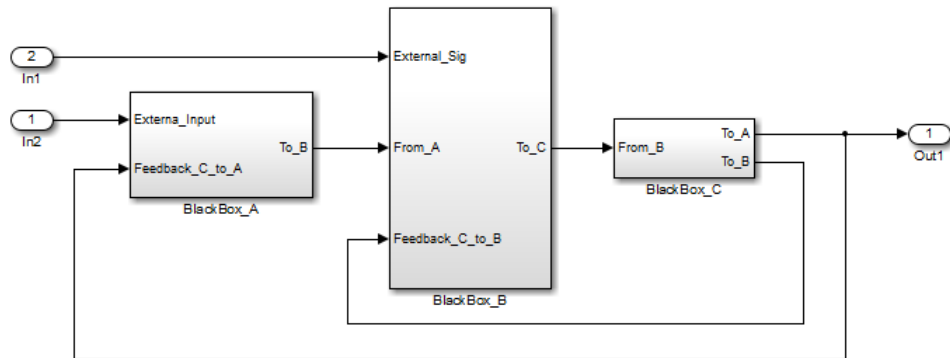
Introducing a Delay.

Algebraic loops can occur in large models when atomic subsystems create feedback loops.

In the following generic model, there are two algebraic loops that involve subsystems.

- BlackBox_A → BlackBox_B → BlackBox_C → BlackBox_A
- BlackBox_B → BlackBox_C → BlackBox_B

3. Rabinowitz, Philip, ed. *Numerical Methods for Nonlinear Algebraic Equations*, New York: Gordon and Breach Science Publishers, 1970.



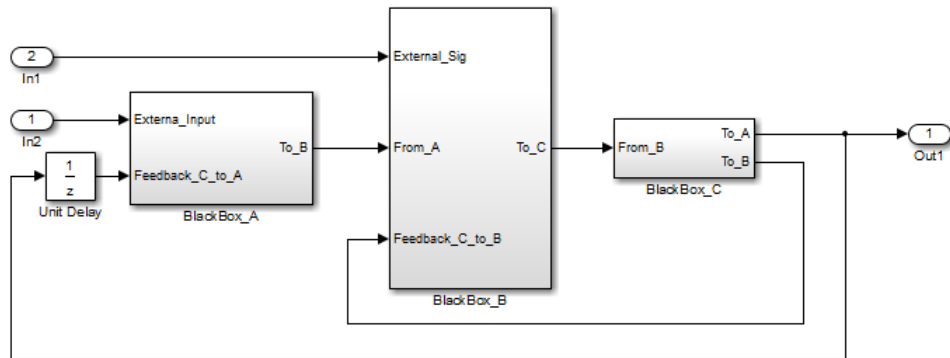
When you update this model, Simulink detects the loop BlackBox_A → BlackBox_B → BlackBox_C → BlackBox_A.

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

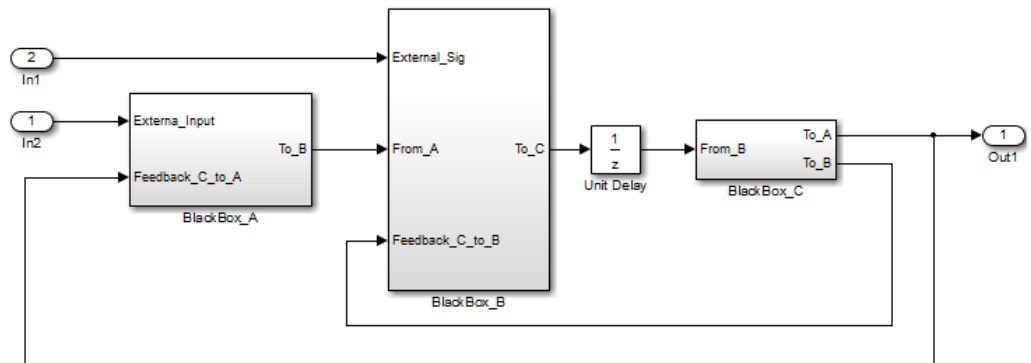
Add a unit delay between BlackBox_A and BlackBox_C

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.



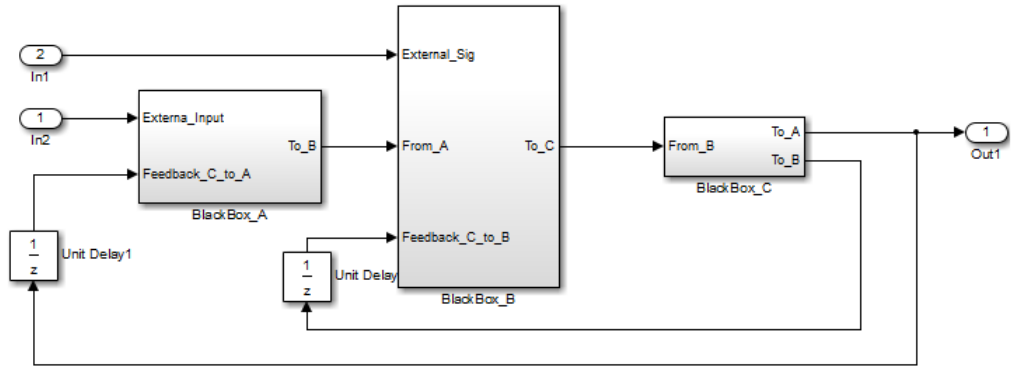
Add a unit delay between BlackBox_B and BlackBox_C

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 following example, 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.



Additional Techniques to Help the Algebraic Loop Solver

If the Simulink software cannot solve the algebraic loop, the software reports an error. If the Simulink algebraic loop solver cannot solve the algebraic loop, use one of the following 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 3-54

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.

Changing Block Priorities Does Not Remove Algebraic Loops

During the updating phase of simulation, Simulink determines the order in which to execute the block methods during simulation. This block invocation ordering is the *sorted order*.

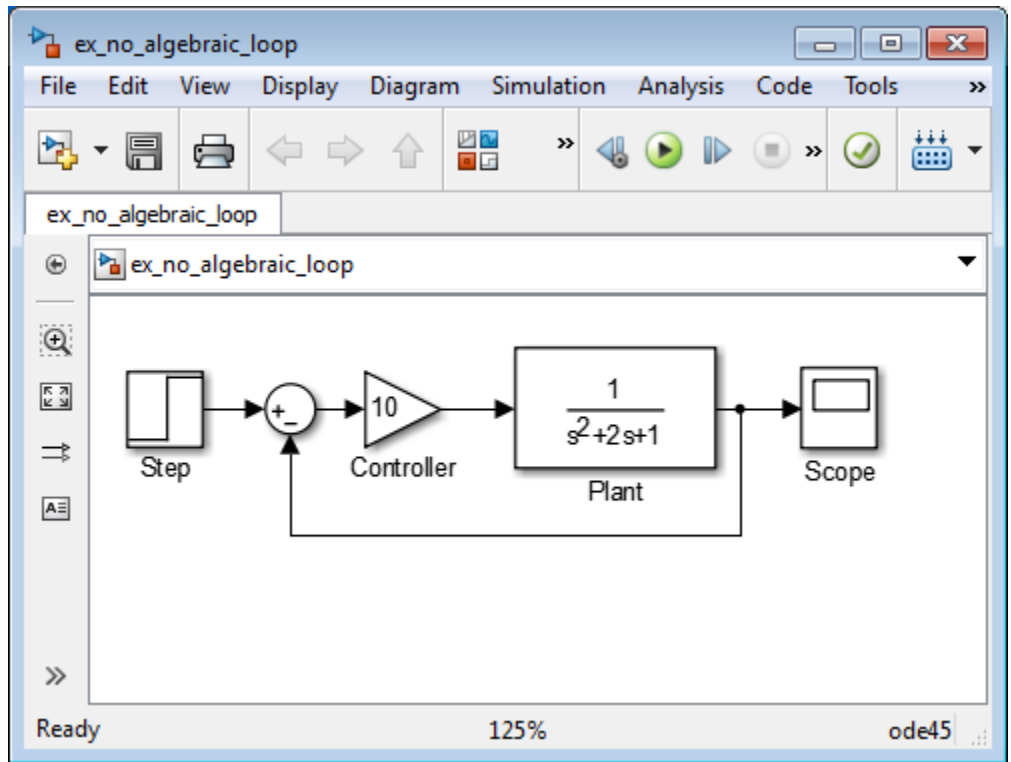
If you assign priorities to nonvirtual blocks to indicate to Simulink their execution order relative to other blocks, the algebraic loop solver does not honor these priorities when attempting to solve any algebraic loops.

Artificial Algebraic Loops

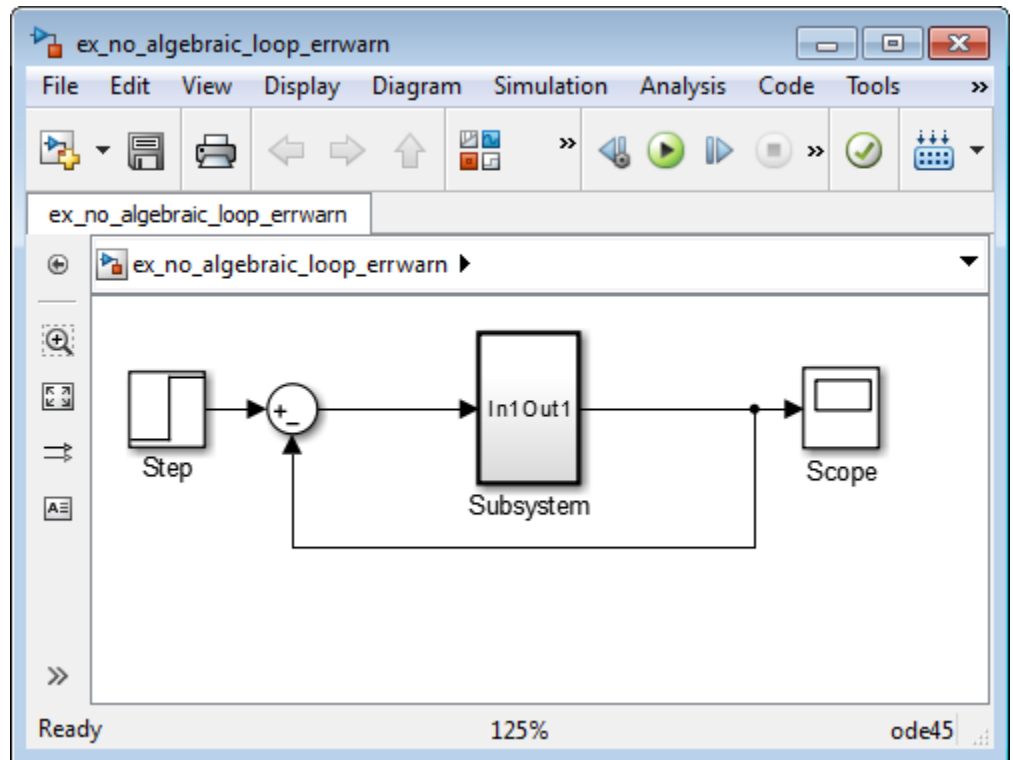
- “What Is an Artificial Algebraic Loop?” on page 3-55
- “Eliminating Artificial Algebraic Loops Caused by Atomic Subsystems” on page 3-57
- “Bundled Signals That Create Artificial Algebraic Loops” on page 3-58
- “Model and Block Parameters for Diagnosing and Eliminating Artificial Algebraic Loops” on page 3-63
- “Block Reduction and Artificial Algebraic Loops” on page 3-64
- “How Simulink Eliminates Artificial Algebraic Loops” on page 3-69
- “When Simulink Cannot Eliminate Artificial Algebraic Loops” on page 3-76
- “Managing Large Models with Artificial Algebraic Loops” on page 3-78

What Is an Artificial Algebraic Loop? 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 an algebraic constraint. When you create an atomic subsystem, all Inport blocks are direct feedthrough, resulting in an algebraic loop.

The following model does not contain an algebraic loop. The model simulates without error.



Suppose you enclose the Controller and Plant blocks in a subsystem and, in the Subsystem Parameters dialog box, select **Treat as atomic unit** to make the subsystem atomic.



When simulating this model, Simulink detects an algebraic loop because the subsystem is direct feedthrough, even though the path within the atomic subsystem is not direct feedthrough.

In the Configuration Parameters dialog box, on the main Diagnostics pane, if you set the **Algebraic loop** parameter to error, Simulink stops the simulation with an algebraic loop error.

Eliminating Artificial Algebraic Loops Caused by Atomic Subsystems.

One way to eliminate an artificial algebraic loop caused by an atomic subsystem is to convert the atomic subsystem to a virtual subsystem. Doing so 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 algebraic loop solver terminates after one

iteration. The algebraic loop is automatically solved because there is no algebraic constant. After you make the subsystem virtual, during simulation, Simulink does not invoke the algebraic loop solver.

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 Click **OK** to save the changes and close the dialog box.
- 5 Save the model.

If you replace the atomic subsystem with a virtual subsystem and the simulation still fails with an algebraic loop error, you probably have one of the following elsewhere in your model:

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

Examine your model, try to identify the location of the algebraic loop, and use some of the techniques described in this section to solve the loop.

Bundled Signals That Create Artificial Algebraic Loops. Some models bundle signals together. This bundling might cause Simulink to detect an algebraic loop, even when an algebraic constraint does not exist. If you redirect one or more signals, you might remove the artificial algebraic loop.

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

- Output q_1 is the rate of the fluid flow into the tank from the pump.
- Output h_2 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:

Parameters

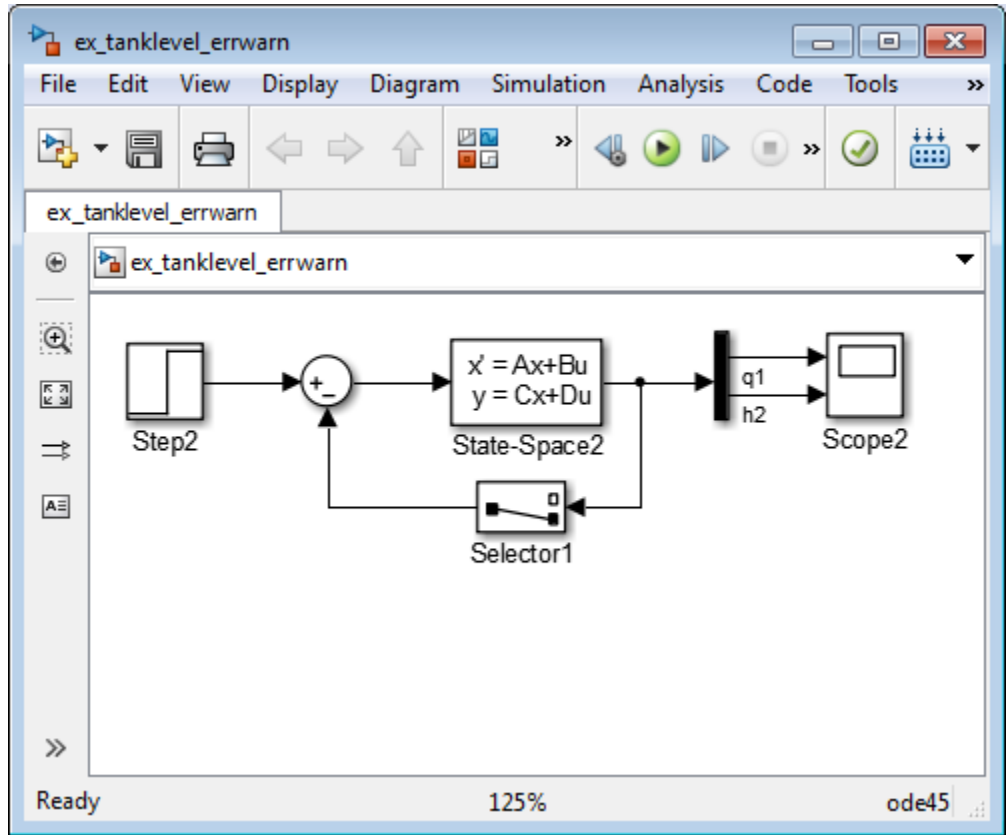
A:

B:

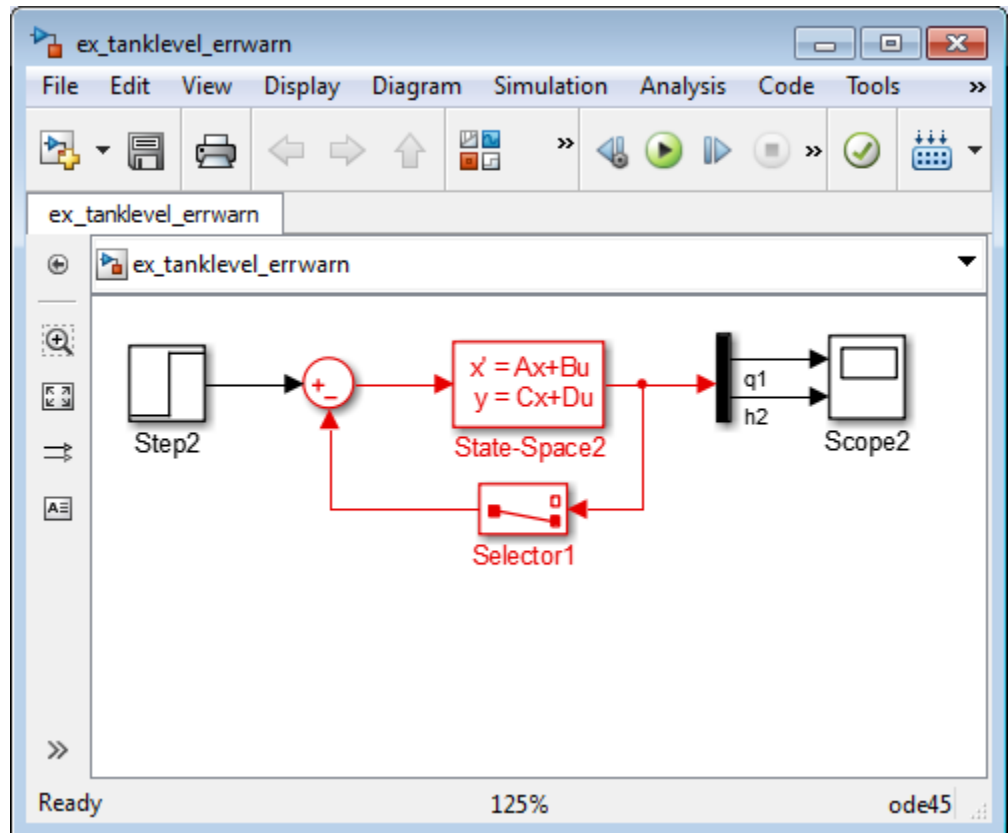
C:

D:

- The output from the State-Space block is a vector that contains q_1 and h_2 .



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:

Parameters

A:

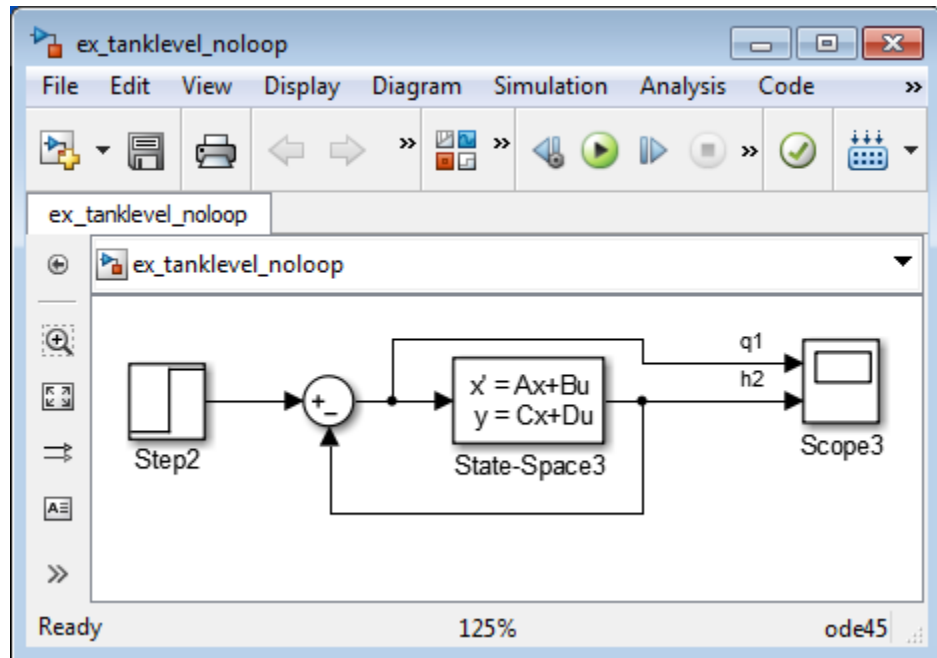
B:

C:

D:

- 2** Pass q_1 directly to the Scope instead of through the State-Space block.

Now, the input (q_1) 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 for Diagnosing and Eliminating 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 Configuration Parameters dialog box, on the **Model Referencing** pane, select **Minimize algebraic loop occurrences**
- **Minimize algebraic loop** parameter — If the **Minimize algebraic loop occurrences** parameter has no effect, specifies what diagnostic action Simulink takes.

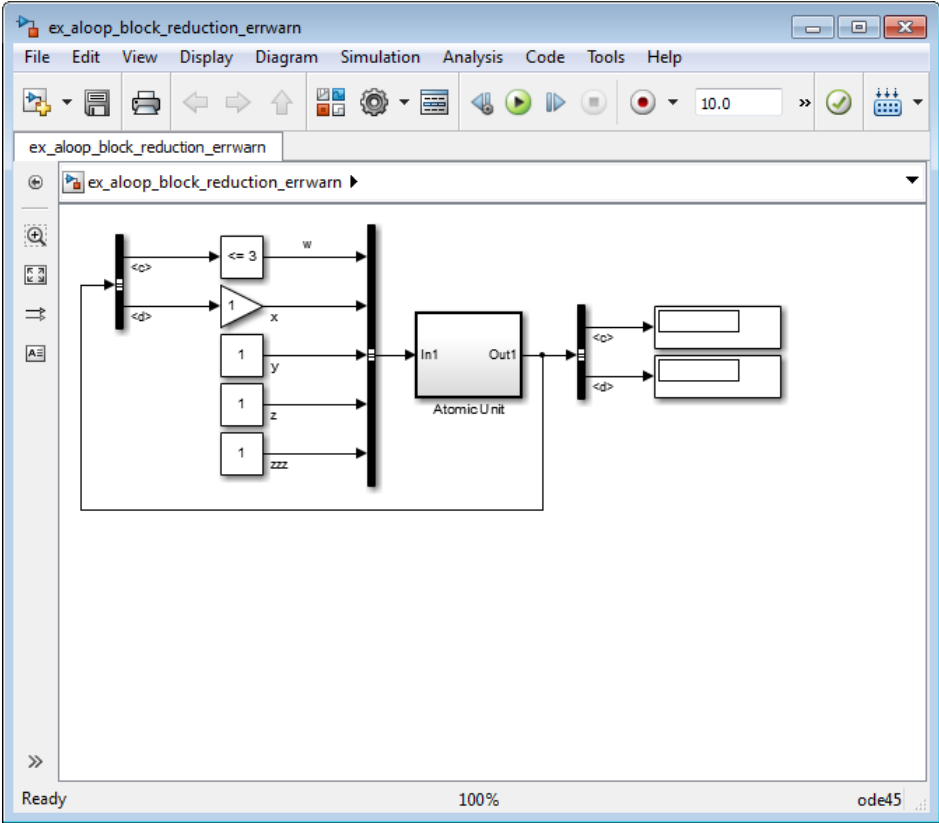
The **Minimize algebraic loop** parameter is in the Configuration Parameters dialog box, on the main **Diagnostics** pane. The diagnostic actions for this parameter are:

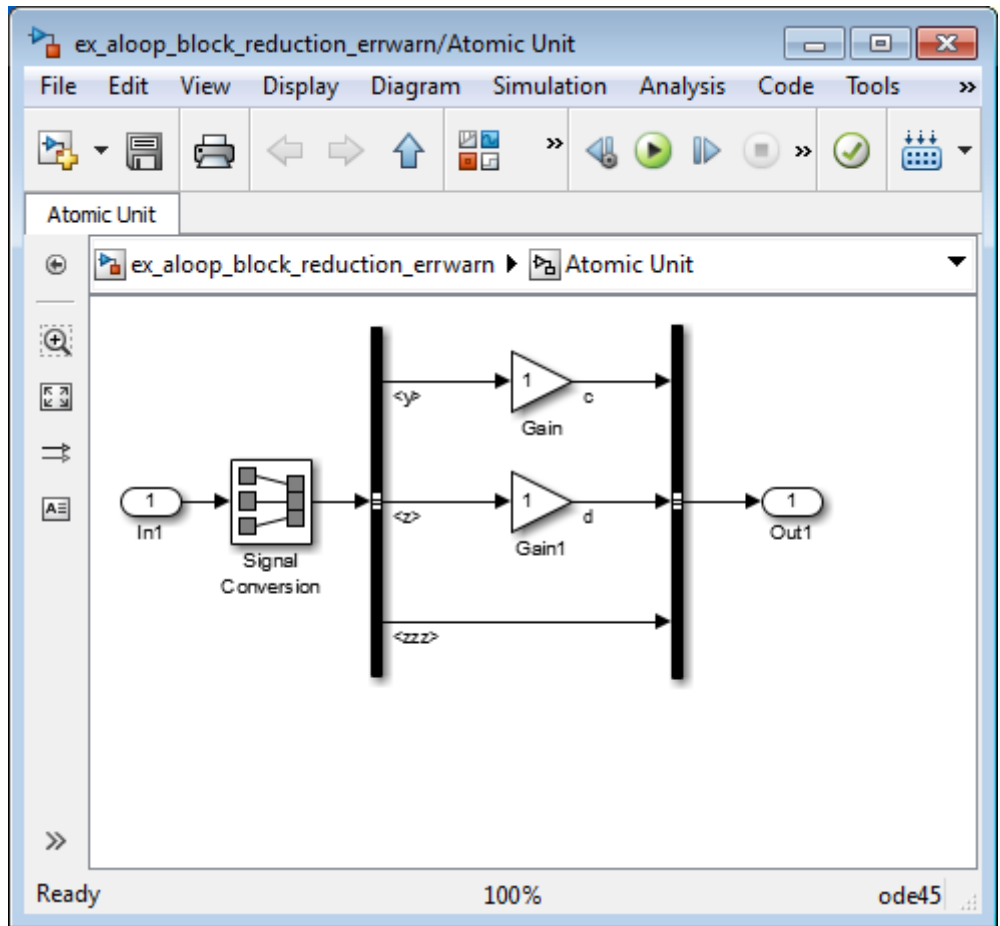
Setting	Simulation Response
none	Simulink takes no action.
warning	Simulink displays a warning that the Minimize algebraic loop occurrences parameter has no effect.
error	Simulink terminates the simulation and displays an error that the Minimize algebraic loop occurrences parameter has no effect.

Block Reduction and Artificial Algebraic Loops. When you enable the **Block reduction** optimization in the Configuration Parameters dialog box, 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 generated code.

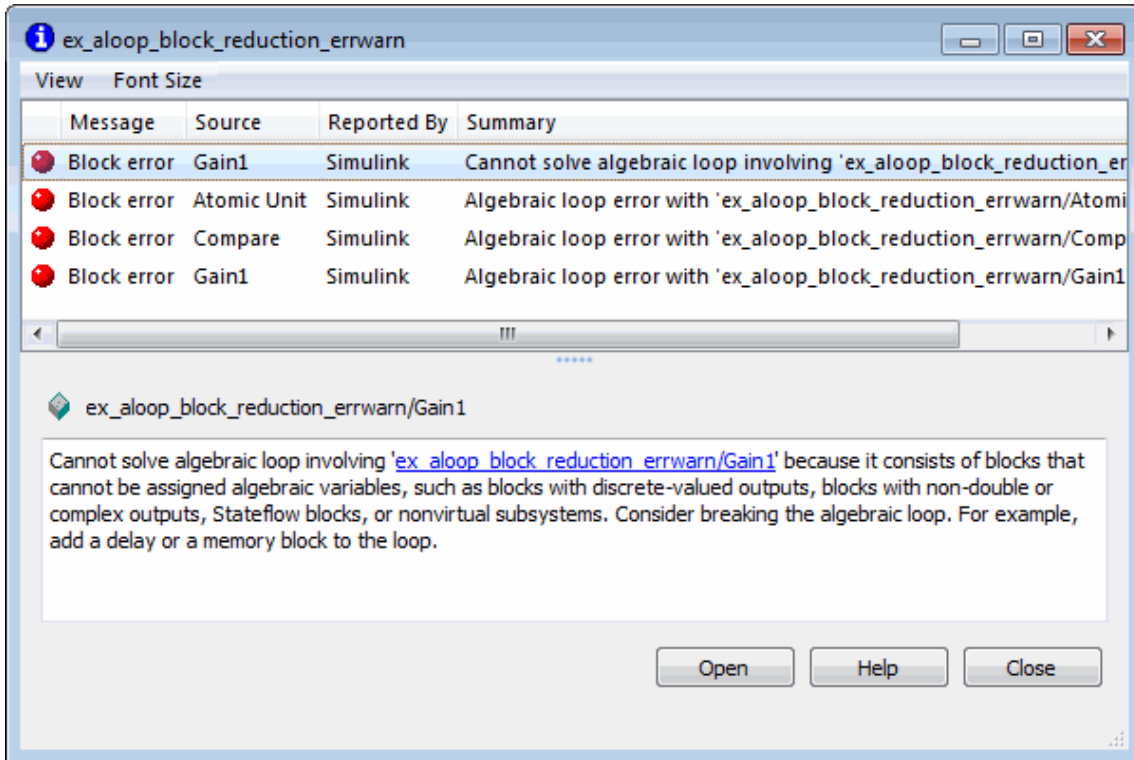
Enabling block reduction may also help Simulink solve artificial algebraic loops.

Consider the following example model.

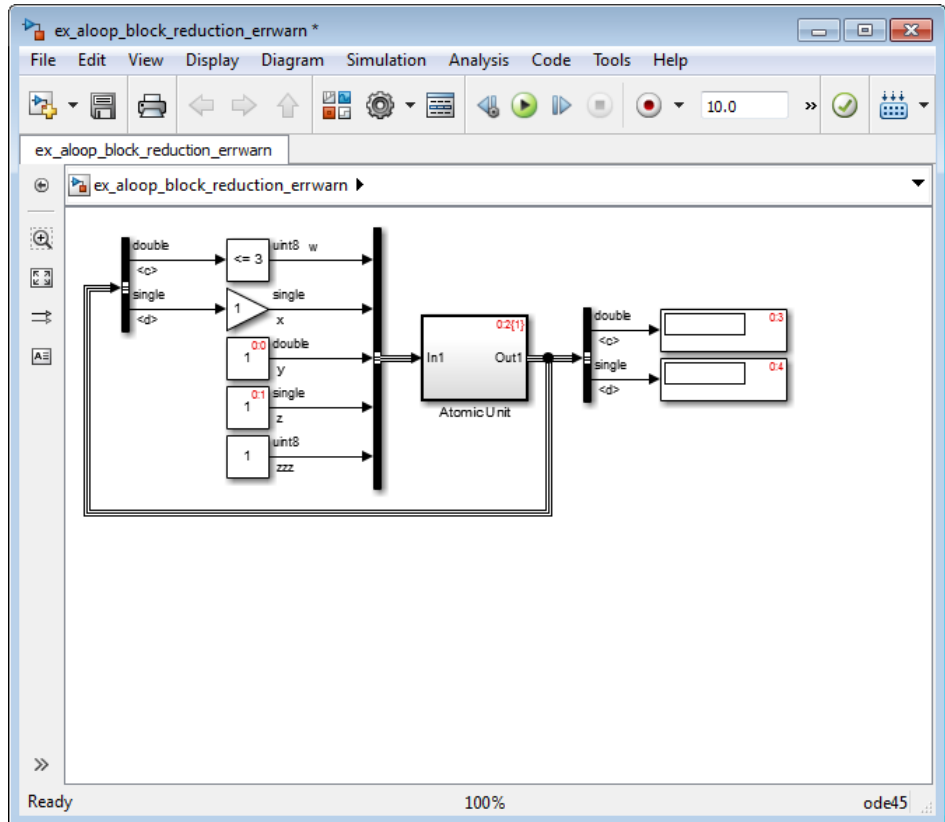




Initially, block reduction is turned off. When you simulate this model, the Atomic Unit subsystem and Gain and Compare to Constant blocks are part of an algebraic loop that Simulink cannot solve.



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

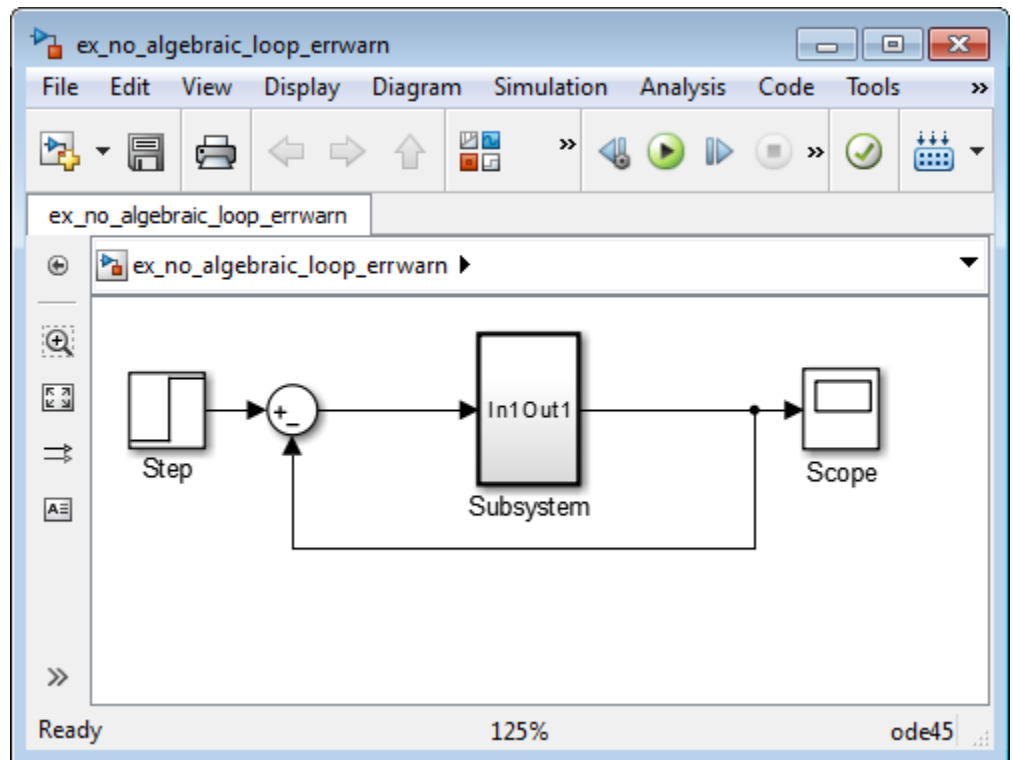


The Compare to Constant and Gain blocks have been eliminated from the model, so they no longer generate an algebraic loop error. The Atomic Unit subsystem generates a warning:

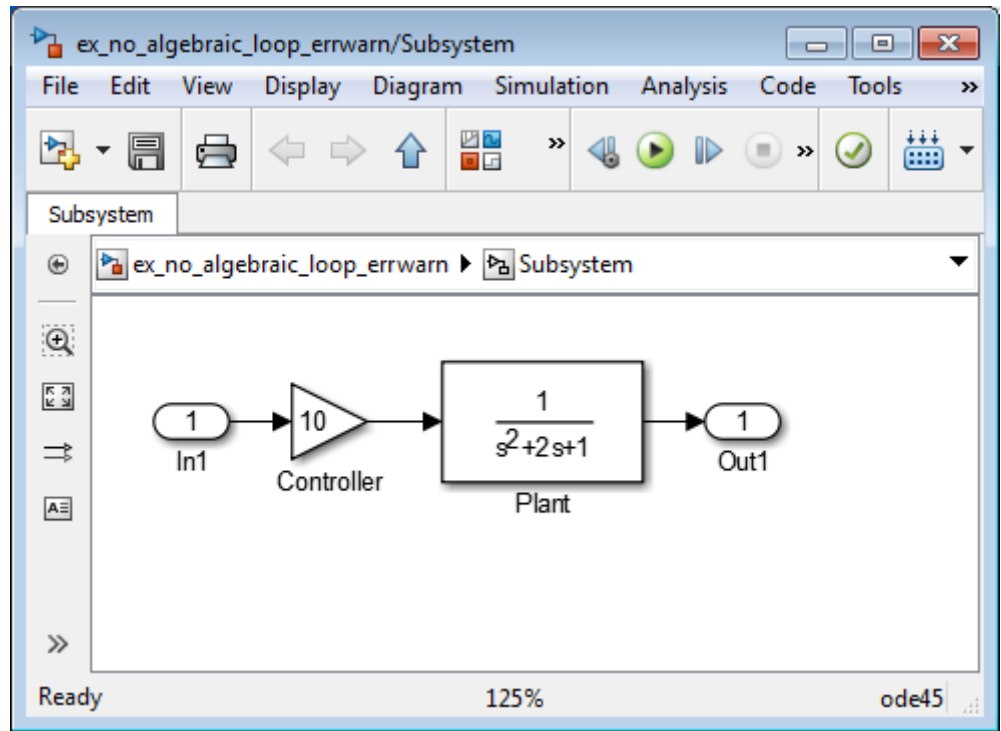
Warning: If the inport 'ex_aloop_block_reduction_errwarn/Atomic Unit/In1' of subsystem 'ex_aloop_block_reduction_errwarn/Atomic Unit' involves direct feedback, then an algebraic loop exists, which Simulink cannot remove. Consider clearing the 'Minimize algebraic loop occurrences' parameter to avoid this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

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

How Simulink Eliminates Artificial Algebraic Loops. When you enable **Minimize algebraic loop occurrences**, Simulink tries to eliminate artificial algebraic loops. This section describes this process, using this model, which 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.

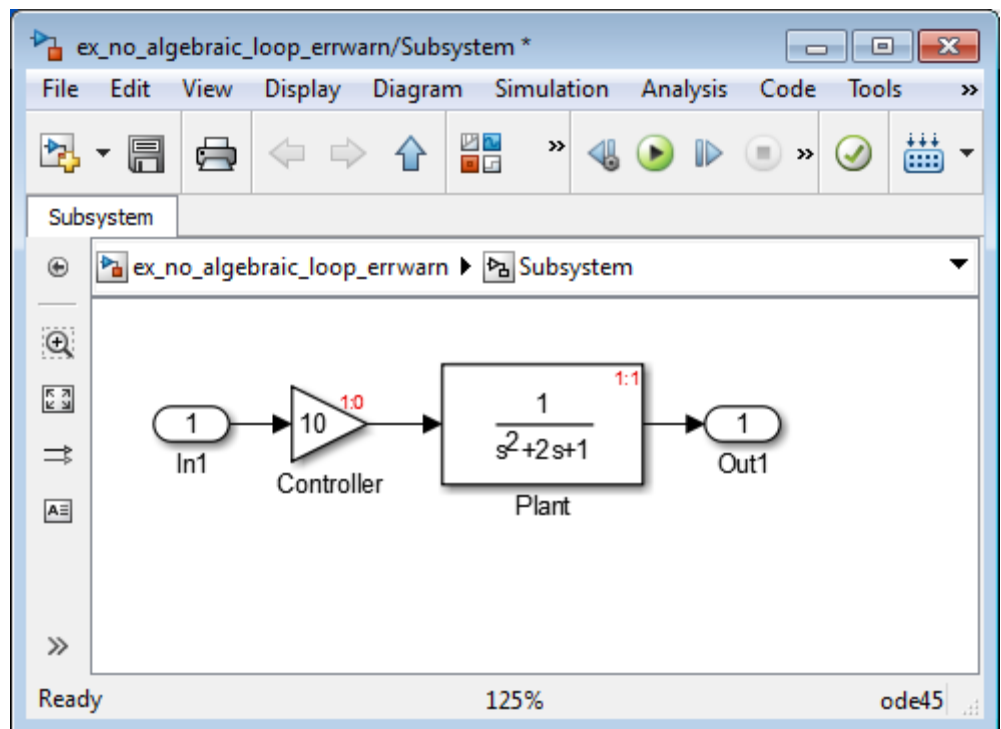
To see how Simulink tries to eliminate this algebraic loop, follow these steps:

- 1 Create the model from the preceding graphics, with the atomic subsystem that causes the artificial algebraic loop.
- 2 Select **Simulation > Model Configuration Parameters**.
- 3 In the Configuration Parameters dialog box, on the main **Diagnostics** pane, set the **Algebraic loop** parameter to warning or none.

- 4 On the **Data Import/Export** pane, make sure the **Signal logging** parameter is disabled. If signal logging is enabled, Simulink cannot eliminate artificial algebraic loops.
- 5 Click **OK**.
- 6 To display the sorted order for this model and the atomic subsystem, select **Display > Blocks > Sorted Execution Order**.

Reviewing the sorted order might 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 sorted order, see “Control and Display the Sorted Order” on page 23-36.

7 In the top-level model, right-click the subsystem and select **Subsystem Parameters**.

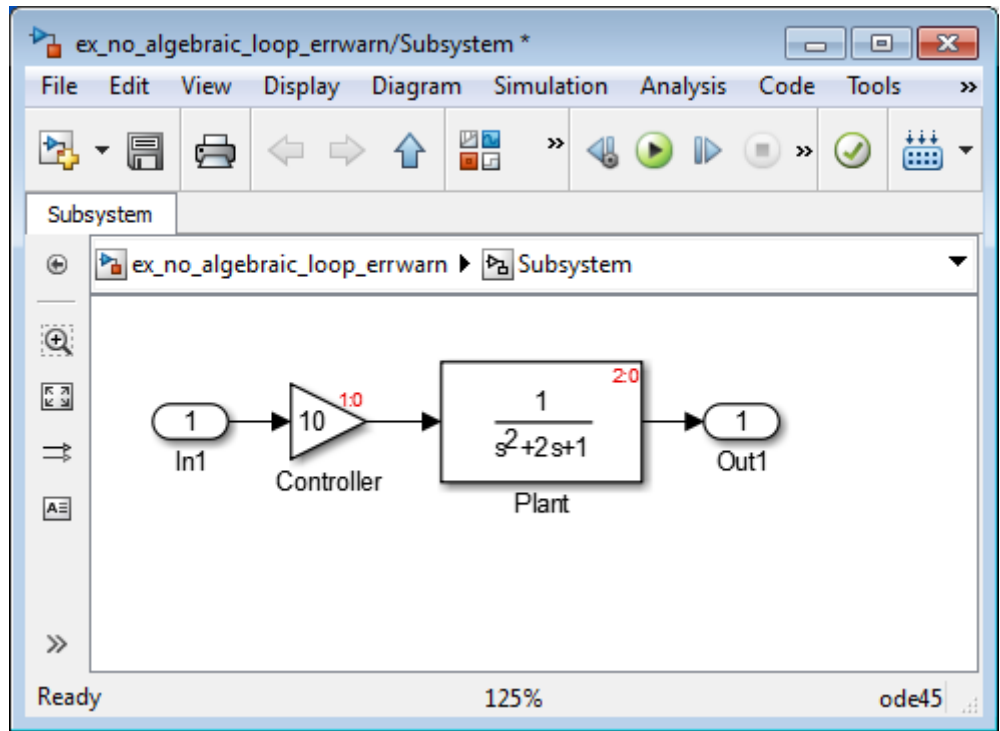
8 Select **Minimize algebraic loop occurrences**.

This parameter indicates to Simulink to try to eliminate the algebraic loop that contains the atomic subsystem when it simulates the model.

9 Click **OK**.

10 Click **Simulation > Update Diagram** to recalculate the sorted order.

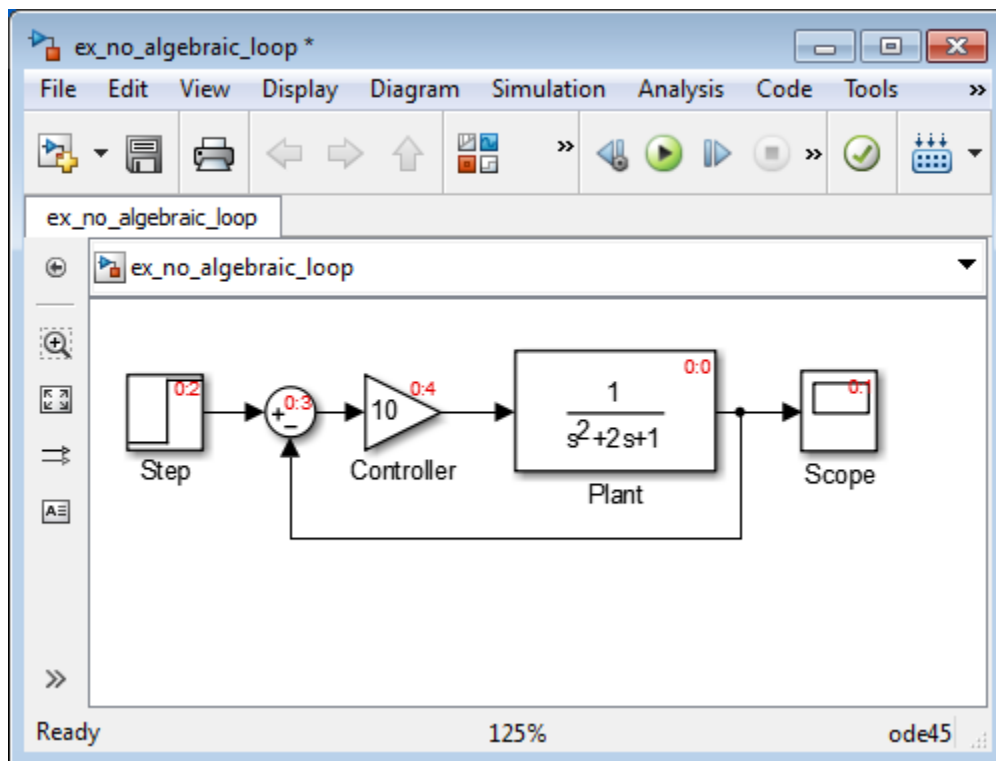
Now there are two levels of sorted order inside the subsystem: 1 and 2.



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

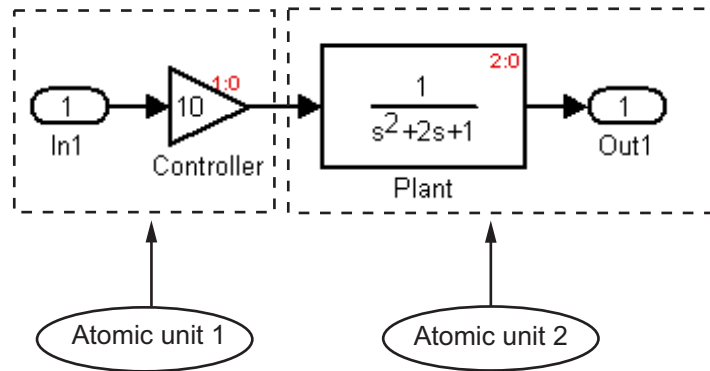
When you simulate a model, all the blocks first execute their `mdlOutputs` method, and then their `mdlDerivatives` and `mdlUpdate` methods.

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.



Note For more information about these methods, see “Block Methods” on page 3-16.

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



The following conditions are true:

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

The output of Atomic unit 1 is needed only by the `mdlDerivatives` or `mdlUpdate` methods of Atomic unit 2. 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:

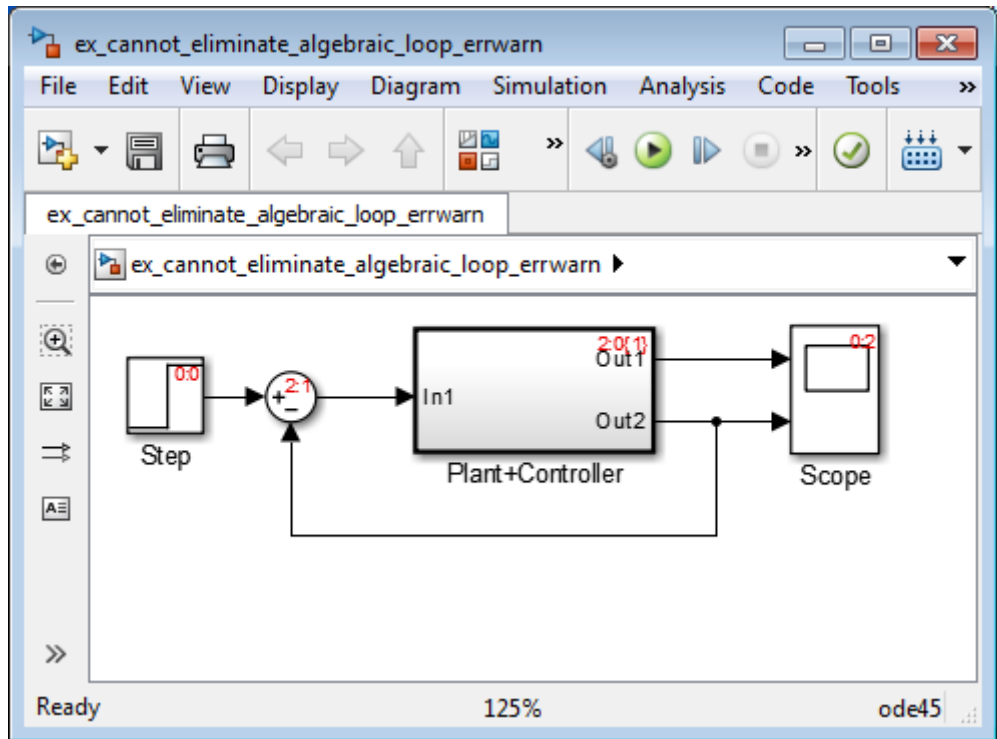
- `mdlOutputs` method of model
 - `mdlOutputs` method of Atomic unit 2
 - `mdlOutputs` methods of other blocks
- `mdlDerivatives` method of model
 - `mdlOutputs` method of Atomic unit 1
 - `mdlDerivatives` method of Atomic unit 2
 - `mdlDerivatives` method of other blocks

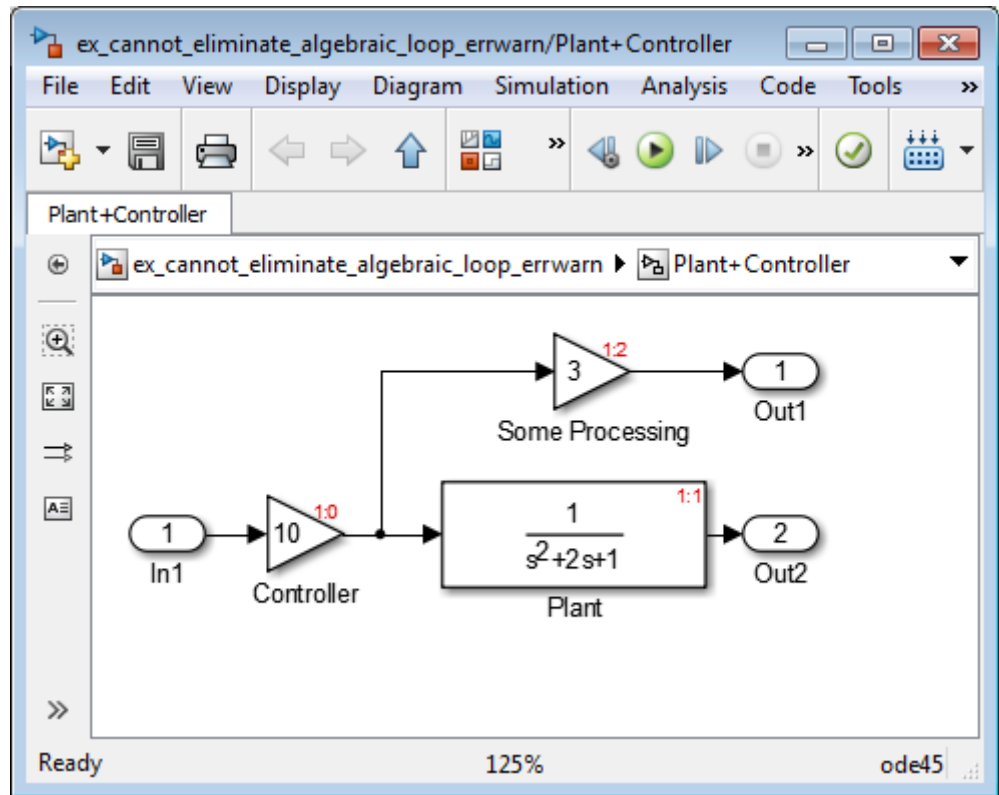
For the **Minimize algebraic loop occurrences** technique to be successful, the subsystem or referenced model must have a non-direct-feedthrough block connected directly to an Inport. Simulink can then set the `DirectFeedthrough`

property of the block Inport to false to indicate that the input port does not have direct feedthrough.

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 for an atomic subsystem if the Inport is connected to an Outport only through direct-feedthrough blocks.

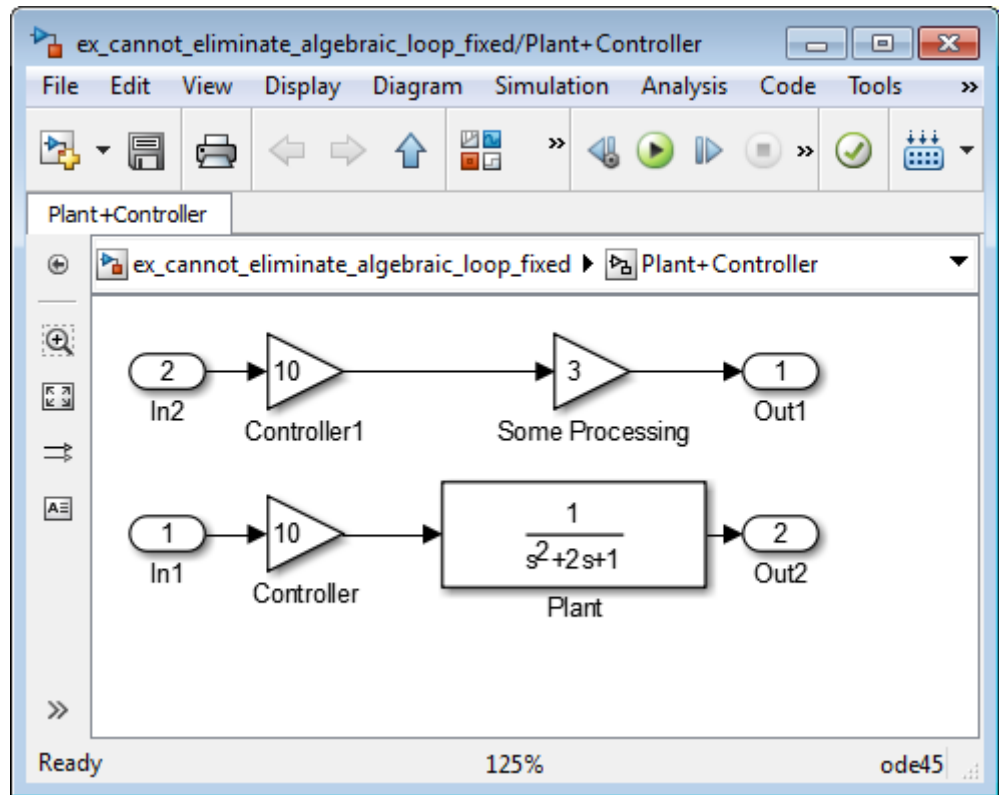
Consider the following model. The subsystem Plant+Controller causes an algebraic loop, but it has an extra Gain block and an extra output.





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 following model. Doing so makes In1 non-direct-feedthrough and In2 direct feedthrough, breaking the algebraic loop.



Managing Large Models with Artificial Algebraic Loops. For large models with algebraic loops, MathWorks recommends the following techniques:

- 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 might 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.

Modeling Dynamic Systems

- Chapter 4, “Creating a Model”
- Chapter 5, “Working with Sample Times”
- Chapter 6, “Referencing a Model”
- Chapter 7, “Create Conditional Subsystems”
- Chapter 8, “Modeling Variant Systems”
- Chapter 9, “Exploring, Searching, and Browsing Models”
- Chapter 10, “Managing Model Configurations”
- Chapter 11, “Configuring Models for Targets with Multicore Processors”
- Chapter 12, “Modeling Best Practices”
- Chapter 13, “Managing Projects”

Creating a Model

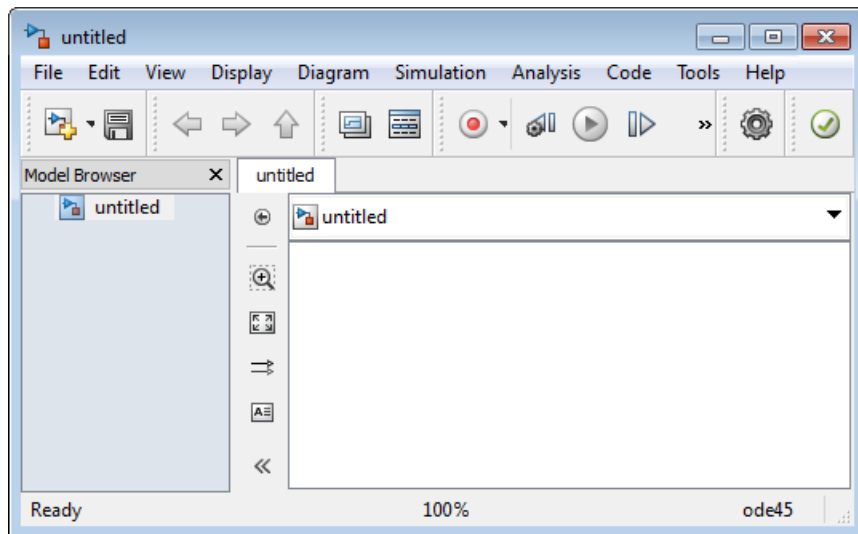
- “Create an Empty Model” on page 4-2
- “Populate a Model” on page 4-4
- “Select Modeling Objects” on page 4-6
- “Specify Block Diagram Colors” on page 4-8
- “Connect Blocks” on page 4-12
- “Align, Distribute, and Resize Groups of Blocks” on page 4-22
- “Annotate Diagrams” on page 4-23
- “Create a Subsystem” on page 4-37
- “Use Control Flow Logic” on page 4-46
- “Callbacks for Customized Model Behavior” on page 4-56
- “Model Callbacks” on page 4-58
- “Block Callbacks” on page 4-64
- “Port Callbacks” on page 4-72
- “Callback Tracing” on page 4-73
- “Model Workspaces” on page 4-74
- “Symbol Resolution” on page 4-83
- “Consult the Model Advisor” on page 4-88
- “Manage Model Versions” on page 4-115
- “Model Discretizer” on page 4-130

Create an Empty Model

To create an empty model:

- 1 Open the Library Browser, by typing `simulink` at the MATLAB command line.
- 2 In the Library Browser, select **File > New > Model** or click the **New model** button (📁).

The Simulink software creates an empty model in memory and displays the empty model in a Simulink Editor window.



Create a Model Template

When you create a model, Simulink uses defaults for many configuration parameters. For example, by default new models have a white canvas, the `ode45` solver, and a visible toolbar. If these or other defaults do not meet your needs, use the Simulink model construction functions listed in “Modeling Basics” to write a function that creates a model with the defaults that you prefer. For example, the following function creates a model that has a green canvas 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);
```

Populate a Model

In this section...
“Copy Blocks to Your Model” on page 4-4
“Browse Block Libraries” on page 4-4
“Search Block Libraries” on page 4-5
“Copy Blocks to Models” on page 4-5

Copy Blocks to Your Model

Use the Simulink Library Browser as the source for populating your model. In the Library Browser, browse and search blocks from built-in and user libraries. Copy blocks from the Library Browser to your model in the Simulink Editor.

To open the Library Browser, at the MATLAB command line, enter:

```
simulink
```

For details, see “Open the Simulink Library Browser”.

For information about creating your own libraries and adding them to the Library Browser, see “Block Libraries”.

Browse Block Libraries

The **Libraries** pane on the left displays a tree-structured folder of the block libraries on your system. Initially, the Simulink library is selected and its top level is open. You can scroll the pane and expand and collapse libraries and sublibraries to browse the libraries on your system and the blocks that the libraries contain.

The contents of the library that you select in the **Libraries** pane appear in the **Library** tab to the right of the pane. The contents can be sublibraries, blocks, or a mixture of the two. An icon and a name identifies each member of the selected library.

To open a sublibrary, use *one* of the following approaches:

- Select the library in the **Libraries** pane.
- Double-click the library in the **Library** tab.

To get help for a block:

- 1 Right-click the block.
- 2 From the context menu, select **Help**.

The help text and the context menu are the same as what appear when you right-click an instance of that block in a model.

Search Block Libraries

To search for library blocks whose names contain a specified character string:

- 1 In the Library Browser, in **Search** text field, enter the search character string.
- 2 Press **Return** or click **Search**.

The browser searches all libraries for blocks whose names match the specified string. The browser displays the results in the **Found** pane. The pane shows the blocks from each library separately.

By default, the search finds any substring and is not case-sensitive. To change these defaults or to enable use of MATLAB regular expressions in the **Search** field, click the **Library Browser Options** button and select the appropriate commands. Work with blocks that you find by searching just as you do with blocks that you find by selecting a library.

Copy Blocks to Models

To copy a block from the Library Browser into a model, drag and drop the library block into the model window at the location where you want to create the copy. Simulink copies the block to the model at the point that you select.

The resulting block retains a link to its source library. Updates to the source library automatically propagate to all of its linked copies. See “Libraries” for information about library links.

Select Modeling Objects

In this section...

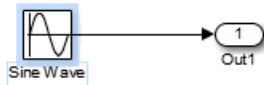
“Select an Object” on page 4-6

“Select Multiple Objects” on page 4-6

Select an Object

To select a modeling object (for example, a block or line), click it. When you place the cursor on a block, small black square handles appear at the corners of the block. Placing the cursor on a line highlights part of the line in blue.

Clicking a block or line highlights the whole object in blue. For example, the figure below shows a selected Sine Wave block.



Selecting an object by clicking it deselects any other selected objects.

Select Multiple Objects

To select more than one object, use *one* of these approaches:

- Select objects one at a time.
- Use a bounding box to select objects located near each other.
- Select the entire model.

Select Multiple Objects One at a Time

To select more than one object by selecting each object individually, hold down the **Shift** key and click each object that you want to select.

To deselect a selected object, click the object again while holding down the **Shift** key.

Select Multiple Objects Using a Bounding Box

To select more than one object in the same area of the window, draw a bounding box around the objects. This type of selection is also known as “marquee selection” and “drag selection.”

- 1 Define the starting corner of a bounding box. Position the cursor (pointer) at one corner of the box, then press and hold the left mouse button.
- 2 Drag the cursor to the opposite corner of the box. A blue rectangle encloses the selected blocks and lines.
- 3 Release the mouse button. Simulink selects all blocks and lines that are at least partially enclosed by the bounding box. For example, in the figure below, the bounding box enclosed at least part of the Sine Wave and the line from the Sine Wave block.



Select All Objects

To select all objects in the active window, select **Edit > Select All**.

Note To create a subsystem, you cannot select all blocks and signals. For more information, see “Create a Subsystem” on page 4-37.

Specify Block Diagram Colors

In this section...
“Set Block Diagram Colors Interactively” on page 4-8
“Platform Differences for Custom Colors” on page 4-9
“Choose a Custom Color” on page 4-9
“Define a Custom Color” on page 4-10
“Specify Colors Programmatically” on page 4-11

Set Block Diagram Colors Interactively

You can specify the foreground and background colors of any block or annotation in a diagram, as well as the background color of the diagram.

Type of Color	How to Set
Block diagram background	Select Diagram > Format > Canvas Color .
Block or annotation background	<ol style="list-style-type: none"> 1 Select the blocks and annotations. 2 Select Diagram > Format > Background Color
Block or annotation foreground	<ol style="list-style-type: none"> 1 Select the blocks and annotations. 2 Select Diagram > Format > Foreground Color

In all cases, you see a menu of color choices. Choose the desired color from the menu. If you select a color other than **Custom**, the background or foreground color of the diagram or diagram element changes to the selected color.

Platform Differences for Custom Colors

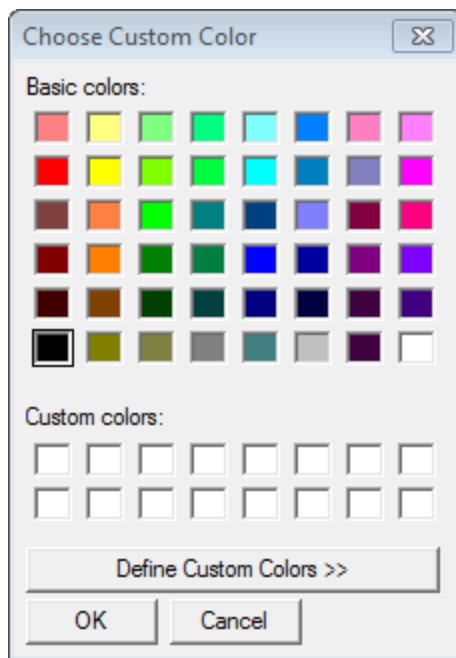
On Mac platforms, choosing **Custom** invokes the Mac color picker interface. Use the color picker to choose and define custom colors.

On Windows and Linux platforms, use the Simulink interface, as described in:

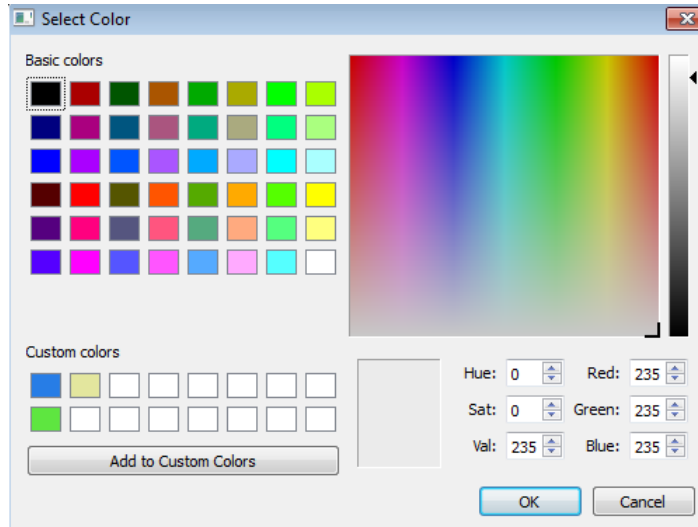
- “Choose a Custom Color” on page 4-9
- “Define a Custom Color” on page 4-10

Choose a Custom Color

If you choose **Custom**, and there are no custom colors already defined, Simulink displays the Choose Custom Color dialog box.



If you choose **Custom**, and there are custom colors already defined, Simulink displays the Select Color dialog box.



The Select Color dialog box displays a palette of basic colors and a palette of already defined custom colors. To choose a color from either palette, click the color and then click **OK**.

Define a Custom Color

To define the first custom color, in the Choose Custom Color dialog box, click the **Define Custom Colors** button.

The dialog box expands to display the Select Color dialog box. To define a custom color:

- 1 Specify the color using *one* of these approaches:
 - Enter the red, green, and blue components of the color as values between 0 (darkest) and 255 (brightest).
 - Enter hue, saturation, and luminescence components of the color as values in the range 0 to 255.
 - Move the hue-saturation cursor to select the hue and saturation of the desired color and the luminescence cursor to select the luminescence of the desired color.

- 2 Adjust the values until the color in the box to the right of the custom colors palette is the custom color that you want.
- 3 Click the **Add to Custom Colors** button.

To replace an existing custom color, select the custom color in the custom color palette before defining the new custom color.

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
Block diagram background	ScreenColor
Block and annotation background	BackgroundColor
Block and annotation foreground	ForegroundColor

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

- Named color: '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]')
```

Connect Blocks

In this section...
“Automatically Connect Blocks” on page 4-12
“Manually Connect Blocks” on page 4-15
“Disconnect Blocks” on page 4-21

Automatically Connect Blocks

You can have the Simulink software connect blocks automatically. This eliminates the need to draw the connecting lines yourself. When connecting blocks, Simulink routes the lines around intervening blocks to avoid cluttering the diagram.

Autoconnect Two Blocks

When connecting two blocks with multiple ports, Simulink draws as many connections as possible between the two blocks.

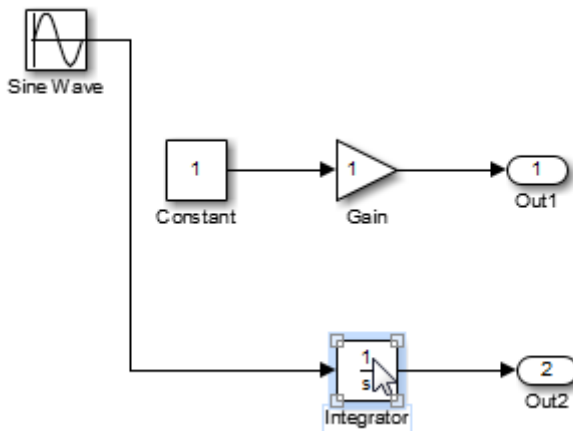
To autoconnect two blocks:

- 1 Select the source block. In this example, the Sine Wave block is the source block.



- 2** Hold down **Ctrl** and left-click the destination block. In this example, the Integrator block is the destination block.

The source block is connected to the destination block, and the lines are routed around intervening blocks if necessary.

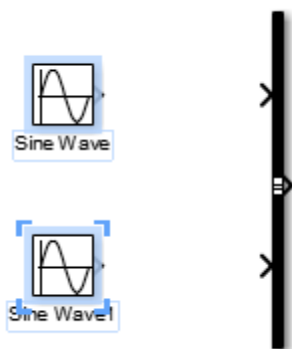


Note On Macintosh platforms, use the **command** key instead of **Ctrl**.

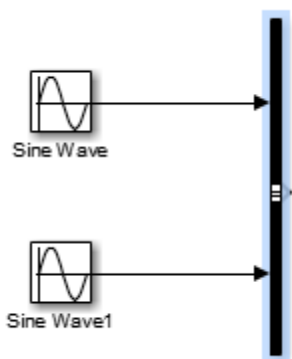
Connect Groups of Blocks

To connect a group of source blocks to a destination block:

- 1 Select the source blocks.

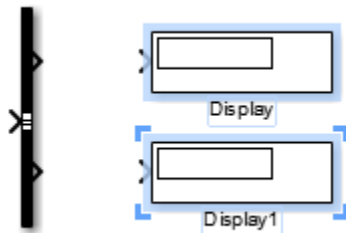


- 2 Hold down **Ctrl** and left-click the destination block.

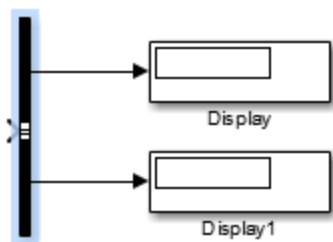


To connect a source block to a group of destination blocks:

- 1 Select the *destination* blocks.



- 2 Hold down **Ctrl** and left-click the *source* block.



Manually Connect Blocks

You can draw lines manually between blocks or between lines and blocks. You might want to do this if you need to control the path of the line or to create a branch line.

Draw a Line Between Blocks

You can create lines either from output to input ports, or from input to output ports. For example, to connect the output port of a Constant block to the input port of Gain block:

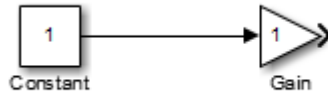
- 1 Position the cursor over the output port of the Constant block. You do not need to position the cursor precisely on the port.

The cursor shape changes to crosshairs.



- 2 Hold down the left mouse button.
- 3 Drag the cursor to the input port of the Gain block. Position the cursor on or near the port or in the block. If you position the cursor in the block, the line connects to the closest input port.
- 4 Release the mouse button. A connecting line with an arrow showing the direction of the signal flow replaces the port symbol.

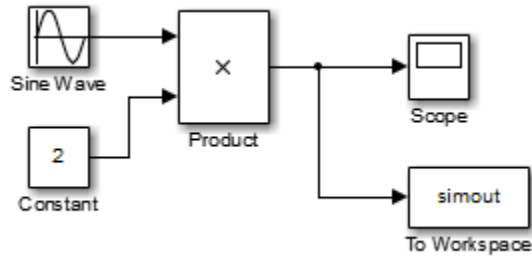
The arrow appears at the appropriate input port, and the signal is the same.



Draw a Branch Line

A *branch line* is a line that starts from an existing line and carries its signal to the input port of a block. Both the existing line and the branch line represent the same signal. Use branch lines to connect a signal to more than one block.

This example shows how to connect the Product block output to both the Scope block and the To Workspace block.



To add a branch line:

- 1 Position the cursor on the line where you want the branch line to start.
- 2 While holding down the **Ctrl** key, press and hold down the left or right mouse button.
- 3 Drag the cursor to the input port of the target block, then release the mouse button and the **Ctrl** key.

Draw Line Segments

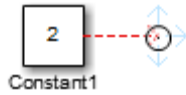
Manually draw line segments when you want to draw:

- Line segments differently than autoconnect feature draws the lines
- A line, before you copy the block to which the line connects

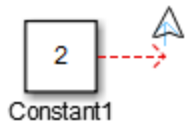
To draw a line segment:

- 1 Draw a line from the block port to an unoccupied area of the canvas. Release the mouse button where you want the line segment to end.

The cursor turns into a circle, and blue arrow guides appear.



- 2 For each additional line segment, position the cursor over the blue arrow guide that points in the direction in which you want to draw a line segment. The cursor turns into an empty arrowhead.



Tip To reroute the whole line instead of extending it, select the end of the line itself when the circle cursor is displayed and drag the line end to a new location.

- 3 Drag the cursor to draw the second line segment and release the mouse button to finish drawing the line.

Move a Line Segment

To move a line segment:

- 1 Position the cursor on the segment that you want to move.
- 2 Press and hold down the left mouse button.
- 3 Drag the cursor to the desired location and release the mouse button.

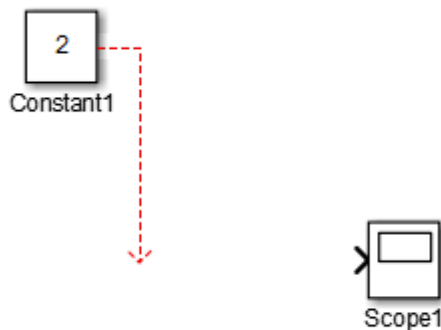
Draw a Diagonal Line

You cannot draw a single diagonal line between two ports. You can draw very short line segments connecting to each port, with a longer diagonal segment in the middle. For example, suppose you position two blocks as shown below:



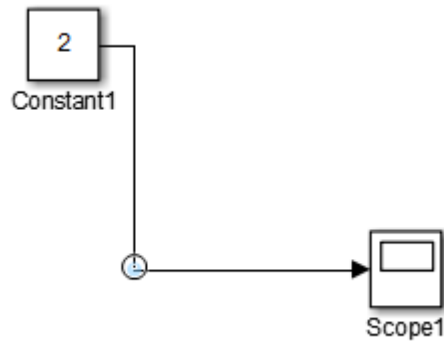
To approximate a diagonal line:

- 1 Draw a short line segment from the output port of the Constant block.
- 2 At the end of the first line segment, draw a second line segment.

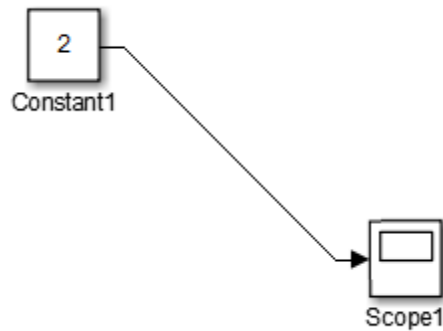


- 3 Draw a third line segment to connect to the Scope block.

- 4 Position the cursor at the bend of the second and third line segments. The cursor turns into a circle.



- 5 Hold the **Shift** key down and drag the cursor to make the second line segment a diagonal.

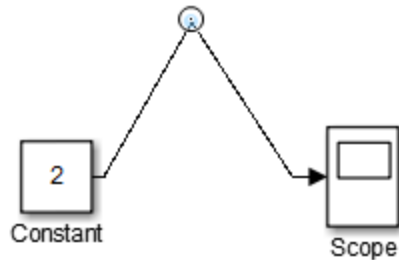


Move a Line Vertex

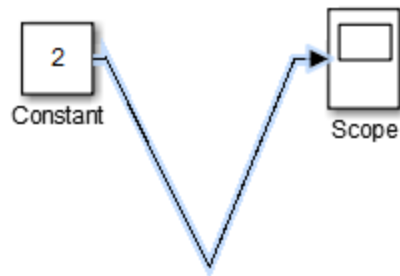
To move a vertex of a line:

- 1 Position the cursor on the vertex, then press and hold down the left mouse button.

The cursor changes to a circle.



2 Drag the vertex to the desired location.



3 Release the mouse button.

Insert a Block in a Line

You can insert a block in a line, if the block has only one input and one output.

1 Drag the block over the line in which you want to insert the block.

2 Release the mouse button. Simulink inserts the block for you at the point where you drop the block.

Disconnect Blocks

To disconnect a block from its connecting lines, hold down the **Shift** key, then drag the block to a new location.

Align, Distribute, and Resize Groups of Blocks

To align, distribute, or resize a group of blocks:

- 1 Select the blocks that you want to align, as described in “Select Multiple Objects” on page 4-6.

Simulink highlights one of the selected blocks. Simulink uses the highlighted block as the reference for aligning the other selected blocks. For example, in the following model, the Constant block is the alignment reference block.



One of the selected blocks displays empty selection handles. The model editor uses this block as the reference for aligning the other selected blocks. If you want another block to serve as the alignment reference, click that block.

- 2 From the **Diagram > Arrange** menu, select an alignment, distribution, or sizing option. For example, select **Align Top** to align the top edges of the selected blocks with the top edge of the reference block.



Annotate Diagrams

In this section...

“Add and Edit Annotations” on page 4-23

“Summary of Annotations Properties Dialog Box” on page 4-28

“Annotation Callback Functions” on page 4-29

“Associate Click Functions with Annotations” on page 4-30

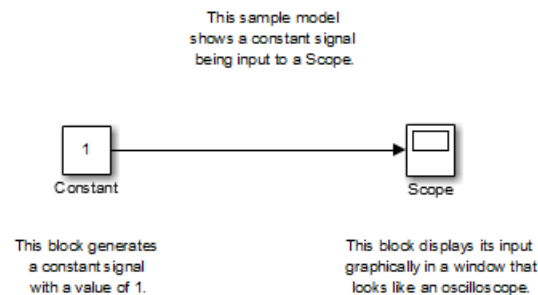
“Annotations API” on page 4-33

“TeX Formatting Commands in Annotations” on page 4-33

“Create Annotations Programmatically” on page 4-35

Add and Edit Annotations

Annotations provide textual information to document a model. You can add an annotation to any unoccupied area of your block diagram.




Add an Annotation

To add an annotation:

- 1 Double-click an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point.

- 2 Type the annotation contents. Each line is centered in the rectangle that surrounds the annotation.

Alternatively, from the Simulink Editor palette (to the left of the model), select the **Annotation** button () and either drag the cursor into the model window or click in the model window.

Move an Annotation

To move an annotation, drag it to a new location.

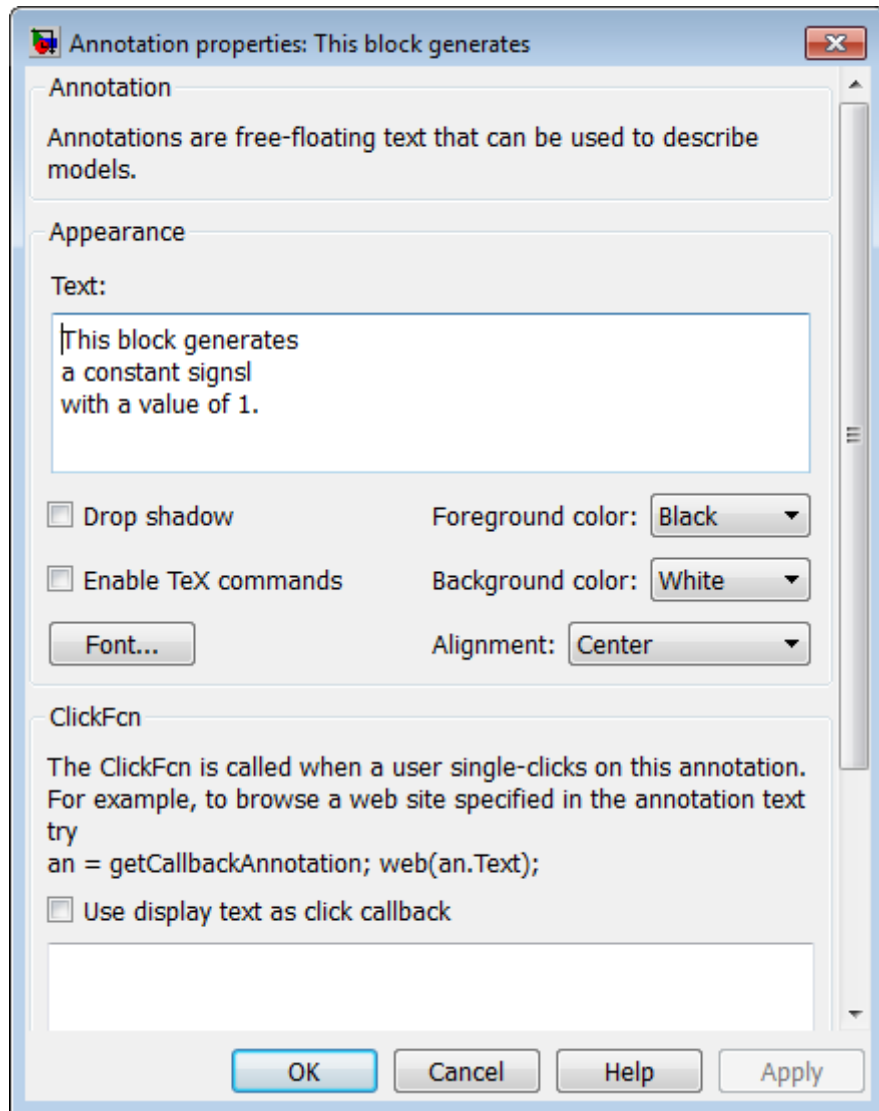
Annotations Properties Dialog Box

As an alternative to directly editing an annotation, you can use the Annotation properties dialog box to specify the contents and format of the currently selected annotation and to associate a click function with the annotation. The Annotations properties dialog box consolidates the options for specifying the content and appearance of an annotation.

To display the Annotation properties dialog box for an annotation:

- 1 Select an annotation.
- 2 Select **Diagram > Properties**.

The Annotation properties dialog box opens.



Edit an Annotation

To edit an annotation directly, select the annotation.

- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To edit annotation text using the Annotations properties dialog box, edit the **Text** field.

Change Annotation Font

To change the annotation font:

- 1 Select the annotation.
- 2 Select **Diagram > Format > Font Style**.
- 3 Select a font and size from the dialog box.

To change the annotation font using the Annotations properties dialog box, press the **Font** button.

Use TeX commands

To use TeX (LaTeX) commands in an annotation:

- 1 Select the annotation.
- 2 Select **Diagram > Format > Enable TeX Commands**.
- 3 Enter TeX commands.

As an alternative, to enable the use of TeX commands using the Annotations properties dialog box, select **Enable TeX commands**.

For details, see “TeX Formatting Commands in Annotations” on page 4-33.

Text Alignment

To change the text alignment (left, center, or right) in an annotation:

- 1 Select the annotation.

2 Select **Diagram > Format > Text Alignment**.

3 Select an alignment option, such as **Center**.

As an alternative, to change the annotation text alignment using the Annotations properties dialog box, use the **Alignment** field.

Background Color or Text Color

To change the background color or text color of an annotation:

1 Select the annotation.

2 Set the annotation background color by selecting **Diagram > Format > Background**.

3 Set the annotation text color by selecting **Diagram > Format > Foreground**.

As an alternative, to change the background color or text color using the Annotations properties dialog box:

- To specify the color of the annotation box, use the **Background color** field.
- To specify the annotation text color of the annotation box, use the **Foreground color** field.

Border or Drop Shadow

To specify a border for the annotation:

1 Select the annotation.

2 Select the annotation background color by selecting **Diagram > Format > Annotation Border**.

To specify a drop shadow for the annotation (to give it a 3-D appearance), in the Annotations properties dialog box, select **Drop shadow**.

Delete an Annotation

To delete an annotation, do *one* of the following:

- Draw a selection box that includes part of the annotation and press **Delete**.
- Select all of the text in the annotation, press **Delete**, and move the cursor.

Summary of Annotations Properties Dialog Box

The dialog box includes the following controls.

Text

Displays the current text of the annotation. Edit this field to change the annotation text.

Enable TeX commands

Enables the use of TeX formatting commands in this annotation. For details, see “TeX Formatting Commands in Annotations” on page 4-33.

Font

Displays a font chooser dialog box. Use the font chooser to change the font used to render the annotation’s text.

Foreground color

Specifies the color of the annotation text.

Background color

Specifies the color of the background of the bounding box of the annotation.

Alignment

Specifies the alignment of the annotation text relative to its bounding box.

ClickFcn

Specifies MATLAB code to be executed when a user single-clicks this annotation. Simulink stores the code entered in this field with the model. For details, see “Associate Click Functions with Annotations” on page 4-30.

Use display text as click callback

Treats the **Text** field as the annotation click function. The specified text must be a valid MATLAB expression comprising symbols that are defined in the MATLAB workspace when the user clicks this annotation. For details, see “Associate Click Functions with Annotations” on page 4-30. Selecting this option disables the **ClickFcn** edit field.

Annotation Callback Functions

You can make an annotation interactive by adding a callback. For example, you can use an annotation click callback function to set up a link from annotation text.

You can associate the following callback functions with annotations.

Click Function

A click function is a MATLAB function that Simulink invokes when a user single-clicks an annotation. You can associate a click function with any model annotation (for details, see “Associate Click Functions with Annotations” on page 4-30).

You can use click functions to add hyperlinks and custom command “buttons” to a model. For example, you can use a click function to allow a user to display the values of workspace variables referenced by the model or to open related models simply by clicking on annotations displayed on the block diagram.

Simulink uses the color blue to display the text of annotations associated with click functions.

Load Function

This function is invoked when it loads the model containing the associated annotation. To associate a load function with an annotation, set the **LoadFcn** property of the annotation to the desired function (see “Annotations API” on page 4-33).

Delete function

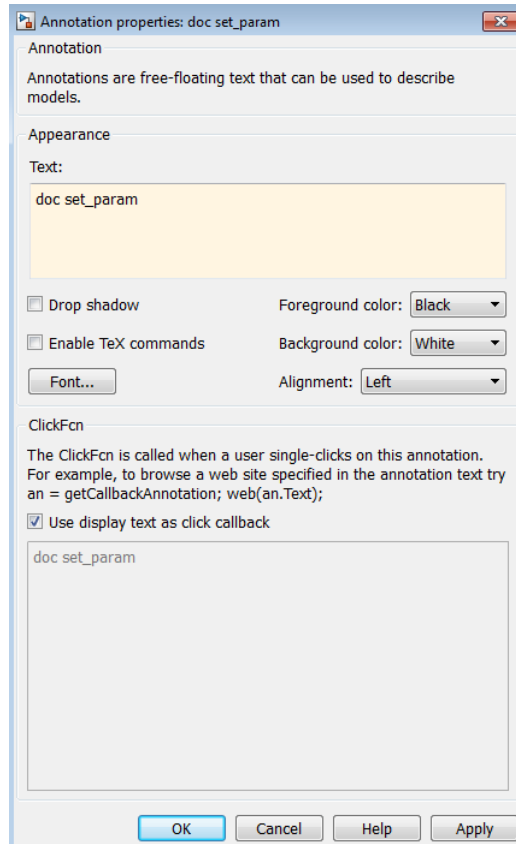
This function is invoked before deleting the associated annotation. To associate a delete function with an annotation, set the `DeleteFcn` property of the annotation to the desired function (see “Annotations API” on page 4-33).

Associate Click Functions with Annotations

Use one of two ways to associate a click function with an annotation with either the Annotation Properties dialog box (see “Summary of Annotations Properties Dialog Box” on page 4-28) or a separately defined function.

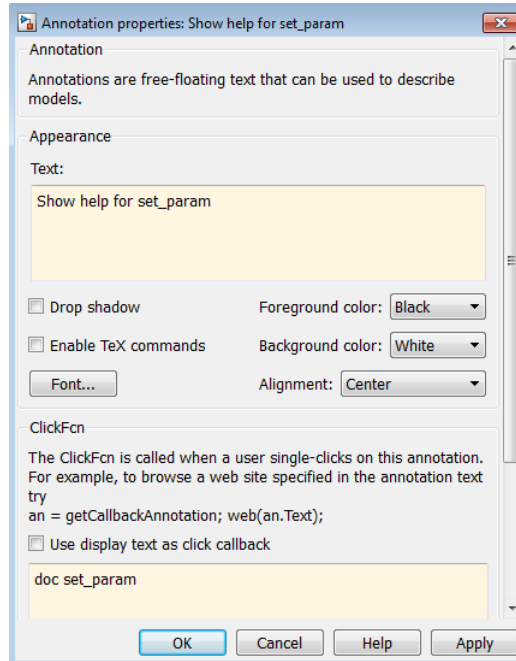
- Specify the annotation itself as the click function.

Enter a valid MATLAB expression in the **Text** field and check the **Use display text as click callback**.



- Specify a separately defined click function.

Enter the MATLAB code that defines the click function in the **ClickFcn** edit field.



Note You can also use MATLAB code to associate a click function with an annotation. See “Annotations API” on page 4-33 for more information.

Select and Edit Annotations Associated with Click Functions

If you associate an annotation with a click function, then you cannot select the annotation by clicking on it. Instead, use a boundary box to select the annotation.

To move the annotation, use the up, down, right, or left arrow keys.

Similarly, you cannot edit the annotation text by clicking on the text. To edit the annotation:

- 1** Drag select the annotation.
- 2** Right-click the selected annotation.
- 3** In the context menu, select **Properties**.
- 4** In the Properties dialog box, in the **Text** field, edit the text.

Annotations API

To use MATLAB code to get and set the properties of annotations, use:

- `Simulink.Annotation` class

Set the properties of annotations, by using MATLAB code. For example, annotation load functions (see “Load Function” on page 4-29).

- `getCallbackAnnotation` function

Get the `Simulink.Annotation` object for the annotation associated with the currently executing annotation callback function. Use this function to determine which annotation was clicked to invoke the current callback.

This function is also useful if you write a callback function in a separate MATLAB file contains multiple callback calls.

TeX Formatting Commands in Annotations

You can use TeX formatting commands to include mathematical and other symbols and Greek letters in block diagram annotations.

4 Creating a Model

Linearization of Double Pendulum

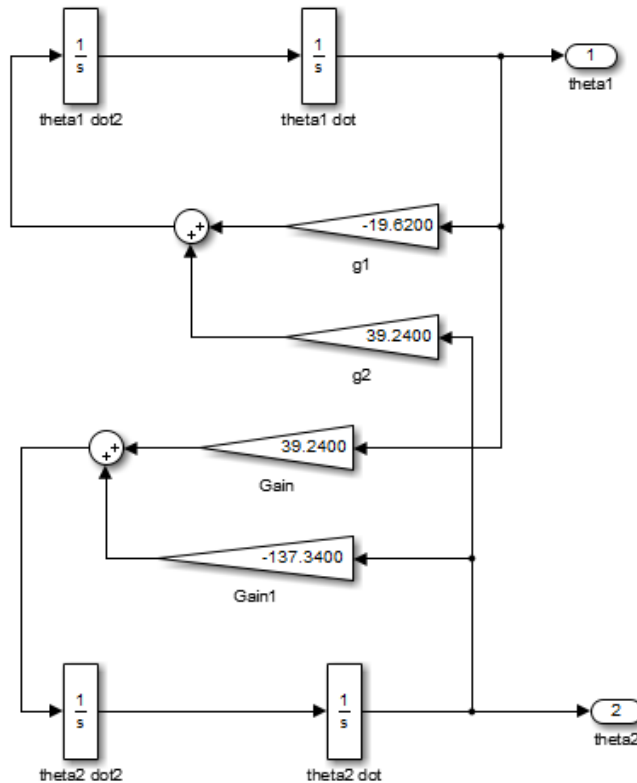
$$\theta_1'' = -19.6200\theta_1 + 39.2400\theta_2$$

$$\theta_2'' = 39.2400\theta_1 - 132.6603\theta_2$$

where

θ_1'' = position of top joint

θ_2'' = position of bottom joint



To use TeX commands in an annotation:

- 1 Select the annotation.
- 2 Select **Diagram > Format > Enable TeX Commands**.
- 3 Enter or edit the text of the annotation, using TeX commands where needed to achieve the desired appearance.

```

Linearization of Double Pendulum

\theta1" = -19.6200*\theta1 + 39.2400*\theta2
\theta2" = 39.2400*\theta1 -132.6603*\theta2

where

\theta1 = position of top joint
\theta2 = position of bottom joint

```

See “Mathematical Symbols, Greek Letters, and TeX Characters” in the MATLAB documentation for information about the supported TeX formatting commands.

- 4 Deselect the annotation by clicking outside it or press the **Esc** key.

The text reflects the TeX formatting.

```

Linearization of Double Pendulum

θ1" = -19.6200*θ1 + 39.2400*θ2
θ2" = 39.2400*θ1 -132.6603*θ2

where

θ1 = position of top joint
θ2 = position of bottom joint

```

Create Annotations Programmatically

To create annotations at the command line or in a MATLAB program, use the `add_block` command.

```

add_block('built-in/Note','path/text','Position', ...
[center_x, 0, 0, center_y]);

```

where *path* is the path of the diagram to be annotated, *text* is the text of the annotation, and `[center_x, 0, 0, center_y]` is the position of the center of the annotation in pixels relative to the upper left corner of the diagram. For example:

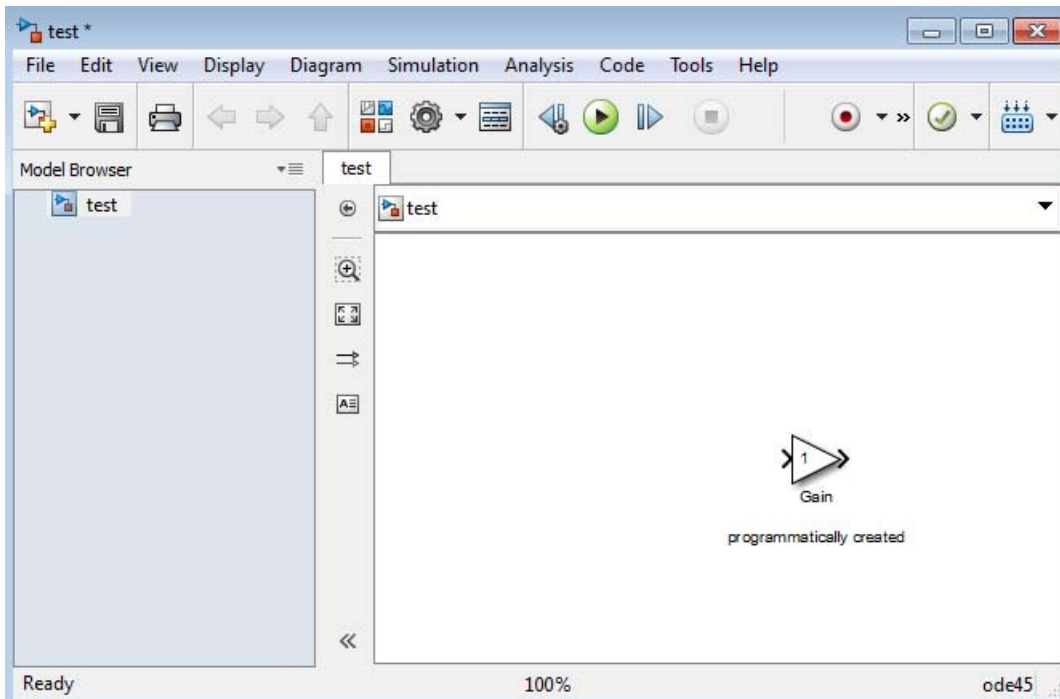
```

new_system('test')

```

```
open_system('test')
add_block('built-in/Gain', 'test/Gain', 'Position', ...
[260, 125, 290, 155])
add_block('built-in/Note', 'test/programmatically created', ...
'Position', [550 0 0 180])
```

The code creates the following model, which includes an annotation.



To delete an annotation, use the `find_system` command to get the annotation handle. Then use the `delete` function to delete the annotation. For example:

```
delete(find_system(gcs, 'FindAll', 'on', 'type', 'annotation'));
```

Create a Subsystem

In this section...

- “Subsystem Advantages” on page 4-37
- “Two Ways to Create a Subsystem” on page 4-38
- “Create a Subsystem with a Subsystem Block” on page 4-38
- “Group Existing Blocks into a Subsystem” on page 4-38
- “Subsystem Execution” on page 4-40
- “Navigate Model Hierarchy” on page 4-40
- “Label Subsystem Ports” on page 4-43
- “Control Access to Subsystems” on page 4-43
- “Control Subsystem Behavior with Callbacks” on page 4-44
- “Interconvert Subsystems and Block Diagrams” on page 4-45
- “Empty Subsystems and Block Diagrams” on page 4-45

Subsystem Advantages

A subsystem is a set of blocks that you replace with a single block called a Subsystem block. As your model increases in size and complexity, you can simplify it by grouping blocks into subsystems. Using subsystems has these advantages:

- Helps reduce the number of blocks displayed in your model window.
- Keeps functionally related blocks together.
- Establishes a hierarchical block diagram, where a Subsystem block is on one layer and the blocks that make up the subsystem are on another.

For additional information about how subsystems compare to other Simulink componentization techniques, see “Componentization Guidelines” on page 12-17.

Two Ways to Create a Subsystem

You can create a subsystem in two ways:

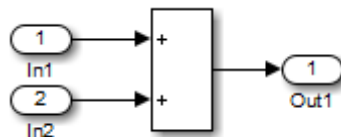
- Add a Subsystem block to your model, then open that block and add the blocks that it contains to the subsystem window.
- Add the blocks that make up the subsystem, then group those blocks into a subsystem.

Create a Subsystem with a Subsystem Block

To create a subsystem before adding the blocks that it contains, add a Subsystem block to the model, then add the blocks that make up the subsystem.

- 1 Copy the Subsystem block from the Ports & Subsystems library into your model.
- 2 Open the Subsystem block by double-clicking it.
- 3 In the empty Subsystem window, create the subsystem. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output.

For example, this subsystem includes a Sum block and Inport and Outport blocks to represent input to and output from the subsystem.



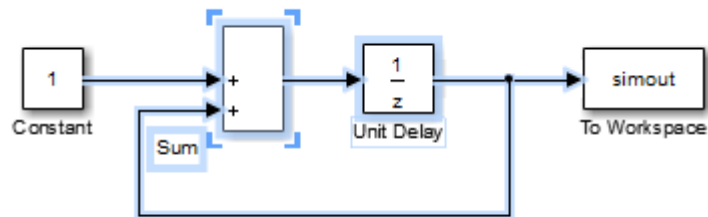
Group Existing Blocks into a Subsystem

If your model already contains the blocks that you want to convert to a subsystem, create the subsystem by grouping those blocks.

- 1 Select individual blocks that you want to include in a subsystem. To select multiple blocks that are in one area of the model, click the left mouse button and drag the cursor to create a bounding box that encloses the

blocks and the connecting lines that you want to include in the subsystem. For more information, see “Select Multiple Objects Using a Bounding Box” on page 4-7.

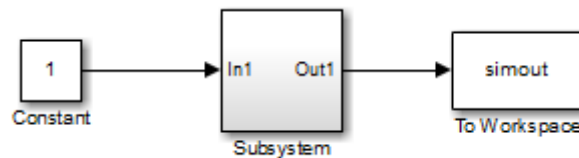
For example, this figure shows a model that represents a counter. The bounding box selects the Sum and Unit Delay blocks.



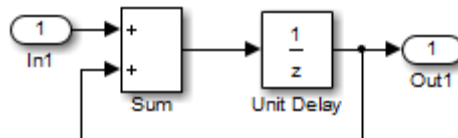
When you release the mouse button, the two blocks and all the connecting lines are selected.

2 Select **Diagram > Subsystems & Model Reference > Create Subsystem from Selection**. A Subsystem block replaces the selected blocks.

3 Resize the Subsystem block so the port labels are readable.



If you open the Subsystem block, the underlying system opens. For example:



Notice that Simulink adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem.

As with all blocks, you can change the name of the Subsystem block. To customize the appearance and dialog box of the block, you can mask the subsystem (see “Masking”).

Undoing Subsystem Creation

Right after you create subsystem by grouping blocks (see “Group Existing Blocks into a Subsystem” on page 4-38), you can undo that operation. To undo subsystem creation, select **Edit > Undo Subsystem Creation**.

Subsystem Execution

You can configure a subsystem to execute either conditionally or unconditionally.

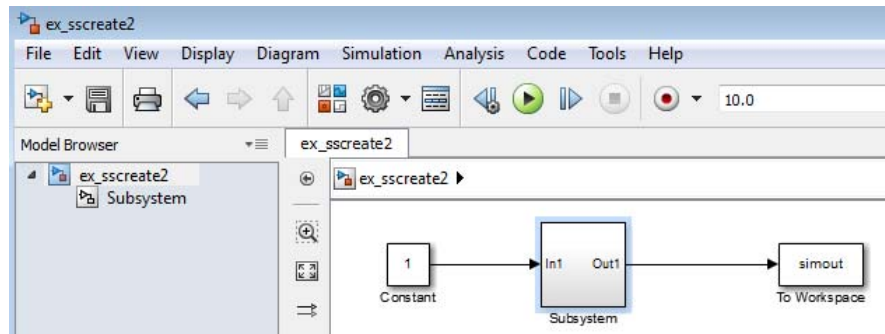
- An unconditionally executed subsystem always executes.
- A conditionally executed subsystem may or may not execute, depending on the value of an input signal. For details, see “Conditional Subsystems”.

Navigate Model Hierarchy

Subsystems allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy using the “Model Browser” on page 9-80 or with Simulink Editor model navigation commands.

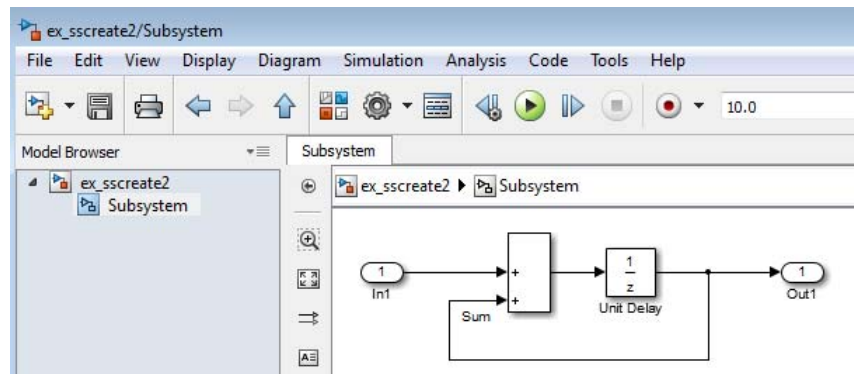
To open a subsystem using the Simulink Editor context menu for the Subsystem block:

- 1 In the Simulink Editor, right-click the Subsystem block.



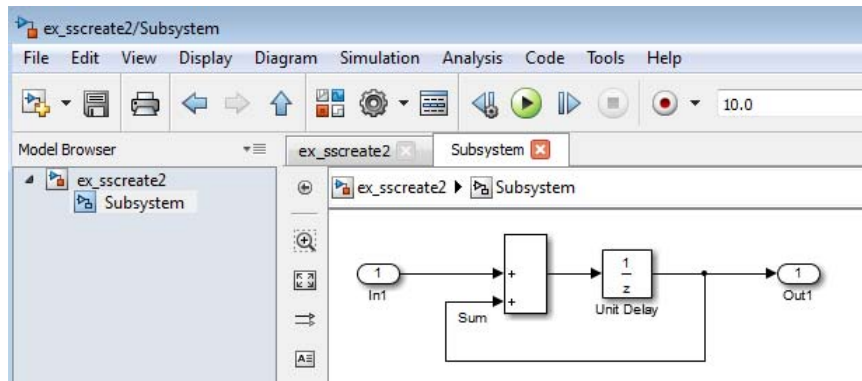
2 From the context menu, select one of these options:

- **Open** — Open the subsystem, in the same window and tab as used for the top model.

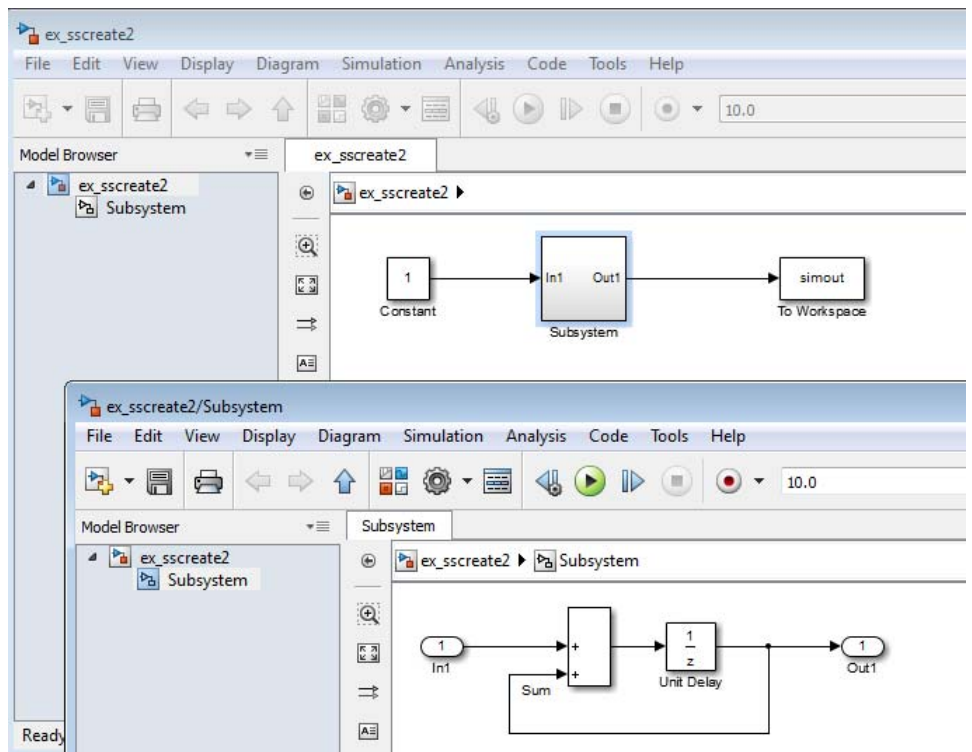


- **Open In New Tab** — Open the subsystem, creating an additional tab for the subsystem.

4 Creating a Model



- **Open In New Window**— Open the subsystem, opening a new Simulink Editor window.



For any operation to open a subsystem, you can use a keyboard shortcut to have the subsystem open in a new tab or window:

Where to Open the Subsystem	Keyboard Shortcut
In a new tab	Hold the CTRL key while opening the subsystem.
In a new window	Hold the SHIFT key while opening the subsystem.

To display the parent of a subsystem:

- 1 Open the tab that contains the subsystem.
- 2 Select **View > Navigate > Up to Parent**.

Preview Content of Subsystems

You can use content preview to display a representation of the contents of a subsystem, without opening the subsystem. Content preview helps you to understand at a glance the kind of processing performed by the subsystem. For details, see “Preview Content of Hierarchical Items” on page 1-28.

Label Subsystem Ports

By default, Simulink labels ports on a Subsystem block. The labels are the names of the Inport and Outport blocks that connect the subsystem to blocks outside of the subsystem.

You can specify how Simulink labels the ports of a subsystem.

- 1 Select the Subsystem block.
- 2 Select one of the labeling options from **Diagram > Format > Port Labels** menu (for example, From Port Block Name).

Control Access to Subsystems

You can control user access to subsystems. For example, you can prevent a user from viewing or modifying the contents of a library subsystem while still allowing the user to employ the subsystem in a model.

Note This method does not necessarily prevent a user from changing the access restrictions. To hide proprietary information that is in a subsystem, consider using protected model referencing models (see “Protected Model” on page 6-69).

To restrict access to a library subsystem, open the subsystem parameter dialog box and set **Read/Write permissions** to one of these values:

- **ReadOnly**: A user can view the contents of the library subsystem but cannot modify the reference subsystem without disabling its library link or changing its **Read/Write permissions** to **ReadWrite**.
- **NoReadOrWrite**: A user cannot view the contents of the library subsystem, modify the reference subsystem, or change reference subsystem permissions.

Both options allow a user to employ the library subsystem in models by creating links (see “Libraries”). For more information about subsystem access options, see the Subsystem block documentation.

Note You do not receive a response if you attempt to view the contents of a subsystem whose **Read/Write permissions** parameter is set to **NoReadOrWrite**. For example, when double-clicking such a subsystem, Simulink does not open the subsystem and does not display any messages.

Control Subsystem Behavior with Callbacks

You can 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
- Closing a subsystem

For details, see “Block Callbacks” on page 23-19.

Interconvert Subsystems and Block Diagrams

To interconvert subsystems and block diagrams, use one of these functions:

`Simulink.SubSystem.copyContentsToBlockDiagram`

Copies the contents of a subsystem to an empty block diagram.

`Simulink.BlockDiagram.copyContentsToSubSystem`

Copies the contents of a block diagram to an empty subsystem.

Empty Subsystems and Block Diagrams

To empty subsystems and block diagrams, use one of these functions:

`Simulink.SubSystem.deleteContents`

Deletes the contents of a subsystem.

`Simulink.BlockDiagram.deleteContents`

Deletes the contents of a block diagram.

Use Control Flow Logic

In this section...

“Equivalent C Language Statements” on page 4-46

“Conditional Control Flow Logic” on page 4-46

“While and For Loops” on page 4-49

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

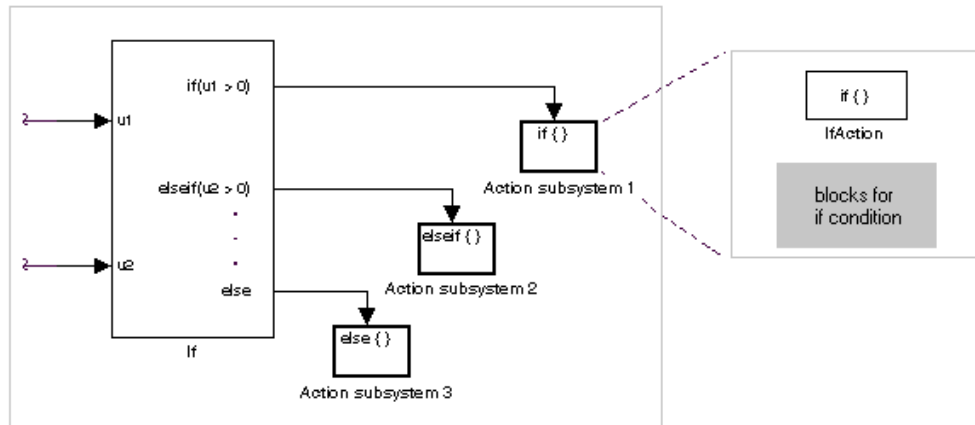
Conditional Control Flow Logic

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 u_1 , u_2 , ..., u_n 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 u_1 , u_2 , ..., u_n 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.

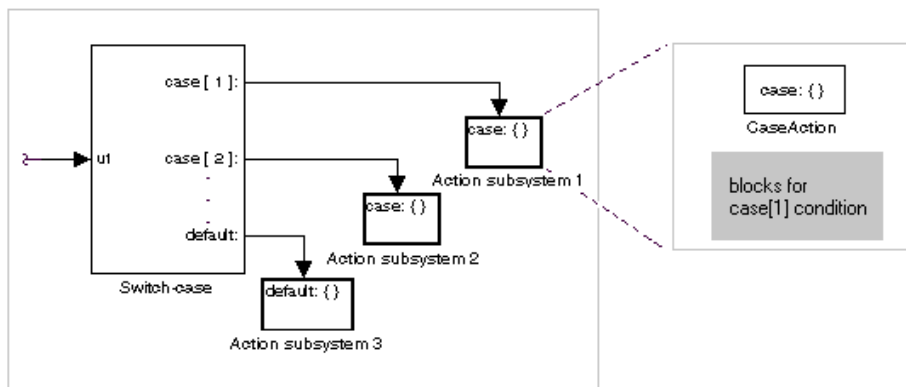
Once connected, the subsystem takes on the identity of the condition it is connected to and behaves like an enabled subsystem.

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.

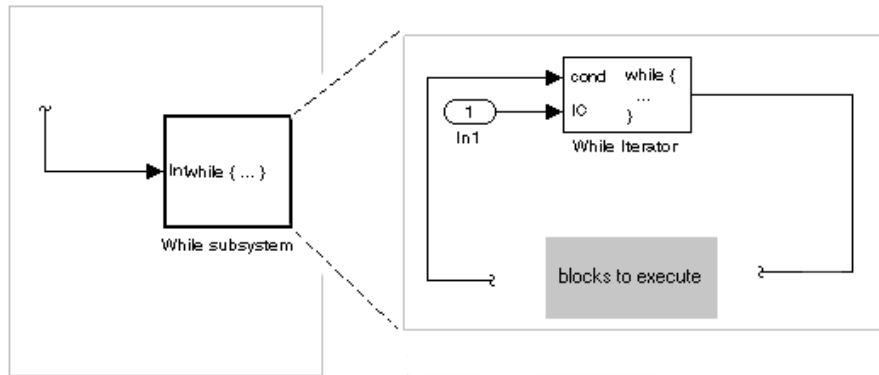
While and For Loops

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.

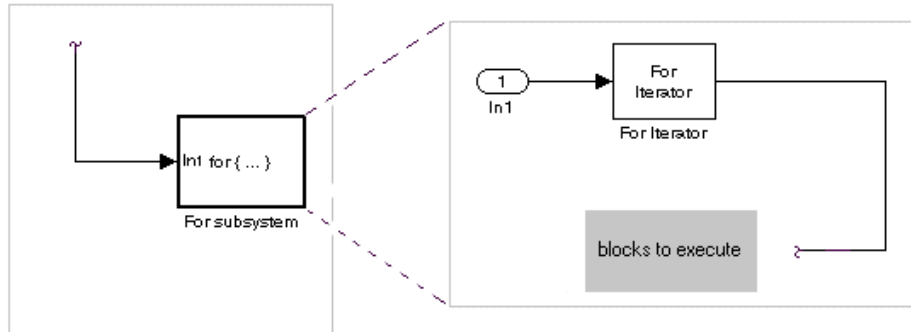
- 6 Create a block diagram in the subsystem that defines the subsystem's outputs.

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.

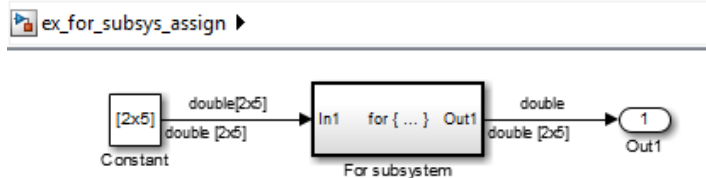
- 3 (Optional) Set the For Iterator block to output its iterator value for use in the block programming of the For Iterator Subsystem.

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.

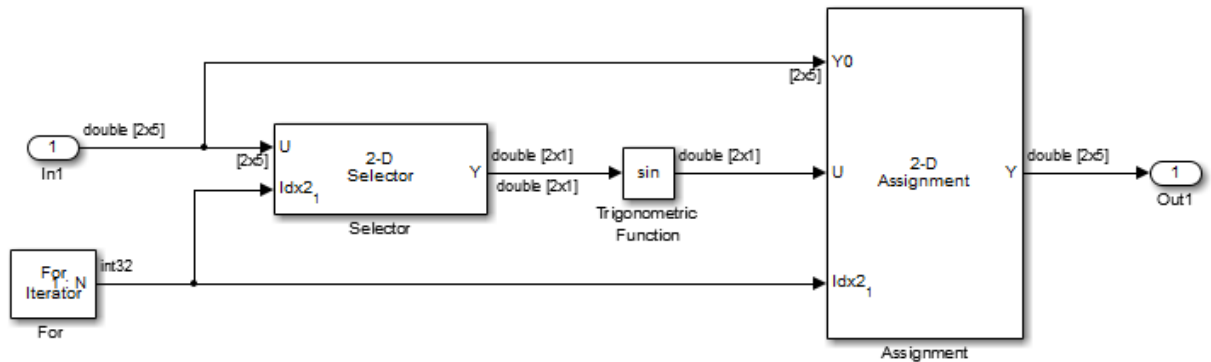
Note The diagram must not contain blocks with continuous states (for example, blocks from the Continuous block library). The sample times of all the blocks must be either inherited (-1) or constant (`inf`).

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.



4 Creating a Model

ex_for_subsys_assign ▶ For subsystem



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.

Callbacks for Customized Model Behavior

In this section...

“Model, Block, and Port Callbacks” on page 4-56

“What You Can Do with Callbacks” on page 4-56

“Avoid run Commands in Callback Code” on page 4-57

“See Also” on page 4-57

Model, Block, and Port Callbacks

Callbacks are a series of user-defined commands 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 on 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 Loading a Model” on page 1-8
- “Modify Behavior for Opening a Block” on page 23-20
- “Execute MATLAB Code Before Starting Simulation” on page 15-10

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

For information about how to define specific kinds of callbacks, see:

- “Model Callbacks” on page 4-58
- “Block Callbacks” on page 4-64
- “Port Callbacks” on page 4-72

For information about viewing the order in which callbacks in a model execute, see “Callback Tracing” on page 4-73.

Model Callbacks

In this section...
“Create Model Callbacks” on page 4-58
“View Model Callbacks” on page 4-59
“Model Callback Parameters” on page 4-60

Create Model Callbacks

You can create model callbacks interactively or programmatically.

To create a model callback interactively:

- 1** In the Simulink Editor, select **File > Model Properties > Model Properties**.
- 2** In the Model Properties dialog box, select the **Callbacks** tab.
- 3** In the left pane, select the callback parameter.
- 4** In the right pane, enter the code to be invoked for the selected callback parameter.

To create a model callback programmatically, use the `set_param` command to assign MATLAB code to a callback parameter. For example, the following command evaluates the variable `testvar` when you double-click the `Test` block in `mymodel`:

```
set_param('mymodel/Test', 'OpenFcn', 'testvar')
```

The execution of callbacks for model referencing reflects the order in which the top model and the model it references execute their callbacks. For example, suppose:

- Model A references model B.
- Model A has a `OpenFcn` callback that creates variables in the MATLAB workspace.

- Model B has a `CloseFcn` callback that clears the MATLAB workspace.

Simulating model A triggers rebuilding the referenced model B.

When Simulink rebuilds model B, it opens and closes model B and 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 could use a `StopFcn` callback in model A to clear from the MATLAB workspace the variables used by the model.

View Model Callbacks

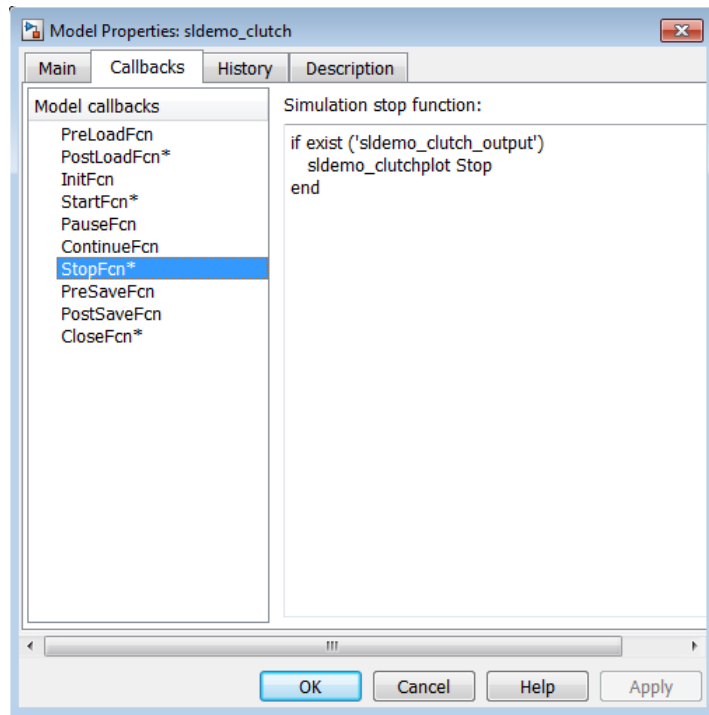
1 In the Simulink Editor, select **File > Model Properties > Model Properties**.

2 In the Model Properties dialog box, select the **Callbacks** tab.

A list of callbacks that the model defines appear in the left pane of the dialog box, highlighted with an asterisk.

3 To view the code for a callback that has an asterisk, click the callback.

The MATLAB code appears in the edit box on the right pane of the dialog box. In this example, `StopFcn*` is one of the defined callbacks for this model. Clicking `StopFcn*` displays the user-defined code for that callback.



Model Callback Parameters

Model Loading and Closing Callback Parameters

Model Callback Parameter	When Executed
PreLoadFcn	<p>Before the model is loaded.</p> <p>Defining a callback code for this parameter is useful for loading variables that the model uses.</p> <p>If you want to call your model from a MATLAB file without opening your model, use the <code>load_system</code> function so that the <code>PreLoadFcn</code> executes.</p> <p>For examples, see:</p>

Model Callback Parameter	When Executed
	<ul style="list-style-type: none"> • “Load Variables When Loading a Model” on page 1-8 • 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. • “Model Reference Simulation Targets” on page 6-37 <p>Limitations include:</p> <ul style="list-style-type: none"> • For the <code>PreLoadFcn</code> callback, the <code>get_param</code> command does not return the model parameter values because the model is not yet loaded. Instead, <code>get_param</code> returns: <ul style="list-style-type: none"> ▪ The default value for a standard model parameter such as <code>solver</code> ▪ An error message for a model parameter added with <code>add_param</code> • Programmatic access to Scopes is not supported.
PostLoadFcn	<p>After the model is loaded.</p> <p>Defining callback code for this parameter may be useful for generating an interface requiring a loaded model.</p> <p>Limitations include:</p> <ul style="list-style-type: none"> • If you make structural changes with <code>PostLoadFcn</code>, the function does not set the model <code>Dirty</code> flag to indicate unsaved changes. When you close the model, Simulink does not prompt you to save.

Model Callback Parameter	When Executed
	<ul style="list-style-type: none"> • Programmatic access to Scopes is not supported.
CloseFcn	<p>Before the block diagram is closed.</p> <p>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.</p>

Model Saving Callback Parameters

Model Callback Parameter	When Executed
PreSaveFcn	Before the model is saved.
PostSaveFcn	<p>After the model is saved.</p> <hr/> <p>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.</p> <hr/>

Model Simulation Callback Parameters

Model Callback Parameter	When Executed
StartFcn	<p>Before the simulation starts.</p> <p>For an example, see “Execute MATLAB Code Before Starting Simulation” on page 15-10.</p>
InitFcn	<p>At start of model simulation.</p> <p>For examples, see:</p>

Model Callback Parameter	When Executed
	<ul style="list-style-type: none"> • “Creating Programmatic Hyperlinks” on page 14-44 • “Track Object Using MATLAB Code” on page 29-178 <hr/> <p>Note During model compilation, Simulink evaluates variant objects before calling the model <code>InitFcn</code> callback. Do not modify the condition of the variant object in the <code>InitFcn</code> callback. For more information, see “Workflow for Implementing Variants” on page 8-3.</p> <hr/>
PauseFcn	After the simulation pauses.
ContinueFcn	Before the simulation continues.
StopFcn	<p>After the simulation stops.</p> <p>Output is written to workspace variables and files before the <code>StopFcn</code> is executed.</p>

Block Callbacks

In this section...
“Create Block Callbacks” on page 4-64
“Block Callback Parameters” on page 4-64

Create Block Callbacks

You can create block callbacks interactively or programmatically.

To create block callbacks interactively:

- 1** Right-click a block, and in the context menu, select **Properties**.
- 2** In the Block Properties dialog box, select the **Callback** tab.
- 3** Create the callback code.

To create a block callback programmatically, use the `set_param` command to assign MATLAB code to the block callback parameter.

For more information, see “Block Callback Parameters” on page 4-64.

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	<p>When the block is opened.</p> <p>Generally, use this parameter with Subsystem blocks.</p> <p>The callback executes when you double-click the block or when you call an <code>open_system</code> command</p>

Block Callback Parameter	When Executed
	<p>with the block as an argument. The <code>OpenFcn</code> parameter overrides the normal behavior associated with opening a block, which is to display the block's dialog box or to open the subsystem. Examples of tasks that you can use <code>OpenFcn</code> 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.</p> <p>For examples of using <code>OpenFcn</code> with model referencing, see:</p> <ul style="list-style-type: none"> • 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. • “Model Reference Simulation Targets” on page 6-37
LoadFcn	<p>After the block diagram is loaded.</p> <p>For Subsystem blocks, the <code>LoadFcn</code> callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a <code>LoadFcn</code> callback defined.</p>

Block Editing Callback Parameters

Block Callback Parameter	When Executed
MoveFcn	When the block is moved or resized.
NameChangeFcn	<p>After a block name or path changes.</p> <p>When a Subsystem block path changes, after calling its own <code>NameChangeFcn</code> callback, the Subsystem block calls this callback for all blocks that it contains.</p>

Block Callback Parameter	When Executed
PreCopyFcn	<p>Before a block is copied. The PreCopyFcn is also executed if an add_block command is used to copy the block.</p> <p>If you copy a Subsystem block that contains a block for which the PreCopyFcn callback is defined, that callback executes also.</p> <p>The block CopyFcn callback is called after all PreCopyFcn callbacks are executed, unless PreCopyFcn invokes the error command, either explicitly or via a command used in any PreCopyFcn.</p>
CopyFcn	<p>After a block is copied. The callback is also executed if an add_block command is used to copy the block.</p> <p>If you copy a Subsystem block that contains a block for which the CopyFcn parameter is defined, the callback is also executed.</p>
ClipboardFcn	<p>When the block is copied or cut to the system clipboard.</p>
PreDeleteFcn	<p>Before a block is graphically deleted (for example, when you graphically delete the block or invoke delete_block on the block).</p> <p>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.</p>

Block Callback Parameter	When Executed
DeleteFcn	<p>After a block is graphically deleted (for example, when you graphically delete the block, invoke <code>delete_block</code> on the block, or close the model containing the block).</p> <p>When the <code>DeleteFcn</code> is called, the block handle is still valid and can be accessed using <code>get_param</code>. If the block is graphically deleted by invoking <code>delete_block</code> or by closing the model, after deletion the block is destroyed from memory and the block's <code>DestroyFcn</code> is called.</p> <p>For Subsystem blocks, the <code>DeleteFcn</code> callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a <code>DeleteFcn</code> callback defined.</p>
DestroyFcn	<p>When the block has been destroyed from memory (for example, when you invoke <code>delete_block</code> on either the block or a subsystem containing the block or close the model containing the block).</p> <p>If the block was not previously graphically deleted, the <code>blockDeleteFcn</code> callback is called prior to the <code>DestroyFcn</code>. When the <code>DestroyFcn</code> is called, the block handle is no longer valid.</p>
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.
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.

Block Callback Parameter	When Executed
CloseFcn	<p>When the block is closed using the <code>close_system</code> command.</p> <p>The <code>CloseFcn</code> 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 <code>close_system</code>.</p> <p>For example, to close all open MATLAB windows, use a command such as:</p> <pre>set_param('my_model', 'CloseFcn', 'close all')</pre>
ModelCloseFcn	<p>Before the block diagram is closed.</p> <p>When the model is closed, the block's <code>ModelCloseFcn</code> is called prior to its <code>DeleteFcn</code>.</p> <p>For Subsystem blocks, the <code>ModelCloseFcn</code> callback is performed for any blocks in the subsystem (including other Subsystem blocks) that have a <code>ModelCloseFcn</code> callback defined.</p>

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 “Masking”). 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	<p>After a block or line is deleted in a subsystem.</p> <p>If the block has a DeleteFcn or DestroyFcn callback, those callbacks execute prior to the DeleteChildFcn callback.</p>
ErrorFcn	<p>When an error has occurred in a subsystem.</p> <p>Use the following form for the callback code for the ErrorFcn parameter:</p> <pre>newException = errorHandler(subsys, ... errorType, originalException)</pre> <p>where</p> <ul style="list-style-type: none"> • errorHandler is the name of the function. • subsys is a handle to the subsystem in which the error occurred. • errorType is a Simulink string indicating the type of error that occurred. • originalException is an MSLEException (see “Error Handling in Simulink Using MSLEException” on page 15-26). • newException is an MSLEException specifying the error message to be displayed to the user. <p>If you provide the original exception, then you do not need to specify the subsystem and the error type.</p> <p>The following command sets the ErrorFcn of the subsystem subsys to call the errorHandler callback:</p> <pre>set_param(subsys, 'ErrorFcn', 'errorHandler')</pre>

Block Callback Parameter	When Executed
	In such calls to <code>set_param</code> , do not include the input arguments of the callback code. Simulink displays the error message returned by the callback.
ParentCloseFcn	<p>Before closing a subsystem containing the block or when the block is made part of a new subsystem using either:</p> <ul style="list-style-type: none">• The <code>new_system</code> command• In the Simulink Editor, the Diagram > Subsystem & Model Reference > Create Subsystem from Selection option <p>When you close the model, Simulink does not call the ParentCloseFcn callbacks of blocks at the root model level.</p>

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(gcf, 'PortHandles');  
set_param(phs.Inport, 'ConnectionCallback', 'foo');
```

where, `foo` is defined as:

```
function foo(portHandle)
```

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.

Model Workspaces

In this section...
“Model Workspace Differences from MATLAB Workspace” on page 4-74
“Troubleshooting Memory Issues” on page 4-75
“Simulink.ModelWorkspace Data Object Class” on page 4-75
“Change Model Workspace Data” on page 4-76
“Specify Data Sources” on page 4-79

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 4-79.

- You can interactively reload and save MAT-file, MATLAB file, and MATLAB code data sources.
- Only `Simulink.Parameter` and `Simulink.Signal` objects for which the storage class is set to `Auto` can reside in a model workspace. You must create all other Simulink data objects in the base MATLAB workspace to ensure the objects are unique within the global Simulink context and accessible to all models.

Note Subclasses of `Simulink.Parameter` and `Simulink.Signal` classes, including `mpt.Parameter` and `mpt.Signal` objects (Embedded Coder® license required), can reside in a model workspace only if their storage class is set to `Auto`.

- In general, parameter variables in a model workspace are not tunable. However, you can tune model workspace variables declared as model arguments for referenced models. For more information, see “Using Model Arguments” on page 6-55.

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.

Simulink.ModelWorkspace Data Object Class

An instance 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

Change Model Workspace Data

The procedure for modifying a workspace depends on the data source of the model workspace.

Change Workspace Data Whose Source Is the Model File

If the data sources of a model workspace is data stored in the model, you can use Model Explorer or MATLAB commands to change the model's workspace (see "Use MATLAB Commands to Change Workspace Data" on page 4-78).

For example, to create a variable in a model workspace:

- 1** Open the Model Explorer by selecting **View > Model Explorer**.
- 2** In the Model Explorer **Model Hierarchy** pane, 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 by selecting **View > 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 by selecting **View > 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 by selecting **View > 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 “Exporting Workspace Variables” on page 9-60 and “Importing Workspace Variables” on page 9-62.

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

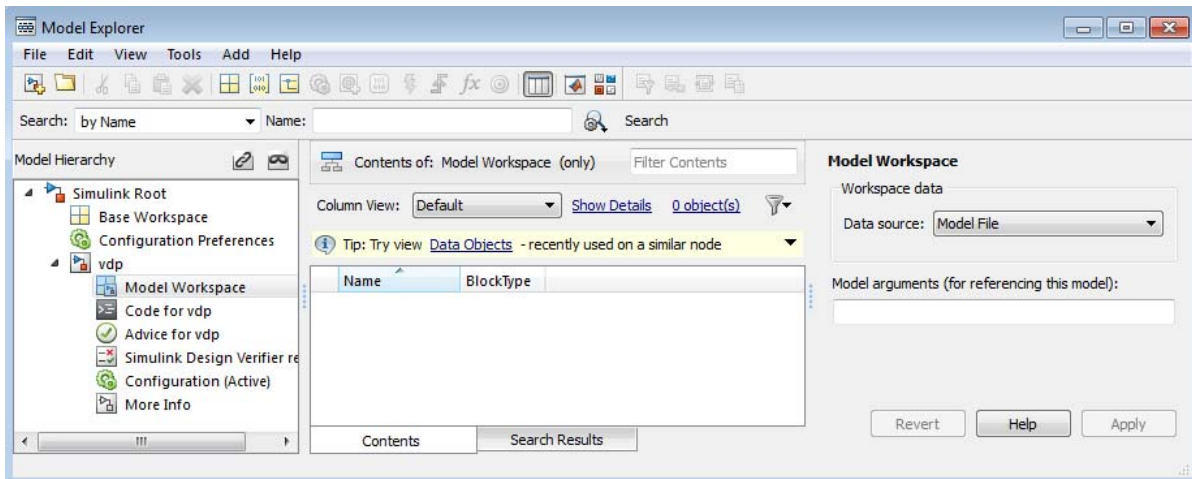
For example, the following MATLAB code creates variables specifying model parameters in the model workspace, saves the parameters, modifies one of them, and then reloads the workspace to restore it to its previous state.

```
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;
```


Specify Data Sources

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 by selecting **View > Model Explorer**.
- 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 4-78.

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 4-80).

- **MATLAB File**

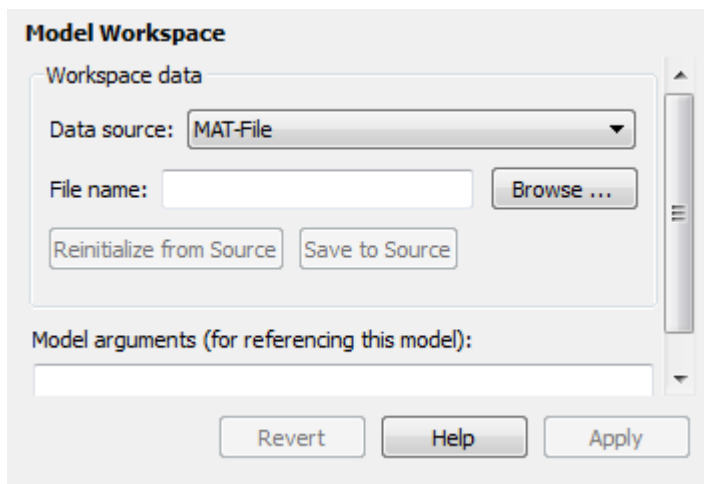
Specifies that the data source is a MATLAB file. Selecting this option causes additional controls to appear (see “MAT-File and MATLAB File Source Controls” on page 4-80).

- **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 4-81).

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.

The screenshot shows the 'Model Workspace' dialog box. It has a title bar 'Model Workspace'. Inside, there is a section titled 'Workspace data' which contains a 'Data source:' dropdown menu currently set to 'MATLAB Code'. Below this is a 'MATLAB Code:' label followed by a large empty text area. At the bottom of this section is a 'Reinitialize from Source' button. Below the 'Workspace data' section is a label 'Model arguments (for referencing this model):' followed by an empty text input field. At the very bottom of the dialog are three buttons: 'Revert', 'Help', and 'Apply'.

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.

Model Arguments (For Referencing This Model)

Specifies arguments that can be passed to instances of a model that another model references. For more information, see “Using Model Arguments” on page 6-55.

Symbol Resolution

In this section...

“Symbols” on page 4-83

“Symbol Resolution Process” on page 4-83

“Numeric Values with Symbols” on page 4-85

“Other Values with Symbols” on page 4-85

“Limit Signal Resolution” on page 4-86

“Explicit and Implicit Symbol Resolution” on page 4-87

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 “How Mask Parameters Work” on page 26-5)
- 2** The model workspace of the model that contains the block (see “Model Workspaces” on page 4-74)
- 3** The MATLAB base workspace (see “What Is the MATLAB 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 *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 Reference”.

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 4-87 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 “Diagnostics Pane: Data Validity” > “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.

Consult the Model Advisor

In this section...

- “About the Model Advisor” on page 4-88
- “Start the Model Advisor” on page 4-89
- “Overview of the Model Advisor Window” on page 4-92
- “Overview of the Model Advisor Dashboard” on page 4-95
- “Run Model Advisor Checks” on page 4-96
- “Run Checks Using Model Advisor Dashboard” on page 4-99
- “Highlight Model Advisor Analysis Results” on page 4-101
- “Fix a Warning or Failure” on page 4-103
- “Revert Changes Using Restore Points” on page 4-107
- “View and Save Model Advisor Reports” on page 4-109
- “Run the Model Advisor Programmatically” on page 4-112
- “Check Support for Libraries” on page 4-112
- “Checks Triggering an Update Diagram” on page 4-113
- “Model Advisor Limitations” on page 4-113
- “Consult the Upgrade Advisor” on page 4-113

About the Model Advisor

The Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation of the system that the model represents. If you have a Simulink Coder or Simulink Verification and Validation™ license, the Model Advisor can also check for model settings that result in generation of inefficient code or code unsuitable for safety-critical applications. (For more information about using the Model Advisor in code generation applications, see “Advice About Optimizing Models for Code Generation” in the Simulink Coder documentation.)

The Model Advisor produces a report that lists the suboptimal conditions or settings that it finds, suggesting better model configuration settings where appropriate. In some cases, the Model Advisor provides mechanisms for automatically fixing warnings and failures or fixing them in batches. For more information on individual checks, see “Simulink Checks”.

Software is inherently complex and may not be completely free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at <http://www.mathworks.com/support/bugreports/>. The bug reports are an integral part of the documentation for each release. Examine periodically all 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 will increase 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 any unintended functionality.

Start the Model Advisor

There are two Model Advisor GUIs to run checks that verify the syntax of your model: the Model Advisor window and the Model Advisor dashboard.

If you want to ...	Use the ...	Set Preferences ...
Consistently run the same set of checks on your model.	Model Advisor dashboard. The Model Advisor dashboard does not reload checks for an	<ol style="list-style-type: none"> 1 From the Model Editor, select Analysis > Model Advisor > Preferences. 2 In the Model Advisor Preferences



If you want to ...	Use the ...	Set Preferences ...
	analysis, saving analysis time.	window, for the Default Mode , select Model Advisor Dashboard.
Select checks to run on your model.	Model Advisor window.	<ol style="list-style-type: none"> 1 From the Model Editor, select Analysis > Model Advisor > Preferences. 2 In the Model Advisor Preferences window, for the Default Mode, select Model Advisor.

Before starting the Model Advisor, make sure that the current folder is writable. If the folder is not writable, when you start the Model Advisor, you see an error message.

The Model Advisor uses the Simulink project (s1prj) folder to store reports and other information. If this folder does not exist in the current folder, the Model Advisor creates it.

Note If you “Comment Blocks” on page 23-14, they are excluded from both simulation and Model Advisor analysis.

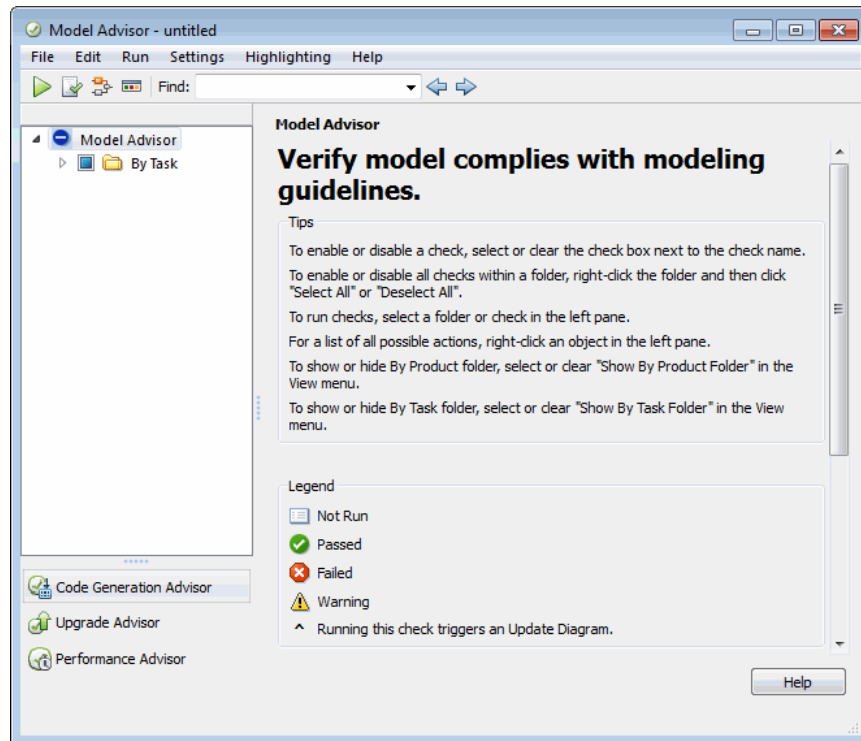
To start the Model Advisor, use one of the following methods:

To open the Model Advisor window or Model Advisor dashboard ...	For a...	Do this action:
From the Model Editor	Model or subsystem	<p>1 Select Analysis > Model Advisor > Model Advisor or Model Advisor Dashboard.</p> <p>Alternately, on the Model Editor toolbar  drop-down list, select Model Advisor or Model Advisor Dashboard.</p> <hr/> <p>Note Clicking  opens the Model Advisor GUI set in the Analysis > Model Advisor > Preferences interface.</p> <hr/> <p>2 In the System Selector window, select the model or subsystem that you want.</p> <p>3 Click OK.</p>
From the Model Explorer	Model	In the Contents pane, select Advice for model1 . <i>model1</i> is the name of the model that you want to check. (For more information, see “Model Explorer Overview” on page 9-2.)

To open the Model Advisor window or Model Advisor dashboard ...	For a...	Do this action:
From the context menu	Subsystem	Right-click the subsystem that you want to check and select Model Advisor .
Programmatically	Model or subsystem	At the MATLAB prompt, enter <code>modeladvisor(model)</code> . <i>model</i> is a handle or name of the model or subsystem that you want to check. (For more information, see the <code>modeladvisor</code> function reference page.)

Overview of the Model Advisor Window


When you start the Model Advisor, the Model Advisor window displays two panes. The left pane lists the folders in the Model Advisor. Expanding the folders displays the available checks. The right pane provides instructions on how to view, enable, and disable checks. It also provides a legend describing the displayed symbols.



You can:


- Select **By Task** to display checks related to specific tasks, such as updating the model to be compatible with the current Simulink version.
- Select some or all of the checks using the check boxes or context menus for the checks, and then run one or all selected checks.
- Reset the status of the checks to not run by right-clicking **Model Advisor** in the left pane and selecting **Reset** from the context menu.
- Specify input parameters for some checks to run.
- Select **Code Generation Advisor** to help configure your model to meet code generation objectives. See “Application Objectives”.
- Select **Upgrade Advisor** to help upgrade and improve models with the current release. See “Consult the Upgrade Advisor” on page 4-113.


- Select **Performance Advisor** to help improve the simulation performance of your model. See “How Performance Advisor Improves Simulation Performance” on page 20-2.

To find checks and folders, enter text in the **Find:** field and click the **Find Next** button (). The Model Advisor searches in check names, folder names, and analysis descriptions for the text.


To customize the Model Advisor, on the toolbar, select **Settings > Preferences** to open the Model Advisor Preferences dialog box.

- **Show By Product Folder** displays the checks available for each product.
- **Show By Task Folder** displays checks related to specific tasks.
- **Show Source Tab** displays the **Source** tab in the right pane of the Model Advisor window. When you click the **Source** tab, the Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.
- **Show Exclusion tab** displays the **Exclusions** tab in the right pane of the Model Advisor window. When you click the **Exclusions** tab, the Model Advisor window displays checks that are excluded from the Model Advisor analysis.


To run the selected checks, on the toolbar of the Model Advisor window, click **Run selected checks** (.

After running checks, the Model Advisor displays the results in the right pane. Additionally, the Model Advisor generates an HTML report of the check results. You can view this report in a separate browser window by clicking either **Open Report** () or the **Report** link at the folder level.

To view the analysis results for individual checks, you can specify that the Model Advisor use color highlighting on the model diagram. To enable Model Advisor highlighting, do one of the following in the Model Advisor window, on the toolbar:

- Select **Highlighting > Enable Highlighting**.
- Click the **Enable highlighting** toggle button (.

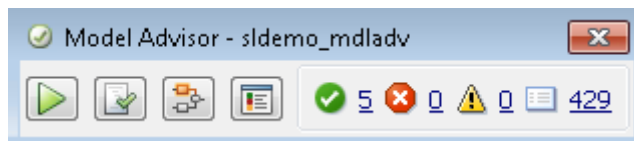
When you use highlighting on the model diagram, you can specify that the Model Advisor highlight model blocks that are excluded from individual Model Advisor checks. On the toolbar of the Model Advisor window, select **Highlighting > Highlight exclusions**. If you have a Simulink Verification and Validation license, you can create or modify exclusions to the Model Advisor checks.


To switch to the Model Advisor dashboard, click the **Switch to Model Advisor Dashboard** toggle button (.


Note When you open the Model Advisor for a model that you have previously checked, the Model Advisor initially displays the check results generated the last time that you checked the model. If you recheck the model, the new results replace the previous results in the Model Advisor window.

Overview of the Model Advisor Dashboard


Using the Model Advisor dashboard, you can efficiently check that your model complies with modeling guidelines. When you use the Model Advisor dashboard, the Model Advisor does not reload checks for an analysis, saving analysis time.




To run checks, on the Model Advisor dashboard toolbar, click **Run checks** (.

After running checks, the Model Advisor generates an HTML report of the check results. You can view this report in a separate browser window by clicking the **Open Report** (.

To view highlighted results, click the **Enable Highlighting** toggle button ()


To switch to the Model Advisor window, click the **Switch to standard view** toggle button ()

Tip Use the Model Advisor window to select and view checks. To open the Model Advisor window, click the **Switch to standard view** toggle button ()

Run Model Advisor Checks

To perform checks on your model and view the results:

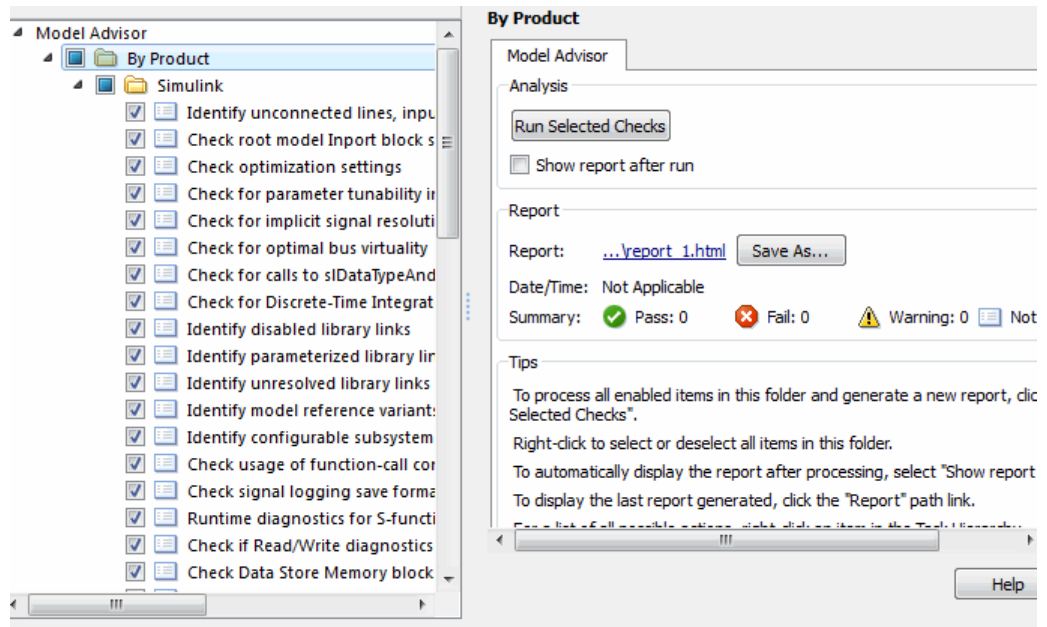
- 1 Open your model. For example, open the Model Advisor example model: `sldemo_md1adv`.
- 2 Start the Model Advisor.
 - a From the Simulink Editor, select **Analysis > Model Advisor > Model Advisor**.

Alternately, from the Simulink Editor toolbar  drop-down list, select **Model Advisor**.

- b In the System Selector window, select the model or system that you want to review. For example, `sldemo_md1adv`, and click **OK**.

The Model Advisor window opens and displays checks for the `sldemo_md1adv` model.

- 3 In the left pane, expand the **By Product** or **By Task** folder to display the subfolders. Then expand the **Simulink** folder to display the available checks. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 4 In the left pane, select the **By Product** folder. The right pane changes to a **By Product** view.



- 5 After you run the checks, select the **Show report after run** check box to see an HTML report of check results.

Tip Use the Model Advisor window for interactive fixing of warnings and failures. Model Advisor reports are best for viewing a summary of checks.

- 6 Run the selected checks by clicking the **Run Selected Checks** button. Alternately, select **Run selected checks** (▶). After the Model Advisor runs the checks, an HTML report displays the check results in a browser window.

Model Advisor Report - sldemo_mdladv.slx

Simulink version: 8.1

System: sldemo_mdladv

Treat as Referenced Model: off

Model version: 1.77

Current run: 26-Oct-2012 16:37:19

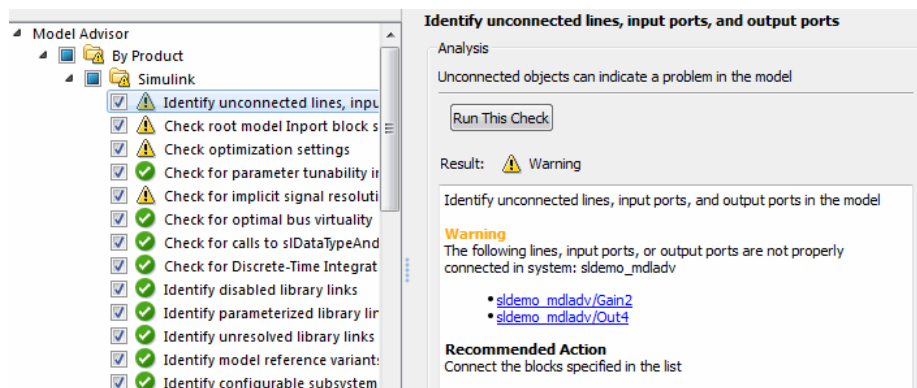
Run Summary

<input checked="" type="checkbox"/> Pass	<input checked="" type="checkbox"/> Fail	<input checked="" type="checkbox"/> Warning	<input checked="" type="checkbox"/> Not Run	Total
14	0	8	9	31

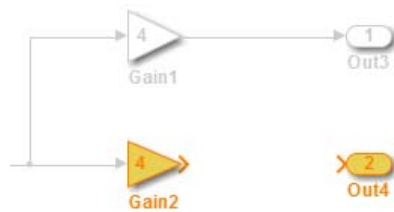
Simulink

Identify unconnected lines, input ports, and output ports

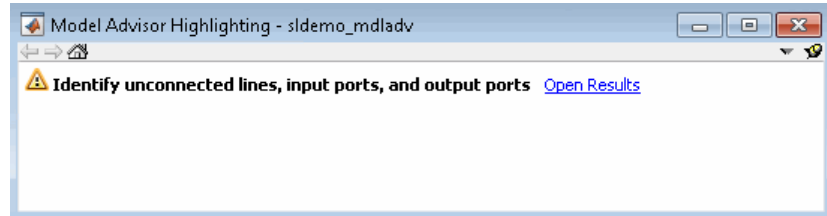
- Return to the Model Advisor window, which shows the check results.
- Select an individual check to open a detailed view of the results in the right pane. For example, selecting **Identify unconnected lines, input ports, and output ports** changes the right pane to the following view. Use this view to examine and exercise a check individually.



- In the Model Advisor window, click the **Enable highlighting** toggle button (). Checks with highlighted results have an icon.
 - The model window opens. The blocks causing the **Identify unconnected lines, input ports, and output ports** check warning are highlighted in yellow.



- 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.




- 10 After reviewing these check results in the Model Advisor window, you can choose to fix warnings or failures as described in “Fix a Warning or Failure” on page 4-103.

Run Checks Using Model Advisor Dashboard


To run checks on your model using the Model Advisor dashboard:

- 1 Open your model. For example, open the Model Advisor model, `sldemo_md1adv`.
- 2 Start the Model Advisor dashboard.
 - a From the Simulink Editor, select **Analysis > Model Advisor > Model Advisor Dashboard** .

Alternately, from the Simulink Editor toolbar  drop-down list, select Model Advisor Dashboard.

- b In the System Selector window, select the model or system that you want to review. For example, `sldemo_md1adv` and click **OK**.


The Model Advisor dashboard opens.

3 Optionally, select or view checks to run on your model by clicking the **Switch to standard view** toggle button (). The Model Advisor window opens.


- a** For example, in the Model Advisor window, open the **By Product** folder and select checks:

Identify unconnected lines, input ports, and output ports
Check root model Inport block specifications

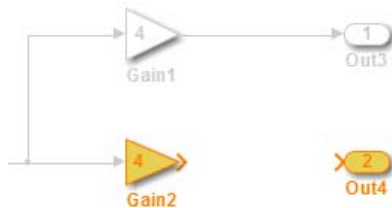
If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

- b** Switch back to the Model Advisor dashboard by clicking the **Switch to Model Advisor Dashboard** toggle button (.

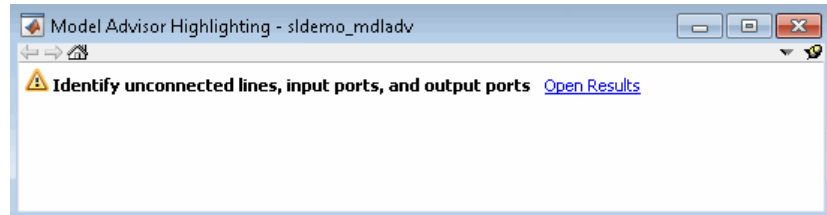
4 On the Model Advisor dashboard, click **Run checks** () to run the checks.

5 Click the **Enable Highlighting** toggle button () to view highlighted results.

- The model window opens. The blocks causing the **Identify unconnected lines, input ports, and output ports** check warning are highlighted in yellow.



- 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 After reviewing the check results, you can choose to fix warnings or failures as described in “Fix a Warning or Failure” on page 4-103. For example, fix the **Identify unconnected lines, input ports, and output ports** check warning:
 - a Connect model blocks Gain2 and Out4.
 - b On the Model Advisor dashboard, click **Run checks** (▶) to rerun the checks. The Model Advisor does not reload all the available checks, saving analysis time.
 The **Identify unconnected lines, input ports, and output ports** check passes.

Highlight Model Advisor Analysis Results




You can use color highlighting on the model diagram to indicate the analysis results for individual Model Advisor checks. Blocks that pass a check, fail a check, or cause a check warning are highlighted in color in the model window. Model Advisor highlighting is available for the following:

- Simulink blocks
- Stateflow charts




After you run a Model Advisor analysis, checks with highlighted results are indicated with an ⚠ icon in the Model Advisor window.



To use Model Advisor highlighting:


- 1 Run Model Advisor checks on your model.
- 2 Enable Model Advisor highlighting by doing one of the following:
 - On the toolbar of the Model Advisor window, select **Highlighting > Enable Highlighting**.

- On the toolbar of the Model Advisor window, click the **Enable highlighting** button (.
 - On the toolbar of the Model Advisor Dashboard, click the **Enable Highlighting** button (.
- 3 Optionally, to view model blocks that are excluded from the Model Advisor checks, select **Highlighting > Highlight Exclusions** on the Model Advisor window toolbar. If you have a Simulink Verification and Validation license, you can create or modify exclusions to the Model Advisor checks .
 - 4 In the left pane of the Model Advisor window, select a check with highlighted results. Checks with highlighted results are indicated with the  icon. Highlighting is not available for some checks.

Both 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.

Highlight Colors in the model window		
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.

Highlight Colors in the model window		
Gray with black border		Blocks that are excluded from the check.
White with black border		Subsystems that are excluded from the check.

- 5 After reviewing the check results in both the model window and the Model Advisor window, you can choose to fix warnings or failures as described in “Fix a Warning or Failure” on page 4-103.
- 6 To view highlighted results for another check, in the left pane of the Model Advisor window, select a check with an  icon.

Tip 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 might fail or warn due to your parameter or diagnostic settings.

Fix a Warning or Failure

Checks fail when a model or submodel has a suboptimal condition. A warning result is informational. You can choose to fix the reported issue, or move on to the next task. For more information on why a specific check does not pass, see the check documentation. You can use Model Advisor highlighting to identify model objects that cause a check warning or failure.

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 Model Advisor might report an invalid check result.

To fix warnings and failures:

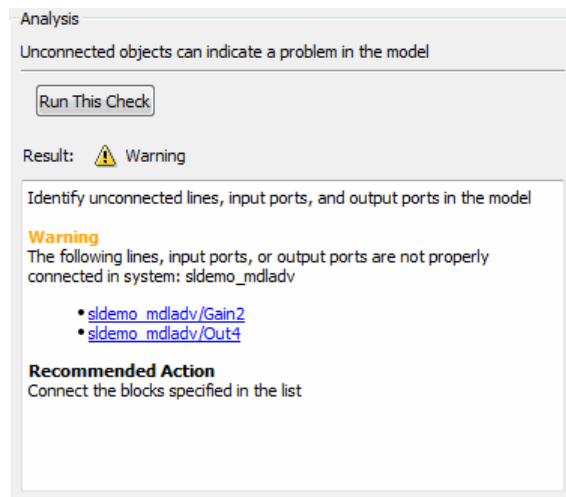
- Follow the instructions in the Analysis Result box to manually fix a warning or failure. See “Manually Fix Warnings or Failures” on page 4-104.
- Use the Action box, when available, to automatically fix failures. See “Automatically Fix Warnings or Failures” on page 4-105.
- Use the Model Advisor Results Explorer, when available, to batch-fix failures. See “Batch-Fix Warnings or Failures” on page 4-106.

Manually Fix Warnings or Failures

Checks have an Analysis Result box that describes the recommended actions to manually fix warnings or failures.

To manually fix warnings or failures within a task:

- 1 Optionally, save a model and data restore point so you can undo the changes that you make. For more information, see “Revert Changes Using Restore Points” on page 4-107.
- 2 In the Analysis Result box, review the recommended actions. Use the information to make changes to your model.



- 3 Rerun the check to verify that it passes.

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 Model Advisor might report an invalid check result.

Automatically Fix Warnings or Failures

Some checks have an Action box where you can automatically fix failures. The action box applies all of the recommended actions listed in the Analysis Result box.

Caution Before automatically fixing failures, review the Analysis Result box to verify that you want to apply all of the recommended actions. If you do not want to apply all of the recommended actions, do not use this method to fix warnings or failures.

To automatically fix all warnings or failures within a check:

- 1** Optionally, save a model and data restore point so you can undo the changes that you made by clicking the **Modify All** button. For more information, see “Revert Changes Using Restore Points” on page 4-107.
- 2** In the Action box, click **Modify All**.

The Action Result box displays a table of changes.

- 3** Rerun the check to verify that it passes.

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 Model Advisor might report an invalid check result.

Batch-Fix Warnings or Failures

Some checks in the Model Advisor have an **Explore Result** button that starts the Model Advisor Result Explorer. The Model Advisor Result Explorer allows you to quickly locate, view, and change elements of a model.

The Model Advisor Result Explorer helps you to modify only the items that the Model Advisor is checking.

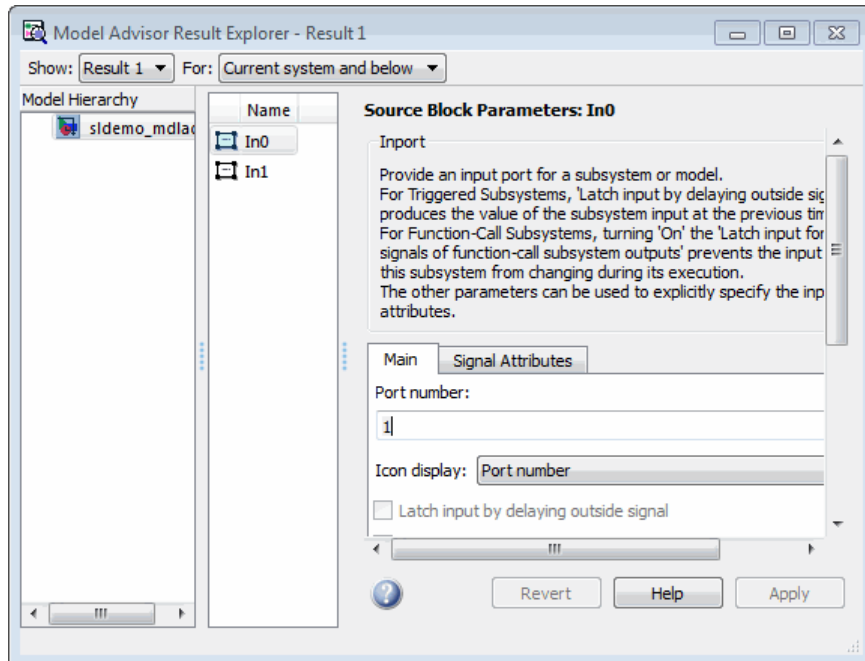
If a check does not pass and you want to explore the results and make batch changes:

- 1 Optionally, save a model and data restore point so you can undo changes that you make. For more information, see “Revert Changes Using Restore Points” on page 4-107.
- 2 In the Analysis box, click **Explore Result**.

The Model Advisor Result Explorer window opens.
- 3 Use the Model Advisor Result Explorer to modify block parameters.
- 4 In the Model Advisor window, rerun the check to verify that it passes.

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 Model Advisor might report an invalid check result.

In the following example, run **Check root model Inport block specifications** for the `sldemo_mdldadv` model. The result is a warning. Clicking the **Explore Result** button opens the Model Advisor Result Explorer window.



Revert Changes Using Restore Points

The Model Advisor provides a model and data restore point capability for reverting changes that you made in response to advice from the Model Advisor. A *restore point* is a snapshot in time of the model, base workspace, and Model Advisor. The Model Advisor maintains restore points for the model or subsystem of interest through multiple sessions of MATLAB.

Caution A restore point saves only the current working model, base workspace variables, and Model Advisor tree. It does not save other items, such as libraries and referenced submodels.

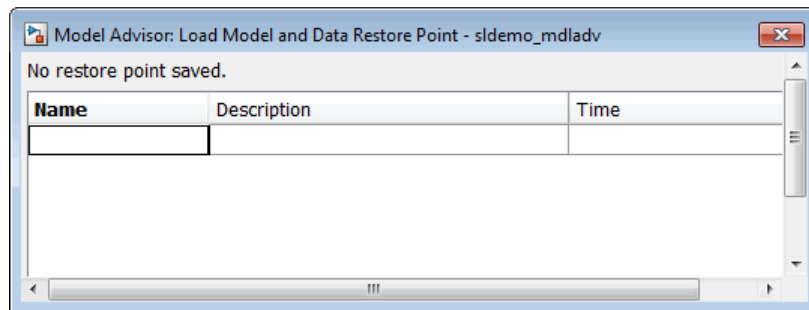
Save a Restore Point

You can save a restore point and give it a name and optional description, or allow the Model Advisor to automatically name the restore point for you.

To save a restore point:

1 Go to **File > Save Restore Point As**.

The Save Model and Data Restore Point dialog box opens.



2 In the **Name** field, enter a name for the restore point.

3 In the **Description** field, you can optionally add a description to help you identify the restore point.

4 Click **Save**.

The Model Advisor saves a restore point of the current model, base workspace, and Model Advisor status.

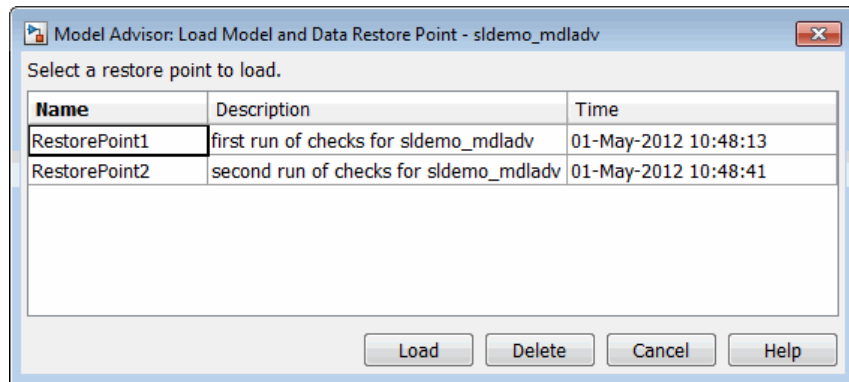
To quickly save a restore point, go to **File > Save Restore Point**. The Model Advisor saves a restore point with the name `autosaven`. n is the sequential number of the restore point. If you use this method, you cannot change the name of, or add a description to, the restore point.

Load a Restore Point

To load a restore point:

- 1 Optionally, save a model and data restore point so you can undo changes that you make.
- 2 Go to **File > Load Restore Point**.

The Load Model and Data Restore Point dialog box opens.



- 3 Select the restore point that you want.
- 4 Click **Load**.

The Model Advisor issues a warning that the restoration will remove changes that you made after saving the restore point.

- 5 Click **Load** to load the restore point that you selected.

The Model Advisor reverts the model, base workspace, and Model Advisor status.

View and Save Model Advisor Reports

When the Model Advisor runs checks, it generates an HTML report of check results. Each folder in the Model Advisor contains a report for the checks in that folder and the subfolders within that folder.

View Model Advisor Reports

You can access a report by selecting a folder and clicking the link in the **Report** box.

Tip While you can fix warnings and failures through Model Advisor reports, use the Model Advisor window for interactive fixing of 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, a message appears in the report. Time stamps 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 time stamp following the check name.

Model Advisor Report - sldemo_mdldv.slx

Simulink version: 8.1 Model version: 1.77
 System: sldemo_mdldv Current run: 26-Oct-2012 16:37:19
 Treat as Referenced Model: off

Run Summary

<input checked="" type="checkbox"/> Pass	<input checked="" type="checkbox"/> Fail	<input checked="" type="checkbox"/> Warning	<input checked="" type="checkbox"/> Not Run	Total
14	0	8	9	31

Simulink

Identify unconnected lines, input ports, and output ports

To show only what you are interested in viewing, you can manipulate the report.

- To view only the checks with the status that you are interested in, next to the Run Summary status, select the appropriate check boxes. For example, you can remove the checks that have not run by clearing the check box next to the Not Run status.
- Minimize folder results in the report by clicking the minus sign next to the folder name. When you minimize a folder, the report is updated to display a run summary for that folder:

Model Advisor Report - `sldemo_mdldv.slx`

Simulink version: 8.1

Model version: 1.77

System: `sldemo_mdldv`

Current run: 26-Oct-2012 16:50:17

Treat as Referenced Model: **off**

Run Summary

<input checked="" type="checkbox"/> Pass	<input checked="" type="checkbox"/> Fail	<input checked="" type="checkbox"/> Warning	<input checked="" type="checkbox"/> Not Run	Total
95	0	69	66	230

By Task

- Code Generation Efficiency 5 0 2 0
- Frequency Response Estimation 0 0 1 0
- Managing Data Store Memory Blocks 2 0 1 0

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.

Check for proper Merge block usage

Passed

Input Parameters Selection

Name	Value
Maximum analysis time (seconds)	inf

For more information, see the `EmitInputParametersToReport` property of the `Simulink.ModelAdvisor` class.

Save Model Advisor Reports

You can archive a Model Advisor report by saving it to a new location.

- 1 In the Model Advisor window, navigate to the folder that contains the report that you want to save.
- 2 Select the folder that you want. The right pane of the Model Advisor window displays information about that folder, including a **Report** box.
- 3 In the Report box, click **Save As**. A save as dialog box opens.
- 4 In the save as dialog box, navigate to the location where you want to save the report, and click **Save**. The Model Advisor saves the report to the new location.

Note If you rerun the Model Advisor, the report is updated in the working folder, not in the save location.

You can find the full path to the report in the title bar of the report window. Typically, the report is in the working folder: `s1prj/modeladvisor/model_name`.

Run the Model Advisor Programmatically

If you have a license for Simulink Verification and Validation, you can create MATLAB scripts and functions that run the Model Advisor programmatically. For example, you can create a function to check whether your model passes a specified set of the Model Advisor checks every time that you open the model, start a simulation, or, if you have Simulink Coder, generate code from the model. For more information, see the `ModelAdvisor.run` function in the Simulink Verification and Validation documentation.

Check Support for Libraries

There are Model Advisor checks available to verify the syntax of library models. When you use the Model Advisor to check a library model, the Model Advisor window indicates (~) checks that do not check libraries. To determine if you can run the check on library models, you can also refer to the check documentation, “Capabilities and Limitations”. You cannot use checks that require model compilation. If you have a license for Simulink Verification

and Validation, you can use an API to create custom checks which support library models.

Checks Triggering an Update Diagram

Model Advisor checks triggering an Update Diagram are marked with \wedge . An Update Diagram triggers a model compilation. Individual checks might require model compilation for either simulation or code generation.

Model Advisor Limitations

When you use the Model Advisor to check systems, the following limitations apply:

- 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 only the active subsystem.
- 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.

For limitations that apply to specific checks, see the Limitations section within the documentation of each check.

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:

- 1** From the Model Editor, select **Analysis > Model Advisor > Upgrade Advisor**
- 2** From the MATLAB command line, use the `upgradeadvisor` function:

```
upgradeadvisor modelName
```
- 3** Alternatively, you can open the Upgrade Advisor from the Model Advisor.
 - a** Under **By Task** checks, expand the folder **Upgrading to the Current Simulink Version** and select the check **Open the Upgrade Advisor**.
 - b** In the right pane under **Recommended Action**, click the link [Open the 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.

For more information on individual checks, see

- “Model Upgrades” for upgrade checks only
- “Simulink Checks” for all upgrade and advisor checks

Manage Model Versions

In this section...

“How Simulink Helps You Manage Model Versions” on page 4-115

“Model File Change Notification” on page 4-116

“Specify the Current User” on page 4-118

“Manage Model Properties” on page 4-118

“Log Comments History” on page 4-125

“Version Information Properties” on page 4-127

How Simulink Helps You Manage Model Versions

The Simulink software has these features to help you to manage multiple versions of a model:

- Use Simulink Projects to manage your project files, connect to source control, review modified files and compare revisions. See “Simulink Projects”.
- Model File Change Notification helps you manage work with source control operations and multiple users. See “Model File Change Notification” on page 4-116.
- As you edit a model, the Simulink software generates version information about the model, including a version number, who created and last updated the model, and an optional comments history log. The Simulink software automatically 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.
 - You can access Simulink version parameters from the MATLAB command line or a MATLAB script.
- 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.

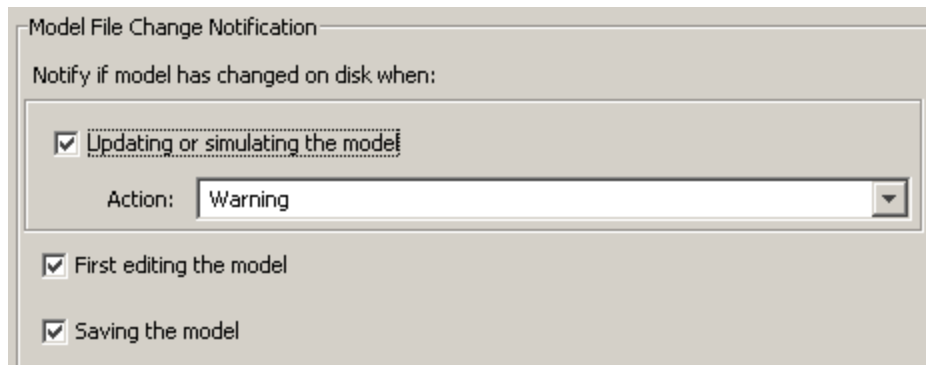
Model File Change Notification

You can use the Simulink Preferences window to specify whether to notify if the model has changed on disk when updating, simulating, editing, or saving the model. This can occur, for example, with source control operations and multiple users.

Note To programmatically check whether the model has changed on disk since it was loaded, use the function `slIsFileChangedOnDisk`.

To access the Simulink Preferences window, use one of these approaches:

- In the Simulink Editor, select **File > Simulink Preferences**.
- From the MATLAB Toolstrip, in the **Home** tab, in the **Environment** section, select **Preferences > Simulink**. Click the **Launch Simulink Preferences** button .



The Model File Change Notification options are in the right pane. You can use the three independent options as follows:

- If you select the **Updating or simulating the model** check box, you can choose what form of notification you want from the **Action** list:
 - Warning — in the MATLAB command window.
 - Error — in the MATLAB command window if simulating from the command line, or if simulating from a menu item, in the Simulation Diagnostics window.
 - Reload model (if unmodified) — if the model is modified, you see the prompt dialog. If unmodified, the model is reloaded.
 - Show prompt dialog — in the dialog, you can choose to close and reload, or ignore the changes.
- If you select the **First editing the model** check box, and the file has changed on disk, and the block diagram is unmodified in Simulink:
 - Any command-line operation that causes the block diagram to be modified (e.g., a call to `set_param`) will result in a warning:

```
Warning: Block diagram 'mymodel' is being edited but file has
changed on disk since it was loaded. You should close and
reload the block diagram.
```
 - Any graphical operation that modifies the block diagram (e.g., adding a block) causes a warning dialog to appear.
- If you select the **Saving the model** check box, and the file has changed on disk:
 - The `save_system` function displays an error, unless the `OverwriteIfChangedOnDisk` option is used.
 - Saving the model by using the menu (**File > Save**) or a keyboard shortcut causes a dialog to be shown. In the dialog, you can choose to overwrite, save with a new name, or cancel the operation.

For more options to help you work with source control and multiple users, see “Simulink Projects”.

Specify the Current User

When you create or update a model, your name is logged in the model for version control purposes. The Simulink software assumes that your name is specified by at least one of the following environment variables: `USER`, `USERNAME`, `LOGIN`, or `LOGNAME`. If your system does not define any of these variables, the Simulink software 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 on to your system. Thus, if you are using a UNIX system, you do not have to do anything to enable the Simulink software to identify you as the current user.

Windows systems, on the other hand, might define some or none of the “user name” environment variables that the Simulink software expects, depending on the version of Windows installed on your system and whether it is connected to a network. Use the MATLAB command `getenv` to determine which of the environment variables is defined. For example, enter

```
getenv('user')
```

at the MATLAB command line to determine whether the `USER` environment variable exists on your Windows system. If not, you must set it yourself.

On Windows, set the `USER` environment variable (if it is not already defined).

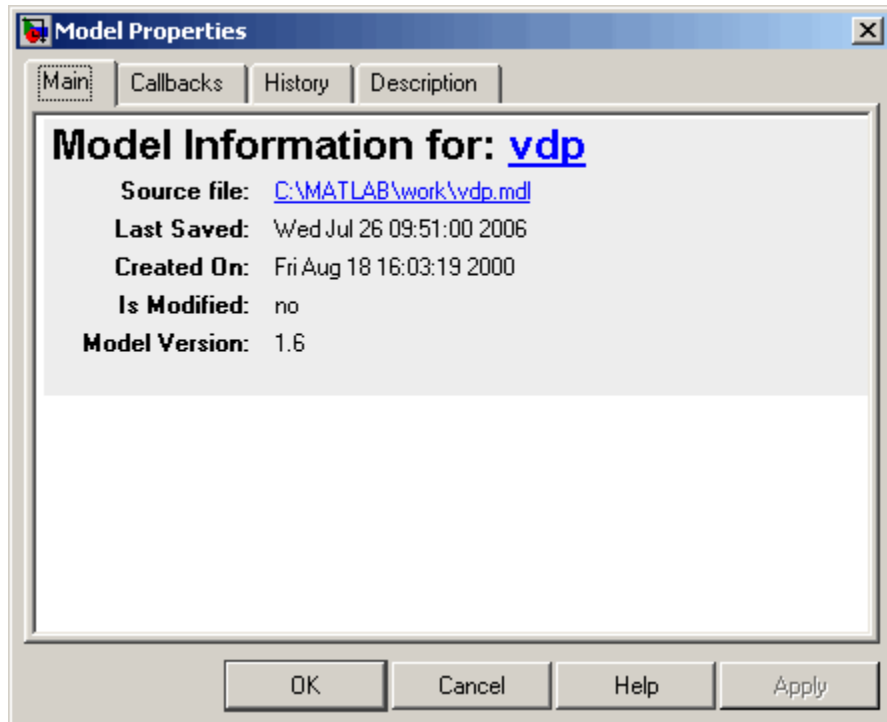
Manage Model Properties

You can use the Model Properties dialog box to view and edit model information (including some version parameters), callback functions, history, and the model description. To open the dialog box,

- In a model, select **File > Model Properties**.
- In a library, select **File > Library Properties**.

Library Properties includes an additional tab, Forwarding Table, for specifying the mapping from old library blocks to new library blocks. For information on using the Forwarding Table, see “Make Backward-Compatible Changes to Libraries” on page 28-22.

Model Properties and Library Properties both include the tabs Main Model Information, Callbacks, History and Description. The controls in the shared tabs are described next on this page.



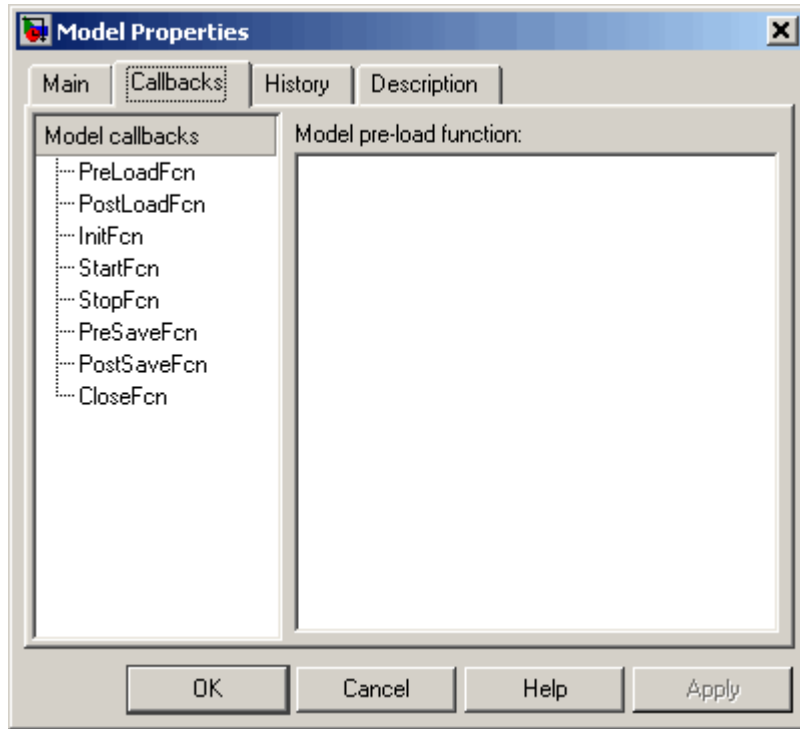
The dialog box includes the following panes.

Viewing Model Information

The **Main** pane summarizes information about the current version of this model, such as whether the model is modified, the Model Version, and Last Saved date. You can edit some of this information in the History tab, see “Viewing and Editing Model Information and History” on page 4-120.

Specifying Callbacks

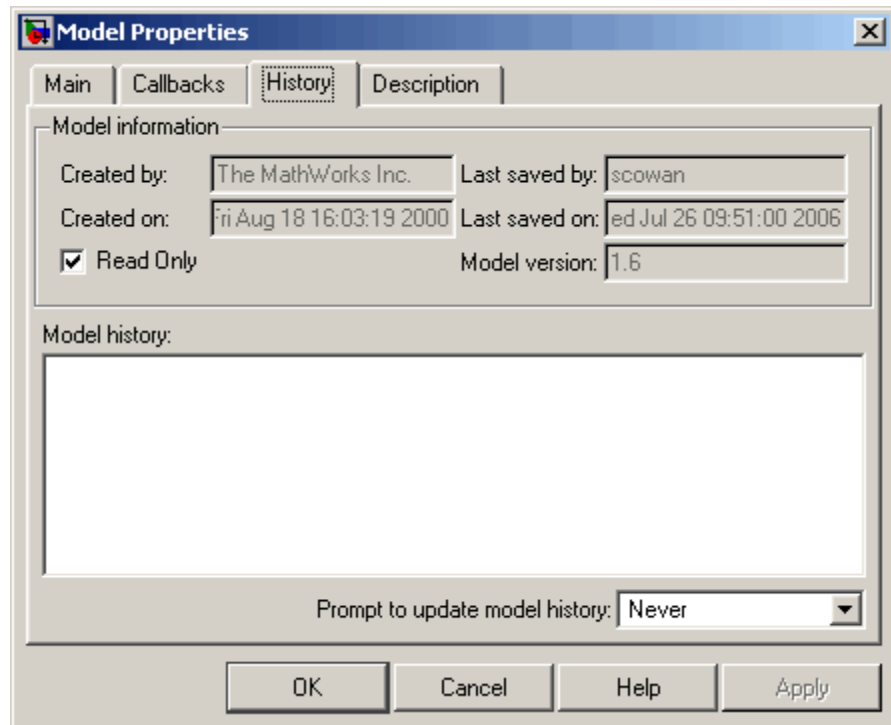
Use the **Callbacks** pane to specify functions to be invoked at specific points in the simulation of the model.



In the left pane, select the callback. In the right pane, enter the name of the function you want to be invoked for the selected callback. See “Create Model Callbacks” on page 4-58 for information on the callback functions listed on this pane.

Viewing and Editing Model Information and History

Use the **History** pane to view and edit model information, and to enable, view, and edit this model’s change history in the lower **Model history** field. To use the history controls see “Log Comments History” on page 4-125.



Viewing Model Information. When the **Read Only** check box is selected, you can view but not edit the following grayed out fields.

- **Created by**

Name of the person who created this model. The Simulink software sets this property to the value of the `USER` environment variable when you create the model.

- **Created on**

Date and time this model was created.

- **Last saved by**

Name of the person who last saved this model. The Simulink software sets the value of this parameter to the value of the `USER` environment variable when you save a model.

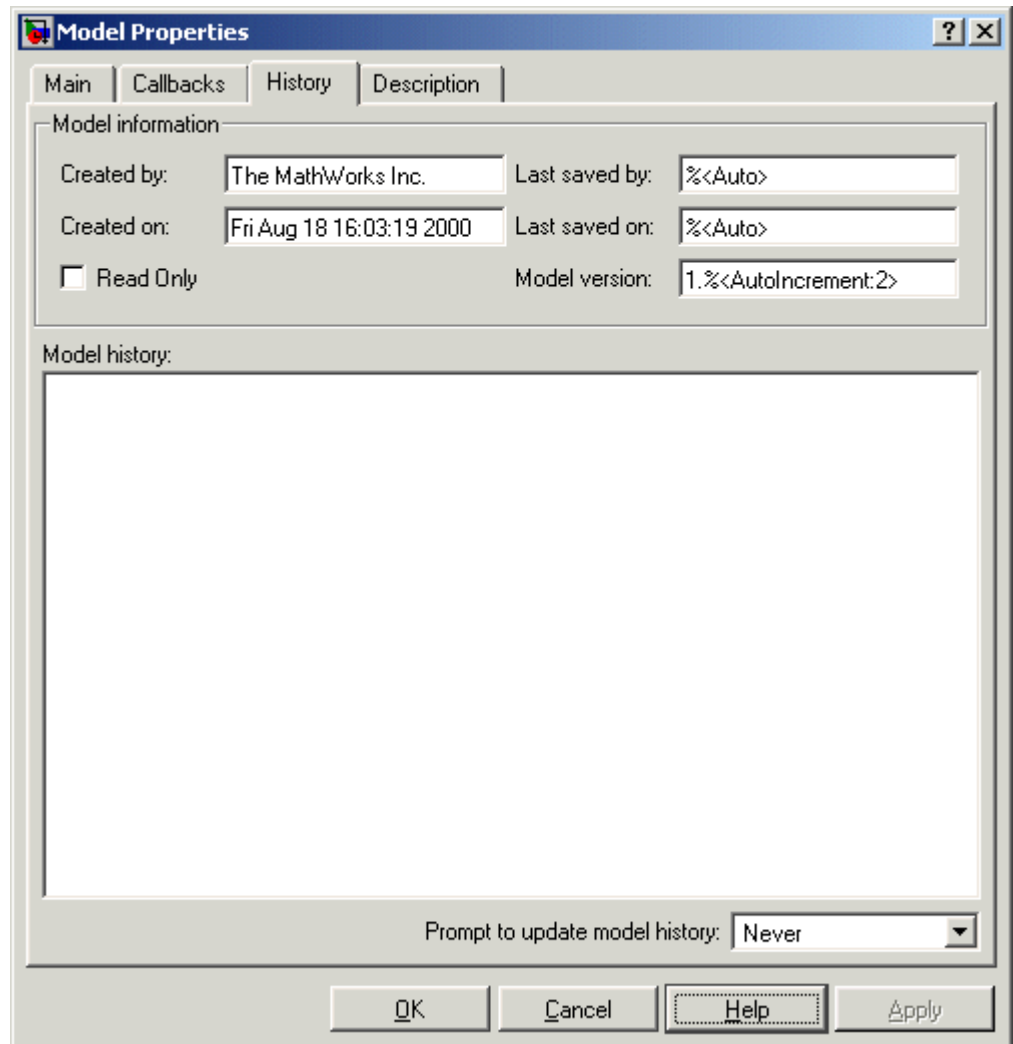
- **Last saved on**

Date that this model was last saved. The Simulink software sets the value of this parameter to the system date and time whenever you save a model.

- **Model version**

Version number for this model.

Editing Model Information. Clear the **Read Only** check box to edit model information. When the check box is deselected, the dialog box shows the format strings or values for the following fields. You can edit all but the **Created on** field, as described.



- **Created by**

Name of the person who created this model. The Simulink software sets this property to the value of the USER environment variable when you create the model. Edit this field to change the value.

- **Created on**

Date and time this model was created. Do not edit this field.

- **Last saved by**

Enter a format string describing the format used to display the **Last saved by** value in the **History** pane and the **ModifiedBy** entry in the history log and Model Info blocks. The value of this field can be any string. The string can include the tag `%<Auto>`. Simulink replaces occurrences of this tag with the current value of the `USER` environment variable.

- **Last saved on**

Enter a format string describing the format used to display the **Last saved on** date in the **History** pane and the **ModifiedOn** entry in the history log and the in Model Info blocks. The value of this field can be any string. The string can contain the tag `%<Auto>`. The Simulink software replaces occurrences of this tag with the current date and time.

- **Model version**

Enter a format string describing the format used to display the model version number in the **Model Properties** pane and in Model Info blocks. The value of this parameter can be any text string. The text string can include occurrences of the tag `%<AutoIncrement:#>` where `#` is an integer. Simulink replaces the tag with an integer when displaying the model's version number. For example, it displays the tag

```
1.%<AutoIncrement:2>
```

as

```
1.2
```

Simulink increments `#` by 1 when saving the model. For example, when you save the model,

```
1.%<AutoIncrement:2>
```

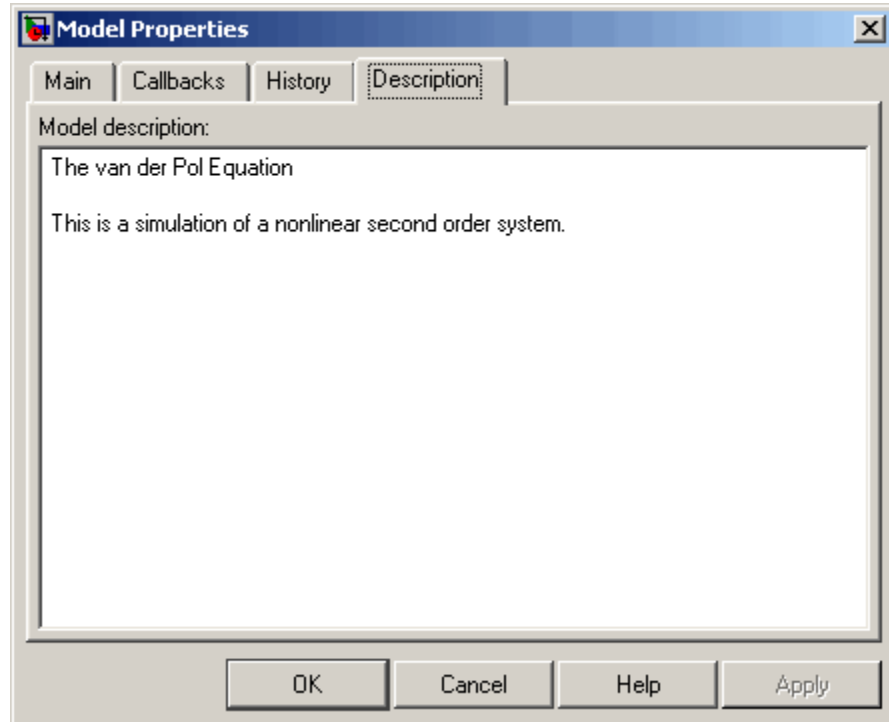
becomes

```
1.%<AutoIncrement:3>
```

and the model version number is reported as `1.3`.

Viewing and Editing the Model Description

Use the Description pane to enter a description of the model. You can view the model description by typing `help` followed by the model name at the MATLAB prompt. The contents of the **Model description** field appear in the Command Window.



Log Comments History

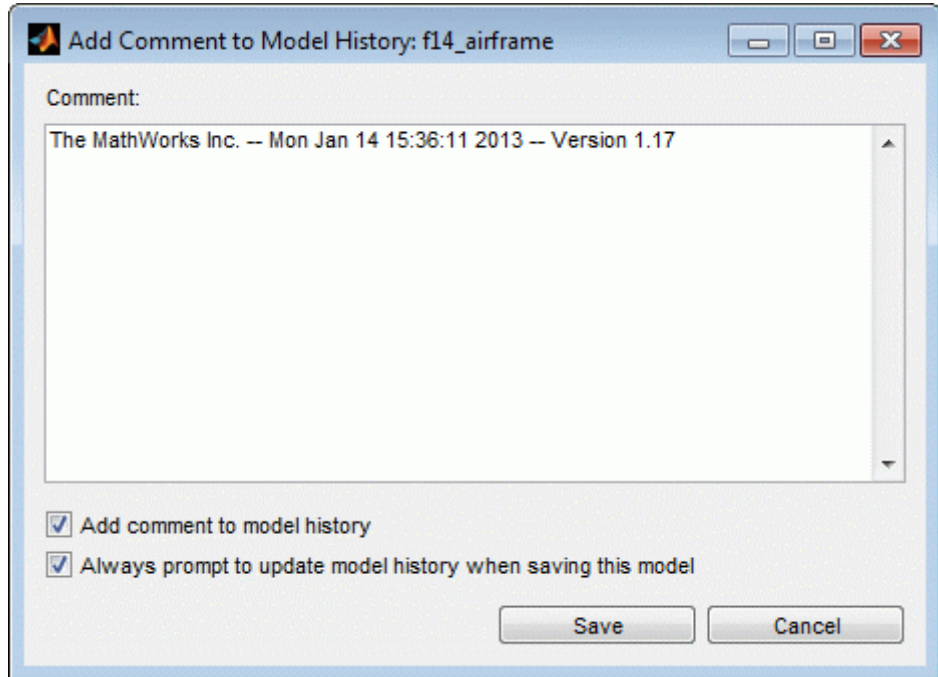
You can create and store a record of changes to a model in the model itself. The Simulink software compiles the history automatically from comments that you or other users enter when they save changes to a model. For more flexibility adding labels and comments to models and submissions, see instead “Simulink Projects”.

Logging Changes

To start a change history,

- 1 Select **File > Model Properties**.
- 2 In the Model Properties dialog box, select the **History** tab.
- 3 In the **Prompt to update model history** list, select **When saving model** and click **OK**.

The next time you save the model, the Add Comment to Model History dialog box prompts you to enter a comment.



For example you could describe changes that you have made to the model since the last time you saved it. To add an item to the model's change history, enter the item in the **Comment** edit field and click **Save**. The information is stored in the model's change history log.

If you do not want to enter an item for this session, clear the **Add comment to model history** check box.

To discontinue change logging, either:

- In the Add Comment to Model History dialog box, clear the check box **Always prompt to update model history when saving this model**.
- In the Model Properties dialog box History pane, select Never from the **Prompt to update model history** list.

Viewing and Editing the Model History Log

In the Model Properties dialog box you can view and edit the model history on the History tab.

The model history text field displays the history for the model in a scrollable text field. To change the model history, edit the contents of this field.

Version Information Properties

Some version information is stored as model parameters in a model. You can access this information from the MATLAB command line or from a MATLAB script, using the Simulink `get_param` command.

The following 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.

Property	Description
Description	<p>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</p> <pre>help 'mymodelname'</pre> <p>at the MATLAB command line.</p>
LastModifiedBy	Name of the user who last saved the model.
LastModifiedDate	Date when the model was last saved.
ModifiedBy	<p>Current value of the user name of the person who last modified this model. When you save, this information is saved with the file as LastModifiedBy.</p>
ModifiedByFormat	<p>Format of the ModifiedBy parameter. Value can be any string. The string can include the tag %<Auto>. The Simulink software replaces the tag with the current value of the USER environment variable.</p>
ModifiedDateFormat	<p>Format of the ModifiedDate parameter. Value can be any string. The string can include the tag %<Auto>. Simulink replaces the tag with the current date and time when saving the model.</p>
ModifiedComment	Comment entered by user who last updated this model.
ModifiedHistory	History of changes to this model.

Property	Description
ModelVersion	Version number.
ModelVersionFormat	Format of model version number. Can be any string. The string can contain the tag %<AutoIncrement:#> where # is an integer. Simulink replaces the tag with # when displaying the version number. It increments # when saving the model.

`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 “Simulink Projects”.

Model Discretizer

In this section...

“What Is the Model Discretizer?” on page 4-130

“Requirements” on page 4-130

“Discretize a Model with the Model Discretizer” on page 4-131

“View the Discretized Model” on page 4-140

“Discretize Blocks from the Simulink Model” on page 4-143

“Discretize a Model with the sldiscmdl Function” on page 4-153

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

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”
- Make sure your model does not contain any masked subsystems. For more information, see “Masking”.

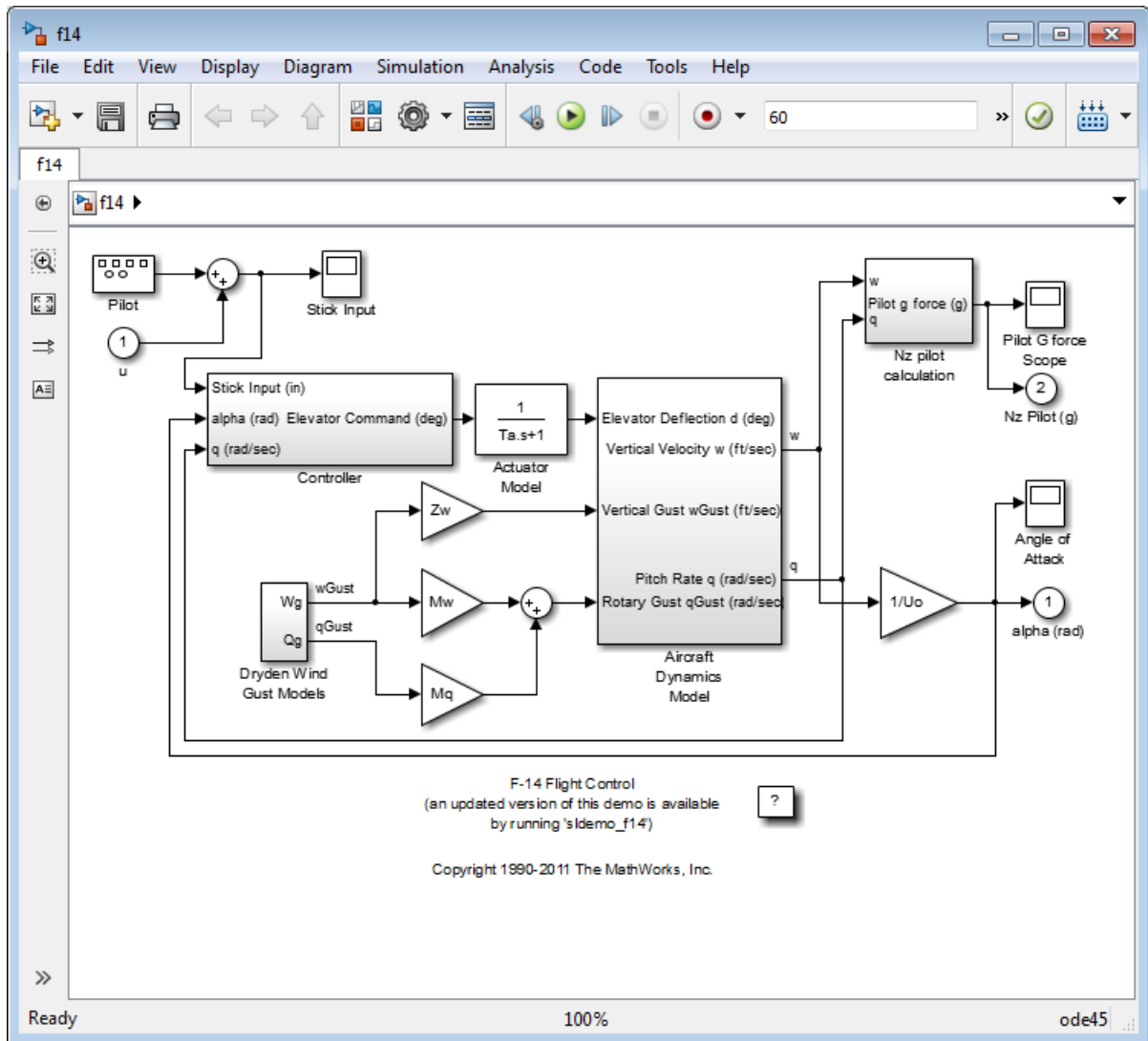
Discretize a Model with the Model Discretizer

To discretize a model:

- Start the Model Discretizer
- Specify the Transform Method
- Specify the Sample Time
- Specify the Discretization Method
- Discretize the Blocks

4 Creating a Model

The f14 model shows the steps in discretizing a model.



Start Model Discretizer

To open the tool, in the Simulink Editor, select **Analysis > Control Design > Model Discretizer**.

The **Simulink Model Discretizer** opens.



Alternatively, you can open Model Discretizer from the MATLAB Command Window using the `slmdliscui` function.

The following command opens the **Simulink Model Discretizer** window with the `f14` model:

```
slmdliscui('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 documentation.

The Transform method drop-down 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.

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 4-135.

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 4-135
Creates a discrete block whose parameters are retained from the corresponding continuous block.
- “Discrete blocks (Enter parameters in z-domain)” on page 4-136
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 4-137
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 4-138
Create multiple discretization candidates in z-domain for the current selection. A configurable subsystem can consist of one or more blocks.

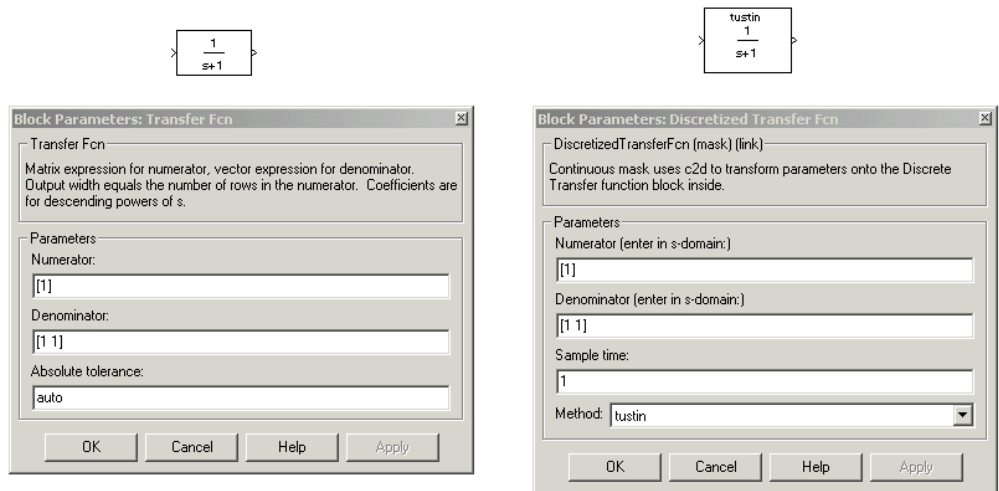
Discrete blocks (Enter parameters in s-domain). Creates a discrete block whose parameters are retained from the corresponding continuous block. The sample time and the discretization parameters are also on the block’s parameter dialog box.

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 4-134.

Note Parameters are not tunable when **Inline parameters** is selected 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. The Block Parameters dialog box for each block appears below the block.

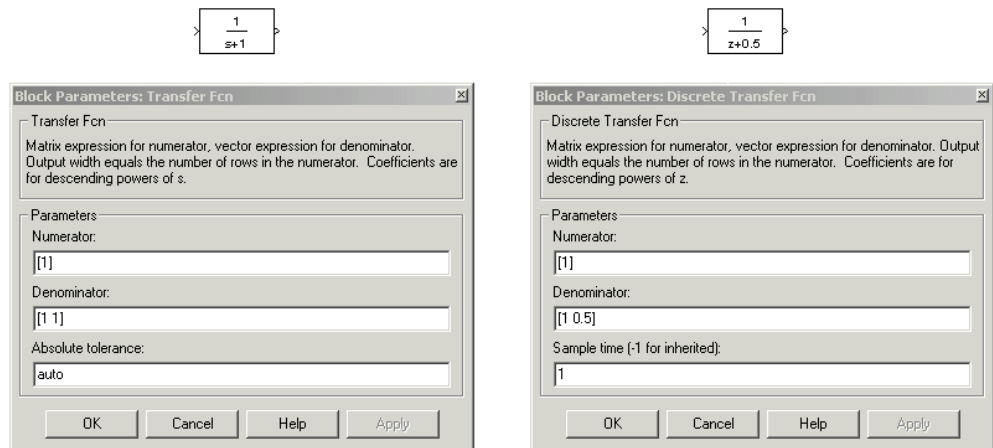


Discrete blocks (Enter parameters in z-domain). Creates a discrete block whose parameters are “hard-coded” values placed directly into the block’s dialog box. Model Discretizer uses the `c2d` function to obtain the discretized parameters, if needed.

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 **Link options > Go to library block** from the pop-up 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 **Link options -> Disable link** from the pop-up 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, shown below.



Undoing a Discretization

To undo a discretization, click the **Undo** discretization button, shown below.



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.

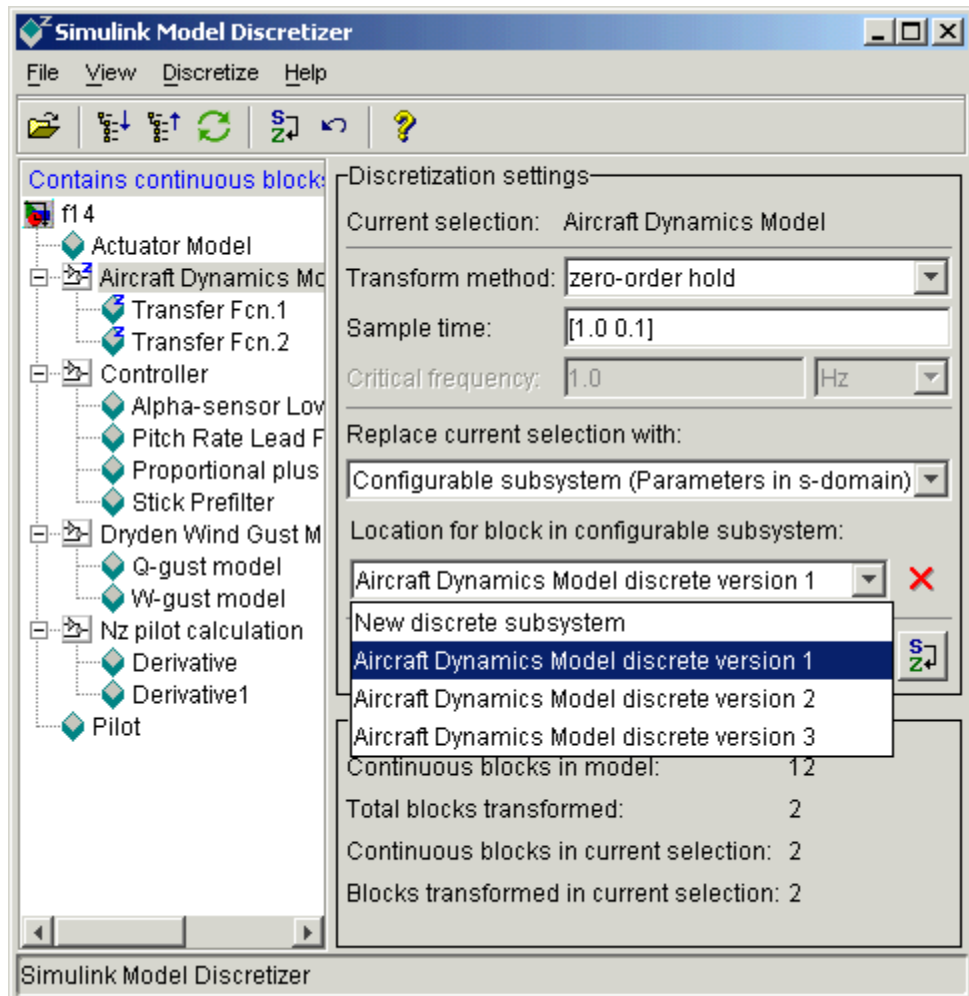
View the Discretized Model

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.

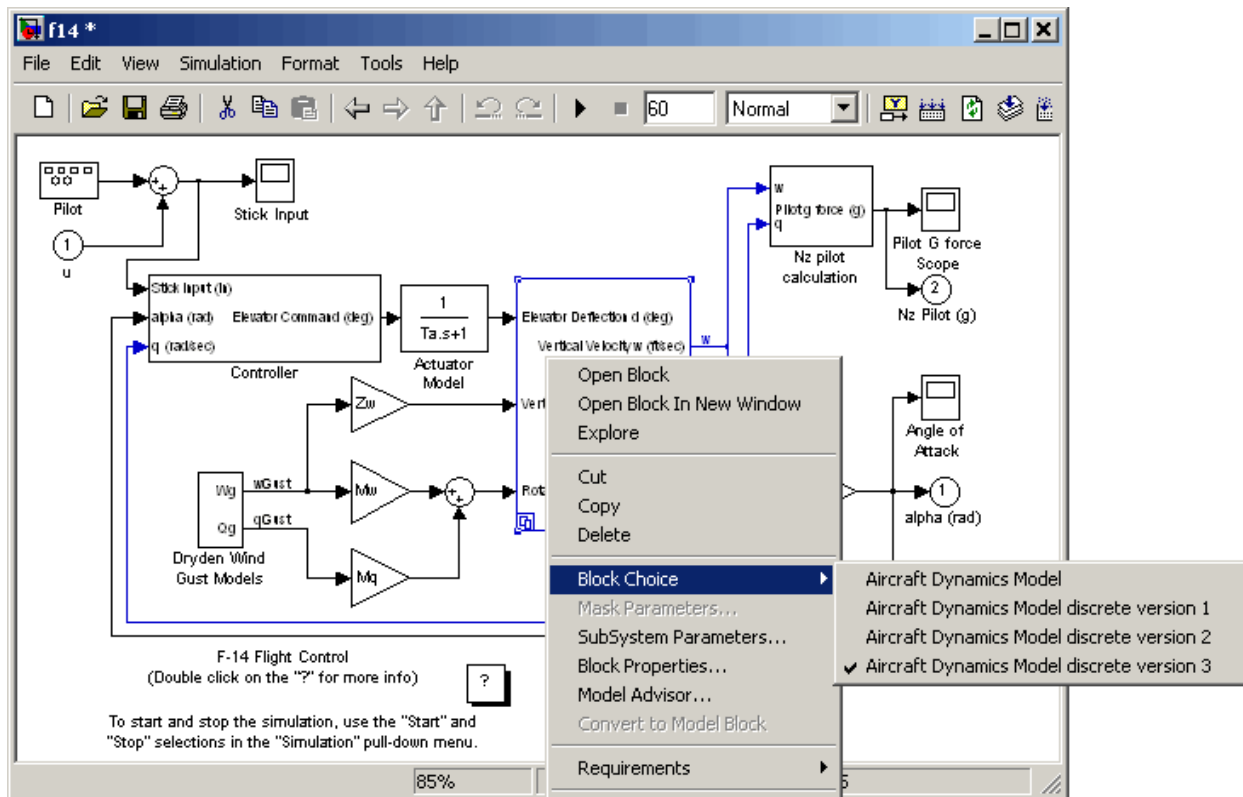
The following figure shows that the Aircraft Dynamics Model subsystem has been discretized into a configurable subsystem with three discretization candidates.



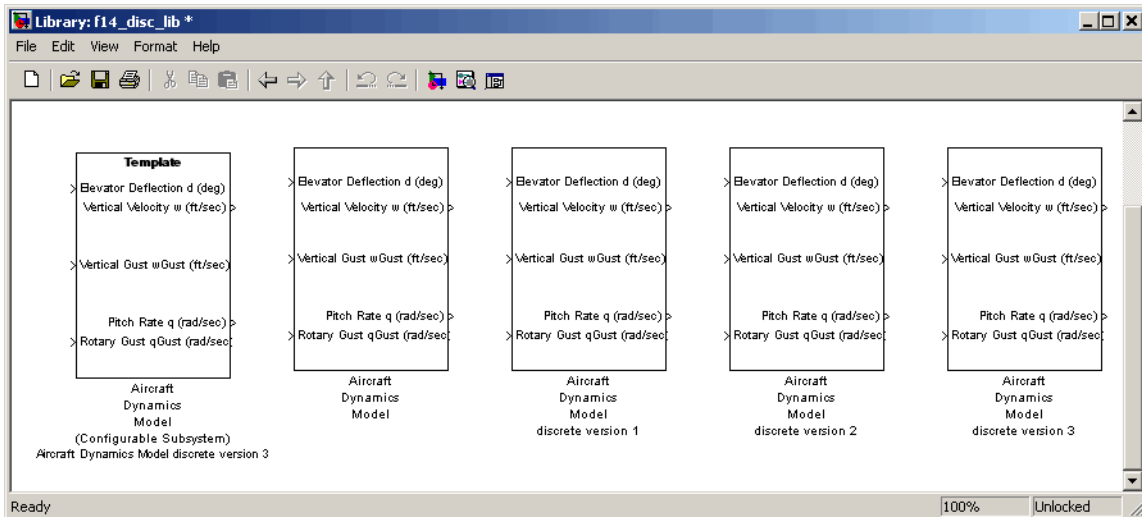
The other blocks in this f14 model have not been discretized.

4 Creating a Model

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 Model Discretizer's tree view of the model when the model has been changed, click the **Refresh** button, shown below.



Alternatively, you can select **Refresh** from the **View** menu.

Discretize Blocks from the Simulink Model

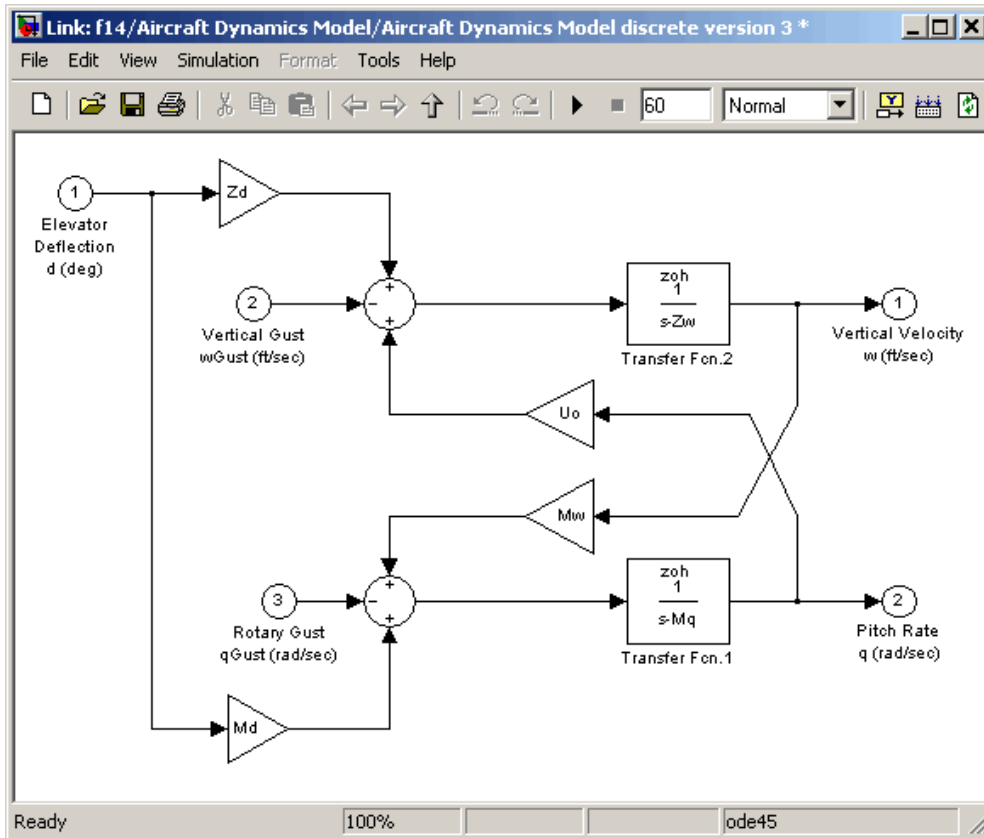
You can replace continuous blocks in a Simulink software model with the equivalent blocks discretized in the s-domain using the Discretizing library.

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

4 Creating a Model

in the s-domain with a zero-order hold transform method and a two second sample time.

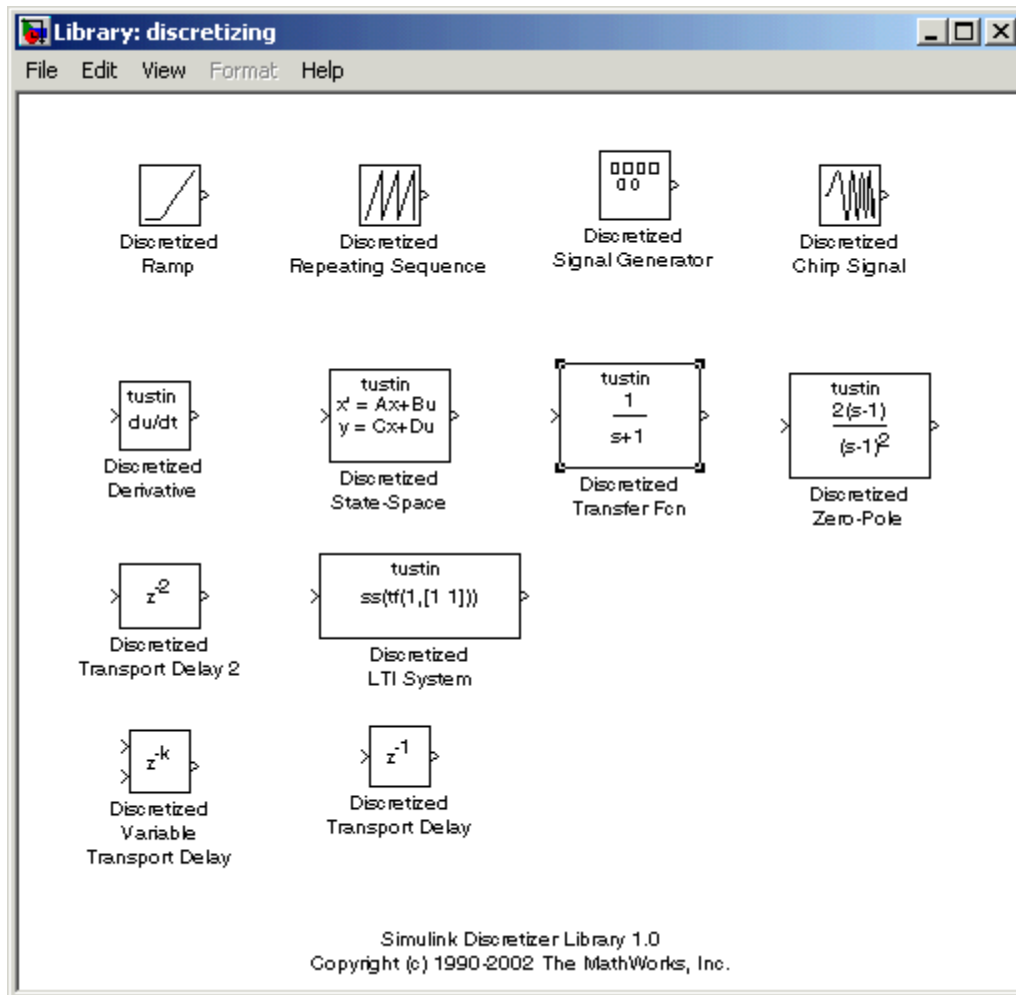
- 1 Open the f14 model.
- 2 Open the Aircraft Dynamics Model subsystem in the f14 model.



- 3 Open the Discretizing library window.

Enter discretizing at the MATLAB command prompt.

The **Library: discretizing** window opens.

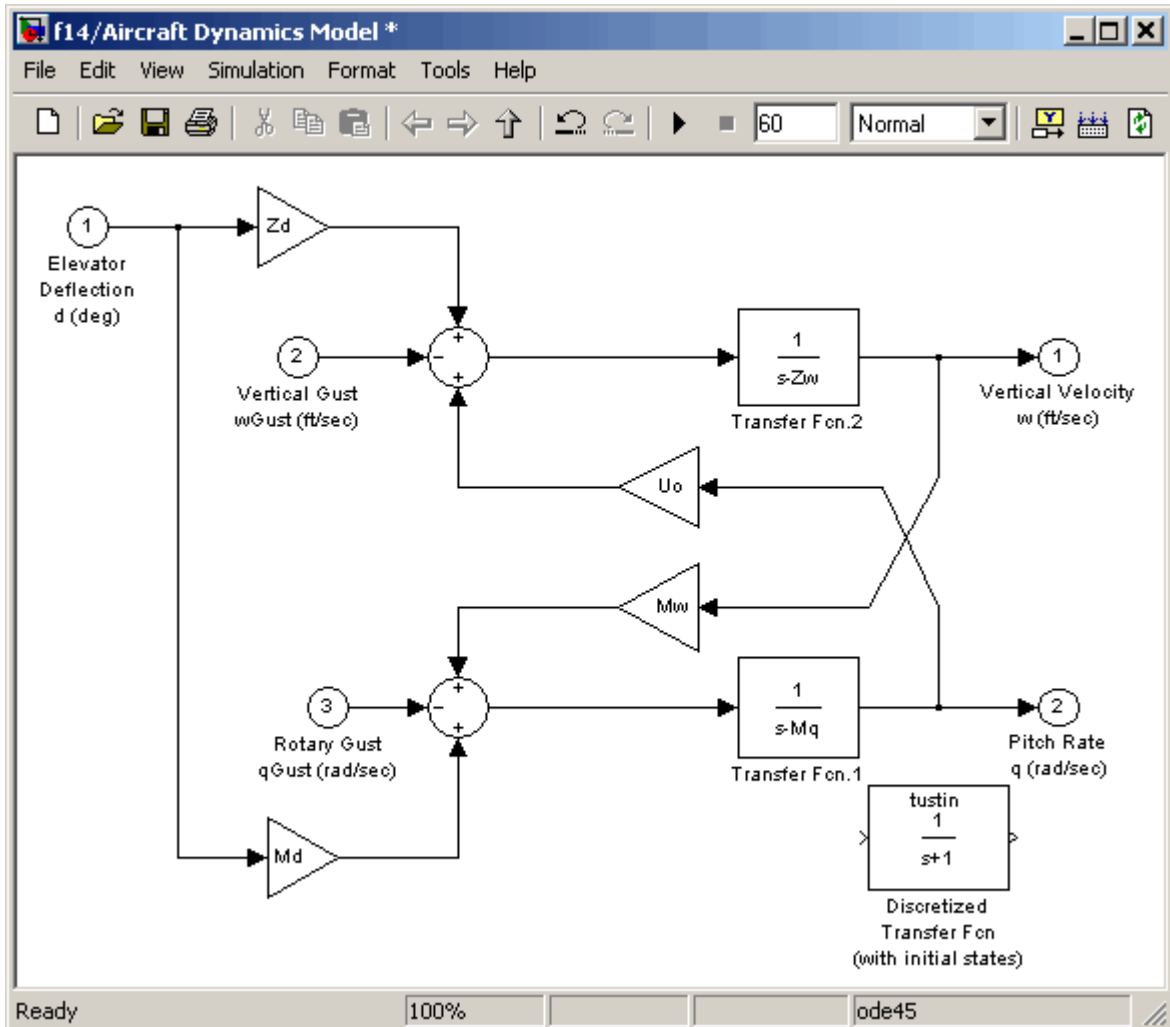


This library contains s-domain discretized blocks.

- 4 Add the Discretized Transfer Fcn (with initial states) block to the **f14/Aircraft Dynamics Model** window.

4 Creating a Model

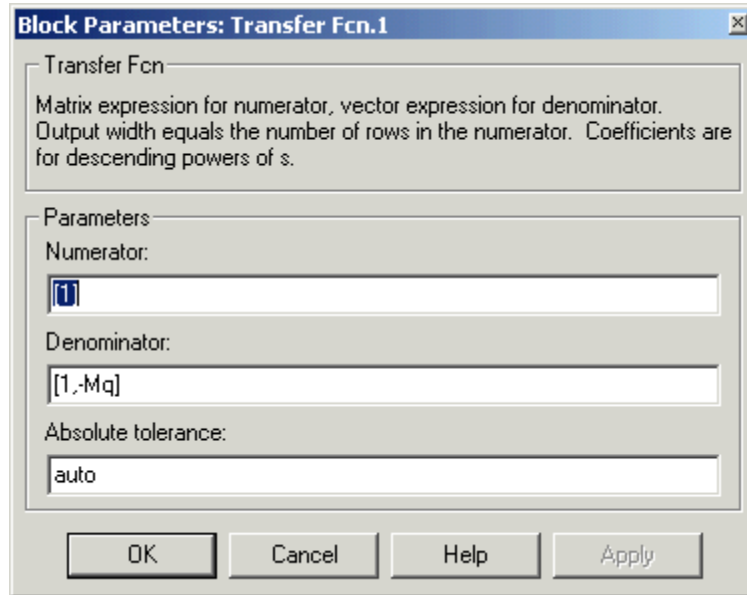
- a Click the Discretized Transfer Fcn block in the **Library: discretizing** window.
- b Drag it into the **f14/Aircraft Dynamics Model** window.



- 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.

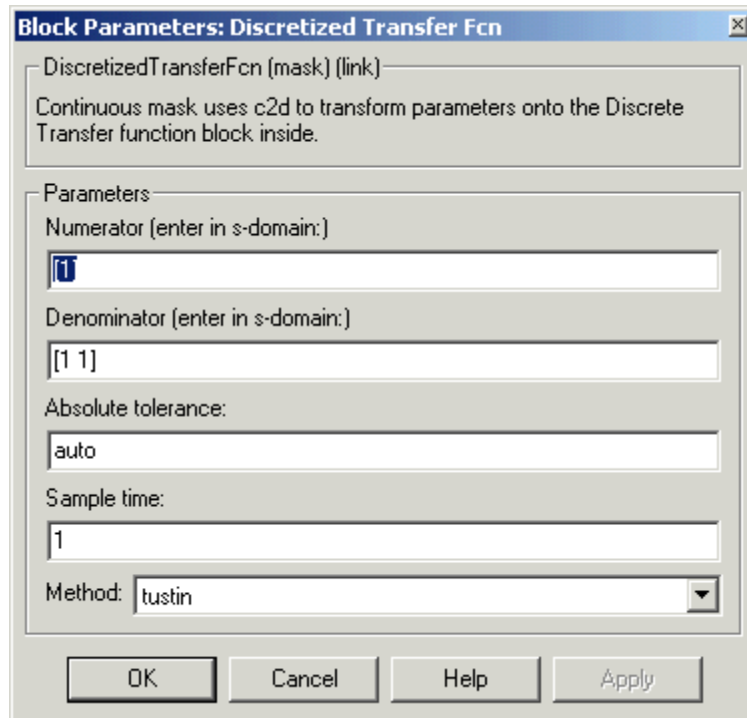
The Block Parameters: Transfer Fcn.1 dialog box opens.



- 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's dialog box to the Discretized Transfer Fcn block's dialog box.

Block Parameters: Discretized Transfer Fcn

DiscretizedTransferFcn (mask) (link)

Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside.

Parameters

Numerator (enter in s-domain:)

[1]

Denominator (enter in s-domain:)

[1,-Mq]

Absolute tolerance:

auto

Sample time:

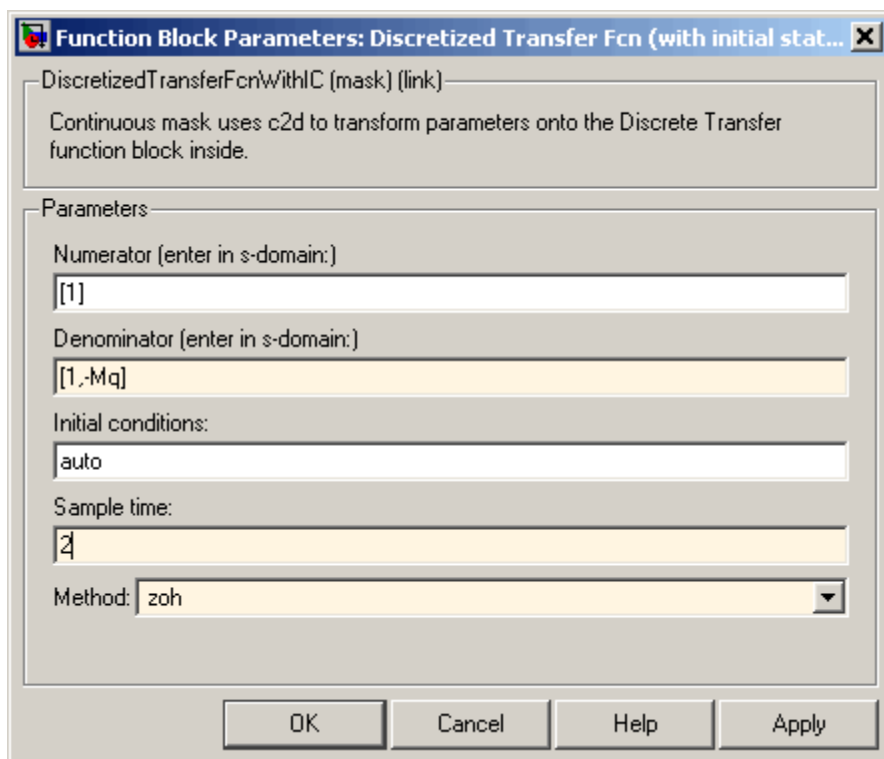
1

Method: tustin

OK Cancel Help Apply

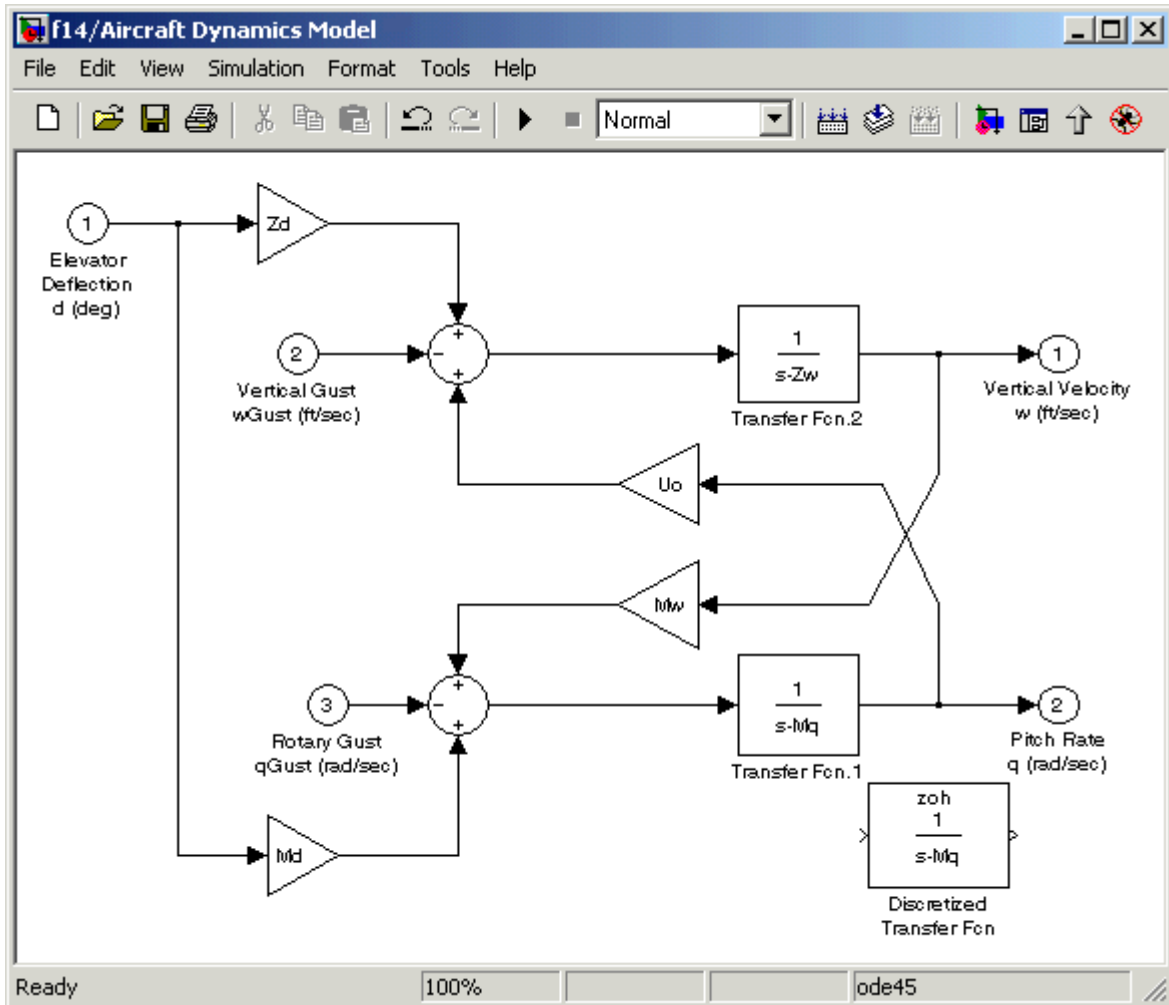
- 7** Enter 2 in the **Sample time** field.
- 8** Select zoh from the **Method** drop-down list.

The parameter dialog box for the Discretized Transfer Fcn now looks like this.



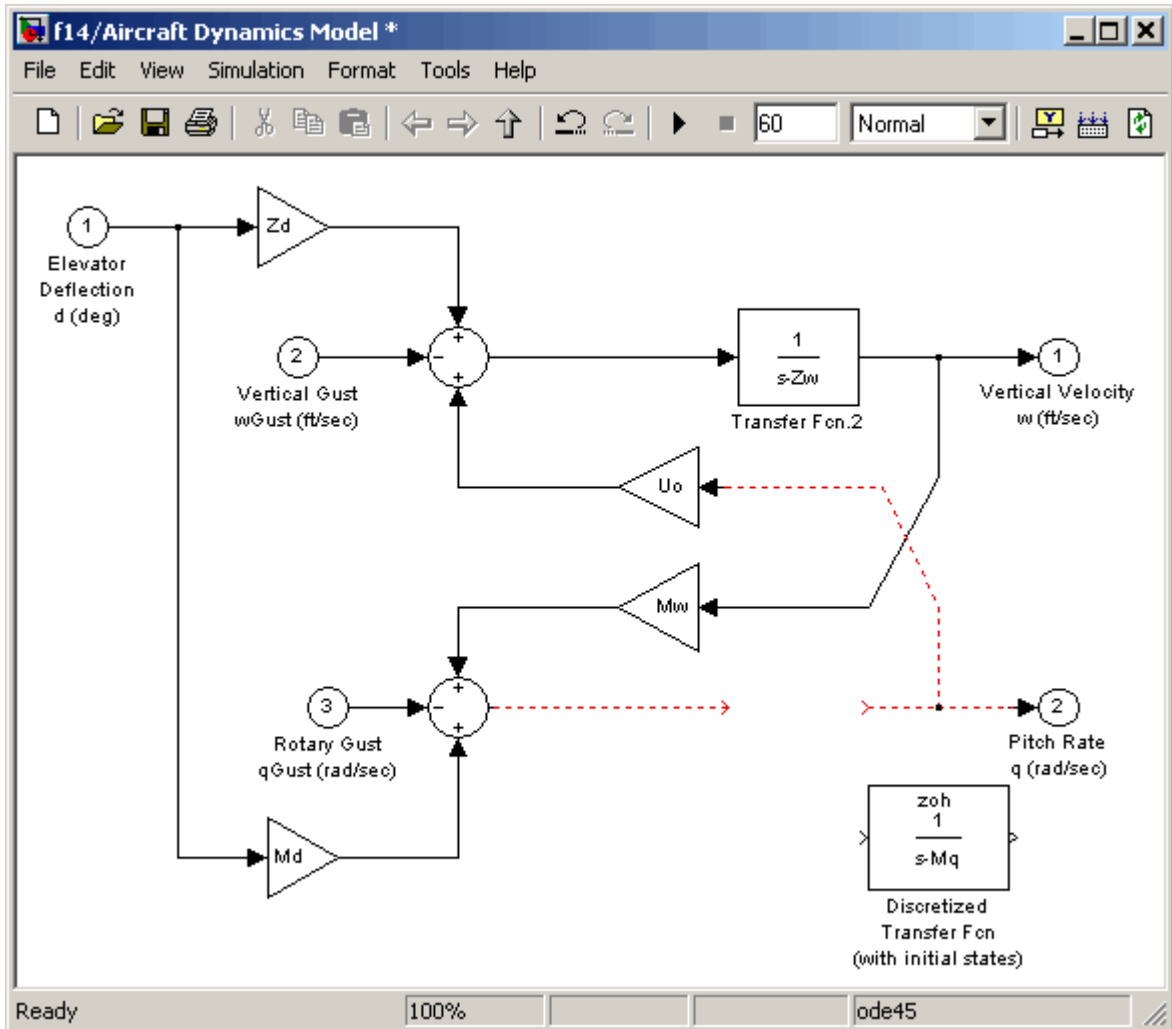
9 Click **OK**.

The f14/Aircraft Dynamics Model window now looks like this.



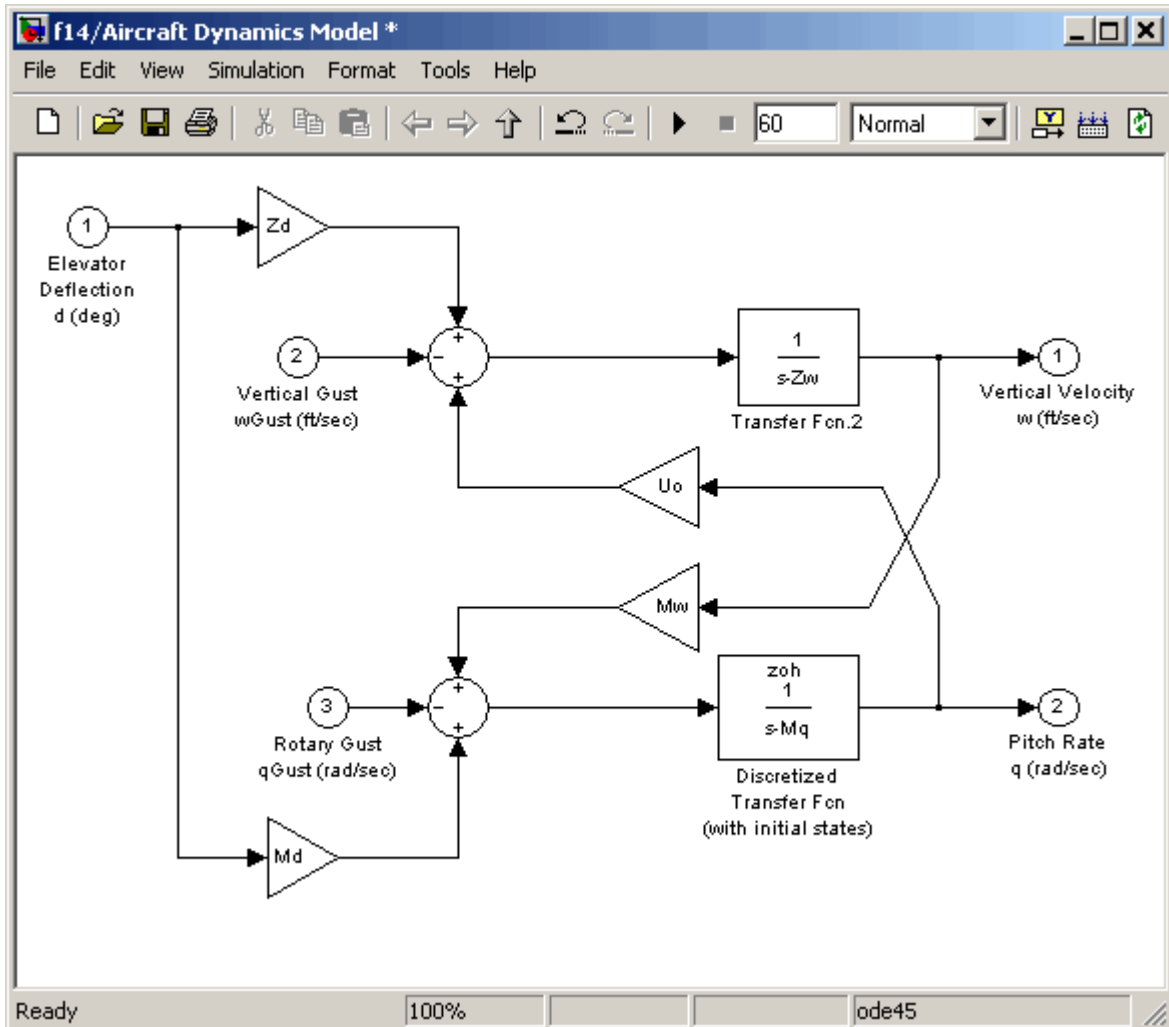
- 10 Delete the original Transfer Fcn.1 block.
 - a Click the Transfer Fcn.1 block.
 - b Press the **Delete** key.

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')
```

Working with Sample Times

- “What Is Sample Time?” on page 5-2
- “Specify Sample Time” on page 5-3
- “View Sample Time Information” on page 5-9
- “Print Sample Time Information” on page 5-14
- “Types of Sample Time” on page 5-15
- “Block Compiled Sample Time” on page 5-21
- “Sample Times in Subsystems” on page 5-24
- “Sample Times in Systems” on page 5-26
- “Resolve Rate Transitions” on page 5-32
- “How Propagation Affects Inherited Sample Times” on page 5-33
- “Backpropagation in Sample Times” on page 5-35

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.

Sample times can also be discrete, continuous, fixed in minor step, inherited, constant, variable, triggered, or asynchronous. The following sections discuss these sample time types, as well as sample time propagation and rate transitions between block-based or port-based sample times. You can use this information to control your block execution rates, debug your model, and verify your model.

Specify Sample Time

In this section...

- “Designate Sample Times” on page 5-3
- “Specify Block-Based Sample Times Interactively” on page 5-5
- “Specify Port-Based Sample Times Interactively” on page 5-6
- “Specify Block-Based Sample Times Programmatically” on page 5-7
- “Specify Port-Based Sample Times Programmatically” on page 5-8
- “Access Sample Time Information Programmatically” on page 5-8
- “Specify Sample Times for a Custom Block” on page 5-8
- “Determining Sample Time Units” on page 5-8
- “Change the Sample Time After Simulation Start Time” on page 5-8

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 5-15. 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 5-9.)

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	$[\text{Inf}, 0]$	magenta	Inf	Yes
Variable	$[-2, T_{vo}]$	brown	V1, V2,... Vi	No
Hybrid	N/A	yellow	N/A	No
Triggered	Source: D1, Source: D2, ...Source: Di	cyan	T1, T2,... Ti	No
Asynchronous	$[-1, -n]$	purple	A1, A2,... Ai	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 Reference”.)

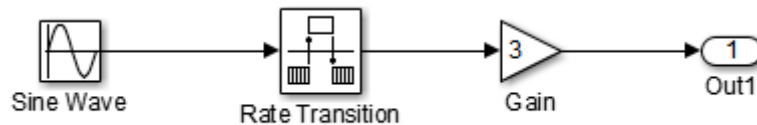
For example, suppose that a model defines three sample times: 1, 2, and 3. Further, suppose that it references a model that defines two sample times: 2 and 3. In this case, blocks operating at the 2 sample rate appear as green in the parent model and as red in the referenced model.

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.

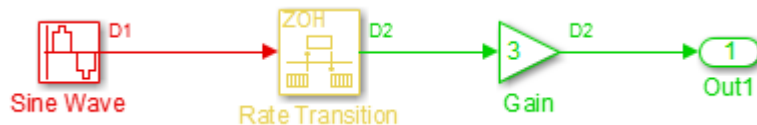
You can use the explicit sample time values in this table to specify sample times interactively or programmatically for either block-based or port-based sample times.

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.)



Sample Time 0.1 Sample Time 0.2

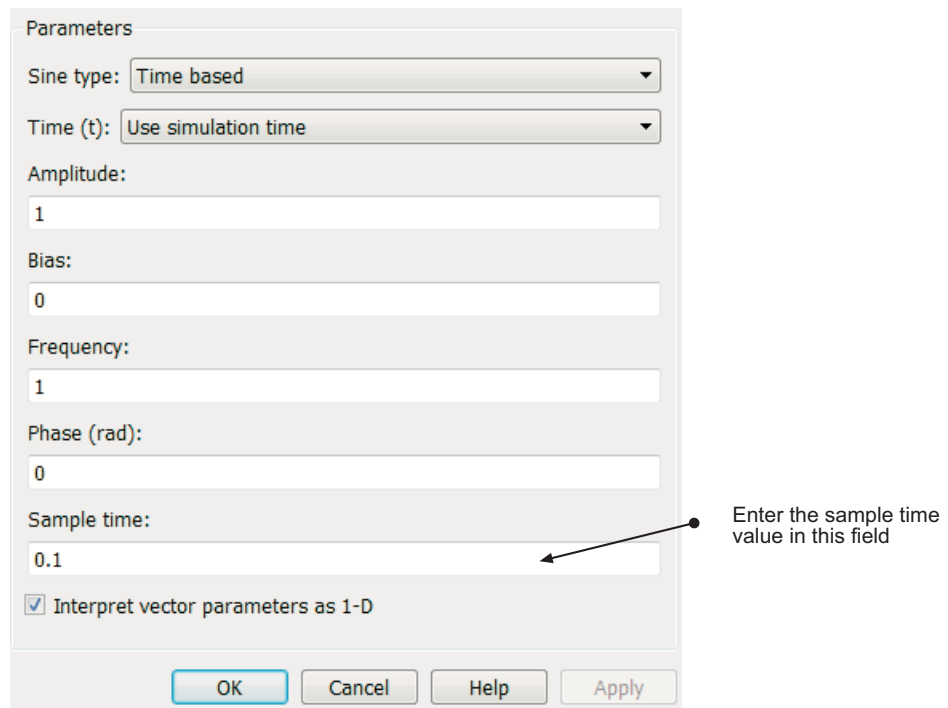
`ex_specify_sample_time` after Setting Sample Times

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.

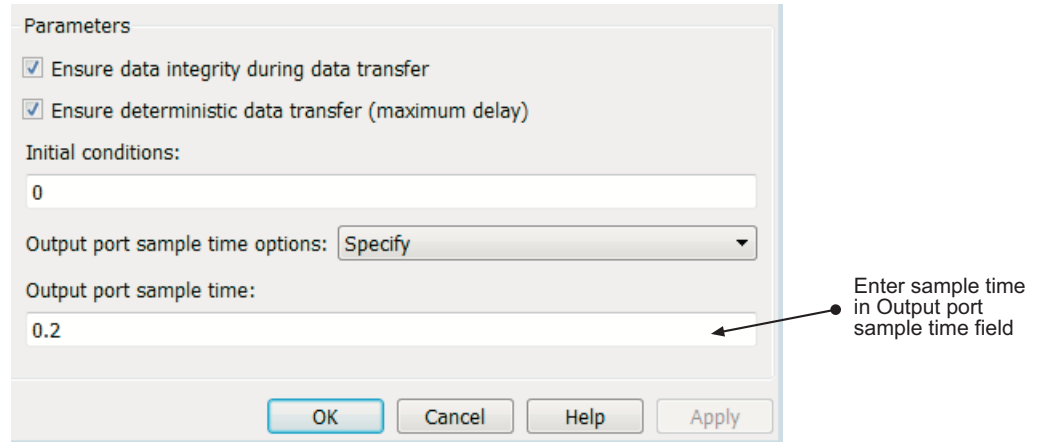


Specify Port-Based Sample Times Interactively

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 “Sample Times”.

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.

View Sample Time Information

In this section...
“View Sample Time Display” on page 5-9
“Sample Time Legend” on page 5-10

View Sample Time Display

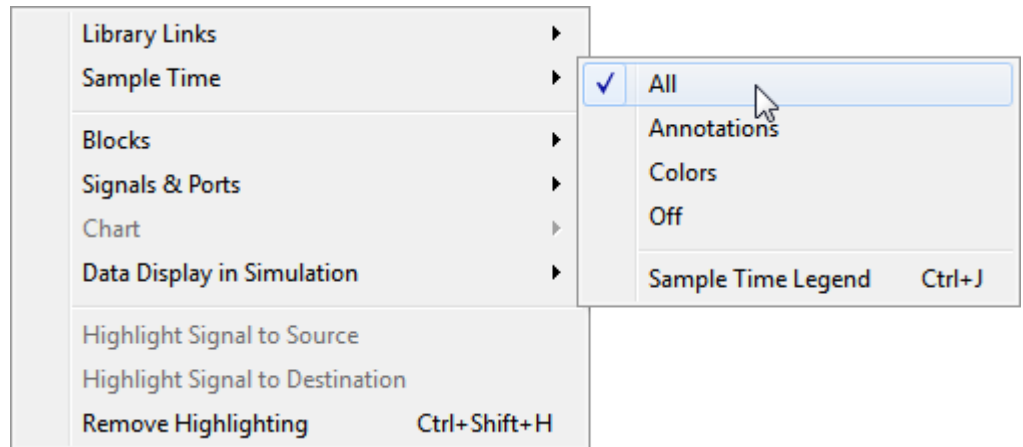
Simulink models can display color coding and annotations that represent specific sample times. As shown in the table Designations of Sample Time Information on page 5-4, each sample time type has one or more colors associated with it. You can display the blocks and signal lines in color, the annotations in black, or both the colors and the annotations. To choose one of these options:

- 1 In the Simulink model window, select **Display > Sample Time**.
- 2 Select **Colors**, **Annotations**, or **All**.

Selecting **All** results in the display of both the colors and the annotations. Regardless of your choice, Simulink performs an **Update Diagram** automatically. To turn off the colors and annotations:

- 1 Select **Display > Sample Time**.
- 2 Select **Off**.

Simulink performs another **Update Diagram** automatically.



Your Sample Time Display choices directly control the information that the Sample Time Legend displays.

Note The discrete sample times in the table Designations of Sample Time Information on page 5-4 represent a special case. Five colors indicate the fastest 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.

Sample Time Legend

You can view the Sample Time 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.

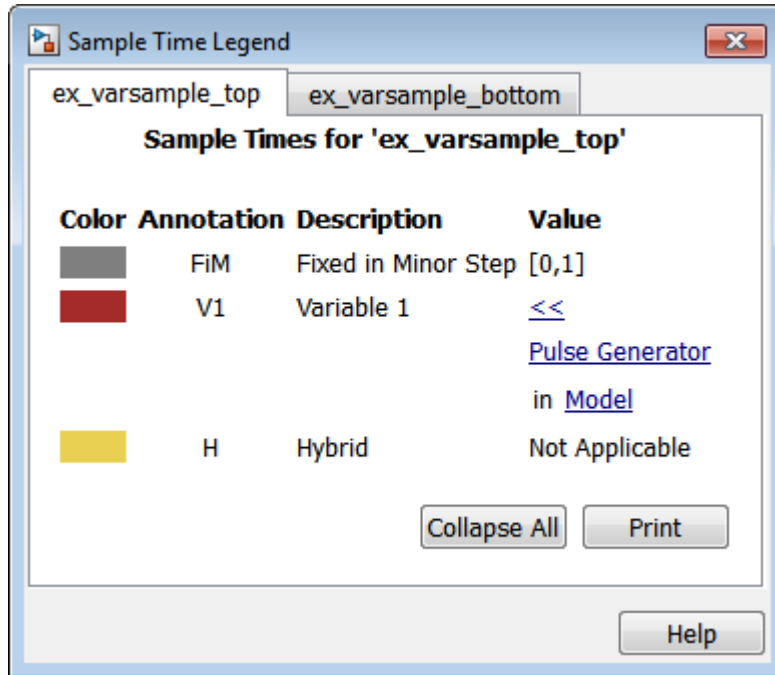
Viewing the Legend

To assist you with interpreting your block diagram, a **Sample Time Legend** is available that contains the sample time color, annotation, description, and value for each sample time in the model. You can use one of three methods to view the legend, but upon first opening the model, you must first perform an **Update Diagram**.

- 1 In the Simulink model window, select **Simulation > Update Diagram**.
- 2 Select **Display > Sample Time > Sample Time Legend** or press **Ctrl +J**.

In addition, whenever you select **Colors**, **Annotations**, or **All** from the **Sample Time** menu, Simulink updates the model diagram and opens the legend by default. The legend contents reflect your **Sample Time Display** choices. By default or if you have selected **Off**, the legend contains a description of the sample time and the sample time value. If you turn colors on, the legend displays the appropriate color beside each description. Similarly, if you turn annotations on, the annotations appear in the legend.

The legend does not provide a discrete rate for all types of sample times. For asynchronous and variable sample times, the legend displays a link to the block that controls the sample time in place of the sample time value. Clicking one of these links highlights the corresponding block in the block diagram. For variable sample times, the legend uses >> to indicate that the controller block of the sample time is in a model reference hierarchy. Click >> to drill down to that block, for example:



The rate listed under hybrid and asynchronous hybrid models is “Not Applicable” because these blocks do not have a single representative sample time.

Note The Sample Time 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 (i.e., 0.1 and 0.2) appear in the legend.

For subsequent viewings of the legend, repeat the **Update Diagram** to access the latest known information.

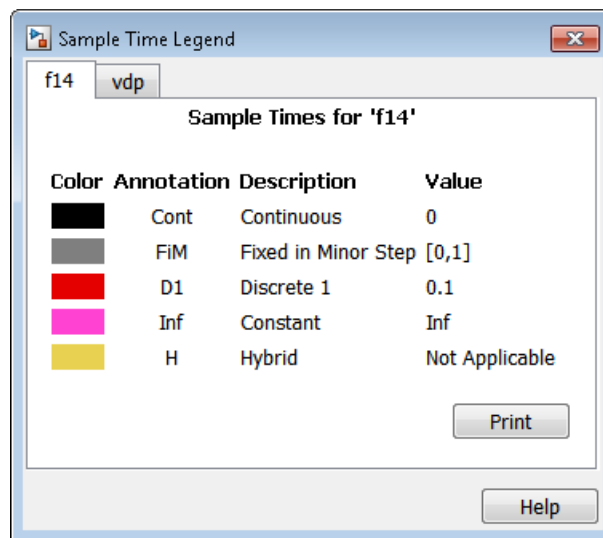
Turning the Legend Off

If you do not want to view the legend upon selecting **Sample Time Display**:

- 1 In the Simulink model window, select **File > Simulink Preferences**.
- 2 Scroll to the bottom of the main **Preferences** pane.
- 3 Clear **Open the sample time legend whenever the sample time display is changed**.

Viewing Multiple Legends

If you have more than one model open and you view the **Sample Time Legend** for each one, a single legend window appears with multiple tabs. Each tab contains the name of the model and the respective legend information. The following figure shows the tabbed legends for the f14 and vdp models.



Print Sample Time Information

You can print the sample time information in the Sample Time Legend by using either of these methods:

- In the Sample Time Legend window, click **Print**.
- Print the legend from the **Print Model** dialog box:
 - 1** In the model window, select **File > Print**.
 - 2** Under **Options**, select the check box beside **Print Sample Time Legend**.
 - 3** Click **OK**.

Types of Sample Time

In this section...

“Discrete Sample Time” on page 5-15
 “Continuous Sample Time” on page 5-16
 “Fixed in Minor Step” on page 5-16
 “Inherited Sample Time” on page 5-16
 “Constant Sample Time” on page 5-17
 “Variable Sample Time” on page 5-18
 “Triggered Sample Time” on page 5-19
 “Asynchronous Sample Time” on page 5-19

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:

$$0 \leq n \leq \frac{T_{sim}}{T_s}$$

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 (see “Minor Time Steps” on page 3-23). 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. The fixed in minor step sample time is also known as the zero-order hold sample time.

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 5-9.)

Examples of inherited blocks include the Gain and Add blocks.

All inherited blocks are subject to the process of sample time propagation, as discussed in “How Propagation Affects Inherited Sample Times” on page 5-33

Constant Sample Time

Specifying a constant (Inf) sample time is a request for an optimization for which the block executes only once during model initialization. Simulink honors such requests if all of the following conditions hold:

- The **Inline parameters** check box is selected on the Configuration Parameters dialog box **Optimization > Signal and Parameters** pane (see the “Optimization Pane: Signals and Parameters”).
- The block has no continuous or discrete states.
- The block allows for a constant sample time.
- The block does not drive an output port of a conditionally executed subsystem (see “Create an Enabled Subsystem” on page 7-17).
- The block has no tunable parameters.

One exception is an empty subsystem. A subsystem that has no blocks—not even an input or output block—always has a constant sample time regardless of the status of the conditions listed above.

This optimization process speeds up the simulation by eliminating the need to recompute the block output at each step. Instead, during each model update, Simulink first establishes the constant sample time of the block and then computes the initial values of its output ports. Simulink then uses these values, without performing any additional computations, whenever it needs the outputs of the block.

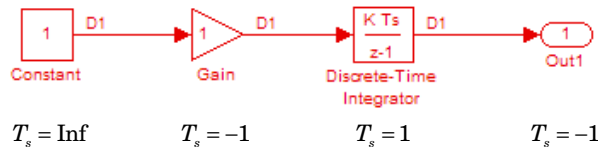
Note The Simulink block library includes several blocks, such as the MATLAB S-Function block, the Level-2 MATLAB S-Function block, and the Model block, whose ports can produce outputs at different sample rates. It is possible for some of the ports of such blocks to have a constant sample time. These ports produce outputs only once—at the beginning of a simulation. Any other ports produce output at their respective sample times.

If Simulink cannot honor the request to use the optimization, then Simulink treats the block as if it had specified an inherited sample time (-1).

One specific case for which Simulink rejects the request is when parameters are tunable. By definition, you can change the parameter values of a block having tunable parameters; therefore, the block outputs are variable rather than constant. For such cases, Simulink performs sample time propagation (see “How Propagation Affects Inherited Sample Times” on page 5-33) to determine the block sample time.

An example of a Constant block with tunable parameters is shown below (ex_inline_parameters_off). Since the **Inline parameters** option is disabled, the **Constant value** parameter is tunable and the sample time cannot be constant, even if it is set to Inf. To ensure accurate simulation results, Simulink treats the sample time of the Constant block as inherited and uses sample time propagation to calculate the sample time. The Discrete-Time Integrator block is the first block, downstream of the Constant block, which does not inherit its sample time. The Constant block, therefore, inherits the sample time of 1 from the Integrator block via backpropagation.

Inline Parameters Off



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”.

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 have either a triggered or a constant 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 “Create a Function-Call Subsystem” on page 7-48.)
- 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 “Handle Asynchronous Events”.)

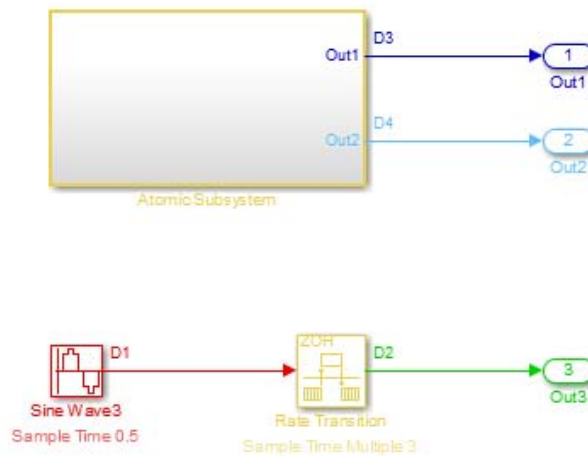
- The sample time is $[-1, -n]$.

For an explanation of how to use blocks to model and generate code for asynchronous event handling, see “Rate Transitions and Asynchronous Blocks”.

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');
```

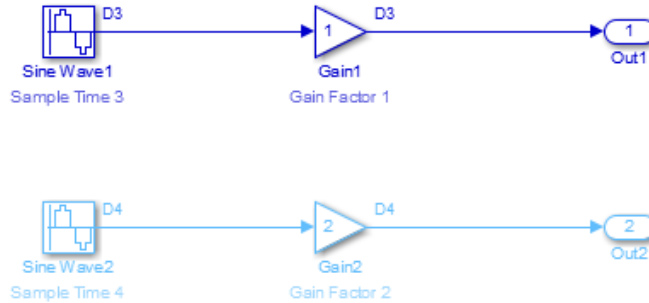
For the Sine Wave3 block,

```
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
```

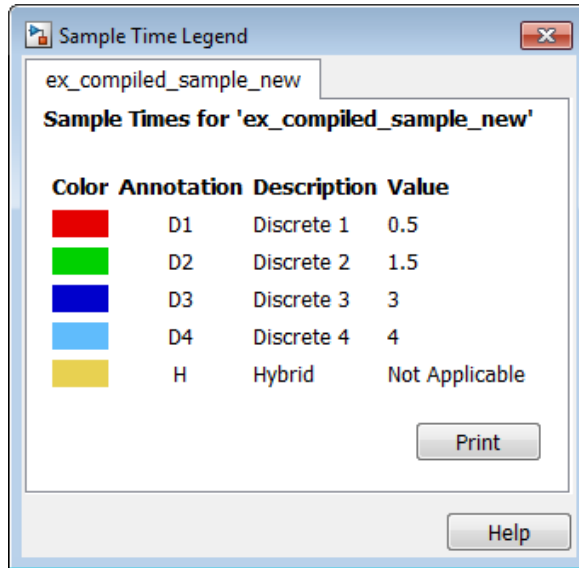
The greatest common divisor (GCD) of the two rates is 1. However, this is not necessarily one of the rates in the model.

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.



Related Examples

- “Sample Times in Subsystems” on page 5-24
- “View Sample Time Information” on page 5-9

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 `SampleTime` 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.

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')
```

Concepts

- “Block Compiled Sample Time” on page 5-21
- “Sample Times in Systems” on page 5-26

Sample Times in Systems

In this section...
“Purely Discrete Systems” on page 5-26
“Hybrid Systems” on page 5-29

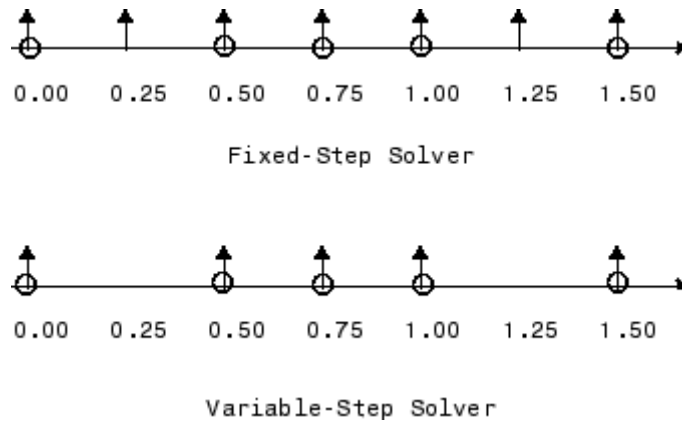
Purely Discrete Systems

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.

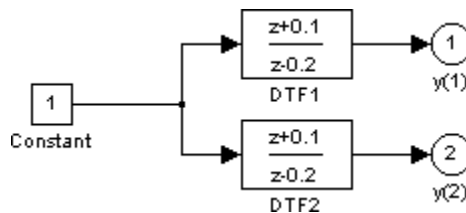
The importance of the fundamental sample time directly relates to whether you direct the Simulink software to use a fixed-step or a variable-step discrete solver to solve your multirate discrete system. A fixed-step solver sets the simulation step size equal to the fundamental sample time of the discrete 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's **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.

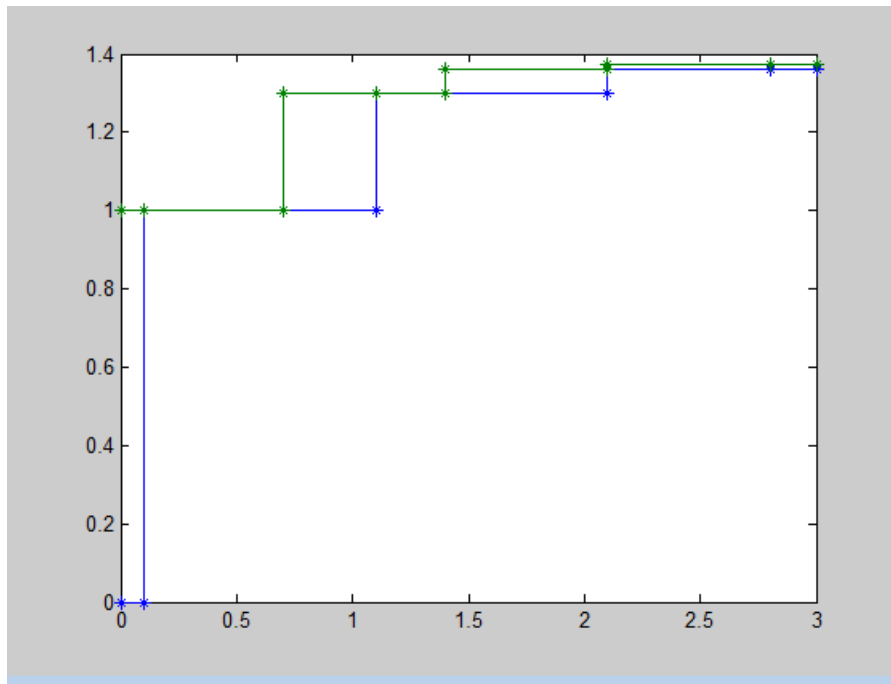


Running the simulation and plotting the outputs using the stairs function

```
simOut = sim('ex_dtf','StopTime', '3');  
t = simOut.find('tout')
```

```
y = simOut.find('yout')  
stairs(t,y, '-*')
```

produces the following plot.



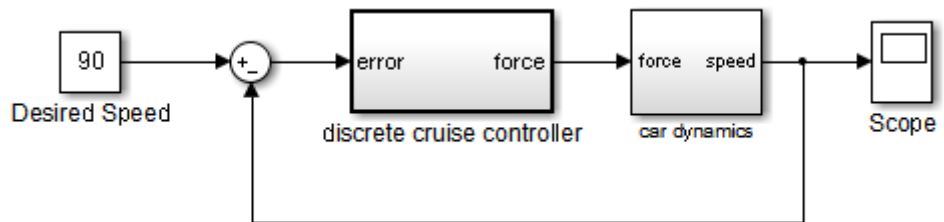
(For information on the `sim` command, see “Run Simulation Using the `sim` Command” on page 15-3.)

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

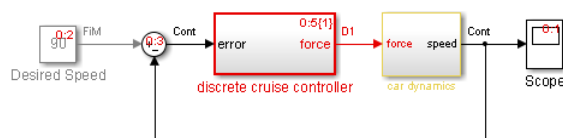
Hybrid systems contain both discrete and continuous blocks and thus have both discrete and continuous states. However, Simulink solvers treat any system that has both continuous and discrete sample times as a hybrid system. For information on modeling hybrid systems, see “Modeling Hybrid Systems” on page 3-8.

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`.)

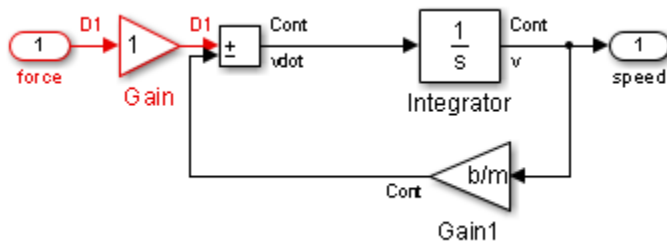


Car Model

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.

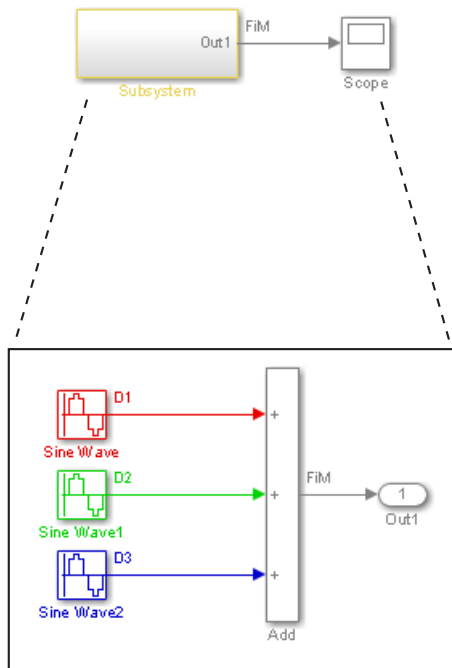


Car Model after an Update Diagram



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 $[\text{inf}, 0]$ or asynchronous $[-1, -n]$ sample times. Thus a subsystem consisting of one block with a constant sample time 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.

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:

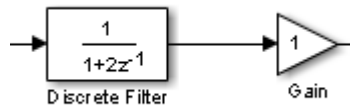
- A constant sample time ($[Inf, 0]$) never has a rate transition with any other rate.
- A continuous sample time (black) and the fastest discrete rate (red) never has a rate transition if you use a fixed-step solver.
- A variable sample time and fixed in minor step do not have a rate transition.

You can resolve rate transitions 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.

For a detailed discussion on rate transitions, see “Single-Tasking and Multitasking Execution Modes” and “Handle Rate Transitions”.

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:

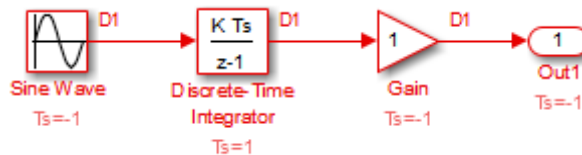
If...	Then...
all of the inputs have the same sample time and the block can accept that sample time	Simulink assigns the sample time to the block
the inputs have different discrete sample times and all of the input sample times are integer multiples of the fastest input sample time	Simulink assigns the 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.
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.

Concepts

- “Backpropagation in Sample Times” on page 5-35

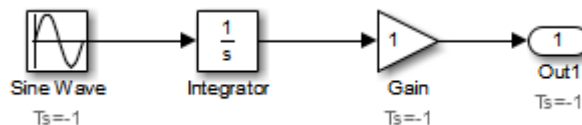
Backpropagation in Sample Times

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 **Sample Time > Colors** from the Simulink **Display** menu 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 selecting **Simulation > Update Diagram** 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.

Referencing a Model

- “Overview of Model Referencing” on page 6-2
- “Create a Model Reference” on page 6-8
- “Convert a Subsystem to a Referenced Model” on page 6-12
- “Referenced Model Simulation Modes” on page 6-21
- “View a Model Reference Hierarchy” on page 6-35
- “Model Reference Simulation Targets” on page 6-37
- “Simulink Model Referencing Requirements” on page 6-47
- “Parameterize Model References” on page 6-54
- “Conditional Referenced Models” on page 6-61
- “Protected Model” on page 6-69
- “Use Protected Model in Simulation” on page 6-71
- “Inherit Sample Times” on page 6-73
- “Refresh Model Blocks” on page 6-77
- “S-Functions with Model Referencing” on page 6-78
- “Buses in Referenced Models” on page 6-81
- “Signal Logging in Referenced Models” on page 6-82
- “Model Referencing Limitations” on page 6-83

Overview of Model Referencing

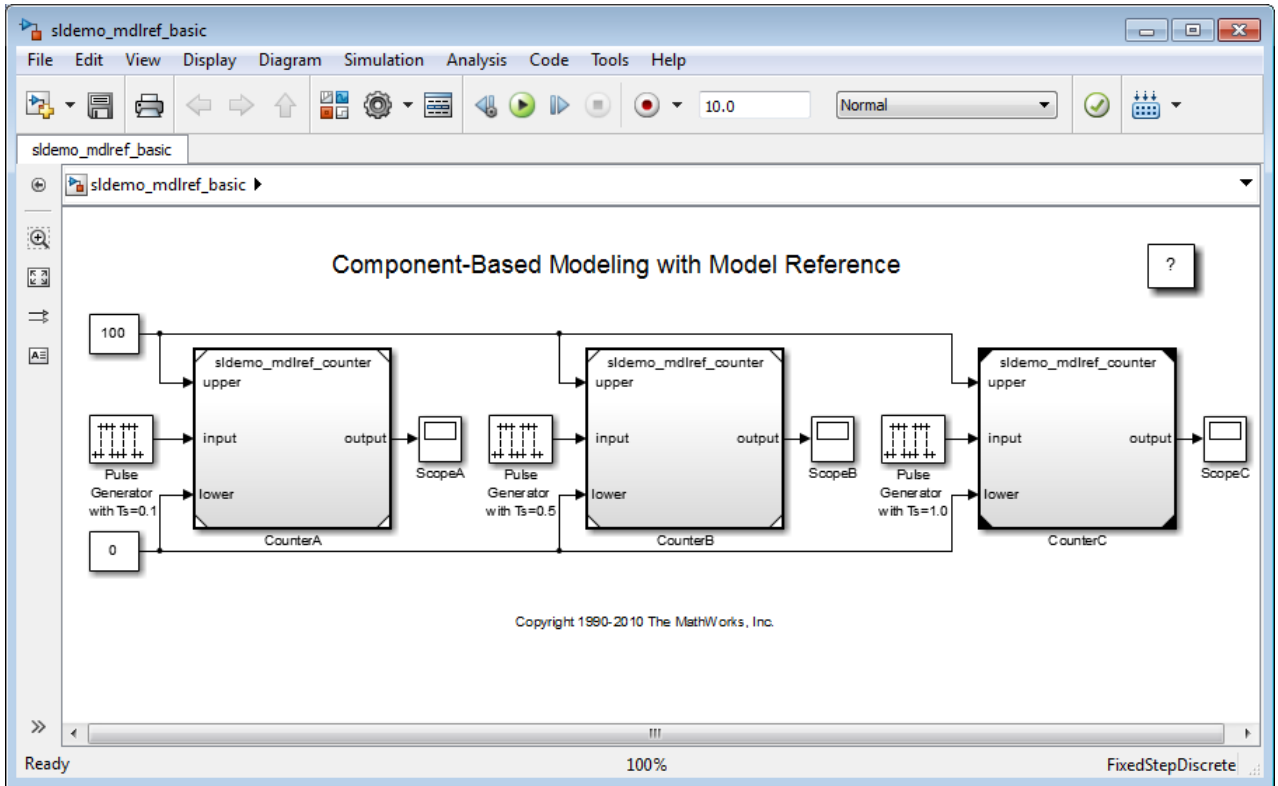
In this section...
“About Model Referencing” on page 6-2
“Referenced Model Advantages” on page 6-5
“Masking Model Blocks” on page 6-6
“Models That Use Model Referencing” on page 6-7
“Model Referencing Resources” on page 6-7

About Model Referencing

You can include one model in another by using Model blocks. Each instance of a Model block represents a reference to another model, called a *referenced model* or *submodel*. For simulation and code generation, the referenced model effectively replaces the Model block that references it. The model that contains a referenced model is its *parent model*. A collection of parent and referenced models constitute a *model reference hierarchy*. A parent model and all models subordinate to it comprise a *branch* of the reference hierarchy.

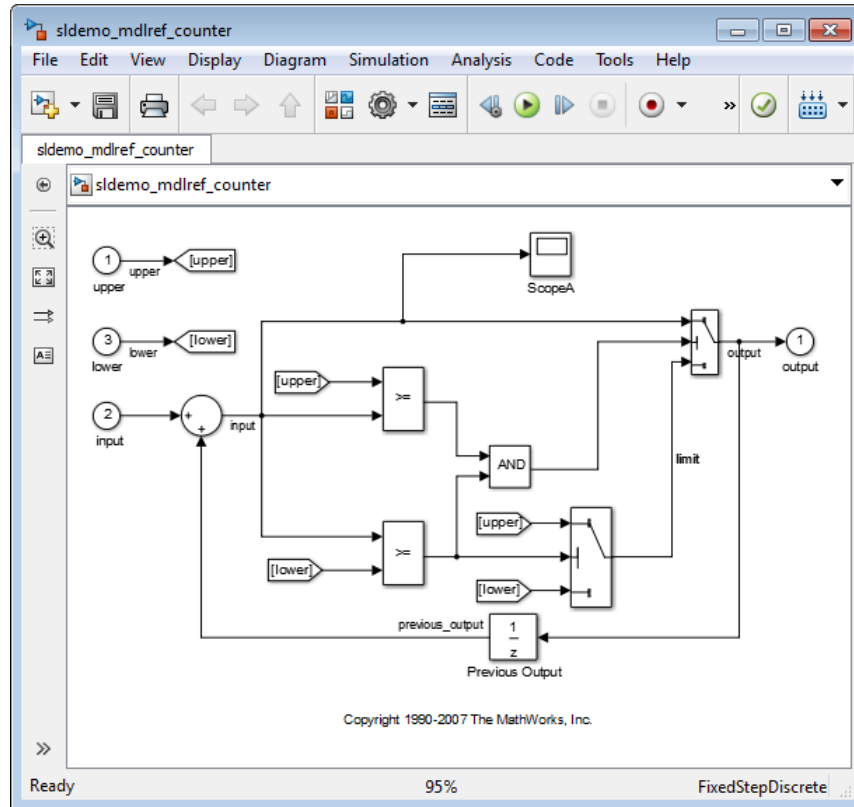
The interface of a referenced model consists of its input and output ports (and control ports, in the case of a conditional referenced model) and its parameter arguments. A Model block displays inputs and outputs corresponding to the root-level inputs and outputs of the model it references. These ports enable you to incorporate the referenced model into the block diagram of the parent model.

For example, the `sldemo_md1ref_basic` model includes Model blocks that reference three instances of the same referenced model, `sldemo_md1ref_counter`.

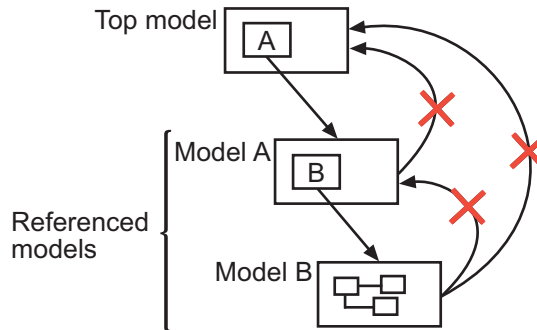


Use the ports on a Model block to connect the submodel to other elements of the parent model. Connecting a signal to a Model block port has the same effect as connecting the signal to the corresponding port in the submodel. For example, the Model block CounterA has three inputs: two Constant blocks and a Pulse Generator block with a sample time of .1. The Model block CounterB also has three inputs: the same two Constant blocks, and a Pulse Generator block with a sample time of .5. Each Model block has an output to a separate Scope block.

The referenced model includes Inport and Outport blocks (and possibly Trigger or Enable blocks) to connect to the parent model. The `sldemo_mdref_counter` model has three Inport blocks (upper, input, and lower) and one Outport block (output).



A referenced model can itself contain Model blocks and thus reference lower-level models, to any depth. The *top model* is the topmost 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 reference hierarchy. This figure shows cyclic inheritance.



A parent model can contain multiple Model blocks that reference the same submodel as long as the submodel does not define global data. The submodel can also appear in other parent models at any level. You can parameterize a referenced model to provide tunability for all instances of the model. Also, you can parameterize a referenced model to let different Model blocks specify different values for variables that define the behavior of the submodel. See “Parameterize Model References” on page 6-54 for details.

By default, the Simulink software executes a top model interpretively, just as it would if the model did not include submodels. Simulink can execute a referenced model interpretively, as if it were an atomic subsystem, or by compiling the submodel to code and executing the code. See “Referenced Model Simulation Modes” on page 6-21 for details.

You can use a referenced model as a standalone model, if it does not depend on data that is available only from a higher-level model. An appropriately configured model can function as both a standalone model and a referenced model, without changing the model or any entities derived from it.

Referenced Model Advantages

Like subsystems, referenced models allow you to organize large models hierarchically; Model blocks can represent major subsystems. Like libraries, referenced models allow you to use the same capability repeatedly without having to redefine it. However, referenced models provide several advantages that are unavailable with subsystems and/or library blocks:

- **Modular development**

You can develop a referenced model independently from the models that use it.

- **Model Protection**

You can obscure the contents of a referenced model, allowing you to distribute it without revealing the intellectual property that it embodies.

- **Inclusion by reference**

You can reference a model multiple times without having to make redundant copies, and multiple models can reference the same model.

- **Incremental loading**

Simulink loads a referenced model at the point it needs to use the model, which speeds up model loading.

- **Accelerated simulation**

Simulink 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 requires code generation 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 additional information about how model referencing compares to other Simulink componentization techniques, see “Componentization Guidelines” on page 12-17. For a video summarizing advantages of model referencing, see [Modular Design Using Model Referencing](#)

Masking Model Blocks

You can use the masking facility to create custom dialog boxes and icons for Model blocks. Masked dialog boxes can make it easier to specify additional parameters for Model blocks. For information about block masks, see “Masking”.

Models That Use Model Referencing

Simulink includes several models that illustrate model referencing.

The Introduction to Managing Data with Model Reference example introduces the basic concepts and workflow related to managing data with model referencing. To explore these topics in more detail, select the Detailed Workflow for Managing Data with Model Reference example.

In addition, the `sldemo_absbrake` model represents a wheel speed calculation as a Model block within the context of an anti-lock braking system (ABS).

Model Referencing Resources

The following are the most commonly needed resources for working with model referencing:

- The Model block, which represents a model that another model references. See the Model block parameters in the “Ports & Subsystems Library Block Parameters” section of the “Block-Specific Parameters” table for information about accessing a Model block programmatically.
- The **Configuration Parameters > Diagnostics > Model Referencing** pane, which controls the diagnosis of problems encountered in model referencing. See “Diagnostics Pane: Model Referencing” for details.
- The **Configuration Parameters > Model Referencing** pane, which provides options that control model referencing and list files on which referenced models depend. See “Model Referencing Pane” for details.

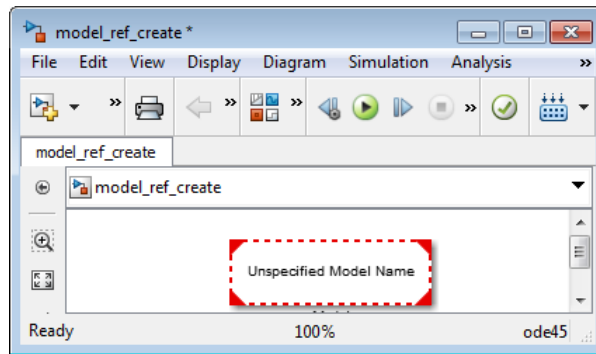
Create a Model Reference

A model becomes a submodel when a Model block in some other model references it. Any model can function as a submodel, and such use does not preclude using it as a separate model also.

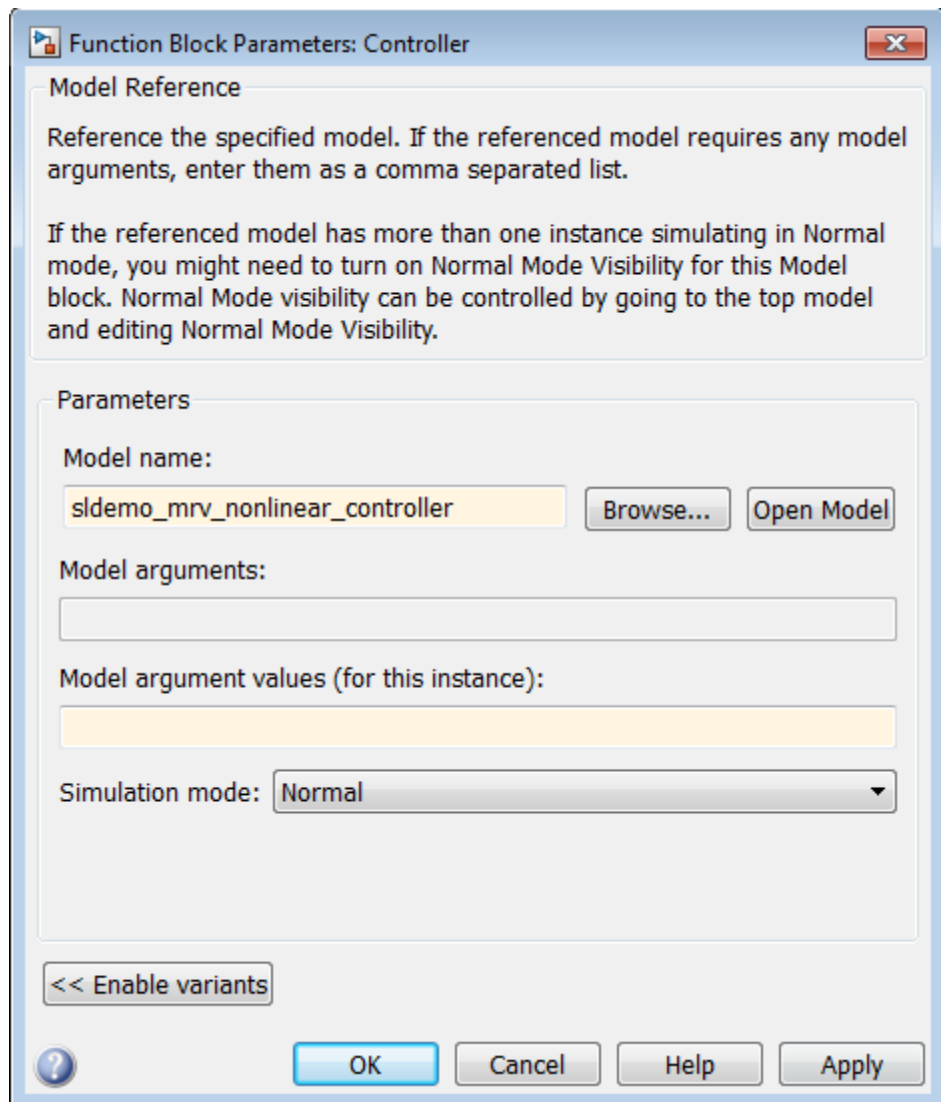
For a video introducing how to create model references, see *Getting Started with Model Referencing*.

To create a reference to a model (submodel) in another model (parent model):

- 1** If the folder containing the submodel you want to reference is not on the MATLAB path, add the folder to the MATLAB path.
- 2** In the submodel:
 - Enable **Configuration Parameters > Optimization > Inline parameters**. You must enable **Inline parameters** for all models in a model reference hierarchy except the top model in the hierarchy. See “Inline Parameter Requirements” on page 6-51 for details.
 - Set **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** to:
 - **One**, if the hierarchy uses the model at most once
 - **Multiple**, to use the model more than once per top model. To reduce overhead, specify **Multiple** only when necessary.
 - **Zero**, which precludes referencing the model
- 3** Create an instance of the Model block in the parent model by dragging a Model block instance from the Ports & Subsystems library to the parent model. The new block is initially unresolved (specifies no submodel) and has the following appearance:



- 4 Open the new Model block's parameter dialog box by double-clicking the Model block. See "Navigating a Model Block" for more about accessing Model block parameters.



- 5 Enter the name of the submodel in the **Model name** field. This name must contain fewer than 60 characters. (See “Name Length Requirement” on page 6-47.)

- For information about **Model Arguments** and **Model argument values**, see “Using Model Arguments” on page 6-55.
- For information about the **Simulation mode**, see “Referenced Model Simulation Modes” on page 6-21.

6 Click **OK** or **Apply**.

If the referenced model contains any root-level inputs or outputs, Simulink displays corresponding input and output ports on the Model block instance that you have created. Use these ports to connect the referenced model to other ports in the parent model.

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. For information about connecting a bus signal to a referenced model, see “Bus Usage Requirements” on page 6-52.

Convert a Subsystem to a Referenced Model

Conversion Process

The process for converting a subsystem to a referenced model involves these tasks:

- 1 “Select Subsystems to Convert” on page 6-12.
- 2 “Prepare the Model for Conversion” on page 6-13.
- 3 “Run a Conversion Tool” on page 6-16.
- 4 “Connect the Model Block and Perform Other Post-Conversion Tasks” on page 6-18.

Select Subsystems to Convert

Model referencing offers several benefits for modeling large, complex systems and for team-based development, as summarized in “Referenced Model Advantages” on page 6-5.

Subsystems or libraries are better suited for some modeling goals than model referencing. Many large models involve using a combination of subsystems and referenced models. For information to help you to decide which subsystems to convert to referenced models, see “Componentization Guidelines” on page 12-17.

The types of subsystems that you can convert to a referenced model are:

- Atomic Subsystem
- “Create a Triggered Subsystem” on page 7-33
- “Create an Enabled Subsystem” on page 7-17
- “Create a Triggered and Enabled Subsystem” on page 7-42
- “Create a Function-Call Subsystem” on page 7-48

Note If you want to create a referenced model that accepts an asynchronous function call, see “Asynchronous Support Limitations”.

If you convert a masked subsystem, you might need to perform some additional tasks to maintain the same general behavior that the masked subsystem provided. For details, see “Convert a Masked Subsystem” on page 6-19.

Prepare the Model for Conversion

Simulink provides a tool that you can use to convert a subsystem to a referenced model.

When the conversion tool identifies a change that you need to make to a model or subsystem to support the conversion, the conversion processing stops. The conversion tool reports conversion issues one at a time. To perform the conversion efficiently, before using the conversion tool, consider modifying your model as described in the following steps.

When you run the conversion tool, it will flag any additional model modifications that you may need to make.

1 Save a backup copy of the model.

You can use the backup copy of the model to help you to configure the model after performing the conversion. For example, if you convert a masked subsystem and want to have a masked Model block, you might want to copy information from the masked subsystem into the masked Model block.

Also, you can revert to the original version of the model if you decide not to convert the subsystem after you have started to modify the model, or if the conversion processing is unsuccessful.

2 Use the Configuration Parameters dialog box to set the following model configuration parameters.

- Select **Optimization > Inline parameters**.

- b** Set **Diagnostics > Data Validity > Signal resolution** to Explicit only.
- c** Set **Diagnostics > Connectivity > Mux blocks used to create bus signals** to Error. For information about proper mux usage, see “Prevent Bus and Mux Mixtures” on page 51-100.

As an alternative for steps b and c, consider using the associated Model Advisor checks.

3 Configure subsystem interfaces to the model.

Subsystem Interface	What to Look For	Model Modification
Goto or From blocks	Crossing of subsystem boundaries.	Add input or output ports to the subsystem. Reconfigure the model as necessary to reflect the added 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 with a <code>Simulink.Signal</code> object. For details, see “Data Stores with Signal Objects” on page 49-12.
Tunable parameters	Global tunable parameters listed in the dialog box that opens when you select the Configuration Parameters > Optimization > Signals and Parameters > Configure button.	Use <code>tunablevars2parameterobjects</code> to create a <code>Simulink.Parameter</code> object for each tunable parameter. The <code>Simulink.Parameter</code> objects must have a non-Auto storage class. For more information, see “Parameterize Model References” on page 6-54 and “Tunable Parameters” on page 3-9.

4 Configure the subsystem and its contents.

Subsystem Configuration	What to Look For	Model Modification
Inactive subsystem variants	Variant Subsystem blocks.	<p>Make active the subsystem variant that you want to convert. The conversion tool does not convert inactive subsystem variants.</p> <p>To have the new Model block behave similar to the subsystem variant, convert each variant subsystem separately.</p> <p>For details, see “Set Up Variant Subsystems” on page 8-14.</p>
Function calls	Function-call signals that cross virtual subsystem boundaries.	Move the Function-Call Generator block inside of the subsystem that you want to convert.
	Function-call outputs.	Change the function-call outputs to data triggers.
	Wide function-call ports.	Eliminate wide signals for function-call subsystems.
Sample times	Sample time of an Inport block that does not match the sample time of the block driving the Inport.	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 input to subsystems.	Consider moving the Constant blocks into the subsystem.

Subsystem Configuration	What to Look For	Model Modification
Buses	Bus signals that enter and exit a subsystem.	Match signal names and bus element names for blocks inside of the subsystem.
	Duplicate signal names in buses.	Make signal names of the bus elements unique.
	Signal names that are not valid MATLAB identifiers. A valid identifier is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the <code>namelengthmax</code> function.	Change any invalid signal names to be valid MATLAB identifiers.

Run a Conversion Tool

Run a conversion tool, using one of these approaches:

- From the context menu of the subsystem, select **Subsystem & Model Reference > Convert Subsystem to > Referenced Model**.
- Use the `Simulink.SubSystem.convertToModelReference` command.

This function provides more capabilities than the **Referenced Model** menu option. For example, with the function, you can replace a subsystem with an equivalent Model block in a single operation.

Simulink saves the contents of the subsystem as a new model, then creates and opens an untitled model that contains a Model block whose referenced model is the new model.

Model Name

Simulink automatically provides a model name that is based on the block name and is unique in the MATLAB path. This name always contains fewer than 60 characters.

Error Handling

During the conversion, if an error occurs, the outcome depends on the kind of error.

- For some errors, a message box appears that gives you the choice of cancelling or continuing.

Before you continue, note any changes that you want to make after the conversion. If you cancel, then the conversion tool exits.

- If continuing is impossible, Simulink cancels the conversion without offering a choice to continue. If the new Model block has *not* been created, then the model is not affected by the conversion tool processing.

What Gets Copied by the Conversion?

The conversion tool copies the following elements from the original model to the newly created referenced model:

- **Configuration sets** – If the referencing model uses a configuration set that is not a referenced configuration set, then the conversion tool copies that entire configuration set to the referenced model.

If the referencing model uses a referenced configuration set, then both the referencing and referenced models use the same referenced configuration set.

- **Variables** – All of the model workspace variables of the original model are copied into the model workspace of the *referenced* model.
- **Requirements links** – Requirements links created with the Simulink Verification and Validation software (for example, requirements links to blocks and signals) are copied. Any requirements links on the subsystem that you convert are transferred to the new Model block.

Connect the Model Block and Perform Other Post-Conversion Tasks

After you successfully complete the conversion:

- 1** Connect the referenced model.
 - a** Delete the subsystem block from the source model.
 - b** Copy the new Model block to the location of the deleted subsystem block.

Simulink automatically reconnects all signals. The source model is a parent model that contains the referenced model.

- 2** For root Inport blocks in the new referenced model, check the **Interpolate data** parameter to make sure it is appropriate. For details, see the Inport block documentation.
- 3** (Optional) Delete unused duplicate variables.

The conversion tool copies into the model workspace of the *referenced* model (for example, `ModelB`) all of the model workspace variables of the *referencing* model (`ModelA`). This step results in duplicate variables in the two model workspaces. To improve the reliability of your model in case the model changes in the future, consider removing unused variables or moving variables that are used by both `ModelA` and `ModelB`:

- For a model workspace variable used only by the referencing model (`ModelA`), delete the variable from the model workspace of the referenced model (`ModelB`).
 - For a model workspace variable used only by the referenced model (`ModelB`), delete the variable from the model workspace of the referencing model (`ModelA`).
 - For model workspace variables used by both the referencing and referenced model, consider moving that variable to the base workspace.
- 4** (Optional) To achieve similar results in the new referenced model as you did by using the Variant Subsystem block, convert each of the variant subsystems to referenced models, and then use a Model Variants block. For more information, see “Set Up Model Variants” on page 8-8.

If you convert a masked subsystem, you may need to perform some additional tasks to maintain the same general behavior that the masked subsystem provided. For details, see “Convert a Masked Subsystem” on page 6-19.

Convert a Masked Subsystem

Perform the tasks described in:

- “Select Subsystems to Convert” on page 6-12.
- “Prepare the Model for Conversion” on page 6-13.
- “Run a Conversion Tool” on page 6-16.
- “Connect the Model Block and Perform Other Post-Conversion Tasks” on page 6-18.

If the subsystem that you converted contains a mask, then you might want to mask the Model block in your new referenced model (see “Masking”). Configure the referenced model to support the functionality that was in the masked subsystem.

Note The referenced model does not support the functionality that you can achieve with mask initialization code to create masked parameters.

Masked Subsystem Functionality	Referenced Model Configuration
Masked parameters	<ol style="list-style-type: none"> 1 In the model workspace of the referenced model, create a variable for each masked parameter. 2 In the Model Explorer, select the Model Workspace node. In the Dialog pane, in the Model arguments parameter, enter the variables that you want to be model arguments.

Masked Subsystem Functionality	Referenced Model Configuration
	3 In the new Model block, for the Model arguments parameter, specify the values for the model arguments.
Masked callbacks, icons, ports, and documentation	In the backup copy of the masked subsystem, open the Mask Editor and select the content to copy into to masked Model block. In the Mask Editor for the new Model block, copy the appropriate content from the masked subsystem that you converted.

Referenced Model Simulation Modes

In this section...

“Simulation Modes for Referenced Models” on page 6-21

“Specify the Simulation Mode” on page 6-23

“Mixing Simulation Modes” on page 6-23

“Using Normal Mode for Multiple Instances of Referenced Models” on page 6-25

“Accelerating a Freestanding or Top Model” on page 6-33

Simulation Modes for Referenced Models

Simulink executes the top model in a model reference hierarchy just as it would if no referenced models existed. All Simulink simulation modes are available to the top model. Simulink can execute a referenced model in any of four modes: Normal, Accelerator, Software-in-the-loop (SIL), or Processor-in-the-loop (PIL).

Normal Mode

Simulink executes a Normal mode submodel interpretively. Normal mode, compared to other simulation modes:

- Requires no delay for code generation or compilation
- Works with more Simulink and Stateflow tools, supporting tools such as:
 - Scopes, port value display, and other output viewing tools
 - Scopes work with Accelerator mode referenced models, but require using the Signal & Scope Manager and adding test points to signals. Adding or removing a test point necessitates rebuilding the SIM target for a model, which can be time-consuming.
 - Model coverage analysis
 - Stateflow debugging and animation
- Provides more accurate linearization analysis
- Supports more S-functions than Accelerator mode does

Normal mode executes slower than Accelerator mode does.

Simulation results for a given model are nearly the same in either Normal or Accelerator mode. Trivial differences can occur due to differences in the optimizations and libraries that you use.

You can use Normal mode with multiple instances of a referenced model. For details, see “Using Normal Mode for Multiple Instances of Referenced Models” on page 6-25.

Accelerator Mode

Simulink executes an Accelerator mode submodel by creating a MEX-file (or *simulation target*) for the submodel, then running the MEX-file. See “Model Reference Simulation Targets” on page 6-37 for more information. Accelerator mode:

- Takes time for code generation and code compilation
- Does not fully support some Simulink tools, such as Model Coverage and the Simulink Debugger.
- Executes faster than Normal mode

Simulation results for a given model are nearly identical in either Normal or Accelerator mode. Trivial differences can occur due to differences in the optimizations and libraries that you use.

Software-in-the-Loop (SIL) Mode

Simulink executes a SIL-mode referenced model by generating production code using the model reference target for the submodel. This code is compiled for, and executed on, the host platform.

With SIL mode, you can:

- Verify generated source code without modifying the original model
- Reuse test harnesses for the original model with the generated source code

SIL mode provides a convenient alternative to PIL simulation when the target hardware is not available.

This option requires Embedded Coder software.

For more information, see “Numerical Equivalence Testing” in the Embedded Coder documentation.

Processor-in-the-Loop (PIL) Mode

Simulink executes a PIL-mode referenced model by generating production code using the model reference target for the submodel. This code is cross-compiled for, and executed on, a target processor or an equivalent instruction set simulator.

With PIL mode, you can:

- Verify deployment object code on target processors without modifying the original model
- Reuse test harnesses for the original model with the generated source code

This option requires Embedded Coder software.

For more information, see “Numerical Equivalence Testing” in the Embedded Coder documentation.

Specify the Simulation Mode

The Model block for each instance of a referenced model controls its simulation mode. To set or change the simulation mode for a submodel:

- 1** Access the block parameter dialog box for the Model block. (See “Navigating a Model Block”.)
- 2** Set the **Simulation mode** parameter.
- 3** Click **OK** or **Apply**.

Mixing Simulation Modes

The following table summarizes the relationship between the simulation mode of the parent model and its submodels.

Parent Model Simulation Mode	Submodel Simulation Modes
Normal	<ul style="list-style-type: none"> • Submodels can use Normal, Accelerator, SIL, or PIL mode. • A submodel can execute in Normal mode <i>only</i> if every model that is superior to it in the hierarchy also executes in Normal mode. A Normal mode path then extends from the top model through the model reference hierarchy down to the Normal mode submodel.
Accelerator	<ul style="list-style-type: none"> • All subordinate models must also execute in Accelerator mode. • When a Normal mode model is subordinate to an Accelerator mode model, Simulink posts a warning and temporarily overrides the Normal mode specification. • When a SIL-mode or PIL-mode model is subordinate to an Accelerator mode model, an error occurs.
SIL	<ul style="list-style-type: none"> • All subordinate models also execute in SIL mode, provided their simulation modes are Normal, Accelerator, or SIL. Otherwise, an error occurs. See “Simulation Mode Override Behavior in Model Reference Hierarchy”. • The SIL mode Model block uses the model reference targets of the blocks beneath. • Multiple Model blocks, starting at the top of a model reference hierarchy, can execute at a time in SIL mode. However, during code coverage or code execution profile, only one Model block can execute at a time in SIL mode.
PIL	<ul style="list-style-type: none"> • All subordinate models also execute in PIL mode, provided their simulation modes are Normal, Accelerator, or PIL. Otherwise, an

Parent Model Simulation Mode	Submodel Simulation Modes
	<p>error occurs. See “Simulation Mode Override Behavior in Model Reference Hierarchy”.</p> <ul style="list-style-type: none"> • The PIL mode Model block uses the model reference targets of the blocks beneath. • Only one Model block, starting at the top of a model reference hierarchy, can execute at a time in PIL mode.

For more information about SIL and PIL modes, see in the Embedded Coder documentation:

- “Code Interfaces for SIL and PIL”
- “SIL and PIL Simulation Support and Limitations”

Using Normal Mode for Multiple Instances of Referenced Models

You can simulate a model that has multiple references in Normal mode.

Normal Mode Visibility

All instances of a Normal mode referenced model are part of the simulation. However, Simulink displays only one instance in a model window; that instance is determined by the Normal Mode Visibility setting. Normal mode visibility includes the display of Scope blocks and data port values.

If you do not set Normal Mode Visibility, Simulink picks one instance of each Normal mode model to display.

After a simulation, if you try to open a referenced model from a Model block that has Normal Mode Visibility set to off, Simulink displays a warning.

For a description of how 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 That Has Normal Mode Visibility” on page 6-28.

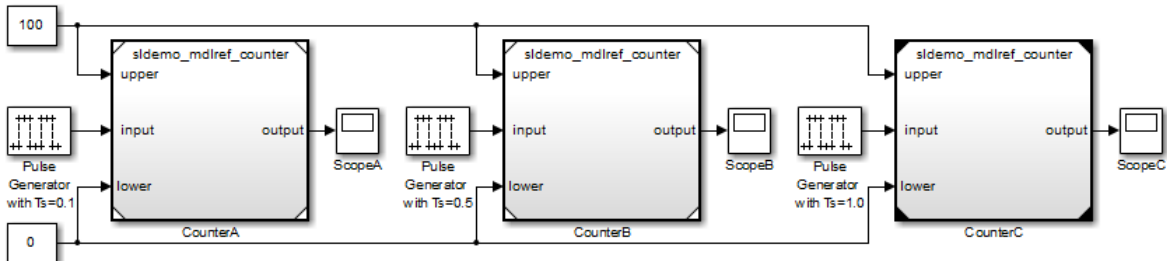
Note If you change the Normal Mode Visibility setting for a referenced model, you must simulate the top model in the model reference hierarchy to make use of the new setting.

Examples of a Model with Multiple Referenced Instances in Normal Mode

sldemo_mdref_basic. The sldemo_mdref_basic model has three Model blocks (CounterA, CounterB, and CounterC) that each reference the sldemo_mdref_counter model.

If you update the diagram, the sldemo_mdref_basic displays different icons for each of the three Model blocks that reference sldemo_mdref_counter.

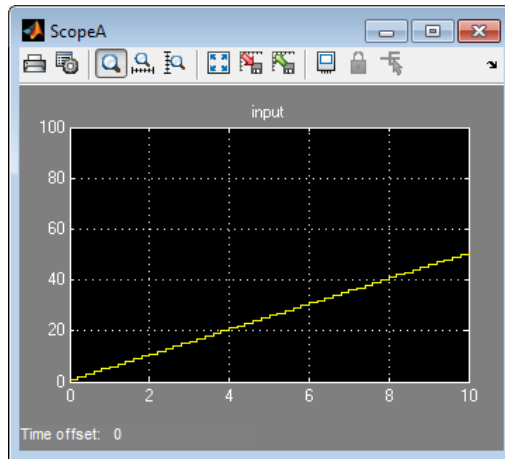
Component-Based Modeling with Model Reference



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)

If you do the following steps, then the ScopeA block appears as shown below:

- 1 Simulate `sldemo_md1ref_basic`.
- 2 Open the `sldemo_md1ref_counter` model.
- 3 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 `md1ref_counter` model from the CounterB Model block (for example, by double-clicking the Model block), ScopeA in `md1ref_counter` still shows the results of the CounterA Model block, because that is the Model block with Normal Mode Visibility set to on.

`sldemo_md1ref_depgraph`. The `sldemo_md1ref_depgraph` model shows the use of the Model Dependency Viewer for a model that has multiple Normal mode instances of a referenced model. The model shows what you need to do to set up a model with multiple referenced instances in Normal mode.

Set Up a Model with Multiple Instances of a Referenced Model in Normal Mode

This section describes how to set up a model to support the use of multiple instances of Normal mode referenced models.

- 1** Set the **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** parameter to **Multiple**.

If you cannot use the **Multiple** setting for your model, because of the requirements described in the “Total number of instances allowed per top model” parameter documentation, then you can have only one instance of that referenced model be in Normal mode.

- 2** For each instance of the referenced model that you want to be in Normal mode, in the block parameters dialog box for the Model block, 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), or 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 have performed the steps in this section, you can specify a non-default instance to have Normal mode visibility. For details, see “Specify the Instance That Has Normal Mode Visibility” on page 6-28.

Specify the Instance That Has Normal Mode Visibility

This section describes how to specify Normal Mode Visibility for an instance other than the one that an instance that Simulink selects automatically.

You need to:

- 1** (Optionally) “Determine Which Instance Has Normal Mode Visibility” on page 6-29.

2 “Set Normal Mode Visibility” on page 6-30.

3 Simulate the top model to apply the new Normal Mode Visibility settings.

Determine Which Instance Has Normal Mode Visibility. If you do not already know which instance currently has Normal mode visibility, you can determine that by using one of these approaches:

- If you update the diagram and have made no other changes to the model, then you can navigate through the model hierarchy to 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.
- When you are editing a model or during compilation, use the `ModelReferenceNormalModeVisibilityBlockPath` parameter.

If you use this parameter while editing a model, you must update the diagram before you use this 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_mdref_basic',...
  'ModelReferenceNormalModeVisibilityBlockPath')
```

```
ans =
```

```
Simulink.BlockPath
Package: Simulink
```

```
Block Path:
  'sldemo_mdref_basic/CounterA'
```

- For a top model that is being simulated or that is in a compiled state, you can use the `CompiledModelBlockInstancesBlockPath` parameter. For example:

```
a = get_param('sldemo_mdref_depgraph',...
  'CompiledModelBlockInstancesBlockPath')
```

```
a =
```

```
sldemo_mdhref_F2C: [1x1 Simulink.BlockPath]
sldemo_mdhref_heater: [1x1 Simulink.BlockPath]
sldemo_mdhref_outdoor_temp: [1x1 Simulink.BlockPath]
```

Set Normal Mode Visibility. To enable Normal Mode Visibility for a different instance of the referenced model than the instance that currently has Normal Mode Visibility, use *one* of these approaches:

- Navigate to the top model and select the **Diagram > Subsystem & Model Reference > Model Block Normal Mode Visibility** menu item.

The Model Block Normal Mode Visibility dialog box appears. That dialog box includes instructions in the right pane. For additional details about the dialog box, see “Model Block Normal Mode Visibility Dialog Box” on page 6-31.

- From the MATLAB command line, set the `ModelReferenceNormalModeVisibility` parameter.

For input, you can specify:

- 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 strings, with the strings being paths to individual blocks and models. The following example has the same effect as the preceding example (which shows how to specify an array of `Simulink.BlockPath` objects):

```
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.

Note You cannot change Normal Mode Visibility during a simulation.

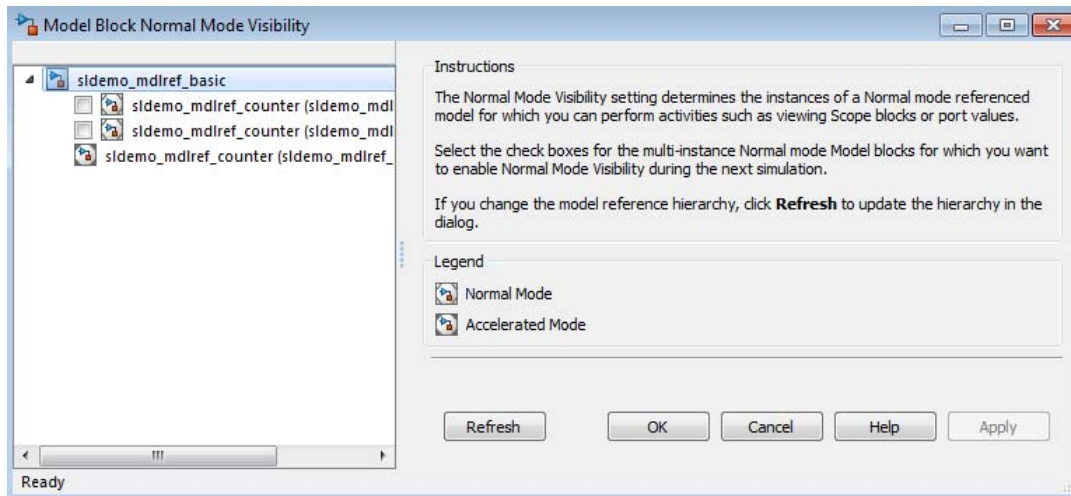
Model Block Normal Mode Visibility Dialog Box

If you have a model that has multiple instances of a referenced model in Normal mode, you can use the Block Model Normal Mode Visibility dialog box to set Normal Mode Visibility for a specific instance. For a description of Normal mode visibility, see “Normal Mode Visibility” on page 6-25.

Alternatively, you can set the `ModelReferenceNormalModeVisibility` parameter. For information about how to specify an instance of a referenced model that is in Normal mode that is different than the instance automatically selected by Simulink, see “Specify the Instance That Has Normal Mode Visibility” on page 6-28.

Open the Model Block Normal Mode Visibility Dialog Box. To open the Model Block Normal Mode Visibility dialog box, navigate to the top model and select **Diagram > Subsystem & Model Reference > Model Block Normal Mode Visibility**.

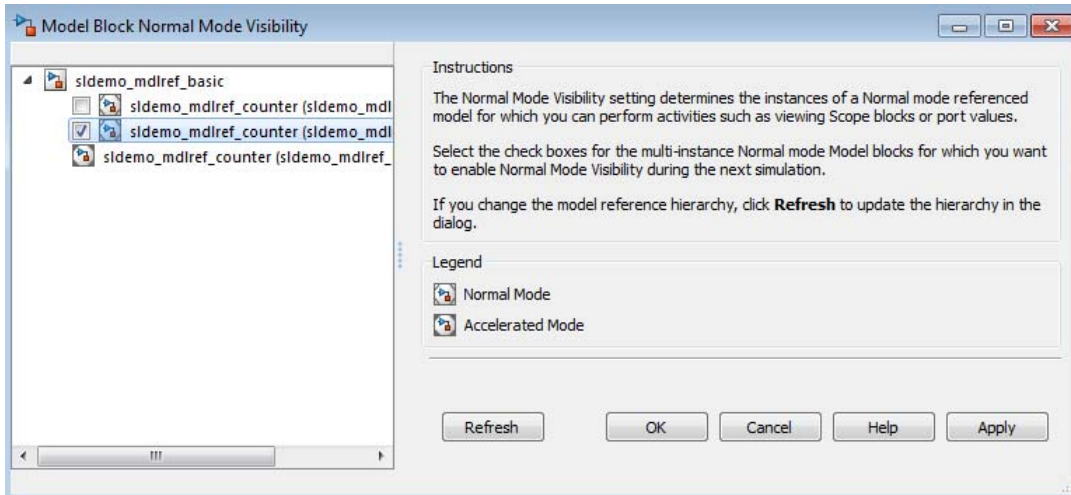
The dialog box for the `sldemo_md1ref_basic` model, with the hierarchy pane expanded, looks like this:



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.

Select a Model for Normal Mode Visibility.

The dialog box shows the complete model block hierarchy for the top model. The Normal mode instances of referenced models have check boxes. Select the check box for the instance of each model that you want to have Normal mode visibility.



When you select a model, Simulink:

- Selects all ancestors of that model
- Deselects all other instances of that model

When a model is deselected, Simulink deselects all children of that model.

Opening a Model from the Model Block Normal Mode Visibility Dialog Box. You can open a model from the Model Block Normal Mode Visibility dialog box by right-clicking the model in the model hierarchy pane and clicking **Open**.

Refreshing the Model Reference Hierarchy. To ensure the model hierarchy pane of the Model Block Normal Mode Visibility dialog box reflects the current model hierarchy, click **Refresh**.

Accelerating a Freestanding or Top Model

You can use Simulink Accelerator mode or Rapid Accelerator mode to achieve faster execution of any Simulink model, including a top model in a model reference hierarchy. For details about Accelerator mode, see the “Acceleration” documentation. For information about Rapid Accelerator mode, see “Rapid Simulations”.

When you execute a top model in Simulink Accelerator mode or Rapid Accelerator mode, all submodels execute in Accelerator mode. For any submodel that specifies Normal mode, Simulink displays a warning message.

Be careful not to confuse Accelerator mode execution of a referenced model with:

- Accelerator mode execution of a freestanding or top model, as described in “Acceleration”
- Rapid Accelerator mode execution of a freestanding or top model, as described in “Rapid Simulations”

While the different types of acceleration share many capabilities and techniques, they have different implementations, requirements, and limitations.

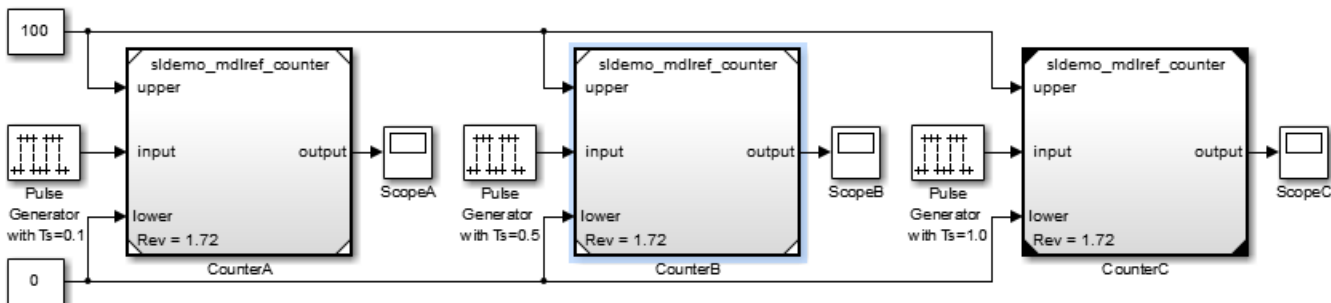
View a Model Reference Hierarchy

Simulink provides tools and functions that you can use to examine a model reference hierarchy:

- Content preview displays a representation of the contents of a referenced model, without opening the Model block. Content preview helps the user of a model understand at a glance the kind of processing performed by the referenced model. For details, see “Enable for Model Blocks” on page 1-30 in “Preview Content of Hierarchical Items.”
- **Model Dependency Viewer** — Shows the structure the hierarchy lets you open constituent models. The Referenced Model Instances view displays Model blocks differently to indicate Normal, Accelerator, and PIL modes. See “Model Dependency Viewer” on page 9-83 for more information.
- **view_mdrefs** function — Invokes the Model Dependency Viewer to display a graph of model reference dependencies.
- **find_mdrefs** function — Finds all models directly or indirectly referenced by a given model.

Display Version Numbers

To display the version numbers of the models referenced by a model, for the parent model, choose **Display > Blocks > Block Version for Referenced Models**. Simulink displays the version numbers in the icons of the corresponding Model block instances.



The version number displayed on a Model block icon refers to the version of the model used to either:

- Create the block
- Refresh the block most recently changed

See “Manage Model Versions” on page 4-115 and “Refresh Model Blocks” on page 6-77 for more information.

Model Reference Simulation Targets

In this section...

“Simulation Targets” on page 6-37

“Build Simulation Targets” on page 6-38

“Simulation Target Output File Control” on page 6-39

“Reduce Update Time for Referenced Models” on page 6-42

Simulation Targets

A *simulation target*, or *SIM target*, is a 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 Accelerator mode instances of a given referenced model anywhere in a reference hierarchy.

If you have a Simulink Coder license, be careful not to confuse the simulation target of a submodel with any of these other types of target:

- 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

Simulink creates a simulation target only for a submodel that has one or more Accelerator mode instances in a reference hierarchy. A submodel that executes only in Normal mode always executes interpretively and does not use a simulation target. When one or more instance of a submodel executes in Normal mode, and one or more instance executes in Accelerator mode:

- Simulink creates a simulation target for the Accelerator mode instances.

- The Normal mode instances do not use that simulation target.

Because Accelerator mode requires code generation, it imposes some requirements and limitations that do not apply to Normal mode. Aside from these constraints, you can generally ignore simulation targets and their details when you execute a referenced model in Accelerator mode. See “Limitations on Accelerator Mode Referenced Models” on page 6-87 for details.

Build Simulation Targets

Simulink by default generates the needed target from the referenced model:

- If a simulation target does not exist at the beginning of a simulation
- When you perform an update diagram for a parent model

If the simulation target already exists, then by default Simulink checks whether the submodel has structural changes since the target was last generated. If so, Simulink regenerates the target to reflect changes in the model. For details about how Simulink detects whether to rebuild a model reference target, see the “Rebuild” parameter documentation.

You can change this default behavior to modify the rebuild criteria or specify that Simulink always or never rebuilds targets. See “Rebuild” for details.

To generate simulation targets interactively for Accelerator mode referenced models, do one of these steps:

- Update the diagram on a model that directly or indirectly references the model that is in Accelerator mode
- Execute the `slbuild` command with appropriate arguments at the MATLAB command line

While generating a simulation target, Simulink displays status messages at the MATLAB command line to enable you to monitor the target generation process. Target generation entails generating and compiling code and linking the compiled target code with compiled code from standard code libraries to create an executable file.

Reduce Change Checking Time

You can reduce the time that Simulink spends checking whether any or all simulation targets require rebuilding by setting configuration parameter values as follows:

- In all referenced models throughout the hierarchy, set **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** to **Explicit only**. (See “Signal resolution”.)
- To minimize change detection time, consider setting **Configuration Parameters > Model Referencing > Rebuild options** to **If any changes in known dependencies detected on the top model**. See “Rebuild”.

These parameter values exist in the configuration set of a referenced model, not in the individual Model block, so setting either value for any instance of a referenced model sets it for all instances of that model.

Simulation Target Output File Control

Simulink creates simulation targets in the `s1prj` subfolder of the working folder. If `s1prj` does not exist, Simulink creates it.

Note Simulink Coder code generation also uses the `s1prj` folder. Subdirectories in `s1prj` provide separate places for simulation code, Simulink Coder code, and other files. For details, see “Control the Location for Generated Files”.

By default, the files generated by Simulink diagram updates and model builds are placed in a build folder, the root of which is the current working folder (`pwd`). However, in some situations, you might want the generated files to go to a root folder outside the current working folder. For example:

- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to reuse or share previously-built simulation targets without having to set the current working folder back to a previous working folder.

You might also want to separate generated simulation artifacts from generated production code.

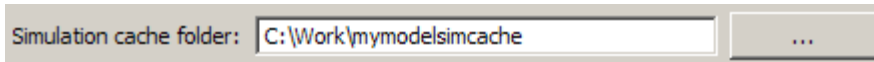
To allow you to control the output locations for the files generated by diagram updates and model builds, the software allows you to separately specify the *simulation cache folder* build folder. The simulation cache folder is the root folder in which to place artifacts used for simulation.

To specify the simulation cache folder, use *one* of these approaches:

- Use the CacheFolder MATLAB session parameter.
- Open the Simulink Preferences dialog box (**File > Simulink Preferences**) and specify a location on your file system for the **Simulation cache folder**, which, if specified, provides the initial defaults for the MATLAB session parameters.

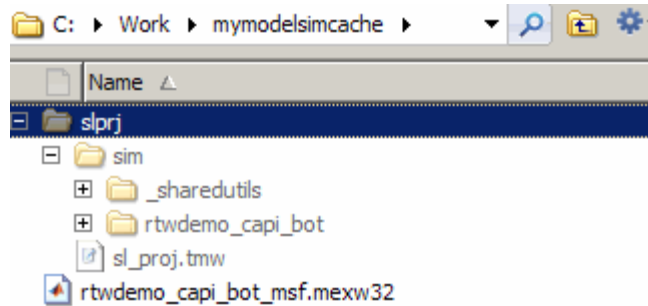
Control Output Location for Model Build Artifacts Used for Simulation

The Simulink preference **Simulation cache folder** provides control over the output location for files generated by Simulink diagram updates. The preference appears in the Simulink Preferences Window, Main Pane, in the **File generation control** group. To specify the root folder location for files generated by Simulink diagram updates, set the preference value by entering or browsing to a folder path, for example:



The folder path that you specify provides the initial default for the MATLAB session parameter CacheFolder. When you initiate a Simulink diagram update, any files generated are placed in a build folder at the root location specified by CacheFolder (if any), rather than in the current working folder (pwd).

For example, using a 32-bit Windows host platform, if you set the **Simulation cache folder** to 'C:\Work\mymodelsimcache' and then simulate the model `rtwdemo_capi`, files are generated into the specified folder as follows:



As an alternative to using the Simulink preferences GUI to set **Simulation cache folder**, you also can get and set the preference value from the command line using `get_param` and `set_param`. For example,

```
>> get_param(0, 'CacheFolder')

ans =

    ''

>> set_param(0, 'CacheFolder', fullfile('C:', 'Work', 'mymodelsimcache'))
>> get_param(0, 'CacheFolder')

ans =

C:\Work\mymodelsimcache
```

Also, you can choose to override the **Simulation cache folder** preference value for the current MATLAB session.

Override Build Folder Settings for the Current MATLAB Session

The Simulink preferences **Simulation cache folder** and **Code generation folder** provide the initial defaults for the MATLAB session parameters `CacheFolder` and `CodeGenFolder`, which determine where files generated by Simulink diagram updates and model builds are placed. However, you can override these build folder settings during the current MATLAB session, using the `Simulink.fileGenControl` function. This function allows you to directly manipulate the MATLAB session parameters, for

example, overriding or restoring the initial default values. The values you set using `Simulink.fileGenControl` expire at the end of the current MATLAB session. For more information and detailed examples, see the `Simulink.fileGenControl` function reference page.

Reduce Update Time for Referenced Models

- “Parallel Building for Large Model Reference Hierarchies” on page 6-42
- “Parallel Building Configuration Requirements” on page 6-43
- “Update Models in a Parallel Computing Environment” on page 6-43
- “Locate Parallel Build Logs” on page 6-45

Parallel Building for Large Model Reference Hierarchies

In a parallel computing environment, you can increase the speed of diagram updates for models containing large model reference hierarchies by building referenced models that are configured in Accelerator mode in parallel whenever conditions allow. For example, if you have Parallel Computing Toolbox™ software, updating of each referenced model can be distributed across the cores of a multicore host computer. If you additionally have MATLAB Distributed Computing Server™ (MDCS) software, updating of each referenced model can be distributed across remote workers in your MATLAB Distributed Computing Server configuration.

The performance gain realized by using parallel builds for updating referenced models depends on several factors, including how many models can be built in parallel for a given model referencing hierarchy, the size of the referenced models, and parallel computing resources such as number of local and/or remote workers available and the hardware attributes of the local and remote machines (amount of RAM, number of cores, and so on).

For configuration requirements that might apply to your parallel computing environment, see “Parallel Building Configuration Requirements” on page 6-43.

For a description of the general workflow for building referenced models in parallel whenever conditions allow, see “Update Models in a Parallel Computing Environment” on page 6-43.

Parallel Building Configuration Requirements

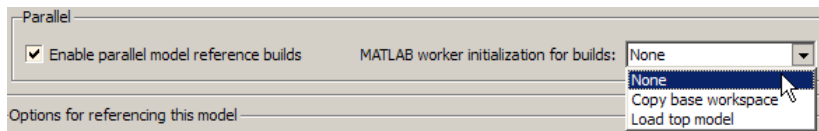
The following requirements apply to using parallel builds for updating model reference hierarchies:

- For local pools, the host machine should have an appropriate amount of RAM available for supporting the number of local workers (MATLAB sessions) that you plan to use. For example, using `parpool(4)` to create a parallel pool with four workers results in five MATLAB sessions on your machine, each using approximately 120 MB of memory at startup.
- Remote MDCS workers participating in a parallel build must use a common platform and compiler.
- A consistent MATLAB environment must be set up in each MATLAB worker session as in the MATLAB client session — for example, shared base workspace variables, MATLAB path settings, and so forth. One approach is to use the `PreLoadFcn` callback of the top model. If you configure your model to load the top model with each MATLAB worker session, its preload function can be used for any MATLAB worker session setup.

Update Models in a Parallel Computing Environment

To take advantage of parallel building for a model reference hierarchy:

- 1** Set up a pool of local and/or remote MATLAB workers in your parallel computing environment.
 - a** Make sure that Parallel Computing Toolbox software is licensed and installed.
 - b** To use remote workers, make sure that MATLAB Distributed Computing Server software is licensed and installed.
 - c** Issue MATLAB commands to set up the worker pool, for example, `parpool(4)`.
- 2** In the Configuration Parameters dialog box, go to the **Model Referencing** pane and select the **Enable parallel model reference builds** option. This selection enables the parameter **MATLAB worker initialization for builds**.



For **MATLAB worker initialization for builds**, select one of the following values:

- **None** if the software should perform no special worker initialization. Specify this value if the child models in the model reference hierarchy do not rely on anything in the base workspace beyond what they explicitly set up (for example, with a model load function).
- **Copy base workspace** if the software should attempt to copy the base workspace to each worker. Specify this value if you use a setup script to prepare the base workspace for multiple models to use.
- **Load top model** if the software should load the top model on each worker. Specify this value if the top model in the model reference hierarchy handles all of the base workspace setup (for example, with a model load function).

- 3 Optionally, turn on verbose messages for simulation builds. If you select verbose builds, the build messages report the progress of each parallel build with the name of the model.

To turn on verbose messages for simulation target builds, go to the **Optimization** pane of the Configuration Parameters dialog box and select **Verbose accelerator builds**.

The **Verbose accelerator builds** option controls the verbosity of build messages both in the MATLAB Command Window and in parallel build log files.

- 4 Optionally, inspect the model reference hierarchy to determine, based on model dependencies, which models will be built in parallel. For example, you can use the Model Dependency Viewer from the Simulink **Analysis > Model Dependencies** menu.
- 5 Update your model. Messages in the MATLAB command window record when each parallel or serial build starts and finishes.

If you need more information about a parallel build, for example, if a build fails, see “Locate Parallel Build Logs” on page 6-45.

Locate Parallel Build Logs

When you update a model for which referenced models are built in parallel, if verbose builds are turned on, messages in the MATLAB Command Window record when each parallel or serial build starts and finishes. For example,

```
### Initializing parallel workers for parallel model reference build.
### Parallel worker initialization complete.
### Starting parallel model reference SIM build for 'bot_model001'
### Starting parallel model reference SIM build for 'bot_model002'
### Starting parallel model reference SIM build for 'bot_model003'
### Starting parallel model reference SIM build for 'bot_model004'
### Finished parallel model reference SIM build for 'bot_model001'
### Finished parallel model reference SIM build for 'bot_model002'
### Finished parallel model reference SIM build for 'bot_model003'
### Finished parallel model reference SIM build for 'bot_model004'
```

To obtain more detailed information about a parallel build, you can examine the parallel build log. For each referenced model built in parallel, the build process generates a file named *model_buildlog.txt*, where *model* is the name of the referenced model. This file contains the full build log for that model.

If a parallel build completes, you can find the build log file in the build subfolder corresponding to the referenced model. For example, for a build of referenced model *bot_model004*, look for the build log file *bot_model004_buildlog.txt* in the referenced model subfolder *build_folder/slprj/sim/bot_model004*.

If a parallel builds fails, you might see output similar to the following:

```
### Initializing parallel workers for parallel model reference build.
### Parallel worker initialization complete.
### Starting parallel model reference SIM build for 'bot_model002'
### Starting parallel model reference SIM build for 'bot_model003'
### Finished parallel model reference SIM build for 'bot_model002'
### Finished parallel model reference SIM build for 'bot_model003'
### Starting parallel model reference SIM build for 'bot_model001'
```

```
### Starting parallel model reference SIM build for 'bot_model1004'  
### Finished parallel model reference SIM build for 'bot_model1004'  
### The following error occurred during the parallel model reference SIM build for  
'bot_model1001':
```

```
Error(s) encountered while building model "bot_model1001"
```

```
### Cleaning up parallel workers.
```

If a parallel build fails, you can find the build log file in a referenced model subfolder under the build subfolder `/par_md1_ref/model`. For example, for a failed parallel build of model `bot_model1001`, look for the build log file `bot_model1001_buildlog.txt` in the subfolder `build_folder/par_md1_ref/bot_model1001/slprj/sim/bot_model1001`.

Simulink Model Referencing Requirements

In this section...
“About Model Referencing Requirements” on page 6-47
“Name Length Requirement” on page 6-47
“Configuration Parameter Requirements” on page 6-47
“Model Structure Requirements” on page 6-52

About Model Referencing Requirements

A model reference hierarchy must satisfy various Simulink requirements, as described in this section. Some limitations also apply, as described in “Model Referencing Limitations” on page 6-83.

Name Length Requirement

The name of a referenced model must contain fewer than 60 characters, exclusive of the .slx or .mdl suffix. An error occurs if the name of a referenced model is too long.

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way that any other model does, as described in “Manage a Configuration Set” on page 10-12. By default, every model in a hierarchy has its own configuration set. Each model uses its configuration set the same way that it would if the model executed independently.

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 referencing. The Simulink response to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, or a trivial resolution exists that carries no risk, 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 “Model configuration mismatch” for details.
- Where no acceptable resolution is possible, Simulink generates an error. Change some or all parameter values to eliminate the problem.

Manually eliminating all configuration parameter incompatibilities can be tedious when:

- A model reference hierarchy contains many submodels that have incompatible parameter values
- A changed parameter value must propagate to many submodels

You can control or eliminate such overhead by using configuration references to assign an externally stored configuration set to multiple models. See “Manage a Configuration Reference” on page 10-18 for details.

Note Configuration parameters on the **Code Generation** pane of the Configuration Parameters dialog box do not affect simulation in either Normal or Accelerated mode. **Code Generation** parameters affect only code generation by Simulink Coder itself. Accelerated 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.

The tables in the following sections list Configuration parameter options that can cause problems if set:

- In certain ways, as indicated in the table
- Differently in a referenced model than in a parent model

Where possible, Simulink resolves violations of these requirements automatically, but most cases require changes to the parameters in some or all models.

Configuration Requirements for All Referenced Model Simulation

Dialog Box Pane	Option	Requirement
Solver	Start time	The start time of the top model and all referenced models must be the same, but need not be zero.
	Stop time	Simulink uses Stop time of the top model for simulation, overriding any differing Stop time in a submodel.
	Type Solver	The Type and Solver of the top model apply throughout the hierarchy. See “Solver Requirements” on page 6-50.
Data Import/Export	Initial state	Can be on for the top model, but must be off for a referenced model.
Optimization	Inline parameters	Can be on or off for a top model, but must be on for a referenced model. See “Inline Parameter Requirements” on page 6-51.
	Application lifespan (days)	Must be the same for top and referenced models.

Dialog Box Pane	Option	Requirement
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 in some cases. See “Model Instance Requirements” on page 6-51.
Hardware Implementation	Production hardware options	All values must be the same for top and referenced models.

Solver Requirements. Model referencing works with both fixed-step and variable-step solvers. All models in a model reference 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 submodel.

Top Model Solver Type	Submodel Solver Type	Compatibility
Fixed Step	Fixed Step	Allowed
Variable Step	Variable Step	Allowed
Variable Step	Fixed-step	Allowed unless the submodel is multi-rate 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 submodel solver, one or both models must change to use compatible solvers. For information about solvers, see “Solvers” on page 3-21 and “Choose a Solver” on page 14-9.

Inline Parameter Requirements. Simulink requires enabling **Configuration Parameters > Optimization > Inline parameters** (see “Inline parameters”) for all referenced models in a reference hierarchy. The top model can enable or disable inline parameters. If a referenced model disables inline parameters, and you try to simulate the parent model:

- For a Normal mode submodel, Simulink generates an error and cancels the build. All models and compiled files remain unchanged after the failed build.
- For an Accelerator mode submodel, Simulink temporarily enables inline parameters, posts no warning, and builds the model. Inline parameters remain disabled after the build completes.

Simulink ignores tunable parameter specifications in the Model Parameter Configuration dialog box for both the top model and referenced models. Do not use this dialog box to override the inline parameters optimization for selected parameters to permit them to be tuned. Instead, see “Parameterize Model References” on page 6-54 for alternate techniques.

Model Instance Requirements. 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 **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** parameter provides this specification. See “Total number of instances allowed per top model” for more information. 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 reference 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 reference. An error occurs if the model cannot be multiply referenced, 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 “General Reusability Limitations” on page 6-84 and “Accelerator Mode Reusability Limitations” on page 6-88.

Model Structure Requirements

The following requirements relate to the structure of a model reference hierarchy.

Signal Propagation Requirements

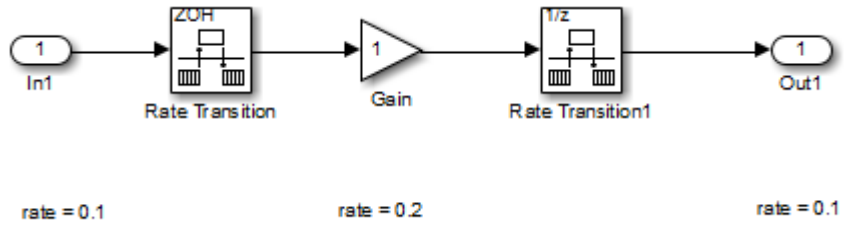
The signal name must explicitly appear on any signal line connected to an Output block of a referenced model. A signal connected to an unlabeled line of an Output block of a referenced model cannot propagate out of the Model block to the parent model.

Bus Usage Requirements

A bus that propagates between a parent model and a referenced model must be nonvirtual. Use the same bus object to specify the properties of the bus in both the parent and the referenced model. Define the bus object in the MATLAB workspace. See “Composite Signals” on page 51-3 for more information.

Sample Time Requirements

The first nonvirtual block connected to a root-level Inport or Output block of a referenced model must have the same sample time as the port to which it connects. Use Rate Transition blocks to match input and output sample times, as illustrated in the following diagram.



Parameterize Model References

In this section...
“Introduction” on page 6-54
“Global Nontunable Parameters” on page 6-55
“Global Tunable Parameters” on page 6-55
“Using Model Arguments” on page 6-55

Introduction

A parameterized referenced model obtains values that affect the behavior of the referenced model from some source outside the model. Changing the values changes the behavior of the model, without recompiling the model.

Due to the constraints described in “Inline Parameter Requirements” on page 6-51, you cannot use the Model Parameter Configuration dialog box to parameterize referenced models. Simulink provides three other techniques that you can use to parameterize referenced models:

- Global nontunable parameters
- Global tunable parameters
- Model arguments

Global parameters work the same way with referenced models that they do with any other model construct. Each global parameter has the same value in every instance of a referenced model that uses it. Model arguments allow you to provide different values to each instance of a referenced model. Each instance can then behave differently from the others. The effect is analogous to calling a function more than once with different arguments in each call.

You can use a structure parameter to group variables for parameterizing model references. For details, see “Structure Parameters” on page 24-17.

Global Nontunable Parameters

A *global nontunable parameter* is a MATLAB variable or a Simulink.Parameter object whose storage class is auto. The parameter must exist in the MATLAB workspace.

Using a global nontunable parameter in a referenced model sets the parameter value before the simulation begins. This allows you to control the behavior of the referenced model. All instances of the model use the same value. You cannot change the value during simulation, but you can change it between one simulation and the next. The change requires rebuilding the model in which the change occurs, but not any models that it references. See “Specify Parameter Values” on page 24-6 for details.

Global Tunable Parameters

A *global tunable parameter* is a Simulink.Parameter object whose storage class is other than auto. The parameter exists in the MATLAB workspace.

Using a global tunable parameter in a referenced model allows you to control the behavior of the referenced model by setting the parameter value. All instances of the model use the same value. You can change the value during simulation or between one simulation and the next. The change does not require rebuilding the model in which the change occurs, or any models that it references. See “Tunable Parameters” on page 24-14 for details.

To reference a model that uses tunable parameters defined with the “Model Parameter Configuration Dialog Box”, change the model to implement tunability another way. To facilitate this task, Simulink provides a command that converts tunable parameters specified in the Model Parameter Configuration dialog box to global tunable parameters. See `tunablevars2parameterobjects` for details.

Using Model Arguments

Model arguments let you parameterize references to the same model so that each instance of the model behaves differently. Without model arguments, a variable in a referenced model has the same value in every instance of the model. Declaring a variable to be a model argument allows each instance of the model to use a different value for that variable.

To create model arguments for a referenced model:

- 1 Create MATLAB variables in the model workspace.
- 2 Add the variables to a list of model arguments associated with the model.
- 3 Specify values for those variables separately in each Model block that references the model

The values specified in the Model block replace the values of the MATLAB variables for that instance of the model.

A referenced model that uses model arguments can also appear as a top model or a standalone model. No Model block exists to provide model argument values. The model uses the values of the MATLAB variables as defined in the model workspace. You can use the same model, without changing it, as a top model, a standalone model, and a parameterized referenced model.

The `sldemo_md1ref_datamngt` model demonstrates techniques for using model arguments. The model passes model argument values to referenced models through masked Model blocks. Such masking can be convenient, but is independent of the definition and use of model arguments themselves. See “Masking” for information about masking.

The rest of this section describes techniques for declaring and using model arguments to parameterize a referenced model independently of any Model block masking. The steps are:

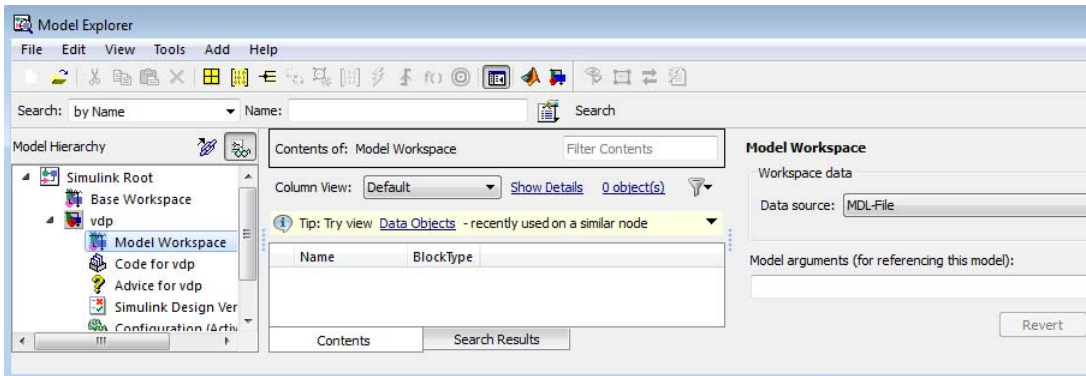
- Create MATLAB variables in the model workspace.
- Register the variables to be model arguments.
- Assign values to those arguments in Model blocks.

Creating the MATLAB Variables

To create MATLAB variables for use as model arguments:

- 1 Open the model for which you want to define model arguments.
- 2 Open the Model Explorer.

- 3** In the Model Explorer **Model Hierarchy** pane, select the workspace of the model:



- 4** From Model Explorer's **Add** menu, select **MATLAB Variable**.

A new MATLAB variable appears in the **Contents** pane with a default name and value.

- 5** In the **Contents** pane:
- a** Change the default name of the new MATLAB variable to a name that you want to declare as a model argument.
 - b** If you also use the model as a top or standalone model, specify the value for that variable for use in that context. This value must be numeric.
 - c** If the variable type does not match the dimensions and complexity of the model argument, specify a value that has the correct type. This type must be numeric.
- 6** Repeat adding and naming MATLAB variables until you have defined all the variables that you need.

Register the Model Arguments

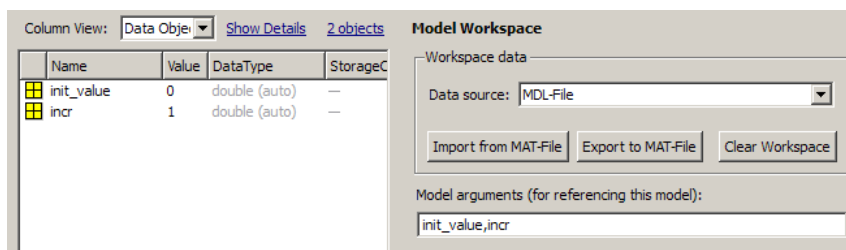
To register MATLAB variables as model arguments:

- 1** Again, in the Model Explorer **Model Hierarchy** pane, select the workspace of the model.

The Dialog pane displays the Model Workspace dialog.

- 2 In the Model Workspace dialog, enter the names of the MATLAB variables that you want to declare as model arguments. Use a comma-separated list in the **Model arguments** field.

For example, suppose you added two MATLAB variables named `init_value` and `incr`, and you declared them to be model arguments. The **Contents** and **Dialog** panes of the Model Explorer could look like this:

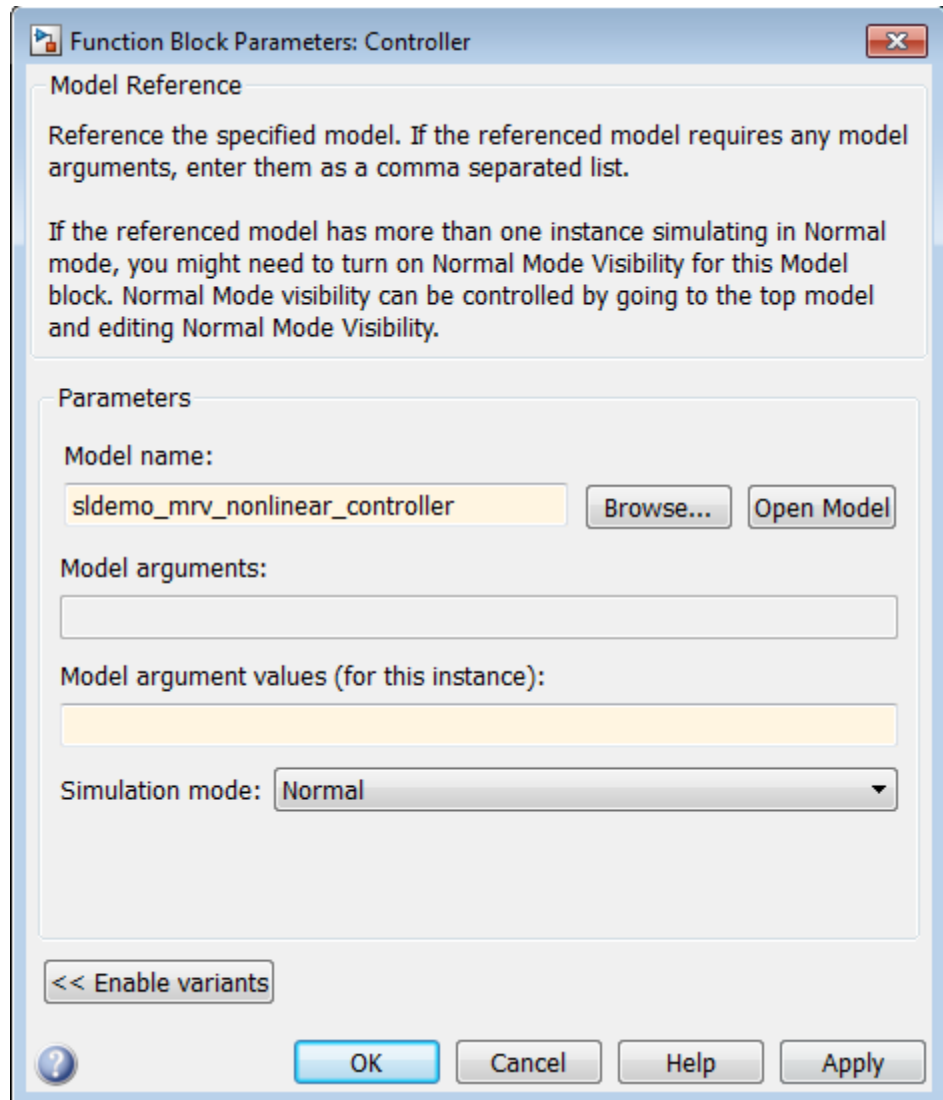


- 3 Click **Apply** to confirm the entered names.

Assign Model Argument Values

If a model declares model arguments, assign values to those arguments in each Model block that references the model. Failing to assign a value to a model argument causes an error: the value of the model argument does *not* default to the value of the corresponding MATLAB variable. That value is available only to a standalone or top model. To assign values to the arguments of a referenced model:

- 1 Open the block parameters dialog box of the Model block by right-clicking the block and choosing **Block Parameters (ModelReference)** from the context menu.



The second field, **Model arguments**, specifies the same MATLAB variables, in the same order, that you previously typed into the **Model arguments** field of the Model Workspace dialog. This field is not editable.

It provides a reminder of which model arguments need values assigned, and in what order.

- 2** In the **Model argument values** field, enter a comma-delimited list of values for the model arguments that appear in the **Model arguments** field. Simulink assigns the values to arguments in positional order, so they must appear in the same order as the corresponding arguments.

You can enter the values as literal values, variable names, MATLAB expressions, and Simulink parameter objects. Any symbols used resolve to values as described in “Symbol Resolution Process” on page 4-83. All values must be numeric (including objects with numeric values).

Note Simulink generates an error if you use a model argument in an expression that is nontunable because of “Tunable Expression Limitations”.

The value for each argument must have the same dimensions and complexity as the MATLAB variable that defines the model argument in the model workspace. The data types need not match. If necessary, the Simulink software casts a model argument value to the data type of the corresponding variable.

- 3** Click **OK** or **Apply** to confirm the values for the Model block.

When the model executes in the context of that Model block, the **Model arguments** have the values specified in the **Model argument values** field of the Model block.

Conditional Referenced Models

In this section...

“Kinds of Conditional Referenced Models” on page 6-61

“Working with Conditional Referenced Models” on page 6-62

“Create Conditional Models” on page 6-63

“Reference Conditional Models” on page 6-65

“Simulate Conditional Models” on page 6-66

“Generate Code for Conditional Models” on page 6-66

“Requirements for Conditional Models” on page 6-67

Kinds of Conditional Referenced Models

You can set up referenced models so that they execute conditionally, similar to conditional subsystems. For information about conditional subsystems, see “Conditional Subsystems” on page 7-2.

You can use the following kinds of conditionally executed referenced models:

- Enabled
- Triggered
- Enabled and triggered
- Function-call

Enabled Models

Use an Enable block to insert an enable port in a model. Add an enable port to a model if you want a referenced model to execute at each simulation step for which the control signal has a positive value.

To see an example of an enabled *subsystem*, see `enablesub`. A corresponding enabled referenced model would use the same blocks as are in the enabled subsystem.

Triggered Models

Use a Trigger block to insert a trigger port in a model. Add a trigger port to a model if you want to use an external signal to trigger the execution of that model. You can add a trigger port to a root-level model or to a subsystem.

This section focuses on models that contain a trigger port with an edge-based trigger type (rising, falling, or either).

To view a model that illustrates how you can use trigger ports in referenced models, see the Introduction to Managing Data with Model Reference example. In that example, see the “Top Model: Scheduling Calls to the Referenced Model” section.

Triggered and Enabled Models

A triggered and enabled model executes once at the time step for which a trigger event occurs, if the enable control signal has a positive value at that step.

Function-Call Models

Simulink allows certain blocks to control execution of a referenced model during a time step, using a function-call signal. Examples of such blocks are a Function-Call Generator or an appropriately configured custom S-function. See “Create a Function-Call Subsystem” on page 7-48 for more information. A referenced model that you can invoke in this way is a *function-call model*.

For an example of a function-call model, see the `sldemo_md1ref_fcncall` model.

Working with Conditional Referenced Models

Use a similar approach for each kind of conditionally executed referenced model for these tasks:

- “Create Conditional Models” on page 6-63
- “Reference Conditional Models” on page 6-65
- “Simulate Conditional Models” on page 6-66
- “Generate Code for Conditional Models” on page 6-66

Each kind of conditionally executed model has some model creating requirements. For details, see “Requirements for Conditional Models” on page 6-67.

Create Conditional Models

To create a conditional model:

- 1 At the root level of the referenced model, insert one of the following blocks:

Kind of Model	Blocks to Insert
Enabled	Enable
Triggered	Trigger
Triggered and Enabled	Trigger and Enable
Function-Call	Trigger

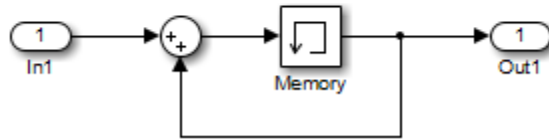
For an enabled model, go to Step 3.

- 2 For the Trigger block, set the **Trigger type** parameter, based on the kind of model:

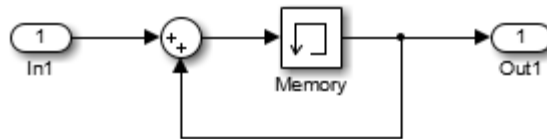
Kind of Model	Trigger Type Parameter Setting
Triggered Triggered and enabled	One of the following: <ul style="list-style-type: none"> • rising • falling • either
Function-Call	function-call

- 3 Create and connect other blocks to implement the model.

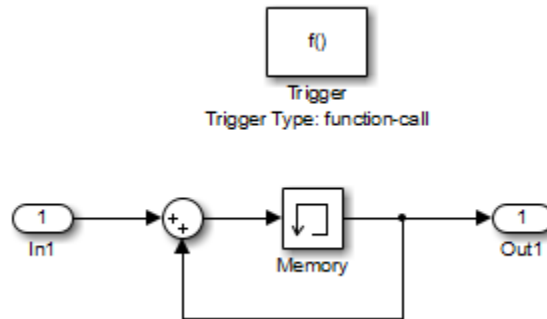
Enabled model example:



Triggered model example:



Function-call model example:



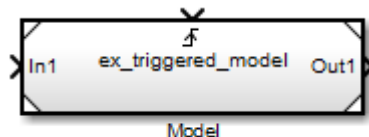
- 4** Ensure that the model satisfies the requirements for a conditional model. See the appropriate section:
- “Enabled Model Requirements” on page 6-67
 - “Triggered Model Requirements” on page 6-67
 - “Function-Call Model Requirements” on page 6-67

Reference Conditional Models

To create a reference to a conditional model:

- 1** Add a Model block to the model that you want to reference the triggered model. See “Create a Model Reference” on page 6-8 for details.

The top of the Model block displays an icon that corresponds to the kind of port used in the referenced model. For example, for a triggered model, the top of the Model block displays the following icon.



For enabled, triggered, and triggered and enabled models, go to Step 3.

- 2** For a function-call model, connect a Stateflow chart, Function-Call Generator block, or other function-call-generating block to the function-call port of the Model block. The signal connected to the port must be scalar.

- 3 Create and connect other blocks to implement the parent model.
- 4 Ensure that the referencing model satisfies the conditions for model referencing. See “Simulink Model Referencing Requirements” on page 6-47 and “Model Referencing Limitations” on page 6-83 for details.

Simulate Conditional Models

Use Normal, Accelerator, or Rapid Accelerator mode to simulate a conditional model.

You can run a standalone simulation of a referenced 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. In the Trigger or Enable block, in the **Signal Attributes** pane of the Block Parameters dialog box, specify values to lock down the signal data type, dimension, and sample time.

To run a standalone simulation of a conditional model, specify the inputs using the **Configuration Parameters > Data Import/Export > Input** parameter. The following conditions apply when you use the “Input” parameter for trigger and enable port 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.

For details about how to specify the input, see “Techniques for Importing Signal Data” on page 48-72.

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**.

Generate Code for Conditional Models

You can build model reference Simulink Coder and SIM targets for referenced models that contain a trigger or enable port. You cannot generate standalone Simulink Coder or PIL code. For information about code generation for

referenced models, see “Reusable Code and Referenced Models” and “Generate Code for Referenced Models”.

Requirements for Conditional Models

Conditional models must meet the requirements for:

- Conditional subsystems (see “Conditional Subsystems”)
- Referenced models (see “Simulink Model Referencing Requirements” on page 6-47)

In addition, conditional models must meet the requirements described below.

Enabled Model Requirements

- Multi-rate enabled models cannot use multi-tasking solvers. You must 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.
- The 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 Model Requirements

The 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.

Function-Call Model Requirements

- A function-call model cannot have an output 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.
 - 2 Enable the **Exclude this block from 'Block reduction' optimization** option of the inserted block.

- The referencing 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 referencing 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.

Protected Model

A protected model provides the ability to deliver a model without revealing the intellectual property of the model. A protected model is a referenced model that hides all block and line information. It does not use encryption technology. Creating a protected model requires a Simulink Coder license. A third party that receives a protected model must match the platform and the version of Simulink for which the protected model was generated.

You can view the contents of a protected model if the Web view functionality is enabled during creation. Otherwise, you cannot log signals in a protected model. Protected models do not appear in the model hierarchy in the Model Explorer.

Simulating a protected model requires that the protected model:

- Be available somewhere on the MATLAB path.
- Be referenced by a Model block in a model that executes in Normal or Accelerator mode. To run in Accelerator mode, the protected model must include code generation capabilities.
- Receives from the Model block the values needed by any defined model arguments.
- Connects via the Model block to input and output signals that match the input and output signals of the protected model.

To locate protected models in your model:

- The MATLAB Folder Browser shows a small image of a lock on the node for the protected model file.
- A Model block that references a protected model shows a small image of a shield in the lower left corner of the Model block.

If you use a protected model for operations like viewing a Web view, update diagram, simulation, or code generation, then the licenses used in the protected model will be checked out before those operations begin. The creator of the protected model can view the licenses of a protected model that will be checked out by following this procedure:

- 1 Open the unprotected model.
- 2 Select **Analysis > Model Dependencies > Generate Manifest**.
- 3 Click **OK** to generate the manifest.
- 4 The licenses required for the model are listed in the Model Manifest Report under *Toolboxes required by this model*.

For more information, see “Use Protected Model in Simulation” on page 6-71. For more information about creating protected referenced models, see “Protect a Referenced Model”.

Use Protected Model in Simulation

When you receive a protected model, it might be included in a protected model package. The package could include additional files, such as a harness model and a MAT-file. A protected model file has an `.slxp` extension. A typical workflow for using a protected model in a simulation is:

- 1 If necessary, unpack the files according to any accompanying directions.
- 2 If there is a MAT-file containing workspace definitions, load that MAT-file.
- 3 If there is a harness model, copy the Model block referencing the protected model into your model.
- 4 If the protected model is password protected, then right-click the protected-model badge icon and select **Authorize**. Enter the required passwords, and then click **OK**.
- 5 If a protected model report was generated when the protected model was created, double-click the Model block to open it. In the **Summary** of the report, check that your Simulink version and platform match the software and platform used to create the protected model.
- 6 Connect signals to the Model block that match its input and output port requirements.
- 7 Provide any needed model argument values. See “Assign Model Argument Values” on page 6-58.

There are also other ways to include the protected model into your model:

- Use your own Model block rather than the Model block in the harness model.

Note When you change a Model block to reference a protected model, the **Simulation mode** of the block is set to **Accelerator**. You cannot change this mode. Furthermore, you cannot use the protected reference model block in **External** mode or in **Rapid Accelerator** mode.

- Start with the harness model, add more constructs to it, and use it in your model.
- Use the protected model as a variant in a Model Variant block, as described in “Set Up Model Variants” on page 8-8.
- Apply a mask to the Model block that references the protected model. See “Masking”.
- Configure a callback function, such as **LoadFcn**, to load the MAT-file automatically. See “Callbacks for Customized Model Behavior” on page 4-56.

Now you can simulate the model that includes the protected model. Because the protected model is set to Accelerator mode, the simulation produces the same outputs that it did when used in Accelerator mode in the source model.

Protected Model Web View

The Web view is a read-only reference of the protected model. To view the Web view of a protected model, right-click the protected-model badge icon and select **Show Web view**. Hover over a block in the model Web view to show the parameter values. A Report Generator license is required to use the model Web view.

If the Web view is password protected, then right-click the protected-model badge icon and select **Authorize**. In the **Model view** box, enter the password, and then click **OK**.

Inherit Sample Times

In this section...

“How Sample-Time Inheritance Works for Model Blocks” on page 6-73

“Conditions for Inheriting Sample Times” on page 6-73

“Determining Sample Time of a Referenced Model” on page 6-74

“Blocks That Depend on Absolute Time” on page 6-74

“Blocks Whose Outputs Depend on Inherited Sample Time” on page 6-76

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. Additionally, allowing a Model block to inherit sample time maximizes its reuse potential. For example, a model might 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, and so on).

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 6-74
- 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 6-76.

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.

Determining Sample Time of a Referenced Model

To determine whether a referenced model can inherit its sample time, set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane 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(gcf, '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 (only when the chart uses 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. For this reason, when building a submodel that does not need to run at a specified rate, Simulink checks whether the model contains any blocks whose outputs are functions of the inherited simulation time. This includes checking S-Function blocks. If Simulink finds any such blocks, it specifies a default sample time. Simulink displays an error if you have set the **Configuration Parameters > Solver > Periodic sample time constraint** to **Ensure sample time independent**. 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
- 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 “Sample Times” for information on how to create S-functions that declare whether their output depends on their inherited sample time.

To avoid simulation errors with referenced models that inherit their sample time, do not include S-functions in the referenced models that fail to declare whether their output depends on their inherited sample time. 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.

Refresh Model Blocks

Refreshing a Model block updates its internal representation to reflect changes in the interface of the model that it references.

Examples of when to refresh a Model block include:

- Refresh a Model block that references model that has gained or lost a port.
- Refresh all the Model blocks that reference a model whose interface has changed.

You do not need to refresh a Model block if the changes to the interface of the referenced model do not affect how the referenced model interfaces to its parent.

To update a specific Model block, from the context menu of the Model block, select **Subsystem & Model Reference > Refresh Selected Model Block**.

To refresh all Model blocks in a model (as well as linked blocks in a library or model), in the Simulink Editor select **Diagram > Refresh Blocks**. You can also refresh a model by starting a simulation or generating code.

You can use Simulink diagnostics to detect changes in the interfaces of referenced models that could require refreshing the Model blocks that reference them. The diagnostics include:

- **Model block version mismatch**
- **Port and parameter mismatch**

S-Functions with Model Referencing

In this section...
“S-Function Support for Model Referencing” on page 6-78
“Sample Times” on page 6-78
“S-Functions with Accelerator Mode Referenced Models” on page 6-79
“Using C S-Functions in Normal Mode Referenced Models” on page 6-80
“Protected Models” on page 6-80
“Simulink® Coder™ Considerations” on page 6-80

S-Function Support for Model Referencing

Each kind of S-function provides its own level of support for model referencing.

Type of S-Function	Support for Model Referencing
Level-1 MATLAB S-function	Not supported
Level-2 MATLAB S-function	<ul style="list-style-type: none"> • Supports Normal and Accelerator mode • Accelerator mode requires a TLC file
Handwritten C MEX S-function	<ul style="list-style-type: none"> • Supports Normal and Accelerator mode • May be inlined with TLC file
S-Function Builder	Supports Normal and Accelerator mode
Legacy Code Tool	Supports Normal and Accelerator mode

Sample Times

Simulink software assumes that the output of an S-function does not depend on an inherited sample time unless the S-function explicitly declares a dependence on an inherited sample time.

You can control inheriting sample time by using `ssSetModelReferenceSampleTimeInheritanceRule` in different ways,

depending on whether an S-function permits or precludes inheritance. For details, see “Inherited Sample Time for Referenced Models”.

S-Functions with Accelerator Mode Referenced Models

For a referenced model that executes in Accelerator mode, set the **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** to One if the model contains an S-function that is either:

- Inlined, but has not set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag
- Not inlined

Inlined S-Functions with Accelerator Mode Referenced Models

For Accelerator mode referenced models, if the referenced model contains an S-function that should be inlined 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 the following cases:

- The model uses a variable-step solver.
- Simulink Coder generated the S-function.
- The S-function supports use of fixed-point numbers as inputs, outputs, or parameters.
- The model is referenced more than once in the model reference hierarchy. To work around this limitation, use Normal mode.
- The S-function uses string parameters.

Using C S-Functions in 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”.

Use the `ssSetModelReferenceNormalModeSupport SimStruct` function to specify whether an S-function can be used in a Normal mode referenced model.

You may need to modify S-functions that are used by a model so that the S-functions work with multiple instances of referenced models in Normal mode. The S-functions must indicate explicitly that they support 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.

Simulink Coder Considerations

A referenced model in Accelerator mode cannot use S-functions generated by the Simulink Coder software.

Noninlined S-functions in referenced models are supported when generating Simulink Coder code.

The Simulink Coder S-function target does not support model referencing.

For general information about using Simulink Coder and model referencing, see “Referenced Models”.

Buses in Referenced Models

To have bus data cross model reference boundaries, use a nonvirtual bus. Use a bus object (`Simulink.Bus`) to define the bus.

For an example of a model referencing model that uses buses, see `sldemo_mdref_bus`. For more information, see “Bus Data Crossing Model Reference Boundaries” on page 51-98.

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 of the signals configured for signal logging for a model reference hierarchy. For details, see “Models with Model Referencing: Overriding Signal Logging Settings” on page 48-50.

Simulink does not log signals inside referenced models in Rapid Accelerator mode.

For additional information, see “Export Signal Data Using Signal Logging” on page 48-21.

Model Referencing Limitations

In this section...

“Introduction” on page 6-83
“Limitations on All Model Referencing” on page 6-83
“Limitations on Normal Mode Referenced Models” on page 6-86
“Limitations on Accelerator Mode Referenced Models” on page 6-87
“Limitations on Rapid Accelerator Mode Referenced Models” on page 6-90
“Limitations on PIL Mode Referenced Models” on page 6-90

Introduction

The following limitations apply to model referencing. In addition, a model reference hierarchy must satisfy all the requirements listed in “Simulink Model Referencing Requirements” on page 6-47.

Limitations on All Model Referencing

Index Base Limitations

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.

General Reusability Limitations

If a referenced model has any of the following characteristics, the model must specify **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** as One. No other instances of the model can exist in the hierarchy. An error occurs if you do not set the parameter correctly, or if more than one instance of the model exists in the hierarchy. The model characteristics that require that the **Total number of instances allowed per top model** setting be One are:

- The model contains any To File blocks
- The model references another model which is set to single instance
- The model contains a state or signal with non-auto storage class
- The model uses any of the following Stateflow constructs:
 - Stateflow graphical functions
 - Machine-parented data

Block Mask Limitations

- Mask callbacks cannot add Model blocks. Also, mask callbacks cannot change the Model block name or simulation mode. These invalid callbacks generate an error.
- If a mask specifies the name of the model that a Model block references, 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 model that the Mask block references. 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

For information about creating and using block masks, see “Masking”.

Simulink Tool Limitations

Working with the Simulink Debugger in a parent model, you can set breakpoints at Model block boundaries. Setting those breakpoints allows you to look at the input and output values of the Model block. However, you cannot set a breakpoint inside the model that the Model block references. See “Debugging” for more information.

Stateflow Limitations

You cannot reference a model multiple times in the same model reference hierarchy if that model that contains a Stateflow chart that:

- Contains exported graphical functions
- Is part of a Stateflow model that contains machine-parented data

Subsystem Limitations

- You cannot place a Model block in an iterator subsystem, if the Model block references a model that contains Assignment blocks that are not in an iterator subsystem.
- In a configurable subsystem with a Model block, during model update, do not change the subsystem that the configurable subsystem selects.

S-function Target Limitation

The Simulink Coder S-function target does not support model referencing.

Other Limitations

- 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.
- A referenced model can input or output only those user-defined data types that are fixed-point or that `Simulink.DataType` or `Simulink.Bus` objects define.
- 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 “Import and Export State Information for Referenced Models” on page 48-149.
- A referenced model cannot directly access the signals in a multirate bus. To overcoming this limitation, see [Connecting Multi-Rate Buses to Referenced Models](#).
- A continuous sample time cannot be propagated to a Model block that is sample-time independent.
- Goto and From blocks cannot cross model reference boundaries.
- You cannot print a referenced model from a top model.
- To use a masked subsystem in a referenced model that uses model arguments, do not create in the mask workspace a variable that derives its value from a mask parameter. Instead, use blocks under the masked subsystem to perform the calculations for the mask workspace variable.

Limitations on Normal Mode Referenced Models

Normal Mode Visibility for Multiple Instances of a Referenced Model

You can simulate a model that has multiple instances of a referenced model that are in Normal mode. All of the instances of the referenced model are part of the simulation. However, Simulink displays only one of the instances in a model window. The Normal Mode Visibility setting determines which instance Simulink displays. Normal Mode Visibility includes the display of Scope blocks and data port values.

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 “Set Up a Model with Multiple Instances of a Referenced Model in Normal Mode” on page 6-28.

Simulink Profiler

In Normal mode, enabling the Simulink Profiler on a parent model does not enable profiling for referenced models. You must enable profiling separately for each referenced model. See “Capture Performance Data” on page 22-32.

Limitations on Accelerator Mode Referenced Models

Subsystem Limitations

If you generate code for an atomic subsystem as a reusable function, when you use Accelerator mode, the inputs or outputs that connect the subsystem to a referenced model can affect code reuse. See “Reusable Code and Referenced Models” for details.

Simulink Tool Limitations

Simulink tools that require access to the internal data or the configuration of a model have no effect on referenced models executing in Accelerator mode. Specifications made and actions taken by such tools are ignored and effectively do not exist. Examples of tools that require access to model internal data or configuration include:

- Model Coverage
- Simulink Report Generator
- Simulink Debugger
- Simulink Profiler

Runtime Checks

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, and MATLAB Function blocks)

Data Logging Limitations

The following logging methods have no effect when specified in referenced models executing in Accelerator mode:

- To Workspace blocks (for formats other than `Timeseries`)
- Scope blocks
- All types of runtime display, such as Port Values Display

During simulation, the result is the same as if the constructs did not exist.

Accelerator Mode Reusability Limitations

You must set **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** to One for a referenced model that executes in Accelerator mode and has any of the following characteristics:

- A subsystem that is marked as function
- An S-function that is:
 - Inlined but has not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`
 - Not inlined
- A function-call subsystem that:
 - Has been forced by Simulink to be a function
 - Is called by a wide signal

An error occurs in either of these cases:

- You do not set the parameter correctly.
- Another instances of the model exists in the hierarchy, in either Normal mode or Accelerator mode

Customization Limitations

- For restrictions that apply to grouped custom storage classes in referenced models in Accelerator mode, see “Custom Storage Class Limitations”.

- Simulation target code generation for referenced models in Accelerator mode does not support data type replacement.

S-Function Limitations

- If a referenced model in Accelerator mode contains an S-function that should be inlined 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.
- You cannot use the Simulink Coder S-function target in a referenced model in Accelerator mode.
- A referenced model in Accelerator mode cannot use noninlined S-functions in the following cases:
 - The model uses a variable-step solver.
 - Simulink Coder generated the S-function.
 - The S-function supports use of fixed-point numbers as inputs, outputs, or parameters.
 - The S-function uses string parameters.
 - The model is referenced more than once in the model reference hierarchy. To work around this limitation:
 - 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.

Limitation

A MATLAB Function block in a referenced model that executes in Accelerator mode cannot call MATLAB functions.

Stateflow Limitation

A Stateflow chart in a referenced model that executes in Accelerator mode cannot call MATLAB functions.

MATLAB Function Block Limitation

A MATLAB Function block in a referenced model that executes in Accelerator mode cannot call MATLAB functions.

Other Limitations

- When you create a model, you cannot use that model as an Accelerator mode referenced model until you have saved the model to disk. You can work around this limitation by setting the model to Normal mode. See “Specify the Simulation Mode” on page 6-23.
- When the `sim` command executes a referenced model in Accelerator mode, the source workspace is always the MATLAB base workspace.
- Accelerator mode does not support the **External mode** option. If you enable the **External mode** option, Accelerator mode ignores it.

Limitations on Rapid Accelerator Mode Referenced Models

- Simulink does not log signals inside referenced models in Rapid Accelerator mode.
- Simulink does not update a Model block with a `sim` viewing device.

Limitations on PIL Mode Referenced Models

- Only one branch (top model and all subordinates) in a model reference hierarchy can execute in PIL mode.
- If you create a model, you cannot use that model as a PIL mode referenced model until you have saved the model to disk. To work around this limitation, set the model to Normal mode. See “Specify the Simulation Mode” on page 6-23.

For more information about using PIL mode with model referencing, see “Referenced Models” in the Embedded Coder documentation.

Create Conditional Subsystems

- “Conditional Subsystems” on page 7-2
- “Export-Function Models” on page 7-4
- “Create an Enabled Subsystem” on page 7-17
- “Create a Triggered Subsystem” on page 7-33
- “Create an Action Subsystem” on page 7-38
- “Create a Triggered and Enabled Subsystem” on page 7-42
- “Create a Function-Call Subsystem” on page 7-48
- “Conditional Execution Behavior” on page 7-50
- “Conditional Subsystem Output Initialization” on page 7-56
- “Specify or Inherit Conditional Subsystem Initial Values” on page 7-61
- “Set Initialization Mode to Simplified or Classic” on page 7-63
- “Convert from Classic to Simplified Initialization Mode” on page 7-64
- “Address Classic Mode Issues by Using Simplified Mode” on page 7-65

Conditional Subsystems

A subsystem is a set of blocks that have been replaced by a single block called a Subsystem block. This chapter describes a special kind of subsystem for which execution can be externally controlled. For information that applies to all subsystems, see “Create a Subsystem” on page 4-37.

A *conditional subsystem*, also known as a *conditionally executed subsystem*, is a subsystem whose execution depends on the value of an input signal. The signal that controls whether a subsystem executes is called the *control signal*. The signal enters a subsystem block at the *control input*.

Conditional subsystems can be very useful when you are building complex models that contain components whose execution depends on other components. Simulink supports the following types of conditional subsystems:

- An *enabled subsystem* executes while the control signal is positive. It starts execution at the time step where the control signal crosses zero (from the negative to the positive direction) and continues execution as long as the control signal remains positive. For a more detailed discussion, see “Create an Enabled Subsystem” on page 7-17.
- A *triggered subsystem* executes once each time a trigger event occurs. A trigger event can occur on the rising or falling edge of a trigger signal, which can be continuous or discrete. For more information about triggered subsystems, see “Create a Triggered Subsystem” on page 7-33.
- A *triggered and enabled subsystem* executes once at the time step for which a trigger event occurs if the enable control signal has a positive value at that step. See “Create a Triggered and Enabled Subsystem” on page 7-42 for more information.
- 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*, and *for*). See “Use Control Flow Logic” on page 4-46 for more information.

Note The Simulink software imposes restrictions on connecting blocks with a constant sample time to the output port of a conditional subsystem. See “Using Blocks with Constant Sample Times in Enabled Subsystems” on page 7-28 for more information.

For examples of conditional subsystems, see:

- Simulink Subsystem Semantics
- Triggered Subsystems
- Enabled Subsystems
- Advanced Enabled Subsystems

Export-Function Models

In this section...

“About Export-Function Models” on page 7-4

“Requirements for Export-Function Models” on page 7-5

“Specifying periodic sample time on function-call root-level Inport blocks” on page 7-6

“Execution Order for Function-Call Root-level Inport Blocks” on page 7-7

“Workflow for Export-Function Models” on page 7-10

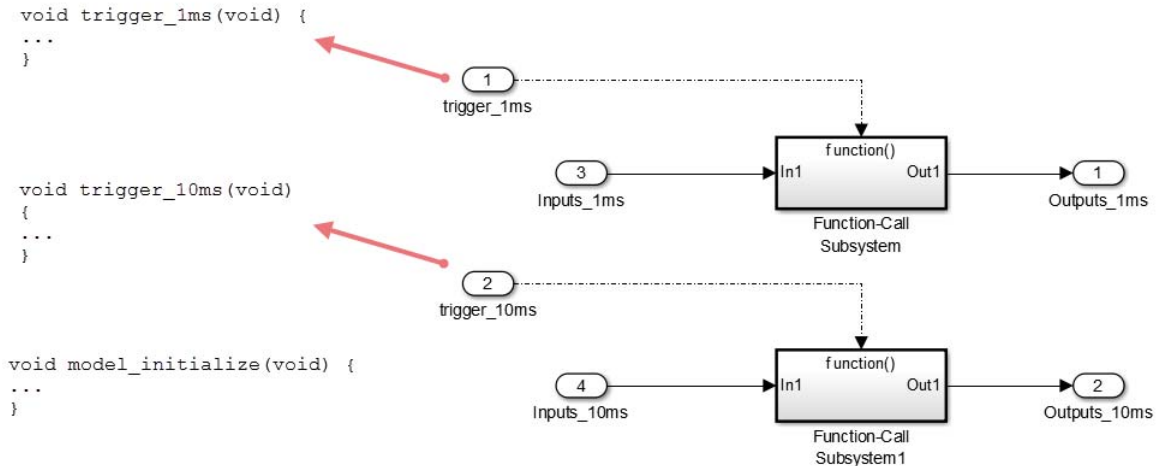
“Nested Export-Function Models” on page 7-14

“Comparison between Export-Function Models and Models with Asynchronous Function-Call Inputs” on page 7-15

About Export-Function Models

Simulink provides the capability to export functions from Simulink models that are invoked by controlling logic that is outside the model. Such models are called export-function models, and are constructed in Simulink by building a model whose functional blocks are made up exclusively of function-call subsystems, function-call model blocks or other export-function models. These blocks are invoked using function-call triggers passed via root-level Inport blocks. You can test these functions in all simulation modes, by providing model inputs through the root-level Inport blocks or by referencing this model in a top model to invoke the function-calls. This is subject to the guidelines described in “Requirements for Export-Function Models” on page 7-5.

The figure below shows an export-function model and the resulting generated functions.



Requirements for Export-Function Models

Export function models must meet a set of requirements that pertain to ensuring that the executable components of the model are made up exclusively of function-call blocks. The model must conform to the requirements listed below.

- The model solver must be a fixed-step discrete solver.
- Configure each root-level Inport block triggering a function-call subsystem to output a function-call trigger. These Inport blocks cannot be connected to an Asynchronous Task Specification block.
- Export-function models generate functions to be integrated with an external environment. Simulink does not generate a step function, and all blocks in these models must be executed in a function-call context. Thus, the model must contain only the following blocks at root level:
 - Function-call blocks (such as function-call subsystem, S-functions, Function-Call Feedback Latch blocks, and referenced function-call models)
 - Inport and Outport blocks

- Blocks with constant sample time (including blocks that resolve to constant sample time)
- Merge and Data Store Memory blocks
- Virtual connection blocks (Function-Call Split, Mux, Demux, Bus Creator, Bus Selector, Signal Specification, and any Virtual Subsystem that contains any of the blocks listed above.)
- Blocks inside function-call subsystems must support code generation. These blocks must not use absolute time or elapsed time unless you specify a discrete sample time on the model's function-call root-level Inport block.
- Data signals connected to root-level Inport and root-level Outport blocks cannot be a virtual bus.
- Data logging and signal-viewer blocks, such as the Scope block, are not allowed at the root level and within the function-call blocks.

Note Two or more function-call root-level Inport blocks must be driven by the same sample time in the top model if any of the following conditions are met.

- The outputs of the function-call blocks they call are being merged.
 - The function-call blocks they call access the same Data Store memory.
 - The function-call blocks they call access input data from the same root-level Inport block.
-

Specifying periodic sample time on function-call root-level Inport blocks

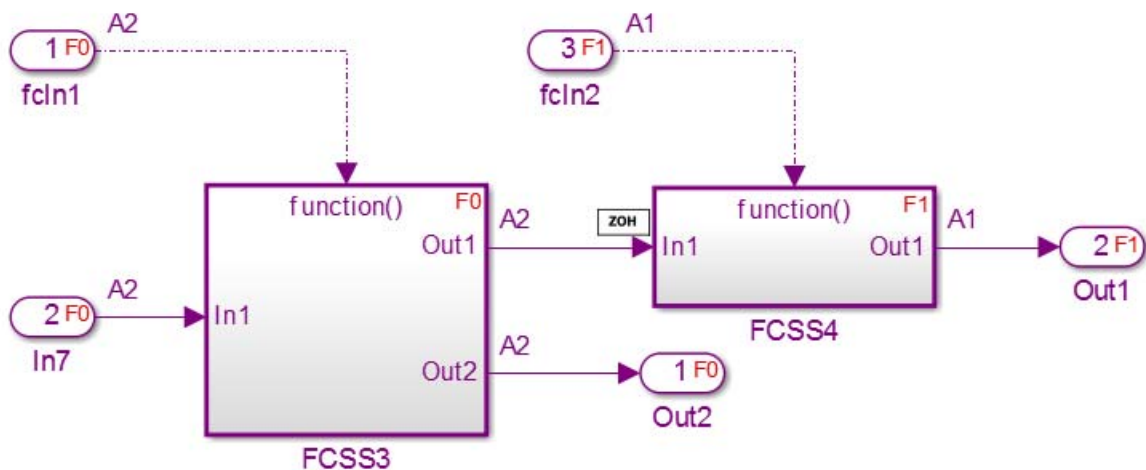
When the sample time is specified on function-call root-level Inport blocks, the function will be executed periodically at that rate. Thus, periodic function-call run-time checks apply if this model is used as a reference model in normal simulation mode.

When using periodic sample time, the model configuration parameter **Fixed-step size (fundamental sample time)** cannot be set to auto. The periodic sample time of the Inport block must be an integer multiple of the model sample time.

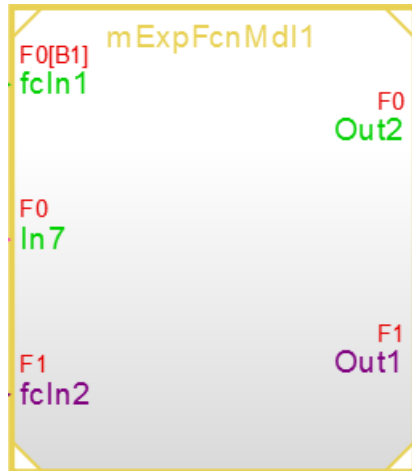
If an Inport block has periodic sample time specified, then its source block in the top model must also have the same sample time. Function-call subsystems driven by these periodic function-call root-level Inport blocks must set their function-call trigger block's **Sample time type** parameter set to **Periodic** and **Sample time** set to -1 (inherited) or the sample time of the function-call root-level Inport block. Periodic sample time function-call subsystems can contain blocks that use elapsed time (e.g. Discrete-Time Integrator) and also blocks that use absolute time (e.g. Digital Clock).

Execution Order for Function-Call Root-level Inport Blocks

You can display the sorted execution order to interpret simulation results. This has no impact on generated code. To display the sorted execution order, select **Display > Blocks > Sorted Execution Order**. In the example below, notice the sorted order for both the function-call triggers. Based on sorted order display labels, fcIn1 (F0) executes before fcIn2 (F1) when both have a sample hit at the same time step.



The referenced export-function model in the top model shows the local execution order of the Inport and Outport blocks in the model.



By default, the sorted execution order follows the Inport block number. Blocks with the lower port number execute first. For control on the sorted execution order for Inport blocks, specify a value for the **Priority** parameter in the Block Properties dialog box. Blocks with lower priority numbers execute first.

If the export-function model is referenced by a top model, there are restrictions on the order of specifying function-calls. These restrictions ensure consistency with standalone simulation results.

- The function-call inputs from the top model must follow the execution order of the function-call Inport blocks in the referenced export-function model.
- In the top model, the same function-call initiator must output function-calls originating within the same sample time. Thus, you cannot use two function-call initiators with the same sample time.

Simulink displays an errors if the top model calls the referenced model functions out of order at any time step. For information on Sorted Execution Order, see “Control and Display the Sorted Order” on page 23-36.

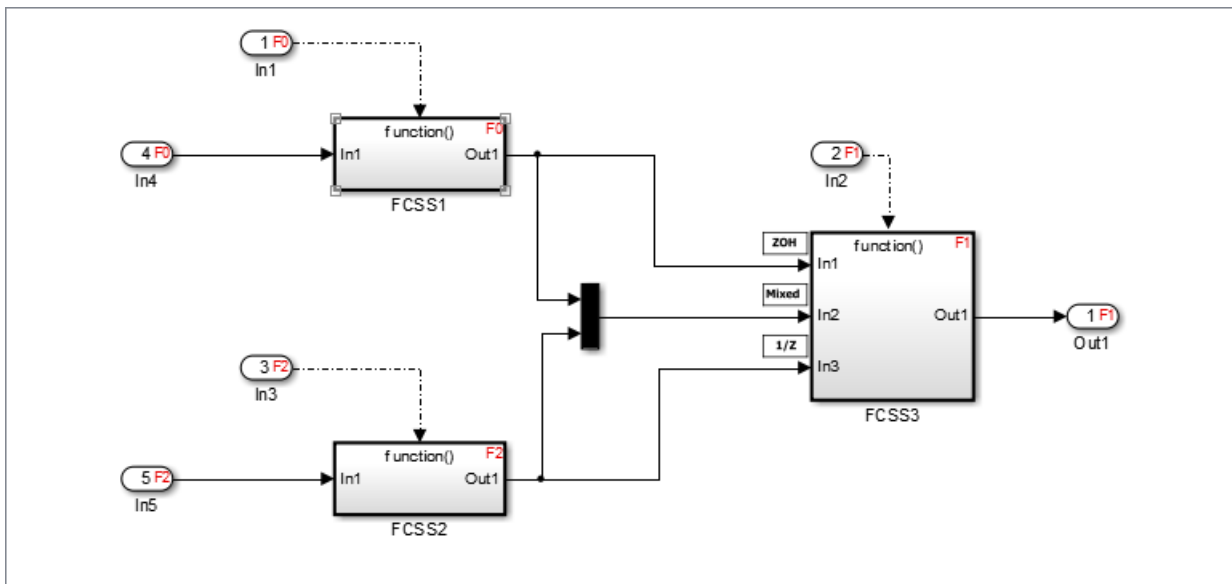
Data Transfer Between Function-Call Subsystems

You must know the timing of the data being transferred between function-call subsystems to correctly understand and interpret simulation results.

To display which subsystem executes first during simulation, the signal lines are annotated with different symbols at the input ports of the subsystems:

- **ZOH** indicates that all source function-call subsystems execute before the function-call block reading this signal.
- **1/z** indicated that all source function-call subsystems execute after the function-call block reading this signal.
- **Mixed** indicates that some source function-call subsystems execute before and some function-call subsystems execute after the function-call block reading this signal.

In the block diagram below, notice the sorted execution order for each block. We can see that for the input port **In1** of subsystems **FCSS3** (F1), the source subsystem **FCSS1** (F0) executes before **FCSS3** (F1). Hence, an annotation of **ZOH** is added next to **In1**. Similarly, **FCSS2** (F2) executes after **FCSS3** (F1). Hence, Simulink adds an annotation of **1/z** next to **In3** of subsystem **FCSS3** (F1). Port **In2** inputs signals from both **FCSS1** (F0) and **FCSS2** (F2). Hence, it has an annotation **Mixed** next to it.



Note Data transfer signals are unprotected in the generated code by default. You must prevent data corruption in these signals due to pre-emption in the target environment or implementing protection using custom storage classes.

Workflow for Export-Function Models

The most common workflow is to test function-call behavior through simulation and generate the functions using standalone code generation.

Standalone Simulation

When function-call sequencing is simple to be specified as a model input, standalone simulation is the preferred workflow. For a standalone simulation, create data sets for the function-call and data root-level Inport blocks. For more information on function-call inputs, see “Specifying Function-Call Inputs” on page 7-10.

You can also specify the execution order for function-call subsystems. For more information, see “Execution Order for Function-Call Root-level Inport Blocks” on page 7-7.

Specifying Function-Call Inputs

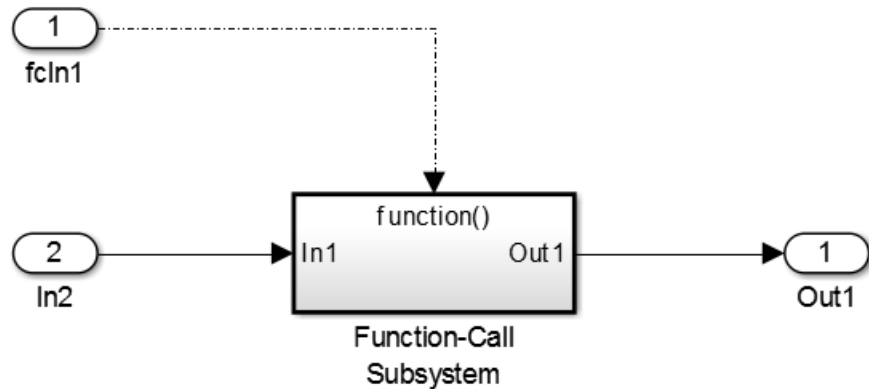
You can create data sets for the function-call and data root-level Inport blocks in **Simulation > Model Configuration Parameters > Data Import/Export > Input**.

For function-call inputs, specify a time-vector indicating when events occur.

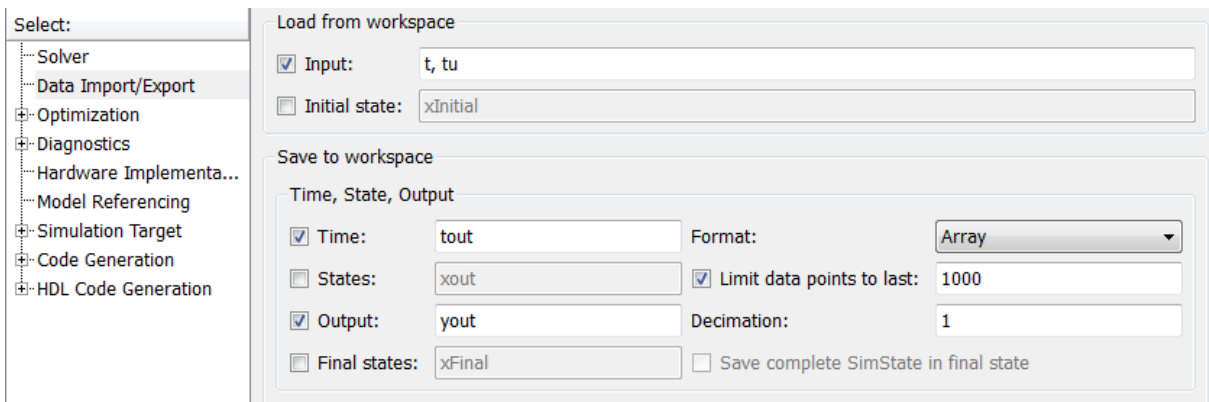
- The time vector must be of data type double and monotonically increasing.
- All time data must be integer multiples of the model sample time.
- To specify multiple function-calls at a given time step, you must repeat the time value accordingly. For example, to specify three events at $t = 1$ and two events at $t = 9$, then you must list 1 three times and 9 twice in your time vector i.e., $t = [1 \ 1 \ 1 \ 9 \ 9]'$.

- The table corresponding to normal data input port can be of any other supported format as described in “Enable Data Import” on page 48-88.

Follow the steps below to input data into a standalone export-function model via the **Model Configuration Parameters** dialog box.



- 1 Select **Simulation > Configuration Parameters > Data Import/Export**.
- 2 Select the **Input** parameter.
- 3 Enter the data as t, tu in the Data Import/Export pane:



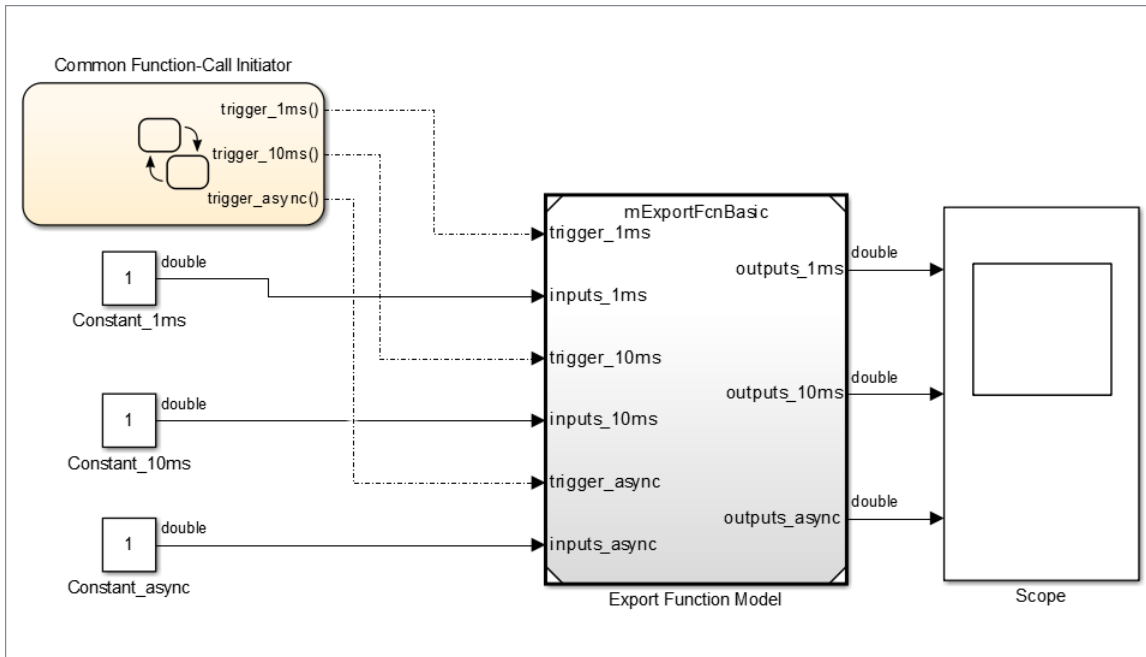
Here, t is a column vector containing the times of events for the Inport block **fcIn1** while tu is a table of input values versus time for Inport block **In2**.

If the function-call root-level Inport block has a discrete sample time specified, use an empty matrix ($[]$) for the time vector.

Top-Model Simulation Using Model Reference

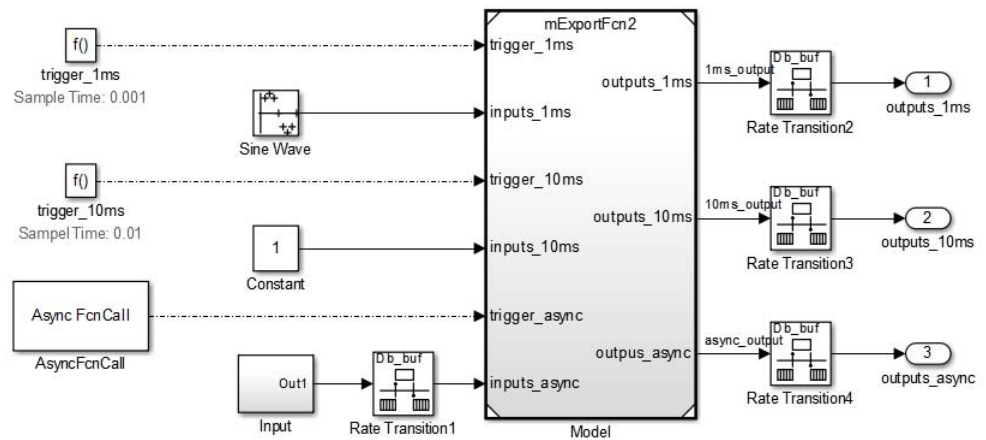
The more common simulation workflow of export-function models is by referencing export-function models. When function-call sequencing is too complicated to specify with data sets in a standalone simulation, create a harness top-model to mimic the behavior of the target environment, giving inputs to the exported functions. There are two forms to mimic behavior of the scheduling environment:

- Common function-call initiator, in which you fully control the scheduling process. Use Stateflow or S-functions to create arbitrary call sequences.



Note Simulink does not simulate preempting function-calls.

- Multiple function-call initiators having distinct sample times: You can use Simulink scheduling for simulation in this case, which is especially useful when Simulink's rate monotonic scheduling behavior is similar to the target OS behavior.



Standalone Code Generation

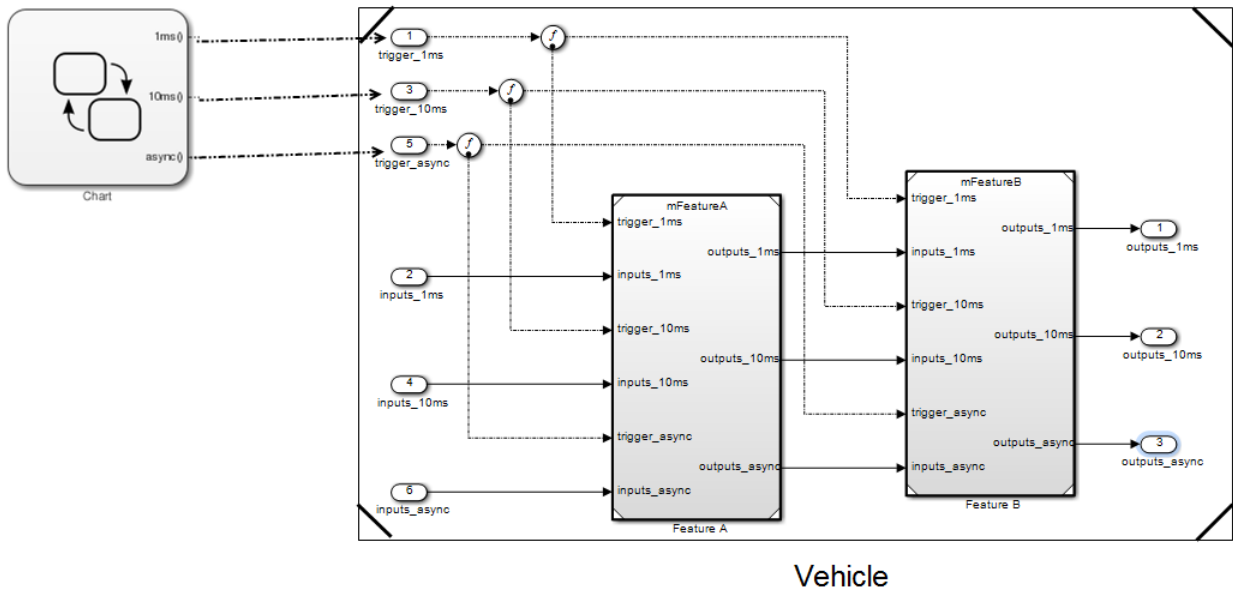
For standalone code generation, specify an ERT code-generation target, such as `ert.tlc`, and select **Code > C/C++ Code > Build Model >** to generate code. In the generated code, each function-call root Inport generates a void-void function. The function name for each function-call root Inport block is the name of the output function-call signal of the block. If there is no signal name, then the function name is derived from the name of the root Inport block. Building the model generates a model initialization function but does not generate a model step function or an enable/disable function.

To customize the model initialize function name for the referenced export-function model, open the top model and complete these:

- Select **Model Configuration Parameters > Code Generation > Interface**.
- Click **Configure Model Functions**.
- In the Model Interface dialog box, set **Function specification** to Model specific C prototypes and click **Validate**.
- Type the function name in the **Initialize function** name text box and click **Apply**.
- Generate code again to see the new function name.

Nested Export-Function Models

Nested export-function models provide an additional layer of organization for your model. The schematic below explains how the user may export functions at the vehicle level or the individual feature level.



Note An export-function model cannot contain a 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, function-call subsystem or a function-call model.

Comparison between Export-Function Models and Models with Asynchronous Function-Call Inputs

A similar feature to export-function models is available since R2011a for models with asynchronous function-call inputs. These models are used primarily in the Simulink environment, where generated functions are called by the Simulink scheduler. Since the generated asynchronous function-call entry-points will be called by Simulink scheduler, the model can only be used as a referenced model for code generation. In contrast, the workflow for export-function models involves code generation of functions in standalone models.

	Export-Function Models	Models with Asynchronous Function-Call Inputs
Definition	These models have function-call root-level Inport blocks that are not connected to an Asynchronous Task Specification block. These Inport blocks trigger function-call subsystems or referenced models with function-call trigger inputs.	These models have function-call root-level Inport blocks connected to an Asynchronous Task Specification block. These Inport blocks trigger function-call subsystems or referenced models with function-call trigger inputs.
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.

	Export-Function Models	Models with Asynchronous Function-Call Inputs
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 “Data Transfer Between Function-Call Subsystems” on page 7-8.	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 top-model simulation in all simulation modes.	These models support top-model simulation in all modes and standalone simulation in Normal, Accelerator, and Rapid Accelerator modes.
Code generation support	Top-model and standalone code generation is supported.	Top-model code generation is supported. Standalone code generation is not supported.

Create an Enabled Subsystem

In this section...

“What Are Enabled Subsystems?” on page 7-17

“Creating an Enabled Subsystem” on page 7-18

“Blocks that an Enabled Subsystem Can Contain” on page 7-24

“Using Blocks with Constant Sample Times in Enabled Subsystems” on page 7-28

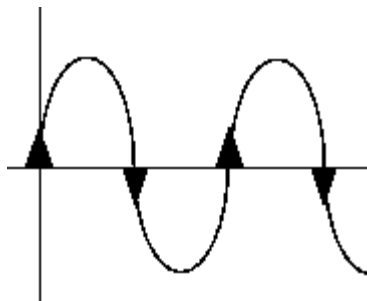
What Are Enabled Subsystems?

Enabled subsystems are subsystems that execute at each simulation step for which the control signal has a positive value.

An enabled subsystem has a single control input, which can be a scalar or a vector.

- If the input is a scalar, the subsystem executes if the input value is greater than zero.
- If the input is a vector, the subsystem executes if *any one* of the vector elements is greater than zero.

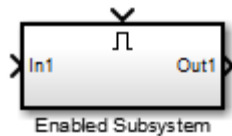
For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled. This behavior is shown in the following figure, where an up arrow signifies enable and a down arrow disable.



The Simulink software uses the zero-crossing slope method to determine whether an enable event is to occur. If the signal crosses zero and its slope is positive, then the subsystem becomes enabled. If the slope is negative at the zero crossing, then the subsystem becomes disabled. Note that a subsystem is only enabled or disabled at major time steps. Therefore, if zero-crossing detection is turned off and the signal crosses zero during a minor time step, then the subsystem will not become enabled (or disabled) until the next major time step.

Creating an Enabled Subsystem

You create an enabled subsystem by copying an Enable block from the Ports & Subsystems library into a Subsystem block. An enable symbol and an enable control input port is added to the Subsystem block.



Setting Initial Conditions for an Enabled Subsystem

You can set the initial output of an enabled subsystem using the subsystems Output blocks. The initial output value can be either explicitly specified, or inherited from its input signal.

Specifying Initial Conditions. To specify the initial output value of the subsystem:

- 1 Double-click each Output block in the subsystem to open its block parameters dialog box.
- 2 In the **Source of initial output value** drop-down list, select Dialog.
- 3 Specify a valid value for the **Initial output** parameter. Valid values include the empty matrix. But Inf and NaN are not valid values.

If you select **Dialog**, you can also specify what happens to the output when the subsystem is disabled. For more information, see the next section: “Setting Output Values While the Subsystem Is Disabled” on page 7-20.

Inheriting Initial Conditions. The Output block can inherit an initial output value for the subsystem from the following sources:

- Output port of another conditionally executed subsystem
- Merge block (with Initial output specified)
- Function-Call Model Reference block
- Constant block (simplified initialization mode only)
- IC block (simplified initialization mode only)

The procedure you use to inherit the initial conditions of the subsystem differs depending on whether you are using classic initialization mode or simplified initialization mode.

To inherit initial conditions in classic initialization mode:

- 1** Double-click each Output block in the subsystem to open its block parameters dialog box.
- 2** In the **Source of initial output value** drop-down list, select **Dialog**.
- 3** Set the **Initial output** parameter to [] (empty matrix).
- 4** Click **OK**.

Note For all other driving blocks, specify an explicit initial output value.

To inherit initial conditions in simplified initialization mode:

- 1** Double-click each Output block in the subsystem to open its block parameters dialog box.
- 2** In the **Source of initial output value** drop-down list, select **Input signal**.

3 Click **OK**.

The **Initial output** and **Output when disabled** parameters are disabled, and both values are inherited from the input signal.

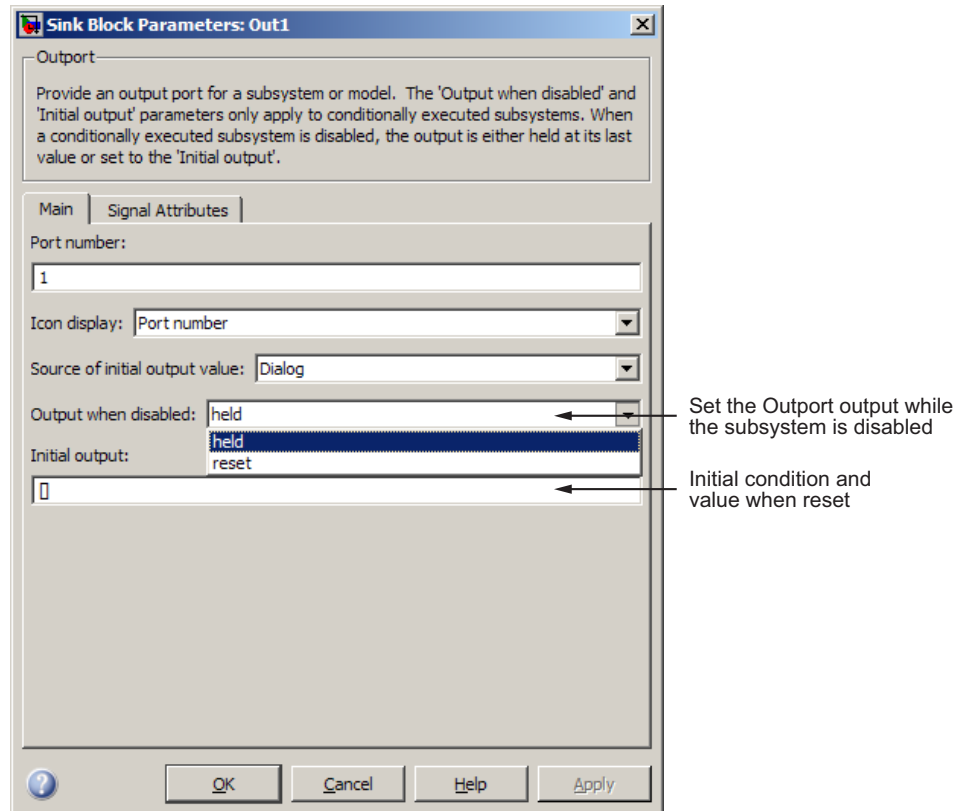
For more information on classic and simplified initialization mode, see “Underspecified initialization detection”.

Setting Output Values While the Subsystem Is Disabled

Although an enabled subsystem does not execute while it is disabled, the output signal is still available to other blocks. While an enabled subsystem is disabled, you can choose to hold the subsystem outputs at their previous values or reset them to their initial conditions.

Open the block parameters dialog box for each Output block and for the **Output when disabled** parameter, select one of the choices.

- Choose **held** to maintain the most recent value.
- Choose **reset** to revert to the initial condition. Set the **Initial output** to the initial value of the output.



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.

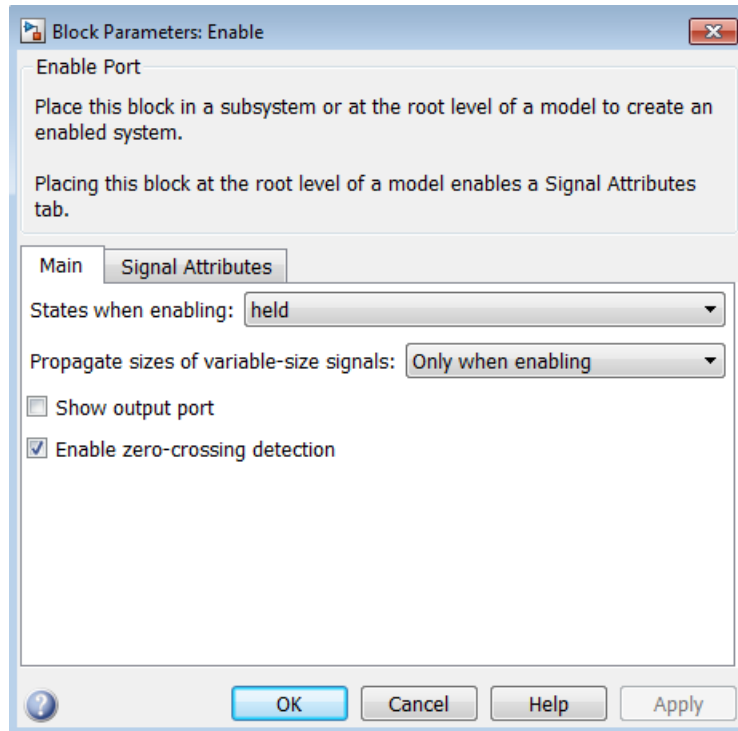
If you are using simplified initialization mode, you must select **held** when connecting a conditionally executed subsystem to a Merge block. For more information, see “Underspecified initialization detection”.

Setting States When the Subsystem Becomes Enabled

When an enabled subsystem executes, you can choose whether to hold the subsystem states at their previous values or reset them to their initial conditions.

To do this, open the **Block Parameters: Enable** dialog box and select one of the choices for the **States when enabling** parameter:

- Choose **held** to cause the states to maintain their most recent values.
- Choose **reset** to cause the states to revert to their initial conditions.



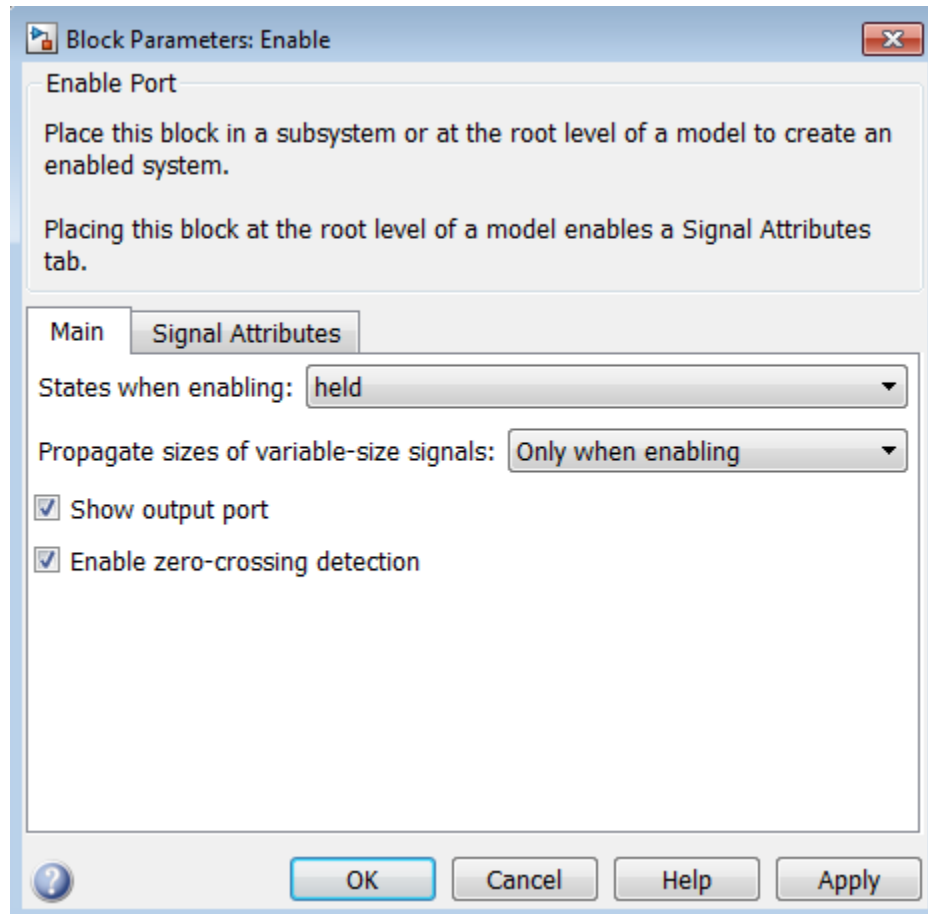
Note If you are using simplified initialization mode, the subsystem elapsed time is always reset during the first execution after becoming enabled, whether or not the subsystem is configured to reset on enable.

For more information on simplified initialization mode, see “Underspecified initialization detection”.

Note For nested subsystems whose Enable blocks have different parameter settings, the settings on the child subsystem’s dialog box override those inherited from the parent subsystem.

Outputting the Enable Control Signal

An option on the Enable block dialog box lets you output the enable control signal. To output the control signal, select the **Show output port** check box.



This feature allows you to pass the control signal down into the enabled subsystem, which can be useful where logic within the enabled subsystem is dependent on the value or values contained in the control signal.

Blocks that an Enabled Subsystem Can Contain

An enabled subsystem can contain any block, whether continuous or discrete. 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. Enabled subsystems and the model use a common clock.

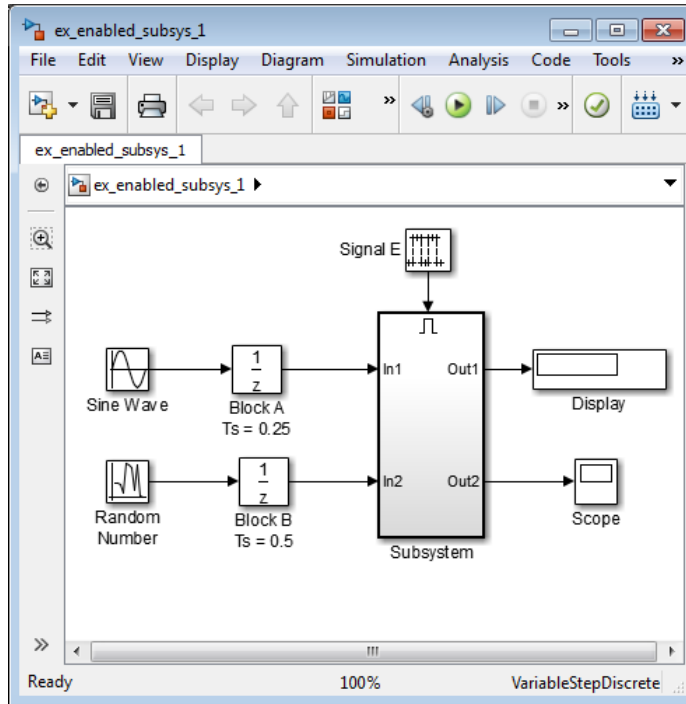
Note Enabled subsystems can contain Goto blocks. However, only state ports can connect to Goto blocks in an enabled subsystem. In the `sldemo_clutch` model, see the Locked subsystem for an example of how to use Goto blocks in an enabled subsystem.

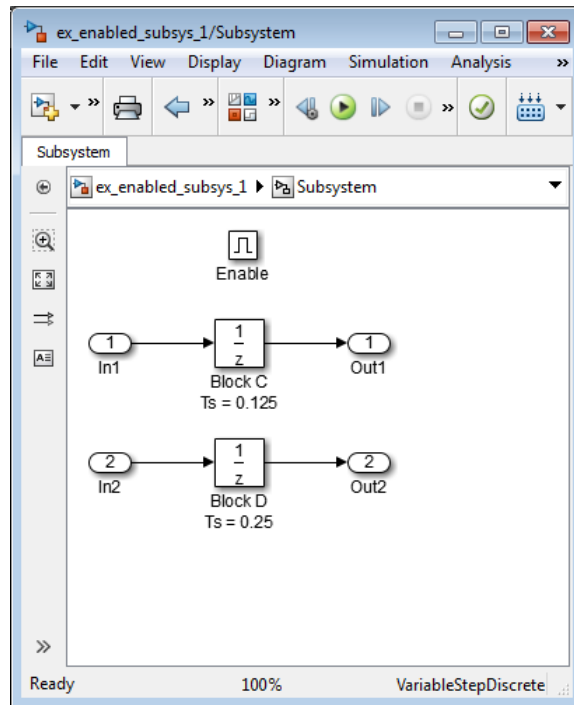
For example, this system contains four discrete blocks and a control signal. The discrete blocks are:

- Block A, which has a sample time of 0.25 second
- Block B, which has a sample time of 0.5 second
- Block C, within the enabled subsystem, which has a sample time of 0.125 second
- Block D, also within the enabled subsystem, which has a sample time of 0.25 second

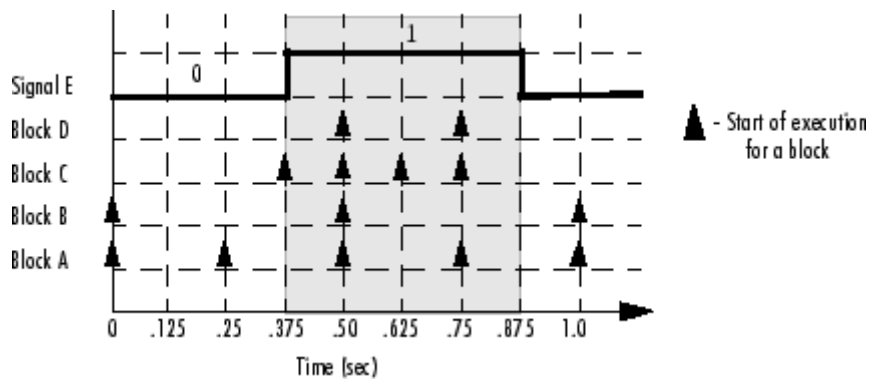
The enable control signal is generated by a Pulse Generator block, labeled Signal E, which changes from 0 to 1 at 0.375 second and returns to 0 at 0.875 second.

7 Create Conditional Subsystems





The chart below indicates when the discrete blocks execute.



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. Note that block C does not execute at 0.875 second when the enable control signal changes to zero.

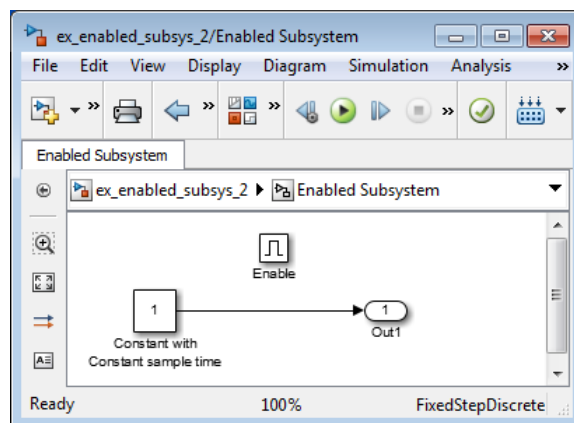
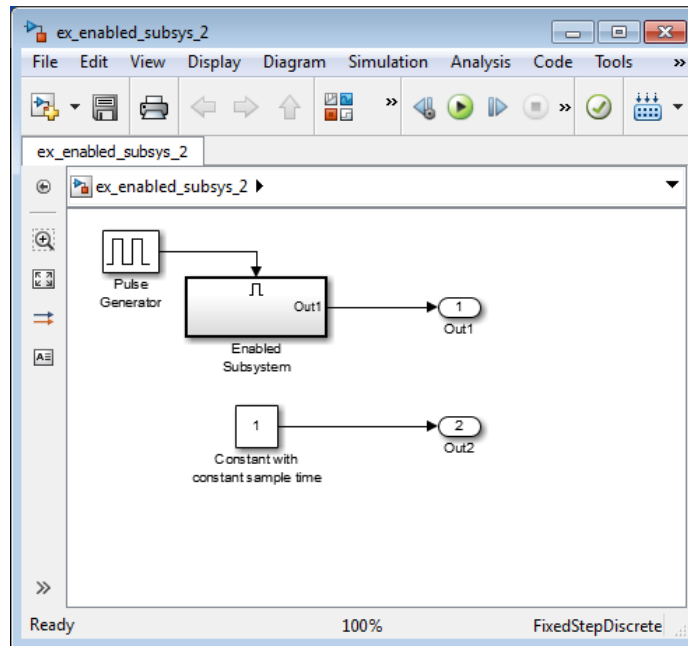
Using Blocks with Constant Sample Times in Enabled Subsystems

Certain restrictions apply when you connect blocks with constant sample times (see “Constant Sample Time” on page 5-17) to the output port of a conditional subsystem.

- An error appears when you connect a Model or S-Function block with constant sample time to the output port of a conditional subsystem.
- The sample time of any built-in block with a constant sample time is converted to a different sample time, such as the fastest discrete rate in the conditional subsystem.

To avoid the error or conversion, either manually change the sample time of the block to a non-constant sample time or use a Signal Conversion block. The example below shows how to use the Signal Conversion block to avoid these errors.

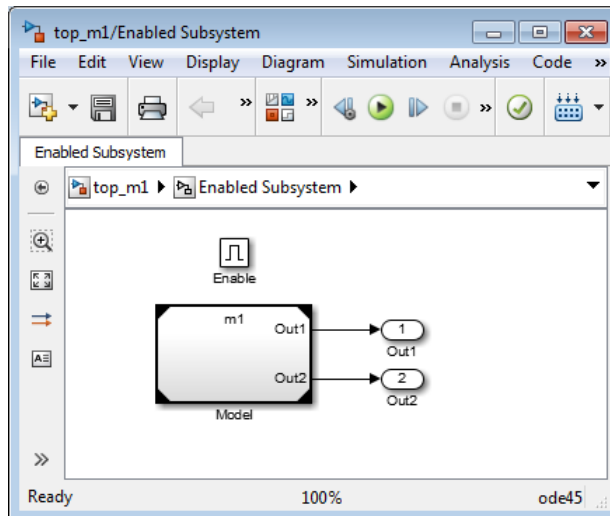
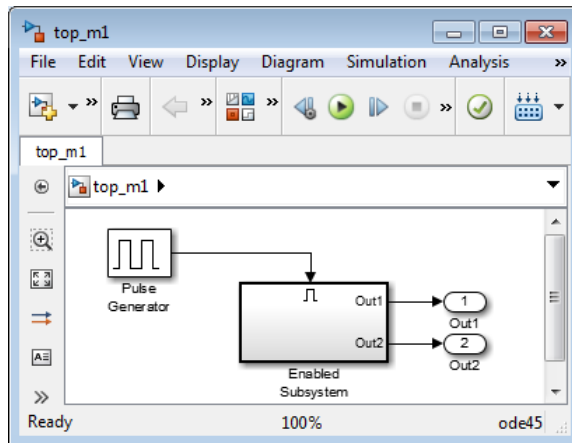
Consider the following model `ex_enabled_subsys_2.mdl`.



The two Constant blocks in this model have constant sample times. When you simulate the model, the Simulink software converts the sample time of the Constant block inside the enabled subsystem to the rate of the Pulse Generator. If you simulate the model with sample time colors displayed

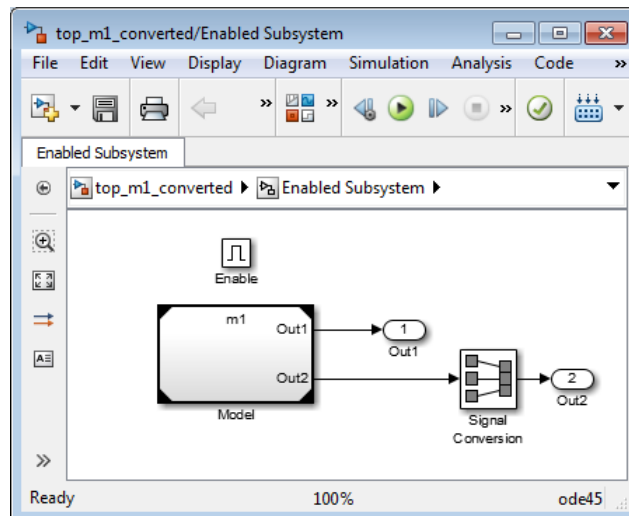
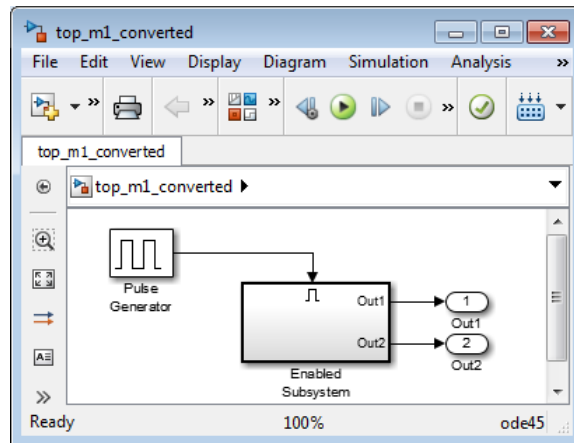
(see “View Sample Time Information” on page 5-9), the Pulse Generator and Enabled Subsystem blocks are colored red. However, the Constant and Output blocks outside of the enabled subsystem are colored magenta, indicating that these blocks still have a constant sample time.

Suppose the model above is referenced from a Model block inside an enabled subsystem, as shown below. (See “Model Reference”.)



An error appears when you try to simulate the top model, indicating that the second output of the Model block may not be wired directly to the enabled subsystem output port because it has a constant sample time. (See “Model Reference”.)

To avoid this error, insert a Signal Conversion block between the second output of the Model block and the Output block of the enabled subsystem.



This model simulates with no errors. With sample time colors displayed, the Model and Enabled Subsystem blocks are colored yellow, indicating that these are hybrid systems. In this case, the systems are hybrid because they contain multiple sample times.

Create a Triggered Subsystem

In this section...

“What Are Triggered Subsystems?” on page 7-33

“Using Model Referencing Instead of a Triggered Subsystem” on page 7-35

“Creating a Triggered Subsystem” on page 7-35

“Blocks That a Triggered Subsystem Can Contain” on page 7-36

What Are Triggered Subsystems?

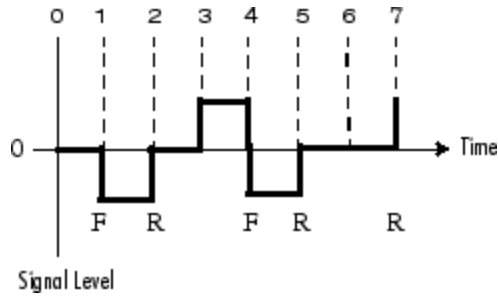
Triggered subsystems are subsystems that execute each time a trigger event occurs.

A triggered subsystem has a single control input, called the *trigger input*, that determines whether the subsystem executes. You can choose from three types of trigger events to force a triggered subsystem to begin execution:

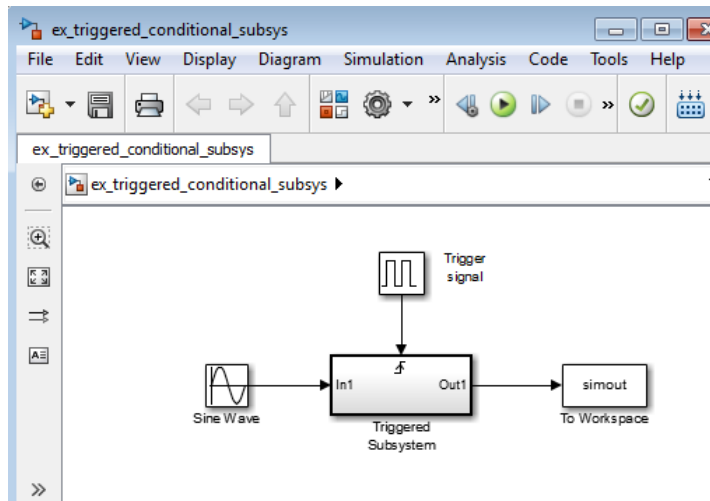
- **rising** triggers execution of the subsystem when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative).
- **falling** triggers execution of the subsystem when the control signal falls from a positive or a zero value to a negative value (or zero if the initial value is positive).
- **either** triggers execution of the subsystem when the signal is either rising or falling.

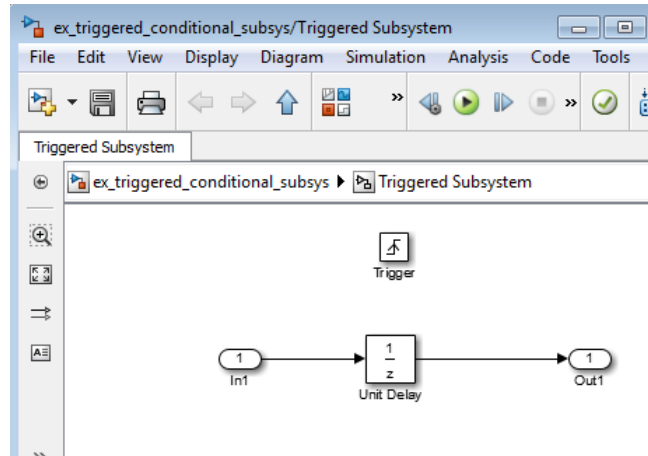
Note For discrete systems: If after rising, a signal remains at zero for more than one time step, then and only then, does the subsequent rising of the signal constitute a trigger event. Similarly, a falling trigger event occurs only if there are at least two time steps between two occurrences of the signal falling. This trigger event scheme eliminates false triggers caused by control signal sampling.

For example, in the following timing diagram for a discrete system, a rising trigger (R) does not occur at time step 3 because the signal remains at zero for only one time step prior to the rise.



A simple example of a triggered subsystem is illustrated.





In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Using Model Referencing Instead of a Triggered Subsystem

You can use triggered ports in referenced models. Add a trigger port to a referenced model to create a simpler, cleaner model than when you include either:

- A triggered subsystem in a referenced model
- A Model block in a triggered subsystem

For information about using trigger ports in referenced models, see “Conditional Referenced Models” on page 6-61.

To convert a subsystem to use model referencing, see “Convert a Subsystem to a Referenced Model” on page 6-12.

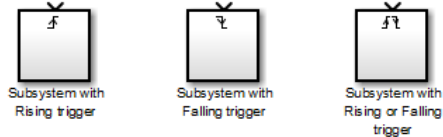
Creating a Triggered Subsystem

You create a triggered subsystem by copying the Trigger block from the Ports & Subsystems library into a subsystem. The Simulink software adds a trigger symbol and a trigger control input port to the Subsystem block.



To select the trigger type, open the Trigger block dialog box and select one of the choices for the **Trigger type** parameter.

Different symbols appear on the Trigger and Subsystem blocks to indicate rising and falling triggers (or either). This figure shows the trigger symbols on Subsystem blocks.



Outputs and States Between Trigger Events

Unlike enabled subsystems, triggered subsystems always hold their outputs at the last value between triggering events. Also, triggered subsystems cannot reset their states when triggered; the states of any discrete block is held between trigger events.

Outputting the Trigger Control Signal

An option on the Trigger block dialog box lets you output the trigger control signal. To output the control signal, select the **Show output port** check box.

In the **Output data type** field, specify the data type of the output signal as `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 port to which the signal connects.

Blocks That a Triggered Subsystem Can Contain

All blocks in a triggered subsystem must have either inherited (`-1`) or constant (`inf`) sample time. This is to indicate that the blocks in the triggered subsystem run only when the triggered subsystem itself runs, for example,

when it is triggered. This requirement means that a triggered subsystem cannot contain continuous blocks, such as the Integrator block.

Create an Action Subsystem

In this section...
“What Are Action Subsystems?” on page 7-38
“Set States when an Action Subsystem Executes” on page 7-40

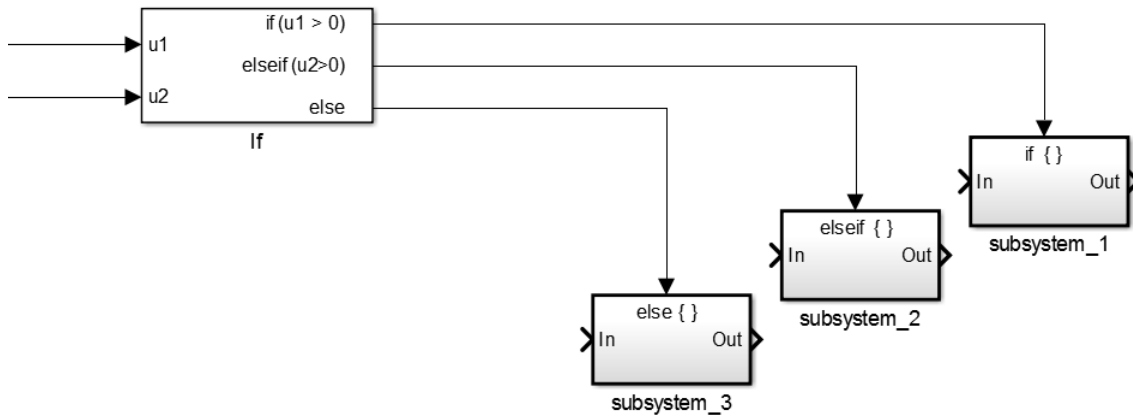
What Are Action Subsystems?

Action subsystems are subsystems that execute in response to a conditional output from an If block or a Switch Case block. In essence, they are subsystems with an Action port, which allow for block execution based on conditional inputs from an If block or Switch Case block.

Simulink has two types of action subsystems, based on the type of block they receive conditional input from.

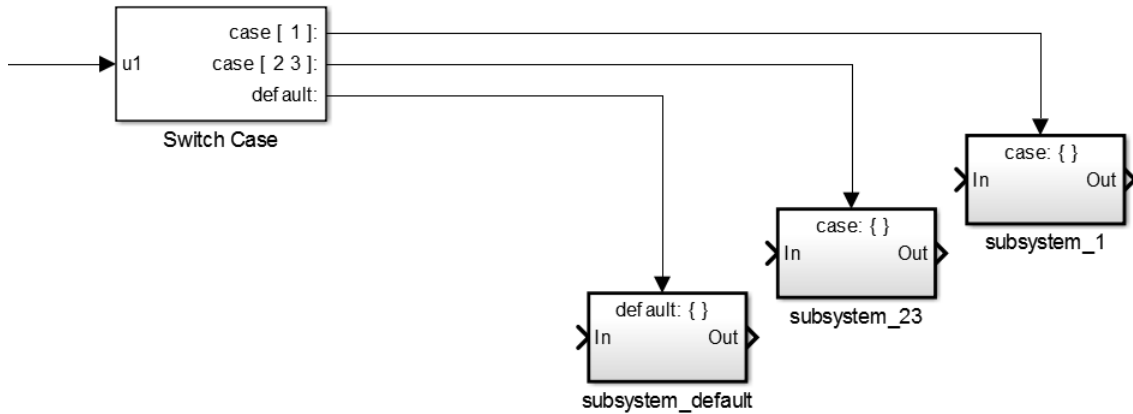
If Action Subsystem

The If Action Subsystem is preconfigured to serve as a starting point for creating a subsystem whose execution is triggered by an If block. To implement an if-else condition, connect If action subsystem blocks to the outputs of an If block.



Switch Case Action Subsystem

The Switch Case Action Subsystem is preconfigured to serve as a starting point for creating a subsystem whose execution is triggered by a Switch Case block. To implement a switch condition, connect Switch Case Action Subsystem blocks to the outputs of a Switch Case block.

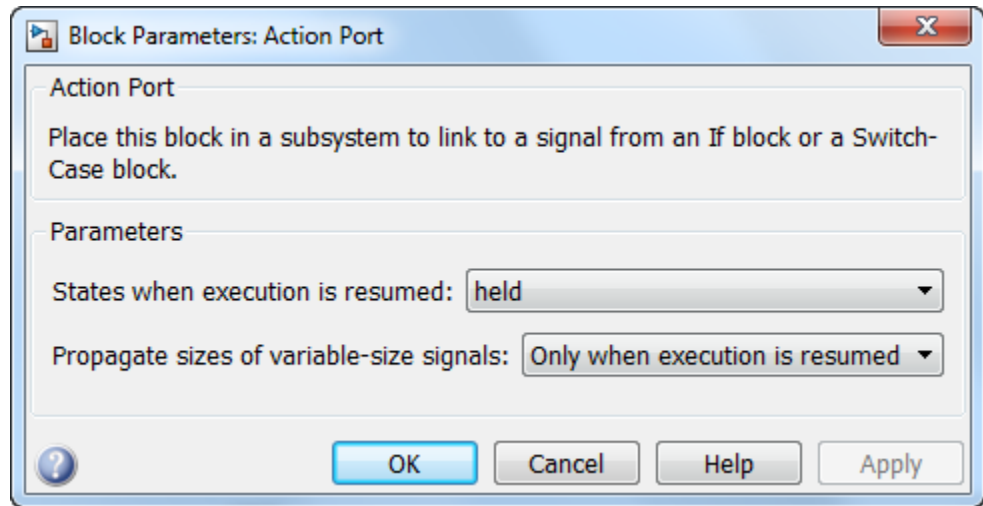


Note All blocks in an action subsystem must run at the same sample time as the If or Switch Case block that triggers its execution. You can achieve this automatically by setting the **Sample time** parameter of each block in the subsystem to -1 (inherited).

Set States when an Action Subsystem Executes

When an action subsystem is triggered to execute, you can choose whether to hold the subsystem states at their previous values or reset them to their initial conditions.

- 1 Open the Action Port block inside the action subsystem.
- 2 Select one of the following for the **States when execution is resumed** parameter:
 - **held** if you want the states to maintain their most recent values
 - **reset** if you want the states to revert to their initial conditions



Note For nested subsystems whose Action Port blocks have different parameter settings, the settings on the child subsystem’s dialog box override those inherited from the parent subsystem.

For more information, see Action Port, If, and Switch Case.

Create a Triggered and Enabled Subsystem

In this section...

“What Are Triggered and Enabled Subsystems?” on page 7-42

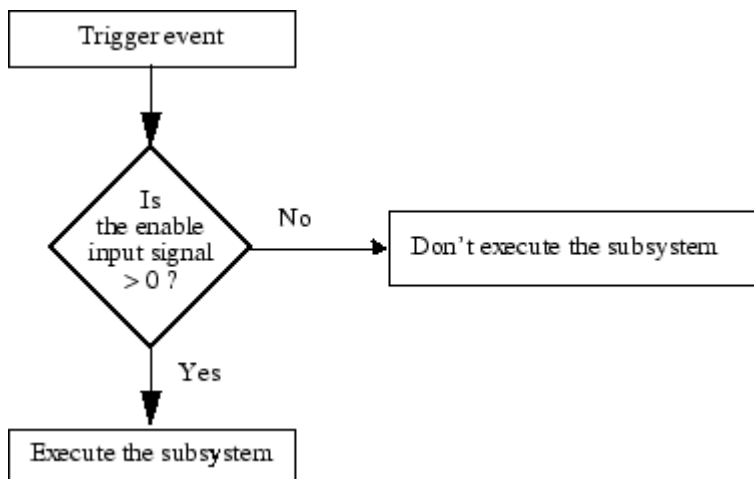
“Creating a Triggered and Enabled Subsystem” on page 7-43

“A Sample Triggered and Enabled Subsystem” on page 7-44

“Creating Alternately Executing Subsystems” on page 7-45

What Are Triggered and Enabled Subsystems?

A third kind of conditional subsystem combines two types of conditional execution. The behavior of this type of subsystem, called a *triggered and enabled* subsystem, is a combination of the enabled subsystem and the triggered subsystem, as shown by this flow diagram.



A triggered and enabled subsystem contains both an enable input port and a trigger input port. When the trigger event occurs, the enable input port is checked to evaluate the enable control signal. If its value is greater than zero, the subsystem is executed. If both inputs are vectors, the subsystem executes if at least one element of each vector is nonzero.

The subsystem executes once at the time step at which the trigger event occurs.

Creating a Triggered and Enabled Subsystem

You create a triggered and enabled subsystem by dragging both the Enable and Trigger blocks from the Ports & Subsystems library into an existing subsystem. The Simulink software adds enable and trigger symbols and enable and trigger control inputs to the Subsystem block.

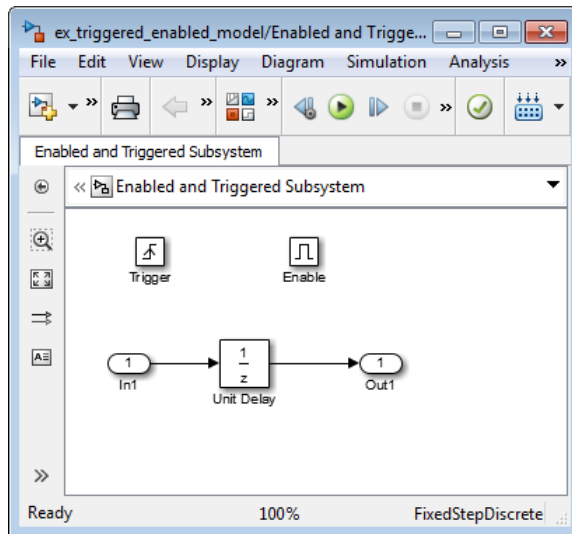
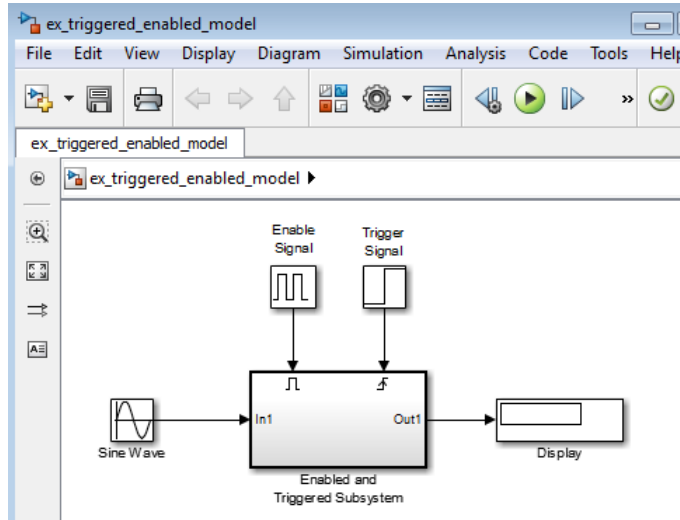


You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see “Setting Output Values While the Subsystem Is Disabled” on page 7-20. Also, you can specify what the values of the states are when the subsystem is reenabled. See “Setting States When the Subsystem Becomes Enabled” on page 7-22.

Set the parameters for the Enable and Trigger blocks separately. The procedures are the same as those described for the individual blocks.

A Sample Triggered and Enabled Subsystem

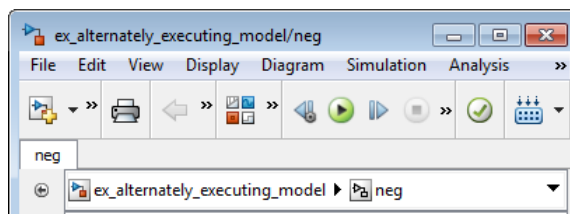
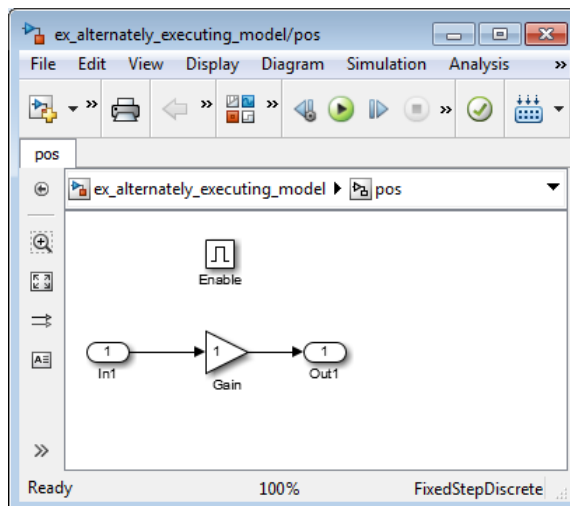
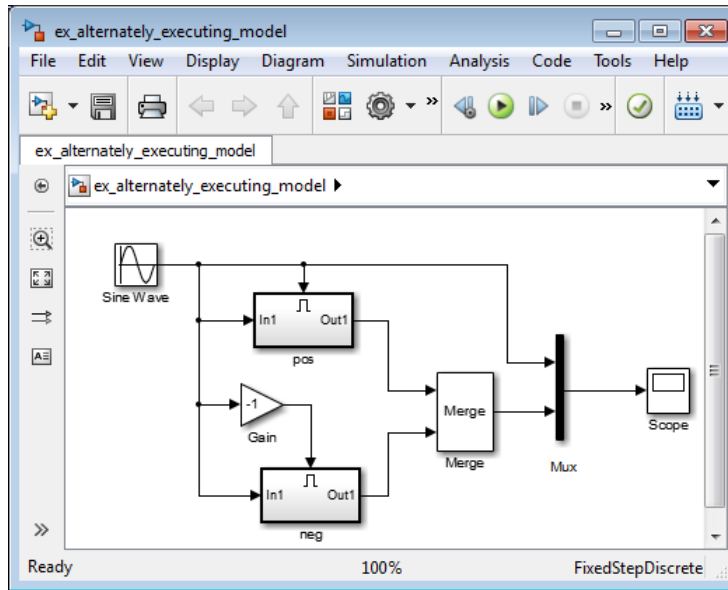
A simple example of a triggered and enabled subsystem is illustrated in the following model.



Creating Alternately Executing Subsystems

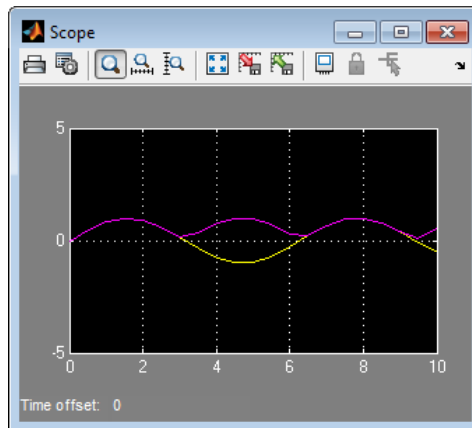
You can use conditional subsystems in combination with Merge blocks to create sets of subsystems that execute alternately, depending on the current state of the model.

The following figure shows a model that uses two enabled blocks and a Merge block to model a full-wave rectifier – a device that converts AC current to pulsating DC current.



The block labeled `pos` is enabled when the AC waveform is positive; it passes the waveform unchanged to its output. The block labeled `neg` is enabled when the waveform is negative; it inverts the waveform. The Merge block passes the output of the currently enabled block to the Mux block, which passes the output, along with the original waveform, to the Scope block.

The Scope creates the following display.



Create a Function-Call Subsystem

What is a Function-Call Subsystem?

A function-call subsystem is a subsystem that another block can invoke directly during a simulation. It is analogous to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods (See “Block Methods” on page 3-16) of the blocks that the subsystem contains in sorted order (See “How Simulink Determines the Sorted Order” on page 23-47). The block that invokes a function-call subsystem is called the function-call initiator. Stateflow, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

Creating Function-Call Subsystems

To create a function-call subsystem, drag a Function-Call Subsystem block from the Ports & Subsystems library into your model and connect a function-call initiator to the function-call port displayed on top of the subsystem. You can also create a function-call subsystem from scratch. First create a Subsystem block in your model and then create a Trigger block in the subsystem. Next, on the Trigger block parameters pane, set the **Trigger type** to function-call.

Sample Time Propagation in Function-Call Subsystems

You can configure a function-call subsystem to be triggered (the default) or periodic by setting the **Sample time type** of its Trigger port to be **triggered** or **periodic**, respectively. A function-call initiator can invoke a triggered function-call subsystem zero, one, or multiple times per time step. The sample times of all the blocks in a triggered function-call subsystem must be set to inherited (-1).

A function-call initiator can invoke a periodic function-call subsystem only once per time step and must invoke the subsystem periodically. If the initiator invokes a periodic function-call subsystem aperiodically, the simulation is halted and an error message is displayed. The blocks in a periodic function-call subsystem can specify a noninherited sample time or inherited (-1) sample time. All blocks that specify a noninherited sample time

must specify the same sample time. For example, if one block specifies 0.1 as the sample time, all other blocks must specify a sample time of 0.1 or -1. If a function-call initiator invokes a periodic function-call subsystem at a rate that differs from the sample time specified by the blocks in the subsystem, the simulation halts and an error message appears.

Note During range checking, the design minimum and maximum 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, add a Signal Conversion block and a Signal Specification block after the source port, set the **Output** of the Signal Conversion block to **Signal copy**, and specify the design minimum and maximum on the Signal Specification block instead of specifying them on the source port.

For more information, see in “Writing S-Functions” in the online documentation.

Conditional Execution Behavior

In this section...

“What Is Conditional Execution Behavior?” on page 7-50

“Propagating Execution Contexts” on page 7-52

“Behavior of Switch Blocks” on page 7-53

“Displaying Execution Contexts” on page 7-54

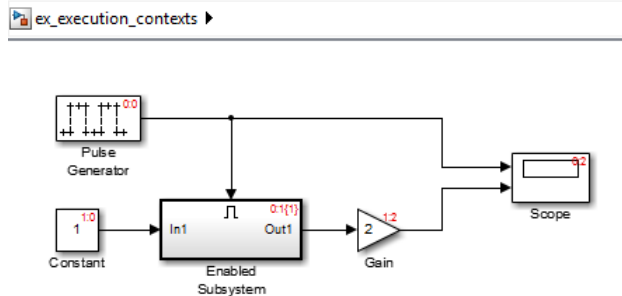
“Disabling Conditional Execution Behavior” on page 7-55

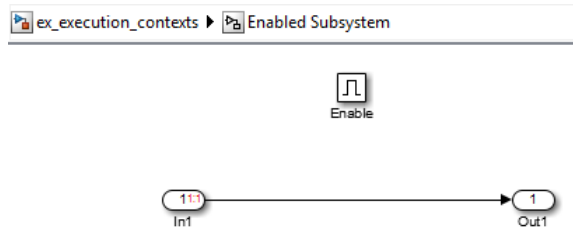
“Displaying Execution Context Bars” on page 7-55

What Is Conditional Execution Behavior?

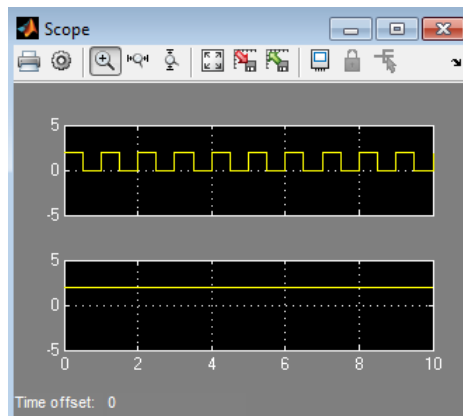
To speed the simulation of a model, by default the Simulink software avoids unnecessary execution of blocks connected to Switch, Multiport Switch, and of conditionally executed blocks. This behavior is *conditional execution (CE)* behavior. You can disable this behavior for all Switch and Multiport Switch blocks in a model, or for specific conditional subsystems. See “Disabling Conditional Execution Behavior” on page 7-55.

The following model illustrates conditional execution behavior.





The Scope block shows the simulation result:



This model:

- Has the **Display > Signals & Ports > Execution Context Indicator** menu option enabled.
- The Pulse Generator block has the following parameter settings:
 - **Pulse type** — Sample based
 - **Period** — 100
 - **Pulse width** — 50
 - **Phase delay** — 50
 - **Sample Time** — 0.01

- The Gain block's sorted order (1:2) is second (2) in the enabled subsystem's execution context (1).
- The Enabled Subsystem block has the **Propagate execution context across subsystem boundary** parameter enabled.
- In the enabled subsystem, the Out1 block has the following parameter settings:
 - **Initial output** — []
 - **Output when disabled** — held

The outputs of the Constant block and Gain blocks are computed only while the enabled subsystem is enabled (for example, at time steps 0.5 to 1.0 and 1.0 to 1.5). This behavior is necessary because the output of the Constant block is required and the input of the Gain block changes only while the enabled subsystem is enabled. When CE behavior is off, the outputs of the Constant and Gain blocks are computed at every time step, regardless of whether the outputs are needed or change.

In this example, the enabled subsystem is regarded as defining an execution context for the Constant and Gain blocks. Although the blocks reside graphically in the root system of the model, the Simulink software invokes the block methods during simulation as if the blocks reside in the enabled subsystem. This is indicated in the sorted order labels displayed on the diagram for the Constant and Gain blocks. The notations list the subsystem's (id = 1) as the execution context for the blocks even though the blocks exist graphically at the root level (id = 0) of the model. The Gain block's sorted order (1:2) is second (2) in the enabled subsystem's execution context (1).

Propagating Execution Contexts

In general, the Simulink software defines an *execution context* as a set of blocks to be executed as a unit. At model compilation time, the Simulink software associates an execution context with the model's root system and with each of its nonvirtual subsystems. Initially, the execution context of the root system and each nonvirtual subsystem is simply the blocks that it contains.

When compiling, each block in the model is examined to determine whether it meets the following conditions:

- The block output is required only by a conditional subsystem or the block input changes only as a result of the execution of a conditionally executed subsystem.
- The execution context of the subsystem can propagate across the subsystem boundaries.
- The output of the block is not a test point (see “Test Points” on page 50-59).
- The block is allowed to inherit its conditional execution context.

The Simulink software does not allow some built-in blocks, such as the Delay block, ever to inherit their execution context. Also, S-Function blocks can inherit their execution context only if they specify the `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` option.

- The block is not a multirate block.
- The block sample time is set to inherited (-1).

If a block meets these conditions and execution context propagation is enabled for the associated conditional subsystem (see “Disabling Conditional Execution Behavior” on page 7-55), the Simulink software moves the block into the execution context of the subsystem. This ensures that the block methods execute during the simulation loop only when the corresponding conditional subsystem executes.

Note Execution contexts are not propagated to blocks having a constant sample time.

Behavior of Switch Blocks

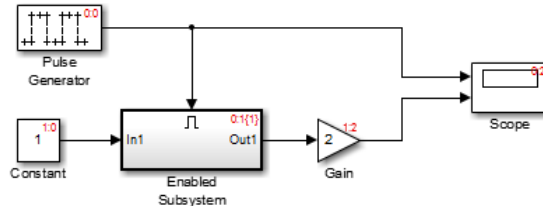
This behavior treats the input branches of a Switch or Multiport Switch block as invisible, conditional subsystems, each of which has its own execution context. This CE is enabled only when the control input of the switch selects the corresponding data input. As a result, switch branches execute only when selected by switch control inputs.

Displaying Execution Contexts

To determine the execution context to which a block belongs, in the Simulink Editor, select **Display > Blocks > Sorted Execution Order**. The sorted order index for each block in the model is displayed in the upper-right corner of the block. The index has the format $s:b$, where s specifies the subsystem to whose execution context the block belongs and b is an index that indicates the block sorted order in the execution context of the subsystem. For example, 0:0 indicates that the block is the first block in the execution context of the root subsystem.

If a bus is connected to a block input, the block sorted order is displayed as $s:B$. For example, 0:B indicates that the block belongs to the execution context of the root system and has a bus connected to its input.

The sorted order index of conditional subsystems is expanded to include the system ID of the subsystem itself in curly brackets as illustrated in the following figure.



In this example, the sorted order index of the enabled subsystem is $0:1\{1\}$. The 0 indicates that the enabled subsystem resides in the root system of the model. The first 1 indicates that the enabled subsystem is the second block on the sorted list of the root system (zero-based indexing). The 1 in curly brackets indicates that the system index of the enabled subsystem itself is 1. Thus any block whose system index is 1 belongs to the execution context of the enabled subsystem and hence executes when it does. For example, the fact that the Constant block has an index of $1:0$ indicates that it is the first block on the sorted list of the enabled subsystem, even though it resides in the root system.

Disabling Conditional Execution Behavior

To disable conditional execution behavior for all Switch and Multiport Switch blocks in a model, turn off the **Conditional input branch execution** optimization on the **Optimization** pane of the Configuration Parameters dialog box (see “Optimization Pane: General”). To disable conditional execution behavior for a specific conditional subsystem, clear the **Propagate execution context across subsystem boundary** check box on the subsystem parameter dialog box.

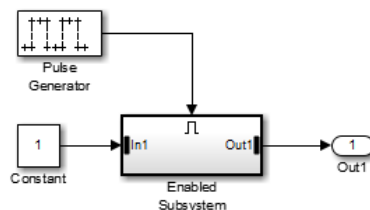
Even if this option is enabled, the execution context of the subsystem cannot propagate across its boundaries under either of the following circumstances:

- The subsystem is a triggered subsystem with a latched input port.
- The subsystem has one or more output ports that specify an initial condition other than []. In this case, a block connected to the subsystem output cannot inherit the execution context of the subsystem.

Displaying Execution Context Bars

Simulink can optionally display bars next to the ports of subsystems across which execution contexts cannot propagate. To display the bars, select **Display > Signals & Ports > Execution Context Indicator**.

For example, it displays bars on subsystems from which no block can inherit its execution context. In the following figure, the context bars appear next to the In1 and Out1 ports of the Enabled Subsystem block.



Conditional Subsystem Output Initialization

In this section...

“Why Initialize Conditional Subsystem Output with Explicit Values?” on page 7-56

“Initialization Mode” on page 7-56

“When to Use Simplified Initialization” on page 7-57

“Simplified Mode Behavior and Requirements” on page 7-58

“When to Use Classic Initialization” on page 7-59

Why Initialize Conditional Subsystem Output with Explicit Values?

By default, Simulink uses values from the inputs to a conditional subsystem during simulation. Alternatively, you can explicitly set the initial values, which can be useful for:

- Specifying initial behaviors that meet your modeling goals, such as testing a model
- Setting initial values that reduce simulation time by reaching steady state faster
- Making the model behavior easier to understand without having to trace input signals to determine the initial values

For details, see “Specify or Inherit Conditional Subsystem Initial Values” on page 7-61.

Initialization Mode

Initialization mode controls how Simulink handles the initialization values for conditionally executed subsystems.

The initialization mode also specifies how Simulink handles the initial values for Merge blocks, subsystem elapsed time, and Discrete-Time Integrator blocks. Usually the main impact of the initialization mode for Merge blocks

and Discrete-Time Integrator blocks is in the context of those blocks being used with a conditional subsystem.

The default initialization mode is **Classic**. **Simplified** mode uses enhanced processing, which can improve consistency of simulation results. For examples of some of the issues that simplified mode addresses, see “Address Classic Mode Issues by Using Simplified Mode” on page 7-65.

Note Classic mode was the only way that Simulink handled initial conditions prior to R2008b.

To help you choose which initialization mode to use, see:

- “When to Use Simplified Initialization” on page 7-57
- “Simplified Mode Behavior and Requirements” on page 7-58
- “When to Use Classic Initialization” on page 7-59

For details about setting initialization mode, see “Set Initialization Mode to Simplified or Classic” on page 7-63.

When to Use Simplified Initialization

In general, using simplified initialization mode helps you 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.

Use simplified initialization mode for models that contain one or more of the following blocks:

- Conditional subsystem
- Merge block
- 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.

Determine whether using simplified mode meets your modeling requirements. For details, see “Simplified Mode Behavior and Requirements” on page 7-58.

Simplified Mode Behavior and Requirements

Simplified mode affects the output of the conditional subsystem and the behavior of some blocks. Also, simplified mode has some model configuration requirements.

Output

- The output signal of a conditional subsystem is stored in separate memory that is not shared by with any other Outputport block. The initialization behavior of the memory is fully specified at the start of simulation.
- Conditional subsystem output ports can either explicitly specify an initial condition value or inherit that value from the block within the subsystem that is connected to the Outputport block input port.
- Simplified mode always uses the initial value as both the initial and reset value for output for a Discrete-Time Integrator block.

Outputport Blocks

Simplified mode has the following behavior with Outputport blocks:

- If the **Source of initial output value** parameter is set to `Input signal`, in simplified mode Simulink assumes the **Initial output** value is derived from the input signal. This can result in initialization behavior that is different from classic mode. However, the initialization behavior in simplified mode is generally more robust.
- The Outputport block can inherit the initial output value from a limited number of blocks. If it inherits from outside this limited list of blocks, then the Outputport block uses the default initial value of the output data

type. For details, see “Specify or Inherit Conditional Subsystem Initial Values” on page 7-61.

You cannot use a `Simulink.Signal` object to specify the **Initial output** value.

Merge Blocks

- If a root Merge block has an empty matrix (`[]`) for its initial output value, Simulink uses the default ground value of the output data type. A root Merge block is any Merge block with an output port that does not connect to another Merge block.
- You cannot use single-input Merge blocks.

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 When Using 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 need to make so that the library block works with simplified mode.

When to Use Classic Initialization

Simplified initialization mode offers many benefits, compared to classic mode. However, for an existing model, 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. See “Simplified Mode Behavior and Requirements” on page 7-58.

- 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 7-64.

Specify or Inherit Conditional Subsystem Initial Values

In this section...
“Inherit Initial Values from the Input Signal” on page 7-61
“Explicitly Specify an Initial Value” on page 7-62

To initialize the input values of a conditional subsystem, initialize its Outputport block, using one of these approaches:

- Explicitly specify the initial values by listing them in the Block Parameters dialog box of the Outputport block.
- Inherit the initial values from the input signals.

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’s Outputport block.

Inherit Initial Values from the Input Signal

Valid sources for the Outputport block to inherit its initial output value from are:

- Output port of another conditionally executed subsystem
- Merge block
- Function-Call Model Reference block
- Constant block (simplified initialization mode only)
- IC block (simplified initialization mode only)
- Stateflow Chart

How you configure the Outputport block for inheriting initial values depends on whether the model uses simplified or classic initialization mode. For details about initialization modes, see “Set Initialization Mode to Simplified or Classic” on page 7-63.

- If you use simplified initialization mode, either set the **Source of initial output value** to Input signal or do the following:

- 1 Set the **Source of initial output value** parameter to Dialog.
- 2 In the **Initial output** parameter, enter an empty matrix ([]).
- 3 Set **Output when disabled** parameter to held.

If an initial condition source is not a valid source for inheriting an initial value, the Outport block uses the default initial value of the output data type. For more information, see “Initialize Signals and Discrete States” on page 50-52.

- If you use classic initialization mode:
 - 1 Set the **Source of initial output value** parameter to Dialog.
 - 2 In the **Initial output** parameter, enter an empty matrix ([]).
 - 3 Set **Output when disabled** parameter to held.

Explicitly Specify an Initial Value

- 1 For the **Source of initial output value** parameter, select Dialog.
- 2 In the **Initial output** parameter, enter the initial value.

Note In simplified initialization mode, do not specify an empty matrix ([]) or a Simulink.Signal object.

Set Initialization Mode to Simplified or Classic

- 1 Determine which initialization mode to use.

In most situations, using simplified mode offers many benefits. To determine which mode to use, see:

- “When to Use Simplified Initialization” on page 7-57
- “Simplified Mode Behavior and Requirements” on page 7-58
- “When to Use Classic Initialization” on page 7-59

- 2 Set the initialization mode using the **Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection** parameter. Set the parameter to either **Classic** (default) or **Simplified**.

- 3 If you use simplified mode, you might need to modify your model to change the initialization mode for an existing model that is in classic mode to use simplified mode. For details, see “Convert from Classic to Simplified Initialization Mode” on page 7-64.

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 need to address, one-by-one. 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.

Because you might need to make several changes to your model during the conversion process, it is helpful to have the original model for reference and for comparing simulation results.

- 2** Set the **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** diagnostic to error.
- 3** Simulate the model and address any warnings.
- 4** In the Model Advisor, in the Simulink checks section, run **Check consistency of initialization parameters for Outport and Merge blocks**.
- 5** Address the issues that Model Advisor identifies.
- 6** Simulate the model to make sure that there are no errors.
- 7** Rerun the Model Advisor **Check consistency of initialization parameters for Outport and Merge blocks** 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 “Address Classic Mode Issues by Using Simplified Mode” on page 7-65.

Address Classic Mode Issues by Using Simplified Mode

In this section...

“Classic Mode Issues” on page 7-65

“Identity Transformation Can Change Model Behavior” on page 7-66

“Discrete-Time Integrator or S-Function Block Can Produce Inconsistent Output” on page 7-68

“Sorted Order Can Affect Merge Block Output” on page 7-71

Classic Mode Issues

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 need to make to use simplified mode.

- “Identity Transformation Can Change Model Behavior” on page 7-66.
Conditional subsystems that include identical subsystems can display different initial values before the first execution if both of these apply:
 - The model uses simplified initialization mode.
 - One or more of the identical subsystems outputs to an identity transformation block.
- “Discrete-Time Integrator or S-Function Block Can Produce Inconsistent Output” on page 7-68
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.
- “Sorted Order Can Affect Merge Block Output” on page 7-71
The sorted 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.

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 7-64.

Identity Transformation Can Change Model Behavior

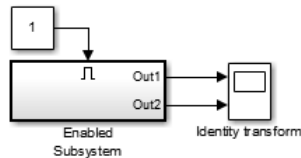
Conditional subsystems that include identical subsystems can display different initial values before the first execution if both of these apply:

- The model uses simplified initialization mode.
- One or more of the identical subsystems outputs to an identity transformation block.

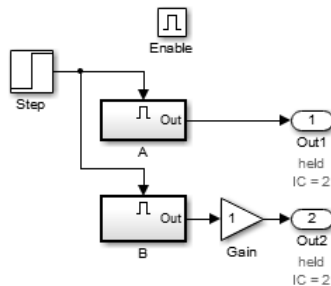
An identity transformation block is a block that does not change the value of its input signal. Examples of identify transform blocks are a Signal Conversion block or a Gain block with a value of 1.

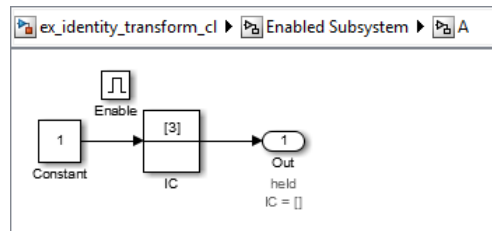
In the `ex_identity_transform_cl` model, subsystems A and B are identical, but B outputs to a Gain block, which in turn outputs to an Outport block.

`ex_identity_transform_cl` ▶

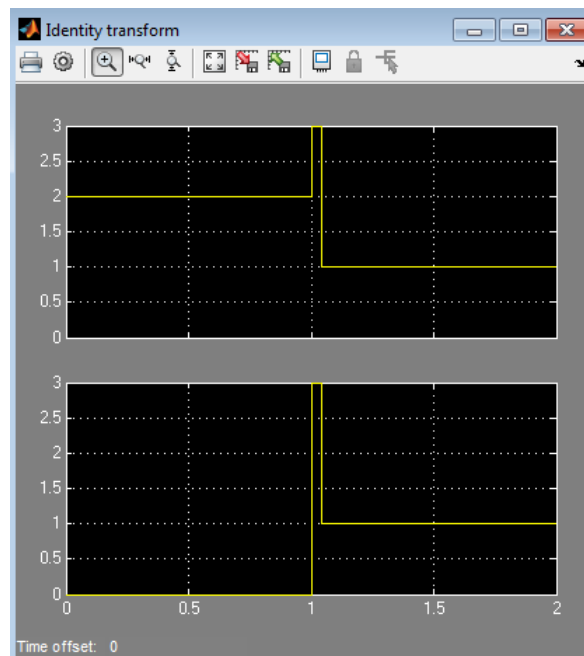


`ex_identity_transform_cl` ▶ `Enabled Subsystem` ▶





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.

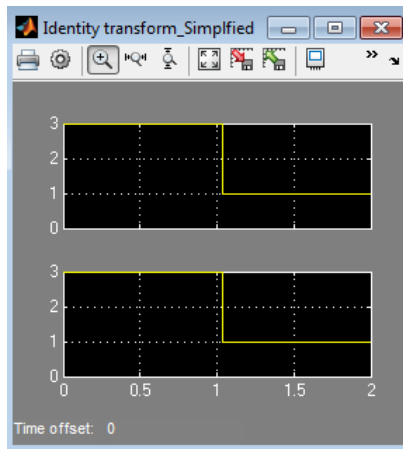


If you update the model to use simplified initialization mode (see `ex_identity_transform_simpl`), the model looks the same. The steps required to convert `ex_identity_transform_cl` to `ex_identity_transform_simpl` are:

- 1 Set **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** to error.
- 2 Set **Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection** to Simplified.
- 3 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.

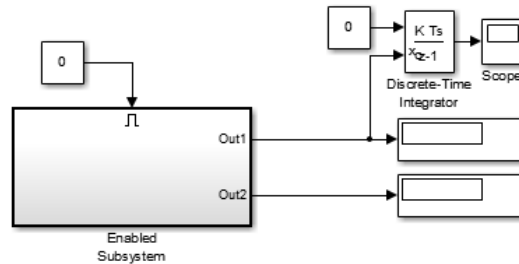


Discrete-Time Integrator or S-Function Block Can Produce Inconsistent Output

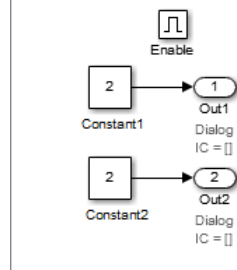
Conditional subsystems that use classic initialization mode and whose output connects to a Discrete-Time Integrator block or S-Function block can produce inconsistent output.

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.

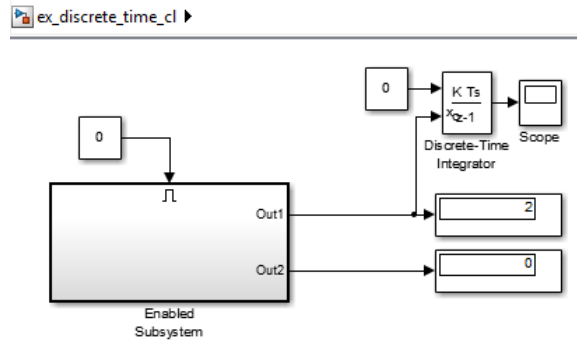
`ex_discrete_time_cl` ▶



`ex_discrete_time_cl` ▶ `Enabled Subsystem`



When you simulate the model, the two display blocks show different values.



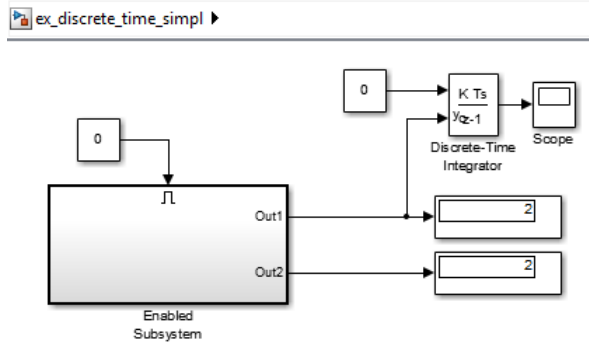
The Constant1 block, which is connected to the Discrete-Time Integrator block, executes, even though the conditional subsystem is disabled. The top Display block shows a value of 2, which is the value of the Constant1 block. The Constant2 block does not execute, so the bottom Display block shows a value 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 Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals to error.**
- 2 Set Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection to Simplified.**
- 3 For the Output 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 Output blocks inherit their initial value.

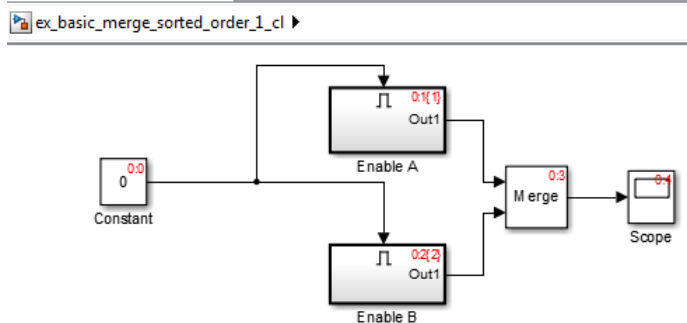


Sorted Order Can Affect Merge Block Output

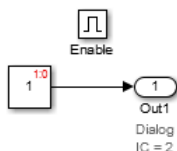
The sorted order of conditional subsystems that used classic mode initialization, when connected to a Merge block, can affect the output of that Merge block. A change in block execution order can produce unexpected results. The behavior depends on how you set the **Output When Disabled** parameter.

Example Using Default Setting for the Output When Disabled Parameter

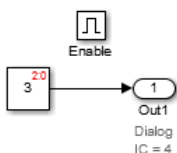
The `ex_basic_merge_sorted_order_1_cl` model has two identical enabled subsystems (Enable A and Enable B) that connect to a Merge block. When you simulate the model, the red numbers show the sorted execution order of the blocks.



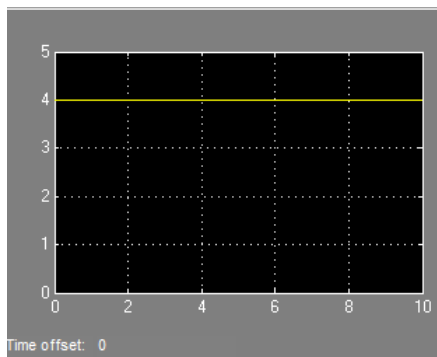
ex_basic_merge_sorted_order_1_c1 ▶ Enable A



ex_basic_merge_sorted_order_1_c1 ▶ Enable B

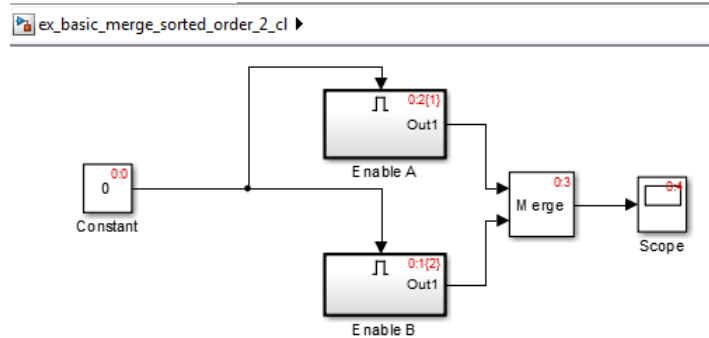


When you simulate the model, the Scope block looks like this:

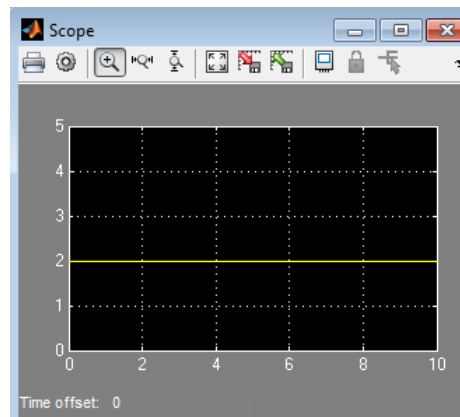


The `ex_basic_merge_sorted_order_2_c1` model is the same as `ex_merge_sorted_1_c1`, except that the block execution order is the reverse of the default execution order. To change the execution order:

- 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 this:



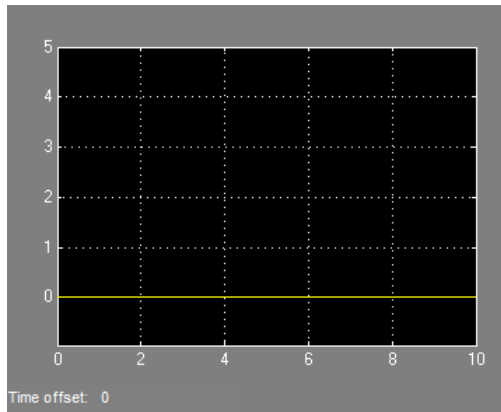
The change in sorted order produces different results from identical conditional subsystems.

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 Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals to error.

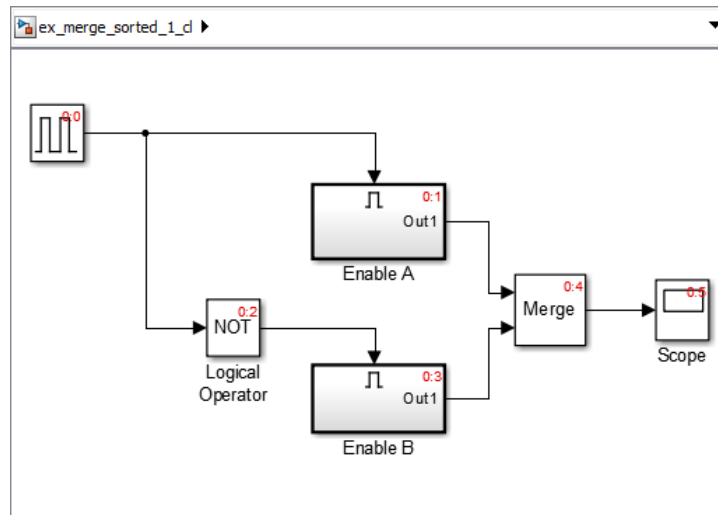
2 Set Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection to Simplified.

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 “Initializing Signal Values” on page 50-16. When you simulate each simplified mode model, both models produce the same results.

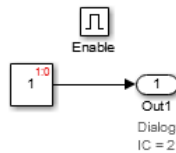


Example Using Output When Disabled Parameter Set to Reset

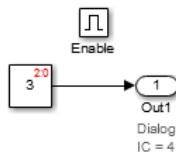
The `ex_merge_sorted_1_c1` model has two enabled subsystems (Enable A and Enable B) that connect to a Merge block. When you simulate the model, the red numbers show the sorted execution order of the blocks.



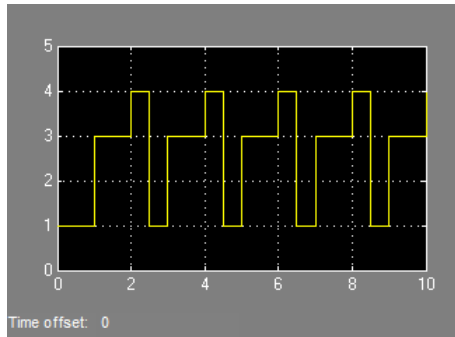
ex_merge_sorted_1_cl ▶ Enable A



ex_merge_sorted_1_cl ▶ Enable B

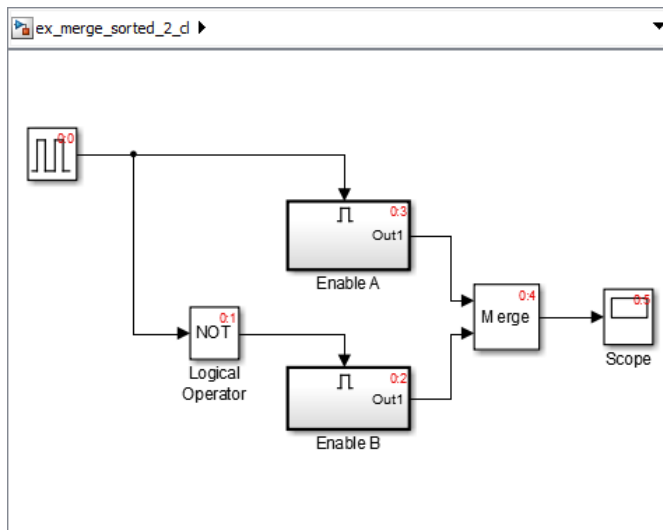


When you simulate the model, the Scope block looks like this:

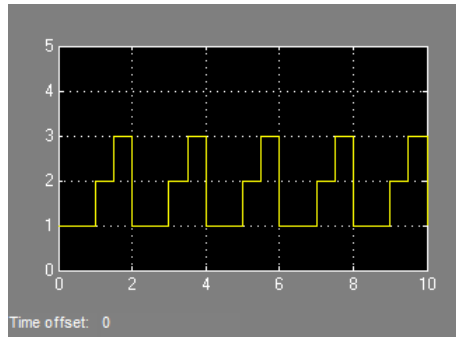


The `ex_merge_sorted_2_c1` model is the same as `ex_merge_sorted_1_c1`, except that the block execution order is the reverse of the default execution order. To change the execution order:

- 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 this:

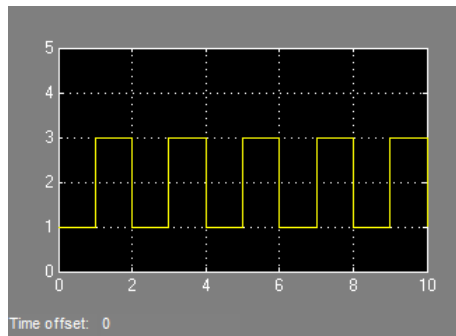


The change in sorted order produces different results from identical conditional subsystems.

To update the models to use simplified initialization mode (see `ex_merge_sorted_1_simpl` and `ex_merge_sorted_2_simpl`):

- 1** Set **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** to error.
- 2** Set **Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection** to Simplified.
- 3** For the Output blocks in Enable A and Enable B, set the **Output when disabled** parameter to held. Simplified mode does not support reset.

When you simulate each simplified mode model, both models produce the same results.



Modeling Variant Systems

- “Variant Systems Overview” on page 8-2
- “Variant Controls” on page 8-4
- “Set Up Model Variants” on page 8-8
- “Set Up Variant Subsystems” on page 8-14
- “Convert Subsystem Blocks to Variant Subsystem Blocks” on page 8-19
- “Select the Active Variant for Simulation” on page 8-20
- “Variant Management” on page 8-22
- “Manage Variants Using Variant Manager” on page 8-25

Variant Systems Overview

In this section...
“What Is A Variant?” on page 8-2
“Variant Controls and Active Variants” on page 8-2
“How Simulink Represents Variants” on page 8-3
“Workflow for Implementing Variants” on page 8-3

What Is A Variant?

A variant is each implementation of a model representing its different aspects. A variant differs in some selections from other, similar models. Using variants, you can specify multiple configurations in a single, unified block diagram. Prior to model simulation, you can switch programmatically between variants.

For example, suppose a design engineer has a model that simulates an automobile. Different models of an automobile can have many similarities but differ in terms of fuel usage, engine size, or emission standards. Each implementation of the model representing a specific automobile model is a variant. Use variants to:

- Represent a single model with different configurations.
- Programmatically select one active variant from multiple choices.
- Select one variant from available choices, irrespective of the variant control (override variants) which determines which variant is active.
- Optionally, generate code using Simulink Coder with preprocessor conditionals (`#if`, `#elif`, `#endif`) to enable selection of the active variant at compile time.

Variant Controls and Active Variants

The variant control determines which variant is active. It can be a Boolean condition expression or a `Simulink.Variant` object specifying the condition expression. During model compilation, Simulink evaluates the variant control

to true or false. The active variant is the one for which variant control evaluates to true.

How Simulink Represents Variants

To add variants to models, Simulink provides the following blocks:

- Model Variants
 - Models as variant choices.
 - Each variant has an associated variant control.
- Variant Subsystem
 - Subsystems as variant choices.
 - Each variant has an associated variant control.

Workflow for Implementing Variants

You can implement variants in your model by using the Model Variants block or the Variant Subsystem block. The workflow for setting up variants and simulating a model using the active variants is:

- 1** Create the subsystems or referenced models that you want to use as variants.
- 2** Depending on your application, use the Model Variants block or the Variant Subsystem block.
- 3** Define a variant control for each variant. The active variant is the variant control that evaluates to true during model compilation.
- 4** Simulate using the active variant.
- 5** Modify the variant control to select another variant and simulate again.
- 6 (Optional)** Depending on your application, you can generate code for the active variant or for all variants. See “Generate Code for Model Variants” and “Generate Code for Variant Subsystems”.
- 7** For variants defined in the base workspace, save the control variables to a MAT-file. See “Exporting Workspace Variables” on page 9-60.

Variant Controls

In this section...

“Create Variant Controls” on page 8-4

“Save Variant Controls” on page 8-5

“Reuse Simulink.Variant Objects” on page 8-5

“Enumerated Types as Variant Control” on page 8-6

The variant control can be a Boolean condition expression, or a `Simulink.Variant` object representing a Boolean condition expression. You define variant controls in the MATLAB workspace. You can specify the condition expression directly without using variant objects in the **Variant control** field. For example, you can specify `VSS_MODE == 1` directly without creating variant objects.

The condition expression can include:

- Scalar variables
- `Simulink.parameter` object with valid data type including enumerated types
- Operators: `==`, `~=`, `&&`, `||`, `~`
- Parentheses for grouping

You can programmatically change the active variant by modifying the values of variant control variables in the base work space, in the block parameter dialog box, or from the command line. You can also manually override the variant selection from the **Variant > Override using > variant choice** menu.

For more information, see “Select the Active Variant for Simulation” on page 8-20.

Create Variant Controls

- To create variant controls, define the condition expression as follows:

```
Variable == Value
```

For example:

```
VSS_MODE == 1
```


- Define the variant object as follows:

```
variantObjectName = Simulink.Variant(conditionExpression)
```

For example:

```
VSS_NONLINEAR_CONTROLLER = Simulink.Variant ('VSS_MODE == 1')
```

Note You must define the `Simulink.Variant` objects every time you simulate the model.

You can create variant controls in the block dialog box. Under **Variant choices**, click  **Create/edit variant control** button.

If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects. You define the `Simulink.Parameter` object from the Model Explorer. See “Generate Preprocessor Conditionals for Variant Systems”.

Save Variant Controls

If you want to reload variant control variables in your model, you must save them to a MAT-file. Right-click the Workspace window and select **Save**.

Reuse Simulink.Variant Objects

If you want to reuse some common conditions for multiple variant choices, use `Simulink.Variant` objects. For example, the models `AutoMRVar` and `AutoSSVar`, highlight only the current active variant. The inactive choice is greyed out.

Reusing variant objects allows you to globally reconfigure the model hierarchy to suit different environments by changing only the variant control values.

No matter how many simulation environments you define, selecting an environment requires setting only a variable or parameter value.

Enumerated Types as Variant Control

You can use enumerated types to give meaningful names to the integers that you use as variant control values. For more information, see “About Simulink Enumerations” on page 47-2.

For example, for the `sldemo_mdhref_variants_enum` model, the following classes are defined:

```
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
```

The variant objects defined for these classes are:

```
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')
```

For example, for the `sldemo_variant_subsystems_enum` model, we have the following classes defined:

```
classdef sldemo_vss_CONTROLLER_TYPE < Simulink.IntEnumType
    enumeration
```

```

        NONLINEAR (1)
        SECOND_ORDER (2)
    end
end
classdef sldemo_vss_BUILD_TYPE < Simulink.IntEnumType
    enumeration
        PROTOTYPE (1)
        PRODUCTION (2)
    end
end
end

```

The variant objects defined for these classes are:

```

VSSE_NONLINEAR_CONTROLLER = Simulink.Variant...
('VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.NONLINEAR')
VSSE_SECOND_ORDER_CONTROLLER =Simulink.Variant...
('VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.SECOND_ORDER')
VSSE_PROTOTYPE =Simulink.Variant...
('VSSE_CURRENT_BUILD==sldemo_vss_BUILD_TYPE.PROTOTYPE')
VSSE_PRODUCTION =Simulink.Variant...
('VSSE_CURRENT_BUILD==sldemo_vss_BUILD_TYPE.PRODUCTION')

```

You define the variant objects in the MATLAB workspace.

Using enumerated types makes the generated code easy to read because it contains the names of the values rather than integers.

Set Up Model Variants

In this section...

“Configure the Model Variants Block” on page 8-9

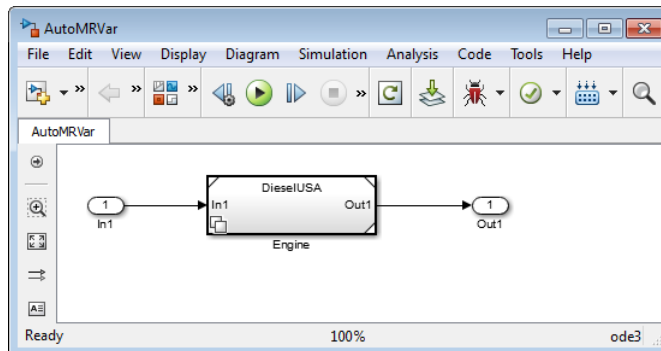
“Disable and Enable Model Variants” on page 8-12


“Parameterize Model Variants” on page 8-13

“Additional Examples” on page 8-13

Open the example model `AutoMRVar`. You can also open the model from the MATLAB prompt as:

```
addpath([docroot ' /toolbox/simulink/ug/examples/variants/mdlref/ ']);
open('AutoMRVar');
```



- The symbol  appears in the lower-left corner of the block to indicate that it uses variants. The outline of model variant block depends on the simulation mode.
- The name of the variant that was active the last time you saved the model appears on the block.
- When you change the active variant, the variant block refreshes. Its name changes to reflect the current active variant.
- When you open the example model, the load function loads a MAT-file that populates the base workspace with the variables and objects used by the model.



Name ^	Value
DE	<1x1 Simulink.Variant>
DU	<1x1 Simulink.Variant>
EMIS	1
FUEL	2
GE	<1x1 Simulink.Variant>
GU	<1x1 Simulink.Variant>

The example shows the use of variants for the following cases:

- The automobile can use a diesel or a gasoline engine.
- Each engine must meet the European or American (USA) emission standard.

AutoMRVar implements the automobile application using the Model Variants block named `Engine`. The `Engine` block specifies four referenced models. Each referenced model represents one permutation of engine fuel and emission standards. The variant choices are listed in the table:

Model Name	Variant Control	Condition (read only)
GasolUSA	GU	FUEL==1 && EMIS==1
GasolEuro	GE	FUEL==1 && EMIS==2
DieselUSA	DU	FUEL==2 && EMIS==1
DieselEuro	DE	FUEL==2 && EMIS==2

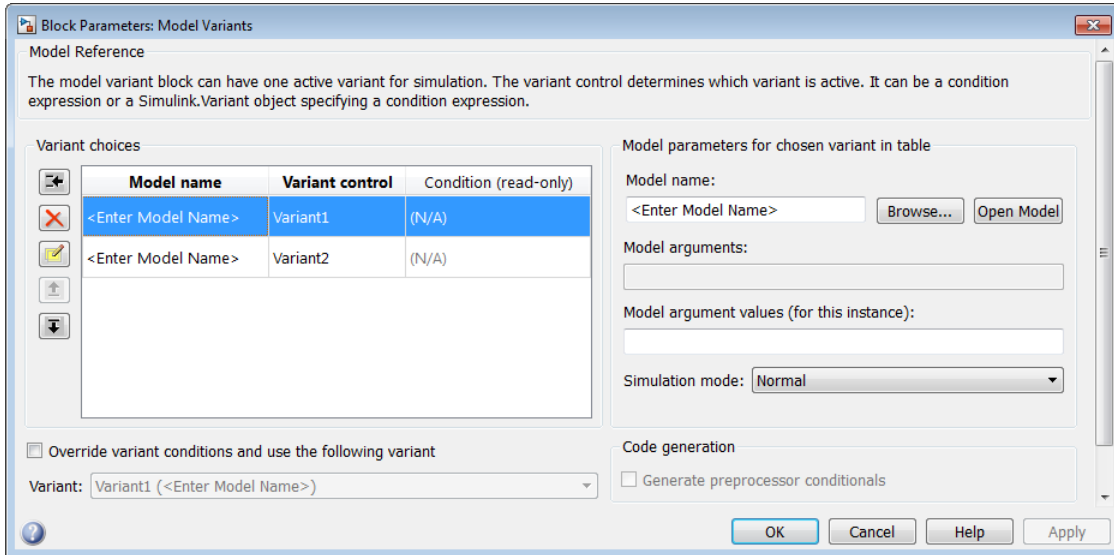
Note You can use condition expressions directly in the **Variant control** field. You do not need to create `Simulink.Variant` objects.

Configure the Model Variants Block

You can configure a Model Variants block and specify your variant choices.

- 1 Open a new, empty model.
- 2 From the **Simulink Library Browser**, **Ports & Subsystems** library, add a Model Variants block to the model.

- 3 Right-click the Model Variants block and select **Block Parameters (ModelReference)** from the context menu.



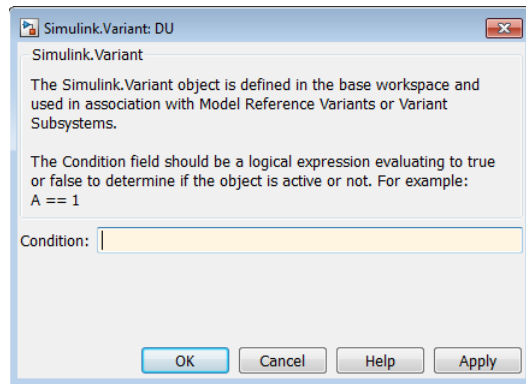
- 4 Under **Variant choices**, specify the model choices in the **Model name** column. To specify a protected model, use the extension `.slxp` or `.mdl.p`. For more information, see “Protected Model” on page 6-69.


- 5 For each model choice, specify the variant control in the **Variant control** column. The variant control can be a Boolean condition expression or a `Simulink.Variant` object representing a Boolean condition expression.

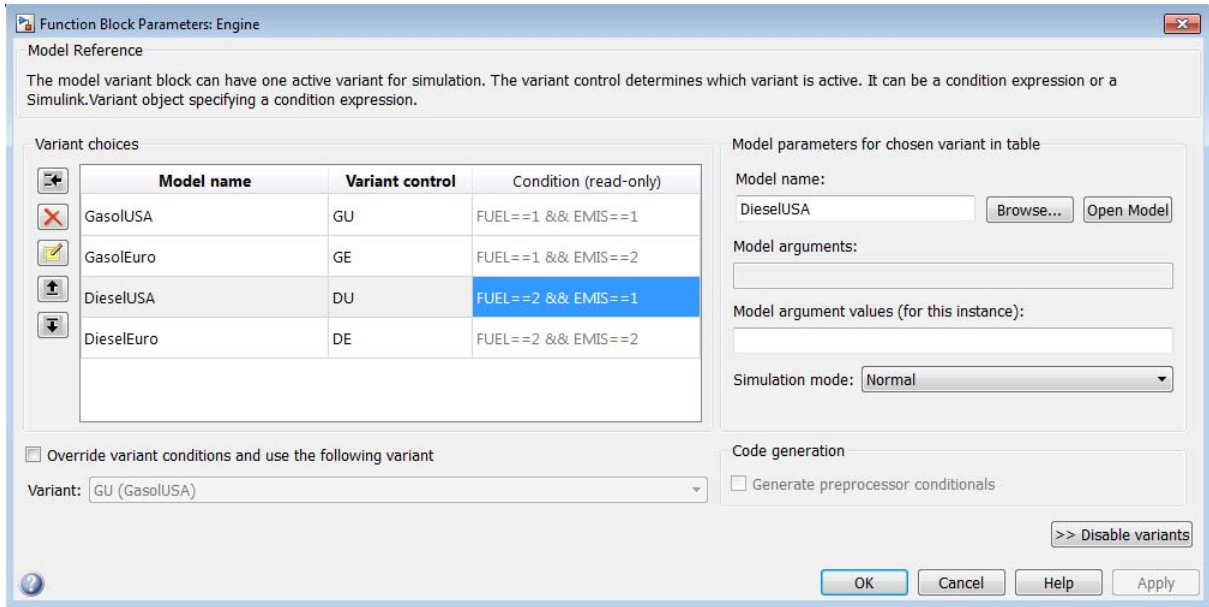
You must populate the **Variant control** column for each choice. You cannot comment out (%) variant control values for the Model Variants block. However, for Variant Subsystem blocks, you can comment out variant choices.

- 6 To edit the condition which determines the active variant choice, click **Create/Edit selected variant**. In the dialog box enter the condition.





- 7 (Optional)** Specify the model arguments and the **Simulation mode**. All simulation modes work with model variants. For more information, see “Parameterize Model Variants” on page 8-13 and “Referenced Model Simulation Modes” on page 6-21.
- 8** If you want to add more variant choices, click  **Add a new variant**.
- 9** Repeat the previous steps until you have specified all your referenced models and their associated variant controls. For more information, see Model Variants block page.
- 10** After you have added all your variant choices click **OK** to close the dialog box.



For next steps, see “Select the Active Variant for Simulation” on page 8-20.

Disable and Enable Model Variants

You can disable model variants without losing your variant settings. After you enable variants, they remain enabled until you explicitly disable them. You can close and reopen the Model Variants block without affecting the variant specification.

To disable variants from a Model Variants block:

- 1 Right-click the block and select **Block Parameters (ModelReference)** to open the block parameters dialog box.
- 2 Click **Disable Variants**.

Disabling variants:

- Hides and ignores the content of the **Variant choices** section of the dialog box.

- Retains the active variant as the model name.
- Ignores subsequent changes to variant control variables and other models, other than the current model.

If you decide to reenable variants later, select **Enable Variants**. The Model Variants block selects an active variant according to the current base workspace variables and conditions.

Parameterize Model Variants

If you want, you can apply a parameter to a variant control. Parameter values are the same as for any referenced model. For more information, see “Parameterize Model References” on page 6-54).

- 1** Under **Variant choices** select the row in the table for the variant control that you want to parameterize.
- 2** In the **Model argument values (for this instance)** field, specify the parameter.
- 3** Click **Apply**.

Repeat these steps for each parameter that you want to parameterize.

Additional Examples

For additional examples of model reference variants, in the Help browser, select **Simulink > Examples > Modeling Features > Model Reference > Model Reference Variants**.

The example `sldemo_mdhref_variants` shows a model variant.

Set Up Variant Subsystems

In this section...

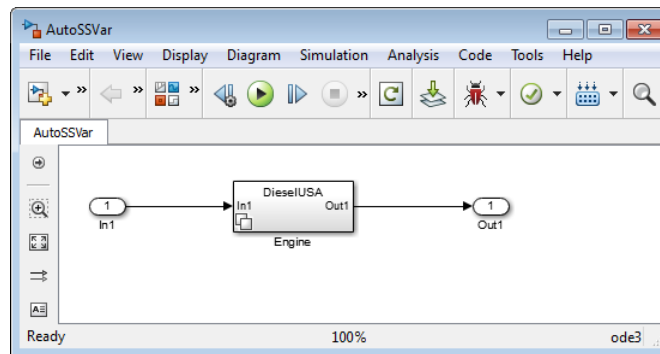
“Configure the Variant Subsystem Block” on page 8-17


“Disable or Enable Subsystem Variants” on page 8-17

“Additional Examples” on page 8-18

The figure shows the example model AutoSSVar. Open the example model AutoSSVar from the MATLAB prompt:

```
addpath([docroot ' /toolbox/simulink/ug/examples/variants mdlref/ ']);
open('AutoSSVar');
```



- The symbol  appears in the lower-left corner of the block to indicate that variants is enabled.
- The name of the variant that was active appears on the block.
- When you change the active variant, the variant block name changes.
- When you open the example model, a callback function loads a MAT-file that populates the base workspace with the variables that the model uses. The base workspace contains the variant control variables.



The screenshot shows the Simulink Workspace window with a table of variant parameters. The table has two columns: 'Name' and 'Value'. The parameters listed are DE, DU, EMIS, FUEL, GE, and GU. The values for DE, DU, GE, and GU are '<1x1 Simulink.Variant>', while EMIS is '1' and FUEL is '2'.

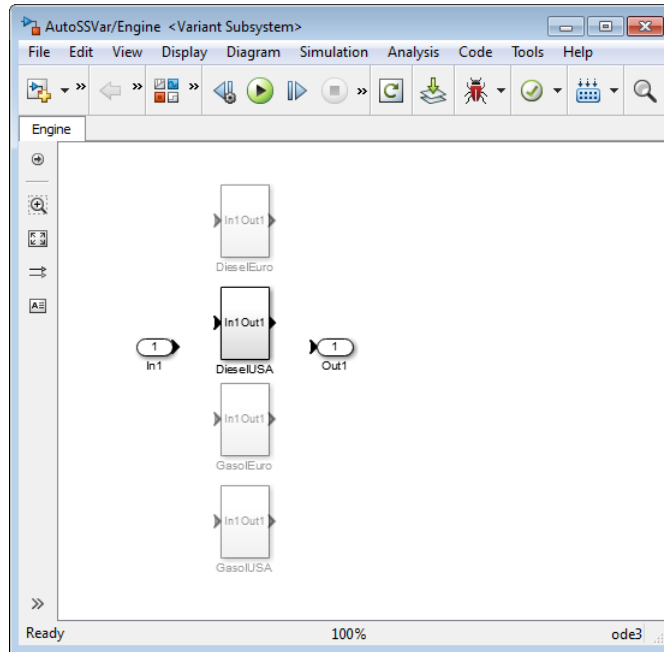
Name	Value
DE	<1x1 Simulink.Variant>
DU	<1x1 Simulink.Variant>
EMIS	1
FUEL	2
GE	<1x1 Simulink.Variant>
GU	<1x1 Simulink.Variant>

The example model, AutoSSVar, implements the automobile models using a Variant Subsystem block, named Engine. The Engine block consists of a set of four subsystem blocks. Each subsystem represents one permutation of engine fuel and emission standards.

The example shows the use of variants for the following cases:

- The automobile can use a diesel or a gasoline engine.
- Each engine must meet the European or American (USA) emission standard.

Double-click the Model Variant block, Engine, to view the subsystems.





The only blocks allowed in the Variant Subsystem block diagram are Inport, Outport, and Subsystem. The subsystem diagram for a Variant Subsystem block has no connections. The variant choices are listed in this table:

Name (read-only)	Variant control	Condition (read-only)
DieselEuro	DE	FUEL==2 && EMIS==2
DieselUSA	DU	FUEL==2 && EMIS==1
GasolEuro	GE	FUEL==1 && EMIS==2
GasolUSA	GU	FUEL==1 && EMIS==1

Note You can use condition expressions directly in the **Variant control** field. You do not need to create `Simulink.Variant` objects.

Configure the Variant Subsystem Block

To configure a Variant Subsystem block and specify your variants:

- 1 Open a new model.
- 2 From the **Simulink Library Browser**, **Ports & Subsystems** library, add a Variant Subsystem block to the model.
- 3 Double-click the Variant Subsystem block to expand it.
- 4 To represent each variant choice, drag Subsystems blocks from the **Simulink Library Browser** to the model.
- 5 Right-click the Variant Subsystem block and select **Block Parameters (Subsystem)**. The block parameters dialog box for the Variant Subsystem appears. The names of the subsystem blocks that were added as variant choice appear in the **Name (read-only)** column under **Variant choices**.
- 6 For each variant choice, add the variant control values in the **Variant control** column. If the variable is a `Simulink.Variant` object, you must first define the variant object in the MATLAB workspace. If you want to use a condition expression, directly type it in the **Variant control** column. For more information, see “Variant Controls” on page 8-4.
- 7 If you want to edit a variant control which is a `Simulink.Variant` object, click  **Create/Edit selected variant object** button and use the dialog box.  **Create/Edit selected variant object** is disabled for condition expressions.
- 8 To apply your changes, click **Apply**.

Disable or Enable Subsystem Variants

You can disable subsystem variants:

- 1 Under **Variant choices**, double-click the name of the variant control that you want to comment out.
- 2 Add % to the beginning of the variant control name.

3 Apply your changes.

To re-enable the subsystem variant, remove the % symbol. You can choose to keep the **Variant control** field empty.

You can override variants and make a specific variant always the active choice. For more information, see “Select the Active Variant for Simulation” on page 8-20 .

Additional Examples

For additional examples, in the Help browser, select **Simulink > Examples > Modeling Features > Subsystems > Variant Subsystems**. The example `sldemo_variant_subsystems` shows a variant subsystem.

Convert Subsystem Blocks to Variant Subsystem Blocks

You can convert normal Subsystem blocks and Configurable Subsystems blocks to Variant Subsystem block. For more information, see Configurable Subsystem.

To convert a normal Subsystem block to a Variant Subsystem block:

- Right-click the block and select **Subsystem & Model Reference > Convert Subsystem to > Variant Subsystem**.

Converting a Subsystem block to a Variant Subsystem block in a model does the following:

- Creates a Variant Subsystem block in place of the Subsystem block with all the connections to the ports in place.
- Adds the original subsystem as a variant choice in the new Variant Subsystem block.
- Overrides the Variant Subsystem block to use the subsystem that was originally the active choice.
- Preserves the link to the library. For linked subsystems, adds the linked subsystem as a variant choice.

Preserves the mask on subsystem blocks and copies it to the variant choice.

Select the Active Variant for Simulation

In this section...

“Set the Active Variant” on page 8-20

“Override Variants” on page 8-20

“Ignore Variant Choices” on page 8-21

“Open the Active Variant” on page 8-21

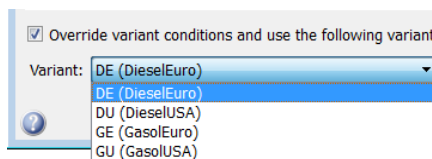
Set the Active Variant

At the MATLAB command line, set the variant control variable to the variant choice that you want to make active. For example, in the model `sldemo_variant_subsystems`, set `VSS_MODE=2` to make the Second-Order Controller the active variant.

Note When you change the active variant, the block display refreshes. You are not required to update the diagram (**Ctrl + D**), to reflect the change in an active variant. The blocks that are part of the inactive choice appear greyed out.

Override Variants

- Right-click the block and select **Variant > Override using > variant choice**.
- In the block parameter dialog box, select the **Override variant conditions and use the following variant** check box. Select the variant that you want to set as the active variant as the **Variant** value.



Use the `set_param` command to override the active variant.

For example, to make the variant choice `VSS_NONLINEAR_CONTROLLER` active:

```
set_param(gcb, 'OverrideUsingVariant', 'VSS_NONLINEAR_CONTROLLER')
```

Ignore Variant Choices

Variant control names which are empty or commented (starts with a %) are ignored in selecting active variants.

Open the Active Variant

When you open a model, variant blocks display the name of the variant that was active the last time that you saved model was saved. You can use the **Variant** menu to open the active variant. Right-click the block and select **Variant > Open > (active variant)**.

To find the current active variant block, at the command line, enter:

```
get_param(gcb, 'ActiveVariant')
```

To find the path to the current active variant block, at the command line, enter:

```
get_param(gcb, 'ActiveVariantBlock')
```

Note The `ActiveVariantBlock` parameter is supported only for the Variant Subsystem block.

Variant Management

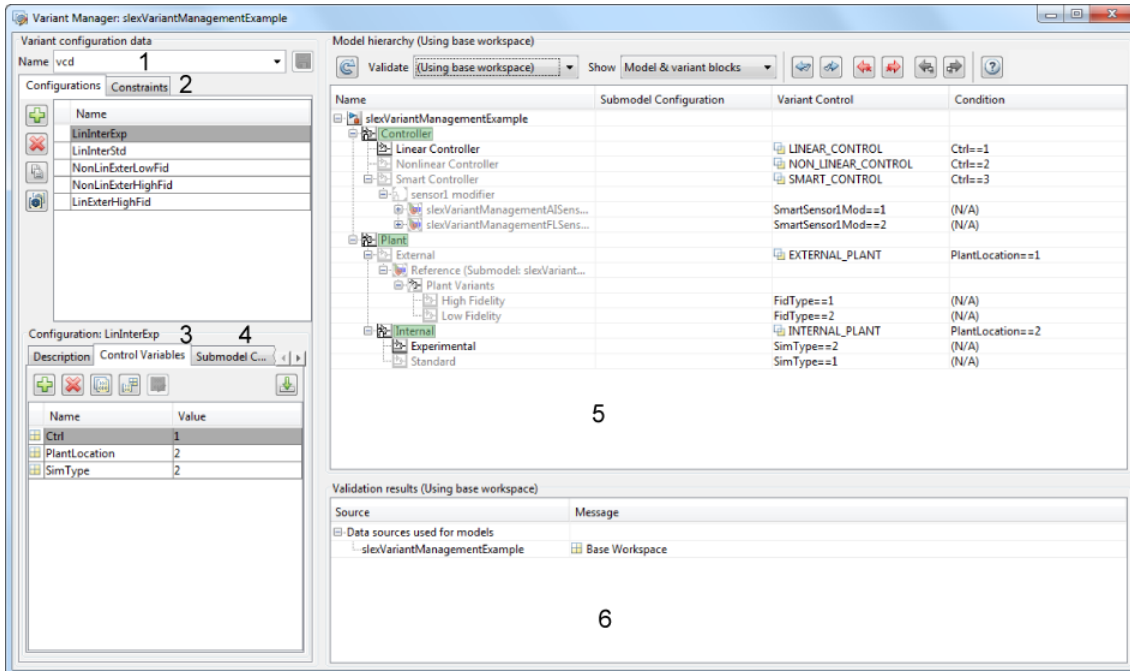
In this section...
“Manage Variants” on page 8-22
“Variant Manager” on page 8-22
“Considerations in Model Hierarchy Validation” on page 8-24

Manage Variants

You can use the variant blocks in Simulink to represent different possible combinations for a model. For example, for an automobile, you can have different combinations of fuel, engine capacity, and emission standards for different automobile models. Each model represents a set of configurations which you can model using choices of variant blocks. You can also nest variant blocks, thereby providing more flexibility. The possibility to create large number of configurations when variants are used, gives rise to several challenges in managing the variant complexity.

Variant Manager

Variant Manager is a tool that allows you to define and manage variant configurations for Simulink models graphically.



- 1 Variant configuration data:** The variant configuration data object, stores a collection of variant configurations, constraints, and the default active configuration.
- 2 Configuration:** The variant configuration is a set of variant control variables and values, submodel configurations, and constraints that must be satisfied. **Constraints** are Boolean expression that must evaluate to true.
- 3 Control Variables:** The Control variables are name-value pairs defined as vector of structures having fields **Name** and **Value**. The values of control variables are checked during validation of the configuration. The variant control variables are used to control which variants are active.
- 4 Submodel Configuration:** Submodel configurations correspond to configurations for models referenced by model reference blocks.

5 Model hierarchy: The **Model hierarchy** pane, allows you to visualize and explore the variant hierarchy of a model and edit the properties of variant blocks, their choices, and variant objects.

6 Validation results: The **Validation results** pane displays information on source of control variables for the models in the hierarchy. The **Validation results** pane also displays information on errors encountered during validation of the variant configuration.

For more information, see “Variant Manager Overview”.

Using **Variant Manager**, you can:

- Explore, visualize, and manipulate variant hierarchy.
- Define, validate, and visualize variant configurations.
- Define constraints which must be satisfied by models.
- Specify one of the variant configurations as a default active configuration.
- Associate variant configuration data object of type `Simulink.VariantConfigurationData` with models.
- Define variant configurations, constraints, and save them as a variant configuration data object.
- Validate a variant configuration. You can validate configurations without updating a model.

Considerations in Model Hierarchy Validation

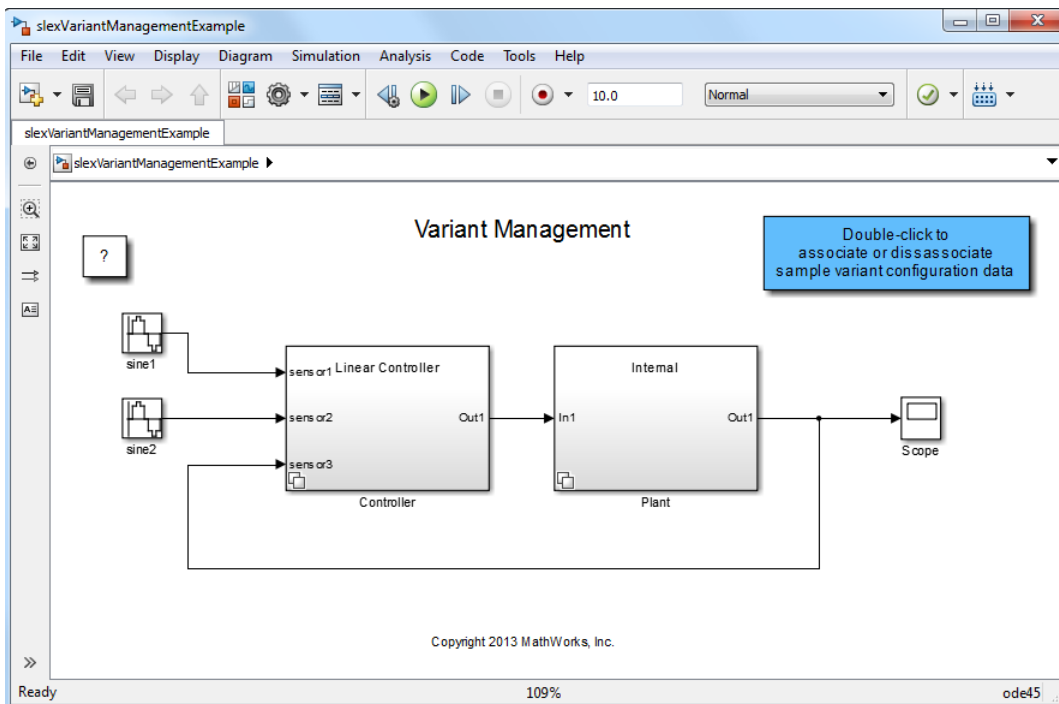
- For a model or referenced model that has a variant configuration data object with a default configuration defined, the control variables from the default configuration are loaded to the base workspace and are used for validation.
- For referenced models if the top model specifies the variant configuration, that specific variant configuration is used to validate the referenced models.
- For a model containing referenced models, we can have multiple variant configurations that use common set of control variables and submodel configurations. In such cases, all the variant configurations must have the same values for control variables and submodel configurations.

Manage Variants Using Variant Manager

In this section...

- “Open Variant Manager” on page 8-26
- “Add New Variant Configurations” on page 8-28
- “Validate Variant Configurations” on page 8-30
- “Set Default Active Configuration” on page 8-31
- “Define Constraints” on page 8-31
- “Save Variant Configuration Data” on page 8-32

The example model `slexVariantManagementExample` has multiple variant blocks forming a variant hierarchy as described below.



- Top level variant blocks named Controller and Plant.
- The Controller block has three choices: Linear Controller, Nonlinear Controller, and Smart Controller.
- The Smart Controller block references two models: `slexVariantManagementAISensor1ModifierMdlRef` and `slexVariantManagementFLSensor1ModifierMdlRef`.
- The Plant block has two choices: External and Internal.
- The Internal block has two choices: Experimental and Standard.
- The External block references the `slexVariantManagementExternalPlantMdlRef` model.

Open Variant Manager

Click `slexVariantManagementExample` to open the model. Optionally double-click the subsystem block on the top right to load sample data.

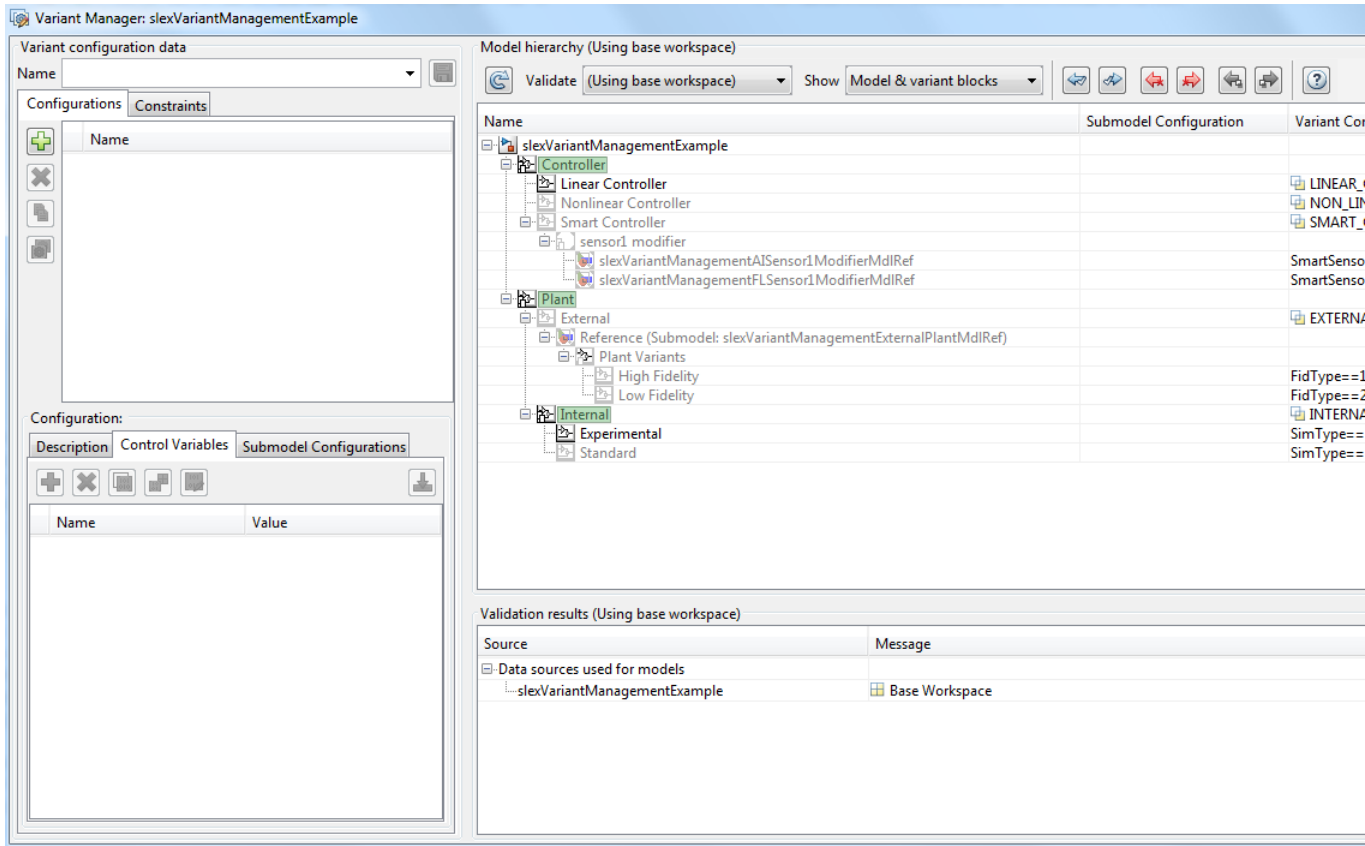
You can open **Variant Manager** in any one of the following ways:

- From the model editor, click **View > Variant Manager**.
- In the **Block Parameters** dialog box of either the Controller or the Plant block, click **Open block in Variant Manager**.

Note You can open the **Block Parameter** dialog box from **Diagram > Block Parameters (Subsystem)** or right-click the variant block and select **Block Parameters (Subsystem)**.







- Right-click either the Controller or the Plant block and select **Variant > Open in Variant Manager**.

The Variant Manager opens up as shown below. The blocks in the model have been expanded to display the complete hierarchy.

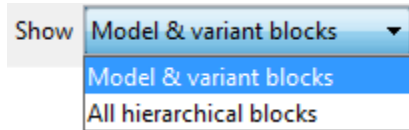


Browse Variant Hierarchy

Once you have **Variant Manager** open for the model, you can navigate between the active, invalid, and overridden choices in the variant hierarchy.



- To navigate between active variant rows, use  and  buttons.
- To navigate between variant rows with errors, use  and  buttons.
- To navigate between rows which represent overridden variant choices, use  and  buttons.

You can selectively display all hierarchical blocks or only model variant blocks. Use the options available on the **Show** drop down.



Add New Variant Configurations

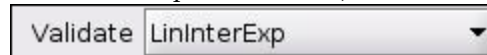
You can add variant configuration LinInterExp manually.

- 1 Click  on the **Configurations** tab and enter LinInterExp in the **Name** column.
- 2 Click  on the **Control Variables** tab to add the following control variables:

Name	Value
Ctrl	1
PlantLocation	2
SimType	2

- 3 Validate the model using the LinInterExp configuration.

From the drop down menu, select LinInterExp configuration from the












. You will see the following in the validation results pane:

Validation results (Configuration: LinInterExp)	
Source	Message
<ul style="list-style-type: none"> [-] Data sources used for models <ul style="list-style-type: none"> [-] slxVariantManagementExample <ul style="list-style-type: none"> [+] Base Workspace: Configuration 'LinInterExp' of variant configuration data object 'vcd' 	

You can add another variant configuration `LinInterStd` by importing control variables.

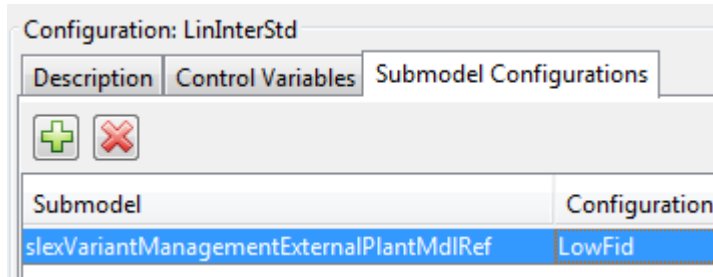
- 1 On the **Configurations** tab click  and enter `LinInterStd` in the **Name** column.
- 2 Click  button on the **Control Variables** tab, to import control variables from the workspace. These control variables are imported:

Configuration: LinInterStd		
Description	Control Variables	Submodel C...
     		
Name	Value	
Ctrl	1	
SmartSensor1Mod	0	
PlantLocation	2	
SimType	2	

You can convert any of the existing control variables to `Simulink.Parameter` object using the  button. You can also add or delete control variables using  and  buttons.

Additionally, from the **Submodel Configurations** tab, you can add a configuration to be used by model reference blocks. For example, for the

slxVariantManagementExternalPlantMdlRef model, add the LowFid configuration as shown below:

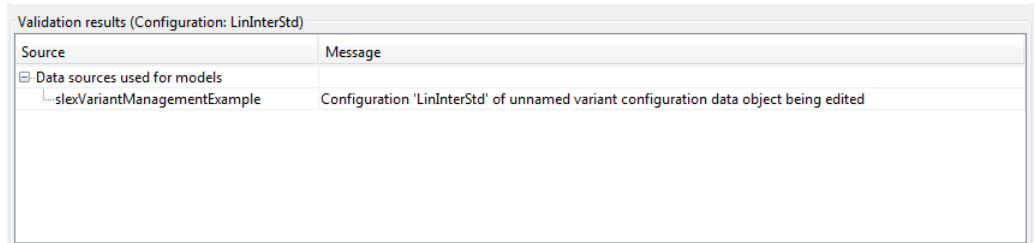



- 3 Validate the LinInterStd configuration.

From the **Validate** drop down menu, select configuration



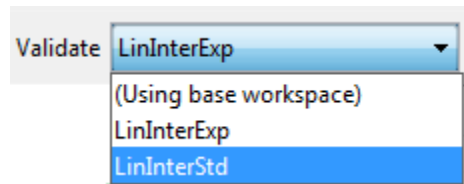
. On validation, you will see the following in the results:



You can also add copy of an existing variant configuration. For example, to add a copy of the LinInterExp variant configuration, click  button.


Validate Variant Configurations

You have LinInterExp and LinInterStd variant configurations defined. For example, if you want to validate using the LinInterStd variant configuration, select the same from the drop down.



Set Default Active Configuration

You can set one of the variant configurations as the default active configuration.

- 1 On the **Configurations** tab, select the `LinInterExp` variant configuration and click  button.
- 2 Verify that the default active variant is set to `LinInterExp`:


```
vcdObj.getDefaultConfiguration
```


You should see the following:

```
Name: 'LinInterExp'
Description: [1x81 char]
ControlVariables: [1x3 struct]
SubModelConfigurations: []
```

Define Constraints

Constraints are conditions that must evaluate to true for all variant configurations you define for a model. For example, you can add a constraint `LinNotExtern` to the example model.

- 1 Click the **Constraints** tab.
- 2 Click .
- 3 Enter `LinNotExtern` as the **Name** and `(Ctrl~=1) || (PlantLocation ~=1)` as the **Condition** for the constraint. For a model that has this constraint defined, you cannot create variant configurations that use the Linear Controller and External Plant Controller configuration.

4 To validate the constraints, click  .

Save Variant Configuration Data

Before closing **Variant Manager**, you can save the variant configuration data.

- Enter a name for the variant configuration data object in the **Name** field and press return.

If variant configuration data object of the specified name exists in base workspace, you will be prompted to **Associate** with it, else a variant configuration data object of the specified name is created and the model is associated with it.

Exploring, Searching, and Browsing Models

- “Model Explorer Overview” on page 9-2
- “Model Explorer: Model Hierarchy Pane” on page 9-9
- “Model Explorer: Contents Pane” on page 9-19
- “Control Model Explorer Contents Using Views” on page 9-26
- “Organize Data Display in Model Explorer” on page 9-36
- “Filter Objects in the Model Explorer” on page 9-46
- “Workspace Variables in Model Explorer” on page 9-52
- “Search Using Model Explorer” on page 9-63
- “Model Explorer: Property Dialog Pane” on page 9-70
- “Finder” on page 9-73
- “Model Browser” on page 9-80
- “Model Dependency Viewer” on page 9-83
- “View Linked Requirements in Models and Blocks” on page 9-96

Model Explorer Overview

In this section...

“What You Can Do Using the Model Explorer” on page 9-2

“Opening the Model Explorer” on page 9-2

“Model Explorer Components” on page 9-3

“The Main Toolbar” on page 9-4

“Adding Objects” on page 9-5

“Customizing the Model Explorer Interface” on page 9-5

“Basic Steps for Using the Model Explorer” on page 9-6


“Focusing on Specific Elements of a Model or Chart” on page 9-7

What You Can Do Using the Model Explorer

Use the Model Explorer to quickly view, modify, and add elements of Simulink models, Stateflow charts, and workspace variables. The Model Explorer provides several ways for you to focus on specific elements (for example, blocks, signals, and properties) without your having to navigate through the model diagram or chart.

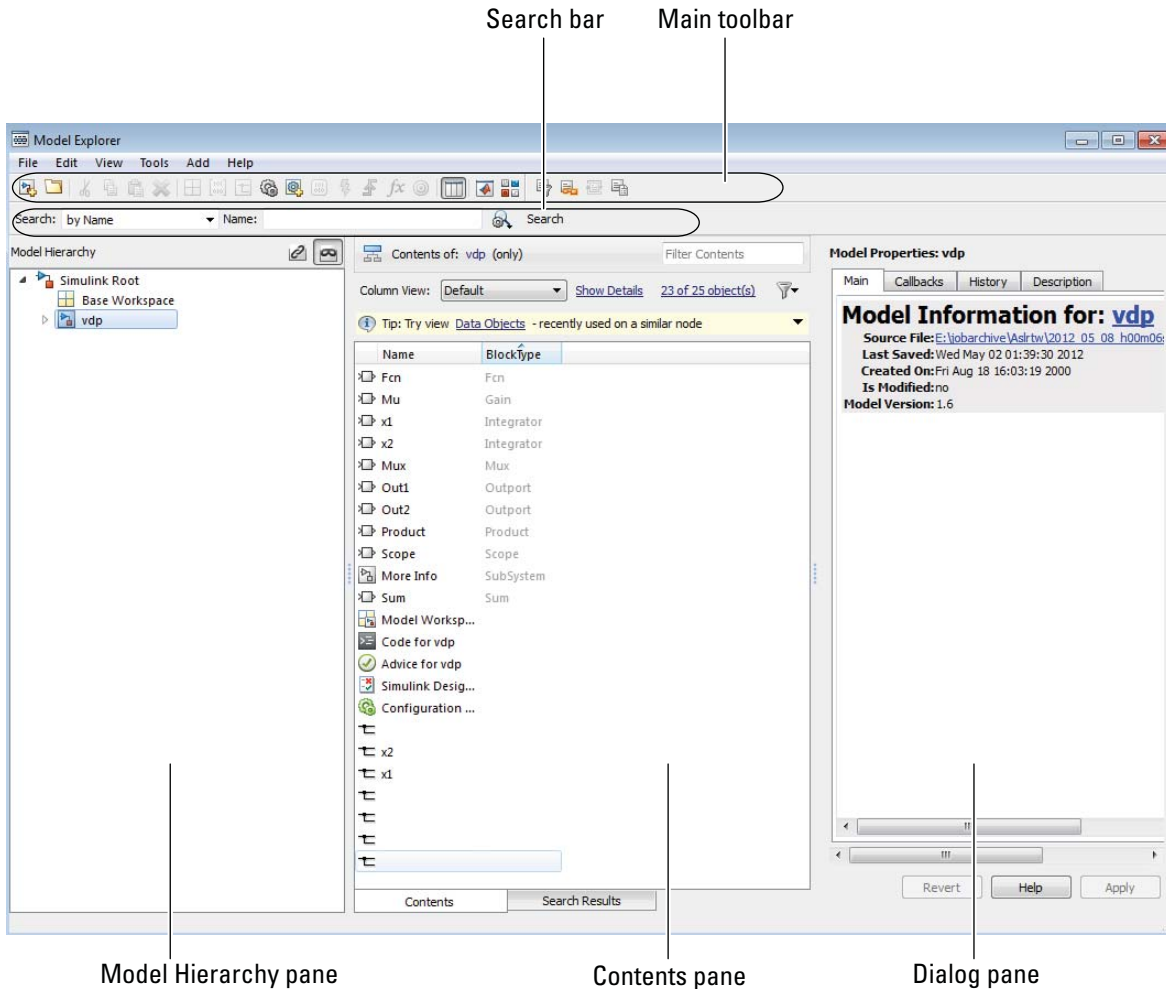
Opening the Model Explorer

To open the Model Explorer, use one of these approaches:

- From the Simulink Editor **View** menu, select **Model Explorer** or select the Model Explorer icon  from the toolbar.
- In an open model in the Simulink Editor, right-click a block and from the context menu, select **Explore**.
- In an open Stateflow chart, right-click in the drawing area and from the context menu, select **Explore**.
- At the MATLAB command line, enter `daexplr`.

Model Explorer Components

By default, the Model Explorer opens with three panes (**Model Hierarchy**, **Contents**, and **Dialog**), a main toolbar, and a search bar.





Component	Purpose	Documentation
Main toolbar	Execute Model Explorer commands	“The Main Toolbar” on page 9-4
Search bar	Perform a search within the context of the selected node in Model Hierarchy pane.	“Search Using Model Explorer” on page 9-63
Model Hierarchy pane	Navigate and explore model, chart, and workspace nodes	“Model Explorer: Model Hierarchy Pane” on page 9-9
Contents pane	Display and modify model or chart objects	“Model Explorer: Contents Pane” on page 9-19
Dialog pane	View and change the details of object properties	“Model Explorer: Property Dialog Pane” on page 9-70

The Main Toolbar

The main toolbar at the top of the Model Explorer provides buttons you click to perform Model Explorer operations. Most of the toolbar buttons perform actions that you can also perform using Model Explorer menu items.

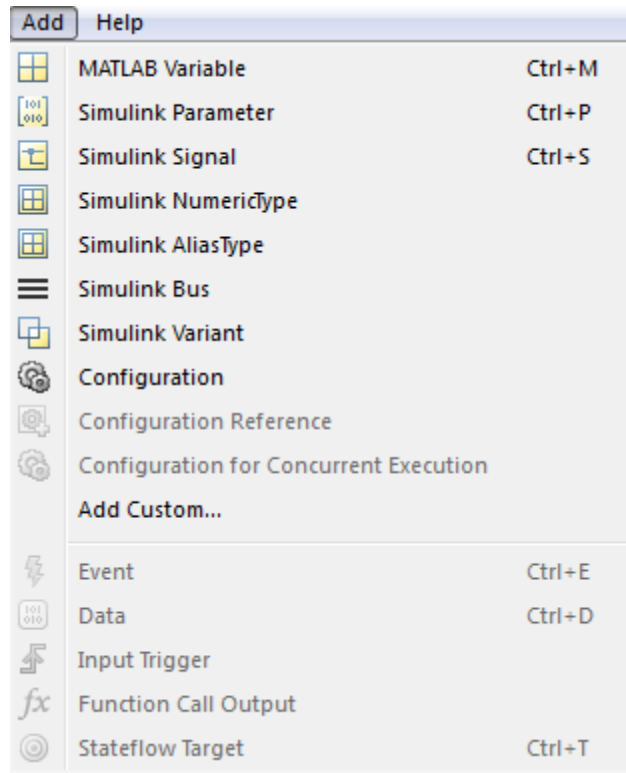
The toolbar buttons in the following table perform actions that you cannot perform using Model Explorer menus:

Button	Usage
	Bring the MATLAB window to the front.
	Display the Simulink Library Browser.

If you have Simulink Verification and Validation installed, you can use additional toolbar buttons relating to requirements links.

Adding Objects

You can use the Model Explorer to add many kinds of objects to a model, chart, or workspace. The types of objects that you can add depend on what node you select in the **Model Hierarchy** pane. Use toolbar buttons or the **Add** menu to add objects. The **Add** menu lists the kinds of objects you can add.



Customizing the Model Explorer Interface

You can customize the Model Explorer interface in several ways. This section describes how to show or hide the main toolbar and how to control the font size.

Other ways you can customize the Model Explorer interface include:

- “Marking Nonexistent Properties” on page 9-45
- “Show and Hide the Search Bar” on page 9-64
- “Showing and Hiding the Dialog Pane” on page 9-70

Showing and Hiding the Main Toolbar

To show or hide the main toolbar, in the Model Explorer select **View > Toolbars > Main Toolbar**.

Controlling the Font Size

You can change the font size in the Model Explorer panes:

- To increase the font size, press the **Ctrl + Plus Sign (+)**.
Alternatively, from the Model Explorer **View** menu, select **Increase Font Size**.
- To decrease the font size, press the **Ctrl + Minus Sign (-)**.
Alternatively, from the Model Explorer **View** menu, select **Decrease Font Size**.

Note The changes remain in effect for the Model Explorer and in the Simulink dialog boxes across Simulink sessions.

Basic Steps for Using the Model Explorer

Use the Model Explorer to perform a wide range of activities relating to viewing and changing model and chart elements. You can perform activities in any order, using panes in the order you choose. Your actions in one pane often affect other panes.

For example, if you want to edit properties of objects in a model, you might use a general workflow such as:

- 1 Open a model.
- 2 Open the Model Explorer.

- 3** Select the model in the **Model Hierarchy** pane, specifying whether the Model Explorer displays only the current system or the whole system hierarchy of the current system
- 4** Control what model information the **Contents** pane displays, and how it displays that information, by using a combination of:
 - The **View > Column View** option to control which property columns to display
 - The **View > Row Filter** option to control which types of objects to display
 - Techniques to directly manipulate column headings
- 5** Identify model elements with specific values, using the search bar.
- 6** Edit the values for model elements, in either the **Contents** pane or the **Dialog** pane. To edit workspace variables, you can use the Variable Editor.

Focusing on Specific Elements of a Model or Chart

As you explore a model or chart, you might want to narrow the contents that you see in the Model Explorer to particular elements of a model or chart. You can use several different techniques. The following table summarizes techniques for controlling what content the Model Explorer displays and how the contents appear.

Technique	When to Use	Documentation
Show partial or whole model hierarchy contents	To control how much of a hierarchical model to display	“Displaying Partial or Whole Model Hierarchy Contents” on page 9-12
Use the Row Filter option	To focus on, or hide, a specific kind of a model object, such as signals	“Using the Row Filter Option” on page 9-46

Technique	When to Use	Documentation
Search	To find objects that might not be currently displayed	“Search Using Model Explorer” on page 9-63
Filter contents	To focus on specific objects in the Contents pane, based on a search string	“Filtering Contents” on page 9-48

Once you have the general set of data that you are interested in, you can use the following techniques to organize the display of contents.

Technique	When to Use	Documentation
Sort	To quickly organize data for a property in ascending or descending order	“Sorting Column Contents” on page 9-36
Group by property column	To logically group data based on values for a property	“How to Group by a Property Column” on page 9-39
Use column views	To display a named subset of property columns to apply to different kinds of nodes in the Model Hierarchy pane	“Control Model Explorer Contents Using Views” on page 9-26
Add, delete, or rearrange property table columns	To customize property columns	“Organize Data Display in Model Explorer” on page 9-36

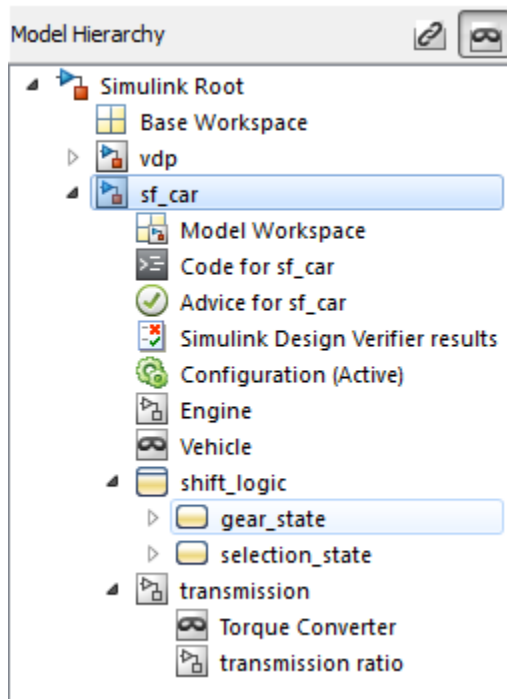
Model Explorer: Model Hierarchy Pane

In this section...

- “What You Can Do with the Model Hierarchy Pane” on page 9-9
- “Simulink Root” on page 9-10
- “Base Workspace” on page 9-10
- “Configuration Preferences” on page 9-11
- “Model Nodes” on page 9-11
- “Displaying Partial or Whole Model Hierarchy Contents” on page 9-12
- “Displaying Linked Library Subsystems” on page 9-13
- “Displaying Masked Subsystems” on page 9-14
- “Linked Library and Masked Subsystems” on page 9-14
- “Displaying Node Contents” on page 9-14
- “Navigating to the Block Diagram” on page 9-15
- “Working with Configuration Sets” on page 9-15
- “Expanding Model References” on page 9-15
- “Cutting, Copying, and Pasting Objects” on page 9-17

What You Can Do with the Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model and Stateflow chart hierarchy. Use the **Model Hierarchy** pane to navigate to the part of the model and chart hierarchy that you want to explore.



Simulink Root

The first node in the hierarchy represents the Simulink root. Expand the root node to display nodes representing the MATLAB workspace, Simulink models, and Stateflow charts that are in the current session.

Base Workspace

This node represents the MATLAB workspace. The MATLAB workspace is the base workspace for Simulink models and Stateflow charts. Variables defined in this workspace are visible to all open models and charts.

For information about exporting and importing workspace variables, see “Exporting Workspace Variables” on page 9-60 and “Importing Workspace Variables” on page 9-62.

Configuration Preferences

To display a Configuration Preferences node in the expanded Simulink Root node, enable the **View > Show Configuration Preferences** option. Selecting this node displays the preferred configuration for new models (see “Manage a Configuration Set” on page 10-12). You can change the preferred configuration by editing the displayed settings and using the **Model Configuration Preferences** dialog box to save the settings (see “Model Configuration Preferences” on page 10-28).

Model Nodes


Expanding a model or chart node in the **Model Hierarchy** pane displays nodes representing the following elements, as applicable for the models and charts you have open.

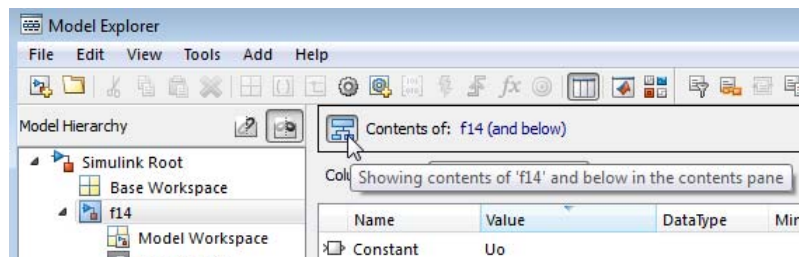
Node	Description
Model workspace	<p>For information about how to use the Model Explorer to work with model workspace variables, see the following sections:</p> <ul style="list-style-type: none"> • “Finding Variables That Are Used by a Model or Block” on page 9-52 • “Finding Blocks That Use a Specific Variable” on page 9-55 • “Editing Workspace Variables” on page 9-58 • “Exporting Workspace Variables” on page 9-60 • “Importing Workspace Variables” on page 9-62 • “Model Workspaces” on page 4-74
Configuration sets	<p>For information about adding, deleting, saving, and moving configuration sets, see “Manage a Configuration Set” on page 10-12.</p>
Top-level subsystems	<p>Expand a node representing a subsystem to display underlying subsystems, if any.</p>

Node	Description
Model blocks	Expand model blocks to show contents of referenced models (see “Expanding Model References” on page 9-15).
Stateflow charts	<ul style="list-style-type: none"> Expand a node representing a Stateflow chart to display the top-level states of the chart. Expand a node representing a state to display its substates.

Displaying Partial or Whole Model Hierarchy Contents

By default, the Model Explorer displays objects for the system that you select in the **Model Hierarchy** pane. It does not display data for child systems. You can override that default, so that the Model Explorer displays objects for the whole hierarchy of the currently selected system. To toggle between displaying only the current system and displaying the whole system hierarchy of the current system, use one of these techniques:

- Select **View > Show Current System and Below**.
- Click the **Show Current System and Below** button () at the top of the **Contents** pane.



When you select the **Show Current System and Below** option:

- The **Model Hierarchy** pane highlights in pale blue the current system and its child systems.

- After the path in the **Contents of** field, the text (and below) appears.
- The appearance of the **Show Current System and Below** button at the top of the **Contents** pane and in the **View** menu changes.
- The status bar indicates the scope of the displayed objects when you hover over the **Show Current System and Below** button.


Loading very large models for the current system and below can be slow. To stop the loading process at any time, either click the **Show Current System and Below** button or click another node in the tree hierarchy.

If you show the current system and below, you might want to change the view to better reflect the displayed system contents. For details about views, see “Control Model Explorer Contents Using Views” on page 9-26.

The setting for the **Show Current System and Below** option is persistent across Simulink sessions.

Displaying Linked Library Subsystems

By default, the Model Explorer does not display the contents of linked library subsystems in the **Model Hierarchy** pane. To display the contents of linked library subsystems, use one of these approaches:


- At the top of the **Model Hierarchy** pane, click the **Show/Hide Library Links** button (.
- From the **View** menu, select **Show Library Links**.

Library-linked subsystems are visible in the **Contents** pane, regardless of how you configure the **Model Hierarchy** pane.

Note Search does not find elements in linked library or masked subsystems that are not displayed in the **Model Hierarchy** pane.

Displaying Masked Subsystems

By default, the Model Explorer does not display the contents of masked subsystems in the **Model Hierarchy** pane. To display the contents of masked subsystems, use one of these approaches:

- At the top of the **Model Hierarchy** pane, click the **Show/Hide Masked Subsystems** button ().
- From the **View** menu, select **Show Masked Subsystems**.

Masked subsystems are visible in the **Contents** pane, regardless of how you configure the **Model Hierarchy** pane.

Linked Library and Masked Subsystems

For subsystems that are both library-linked and masked, how you set the linked library subsystems and masked subsystems options affects which subsystems appear in the **Model Hierarchy** pane, as described in the following table.

Settings	Subsystems Displayed in the Model Hierarchy Pane
Show Library Links Hide Masked Subsystems	Only library-linked, unmasked subsystems
Hide Library Links Show Masked Subsystems	Only masked subsystems that are not library-linked subsystems
Show Library Links Show Masked Subsystems	All library-linked or masked subsystems

Displaying Node Contents

Select the object in the **Model Hierarchy** pane whose contents you want to display in the **Contents** pane.

Navigating to the Block Diagram

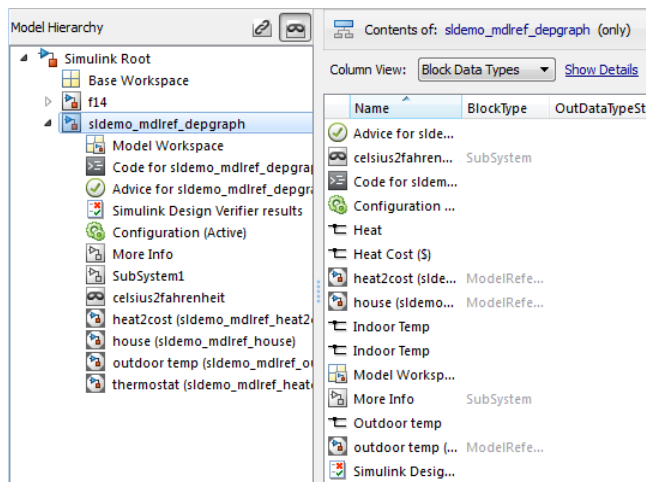
To open a graphical object (for example, a model, subsystem, or chart) in an editor window, right-click the object in the **Model Hierarchy** pane. From the context menu, select **Open**.

Working with Configuration Sets

See “Manage a Configuration Set” on page 10-12 for information about using the **Model Hierarchy** pane to perform tasks such as adding, deleting, saving, and moving configuration sets.

Expanding Model References

To browse a model that includes Model blocks, you can expand the **Model Hierarchy** pane nodes of the Model blocks. For example, the `sldemo_md1ref_depgraph` model includes Model blocks that reference other models. If you open the `sldemo_md1ref_depgraph` model and expand that model node in the **Model Hierarchy** pane, you see that the model contains several Model blocks, including `heat2cost`.



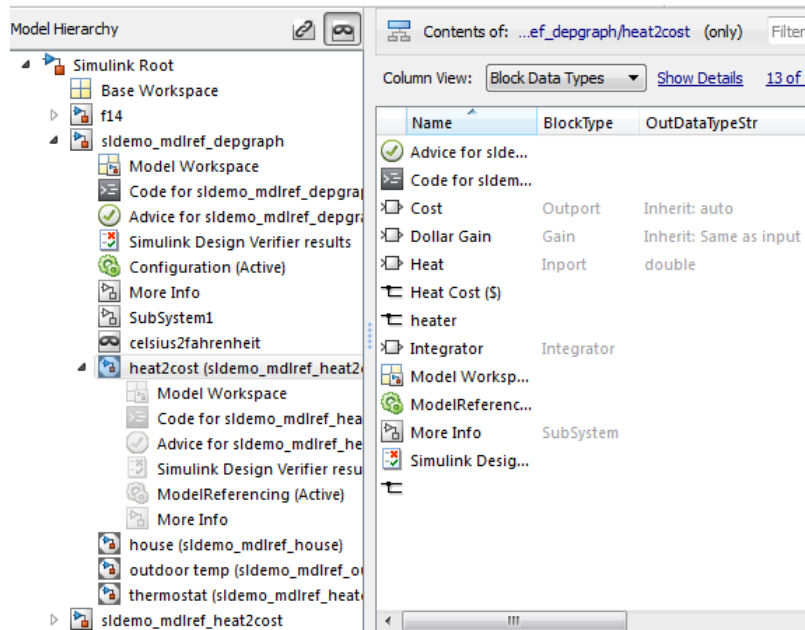
To browse a model referenced by a Model block:

- 1 Right-click the referenced model node in the **Model Hierarchy** pane.

2 From the context menu, choose **Open Model**.

- The referenced model opens.
- The **Model Hierarchy** pane indicates that you can expand the Model block node.
- The **Model Hierarchy** pane displays a separate expandable node for the referenced model (read-only).
- The **Contents** pane displays objects corresponding to the Model block node (read-only).

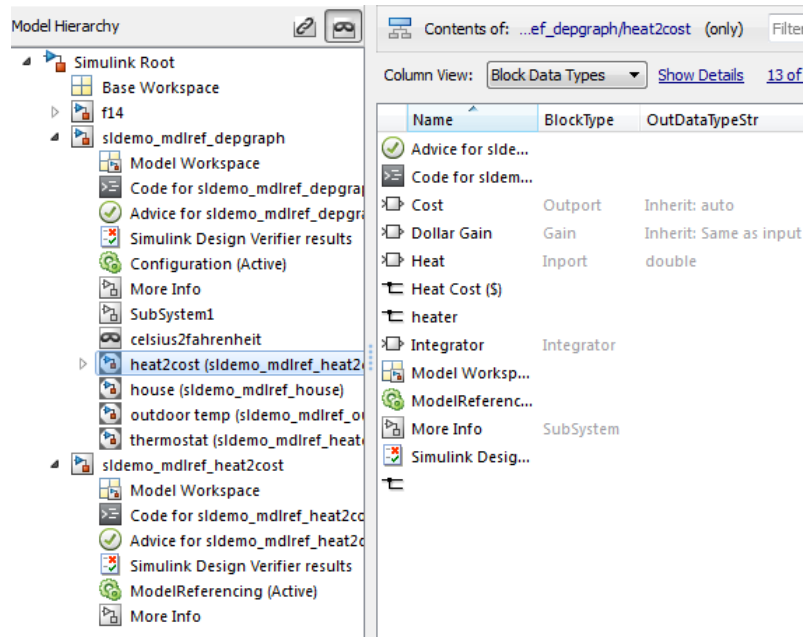
For example, if you right-click the `heat2cost` Model block node and select the **Open Model** option, the **Contents** pane displays the objects corresponding to the `heat2cost` Model block. You can expand the `heat2cost` node.



You can browse the contents of the referenced model, but you cannot edit the model objects that are underneath the Model block.

Editing the Referenced Model

To edit the referenced model, expand the referenced model node in the **Model Hierarchy** pane. For example, expand the `sldemo_md1ref_heat2cost` node:






You can now edit the properties of object in the referenced model.

For information about referenced models, see “Model Reference”.

Cutting, Copying, and Pasting Objects

To cut, copy, and paste workspace objects from one workspace into another workspace:

- 1** In the **Contents** pane, right-click on the workspace object you want to cut or copy.
- 2** From the context menu, select **Cut** or **Copy**.

- You can also cut a workspace object by selecting in the **Contents** pane **Edit > Cut** or by clicking the **Cut** button ()
 - You can also copy a workspace object by selecting **Edit > Copy** or by clicking the **Copy** button ()
- 3** If you want to paste the workspace object that you cut or copied, in the **Model Hierarchy** pane, right-click the workspace into which you want to paste the object, and select **Paste**.
- You can also paste the object by selecting **Edit > Paste** or by clicking the **Paste** button ()

You can also perform cut, copy, and paste operations by selecting an object and performing drag and drop operations.

Model Explorer: Contents Pane

In this section...

“Contents Pane Tabs” on page 9-19

“Data Displayed in the Contents Pane” on page 9-22

“Link to the Currently Selected Node” on page 9-22

“Horizontal Scrolling in the Object Property Table” on page 9-23

“Working with the Contents Pane” on page 9-24

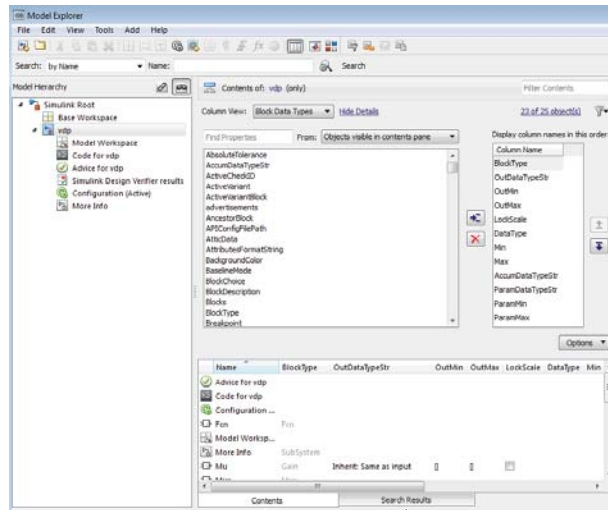
“Editing Object Properties” on page 9-25

Contents Pane Tabs

The **Contents** pane displays one of two tables containing information about models and charts, depending on the tab that you select:

- The **Contents** tab displays an object property table for the node that you select in the **Model Hierarchy** pane.
- The **Search Results** tab displays the search results table (see “Search Using Model Explorer” on page 9-63).

Optionally, you can also open a column view details section in the **Contents** pane. The following graphic shows the **Contents** pane with the column view details section opened.



Contents tab Search Results tab

To open the column view details section, click **Show Details**, at the top of the **Contents** pane.

The screenshot displays the Model Explorer Contents Pane for a model named 'vdp'. The interface is divided into two main sections:

Column View Details section: This section allows for customizing the column view. It includes a 'Column View' dropdown set to 'Block Data Types', a 'Filter Contents' field, and a '23 of 25 object(s)' indicator. A 'Find Properties' field is set to 'Objects visible in contents pane'. A list of properties is shown on the left, including AbsoluteTolerance, AccumDataTypeStr, ActiveCheckID, ActiveVariant, ActiveVariantBlock, advertisements, AncestorBlock, APIConfigFilePath, AtticData, AttributesFormatString, BackgroundColor, BaselineMode, BlockChoice, BlockDescription, Blocks, BlockType, and Breakpoint. On the right, a 'Display column names in this order:' list shows the current column order: Column Name, BlockType, OutDataTypeStr, OutMin, OutMax, LockScale, DataType, Min, Max, AccumDataTypeStr, ParamDataTypeStr, ParamMin, and ParamMax. Navigation arrows are present between the lists.

Object Property Table section: This section displays a table of model and chart object data. The table has the following columns: Name, BlockType, OutDataTypeStr, OutMin, OutMax, LockScale, DataType, and Min. The data rows are as follows:

Name	BlockType	OutDataTypeStr	OutMin	OutMax	LockScale	DataType	Min
Advice for vdp							
Code for vdp							
Configuration ...							
Fcn	Fcn						
Model Worksp...							
More Info	SubSystem						
Mu	Gain	Inherit: Same as input					
Mu...	Mu...						

At the bottom of the pane, there are tabs for 'Contents' and 'Search Results'.

The **Column view details** section provides an interface for customizing the column view (hidden by default).

The **Object property table** section displays a table of model and chart object data (open by default).

Data Displayed in the Contents Pane

In the object property table section of the **Contents** tab and in the **Search Results** tab:

- Table columns correspond to object properties (for example, Name and BlockType).
- Table rows correspond to objects (for example, blocks, and states).

The objects and properties displayed in the **Contents** pane depend on:

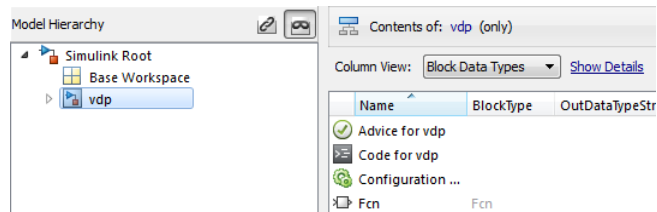
- The column view that you select in the **Contents** pane
- The node that you select in the **Model Hierarchy** pane
- The kind of object (for example, subsystem, chart, or configuration set) that you select in the **Model Hierarchy** pane
- The **View > Row Filter** options that you select

For more information about controlling which objects and properties to display in the **Contents** pane, see:

- “Control Model Explorer Contents Using Views” on page 9-26
- “Organize Data Display in Model Explorer” on page 9-36
- “Filter Objects in the Model Explorer” on page 9-46

Link to the Currently Selected Node

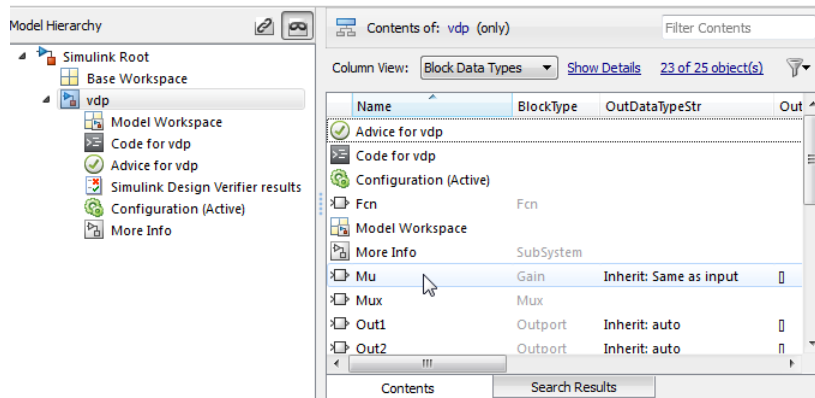
The **Contents of** link at the top left side of the **Contents** pane links to the currently selected node in the **Model Hierarchy** pane. The model data displayed in the Contents pane reflects the setting of the **Current System and Below** option. In the following example, **Contents of** links to the vdp model, which is the currently selected node.



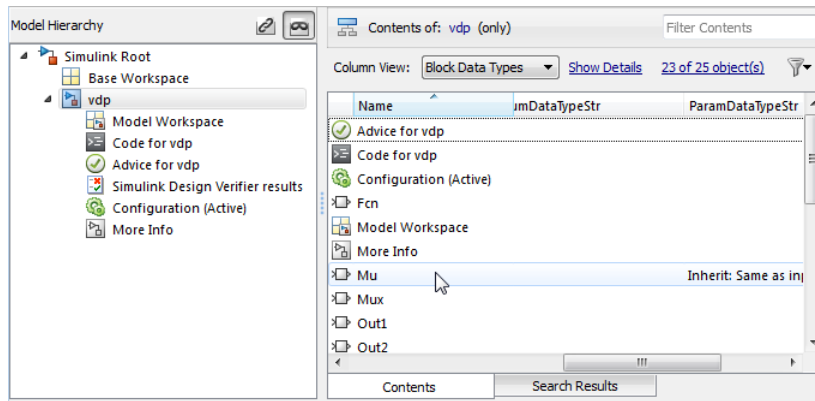
Horizontal Scrolling in the Object Property Table

The object property table displays the first two columns (the object icon and the Name property) persistently. These columns remain visible, regardless of how far you scroll to the right.

For example, the following image shows the initial display of the object property table for the vdp model. The ParamDataTypeStr column is too far to the right to be displayed.



The next image shows the results of scrolling to the right. The icon and Name columns remain visible, but now you can see the ParamDataTypeStr column.



Working with the Contents Pane

The following table summarizes the key tasks to control what is displayed in the **Contents**.

Task	Documentation
Control which kinds of objects to display.	“Using the Row Filter Option” on page 9-46
Search within the selected set of objects.	“Search Using Model Explorer” on page 9-63
Specify a set of properties to display based on the kind of node.	“Control Model Explorer Contents Using Views” on page 9-26
Group data based on unique values in a property column.	“Grouping by a Property” on page 9-37
Manage views (for example, save and export a view).	“Managing Views” on page 9-30
Add, remove, or rearrange columns.	“Organize Data Display in Model Explorer” on page 9-36
Edit object property values.	“Editing Object Properties” on page 9-25

Editing Object Properties

To open a properties dialog box for an object in the **Model Hierarchy** pane, right-click the object, and from the context menu, select **Properties**. Alternatively, click an object and from the **Edit** menu, select **Properties**.

You can change modifiable properties in the **Contents** pane (for example, a block name) by editing the displayed value. To edit a value, first select the row that contains the value, and then click the value. An edit control replaces the value (for example, an edit field for text values or a list for a range of values). For workspace variables that are arrays or structures, you can use the Variable Editor. Use the edit control to change the value of the selected property.

To assign the same property value to multiple objects in the **Contents** pane, select the objects and then change one of the selected objects to have the new property value. An edit control replaces the value with <edit>, indicating that you are doing batch editing. The Model Explorer assigns the new property value to the other selected objects, as well.

You can also change property values using the **Dialog** pane. See “Model Explorer: Property Dialog Pane” on page 9-70.

Control Model Explorer Contents Using Views

In this section...
“Using Views” on page 9-26
“Customizing Views” on page 9-29
“Managing Views” on page 9-30

Using Views

What Is a Column View?

A view in the Model Explorer is a named set of properties.

The Model Explorer uses views to specify sets of property columns to display in the **Contents** pane.

For each kind of node in the **Model Hierarchy** pane, certain properties are most relevant for the objects displayed in the **Contents** pane. For example, for a Simulink model node, such as a model or subsystem, some properties that are useful to display include:

- BlockType (block type)
- OutDataTypeStr (output data type)
- OutMin (minimum value for the block output)

Generally, a column view does not contain the total set of properties for all the objects in a node. Specifying a subset of properties to display can streamline the task of exploring and editing model and chart object properties and increase the density of the data displayed in the **Contents** pane.

What You Can Capture in a View

You can use a view to capture the following characteristics of the model information to show in the Model Explorer:

- Properties that you want to display in the **Contents** pane (see “Customizing Views” on page 9-29)

- Layout of the **Contents** pane (for example, grouping by property, the order of property columns, and sorting), as described in “Organize Data Display in Model Explorer” on page 9-36.

Use Standard Views or Customized Views

You can use views in the following ways:

- Use the standard views shipped with the Model Explorer
- Customize the standard views
- Create your own views

Automatically Applied Views

The first time you open the Model Explorer, the software automatically applies one of the standard views to the node you select in the **Model Hierarchy** pane. The Model Explorer applies a view based on the kind of node you select.

The Model Explorer assigns one of four categories of nodes in the **Model Hierarchy** pane. The Model Explorer initially associates a default view with each node category. The four node categories are:

Node Category	Kinds of Hierarchy Nodes Included	Initial Associated View
Simulink	Models, subsystems, and root level models	Block Data Types
Workspace	Base and model workspace objects	Data Objects
Stateflow	Stateflow charts and states	Stateflow
Other	Objects that do not fit into one of the first three categories; for example, configuration sets	Default

The **Column View** field at the top of the **Contents** pane displays the view that the Model Explorer is currently using.

If you select a view. In the **Contents** pane, from the **Column View** list, you can select a different view. If you select a different view, then the Model Explorer associates that view with the category of the current node. For example, suppose the selected node in the **Model Hierarchy** pane is a Simulink model, and the current view is **Data Objects**. If you change the view to **Signals**, then when you select another Simulink model node, the Model Explorer uses the **Signals** view. See “Selecting a View Manually” on page 9-28.

Selecting a View Manually

By default, the Model Explorer automatically applies a view, based on the category of node that you select and the last view used for that node. You can manually select a view from the **Column View** list that better meets your current task.

You can shift from the default mode of having the Model Explorer automatically apply views to a mode in which you must manually select a view to change views.

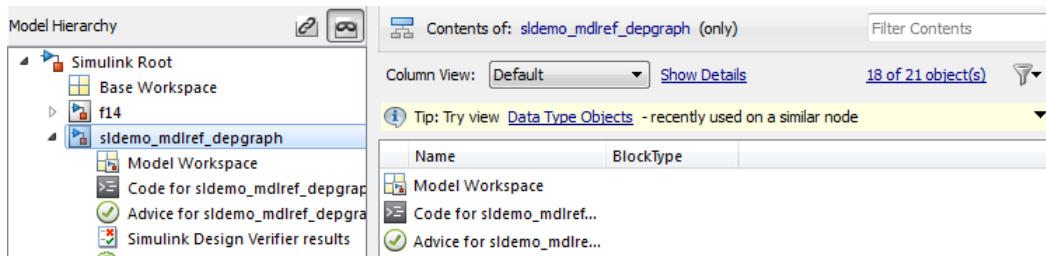
To enable the manual view selection mode:

- 1** Select **View > Column View > Manage Views**.

The View Manager dialog box opens.

- 2** In the View Manager dialog box, click the **Options** button and clear **Change View Automatically**.

In the manual view selection mode, if you switch to a different kind of node in the **Model Hierarchy** pane that has a different view associated with it, the **Contents** pane displays a yellow informational bar suggesting a view to use.



Tip interface. The tip interface appears immediately above the object property table.

The tip does not appear if you use automatic view selection.

To hide the currently displayed tip, from the menu button on the right-hand side of the tip bar, select **Hide This Tip**.

The tip interface displays a link for changing the current view to a suggested view. To choose the suggested view displayed in the tip bar, click the link.

Initially, the suggested view is the default view associated with a node. If you associate a different view with a node category, then the tip suggests the most recently selected view when you select similar nodes.

To change from manual specification of views to automatic specification, from the tip interface, select the down arrow and then the **Change View Automatically** menu item.

Customizing Views

If a standard view does not meet your needs, you can either modify the view or create a new view.

You can customize the object property table represented by the current view in several ways, as described in these sections:

- “Adding Property Columns” on page 9-42
- “Hiding or Removing Property Columns” on page 9-43
- “Changing the Order of Property Columns” on page 9-41

How the Model Explorer Saves Your Customizations

As you modify the object property table, you change the current view definition.

The Model Explorer saves the following changes to the object property table as part of the column view definition:

- Grouping by property
- Sorting in a column
- Changing the order of property columns
- Adding a property column
- Hiding and removing property columns

When you change from one view to another view, the Model Explorer saves any customizations that you have made to the previous view.

For example, suppose you use the **Block Data Types** view and you remove the **LockScale** property column. If you then switch to use the **Data Objects** view, and later use the **Block Data Types** view again, the **Block Data Types** view no longer includes the **LockScale** column that you deleted.

At the end of a Simulink session, the Model Explorer saves the view customizations that you made during that session. When you reopen the Model Explorer, Simulink uses the customized view, reflecting any changes that you made to the view in the previous session.

Managing Views

If a standard view does not meet your needs, you can either modify the view or create a new view. See “Customizing Views” on page 9-29.

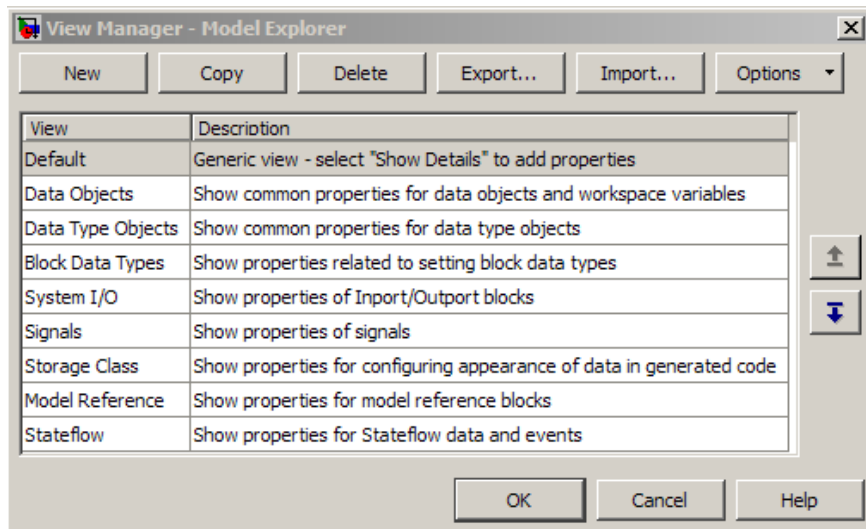
You can manage views (for example, create a new view or export a view) using the View Manager dialog box.

Opening the View Manager Dialog Box

To open the View Manager dialog box, select the **Manage Views** option from either:

- The **View > Column View** menu
- The options listed when you click the **Options** button in the column view details section

The View Manager dialog box displays a list of defined views and provides tools for you to manage views.



You can manage views in several ways, including:

- “Creating a New View” on page 9-32
- “Deleting Views” on page 9-33
- “Reordering Views” on page 9-33
- “Exporting Views” on page 9-33
- “Importing Views” on page 9-34
- “Resetting Views to Factory Settings” on page 9-34

Creating a New View

To create a new view that has a new name, you can use one of these approaches:

- Copy an existing view, rename it, and customize the view.
- Create a completely new view.

After you create a new view, you can customize the view as described in “Customizing Views” on page 9-29.

Copying and renaming an existing view. You can build a new view by copying an existing view, renaming it, and optionally customizing the renamed view. In the View Manager dialog box:

- 1 Select the view that you want to use as the starting point for your new view.
- 2 Click the **Copy** button.

A new row appears at the bottom of the View Manager table of views. The new row contains the name of the view you copied, followed by a number in parentheses. For example, if you copy the `Stateflow` view, the initial name of the copied view is `Stateflow (1)`.

Creating a completely new view. To create a completely view, in the View Manager dialog box, click the **New** button. A new view row appears at the bottom of the View Manager dialog box list of views.

Naming and describing a new view. Once you create a view, you can name the view and provide a description of the view:

- 1 Double-click `New View` in the left column of the table of views and replace the text with a name for the view.
- 2 Double-click `Description` in the table and replace the text with a description of the view.
- 3 Click **OK**.

Deleting Views

To delete a view from the **Column View** list of views:

- 1** In the View Manager dialog box, select one or more views that you want to remove from the list.
- 2** Click the **Delete** button or the **Delete** key.
- 3** Click **OK**.

Deleting a view using the View Manager dialog box permanently deletes that view from the Model Explorer interface.

If you think you or someone else might want to use a view again, consider exporting the view before you delete it (see “Exporting Views” on page 9-33).

Reordering Views

To change the position of a view in the **Column View** list, in the View Manager dialog box:

- 1** Select one or more views that you wish to move up or down one row in the table of views.
- 2** Click the up or down arrow buttons to the right of the table of views. Repeat this step until the view appears where you want it to be in the table.
- 3** Click **OK**.

Exporting Views

To export views that you or others can then import, in the View Manager dialog box:

- 1** In the View Manager dialog box, select one or more views that you want to export.
- 2** Click the **Export** button.

An Export Views dialog box opens, with check marks next to the views that you selected.

3 Click **OK**.

An Export to File Name dialog box opens.

4 Navigate to the folder to which you want to export the view.

By default, the Model Explorer exports views to the MATLAB current folder.

5 Specify the file name for the exported view.

The file format is `.mat`.

6 Click **OK**.

Importing Views

To import view files from another location for use by the Model Explorer:

1 In the View Manager dialog box, click the **Import** button.

The Select `.mat` File to Import dialog box opens.

2 Navigate to the folder from which you want to import the view.

3 Select the MAT-file containing the view that you want to import and then click **Open**.

A confirmation dialog box opens. Click **OK** to import the view.

The imported view appears at the bottom of the **Column View** list of views.

The Model Explorer automatically renames the view if a name conflict occurs.

Resetting Views to Factory Settings

You can reset (restore) the original definition of a specific standard view (that is, a view shipped with the Model Explorer) if that view is the current view. To do so, click the **Options** button in the column view details section and select **Reset This View to Factory Settings**.

To reset the factory settings for *all* standard views in one step, in the View Manager dialog box, click the **Options** button and select **Reset All Views to Factory Settings**.

Note When you reset all views, the Model Explorer removes all the custom views you have created. Before you reset views to factory settings, export any views that you will want to use in the future.

Organize Data Display in Model Explorer

In this section...

- “Layout Options” on page 9-36
- “Sorting Column Contents” on page 9-36
- “Grouping by a Property” on page 9-37
- “Changing the Order of Property Columns” on page 9-41
- “Adding Property Columns” on page 9-42
- “Hiding or Removing Property Columns” on page 9-43
- “Marking Nonexistent Properties” on page 9-45

Layout Options

You can control how the object property table and **Search Results** pane organize the layout of property information by:

- Sorting column contents
- Grouping by a property
- Changing the order of property columns
- Adding a property column
- Hiding and removing property columns

Sorting Column Contents

To sort the column contents in ascending order, click the heading of the property column. A triangle pointing up appears in the column heading. To change the order from ascending to descending, or from descending to ascending, click the heading of the column again.

For example, if properties are in ascending order, based on the Name property (the default), click the heading of the Name column to display objects by name, in descending order.

By default, the **Contents** pane displays its contents in ascending order, based on the name of the object. Objects that have no values in any property columns appear at the end of the object property table.

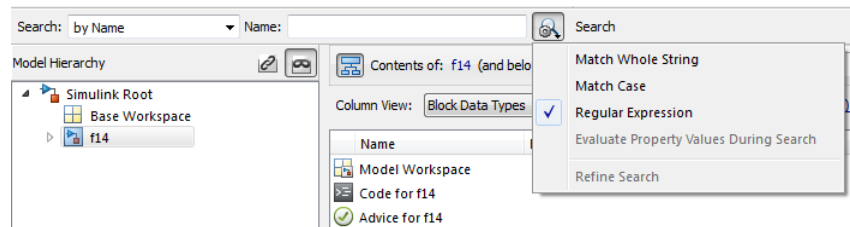
Note When you group by property, the Model Explorer applies sorting of column contents within each group.

Grouping by a Property

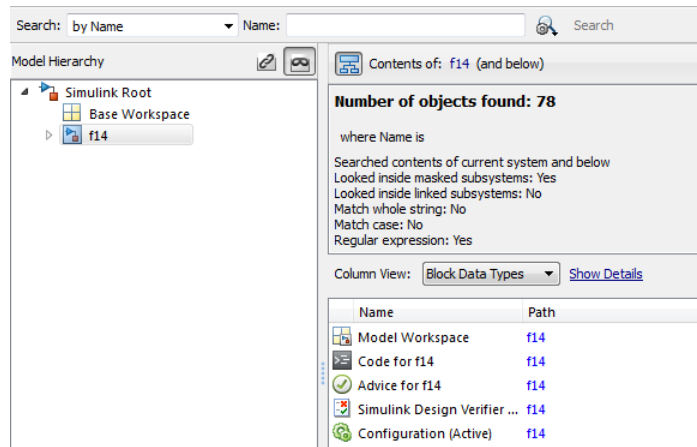
Organizing Contents by Property Values

When you explore a model, you might want to focus on all objects with the same property value. One approach is to group data by a property column.

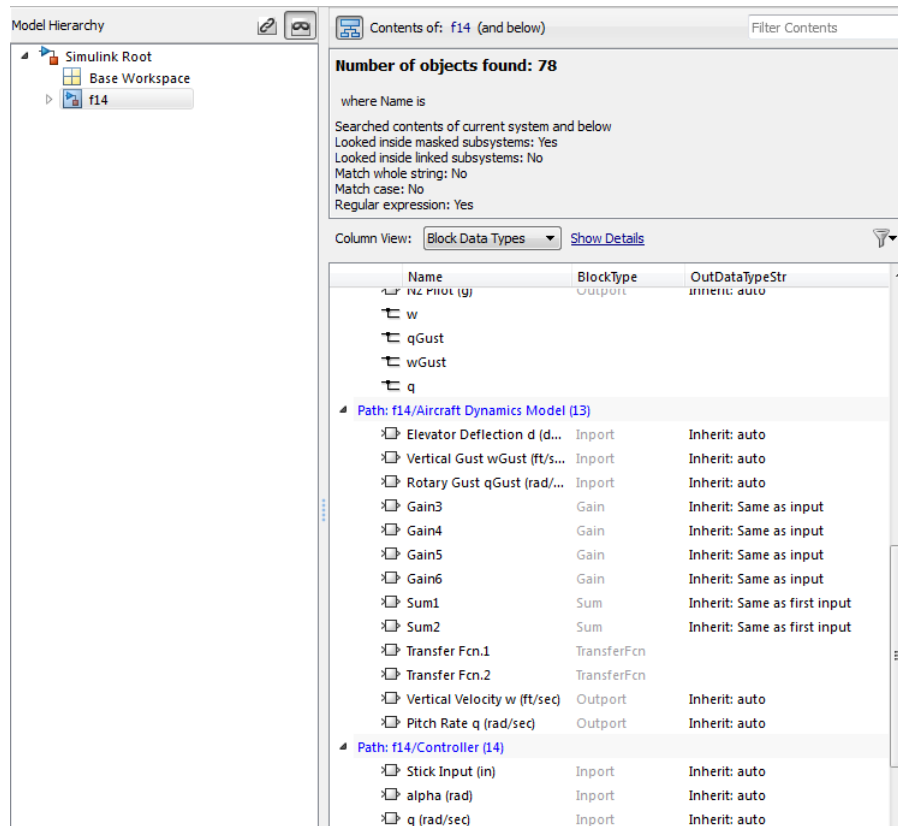
For example, suppose that you want to see all of the blocks in the f14 model. You could perform the following search.



The search results obscure the whole path name for lower-level nodes:



By grouping on the Path property column, you see the whole path for lower-level nodes.



You can also collapse groups to focus on specific portions of a model.

How to Group by a Property Column

To group by a property:

- 1 In the object property table, right-click the column heading of the property by which you want to group contents.

You can group by object icons, such as a block icon (□), which represents a type of object. Right-click the empty column heading in the first column.

- 2 From the context menu, select the **Group By This Column** menu item.

Sorting with Grouped Data

When you group by property, the Model Explorer applies sorting of column contents within each group.

Expanding and Collapsing Grouped Data

By default, Model Explorer displays groups in expanded form. That is, all the objects in each group are visible. You can collapse and expand groups.

- To collapse the contents of a group, click the minus sign icon for that group.
- To expand a group, click the plus sign.
- To collapse or expand all the groups, right-click anywhere in the object property table and select either the **Collapse All Groups** menu item (**Shift+C**) or **Expand All Groups** menu item (**Shift+E**).

Hiding the Group Column

By default, the property column that you use for grouping appears in the property table. That property also appears in the top row for each group.

To hide the group column in the property table, use one of the following approaches:

- From the **View** menu, clear the **Show Group Column** check box.
- Right-click a column heading in the property table and clear the **Show Group Column** check box.

Persistence of Grouped Data Settings

If you group by a property, that grouping is saved as part of the view definition.

When you select a different node in the **Model Hierarchy** pane, the contents for the new node are grouped by that same property. However, all groups are expanded, even if you had collapsed all groups before switching nodes.

Grouping Search Results

You can use grouping to organize the **Search Results** pane. The grouping that you apply to the **Search Results** pane also applies to the object property table, if that property is in the table. If the search results include a property that is not in the object property table, and you group on that property, then the Model Explorer removes the grouping setting that was in effect in the object property table.

Changing the Order of Property Columns

Object Icon and Name Columns Are Always First

The first two columns of every object property table are the object icon column (the column with a blank column heading) and the Name property column. You cannot hide, remove, or change the location of the first two columns.

How to Change the Order of Property Columns

To change the order of property columns in the object property table, use one of these approaches:

- In the object property table, select a column heading and drag it to a new location in the table.

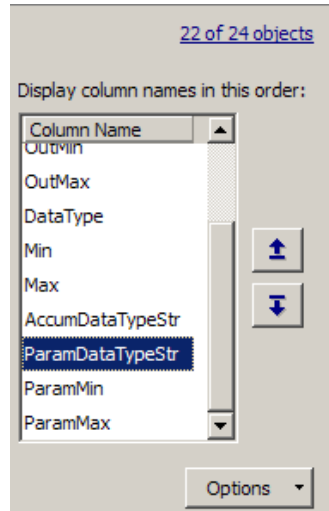
This approach avoids opening the column view details section and makes it easier to move a column a short distance to the right or left.



- In the column view details section, select one or more property columns and move them up or down in the list, using the arrow buttons to the right of the list.

This approach allows you to move several property columns in one step, but it moves the selected columns right or left by only one column at a time.

To move a property column by using the view details interface:

- 1 In the **Display column names in this order** list on the right side of the column view details section, select one or more property columns that you want to move.

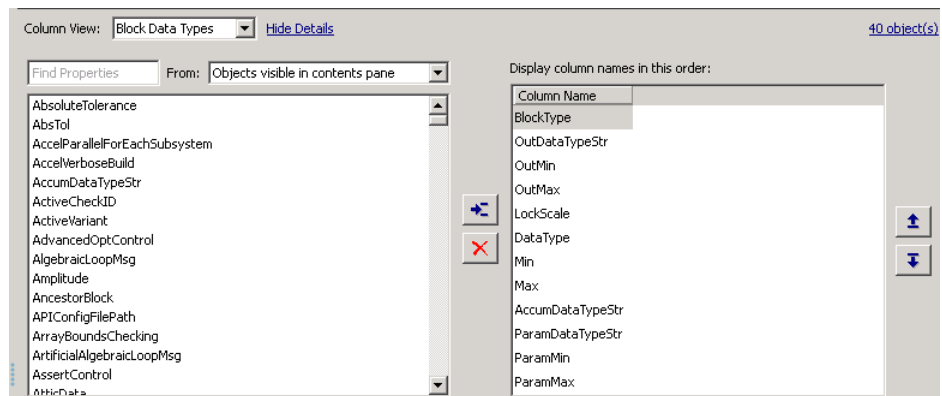



- 2 Click the **Move column left in view** button () or the **Move column right in view** button ()

Adding Property Columns

To add property columns to a view:

- 1 If you do not have the column view details section of the **Contents** pane already open, then at the top of the **Contents** pane, select **Show Details**.



- 2 In the list of properties on left side of the column view details section, select one or more properties that you want to add.
 - The list displays property names in alphabetical order. You can use the **Find Properties** search box in the column view details section to search for properties that contain the text string that you enter. You can specify the scope of the search with the **From** list to the right of the search box.
- 3 In the list of column names on the right side, select the property column that you want to be to the left of the property columns you insert.
- 4 Click the **Display property as column in view** button ()

Adding a Path Property Column

The Model Explorer provides a shortcut for adding a Path property column to a view. To add a Path property column:

- 1 Right-click the column heading in the object property table to the right of which you want to insert a Path column.
- 2 From the context menu, select **Insert Path**.

Hiding or Removing Property Columns

You can choose between two approaches to hide (remove) a property column from the object property table. Hiding and removing a column both have the same result. You can:

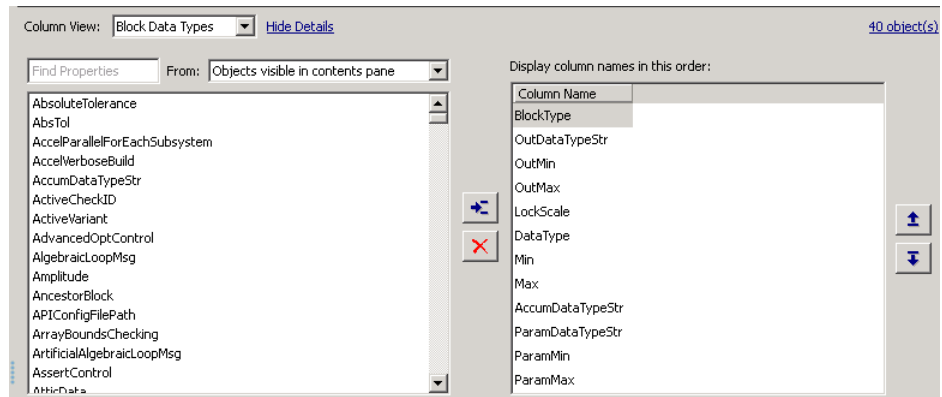
- Hide a column using the context menu for a column heading. This approach avoids needing to open the column view details section.
- Remove a column using the column view details interface. This approach allows you to delete several properties in one step.


Hiding a Column Using the Column Heading Context Menu

- 1 Right-click the column heading of the column that you want to remove.
- 2 From the context menu, select **Hide**.

Removing a Column Using the Column View Details Interface

- 1 If you do not have the column view details section of the **Contents** pane already open, then at the top of the **Contents** pane, select **Show Details**.



- 2 In the column view details section of the **Contents** pane, in the **Display column names in this order** list, select one or more properties that you want to remove.
- 3 Click the **Remove column from view** button () or the **Delete** key.

Inserting Recently Hidden or Removed Columns

The Model Explorer maintains a list of columns you hide or remove for each view during a Simulink session.

To add a recently hidden or removed column back into a view:

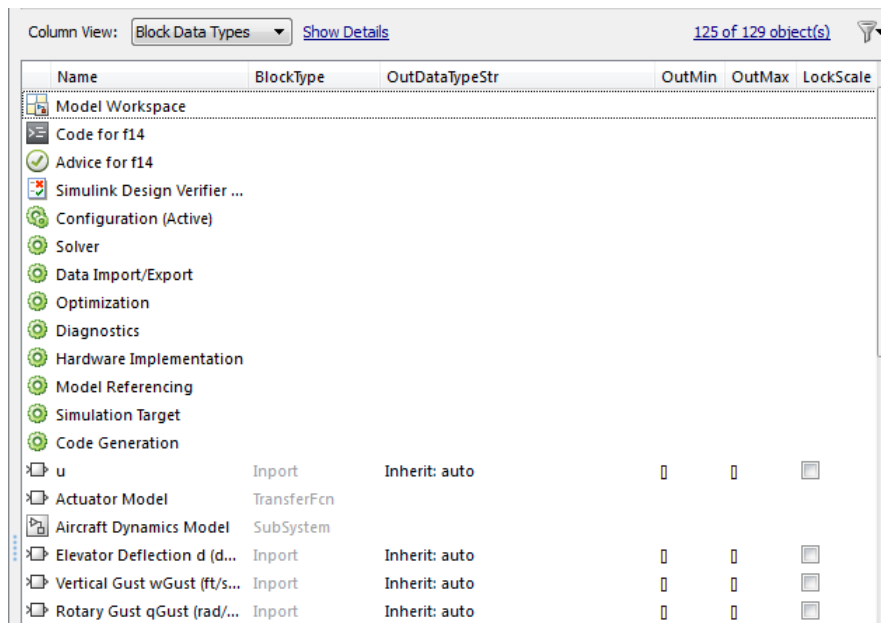
- 1 Right-click the column heading of the column to the right of which you want to insert a recently hidden column.
- 2 From the context menu, select **Insert Recently Hidden Columns**.
- 3 Select the column that you want to insert.

See “Hiding or Removing Property Columns” on page 9-43.

Marking Nonexistent Properties

Usually, some of the properties that the **Contents** pane displays do not apply to all the displayed objects (in other words, some objects do not have values set). By default, the Model Explorer displays a dash (–) to mark properties that do not have a value.

If you want the Model Explorer to display a blank (instead of the default dash) in property cells that have no values, clear the **View > Show Nonexistent Properties as “–”** option. The **Contents** pane looks similar to the following graphic:



Filter Objects in the Model Explorer

In this section...

“Controlling the Set of Objects to Display” on page 9-46

“Using the Row Filter Option” on page 9-46

“Filtering Contents” on page 9-48

Controlling the Set of Objects to Display

Two techniques that you can use to control the set of objects that the **Contents** pane displays are:

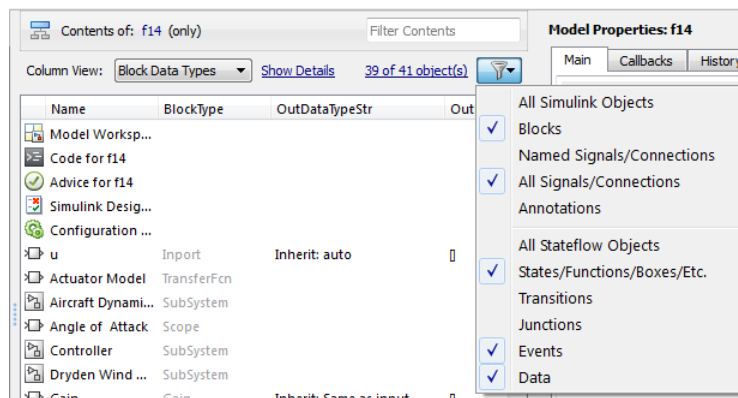
- Using the Row Filter option
- Filtering contents

For a summary of other techniques, see “Focusing on Specific Elements of a Model or Chart” on page 9-7.

Using the Row Filter Option

You can filter the kinds of objects that the **Contents** pane displays:

- 1 Open the **Row Filter** options menu. In the Model Explorer, at the top-right corner of the **Contents** pane, click the **Row Filter** button.

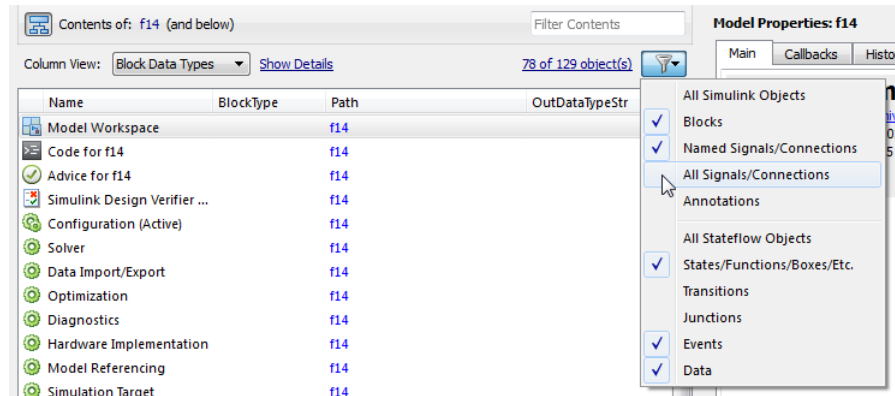


An alternative way to open the Row Filter menu is to select **View > Row Filter**.

By default, the **Contents** pane displays these kinds of objects for the selected node:

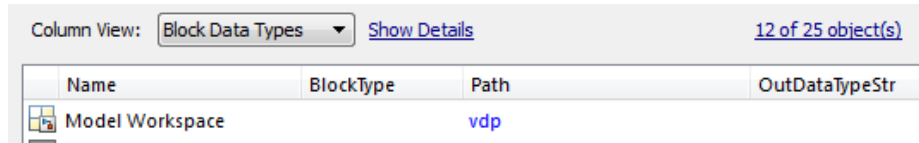
- Blocks
- Signals and connections
- Stateflow states, functions, and boxes
- Stateflow events
- Stateflow data

- 2 Clear the kinds of objects that you do not want to display in the **Contents** pane, or enable any cleared options to display more kinds of objects. For example, clear **All Signals/Connections** to prevent the display of signal and connection objects in the **Contents** pane.



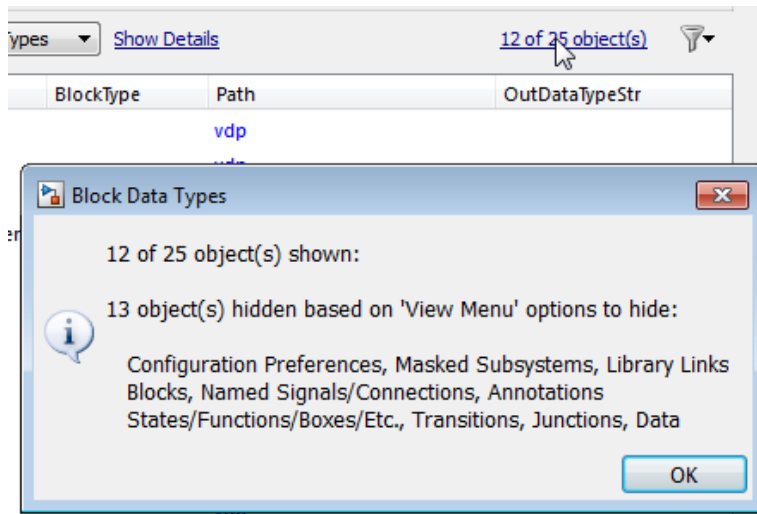
Object Count

The top-right portion of the **Contents** pane includes an object counter, indicating how many objects the **Contents** pane is displaying.



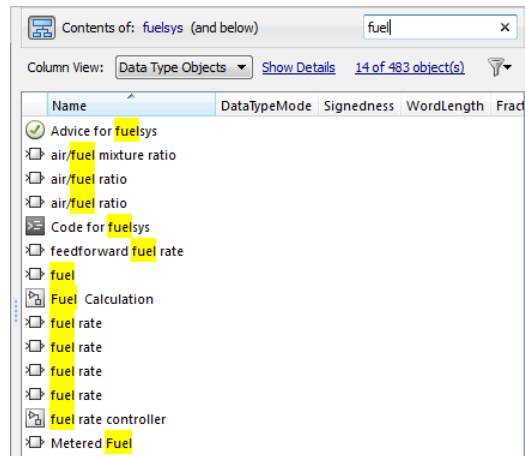
When you use the **Row Filter** option to filter objects, the object count indicator reflects that the **Contents** pane displays a subset of all the model and chart objects.

To view an explanation of the current object count, click the object count link (for example, 12 of 25 objects). That link displays a pop-up information box:



Filtering Contents

To refine the display of objects that are currently displayed in the **Contents** pane, you can use the **Filter Contents** text box at the top of the **Contents** pane to specify search strings for filtering a subset of objects.



Using the **Filter Contents** text box can help you to find specific objects within the set of objects, based on a particular object name, property value, or property that is of interest to you. For example, if you enter the text string `fuel` in the **Filter Contents** edit box, the Model Explorer displays results similar to those shown above. The results highlight the text string that you specified.

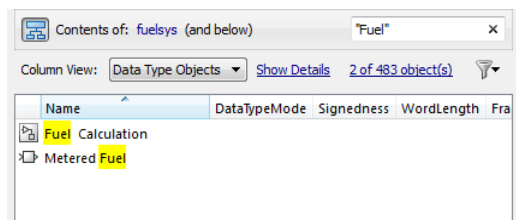
Specifying Filter Text Strings

As you enter text in the **Filter Contents** text box, the Model Explorer performs a dynamic search, displaying results that reflect the text as you enter it.

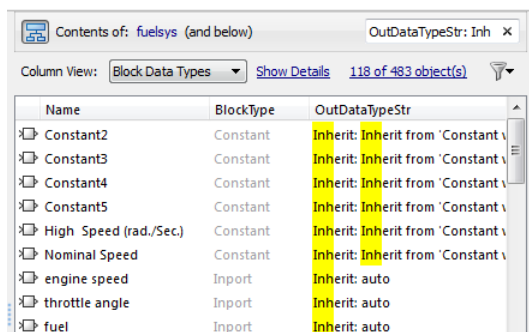
The text strings you enter must be in the format consistent with the guidelines described in the following sections.

Case Sensitivity. By default, the Model Explorer ignores case as it performs the filtering.

To specify that you want the Model Explorer to respect case sensitivity for a text string that you enter, put that text string in quotation marks. For example, if you want to restrict the filtering to display only objects that include the text `Fuel` (with a capital F), enter `"Fuel"` (with quotation marks).



Specifying Property Values. To restrict the filtering to apply to values for a specific property, specify the property name followed by a colon (:) and then the value. For example, to filter the contents to display only objects whose `OutDataTypeStr` property has a value that includes `Inherit`, enter `OutDataTypeStr: Inherit` (alternatively, you could put the whole string in quotation marks to enforce case sensitivity):



Wildcards and MATLAB Expressions Not Supported. The Model Explorer does not recognize wildcard characters, such as an asterisk (*), as having any special meaning. For example, if you enter `fuel*` in the **Filter Contents** text box, you get no results, even if several objects contain the text string `fuel`.

Also, if you specify a MATLAB expression, in the **Filter Contents** text box, the Model Explorer interprets that string as literal text, not as a MATLAB expression.

Clearing the Filtered Contents

To redisplay the object property table as it appeared before you filtered the contents, click the **X** in the **Filter Contents** text box.

Filtering Removes Grouping

If you have set up grouping on a column, then when you filtering contents, the Model Explorer does not retain that grouping.

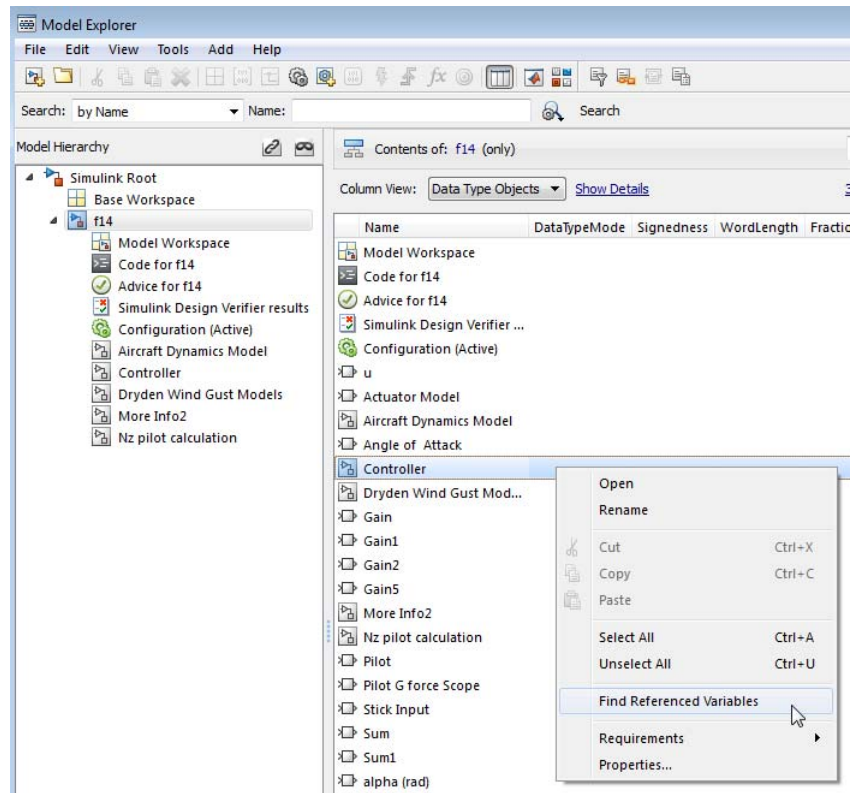
Workspace Variables in Model Explorer

In this section...
“Finding Variables That Are Used by a Model or Block” on page 9-52
“Finding Blocks That Use a Specific Variable” on page 9-55
“Finding Unused Workspace Variables” on page 9-56
“Editing Workspace Variables” on page 9-58
“Exporting Workspace Variables” on page 9-60
“Importing Workspace Variables” on page 9-62

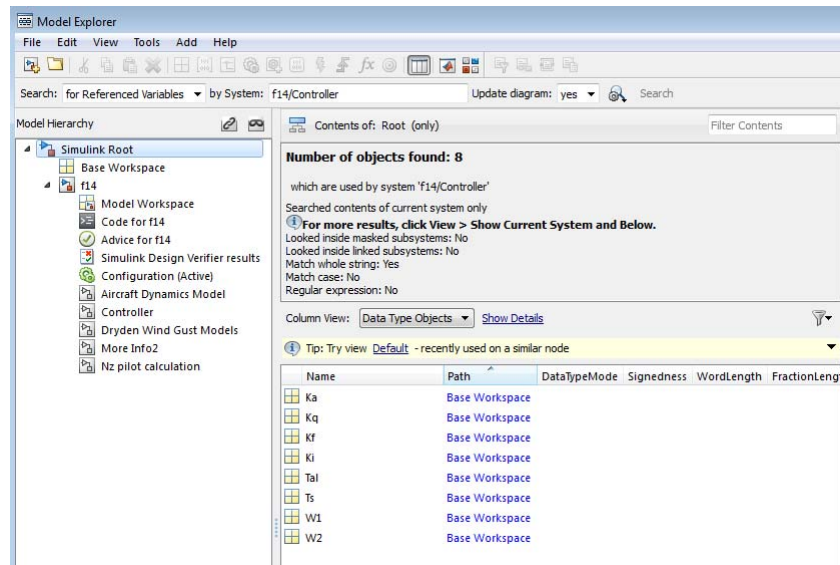
Finding Variables That Are Used by a Model or Block

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 9-55). Also, you can export variables from the returned set of variables. For details, see “Exporting Workspace Variables” on page 9-60.

Finding Blocks That Use a Specific Variable

You can use the Model Explorer to get a list of blocks that use a specific workspace variable. One way to get that list of blocks is to right-click a variable in the **Contents** pane and select the **Find Where Used** menu item. For example:

- 1 Open the f14 model.
- 2 Open the Model Explorer.
- 3 In the **Model Hierarchy** pane, select the Base Workspace node.
- 4 In the **Contents** pane, right-click the Mq variable and select the **Find Where Used** menu item.
- 5 In the Hierarchy dialog box that appears, select f14. The Model Explorer displays output similar to this:

The screenshot shows the Model Explorer interface with the following components:

- Model Hierarchy** pane on the left, showing a tree structure with 'Simulink Root' expanded to 'Base Workspace', which is further expanded to 'f14'. Under 'f14', several sub-items are listed, including 'Model Workspace', 'Code for f14', 'Advice for f14', 'Simulink Design Verifier results', 'Configuration (Active)', 'Aircraft Dynamics Model', 'Controller', 'Dryden Wind Gust Models', 'More Info2', and 'Nz pilot calculation'.
- Contents** pane on the right, titled 'Contents of: f14 (and below)'. It displays the search results for the variable 'Mq'.

The search results in the Contents pane are as follows:

Number of objects found: 2

which use variable 'Mq' from workspace 'base workspace'

Searched contents of current system and below
 Looked inside masked subsystems: No
 Looked inside linked subsystems: No
 Match whole string: Yes
 Match case: No
 Regular expression: No

Column View: Data Type Objects Show Details

Name	Path	DataTypeMode	Signedness	WordLength
Transfer Fcn.1	f14/Aircraft Dynamics Model			
Gain1	f14			

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:

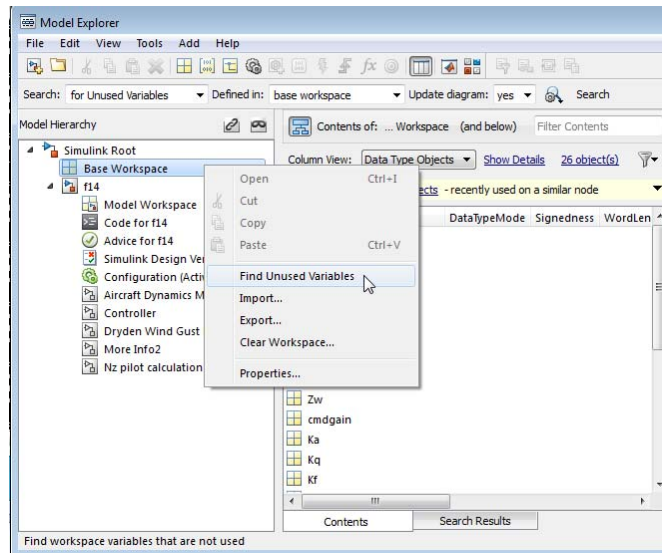
- In the search bar, select the for **Variable Usage** search type option.
- In the **Search Results** pane, right-click a variable and select the **Find Where Used** menu item.

If you do not find the variable that you are looking for because that variable is not in the currently selected system, try selecting **View > Show Current System and Below** and repeat the search.

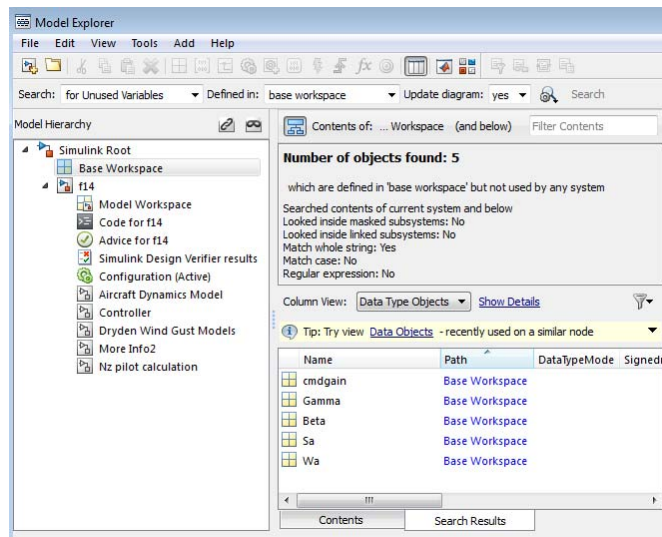
Finding Unused Workspace Variables

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:



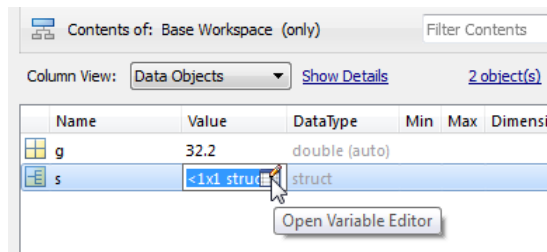
The `Simulink.findVars` function provides additional options for returning information about unused workspace variables that is not available from the Model Explorer or Simulink Editor.

Editing Workspace Variables

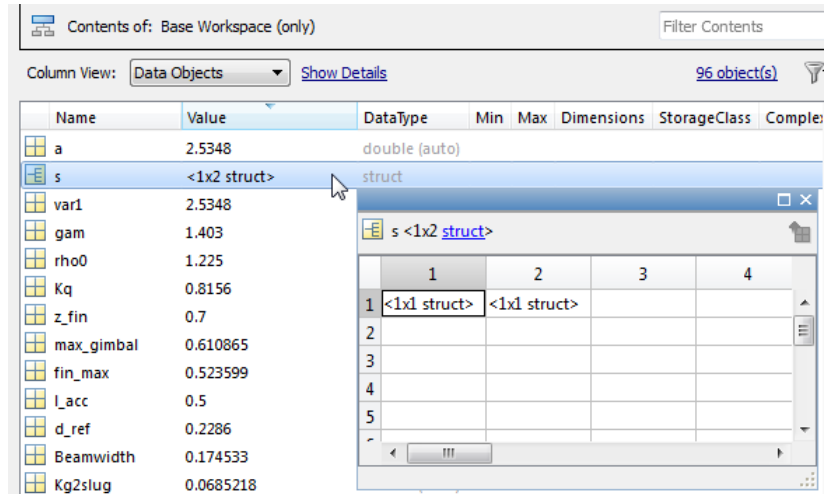
In the Model Explorer **Contents** pane, you can use the Variable Editor to edit variables from the MATLAB workspace or model workspace. The Variable Editor is available for editing large arrays and structures.

To open the Variable Editor for a variable that is an array or structure:

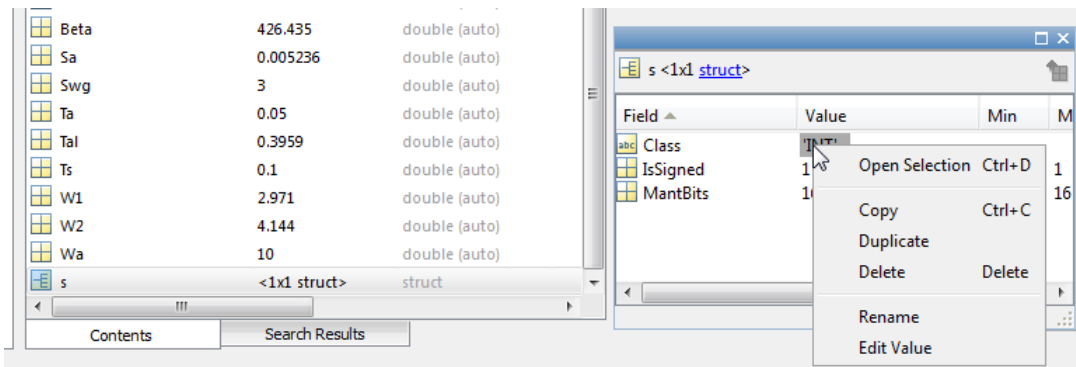
- 1 Click the Value cell for the variable.
- 2 Select the Variable Editor icon.



The Variable Editor opens:



Right-click in an edit box to open a context menu with several editing options:



You can resize and move the Variable Editor. The **Contents** pane reflects the edits that you make in the Variable Editor.

When you finish editing, the Variable Editor closes when you click the **Close** button in the upper right corner of the editor or when you click outside of the editor.

Exporting Workspace Variables

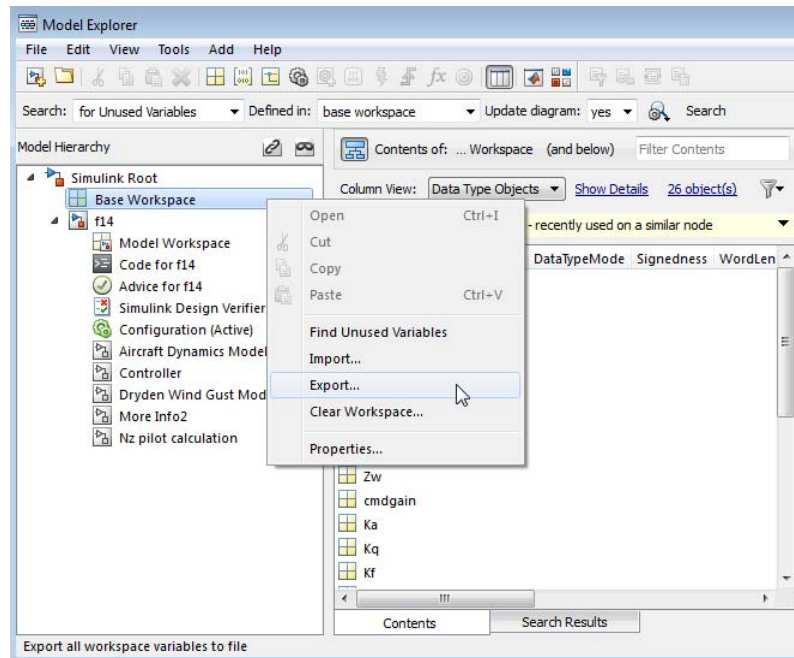
You can export (save) a set of variables listed in the Model Explorer, exporting either individual variables or all the variables in the base or model workspace.

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 9-52.

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.
 - 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 “Grouping by a Property” on page 9-37.

- 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.

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.

Search Using Model Explorer

In this section...

“Searching in the Model Explorer” on page 9-63

“The Search Bar” on page 9-64

“Show and Hide the Search Bar” on page 9-64

“Search Bar Controls” on page 9-64

“Search Options” on page 9-66

“Run a Search” on page 9-68

“Refine a Search” on page 9-69

Searching in the Model Explorer

Use the Model Explorer search bar to search for specific objects from the node you select in the **Model Hierarchy** pane.

The Model Explorer displays search results in the **Search Results** tab of the **Contents** pane.

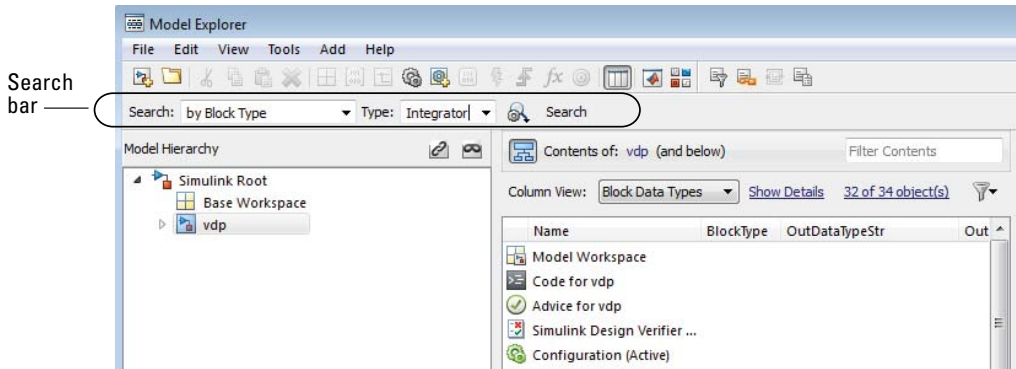
The search results appear in a table that is similar to the object property table in the **Contents** tab. The search results table uses the current column view (the object property table) definition as a starting point, and adds relevant properties that are not already included in the current view. Any additional property columns added to the **Search Results** pane do not affect the view definition.

If you modify the property columns in the search results table that also appear in the property table view, the changes you make affect both tables. For example, if you hide **OutMax** column in the search results table, and the **OutMax** column was also in the object property view table, then the **OutMax** column is hidden in both tables. However, if in the search results table you reorder where the **Complexity** column appears, if the view does not include the **Complexity** property, then that change to the search results table does not affect the view.

You can edit property values in the search results table.

The Search Bar

The search bar appears at the top of the Model Explorer window.

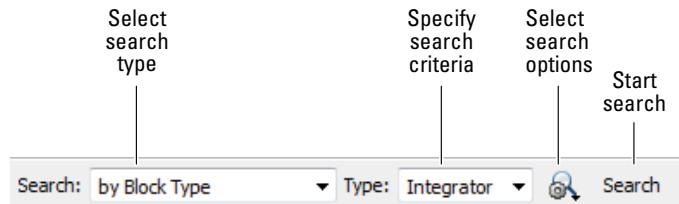


Show and Hide the Search Bar

By default, the search bar is visible. To show or hide the search bar, select or clear the **View > Toolbars > Search Bar** option.

Search Bar Controls

The search bar includes the following controls:



Search Type

Use the **Search Type** control to specify the type of objects or properties to include in the search.

Search Type Option	Description
by Name	Searches a model or chart for all objects that have the specified string in the name of the object. See “Search Strings” on page 9-68.
by Property Name	Searches for objects that have a specified property. Specify the target property name from a list of properties that objects in the search domain can have.
by Property Value	Searches for objects with a property value that matches the value you specify. Specify the name of the property, the value to be matched, and the type of match (for example, equals, less than, or greater than). See “Search Strings” on page 9-68.
by Block Type	Searches for blocks of a specified block type. Select the target block type from the list of types contained in the currently selected model.
by Stateflow Type	Searches for Stateflow objects of a specified type.
for Variable Usage	Searches for blocks that use variables defined in a workspace. Select the base workspace or a model workspace (model name) and, optionally, the name of a variable. See “Search Strings” on page 9-68.
for Referenced Variables	Searches for variables that a model or block uses. Specify the name of the model or block in the by System field. The model or block must be in the Model Hierarchy pane.

Search Type Option	Description
for Unused Variables	Searches for variables that are defined in a workspace but not used by any model or block. Select the name of the workspace from the drop-down list for the in Workspace field.
for Library Links	Searches for library links in the current model.
by Class	Searches for Simulink objects of a specified class.
for Fixed Point Capable	Searches a model for all blocks that support fixed-point computations.
for Model References	Searches a model for references to other models.
by Dialog Prompt	Searches a model for all objects whose dialogs contain the prompt you specify. See “Search Strings” on page 9-68.
by String	Searches a model for all objects in which the string you specify occurs. See “Search Strings” on page 9-68.

Search Options

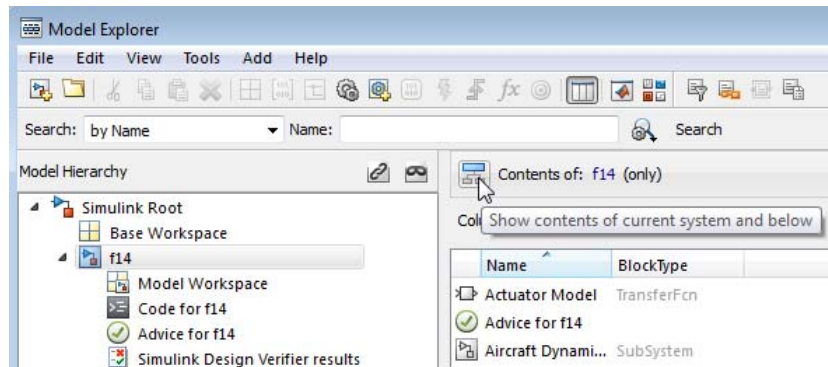
Use the **Search Options** control to specify the scope and how to apply search strings.

Search Option	Description
Match Whole String	Do not allow partial string matches (for example, do not allow sub to match substring).
Match Case	Considers case when matching strings (for example, Gain does not match gain).

Search Option	Description
Regular Expression	Considers a string to be matched as a regular expression.
Evaluate Property Values During Search	Applies only for searches by property value. If enabled, the option causes the Model Explorer to evaluate the value of each property as a MATLAB expression and compare the result to the search value. If this option is disabled (the default), the Model Explorer compares the unevaluated property value to the search value.
Refine Search	Initiates a secondary search that provides additional search criteria to refine the initial search results. The second search operation searches for objects that meet both the original and the new search criteria (see “Refine a Search” on page 9-69).

By default, the Model Explorer searches for objects in the system that you select in the Model Hierarchy pane. It does not search in child systems. You can override that default, so that the Model Explorer searches for objects in the whole hierarchy of the currently selected system. To toggle between searching only in the current system and searching in the whole system hierarchy of the current system, use one of these techniques:

- Select **View > Show Current System and Below**.
- Click the **Show Current System and Below** button at the top of the **Contents** pane.



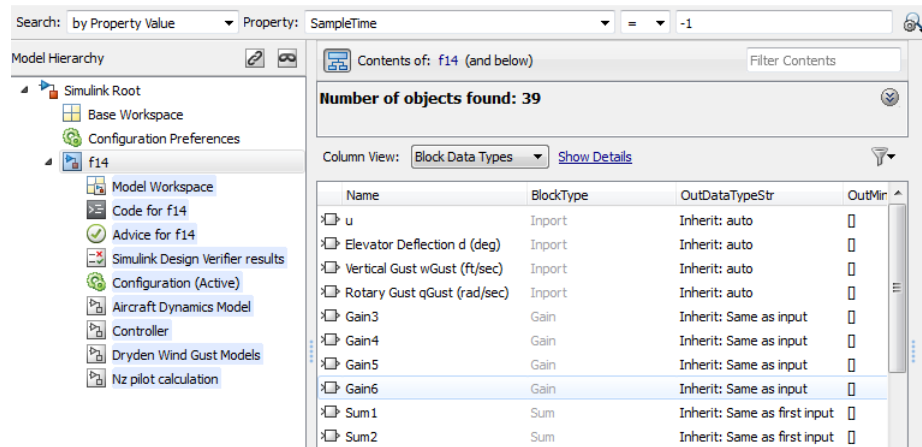
Search Strings


By default, search strings are case-insensitive and are treated as regular expressions.

By default, the search allows partial string matches. You cannot use wildcard characters in search strings. For example, if you enter `*1` as a name search string, you get no search results unless there is an item whose name starts with the two characters `*1`. If there is an `out1` item, the search results do not include that item.

Run a Search


To start the search, click the **Search** button. The Model Explorer displays the results of the search in the **Search Results** pane.



To view a summary of the search options that you used (such as search criteria), click the **Expand Search Results** button .

You can edit the results displayed in the **Search Results** pane. For example, to change all objects found by a search to have the same property value, select the objects in the **Search Results** pane and change the property value of one of the objects.

Refine a Search

To refine the previous search, in the search bar, click the **Select Search Options** button () and select **Refine Search**. A **Refine** button replaces the **Search** button on the search bar. Use the search bar to define new search criteria and then click the **Refine** button. The Model Explorer searches for objects that match both the previous search criteria and the new criteria.

Model Explorer: Property Dialog Pane


In this section...
“What You Can Do with the Dialog Pane” on page 9-70
“Showing and Hiding the Dialog Pane” on page 9-70
“Editing Properties in the Dialog Pane” on page 9-70

What You Can Do with the Dialog Pane

You can use the **Dialog** pane to view and change properties of objects that you select in the **Model Hierarchy** pane or in the **Contents** pane.

Showing and Hiding the Dialog Pane

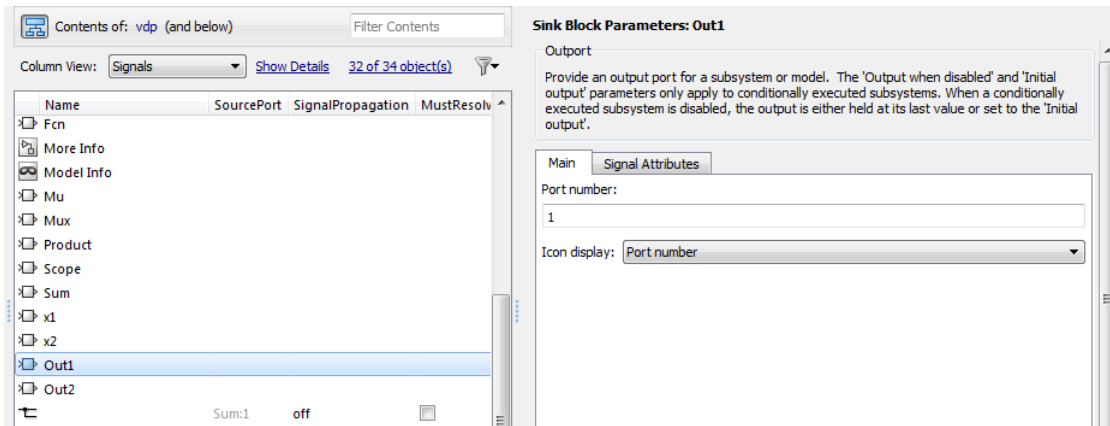
By default, the **Dialog** pane appears in the Model Explorer, to the right of the **Contents** pane. To show or hide the **Dialog** pane, use one of these approaches:

- From the **View** menu, select **Show Dialog Pane**.
- From the main toolbar, click the **Dialog View** button ()

Editing Properties in the Dialog Pane

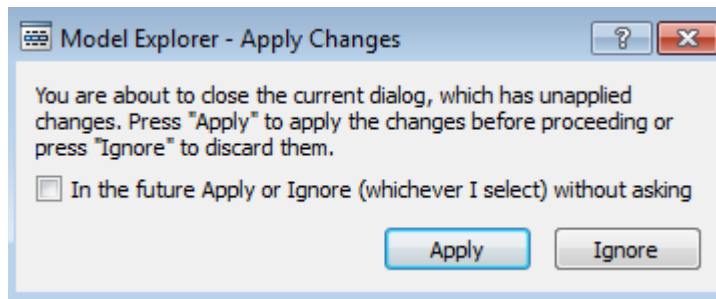
To edit property values using the **Dialog** pane:

- 1 In the **Contents** pane, select an object (such as a block or signal). The **Dialog** pane displays the properties of the object you selected.



- 2 Change a property (for example, the port number of an Outport block) in the **Dialog** pane.
- 3 Click **Apply** to accept the change, or click **Revert** to return to the original value.

By default, clicking outside a dialog box with unapplied changes causes the Apply Changes dialog box to appear:



Click **Apply** to accept the changes or **Ignore** to revert to the original settings.

To prevent display of the **Apply Changes** dialog box:

- 1 In the dialog box, click the **In the future Apply or Ignore (whichever I select) without asking** check box.

- 2** If you want Simulink to apply changes without warning you, press **Apply**.
If you want Simulink to ignore changes without warning you, press **Ignore**.

To restore display of the **Apply Changes** dialog box, from the **Tools** menu, select **Prompt if dialog has unapplied changes**.

Finder

In this section...

“Find Model Objects” on page 9-73

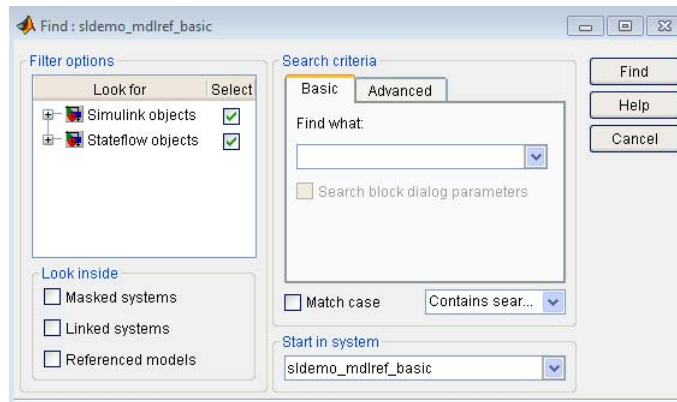
“Filter Options” on page 9-75

“Search Criteria” on page 9-76

Find Model Objects

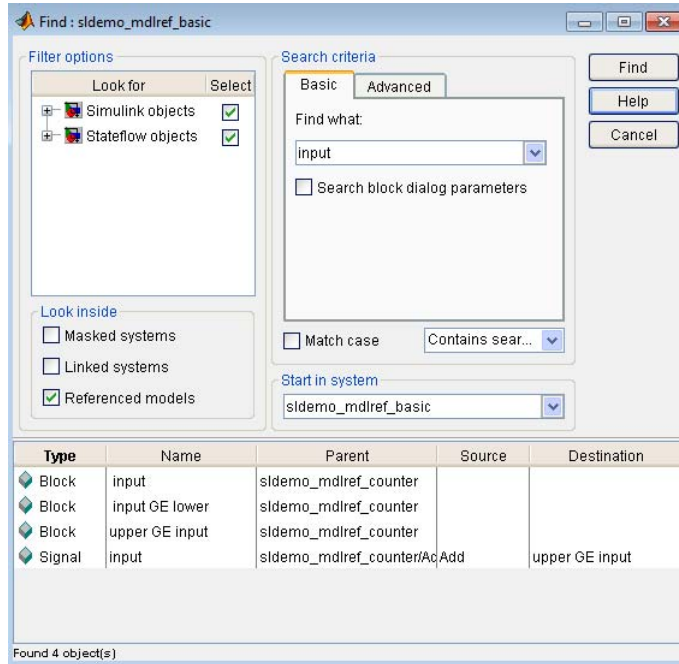
Use the Finder to locate blocks, signals, states, or other objects in a model.

- 1 Open the Finder. In the Simulink Editor, either select **Edit > Find**, or press **Ctrl+F**.



- 2 In the **Filter options** area, specify the kinds of objects to look for, and where to search for them. See “Filter Options” on page 9-75.
- 3 In the **Search criteria** area, specify the criteria that objects must meet to satisfy your search request. See “Search Criteria” on page 9-76.
- 4 If you have more than one system or subsystem open, click the **Start in system** list. From this list, select the system or subsystem where you want the search to begin.
- 5 Click **Find**.

The Finder searches the selected system for objects that meet the criteria that you have specified. Any objects that satisfy the criteria appear in the results panel at the bottom of the dialog box.



Display a Found Object

To display a found object, double-click its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object.

To sort the results list based on the values in a column, click the button at the top of that column. For example, to sort the results by object type, click the **Type** button. To sort the list in ascending order, click the button once. To sort the list in descending order, click the button twice.

To display the parameters or properties of an object, right-click the object in the list. From the context menu, select **Parameter** or **Properties**.

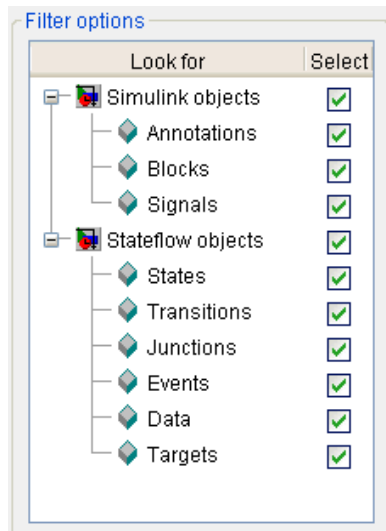
If you close a model that has objects in the Finder search results list, then double-clicking a found object from that model does not open the model. To open an object that was in the Finder search results list, simply rerun the search by clicking the **Find** button.

Filter Options

To specify the kinds of objects to look for, and where to search for them, use the **Filter options** panel.

Object Type List

The object type list lists the types of objects that the Finder can find. To exclude a type of object from the Finder search, clear the check box for the object.



Specifying Kinds of Systems To Search

You can configure the Finder to search inside masked systems, linked library systems, and referenced models. The search starts with the system that you specify in the **Start in system** field.

Object Type Option	Where the Finder Searches
Look inside masked systems	Inside masked subsystems
Look inside linked systems	Inside subsystems linked to libraries
Look inside referenced models	Inside referenced models, down through a model reference hierarchy <hr/> Note The Finder loads unopened referenced models. You can stop the search by using the pop-up cancel dialog box. <hr/>

If you select more than one option, then the Finder searches in systems that fit any of the specified options. For example, if you select the **Look inside linked systems** and **Look inside referenced models** options, then the Finder:

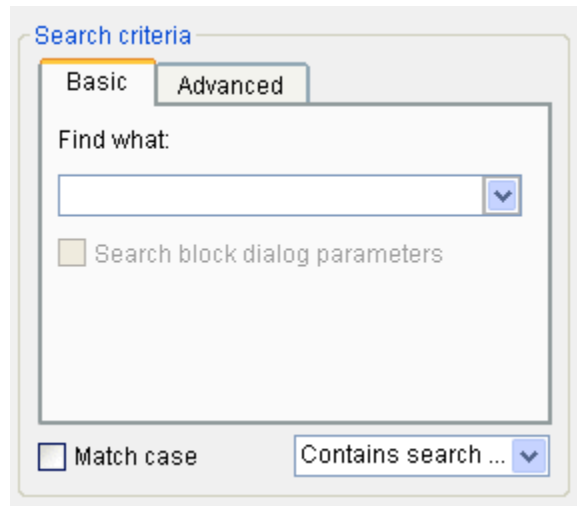
- Searches in linked library systems and referenced models that are not in a masked system
- Does *not* search inside:
 - Masked systems, including those that are in linked library systems or referenced models
 - Linked library systems or model referenced models, if the system or model is in a masked system

Search Criteria

To specify the criteria that objects must meet to satisfy your search request, use the **Search criteria** panel.

Basic

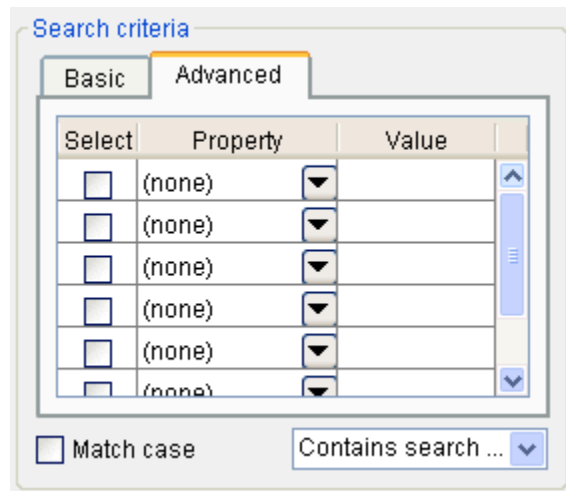
To search for an object whose name matches a specified text string, use the **Basic** panel. Enter search text in the **Find what** edit box. To reuse previous search text, select the appropriate search text from the dropdown list.



To include dialog parameters in the search, select **Search block dialog parameters**.

Advanced

To specify a set of as many as seven properties that an object must have to satisfy your search request, use the **Advanced** panel.



To specify a property, enter its name in one of the cells in the **Property** column or select the property from the property list of a cell. To display the property list, select the down arrow button next to the cell. Next enter the value of the property in the **Value** column. When you enter a property name, the Finder checks the check box next to the property name in the **Select** column. This indicates that the property is to be included in the search. To exclude the property, clear the check box.

Match case

Select this option if you want the Finder to consider case when matching search text against the value of an object property.

Other match options

Next to the **Match case** option is a dropdown list that specifies other match options.

- **Matches search string**

Specifies a match if the property value and the search text are identical except possibly for case.

- **Contains search string**

Specifies a match if a property value includes the search text.

- **Regular expression**

Specifies that the search text should be treated as a regular expression when matched against property values. The following characters have special meanings when they appear in a regular expression.

Character	Meaning
^	Matches start of string.
\$	Matches end of string.
.	Matches any character.
\	Escape character. Causes the next character to have its ordinary meaning. For example, the regular expression \. matches .a and .2 and any other two-character string that begins with a period.

Character	Meaning
*	Matches zero or more instances of the preceding character. For example, <code>ba*</code> matches <code>b</code> , <code>ba</code> , <code>baa</code> , and so on.
+	Matches one or more instances of the preceding character. For example, <code>ba+</code> matches <code>ba</code> , <code>baa</code> , etc.
[]	Indicates a set of characters that can match the current character. A hyphen can be used to indicate a range of characters. For example, <code>[a-zA-Z0-9_]+</code> matches <code>foo_bar1</code> but not <code>foo\$bar</code> . A <code>^</code> indicates a match when the current character is not one of the following characters. For example, <code>[^0-9]</code> matches any character that is not a digit.
<code>\w</code>	Matches a word character (same as <code>[a-zA-Z0-9_]</code>).
<code>\W</code>	Matches a nonword character (same as <code>[^a-zA-Z0-9_]</code>).
<code>\d</code>	Matches a digit (same as <code>[0-9]</code>).
<code>\D</code>	Matches a nondigit (same as <code>[^0-9]</code>).
<code>\s</code>	Matches white space (same as <code>[\t\r\n\f]</code>).
<code>\S</code>	Matches nonwhite space (same as <code>[^\t\r\n\f]</code>).
<code>\<WORD\></code>	Matches <code>WORD</code> where <code>WORD</code> is any string of word characters surrounded by white space.

Model Browser

In this section...
“About the Model Browser” on page 9-80
“Navigating with the Mouse” on page 9-82
“Navigating with the Keyboard” on page 9-82
“Showing Library Links” on page 9-82
“Showing Masked Subsystems” on page 9-82

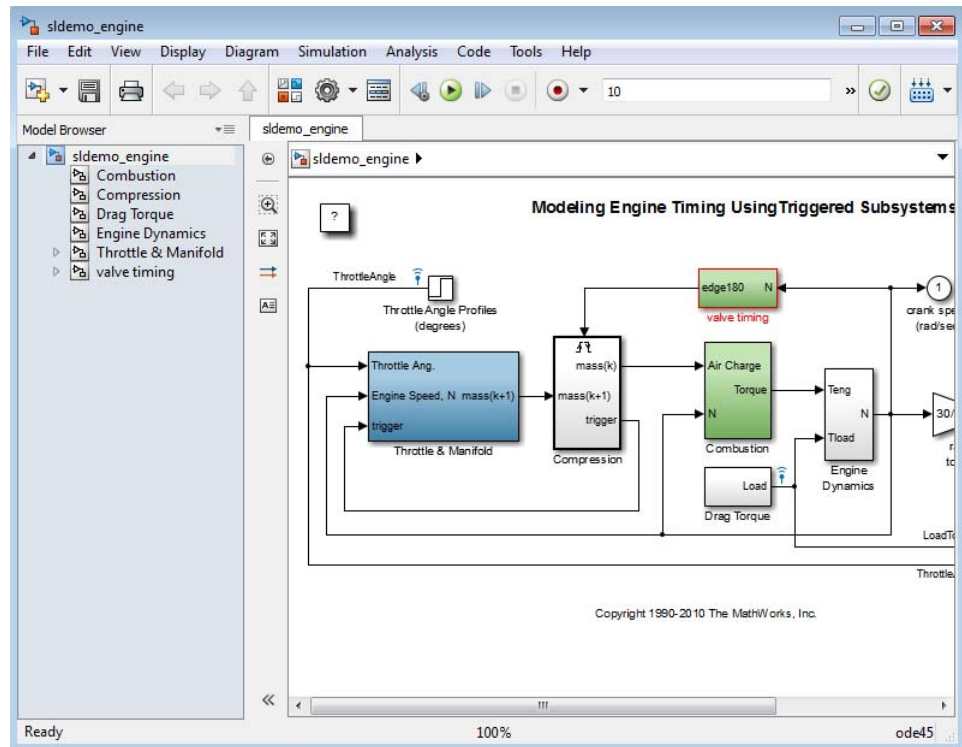
About the Model Browser

The Model Browser enables you to

- Navigate a model hierarchically
- Open systems in a model
- Determine the blocks contained in a model

Note The browser is available only on Microsoft Windows platforms.

To display the Model Browser, in the Simulink Editor, select **View > Model Browser > Show Model Browser**.



The model window splits into two panes. The left pane displays the browser, a tree-structured view of the block diagram displayed in the right pane.

Note The **Browser initially visible** preference causes Simulink to open models by default in the Model Browser. To set this preference, in the Simulink Editor, select **File > Simulink Preferences**.

The top entry in the tree view corresponds to your model. A button next to the model name allows you to expand or contract the tree view. The expanded view shows the model's subsystems. A button next to a subsystem indicates that the subsystem itself contains subsystems. You can use the button to list the subsystem's children. To view the block diagram of the model or any subsystem displayed in the tree view, select the subsystem. You can

use either the mouse or the keyboard to navigate quickly to any subsystem in the tree view.

Navigating with the Mouse

Click any subsystem visible in the tree view to select it. Click the + button next to any subsystem to list the subsystems that it contains. Click the button again to contract the entry.

Navigating with the Keyboard

Use the up/down arrows to move the current selection up or down the tree view. Use the left/right arrow or +/- keys on your numeric keypad to expand an entry that contains subsystems.

Showing Library Links

The Model Browser can include or omit library links from the tree view of a model. Use the Preferences dialog box to specify whether to display library links by default. To toggle display of library links, select **View > Model Browser > Include Library Links**.

Showing Masked Subsystems

The Model Browser can include or omit masked subsystems from the tree view. If the tree view includes masked subsystems, selecting a masked subsystem in the tree view displays its block diagram in the diagram view. Use the Preferences dialog box to specify whether to display masked subsystems by default. To toggle display of masked subsystems, select **View > Model Browser > Include Systems with Mask Parameters**.

Model Dependency Viewer

In this section...

“About Model Dependency Views” on page 9-83

“Opening the Model Dependency Viewer” on page 9-88

“Manipulating a Dependency View” on page 9-89

“Browsing Dependencies” on page 9-95

“Saving a Dependency View” on page 9-95

“Printing a Dependency View” on page 9-95

About Model Dependency Views

The Model Dependency Viewer displays a dependency view of a model. The dependency view is a graph of all the *models* and *libraries* referenced directly or indirectly by the model. You can use the dependency view to quickly find and open referenced libraries and models. To identify and package *all required files*, see instead “Analyze Model Dependencies” on page 13-159.

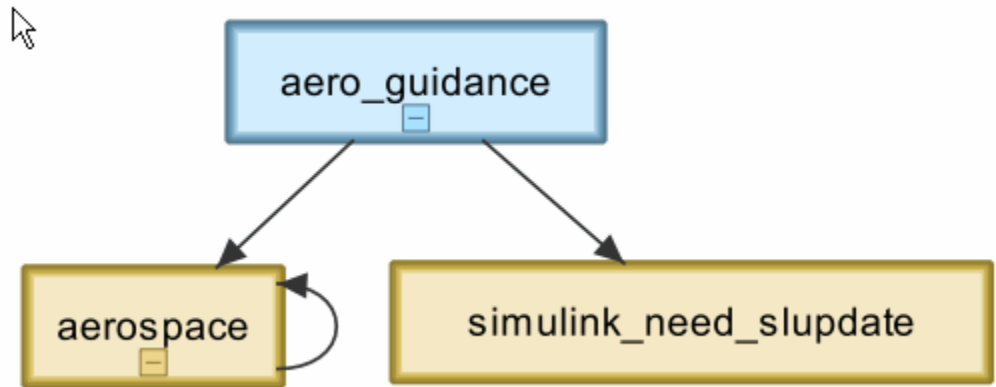
The Model Dependency Viewer allows you to choose between the following types of dependency views of a model reference hierarchy.

- “File Dependency View” on page 9-83
- “Referenced Model Instances View” on page 9-85
- “Processor-in-the-Loop Mode Indicator” on page 9-87
- “Warning Icon” on page 9-88

File Dependency View

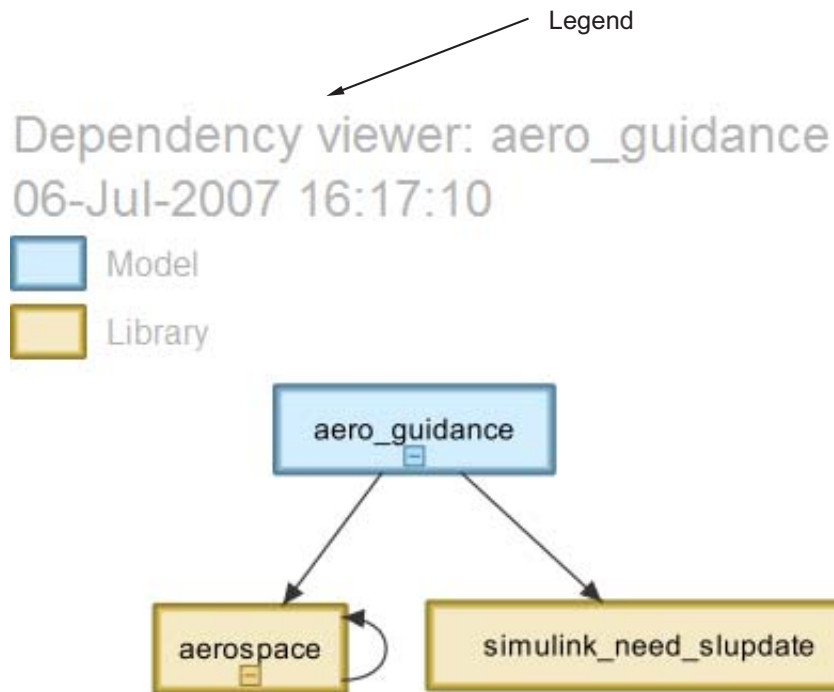
A *file dependency view* shows the model and library files referenced by a top model. A referenced model or library file appears only once in the view even if it is referenced more than once in the model hierarchy displayed in the view. A file dependency view consists of a set of blocks connected by arrows. Blue blocks represent model files; brown boxes, libraries. Arrows represent dependencies. For example, the arrows in the following view

indicate that the `aero_guidance` model references two libraries: `aerospace` and `simulink_need_slupdate`.



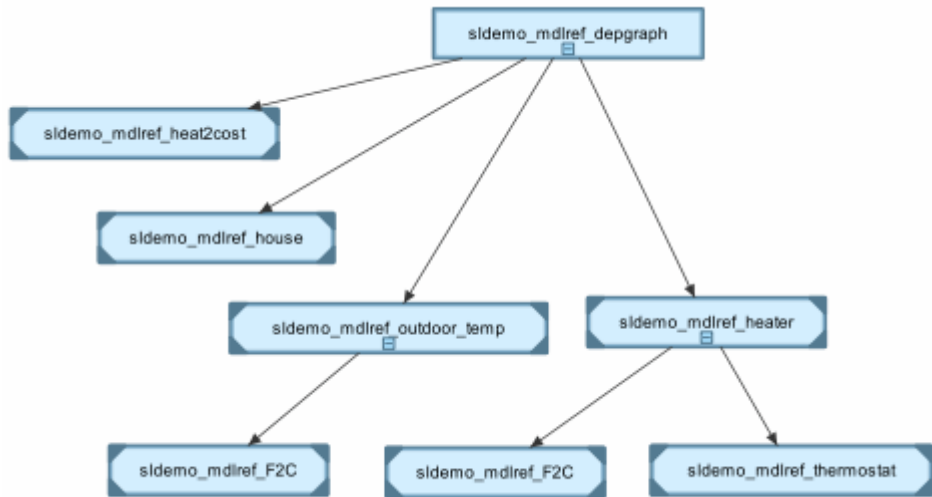
An arrow that points to the library from which it emerges indicates that the library references itself, i.e., blocks in the library reference other blocks in that same library. For example, the preceding view indicates that the `aerospace` library references itself.

A file dependency view optionally includes a legend that identifies the model in the view and the date and time the view was created.

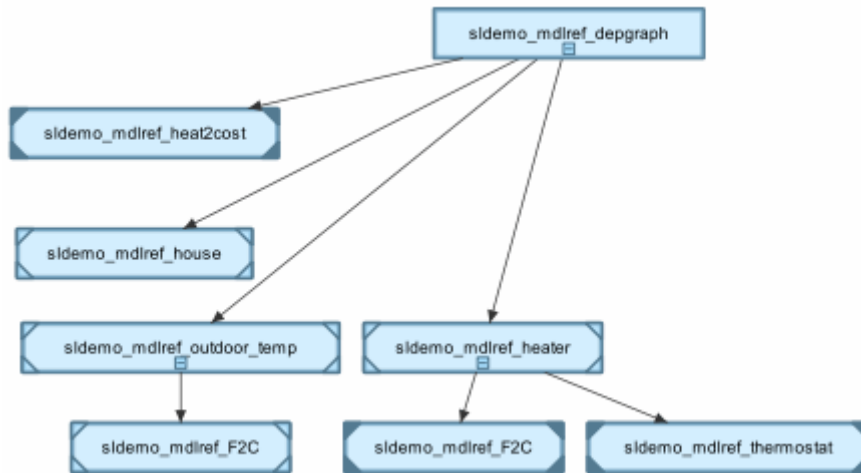


Referenced Model Instances View

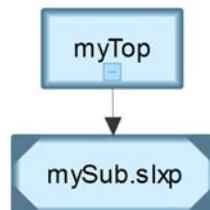
A *referenced model instances* view shows every reference to a model in a model reference hierarchy (see “Model Reference”) rooted at the top model targeted by the view. If a model hierarchy references the same model more than once, the referenced model appears multiple times in the instance view, once for each reference. For example, the following view indicates that the model reference hierarchy rooted at `sldemo_md1ref_depgraph` contains two references to the model `sldemo_md1ref_F2C`.



In an instance view, boxes represent a top model and model references. Boxes representing accelerated-mode instances (see “Referenced Model Simulation Modes” on page 6-21) have filled triangles in their corners; boxes representing normal-mode instances, have unfilled triangles in their corners. For example, the following diagram indicates that one of the references to `sldemo_mdref_F2C` operates in normal mode; the other, in accelerated mode.

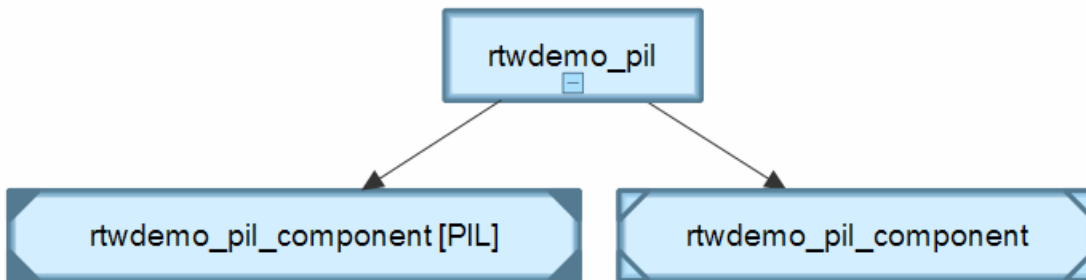


Boxes representing protected referenced models have a lock icon, and the model name has the .slxp extension. Protected referenced model boxes have no expand(+)/collapse(-) button.




Processor-in-the-Loop Mode Indicator

An instance view appends PIL to the names of models that run in Processor-in-the-Loop mode (see “Specify the Simulation Mode” on page 6-23). For example, the following dependency instance view indicates that the model named `rtwdemo_pil_component` runs in processor-in-the-loop mode.



Warning Icon

An instance view displays warning icons on instance boxes to indicate that a reference is configured to run in Normal mode actually runs in Accelerated mode because it is directly or indirectly referenced by another model reference that runs in Accelerated mode.

The warning icon is a yellow triangle .

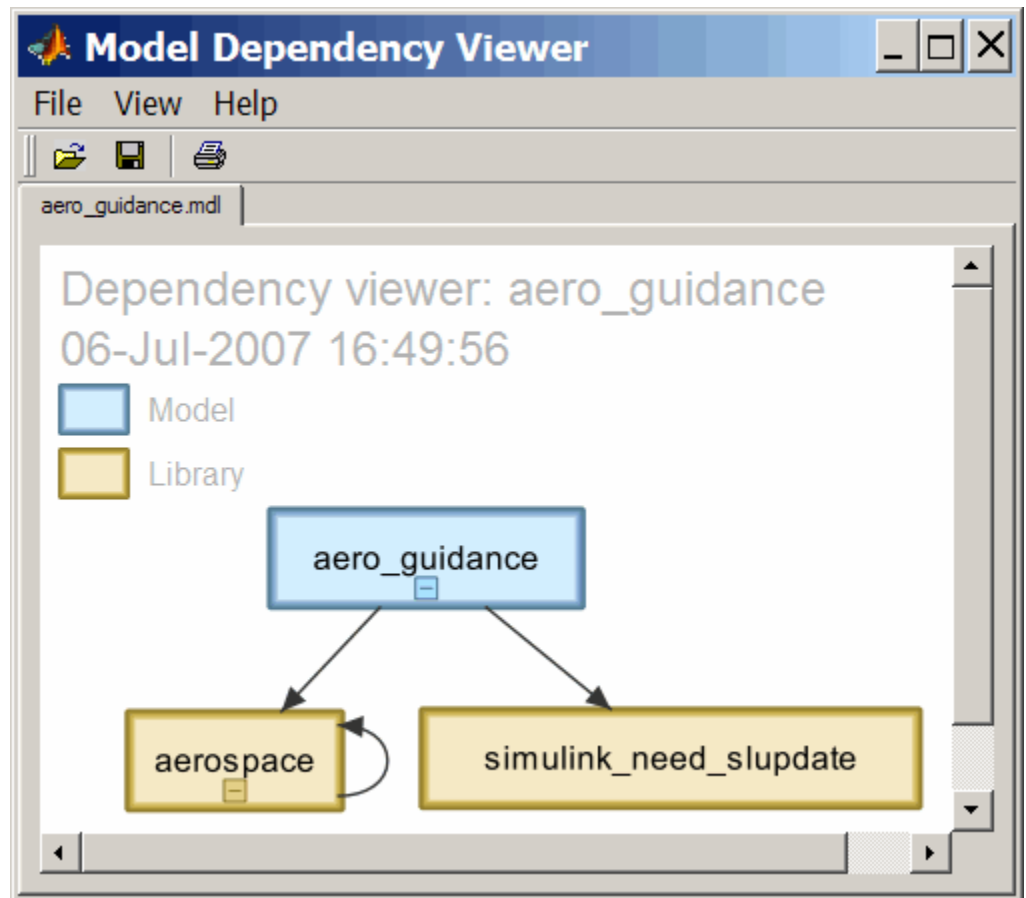
Opening the Model Dependency Viewer

The Model Dependency Viewer displays a graph of all the models and libraries referenced directly or indirectly by the model. You can use the dependency viewer to quickly find and open referenced libraries and models.

To display a dependency view for a model:

- 1 Open the model.
- 2 Select **Analysis > Model Dependencies > Model Dependency Viewer**, then select the type of view you want to see:
 - **Models & Libraries**
 - **Models Only**
 - **Referenced Model Instances**

The Model Dependency Viewer appears, displaying a dependency view of the model.



Manipulating a Dependency View

The Model Dependency Viewer allows you to manipulate dependency views in various ways. See the following topics for more information:

- “Changing Dependency View Type” on page 9-90
- “Excluding Block Libraries from a File Dependency View” on page 9-90

- “Including Simulink Blocksets in a File Dependency View” on page 9-90
- “Changing View Orientation” on page 9-91
- “Expanding or Collapsing Views” on page 9-91
- “Zooming a Dependency View” on page 9-92
- “Moving a Dependency View” on page 9-92
- “Rearranging a Dependency View” on page 9-93
- “Displaying and Hiding a Dependency View’s Legend” on page 9-93
- “Displaying Full Paths of Referenced Model Instances” on page 9-93
- “Refreshing a Dependency View” on page 9-94

Changing Dependency View Type

You can change the type of dependency view displayed in the viewer.

To change the type of dependency view, in the Model Dependency Viewer:

- Select **View > Dependency Type > Model File Dependencies** (see “File Dependency View” on page 9-83)
- or
- Select **View > Dependency Type > Referenced Model Instances** (see “Referenced Model Instances View” on page 9-85).

Excluding Block Libraries from a File Dependency View

By default a file dependency view includes libraries on which a model depends.

To exclude block libraries:

- Deselect **View > Include Libraries**.

Including Simulink Blocksets in a File Dependency View

By default, a file dependency view omits MathWorks block libraries when **View > Include Libraries** is selected.

To include libraries supplied by MathWorks:

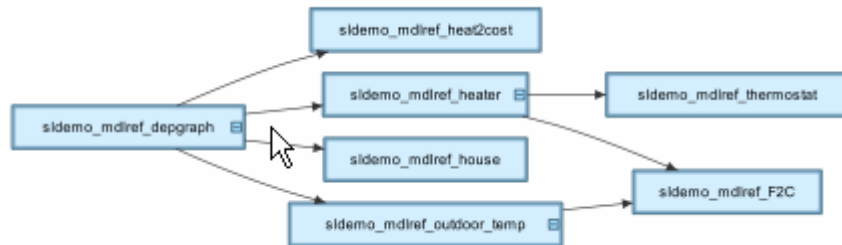
- Select **View > Show MathWorks Dependencies**.

Changing View Orientation

By default the orientation of the dependency graph displayed in the viewer is vertical.

To change the orientation to horizontal:

- Select **View > Orientation > Horizontal**.

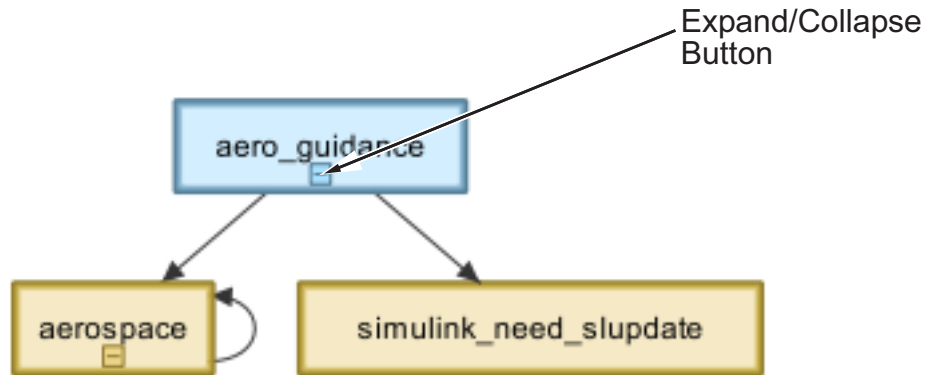


Expanding or Collapsing Views

You can expand or collapse each model or library in the dependency view to display or hide the dependencies for that model or library.

To expand or collapse views:

- Click the expand(+)/collapse(-) button on the box representing the model or library to expand or collapse that view.



Zooming a Dependency View

You can enlarge or reduce the size of the dependency graph displayed in the viewer.

To zoom a dependency view in or out, do either of the following:

- Press and hold down the **spacebar** key and then press the **+** or **-** key on the keyboard.
- Move the scroll wheel on your mouse forward or backward.

To fit the view to the viewer window:

- Press and release the **spacebar** key.

Moving a Dependency View

You can move a dependency view to another location in the viewer window.

To move the dependency view:

- 1 Move the cursor over the view.
- 2 Press your keyboard's space bar and your mouse's left button simultaneously.
- 3 Move the cursor to drag the view to another location.

Rearranging a Dependency View

You can rearrange a dependency view by moving the blocks representing models and libraries. This can make a complex view easier to read.

To move a block to another location:

- 1 Select the block you want to move by clicking it.
- 2 Drag and drop the block in the new location.

Displaying and Hiding a Dependency View's Legend

The dependency view can display a legend that identifies the model in the view and the date and time the view was created.

To display or hide a dependency view's legend:

- Select **View > Show Legend** from the viewer's menu bar.

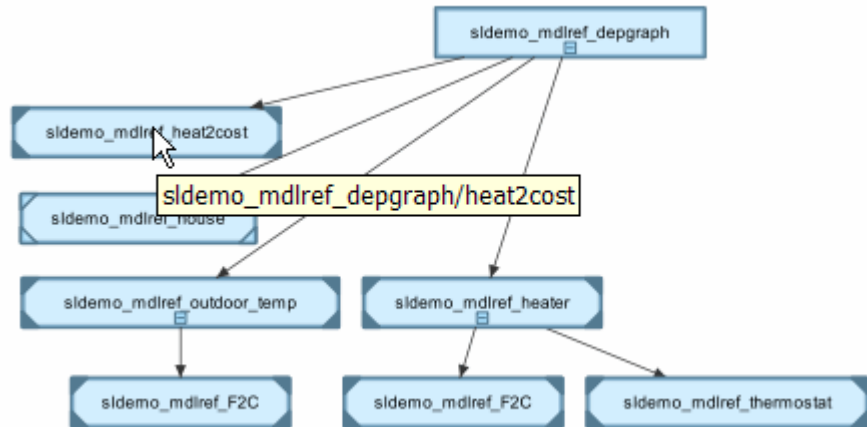
Displaying Full Paths of Referenced Model Instances

In an instance view (see “Referenced Model Instances View” on page 9-85) , you can display the full path of a model reference in a tooltip or in the box representing the reference.

To display the full path in a tooltip:

- Move the cursor over the box representing the reference in the view.

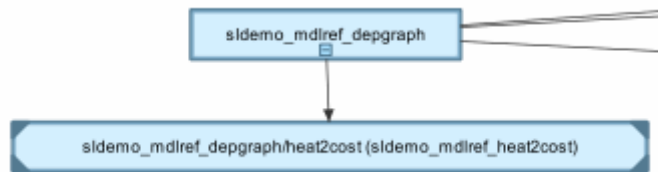
A tooltip appears, displaying the path displays the full path of the Model block corresponding to the instance.



To display full paths in the boxes representing the instances:

- Select **View > Show Full Path**.

Each box in the instance view now displays the path of the Model block corresponding to the instance. The name of the referenced model appears in parentheses as illustrated in the following figure.



Refreshing a Dependency View

After changing a model displayed in a dependency view or any of its dependencies, you must update the view to reflect any dependency changes.

To update the view:

- Select **View > Refresh** from the dependency viewer's menu bar.

Browsing Dependencies

You can use a dependency view to browse a model's dependencies:

- To open a model or library displayed in the view, double-click its block.
- To display the Model block corresponding to an instance in an instance view, right-click the instance and select **Highlight Block** from the menu that appears.
- To open all models displayed in the view, select **File > Open All Models** from the viewer's menu bar.
- To save all models displayed in the view, select **File > Save All Models**.
- To close all models displayed in the view, select **File > Close All Models**.

Saving a Dependency View

You can save a dependency view for later viewing.

To save the current view:

- Select **File > Save As** from the viewer's menu bar, then enter a name for the view.

To reopen the view:

- Select **File > Open** from any viewer's menu bar, then select the view you want to open.

Printing a Dependency View

To print a dependency view:

- Select **File > Print** from the dependency viewer's menu bar.

View Linked Requirements in Models and Blocks

In this section...
“Overview of Requirements Features in Simulink” on page 9-96
“Highlighting Requirements in a Model” on page 9-97
“Viewing Information About a Requirements Link” on page 9-99
“Navigating to Requirements from a Model” on page 9-100
“Filtering Requirements in a Model” on page 9-102

Overview of Requirements Features 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 Verification and Validation 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

Highlighting 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.

- “Highlighting a Model Using the Simulink Editor” on page 9-97
- “Highlighting a Model Using the Model Explorer” on page 9-98

Note If your model contains a Model block whose referenced model contains requirements, those requirements are not highlighted. If you have Simulink Verification and Validation, 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”.

Highlighting a Model Using the Simulink Editor

If you are working in the Simulink Editor and want to see which model objects in the `slvndemo_fuelsys_officereq` model have requirements, follow these steps:

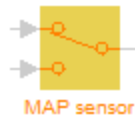
- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```

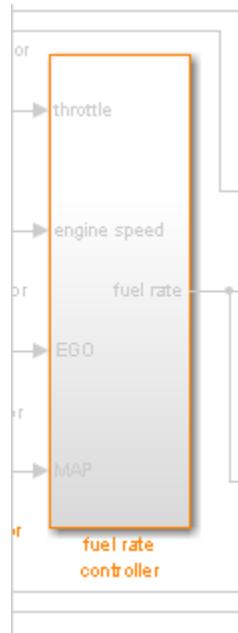
- 2 Select **Analysis > Requirements Traceability > Highlight Model**.

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 To remove the highlighting from the model, select **Analysis > Requirements Traceability > Unhighlight Model**. 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.


Highlighting 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:

```
slvndemo_fuelsys_officereq
```

2 Select **View > 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 top of the page, not at the bookmark location.

Viewing 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 only modify the requirements information if you have a Simulink Verification and Validation license.

For example, to view information about the requirements link from the MAP Sensor block in the `slvndemo_fuelsys_officereq` example model, follow these steps:

1 Open the example model:

```
slvndemo_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.

Navigating to Requirements from a Model

Navigating 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 Traceability > 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.

Navigating 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 `slvndemo_fuelsys_officereq` model, and navigate to the requirements using the links inside the block.

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```

- 2 In the Simulink Editor, select **Analysis > Requirements Traceability > Highlight Model**.

- 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 Editor, select **View > Library Browser**.

- 6 On the **Libraries** pane, select **Simulink Verification and Validation**.

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, `slvndemo_FuelSys_DesignDescription.docx`, opens, with the section **2.1 Mass airflow estimation** selected.

Filtering Requirements in a Model

- “Filtering Requirements Highlighting by User Tag” on page 9-102
- “Filtering Options for Highlighting Requirements” on page 9-103

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 `slvndemo_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:

```
slvndemo_fuelsys_officereq
```

- 2 In the Simulink Editor, select **Analysis > Requirements > Settings**.

The Requirements Settings dialog box opens. If you do not have a Simulink Verification and Validation 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 `slvndemo_fuelsys_officereq` model for requirements. Select **Analysis > Requirements > Highlight Model**.

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, select **Analysis > Requirements > Settings**, and 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.
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.

Option	Description
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.

Managing Model Configurations

- “About Model Configurations” on page 10-2
- “Multiple Configuration Sets in a Model” on page 10-3
- “Share a Configuration for Multiple Models” on page 10-4
- “Share a Configuration Across Referenced Models” on page 10-6
- “Manage a Configuration Set” on page 10-12
- “Manage a Configuration Reference” on page 10-18
- “About Configuration Sets” on page 10-26
- “About Configuration References” on page 10-29
- “Model Configuration Command Line Interface” on page 10-33

About Model Configurations

A model configuration is a named set of values for the parameters of a model. It is referred to as a *configuration set*. Every new model is created with a default configuration set, called **Configuration**, that initially specifies default values for the model parameters. You can change the default values for new models by setting the Model Configuration Preferences. For more information, see “Model Configuration Preferences” on page 10-28.

You can subsequently create and modify additional configuration sets and associate them with the model. The configuration sets associated with a model can specify different values for any or all configuration parameters. For more information, see “About Configuration Sets” on page 10-26. For examples on how to use configuration sets, see:

- “Multiple Configuration Sets in a Model” on page 10-3
- “Manage a Configuration Set” on page 10-12

By default, a configuration set resides within a single model so that only that model can use it. Alternatively, you can store a configuration set independently, so that other models can use it. A configuration set that exists outside any model is a *freestanding configuration set*. Each model that uses a freestanding configuration set defines a *configuration reference* that points to the freestanding configuration set. A freestanding configuration set allows you to single-source a configuration set for several models. For more information, see “About Configuration References” on page 10-29. For examples on how to use configuration references, see:

- “Share a Configuration for Multiple Models” on page 10-4
- “Share a Configuration Across Referenced Models” on page 10-6
- “Manage a Configuration Reference” on page 10-18

Multiple Configuration Sets in a Model

A model can include many different configuration sets. This capability is useful if you want to compare the difference in simulation output after changing the values of several parameters. Attaching additional configuration sets allows you to quickly switch the active configuration.

- 1** To create additional configuration sets in your model, in the Model Explorer, select your model node in the Model Hierarchy pane and do one of the following:
 - Right-click the model node and select **Configuration > Add Configuration**.
 - Right-click an existing configuration set. In the context menu, select **Copy**.
- 2** To import a previously saved configuration set, right-click the model node and select **Configuration > Import**. In the Import Configuration From File dialog box, select a configuration file.
- 3** To modify the newly added configuration set, in the Model Hierarchy pane, select the configuration node. In the **Contents** pane, select a component, and then modify any parameters which are displayed in the right pane.
- 4** To make the new configuration set the active configuration, in the configuration set context menu, select **Activate**. In the Model Hierarchy pane, the new configuration set name is now displayed as (Active).
- 5** To simulate your model using a different configuration set, switch the active configuration by repeating step 4.

Share a Configuration for Multiple Models

To share a configuration set between models, the configuration set must be a configuration set object in the base workspace and you must create a configuration reference in your model to reference the configuration set object. You can create configuration references in other models that also point to the same configuration set object.

For example, first convert an existing configuration set in your model to a configuration reference:

- 1 Open the Model Explorer.
- 2 In the Model Hierarchy pane, right-click the active configuration set to share.
- 3 In the configuration set context menu, select **Convert to Configuration Reference**, which opens a dialog box. Alternatively, you can right-click the model node and select **Configuration > Convert Active Configuration to Reference**.
- 4 In the Convert Active Configuration to Reference dialog box, use the default configuration set object name, `configSetObj`, or type a name.
- 5 Click **OK**, which creates a configuration reference in the model and a configuration set object in the base workspace. The configuration reference points to the configuration set object, which has the same values as the original active configuration set. The configuration reference name in the Model Hierarchy is now marked as (Active).
- 6 To change the name of the configuration reference, select it in the Model Hierarchy, and in the right pane, change the **Name** field.

To share the preceding configuration set, which is stored as `configSetObj` in the base workspace, create a configuration reference in another model:

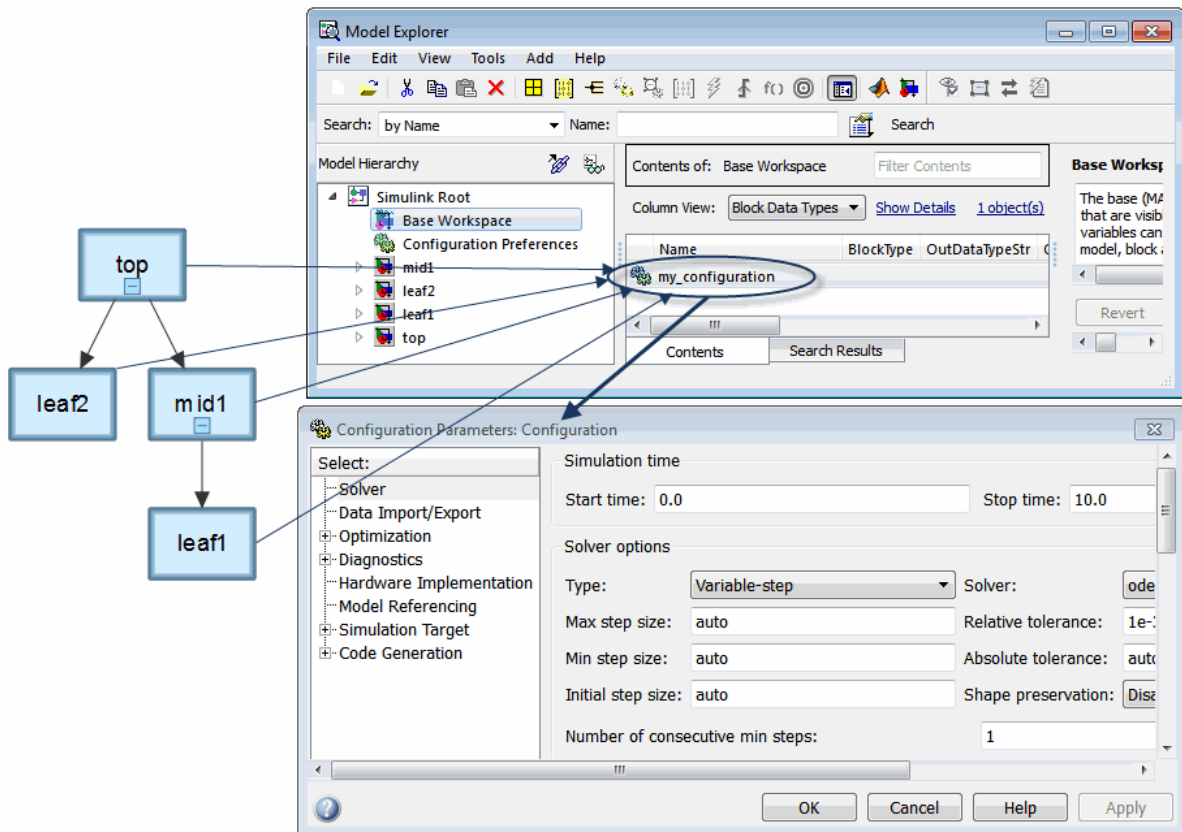
- 1 In the Model Hierarchy pane, right-click the model node.
- 2 In the context menu, select **Configuration > Add Configuration Reference**.

- 3** The Create Configuration Reference dialog box opens. Specify the name of the configuration set object, `configSetObj`, in the base workspace.
- 4** To make the new configuration reference the active configuration, in the Model Hierarchy, right-click the configuration reference. In the context menu, select **Activate**.

Both models now contain a configuration reference that points to the same configuration set object in the base workspace. Before saving and closing your models, follow the instructions to “Save a Referenced Configuration Set” on page 10-24. If you do not save the referenced configuration set from the base workspace, when you reopen your model, 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 4-56.

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 the base workspace.

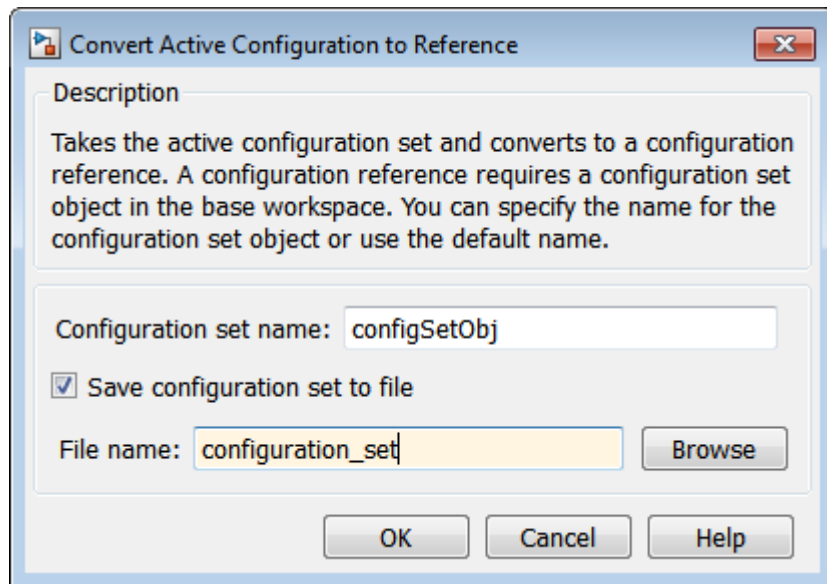


In the diagram, each model shown in the Model Dependency Viewer specifies the configuration reference, `my_configuration`, as its active configuration set. `my_configuration` points to the freestanding configuration set, `Configuration`. Therefore, the parameter values in `Configuration` apply to all four models. Any parameter change in `Configuration` applies to all four models.

Convert Configuration Set to Configuration Reference

In the top model, you must convert the active configuration set to a configuration reference:

- 1 Open the `sldemo_md1ref_depgraph` model and the Model Explorer.
- 2 In the Model Hierarchy pane, expand the top model, `sldemo_md1ref_depgraph`. In the list, right-click **Configuration (Active)**. In the context menu, select **Convert to Configuration Reference**.
- 3 In the **Configuration set name** field, specify a name for the configuration set object, or use the default name, `configSetObj`. This configuration set object is stored in the base workspace.
- 4 Optionally, you can save the configuration set to a MAT-file. Select **Save configuration set to file**. This enables the **File name** parameter.
- 5 In the **File name** field, specify a name for the MAT-file.



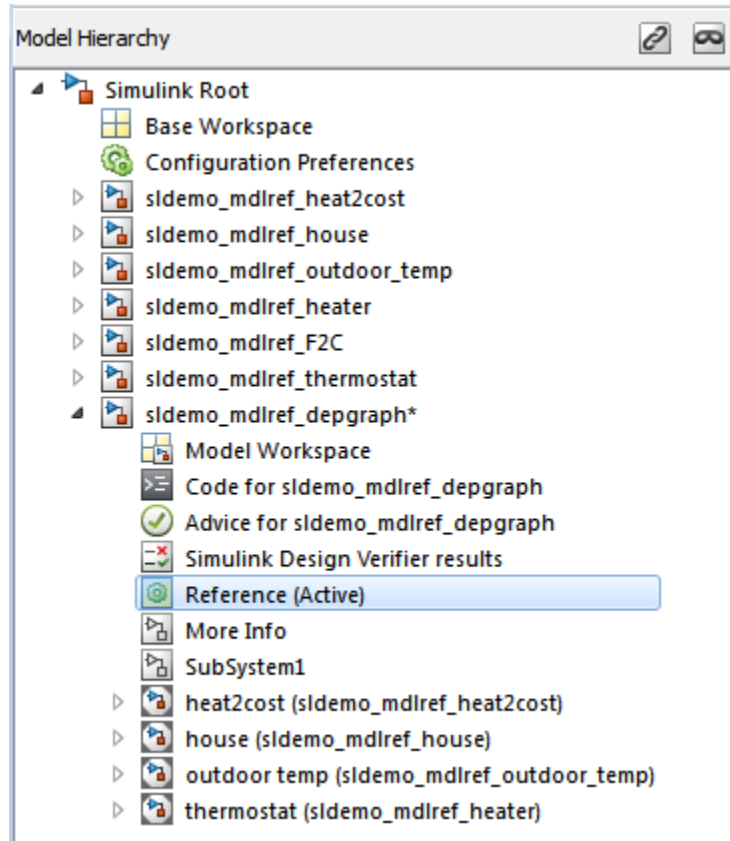
6 Click **OK**.

The original configuration set is now stored as a configuration set object, `configSetObj`, in the base workspace. The configuration set is also stored in a MAT-file, `configuration_set.mat`. The active configuration for the top model is now a configuration reference. This configuration reference points to the configuration set object in the base workspace.

Propagate a Configuration Reference

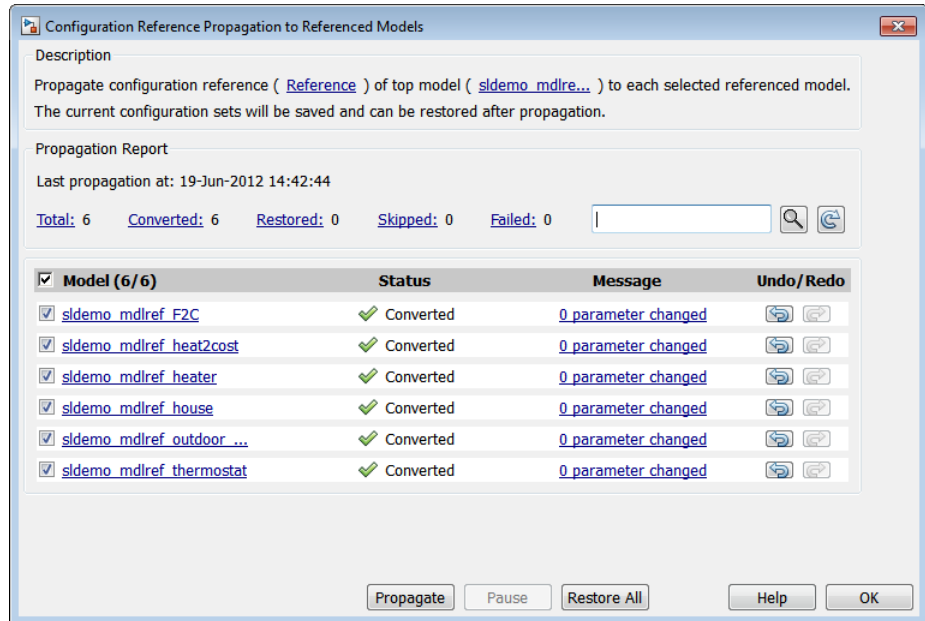
Now that the top model contains an active configuration reference, you can propagate this configuration reference to all of the child models. Propagation creates a copy of the top model configuration reference in each referenced model. For each referenced model, the configuration reference is now the active configuration. The configuration references point to the configuration set object, `configSetObj`, in the base workspace.

- 1 In the Model Explorer, in the Model Hierarchy pane, expand the `sldemo_md1ref_depgraph` node.



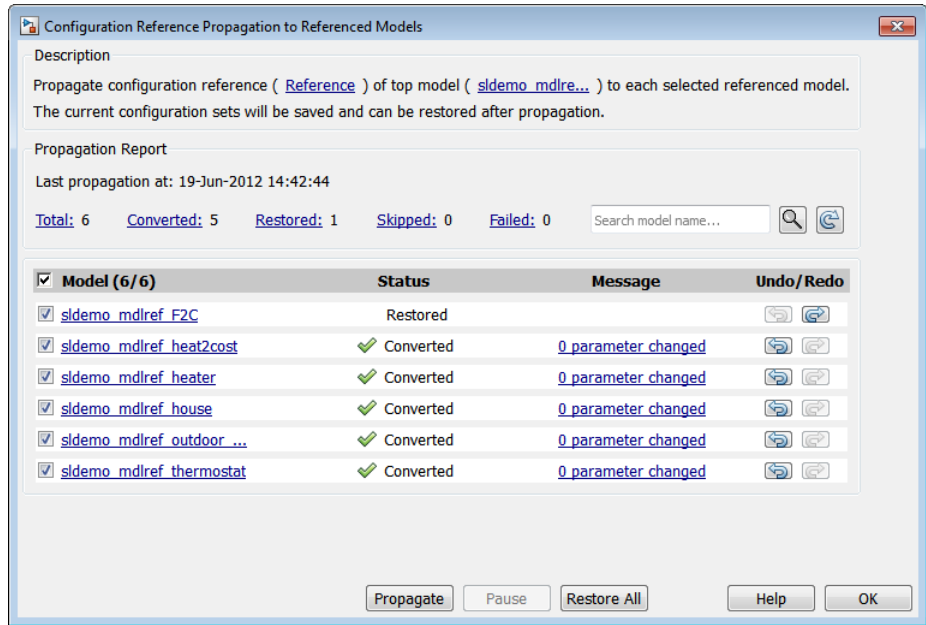
- 2** Right-click the active configuration reference, **Reference (Active)**. In the context menu, select **Propagate to Referenced Models**.
- 3** In the Configuration Reference Propagation dialog box, select the check box for each referenced model. In this example, they are already selected.
- 4** Verify that your current folder is a writable folder. 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**.



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**.



Manage a Configuration Set

In this section...

“Create a Configuration Set in a Model” on page 10-12

“Create a Configuration Set in the Base Workspace” on page 10-12

“Open a Configuration Set in the Configuration Parameters Dialog Box” on page 10-13

“Activate a Configuration Set” on page 10-13

“Set Values in a Configuration Set” on page 10-14


“Copy, Delete, and Move a Configuration Set” on page 10-14

“Save a Configuration Set” on page 10-15

“Load a Saved Configuration Set” on page 10-16

“Copy Configuration Set Components” on page 10-16

Create a Configuration Set in a Model

- 1 Open the Model Explorer.
- 2 In the Model Hierarchy pane, select the model name.
- 3 You can create a new configuration set in any of the following ways:
 - From the **Add** menu, select **Configuration**.
 - On the toolbar, click the **Add Configuration** button .
 - In the Model Hierarchy pane, right-click an existing configuration set and copy and paste the configuration set.

Create a Configuration Set in the Base Workspace

- 1 Open the Model Explorer.
- 2 In the Model Hierarchy pane, select **Base Workspace**.
- 3 You can create a new configuration set object in the following ways:
 - From the **Add** menu, select **Configuration**

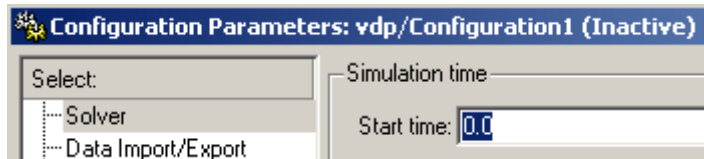
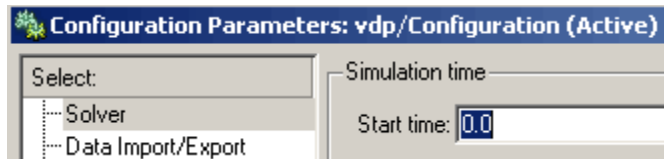
- In the toolbar, click the **Add Configuration** button 

4 The configuration set object appears in the Contents pane, with the default name, ConfigSet.

Open a Configuration Set in the Configuration Parameters Dialog Box

In the Model Explorer, to open the Configuration Parameters dialog box for a configuration set, right-click the configuration set's node to display the context menu, then select **Open**. You can open the Configuration Parameters dialog box for any configuration set, whether or not it is active.

The title bar of the dialog box indicates whether the configuration set is active or inactive.



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.

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 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, for example, testing and production, or apply standard configuration settings to new models.

To activate a configuration set, right-click the configuration set node to display the context menu, then select **Activate**.

Set Values in a Configuration Set

To set the value of a parameter in a configuration set, in the Model Explorer:

- 1 In the Model Hierarchy, select the configuration set node.
- 2 In the Contents pane, select the component from where the parameter resides.
- 3 In the Dialog pane, edit the parameter value.

Copy, Delete, and Move a Configuration Set

You can use edit commands on the Model Explorer **Edit** or context menus or object drag-and-drop operations to delete, copy, and move configuration sets among models displayed in the **Model Hierarchy** pane.

For example, to copy a configuration set from one model to another:

- 1 In the **Model Hierarchy** pane, right-click the configuration set node that you want to copy.
- 2 Select **Copy** in the configuration set context menu.
- 3 Right-click the model node in which you want to create the copy.
- 4 Select **Paste** from the model context menu.

To copy the configuration set using object drag-and-drop, hold down the right mouse button and drag the configuration set node to the 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 node to the node of the destination model.

Note You cannot move or delete an active configuration set from a model.

Save a Configuration Set

You can save the settings of configuration sets as MATLAB functions or scripts. Using the MATLAB function or script, you can share and archive model configuration sets. You can also compare the settings in different configuration sets by comparing the MATLAB functions or scripts of the configuration sets.

To save an active or inactive configuration set from the Model Explorer:

- 1 Open the model.
- 2 Open the Model Explorer.
- 3 Save the configuration set:
 - a In the **Model Hierarchy** pane:
 - Right-click the model node and select **Configuration > Export Active Configuration Set**.
 - Right-click a configuration set and select **Export**.
 - Select the model. In the **Contents** pane, right-click a configuration set and select **Export**.
 - b In the Export Configuration Set to File dialog box, specify the name of the file and the file type. If you specify a `.m` extension, the file contains a function that creates a configuration set object. If you specify a `.mat` extension, the file contains a configuration set object.

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.

-
-
-
- c Click **Save**. The Simulink software saves the configuration set.

Load a Saved Configuration Set

You can load configuration sets that you previously saved as MATLAB functions or scripts.

To load a configuration set from the Model Explorer:

- 1 Open the model.
- 2 Open the Model Explorer.
- 3 In the **Model Hierarchy** pane, right-click the model and select **Configuration > Import**.
- 4 In the Import Configuration Set 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.
- 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**” parameter to `ert.tlc`.

- 6 Optionally, activate the configuration set. For more information, see “Activate a Configuration Set” on page 10-13.

Copy Configuration Set Components

To copy a configuration set component from one configuration set to another:

- 1 Select the component in the Model Explorer **Contents** pane.
- 2 From either the Model Explorer **Edit** menu or the component context menu, select **Copy**.
- 3 Select the configuration set into which you want to copy the component.
- 4 From either the Model Explorer **Edit** menu or the component context menu, select **Paste**.

Note The copy replaces the component of the same name in the destination configuration set. For example, if you copy the Solver component of configuration set A and paste it into configuration set B, the copy replaces the existing Solver component in B.

Manage a Configuration Reference

In this section...

“Create and Attach a Configuration Reference” on page 10-18

“Resolve a Configuration Reference” on page 10-19

“Activate a Configuration Reference” on page 10-21

“Manage Configuration Reference Across Referenced Models” on page 10-22

“Change Parameter Values in a Referenced Configuration Set” on page 10-23

“Save a Referenced Configuration Set” on page 10-24


“Load a Saved Referenced Configuration Set” on page 10-24

“Why is the Build Button Not Available for a Configuration Reference?” on page 10-25

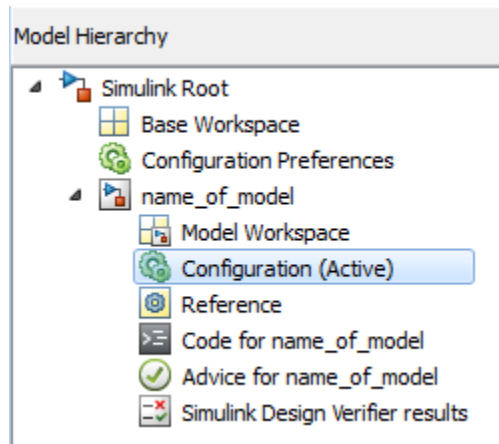
Create and Attach a Configuration Reference

To use a configuration reference, it must point to freestanding configuration set. Create a freestanding configuration set before creating a configuration reference, see “Create a Configuration Set in the Base Workspace” on page 10-12.

To create a configuration reference:

- 1** In the Model Explorer, in the Model Hierarchy pane, select the model.
- 2** Click the **Add Reference** tool  or select **Add > Configuration Reference**. The Create Configuration Reference dialog box opens.
- 3** Specify the **Configuration set name** of the configuration set object in the base workspace to be referenced.
- 4** Click **OK**. If you chose to create a configuration reference without first creating a configuration set object, a dialog box opens asking if you would like to continue. If you choose:

- **Yes**, an unresolved configuration reference is created. For more information, see “Unresolved Configuration References” on page 10-30. Follow the instructions in “Resolve a Configuration Reference” on page 10-19.
 - **No**, then the configuration reference is not created. Follow the instructions in “Create a Configuration Set in the Base Workspace” on page 10-12, and then return to step 1 above.
- 5** A new configuration reference appears in the Model Hierarchy under the selected model. The default name of the new reference is **Reference**.



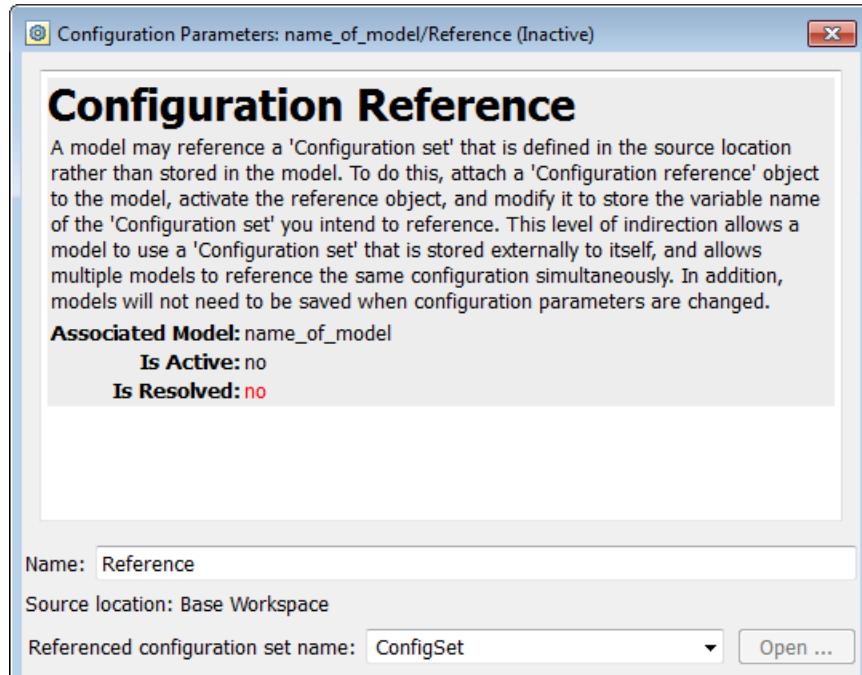
Resolve a Configuration Reference

An unresolved configuration reference is a configuration reference that is not pointing to a valid configuration set object.

To resolve a configuration reference:

- 1** In the Model Hierarchy pane, select the unresolved configuration reference or right-click the configuration reference, and select **Open** from the context menu.

The Configuration Reference dialog box opens in the Dialog pane or a separate window.

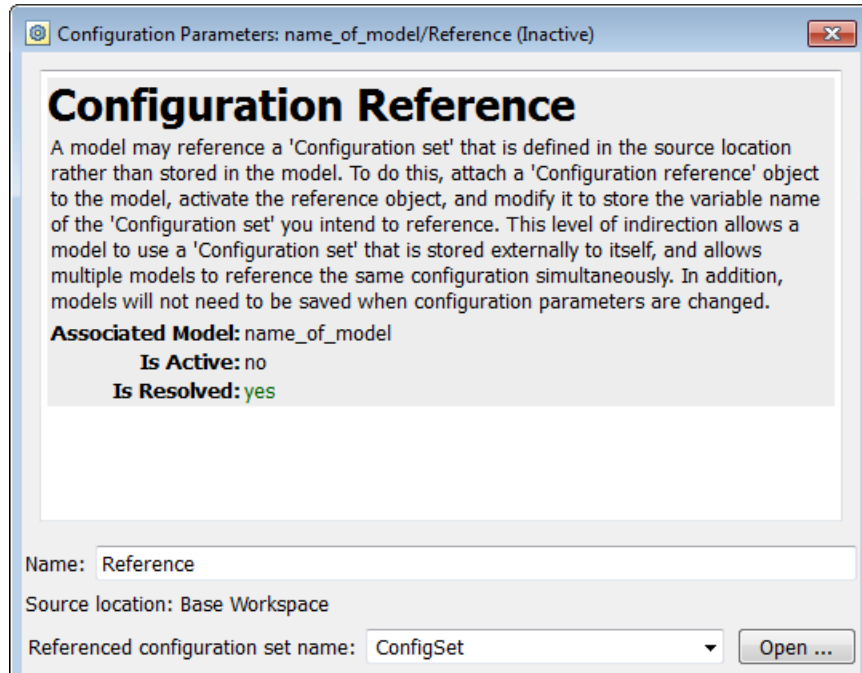


- 2 Specify the **Referenced configuration** set to be a configuration set object already in the base workspace. If one does not exist, see “Create a Configuration Set in the Base Workspace” on page 10-12.

Tip Do not specify the name of a configuration reference. If you nest a configuration reference, an error occurs.

- 3 Click **OK** or **Apply**.

If you specified a Referenced configuration that exists in the base workspace, the **Is Resolved** field in the dialog box changes to **yes**.

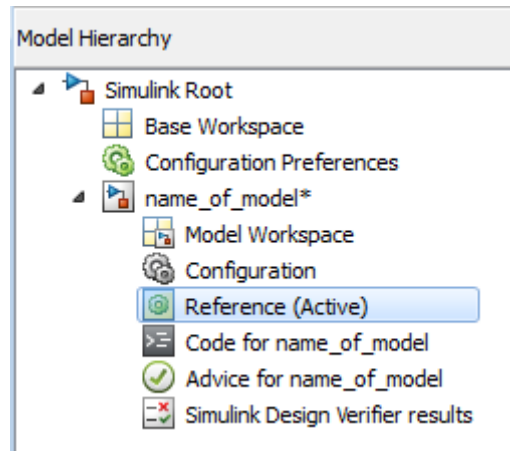


Activate a Configuration Reference

After you create a configuration reference and attach it to a model, you can activate it so that it is the active configuration.

- In the GUI, from the context menu of the configuration reference, select **Activate**.
- From the API, execute `setActiveConfigSet`, specifying the configuration reference as the second argument.

When a configuration reference is active, the **Is Active** field of the Configuration Reference dialog box changes to **yes**. Also, the Model Explorer shows the name of the reference with the suffix (Active).



The freestanding configuration set of the active reference now provides the configuration parameters for the model.

Manage Configuration Reference Across Referenced Models

In a model hierarchy, you can share a configuration reference across referenced models. Using the Configuration Reference Propagation dialog box, you can propagate a configuration reference of a top model to an individual referenced model or to all referenced models in the model hierarchy. The dialog box provides:

- A list of referenced models in the top model.
- The ability to select only specific referenced models for propagation.
- After propagation, the status for the converted configuration for each referenced model.
- A view of the changed parameters after the propagation.
- The ability to undo the configuration reference and restore the previous configuration settings for a referenced model.

To open the dialog box, in the Model Explorer, in the model hierarchy pane, right-click the configuration reference node of a model. In the context menu,

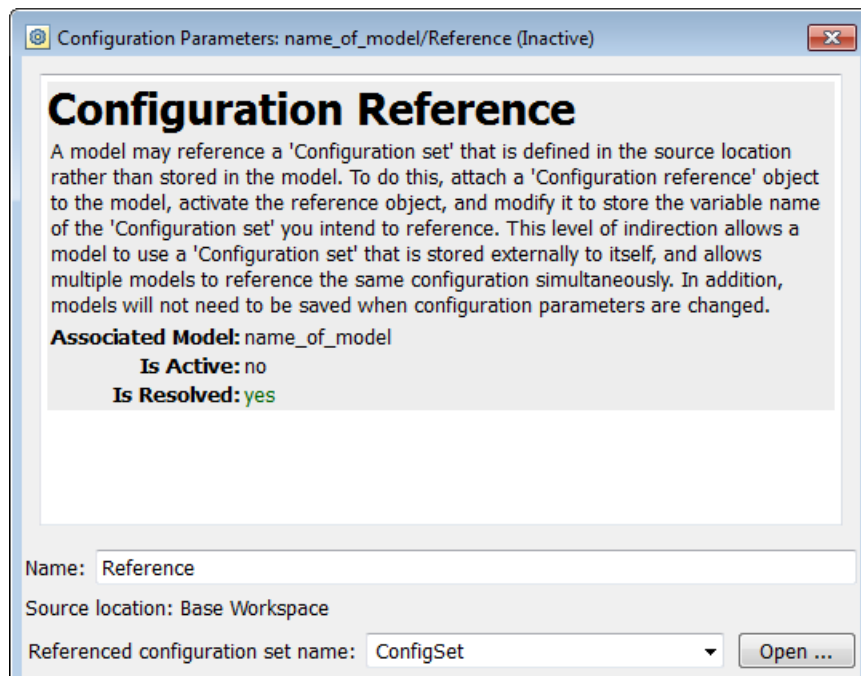
select **Propagate to Referenced Models**. For an example, see “Share a Configuration Across Referenced Models” on page 10-6.

Change Parameter Values in a Referenced Configuration Set

To obtain a referenced configuration set:

- 1 In the Model Hierarchy pane, select the configuration reference, or right-click the configuration reference, and select **Open** from the context menu.

The Configuration Reference dialog box appears in the Dialog pane or in a separate window.



- 2 To the right of the **Referenced configuration** field, click **Open**. The Configuration Parameters dialog box opens. You can now change and apply parameter values as you would for any configuration set.

Save a Referenced Configuration Set

If your model uses a configuration reference to specify the model configuration, before closing your model, 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 4-56.

Load a Saved Referenced Configuration Set

If your model uses a configuration reference to specify the model configuration, you need to load the referenced configuration set from a MAT-file or MATLAB script 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.

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 4-56.

Why is the Build Button Not Available for a Configuration Reference?

The Code Generation pane of the Configuration Parameters dialog box contains a **Build** button. Its availability depends on whether the configuration set displayed by the dialog box resides in a model or is a freestanding configuration set.

- When the pane displays a configuration set stored in a model, the **Build** button is available. Click it to generate and compile code for the model.
- When the pane displays a freestanding configuration set, the **Build** button is unavailable. The configuration set does not know which (if any) models link to it.

To provide the same capabilities whether a configuration set resides in a model or is freestanding, the Configuration Reference dialog box contains a **Build** button. This button has the same capability as its equivalent in the Configuration Parameters dialog box. It operates on the model that contains the configuration reference.

About Configuration Sets

In this section...
“What Is a Configuration Set?” on page 10-26
“What Is a Freestanding Configuration Set?” on page 10-27
“Model Configuration Preferences” on page 10-28

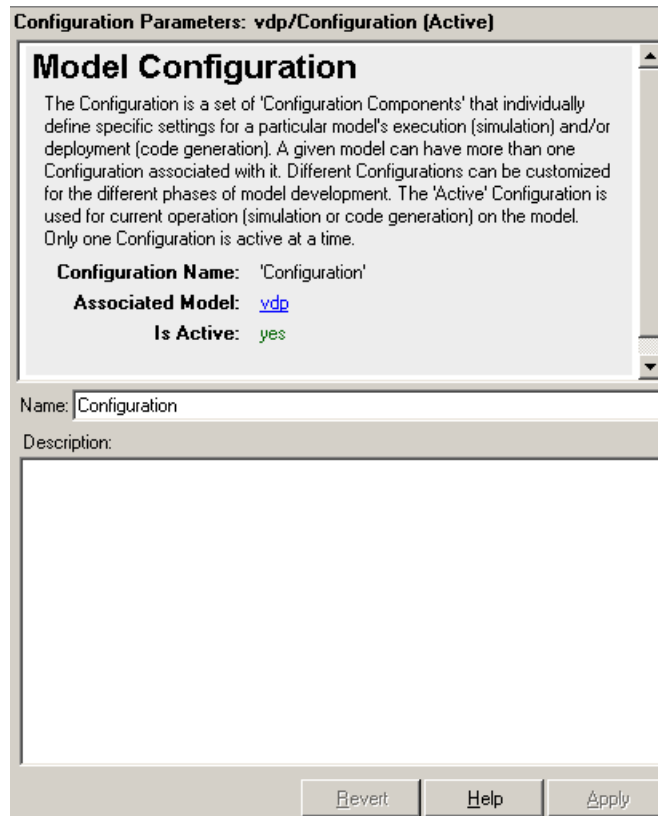
What Is a Configuration Set?

A configuration set comprises groups of related parameters called components. Every configuration set includes the following components:

- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target

Some MathWorks products that work with Simulink, such as Simulink Coder, define additional components. If such a product is installed on your system, the configuration set also contains the components that the product defines.

When you select any model configuration, the Model Configuration dialog appears in the Model Explorer **Dialog** pane. In this location, you can edit the name and description of your configuration.



What Is a Freestanding Configuration Set?

A freestanding configuration set is a configuration set object, `Simulink.ConfigSet`, stored in the base workspace. To use a freestanding configuration set as the configuration for a model, you must create a configuration reference in the model that references it. You can create a freestanding configuration set in the base workspace in these ways:

- Create a new configuration set object.
- Copy a configuration set that resides within a model to the base workspace.
- Load a configuration set from a MAT-file.

You can store any number of configuration sets in the base workspace by assigning each set to a different MATLAB variable.

Note Although you can store a configuration set in a model and point to it with a base workspace variable, such a configuration set is not freestanding. Using it in a configuration reference causes an error.

Model Configuration Preferences

Model Configuration Preferences are the preferred configuration for new models. You can change the preferred configuration by editing the settings in the Model Explorer.

- 1** Select **View > Show Configuration Preferences** to display the **Configuration Preferences** node in the expanded **Simulink Root** node.
- 2** Under the **Simulink Root** node, select **Configuration Preferences**. The Model Configuration Preferences dialog box opens in the **Dialog** pane.
- 3** In the **Contents** pane, select components and change any parameter values.

About Configuration References

In this section...
“What Is a Configuration Reference?” on page 10-29
“Why Use Configuration References?” on page 10-29
“Unresolved Configuration References” on page 10-30
“Configuration Reference Limitations” on page 10-31
“Configuration References for Models with Older Simulation Target Settings” on page 10-31

What Is a Configuration Reference?

A configuration reference in a model is a reference to a configuration set object in the base workspace. A model that has a configuration reference that points to a freestanding configuration set uses that configuration set when configuration reference is active. The model then has the same configuration parameters as if the referenced configuration set resides directly in the model.

You can attach any number of configuration references to a model. Each reference must have a unique name. For more information, see “Why Use Configuration References?” on page 10-29. For an example on how to use configuration references, see “Share a Configuration for Multiple Models” on page 10-4 or “Create and Attach a Configuration Reference” on page 10-18.

Tip Save or export the configuration set object. Otherwise, when you reopen your model 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 4-56.

Why Use Configuration References?

You can use configuration references and freestanding configuration sets to:

- **Assign the same configuration set to any number of models**

Each model that uses a given configuration set contains a configuration reference that points to a MATLAB variable. The value of that variable is a freestanding configuration set. All the models share that configuration set. Changing the value of any parameter in the set changes it for every model that uses the set. Use this feature to reconfigure many referenced models quickly and ensure consistent configuration of parent models and referenced models.

- **Replace the configuration sets of any number of models without changing the model files**

When multiple models use configuration references to access a freestanding configuration set, assigning a different set to the MATLAB variable assigns that set to all models. Use this feature to maintain a library of configuration sets and assign them to any number of models in a single operation.

- **Use different configuration sets for a referenced model used in different contexts without changing the model file**

A referenced model that uses different configuration sets in different contexts contains a configuration reference that specifies the referenced model configuration set as a variable. When you call an instance of the referenced model, Simulink software assigns that variable a freestanding configuration set for the current context.

Unresolved Configuration References

When a configuration reference does not reference a valid configuration set, the **Is Resolved** field of the Configuration Reference dialog box has the value no. If you activate an unresolved configuration reference, no warning or error occurs. However, an unresolved configuration reference that is active provides no configuration parameter values to the model. Therefore:

- Fields that display values known only by accessing a configuration parameter, like Stop Time in the model window, are blank.
- Trying to build the model, simulate it, generate code for it, or otherwise require it to access configuration parameter values, causes an error.

For more information, see “Resolve a Configuration Reference” on page 10-19.

Configuration Reference Limitations

- You cannot nest configuration references. Only one level of indirection is available, so a configuration reference cannot link to another configuration reference. Each reference must specify a freestanding configuration set.
- If you replace the base workspace variable and configuration set that configuration references use, execute `refresh` for each reference that uses the replaced variable and set. See “Use `refresh` When Replacing a Referenced Configuration Set” on page 10-41.
- 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” in the Simulink Coder documentation.

Configuration References for Models with Older Simulation Target Settings

Suppose that you have a nonlibrary model that contains one of these blocks:

- MATLAB Function
- Stateflow chart
- Truth Table
- Attribute Function

In R2008a and earlier, this type of nonlibrary model does not store simulation target (or `sfun`) settings in the configuration parameters. Instead, the model stores the settings outside any configuration set.

When you load this older type of model, the simulation target settings migrate to parameters in the active configuration set.

- If the active configuration set resides internally with the model, the migration happens automatically.

- If the model uses an active configuration reference to point to a configuration set in the base workspace, the migration process is different.

The following sections describe the two types of migration for nonlibrary models that use an active configuration reference.

Default Migration Process That Disables the Configuration Reference

Because multiple models can share a configuration set in the base workspace, loading a nonlibrary model cannot automatically change any parameter values in that configuration set. By default, these actions occur during loading of a model to ensure that simulation results are the same, no matter which version of the software that you use:

- A copy of the configuration set in the base workspace attaches to the model.
- The simulation target settings migrate to the corresponding parameters in this new configuration set.
- The new configuration set becomes active.
- The old configuration reference becomes inactive.

A warning message appears in the MATLAB Command Window to describe those actions. Although this process ensures consistent simulation results for the model, it disables the configuration reference that links to the configuration set in the base workspace.

Model Configuration Command Line Interface

In this section...

“Overview” on page 10-33

“Load and Activate a Configuration Set at the Command Line” on page 10-34

“Save a Configuration Set at the Command Line” on page 10-35

“Create a Freestanding Configuration Set at the Command Line” on page 10-36

“Create and Attach a Configuration Reference at the Command Line” on page 10-37

“Attach a Configuration Reference to Multiple Models at the Command Line” on page 10-38

“Get Values from a Referenced Configuration Set” on page 10-39

“Change Values in a Referenced Configuration Set” on page 10-39

“Obtain a Configuration Reference Handle” on page 10-40

“Use refresh When Replacing a Referenced Configuration Set” on page 10-41

Overview

An application programming interface (API) lets you create and manipulate configuration sets at the command line or in a script. The API includes the `Simulink.ConfigSet` and `Simulink.ConfigSetRef` classes and the following functions:

- `attachConfigSet`
- `attachConfigSetCopy`
- `detachConfigSet`
- `getConfigSet`
- `getConfigSets`
- `setActiveConfigSet`

- `getActiveConfigSet`
- `openDialog`
- `closeDialog`
- `Simulink.BlockDiagram.saveActiveConfigSet`
- `Simulink.BlockDiagram.loadActiveConfigSet`

These functions, along with the methods and properties of `Simulink.ConfigSet` class, allow you to create a script to:

- Create and modify configuration sets.
- Attach configuration sets to a model.
- Set the active configuration set for a model.
- Open and close configuration sets.
- Detach configuration sets from a model.
- Save configuration sets.
- Load configuration sets.

For examples using the preceding functions and the `Simulink.ConfigSet` class, see the function and class reference pages.

Load and Activate a Configuration Set at the Command Line

To load a configuration set from a MATLAB function or script:

- 1** Use the `getActiveConfigSet` or `getConfigSet` function to get a handle to the configuration set that you want to update.
- 2** Call the MATLAB function or execute the MATLAB script to load the saved configuration set.
- 3** Optionally, use the `attachConfigSet` function to attach the configuration set to the model. To avoid configuration set naming conflicts, set `allowRename` to `true`.

- 4 Optionally, use the `setActiveConfigSet` function to activate the configuration set.

Alternatively, to load a configuration set at the command line and make it the active configuration set:

- 1 Open the model.
- 2 Use the `Simulink.BlockDiagram.loadActiveConfigSet` function to load the configuration set and make it active.

Note If you load a configuration set with the same name as the:

- Active configuration set, the software overwrites the active configuration set.
 - Inactive configuration set that is associated with the model, the software detaches the inactive configuration from the model.
-

Save a Configuration Set at the Command Line

To save an active or inactive configuration set as a MATLAB function or script:

- 1 Use the `getActiveConfigSet` or `getConfigSet` function to get a handle to the configuration set.
- 2 Use the `saveAs` method of the `Simulink.Configset` class to save the configuration set as a function or script.

Alternatively, to save the active configuration set:

- 1 Open the model.
- 2 Use the `Simulink.BlockDiagram.saveActiveConfigSet` function to save the active configuration set.

Create a Freestanding Configuration Set at the Command Line

Copy a Configuration Set Stored in a Model

Create a freestanding configuration set to be referenced by a configuration reference in several models. You must copy any configuration set obtained from an existing model, otherwise, `cset` refers to the existing configuration set stored in the model, rather than a new freestanding configuration set in the base workspace. For example, use one of the following:

- ```
cset = copy (getActiveConfigSet(model))
```
- ```
cset = copy (getConfigSet(model, ConfigSetName))
```

`model` is any open model, and `ConfigSetName` is the name of any configuration set attached to the model.

Read a Configuration Set from a MAT-File

To use a freestanding configuration set across multiple MATLAB sessions, you can save it to a MAT-file. To create the MAT-file, you first copy the configuration set to a base workspace variable, then save the variable to the MAT-file:

```
save (workfolder/ConfigSetName.mat, cset)
```

`workfolder` is a working folder, `ConfigSetName.mat` is the MAT-file name, and `cset` is a base workspace variable whose value is the configuration set to save. When you later reopen your model, you can reload the configuration set into the variable:

```
load (workfolder/ConfigSetName.mat)
```

To execute code that reads configuration sets from MAT-files, use one of these techniques:

- The preload function of a top model

- The MATLAB startup script
- Interactive entry of load statements

Create and Attach a Configuration Reference at the Command Line

To create and populate a configuration reference, `Simulink.ConfigSetRef`, using the API:

- 1 Create the reference:

```
cref = Simulink.ConfigSetRef
```

- 2 Change the default name if desired:

```
cref.Name = 'ConfigSetRefName'
```

- 3 Specify the referenced configuration set:

```
cref.WSVarName = 'cset'
```

Tip Do not specify the name of a configuration reference. If you nest a configuration reference, an error occurs.

- 4 Attach the reference to a model:

```
attachConfigSet(model, cref, true)
```

The third argument is optional and authorizes renaming to avoid a name conflict.

Using a configuration reference with an invalid configuration set, `WSVarName`, generates an error and is called an unresolved configuration reference. The GUI equivalent of `WSVarName` is **Referenced configuration**. You can later use the API or GUI to provide the name of a freestanding configuration set. For more information, see “Unresolved Configuration References” on page 10-30.

Attach a Configuration Reference to Multiple Models at the Command Line

After you create a configuration reference and attach it to a model, you can attach copies of the reference to any other models. Each model has its own copy of any configuration reference attached to it, just as each model has its own copy of any attached configuration set.

If you use the GUI, attaching an existing configuration reference to another model automatically attaches a distinct copy to the model. If necessary to prevent name conflict, the GUI adds or increments a digit at the end of the name of the copied reference. If you use the API, be sure to copy the configuration reference explicitly before attaching it, with statements like:

```
cref = copy (getConfigSet(model, ConfigSetRefName))
attachConfigSet (model, cref, true)
```

If you omit the `copy` operation, `cref` becomes a handle to the original configuration reference, rather than a copy of the reference. Any attempt to use `cref` causes an error. If you omit the argument `true` to `attachConfigSet`, the operation fails if a name conflict exists.

The following example shows code for obtaining a freestanding configuration set and attaching references to it from two models. After the code executes, one of the models contains an internal configuration set and a configuration reference that points to a freestanding copy of that set. If the internal copy is an extra copy, you can remove it with `detachConfigSet`, as shown in the last line of the example.

```
open_system('model1')
% Get handle to local cset
cset = getConfigSet('model1', 'Configuration')

% Create freestanding copy; original remains in model
cset1 = copy(cset)

% In the original model, create a configuration
% reference to the cset copy
cref1 = Simulink.ConfigSetRef
cref1.WSVarName = 'cset1'
```

```

% Attach the configuration reference to the model
attachConfigSet('model1', cref1, true)

% In a second model, create a configuration
% reference to the same cset
open_system('model2')
% Rename if name conflict occurs
attachConfigSetCopy('model2', cref1, true)

% Delete original cset from first model
detachConfigSet('model1', 'Configuration')

```

Get Values from a Referenced Configuration Set

You can use `get_param` on a configuration reference to obtain parameter values from the linked configuration set, as if the reference object were the configuration set itself. Simulink software retrieves the referenced configuration set and performs the indicated `get_param` on it.

For example, if configuration reference `cref` links to configuration set `cset`, the following operations give identical results:

```

get_param (cset, 'StopTime')
get_param (cref, 'StopTime')

```

Change Values in a Referenced Configuration Set

By operating on only a configuration reference, you cannot change the referenced configuration set parameter values. If you execute:

```

set_param (cset, 'StopTime', '300')
set_param (cref, 'StopTime', '300')           % ILLEGAL

```

the first operation succeeds, but the second causes an error. Instead, you must obtain the configuration set itself and change it directly, using the GUI or the API.

To obtain a referenced configuration set using the API:

- 1 Follow the instructions in “Obtain a Configuration Reference Handle” on page 10-40.

2 Obtain the configuration set *cset* from the configuration reference *cref*:

```
cset = cref.getRefConfigSet
```

You can now use `set_param` on *cset* to change parameter values. For example:

```
set_param (cset, 'StopTime', '300')
```

Tip If you want to change parameter values through the GUI, execute:

```
cset.openDialog
```

The Configuration Parameters dialog box opens on the specified configuration set.

Obtain a Configuration Reference Handle

Most functions and methods that operate on a configuration reference take a handle to the reference. When you create a configuration reference:

```
cref = Simulink.ConfigSetRef
```

the variable *cref* contains a handle to the reference. If you do not already have a handle, you can obtain one by executing:

```
cref = getConfigSet(model, 'ConfigSetRefName')
```

ConfigSetRefName is the name of the configuration reference as it appears in the Model Explorer, for example, **Reference**. You specify this name by setting the **Name** field in the Configuration Reference dialog box or executing:

```
cref.Name = 'ConfigSetRefName'
```

The technique for obtaining a configuration reference handle is the same technique you use to obtain a configuration set handle. Wherever the same operation applies to both configuration sets and configuration references, applicable functions and methods overload to perform correctly with either class.

Use refresh When Replacing a Referenced Configuration Set

You can replace the base workspace variable and configuration set that a configuration reference uses. However, the pointer from the configuration reference to the configuration set becomes invalid. To avoid this situation, execute:

```
cref.refresh
```

cref is the configuration reference. If you do not execute “refresh”, the configuration reference continues to use the previous instance of the base workspace variable and its configuration set. This example illustrates the problem.

```
% Create a new configuration set
cset1 = Simulink.ConfigSet;

% Set a non-default stop time
set_param (cset1, 'StopTime', '500')

% Create a new configuration reference
cref1 = Simulink.ConfigSetRef;

% Resolve the configuration reference to the configuration set
cref1.WsVarName = 'cset1';

% Attach the configuration reference to an untitled model
attachConfigSet('untitled', cref1, true)

% Set the active configuration set to the reference
setActiveConfigSet('untitled','Reference')

% Replace config set in the base workspace
cset1 = Simulink.ConfigSet;

% Call to refresh is commented out!
% cref1.refresh;

% Set a different stop time
set_param (cset1, 'StopTime', '75')
```

If you simulate the model, the simulation stops at 500, not 75. Calling `cref1.refresh` as shown prevents the problem.

Configuring Models for Targets with Multicore Processors

- “Introduction to Concurrent Execution” on page 11-2
- “Design Considerations” on page 11-5
- “Modeling Process for Concurrent Execution” on page 11-11
- “Configure Your Model” on page 11-12
- “Analyze Baseline” on page 11-18
- “Customize Concurrent Execution Settings” on page 11-21
- “Interpret Simulation Results” on page 11-29
- “Build and Download to a Multicore Target” on page 11-35
- “Concurrent Execution Example Model” on page 11-52
- “Command-Line Interface for Concurrent Execution” on page 11-61

Introduction to Concurrent Execution

In this section...
“About Configuring Models for Concurrent Execution” on page 11-2
“Supported Targets” on page 11-3
“Helpful Terms” on page 11-3
“Software Requirements” on page 11-4

About Configuring Models for Concurrent Execution

Configuring your model for concurrent execution in the Simulink environment enables you to capture and simulate the effects of deploying your design to a multicore system. A model that is configured for concurrent execution gives you control over how a design should execute on a multicore target. It allows the design to be mapped into concurrently executing tasks. Subsequently, the Simulink environment enables the simulation of the mapped design by identifying data transfer points and by simulating the mapped design in conjunction with data transfer latencies. You can deploy the design to a multicore system using Simulink Coder, Embedded Coder, and xPC Target™ software.

You can use simulation results to identify whether a mapping scheme is suitable with respect to the functional characteristics of the design. You can also explore alternate partitioning schemes using a mapping interface. In addition, you can use profiling results on the deployed design to validate if the mapping scheme is efficiently using the multicore system.

- Create a new model configuration or extend existing configurations for concurrent execution.
- Use the Model block to define potential opportunities for concurrency in your design.
- Set up and configure concurrent on-target tasks using a task editing interface.
- Use either the GUI or command-line interface to iteratively map design partitions (based on Model blocks) to tasks and find optimal concurrent execution scenarios.

- Generate code that uses threading APIs on Windows, Linux, or xPC Target platforms for on-target execution.

Supported Targets

You can build and download concurrent execution models for the following targets using system target files:

- Linux, Windows, and Mac OS using `ert.tlc` and `grt.tlc`.
- xPC Target using `xpctarget.tlc` and `xpctargetert.tlc`.
- Linux and Windows using `idelink_ert.tlc` and `idelink_grt.tlc`.
- Embedded Linux and VxWorks® using `idelink_ert.tlc` and `idelink_grt.tlc`.

Note Deploying to embedded Linux and VxWorks targets requires the Embedded Coder product.

You cannot build and download a model, configured for concurrent execution, for field-programmable gate arrays (FPGAs) and programmable logic controllers (PLCs). For FPGA optimization and hardware utilization techniques, see “Speed and Area Optimization”.

Helpful Terms

Task — Object that corresponds to a thread of execution on a target. From within the Simulink environment, you can specify tasks, configure their properties, and map Model blocks to them.

Task configuration — Configuration properties that apply to the set of tasks set up for your multicore system.

Trigger — Abstraction of operating system timers, signals, interrupts, and system events.

Aperiodic trigger — Event trigger that has no inherent periodicity, such as an interrupt. When multiple triggers coexist, the software assumes that they are asynchronous to each other.

Periodic triggers — Periodic event trigger such as an operating system trigger.

Software Requirements

- To build and download your model, you must have Simulink Coder software installed.
- To build and download your model to an xPC Target system, you must have xPC Target software installed. You must also have a multicore target system supported by the xPC Target product.

Design Considerations

In this section...
“Modeling for Concurrency” on page 11-5
“Simulation Limitations” on page 11-9

Modeling for Concurrency

A model configured for concurrent execution is defined as a model designed to execute concurrently on a real-time multicore target. In this situation, different pieces of the model ultimately execute in concurrent threads on the target environment. Modeling for concurrent execution lets you factor in the effect of the concurrent execution semantics within the model, to simulate and provide initial conditions for data transfer points.

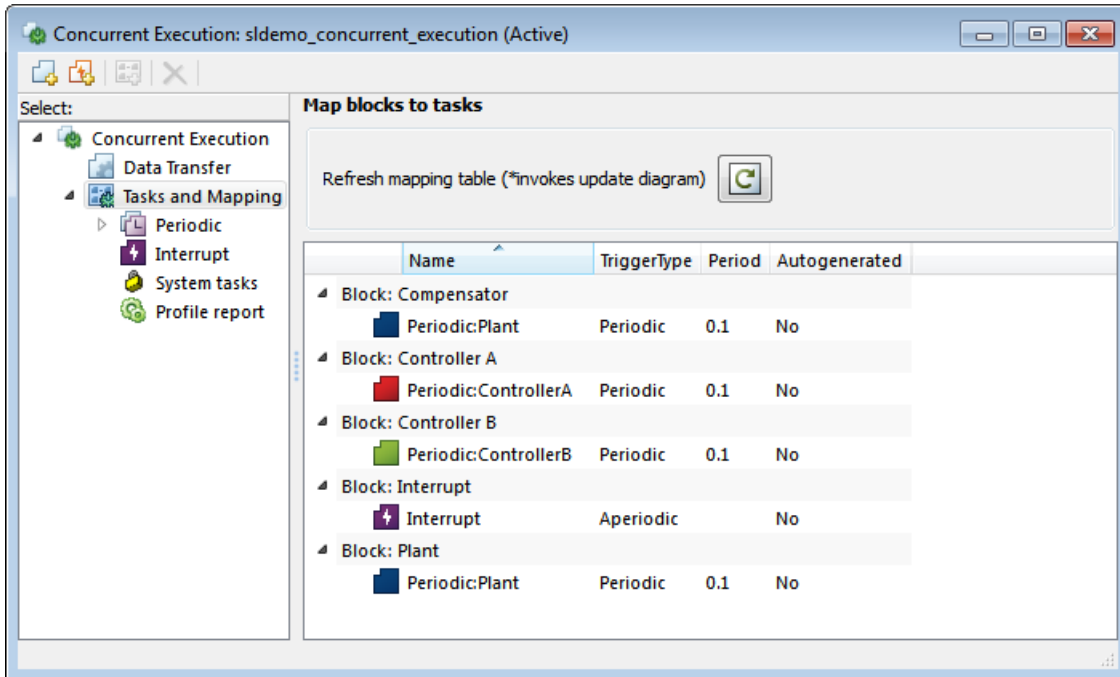
Note The simulation itself does not require a multicore host. The current simulation engine uses only one thread to simulate.

Modeling for concurrency is an iterative process. It ultimately deploys a model on a multicore target environment while making the best use of the computational power provided by the multiple cores. To support the iterative workflow, Simulink provides a mapping interface that allows you to map the potential concurrency in the design to the actual concurrency available on the target. A model configured for concurrent execution consists of three parts:

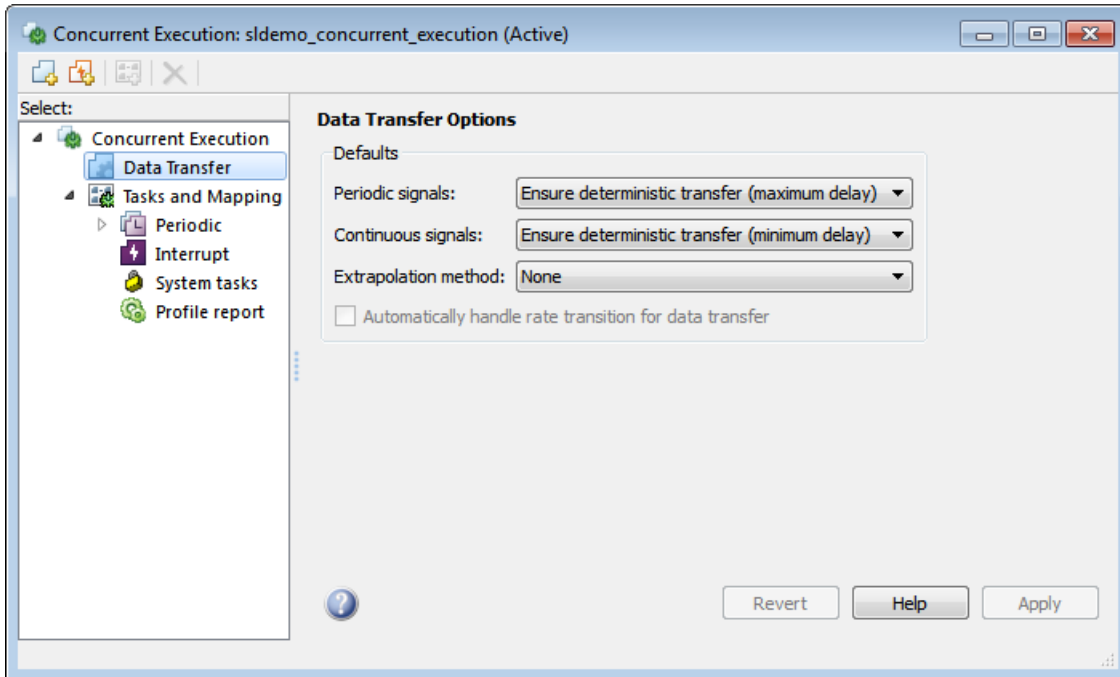
- A model that has identified blocks for concurrent execution
- A description of the available concurrency in terms of the number of tasks and their periodicities
- A mapping that places the identified blocks in the model into actual tasks for concurrent execution

These parts constitute a mapped model. To manage mapped models, Simulink software provides:

- Explicit control over task configuration and how blocks are mapped to tasks



- Control over how data is communicated between tasks



- Simulation of a mapped model that allows you to validate a desired functionality against the data transfer requirements of a particular mapping

To validate if a particular mapping is an acceptable design, first validate that the mapped model produces the desired simulation traces. Simulation results vary from mapping to mapping because each mapping imposes communication (data transfer) requirements that might cause the Simulink software to treat certain signal lines as delays for data transfer. Ultimately, however, you must evaluate the mapping on the target, and profile the behavior to measure whether the mapping meets the computational requirements and makes good use of the available computational power of the target platform.

Identifying Opportunities for Concurrent Execution

To use the software mapping interface, a model must first specify opportunities for concurrency by identifying blocks for concurrent execution. Use Model blocks (referenced models) to identify potential opportunities for

concurrent execution. A Model block defines a unit of functionality that the software can map for execution in one or more target tasks. More precisely, you can map a multirate Model block that contains N rates to exactly N tasks, the periodicity of which agree with the respective rates. You can map several Model blocks to the same task. Map a Model block that contains continuous blocks to a task, the period of which agrees with the fundamental sample time of the model. You can specify the fundamental sample time of the model in the **Fixed-step size (fundamental sample time)** of the **Solver** pane in the Configuration Parameters dialog box.

Because Model blocks must be used to express opportunities for concurrency, a design must consist entirely of Model blocks. You can also use virtual connectivity blocks to organize the block diagram, including:

- Goto and From blocks
- Ground and Terminator blocks
- Inport and Outport blocks
- Virtual subsystem blocks that contain permitted blocks

A model configured for concurrent execution reports an error diagnostic if any other blocks are detected. Place all other blocks in one or more referenced models.

Also consider signal lines when designing for concurrent execution. Depending on the mapping, any signal line can potentially be identified as a data transfer point. In particular, if the source port of a signal originates in a block from one task and the destination port is to a block in another task, the Simulink software identifies that signal as a data transfer point.

Algebraic Loops

An algebraic constraint or algebraic loop might impose tight coupling between tasks and can lead to severe computational penalties. Therefore, a model configured for concurrent execution cannot contain algebraic loops or algebraic constraints that originate from physical modeling in a referenced model. This condition makes an algebraic loop fully contained in a task.

However, the use of Model blocks might lead to artificial algebraic loops. This condition might occur if Simulink cannot identify if a break point is

encapsulated within a referenced model. In such situations, use one of the following methods to remove artificial algebraic loops:

- In the referenced model Configuration Execution dialog box, select **Model Referencing > Minimize algebraic loop occurrences**.
- Insert a Memory or Unit Delay block as the first block at the relevant input port of the referenced model.

Additionally, if the model is configured for the Embedded Coder target (`ert.tlc`), in the Configuration Parameters dialog box, clear the **Code Generation > Interface > Single output/update function** check box. This last operation prevents optimizations that can lead to artificial algebraic loops.

Simulation Limitations

When modeling 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
- Logging to MAT-files (**Configuration Parameters > Interface > 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.

In addition, for all Rate Transition blocks within the Model blocks:

- Select the **Ensure data integrity during data transfer** check box.

- Clear the **Ensure deterministic data transfer (maximum delay)** check box.

Modeling Process for Concurrent Execution

You can model for concurrent execution using the following general workflow. These steps assume that you have a model that meets the modeling guidelines in “Design Considerations” on page 11-5.

- 1** Optionally, save your original model to a temporary folder.
- 2** Configure your model for concurrent execution.
- 3** Perform baseline analysis of your model for concurrent execution.
- 4** Explicitly set configuration values in the concurrent execution configuration.
- 5** Simulate the model and evaluate the results
- 6** As necessary, refine the results.
- 7** Build and download the model to the multicore target and evaluate the results.
- 8** As necessary, refine the results.

Configure Your Model

You can use configuration references or configuration sets to configure a model for concurrent execution.

If Your Model Has...	Create...
A single-level model, or if it has multiple descendants, such as reference models, and the descendants do not share the same configuration settings.	A configuration set for a single model. For more information, see “Creating a Concurrent Execution Configuration Set” on page 11-12.
A top-level model and multiple descendants, such as referenced models, all of which share the same configuration settings.	A configuration reference object that can be shared between all models in the hierarchy. For more information, see “Creating and Propagating a Configuration Reference” on page 11-13.

Creating a Concurrent Execution Configuration Set

You can create a concurrent execution configuration set from an existing configuration set, or create a new configuration set:

- 1 In the Simulink model editor, select **View > Model Explorer**.

The Model Explorer window is displayed.

- 2 In the Model Explorer window:

To...	Do...
Preserve existing configuration settings for your model, convert an existing configuration set to a concurrent execution configuration set.	Expand the node for the model. Under the model, right-click Configuration , then select Show Concurrent Execution options .
Create new configuration settings, add a new concurrent execution configuration set.	Expand the node for the model. Right-click the model node and select Configuration > Add Configuration for Concurrent Execution . Activate the configuration set by right-clicking it and selecting Activate .

- In the third column of the Model Explorer window, the Solver pane is updated to display an **Allow tasks to execute concurrently on target** check box. Click the **Configure Tasks** button.

If you want to create a configuration reference, see “Creating and Propagating a Configuration Reference” on page 11-13. Otherwise, to begin configuring the model for concurrent execution, see “Analyze Baseline” on page 11-18.

Creating and Propagating a Configuration Reference

This topic assumes that you have an existing concurrent execution configuration. If you have not yet created one, see “Creating a Concurrent Execution Configuration Set” on page 11-12.

For more information on configuration references, see “About Configuration References” on page 10-29. This topic describes one way to create and propagate a configuration reference.

- In the Simulink model editor, select **View > Model Explorer**.

The Model Explorer window is displayed.

- 2** In the Model Explorer window, expand the node for the model and do one of the following:

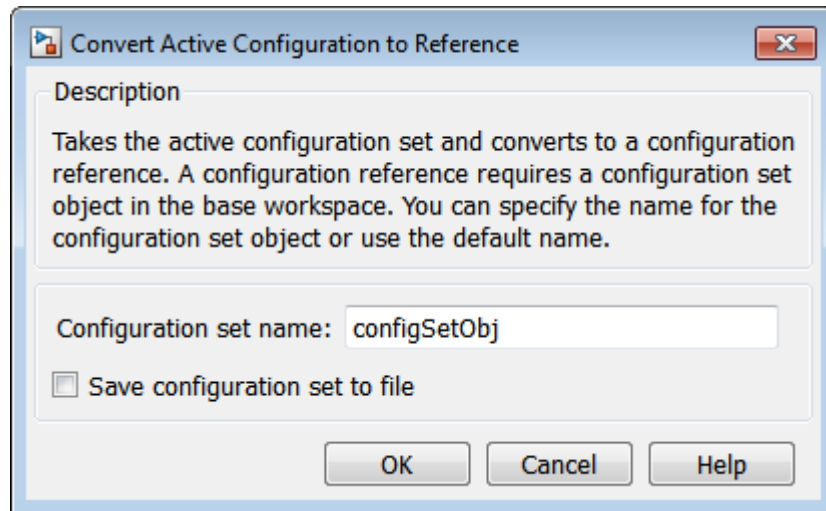
To...	Do...
Preserve existing configuration settings for your model, convert an existing configuration set to a concurrent execution configuration set.	Right-click and select Configuration > Show Concurrent Execution options.
Create new configuration settings, add a new concurrent execution configuration set.	Right-click and select Configuration > Add Configuration for Concurrent Execution.

- 3** Activate the configuration set by right-clicking it and selecting **Activate**.

- 4** Do one of the following:

- Right-click the model name and select **Configuration > Convert Active Configuration to Reference**.
- In the Model Hierarchy column, right-click **Configuration > Convert to Configuration Reference**.

A Convert Active Configuration to Reference dialog box is displayed.

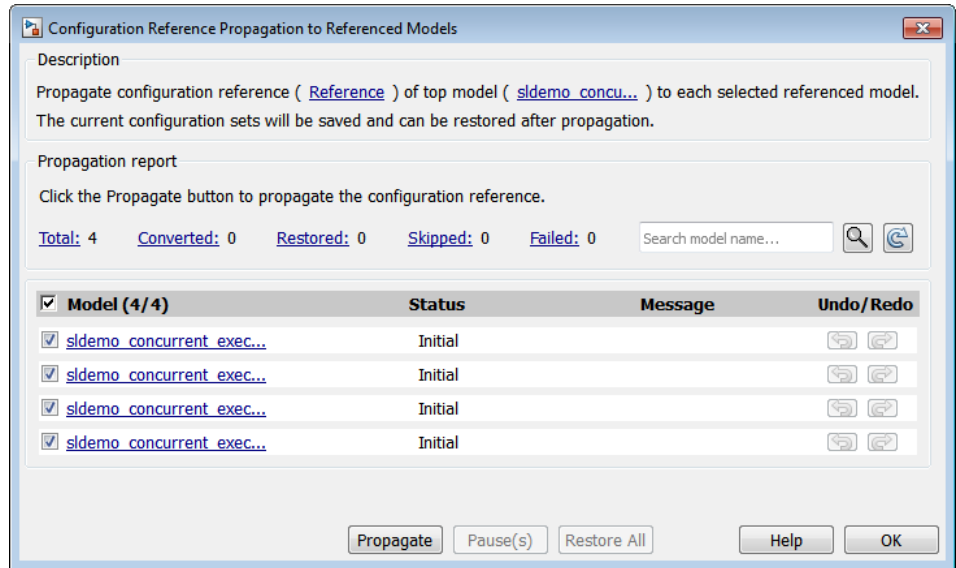


- 5 Click **OK** to accept the default name for the configuration reference.

The configuration is converted to a configuration reference.

- 6 To share the configuration reference with the other referenced models in the model hierarchy, activate the configuration reference, then right-click it and select **Configuration > Propagate to Referenced Models**.

A Propagate to Referenced Models dialog box is displayed.



7 Click **Propagate** to propagate the configuration reference.

A propagation confirmation dialog box is displayed.

8 Click **OK** to continue.

The hierarchy of each referenced model is updated with the configuration reference.

9 Click **OK**.

10 In the Model Explorer, in the left column, select the **Reference (Active)** node.

The right column changes to display the Configuration Parameters reference pane.

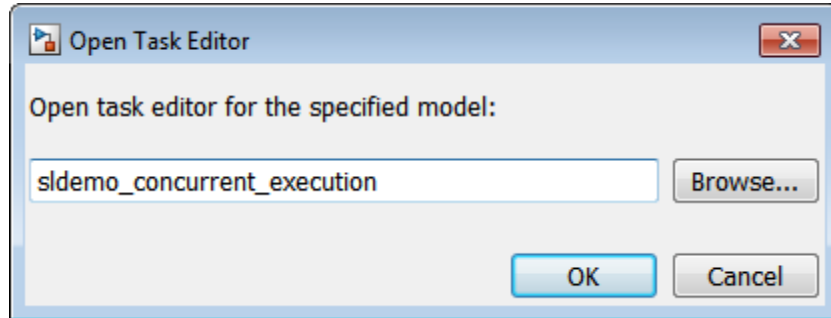
11 In the Configuration Parameters reference pane, click the **Open** button of the **Referenced configuration** parameter.

The Configuration Parameters dialog box is displayed.

12 Navigate to the Solver node.

- Click the **Allow tasks to execute concurrently on target** check box and click **Configure Tasks** button.

The Open Task Editor dialog window is displayed.



Consider saving the configuration reference to MAT-file if you want to use it in future MATLAB sessions. To save the configuration reference named `configSetObj`, enter the following command in the MATLAB Command Window:

```
save configsetobj.mat configSetObj
```

To load the saved configuration reference into another MATLAB session, enter the following command in the MATLAB Command Window:

```
load configsetobj.mat
```

To begin configuring the model for concurrent execution, see “Analyze Baseline” on page 11-18.

Analyze Baseline

Perform a baseline analysis of the model. This step helps you identify initial bottlenecks in the execution of the original model. This topic assumes that you have configured your model for concurrent execution, but have not yet defined tasks for your model (“Configure Your Model” on page 11-12). Simulink can perform a sample time analysis of your model and create system tasks for you.

1 In the Concurrent Execution dialog box, click the System tasks node.

2 In the System tasks pane, click the **Autogenerate tasks and mapping**

button .

Simulink updates the model block diagram and performs a sample time analysis. For each sample time, the software creates a system task. When done, Simulink:

- Creates one periodic system task (*DiscreteM*) for each periodic sample time in the model
- Creates one aperiodic system trigger (*AsynchronousM*) for each function-call input in the model
- Configures the block-to-task mapping so that each block maps to a system task based on sample time analysis
- Updates the **System tasks** pane with a table of the system tasks


3 Customize and save these new tasks.

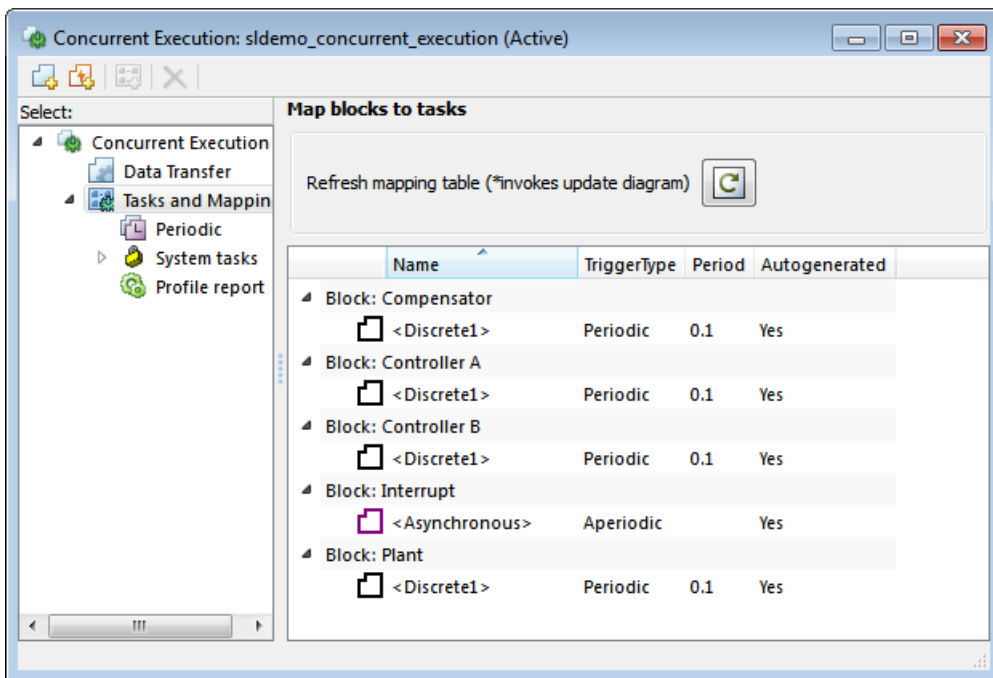
To customize system tasks (for example, name them, make platform-specific changes, and so forth), or save them, convert them to user tasks by right-clicking the system task or trigger and clicking **Convert to editable**

Note Saving the model does not save autogenerated system tasks, only those converted to editable. Be sure to convert system tasks to editable if you want to save them.

The software performs a sample time analysis on the model as it currently exists.

If the software reports a diagnostic during the update phase that prevents the analysis from completing, address the modeling issues before continuing.

The following is the automatic baseline analysis for the `sldemo_concurrent_execution` model. Notice that system tasks appear as empty task icons  in the dialog box.



When the analysis is complete, decide on your next step.

To...	Go to...
Explicitly configure how data is transferred to override the settings detected by automatic analysis.	“Customize Concurrent Execution Settings” on page 11-21.
Simulate the model.	“Interpret Simulation Results” on page 11-29.

Customize Concurrent Execution Settings

After you set perform sample time analysis and create and assign system tasks, you might want to fine-tune simulation results by performing operations such as explicitly creating a different number of tasks or setting data transfer parameters.

Note If you do not want to customize concurrent execution settings, in the **Concurrent Execution** pane clear the **Enable explicit model partitioning for concurrent behavior** check box. Clearing this check box allows the model to use rate-based tasks and ignores task-to-block mappings.

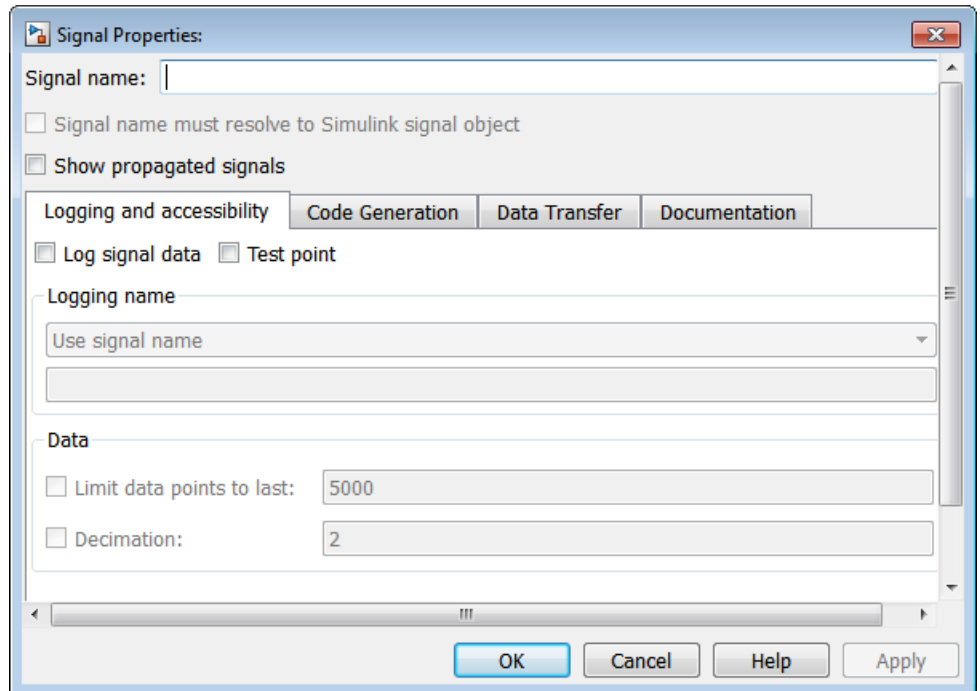
Configuring Data Transfer Communications

Use the **Data Transfer Options** pane to define communications between tasks. The settings you make here are the values used for the Use global setting option of the Data Transfer tab of the Signal Properties dialog box. You can also override these options for each signal from the **Data Transfer** pane of the Signal Properties dialog box.

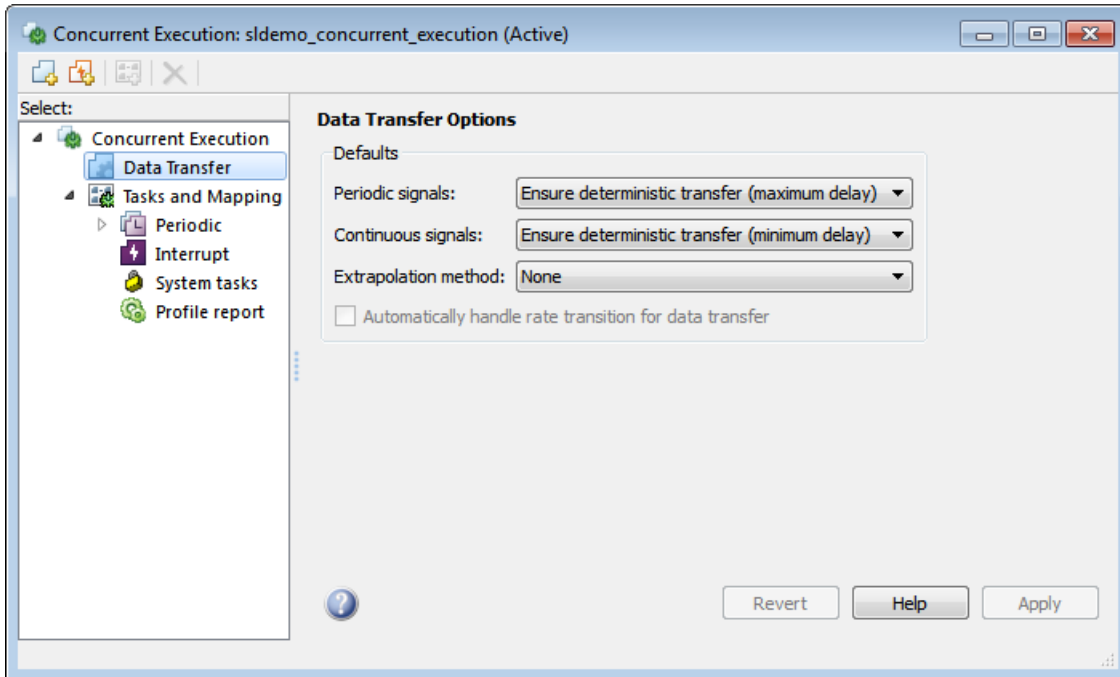
Data Transfer Options

Data Transfer Type	Simulation	Deployment
Ensure data integrity only.	Data transfer is simulated using a one-step delay.	Simulink Coder ensures data integrity during data transfer. Data is transferred as soon as possible.
Ensure deterministic transfer (maximum delay).	Data transfer is simulated using a one-step delay.	Simulink Coder ensures data integrity during data transfer. Data is transferred as soon as possible. In addition, Simulink Coder ensures data transfer is identical with simulation.
Ensure deterministic transfer (minimum delay).	Data transfer occurs within the same step.	

- For continuous signals, the Simulink software uses extrapolation methods to compensate for numerical errors that were introduced due to delays and discontinuities in data transfer.
- For signals that are configured for **Ensure Data Integrity Only** and **Ensure deterministic transfer (maximum delay)** data transfers, you might need to provide an initial condition to avoid numerical errors. 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 on page 11-21 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.

Configuring Periodic Tasks

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 on 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.

When the periodic tasks are complete, configure the aperiodic (interrupt) tasks as necessary. If you do not need aperiodic tasks, continue to “Map Blocks to Tasks Incrementally” on page 11-26.

Configuring Aperiodic Triggers and Tasks

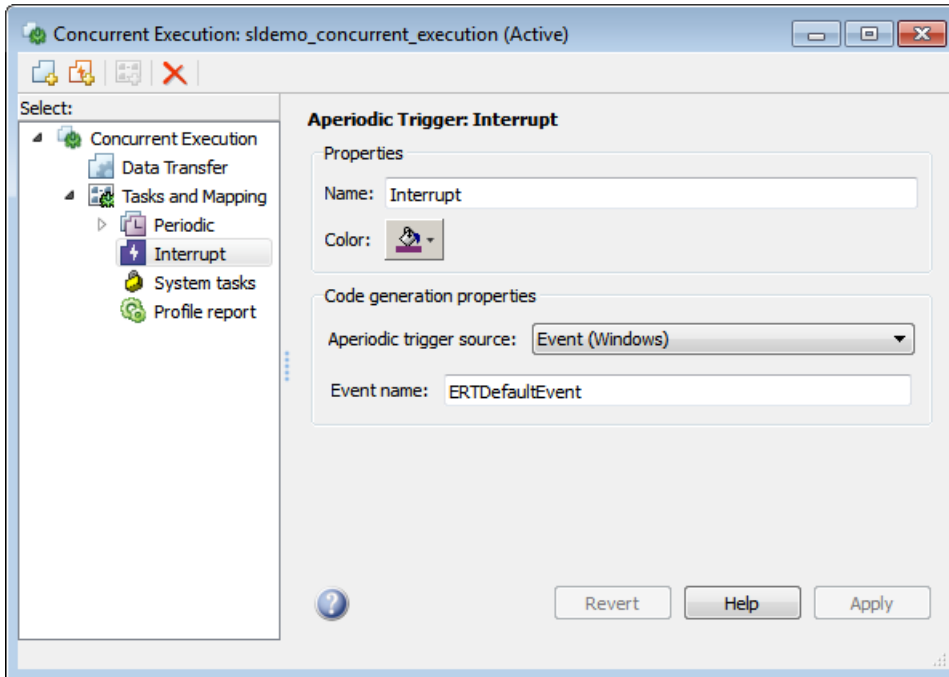
1 To create an aperiodic trigger, in the Concurrent Execution dialog box, right-click the Concurrent Execution node and click the **Add aperiodic trigger** icon.

A node named **InterruptN** appears in the configuration tree hierarchy.

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 the system can respond to the trigger. 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.**

You can delete tasks (both periodic and aperiodic) as well as triggers by right-clicking them in the pane and selecting **Delete**.

When the aperiodic tasks are complete, continue to “Map Blocks to Tasks Incrementally” on page 11-26.

Map Blocks to Tasks Incrementally

After you simulate and evaluate a model that has baseline analysis settings, you can continue to add new Model blocks, change the model, or redo the block-to-task mapping. You do this in one of the following ways:

- Use system tasks to incrementally map blocks to tasks. For more information, see “Map Incrementally” on page 11-26. You can partially map blocks to tasks, assigning tasks only for the blocks you are interested in, then use system tasks to complete the mapping so that you do not need to manually map all blocks to all tasks.
- Manually map blocks to tasks. For more information, see “Mapping Blocks to Tasks Manually” on page 11-27. You can manually map tasks for the blocks you are interested in, then use system tasks to incrementally map blocks to tasks to complete the mapping.

Map Incrementally

- 1 In the Concurrent Execution dialog box, click the Tasks and Mapping node.

The **Map blocks to tasks** pane appears. If you added a new Model block to your model, the new block appears in the table with a **select task** entry under it.

- 2 If you changed the model by adding or removing blocks and have not yet added new tasks and triggers for them, perform a sample time analysis, as described in “Analyze Baseline” on page 11-18.

The software performs a sample time analysis on the model as it currently exists.

- 3 Save the model.

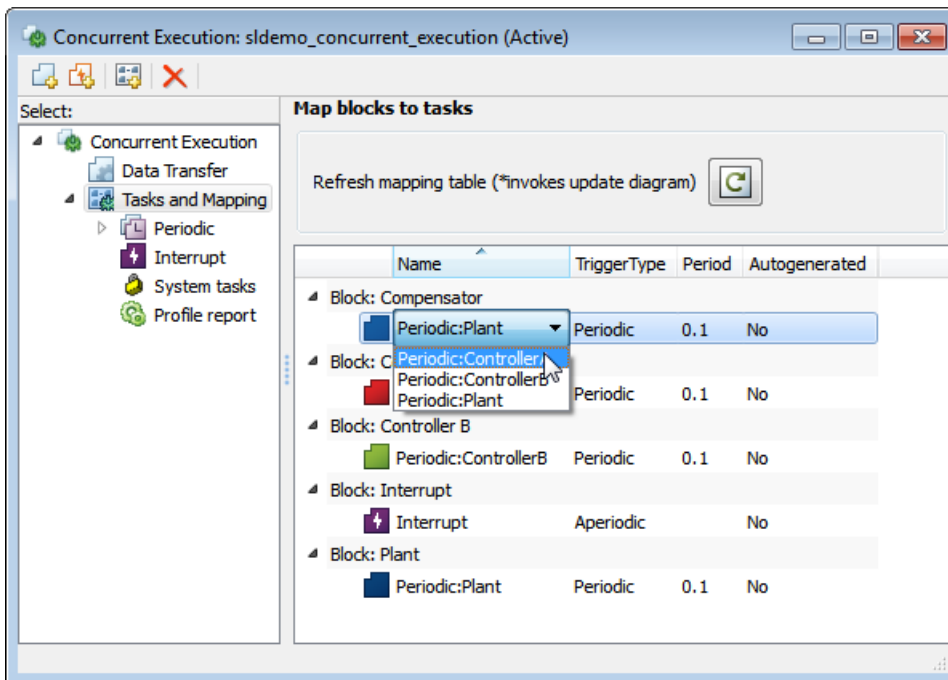
When the mapping is complete, simulate the model again (see “Interpret Simulation Results” on page 11-29).

Mapping Blocks to Tasks Manually

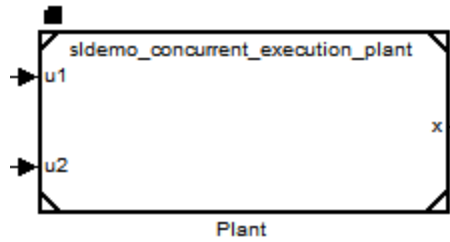
- 1 In the Concurrent Execution dialog box, click the Tasks and Mapping node.

The **Map blocks to tasks** pane appears. If you added a new 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 new 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 a task from the list. For example:



The block-to-task mapping icon appears on the top-left corner of the Model block. If a Model block is assigned to multiple tasks, multiple task icons are displayed in the top-left corner. For example:



4 Click **Apply**.

When the mapping is complete, simulate the model again (see “Interpret Simulation Results” on page 11-29).

Interpret Simulation Results

In this section...

“Introduction” on page 11-29

“Model with System Tasks” on page 11-30

“Sample Configured Model with Multiple Target Tasks” on page 11-31

“How Simulink Determines Data Transfer Requirements” on page 11-34

Introduction

A model configured for concurrent execution is a model that is designed to execute concurrently on a real-time multicore target. Simulink enables you to simulate such models and observe the effects of task-to-task data transfer. You can also observe how this transfer can impact the numerical characteristics of the model.

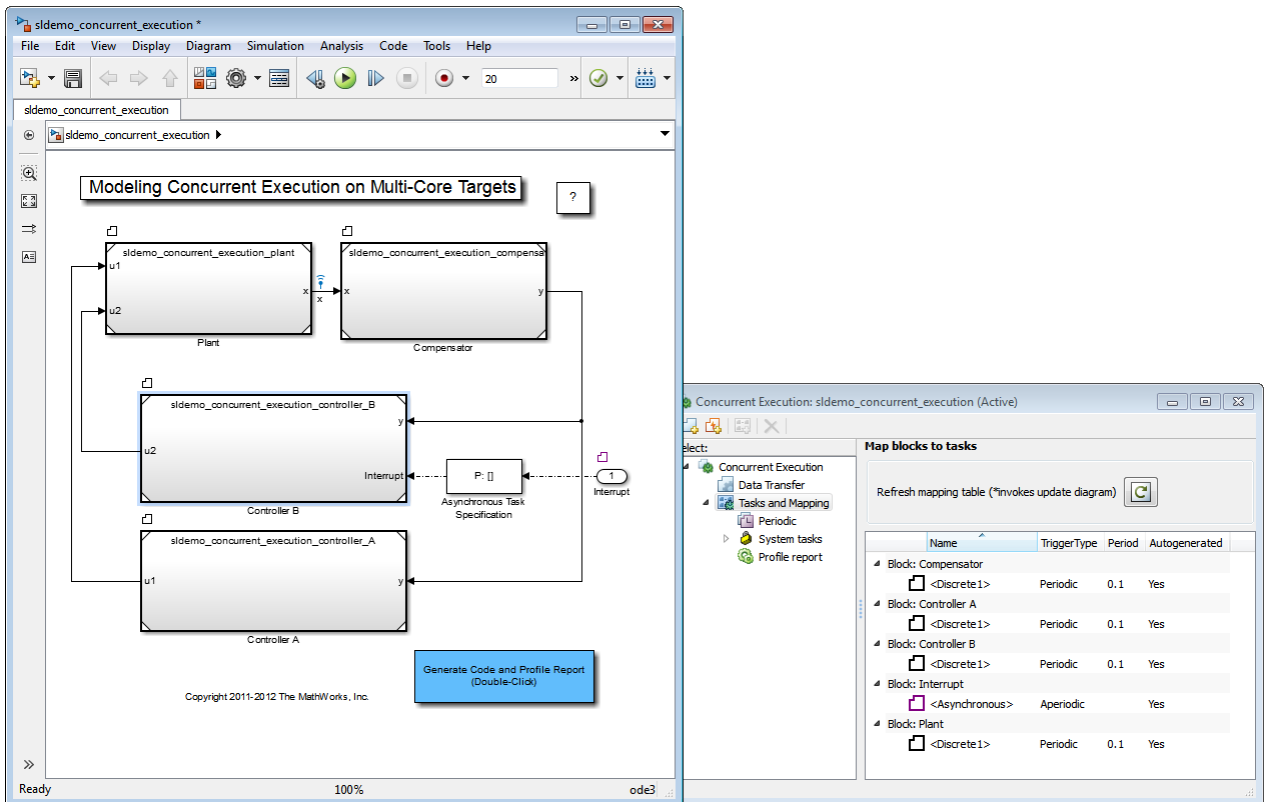
Note The simulation itself does not require a multicore target. The number of tasks that you use to simulate the design is determined by other factors and does not impose any requirements on the number of tasks modeled by the design. Factors that influence the concurrency of the simulation include the use of the Basic Linear Algebra Subprograms (BLAS) library, Rapid Accelerator mode, and the MATLAB `parfor` command.

Consider the model `sldemo_concurrent_execution`. To understand simulation results, compare the signal x in two scenarios:

- Baseline configuration — The target has exactly one periodic task and all blocks are mapped to execute within this task. In this configuration, the target does not allow for any concurrent execution.
- Sample configuration — The target has three periodic tasks and all blocks are mapped to execute concurrently in these tasks.

Model with System Tasks

In the following example, the model is configured as described in “Analyze Baseline” on page 11-18. Because this model contains only one sample time (continuous sample time), the sample time analysis configures the target with one task and maps all blocks to run in that task. The period of the task is selected as 0.1, which agrees with the fixed-step size of the model. Because the design has only one task, no task-to-task data transfer is needed. Therefore, simulation of the model should produce the same numeric results as if the model is not configured for concurrent execution.



Note Configuring a model with sample time analysis system tasks always configures a model to minimize task-to-task communication requirements. In particular, only those communications that require rate transitions are included in the design.

Configure the model to inspect data using the Simulation Data Inspector by clicking the **Record & Inspect Simulation Output** button in the toolbar. (Alternatively, in the Configuration Parameters dialog box, select the **Data Import/Export > Record and inspect simulation output** check box). After you enable the Simulation Data Inspector, click **Simulate > Run** to simulate the model using the single task configuration. After simulation, launch the Simulation Data Inspector as directed in the editor. Observe that the tool has recorded the simulation results for the signal labeled x.

Sample Configured Model with Multiple Target Tasks

In the Concurrent Execution dialog box, create two additional tasks and map the ControllerA block to the first additional task and the ControllerB block to the second additional task.

- 1 Open the Concurrent Execution dialog box.
- 2 Click the **Add Task** button twice.
- 3 Select the first added task and label it ControllerA.
- 4 Select the second added task and label it ControllerB.
- 5 In the **Map blocks to tasks** pane, check that the ControllerA block is assigned to execute within the ControllerA task and the ControllerB block is assigned to execute within the ControllerB task.

The screenshot shows a Simulink model titled "Modeling Concurrent Execution on Multi-Core Targets". The model consists of several interconnected blocks:

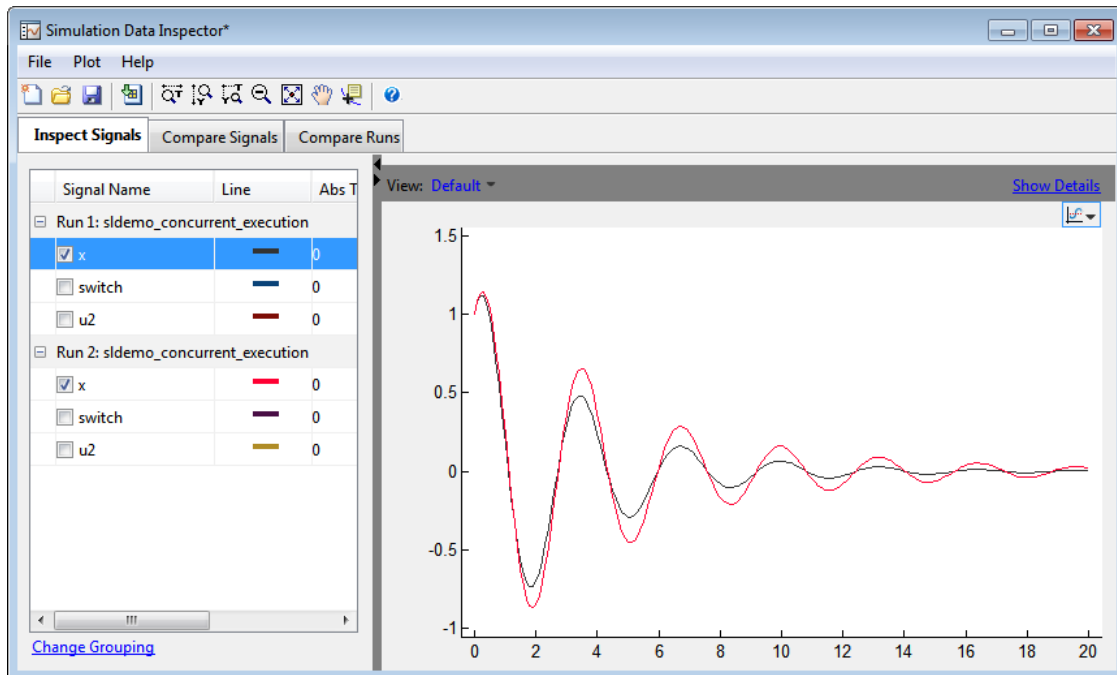
- Plant**: Receives inputs u_1 and u_2 and outputs x .
- Compensator**: Receives x and outputs y .
- Controller B**: Receives y and outputs u_2 .
- Controller A**: Receives y and outputs u_1 .
- Asynchronous Task Specification (P: [])**: Receives an **Interrupt** signal and outputs to the Plant and Compensator.

The "Concurrent Execution: sldemo_concurrent_execution (Active)" window is open, showing the "Map blocks to tasks" configuration. The table below lists the mapping of blocks to tasks:

Name	TriggerType	Period	Autogenerated
Block: Compensator	Periodic:Plant	0.1	No
Block: Controller A	Periodic:ControllerA	0.1	No
Block: Controller B	Periodic:ControllerB	0.1	No
Block: Interrupt	Interrupt		No
Block: Plant	Periodic:Plant	0.1	No

6 Simulate the model again.

7 After simulation completes, compare the results for the baseline and custom configurations using the Simulation Data Inspector.



To understand the source of the phase margin, observe that:

- For the first run of the model, signal x , identified by the black line, shows a default task configuration.
- In the second run of the model, signal x , identified by the red line, shows a custom task configuration and the effects of communication latencies from lines crossing task boundaries. This is different from the first run, which has only one periodic task and therefore, no communication latencies.
- The default setting for data transfer is to ensure determinism, so the data transfer is deterministic (see Data Transfer Options on page 11-21).

Because the data transfer is deterministic, the simulated model takes into account a unit delay to capture the effects of data transfer. This delay causes a small phase margin in the mapped design. The delay agrees with the step size of the model, which is 0.1. This delay is the least possible delay that the software can impose on the signal.

How Simulink Determines Data Transfer Requirements

The software determines data transfer based on how the blocks are mapped to tasks. When a model is simulated, either from an update diagram or upon code generation, Simulink software examines the block-to-task mapping and determines which signals must be configured for task-to-task data transfer. The software considers only the signals where the source of the signal originates from one task and the destination of the signal resides in a different task. You can specify global options that control data transfer (see Data Transfer Options on page 11-21). You can also override these options for each signal from the **Data Transfer** pane of the Signal Properties dialog box.

Therefore, the block-to-task mapping dictates which blocks execute in which tasks. Consequently, this mapping also dictates which signals you configure for task-to-task data transfer.

Build and Download to a Multicore Target

In this section...

“Generating Code” on page 11-35

“Configuring Embedded Coder for Native Threads Example” on page 11-45

“Build and Download” on page 11-46

“Profile and Evaluate” on page 11-47

“Generate Profile Report” on page 11-48

Generating Code

To generate code, in the Simulink editor window, select **Code > C/C++ Code > Build Model**. This code can run concurrently on a multicore target with Simulink Coder or Embedded Coder software. The coder generates all target-dependent code for thread creation, thread synchronization, interrupt service routines, and signal handlers and data transfer. The `main()` program contains these threads. The source files of the mapped model (for example, `model.c` and `model.h`) do not contain target-dependent code for setting up the execution of threads. However, target-dependent code might exist for data transfer types, which generate target-specific data protection or semaphore application programming interface (API) calls.

For each periodic task, Simulink Coder combines the output and update methods of the blocks mapped to that task and binds these methods to a target-specific thread. Additionally, for each periodic task that contains continuous states, a separate solver instance is generated in the source file of the mapped model. This mapped model can concurrently run the continuous parts of the model. (The generated solver instances share the solver type specified in the **Solver** of the Configuration Parameters dialog box of the mapped model.)

For each aperiodic task or aperiodic trigger, Simulink Coder combines the output and update methods of the blocks mapped to that task and binds these to a target-specific Interrupt Service Routine (ISR) or to a target-specific event handler.

Following is an example of generated code in the source file of a mapped model. This code shows the functions generated for the mapped model:

- Two solver instances
- Four periodic tasks with two of them having continuous states
- Two aperiodic tasks
- Two aperiodic triggers

```
/*
 * This function updates continuous states using the ODE3 fixed-step
 * solver algorithm
 */
static void Periodic_Task1_rt_ertODEUpdateContinuousStates(RTWSolverInfo *si )
{ }

/*
 * This function updates continuous states using the ODE3 fixed-step
 * solver algorithm
 */
static void Periodic_Task2_rt_ertODEUpdateContinuousStates(RTWSolverInfo *si )
{ }

/* OutputUpdate for Task: Periodic_Task1 */
void Periodic_Task1_step(void) /* Sample time: [0.1s, 0.0s] */
{
    Periodic_Task1_rt_ertODEUpdateContinuousStates(&task_M[0]->solverInfo);
}

/* OutputUpdate for Task: Periodic_Task2 */
void Periodic_Task2_step(void) /* Sample time: [0.2s, 0.0s] */
{
    Periodic_Task2_rt_ertODEUpdateContinuousStates(&task_M[1]->solverInfo);
}

/* OutputUpdate for Task: Periodic_Task3 */
```

```
void Periodic_Task3_step(void)           /* Sample time: [0.1s, 0.0s] */
{ }

/* OutputUpdate for Task: Periodic_Task4 */
void Periodic_Task4_step(void)         /* Sample time: [0.2s, 0.0s] */
{ }

/* OutputUpdate for Task:Interrupt1_asyncTask1 */
void Interrupt1_asyncTask1(void)
{ }

/* OutputUpdate for Task:Interrupt2_asyncTask2 */
void Interrupt2_asyncTask2(void)
{ }

/* OutputUpdate for Task:Interrupt3 */
void Interrupt3(void)
{ }

/* OutputUpdate for Task:Interrupt4 */
void Interrupt4(void)
{ }
```

In addition, for continuous signals, the coder produces code for the extrapolation method that you select. There are four types of extrapolation methods: zero-order hold, linear, quadratic, and none. None is an ideal case when the two communicating tasks synchronize in minor steps.

The following code snippets contain the generated task functions in the source file of the mapped model. The `sldemo_concurrent_execution` model has been modified so that:

- ControllerA/u1 uses Ensure data Integrity only option in the Signal Properties dialog box.
- Plant is remapped to a new task that executes concurrently with Compensator. The data transfer from Plant to Compensator is Ensure deterministic transfer (minimum delay).

The coder makes sure that the data transfer between concurrently executing tasks behave as described Data Transfer Options on page 11-21.

Generate Code to Share Data in a Concurrent Environment

For Ensure data integrity only, the generated code allows data to be shared in a concurrent environment. This is achieved with a double-buffer algorithm (for same rate data transfer) and a triple buffer algorithm (for rate-transition data transfer) and target-specific protection code for the buffer flag management. The following example shows code generated using the `sldemo_concurrent_execution` model:

- 1 Open the Signal Properties dialog box for ControllerA/u1 and select Data Integrity Only as the data transfer handling option from the **Data Transfer** pane.
- 2 To generate code, select **Code > C/C++ Code > Build Model**.

The coder generates code like the following. This code is for ControllerA/u1 on the Linux platform.

```
/* Output for Task: Periodic_ControllerA */
void Periodic_ControllerA_output(void) /* Sample time: [0.1s, 0.0s] */
{

    /* TaskTransBlk: '<Root>/TaskTransAtController AOut1' */
    sldemo_concurrent_execution_DWork.
        TaskTransAtControllerAOut1_buf_3[sldemo_concurrent_execution_DWork.fw_buf_3]
        = sldemo_concurrent_execution_B.ControllerA;
    rtw_pthread_mutex_lock(sldemo_concurrent_execution_DWork.mw_buf_3);
    sldemo_concurrent_execution_DWork.fw_buf_3 = 1 -
        sldemo_concurrent_execution_DWork.fw_buf_3;
    rtw_pthread_mutex_unlock(sldemo_concurrent_execution_DWork.mw_buf_3);
}

/* Output for Task: Periodic_Plant */
void Periodic_Plant_output(void) /* Sample time: [0.0s, 0.0s] */
{

    if (rtmIsMajorTimeStep(task_M[2])) {
        /* TaskTransBlk: '<Root>/TaskTransAtPlantIn1' */
        rtw_pthread_mutex_lock(sldemo_concurrent_execution_DWork.mw_buf_3);
```



```

        wrBufIdx = 1 - sldemo_concurrent_execution_DWork.fw_buf_3;
        rtw_pthread_mutex_unlock(sldemo_concurrent_execution_DWork.mw_buf_3);
        sldemo_concurrent_execution_B.TaskTransAtPlantIn1 =
            sldemo_concurrent_execution_DWork.TaskTransAtControllerAOut1_buf_3[wrBufIdx];

    }

}

void MdlInitialize(void)
{

    sldemo_concurrent_execution_DWork.fw_buf_3 = 0;
    rtw_pthread_mutex_init(&sldemo_concurrent_execution_DWork.mw_buf_3);

}

```

Generate Code for Maximum Capacity During Data Transfer

For Ensure deterministic transfer (maximum delay), the generated code for the task transition is ANSI[®] C code with writing and reading to and from double buffers. In the Signal Properties dialog box, you provide a value for the **Initial Condition** value that initializes the buffer used for the first read operation.

The software generates code like the following. This example is for ControllerB/u1 on the Linux platform.

```

/* Output for Task: Periodic_ControllerB */
void Periodic_ControllerB_output(void) /* Sample time: [0.1s, 0.0s] */
{

    sldemo_concurrent_execution_DWork.
        TaskTransAtControllerBOut1_buf_4[sldemo_concurrent_execution_DWork.fw_buf_4]
        = sldemo_concurrent_execution_B.ControllerB;
    sldemo_concurrent_execution_DWork.fw_buf_4 = 1 -
        sldemo_concurrent_execution_DWork.fw_buf_4;
}

/* Output for Task: Periodic_Plant */
void Periodic_Plant_output(void) /* Sample time: [0.0s, 0.0s] */

```

```
{

    if (rtmIsMajorTimeStep(task_M[2])) {

        /* TaskTransBlk: '<Root>/TaskTransAtPlantIn2' */
        sldemo_concurrent_execution_B.TaskTransAtPlantIn2 =
            sldemo_concurrent_execution_DWork.
            TaskTransAtControllerB0ut1_buf_4[sldemo_concurrent_execution_DWork.fr_buf_4];
        sldemo_concurrent_execution_DWork.fr_buf_4 = 1 -
            sldemo_concurrent_execution_DWork.fr_buf_4;
    }

}

void MdlInitialize(void)
{

    /* InitializeConditions for TaskTransBlk: '<Root>/TaskTransAtController B0ut1' */
    sldemo_concurrent_execution_DWork.fw_buf_4 = 0;

}
```

Generated Code for Maximum Data Integrity During Data Transfer

For Ensure deterministic transfer (minimum delay), the generated code for the data transfer makes sure that the task reading the data is waiting for the task writing the data to complete its computation of the data. This behavior produces target-specific code for data synchronization (for example, semaphores).

The software generates code like the following. This example code is for Plant/x on the Linux platform.

```
/*
```

```

* This function updates continuous states using the ODE3 fixed-step
* solver algorithm
*/
static void Periodic_Compensator_rt_ertODEUpdateContinuousStates(RTWSolverInfo
*si )
{

/* TaskTransBlk: '<Root>/TaskTransAtCompensatorIn1' */
if (sldemo_concurrent_execution_DWork.br_buf_5) {
sldemo_concurrent_execution_DWork.br_buf_5 = FALSE;
} else {
rtw_pthread_sem_wait(sldemo_concurrent_execution_DWork.sw_buf_5);
sldemo_concurrent_execution_B.TaskTransAtCompensatorIn1 =
sldemo_concurrent_execution_DWork.TaskTransAtPlantOut1_buf_5;
rtw_pthread_sem_post(sldemo_concurrent_execution_DWork.sr_buf_5);
if (rtmIsMajorTimeStep(task_M[0])) {
sldemo_concurrent_execution_DWork.br_buf_5 = TRUE;
}
}
/* End of TaskTransBlk: '<Root>/TaskTransAtCompensatorIn1' */

}
/* Output for Task: Periodic_Plant */
void Periodic_Plant_output(void) /* Sample time: [0.0s, 0.0s] */
{

if (rtmIsMajorTimeStep(task_M[3])) {

/* TaskTransBlk: '<Root>/TaskTransAtPlantOut1' */
rtw_pthread_sem_wait(sldemo_concurrent_execution_DWork.sr_buf_5);
sldemo_concurrent_execution_DWork.TaskTransAtPlantOut1_buf_5 =
sldemo_concurrent_execution_B.x;
rtw_pthread_sem_post(sldemo_concurrent_execution_DWork.sw_buf_5);
}
}
void MdlInitialize(void)
{

/* InitializeConditions for TaskTransBlk: '<Root>/TaskTransAtPlantOut1' */
sldemo_concurrent_execution_DWork.fw_buf_5 = 0;
rtw_pthread_sem_create(&sldemo_concurrent_execution_DWork.sw_buf_5, 0);

```

}

Threading APIs

For Embedded Coder targets and generic real-time targets, the generated code from a mapped model creates a thread for each task and automatically leverages the threading APIs supported by the operating system running on the host machine.

- If the host machine is a Windows platform, the generated code will create and run Windows threads.
- If the host machine is Linux or Mac OS platform, the generated code will create and run POSIX pthreads.

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
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

Aspect of Concurrent Execution	Linux Implementation	Windows Implementation	Mac OS Implementation
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

The data transfer between concurrently executing tasks behave as described in Data Transfer Options on page 11-21. The coders use the following APIs on supported targets for this behavior:

Data Protection and Synchronization APIs that Embedded Coder and Generic Real-Time Targets Use

API	Linux Implementation	Windows Implementation	Mac OS Implementation
Data protection API	<ul style="list-style-type: none"> • pthread_mutex_init • pthread_mutex_ - destroy • pthread_mutex_lock • pthread_mutex_ - unlock 	<ul style="list-style-type: none"> • CreateMutex • CloseHandle • WaitForSingleObject • ReleaseMutex 	<ul style="list-style-type: none"> • pthread_mutex_init • pthread_mutex_ - destroy • pthread_mutex_lock • pthread_mutex_ - unlock
Synchronization API	<ul style="list-style-type: none"> • sem_init • sem_destroy 	<ul style="list-style-type: none"> • CreateSemaphore • CloseHandle 	<ul style="list-style-type: none"> • sem_open • sem_unlink

Data Protection and Synchronization APIs that Embedded Coder and Generic Real-Time Targets Use (Continued)

API	Linux Implementation	Windows Implementation	Mac OS Implementation
	<ul style="list-style-type: none"> • sem_wait • sem_post 	<ul style="list-style-type: none"> • WaitForSingle-Object • ReleaseSemaphore 	<ul style="list-style-type: none"> • sem_wait • sem_post

The main() code is always dynamically generated. The general structure of the generated main file is depicted in the following pseudocode:

```

main()
{
    Model_initialize();
    For_each_periodic_task
    {
        Create_synchronization_event(periodic_task);
        Create_thread(periodic_task);
    }

        For_each_aperiodic_task
    {
        Create_external_event(aperiodic_task);
        Create_thread(aperiodic_task);
    }

        Create_timer();
    Create_thread(Periodic_task_scheduler);

    For_each_aperiodic_trigger
    {
        Hookup_isr_to_aperiodic_event();
    }

    Wait_for_stopping();
    Cleanup();
    Model_terminate();
    return 0;

```

```

}

Periodic_task_scheduler()
{
    Wait_for_event_from_timer(clock);
    For_each_periodic_task
    {
        If periodic_task has a sample time hit
        Set the event to run the task
    }
}

Periodic_Task()
{
    Wait_for_event(Periodic_task_scheduler);
    Run the appropriate periodic_task();
}

Aperiodic_Task()
{
    Wait_for_aperiodic_event();
    Run the appropriate aperiodic_task();
}

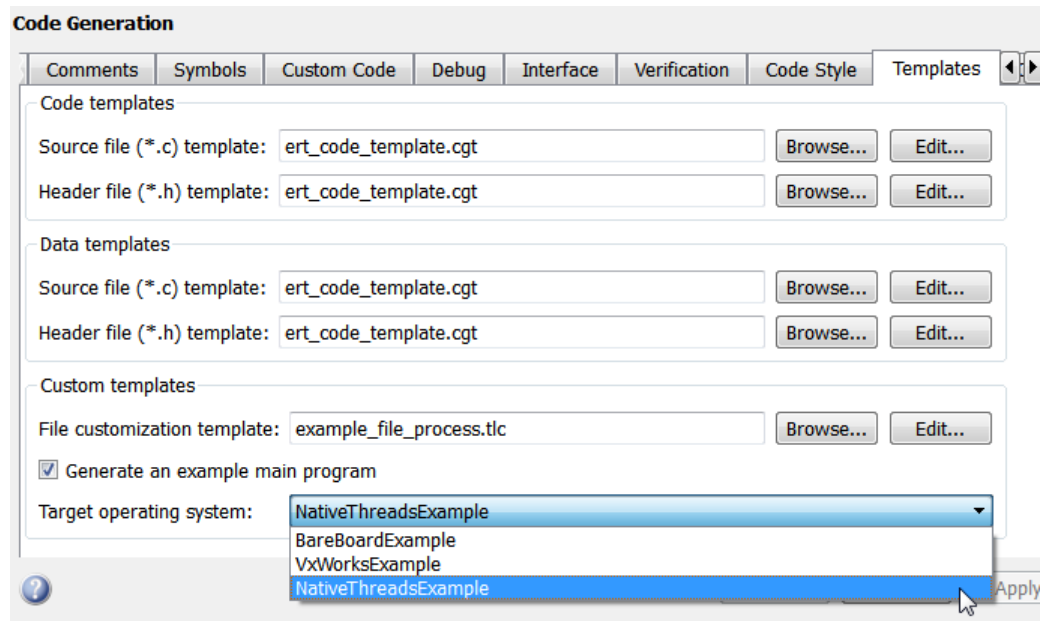
Isr()
{
    Run the appropriate aperiodic_trigger ();
}

```

Configuring Embedded Coder for Native Threads Example

To generate code for an Embedded Coder target, in the Configuration Execution dialog box:

- Select the **Code Generation > Templates > Generate an example main program** check box.
- From the **Code Generation > Templates > Target Operating System** list, select **NativeThreadsExample**.



Build and Download

You can build and download concurrent execution models for the following targets using system target files:

- Linux, Windows, and Mac OS using `ert.tlc` and `grt.tlc`.
- xPC Target using `xpctarget.tlc` and `xpctargetert.tlc`.
- Linux and Windows using `idelink_ert.tlc` and `idelink_grt.tlc`.
- Embedded Linux and VxWorks using `idelink_ert.tlc` and `idelink_grt.tlc`.

Note Deploying to embedded Linux and VxWorks targets requires the Embedded Coder product.


You cannot generate code for C++ (Encapsulated) language option.

Profile and Evaluate

Profile the execution of your code on the multicore system 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.

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 core processes 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, displays the processor number the task was 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 explicitly changing the mapping of Model blocks to efficiently use the concurrency available on your multicore system (see “Map Blocks to Tasks Incrementally” on page 11-26).

Some products, such as xPC Target and Embedded Coder, have tools you can use to profile execution for particular targets:

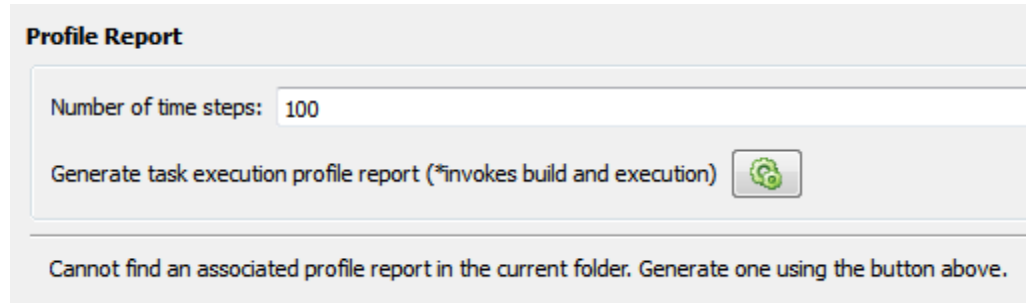
Product	For more information, see...
xPC Target	“Execution Profiling for Target Applications”
Embedded Coder	“Execution Profiling for Embedded Targets”


Generate Profile Report

This topic assumes you have a model you have already configured for concurrent execution that you want to profile. If you do not have a model configured that way, see “Configure Your Model” on page 11-12.

- 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:



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 the code, adds data collection tooling to the code, and then executes the code 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:

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.SoftwareTarget.profile` function.

For example, to create a profile report for the model `sldemo_concurrent_execution`:

```
Simulink.SoftwareTarget.profile('sldemo_concurrent_execution');
```

To create a profile report with a specific number of samples (100) for the model `sldemo_concurrent_execution`:

```
Simulink.SoftwareTarget.profile('sldemo_concurrent_execution',120);
```

The function creates a profile report named `sldemo_concurrent_execution_ProfileReport.html` in your current folder.

Concurrent Execution Example Model

For an interactive example of a model that has been configured to work in a concurrent execution environment:

- 1 In the MATLAB Command Window, click the question mark.

The Documentation Center window opens.

- 2 Navigate to **Simulink > Examples > Modeling Features > Modeling Concurrency > Modeling Concurrent Execution on Multi-Core Targets**.

Modeling Concurrent Execution on Multicore Targets

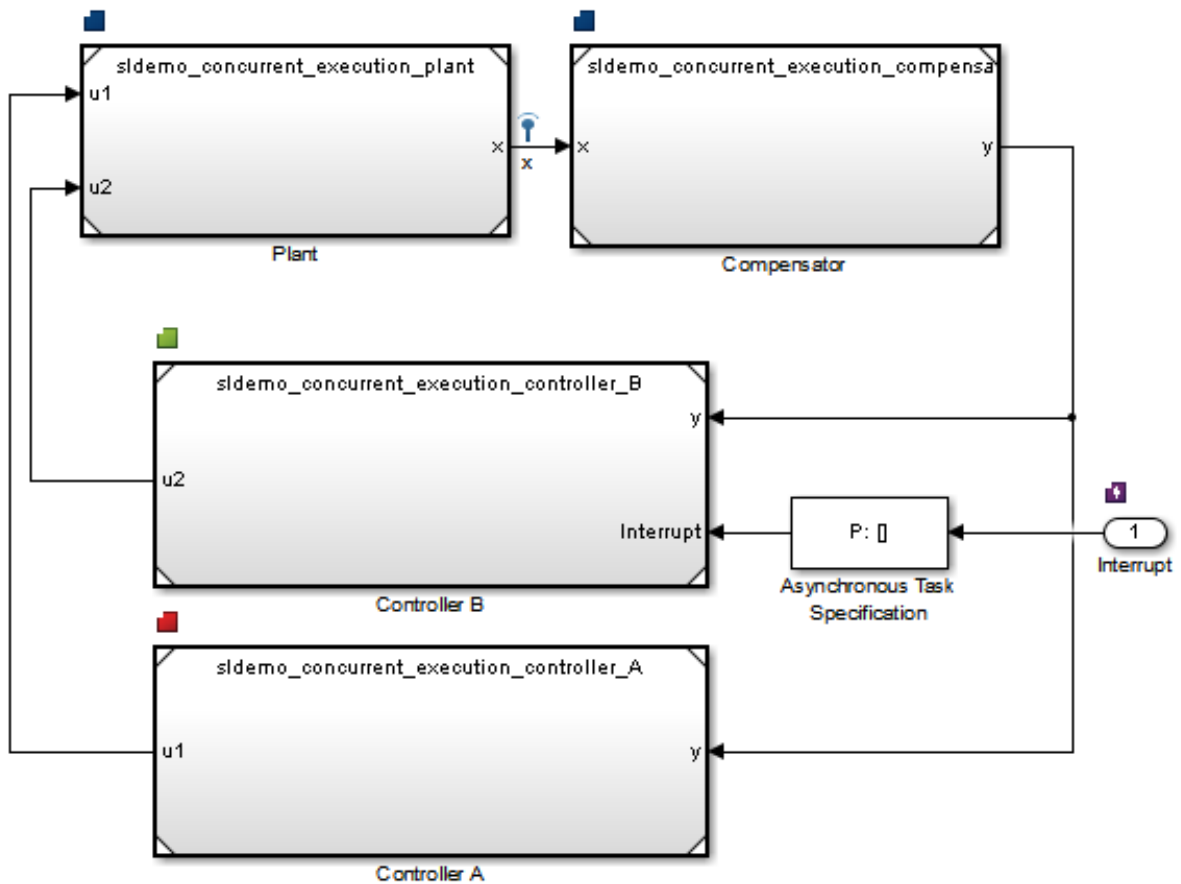
This model shows how you can use Simulink and Simulink Coder™ to realize concurrent execution of a model on a multicore target. Additionally, the example illustrates how simulation captures effects of on-target concurrent execution, such as the transfer of data between tasks. This example also shows how you can specify and simulate the effects of asynchronous external inputs to your system via the root input port.

Example Requirements

If you want to generate code for this model, install Simulink Coder™. Note, when you simulate the example, Simulink and Simulink Coder™ might generate code in the current working folder. If you do not want to (or cannot) generate files in this folder, change the current working folder.

```
topmdl = 'sldemo_concurrent_execution';  
open_system(topmdl);  
curDir = pwd;  
cd(tempdir);
```

Modeling Concurrent Execution on Multi-Core Targets



Generate Code and Profile Report
(Double-Click)

Copyright 2011-2012 The MathWorks, Inc.

Overview of the Model

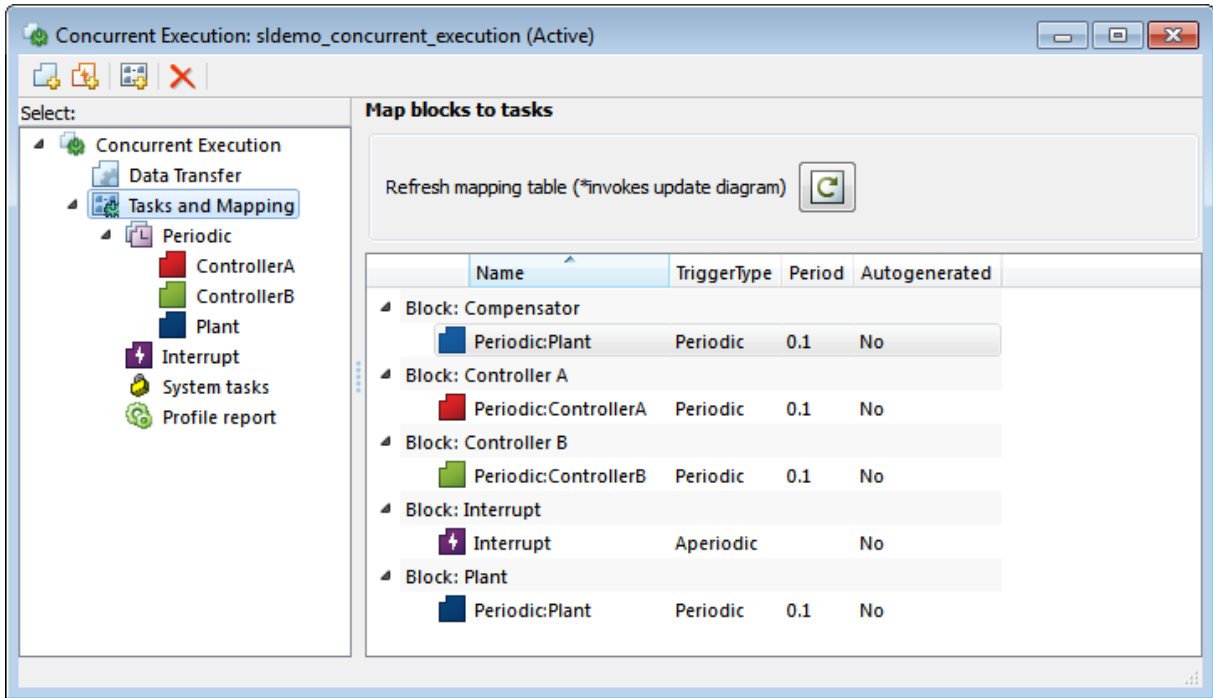
To take advantage of the concurrent execution feature, use Model blocks to partition your model into portions that can potentially execute concurrently. Open the model and examine its partitioning into four Model blocks: Plant, Compensator, ControllerA, and ControllerB.

Exploring Concurrent Execution

This feature allows you to map Model block partitions to tasks via a mapping interface. Mapping Model blocks to different tasks allows these Model blocks to execute concurrently on a real-time target.

To experiment with the mapping interface, consider changing the task configuration. Do this by adding or deleting tasks and using the 'Map blocks to tasks' panel (as shown below) to reassign model block partitions as needed. For more information, see documentation. Mapping may be performed interactively or by using programmatic APIs. For example, the following command line instructions show how to configure the model as shown in the illustration below.

Now, simulate the model and examine the effects of your mapping. In particular, the Simulink editor shows the intertask communication via the new z^{-1} icons representing that the communication introduced a unit delay.



```

% Enable the external inputs to the system.
set_param(topmdl, 'LoadExternalInput', 'on');

% Get active configuration set
csTop = getActiveConfigSet(topmdl);

% Get the task configuration.
tc = csTop.concurrentExecutionComponents;

% Get the block handles for mapping.
plantBlkH      = get_param([topmdl '/Plant'], 'Handle');
compensatorBlkH = get_param([topmdl '/Compensator'], 'Handle');
controlABlkH   = get_param([topmdl '/Controller A'], 'Handle');
controlBBlkH   = get_param([topmdl '/Controller B'], 'Handle');
interruptBlkH  = get_param([topmdl '/Interrupt'], 'Handle');

```

```
% Delete any previously created tasks
periodicTrigger = tc.Triggers(1);
periodicTasks   = periodicTrigger.Tasks;
tnames          = {periodicTasks.Name};
for i = 1:length(periodicTasks)
    periodicTrigger.deleteTask(tnames{i});
end

% Delete aperiodic trigger
aperiodicTrigger = tc.Triggers(2);
tc.deleteTrigger(aperiodicTrigger);

% Create and configure new tasks.
plant            = periodicTrigger.addTask('Plant');
controlA        = periodicTrigger.addTask('ControllerA');
controlB        = periodicTrigger.addTask('ControllerB');
interruptSource = tc.addAperiodicTrigger('Interrupt');

plant.Period     = '0.1';
controlA.Period = '0.1';
controlB.Period = '0.1';

% Perform mapping of blocks to tasks.
tc.map(plantBlkH, plant);
tc.map(compensatorBlkH, plant);
tc.map(controlABlkH, controlA);
tc.map(controlBBlkH, controlB);
tc.map(interruptBlkH, interruptSource);

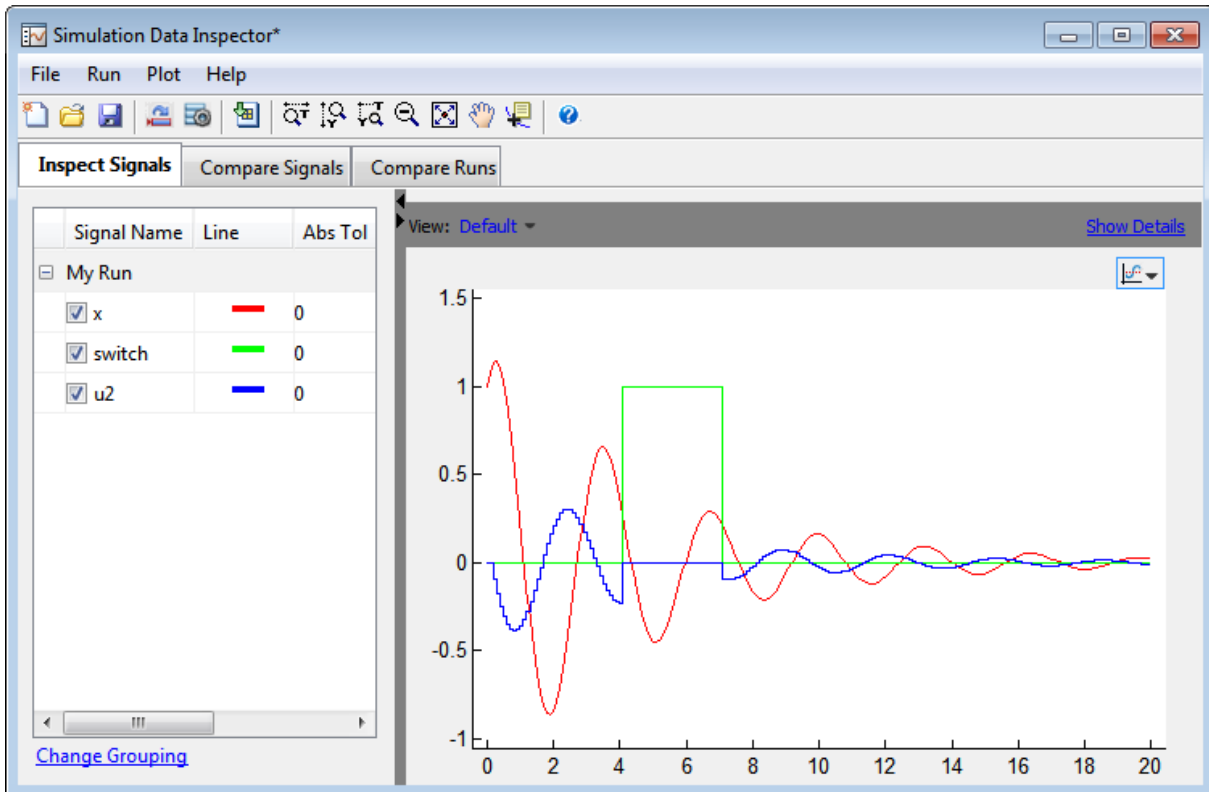
% Launch Concurrent Execution dialog
Simulink.SoftwareTarget.concurrentExecution(topmdl, ...
                                            'OpenDialog', ...
                                            'Configuration');

sim(topmdl);
```

Viewing Simulation Results

You can use the Simulation Data Inspector to view the simulation results. The results may be used to observe both communication effects (discussed

above) as well as aperiodic events. The illustration below shows how the system reacts to events specified at times 4 and 7 in the Data Import/Export panel of the Configuration Parameters dialog.



Configuring Intertask Communication

You can change the modes of the intertask communication between the Model blocks in the model to better suit design criteria. Do this by clicking the link or by following these instructions:

Click on the **Data Transfer** node of the Concurrent Execution dialog. In the **Data Transfer Options** pane, change **Defaults > Periodic signals**

to Ensure data integrity only by selecting it. Click Apply to save the changes.

Now, simulate the model. The lock icon on the signals between the Model blocks indicates that data transfer between concurrent tasks is performed in data-integrity-only mode. For fine-grained control on the data transfer modes between tasks at the level of individual signals, see the documentation.

Generating Code and Profile Report

Generate code and profile report from the model.

The profile report shows how generated code executes on a host computer. For the example profile report illustrated below, the generated code was run on a four core processor host computer. The Task Execution Time section lists the times in microseconds. The Task Affinitization to Processor Cores section shows how the tasks execute concurrently on the different cores. For details of how concurrent execution is achieved, examine the code illustrated in the Code Generation report or see the documentation.


```
close_system(topmdl, 0);
close_system([topmdl, '_plant'], 0);
close_system([topmdl, '_compensator'], 0);
close_system([topmdl, '_controller_A'], 0);
close_system([topmdl, '_controller_B'], 0);
cd(curDir);
```

Conclusion

This example illustrates a simple workflow using Simulink and Simulink Coder™ to examine the effects of concurrent execution on a multicore target. You can iteratively apply this workflow to optimize performance of a real-world application on a multicore xPC Target™ environment, along with the on-target profiling capability of xPC Target™.

Command-Line Interface for Concurrent Execution

The previous topics describe how to configure models for concurrent execution using the Simulink Configuration Execution dialog box. You can also perform the configuration using a command-line interface. The following classes, their methods and properties, and the `Simulink.SoftwareTarget.concurrentExecution` function comprise this interface:

To...	Use...
Create a configuration set for concurrent execution	<code>Simulink.SoftwareTarget.concurrentExecution</code>
Specify aperiodic trigger	<code>Simulink.SoftwareTarget.AperiodicTrigger</code>
Specify periodic trigger	<code>Simulink.SoftwareTarget.PeriodicTrigger</code>
Specify aperiodic trigger for POSIX targets	<code>Simulink.SoftwareTarget.PosixSignalHandler</code>
Specify task that models unit of concurrent execution	<code>Simulink.SoftwareTarget.Task</code>
Configure model for concurrent execution	<code>Simulink.SoftwareTarget.TaskConfiguration</code>
Specify base class for <code>PeriodicTrigger</code> and <code>AperiodicTrigger</code>	<code>Simulink.SoftwareTarget.WindowsEventHandler</code>
Describe aperiodic trigger for Windows target	<code>Simulink.SoftwareTarget.Trigger</code>
Configure concurrent execution data transfers	<code>Simulink.GlobalDataTransfer</code>

Modeling Best Practices

- “General Considerations when Building Simulink Models” on page 12-2
- “Model a Continuous System” on page 12-8
- “Best-Form Mathematical Models” on page 12-11
- “Model a Simple Equation” on page 12-15
- “Componentization Guidelines” on page 12-17
- “Modeling Complex Logic” on page 12-36
- “Modeling Physical Systems” on page 12-37
- “Modeling Signal Processing Systems” on page 12-38

General Considerations when Building Simulink Models

In this section...
“Avoiding Invalid Loops” on page 12-2
“Shadowed Files” on page 12-4
“Model Building Tips” on page 12-6

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 12-8) 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 “Create a Function-Call Subsystem” on page 7-48 for a description of function-call subsystems)
- Self-triggering subsystems and loops containing non-latched triggered subsystems (see “Create a Triggered Subsystem” on page 7-33 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:

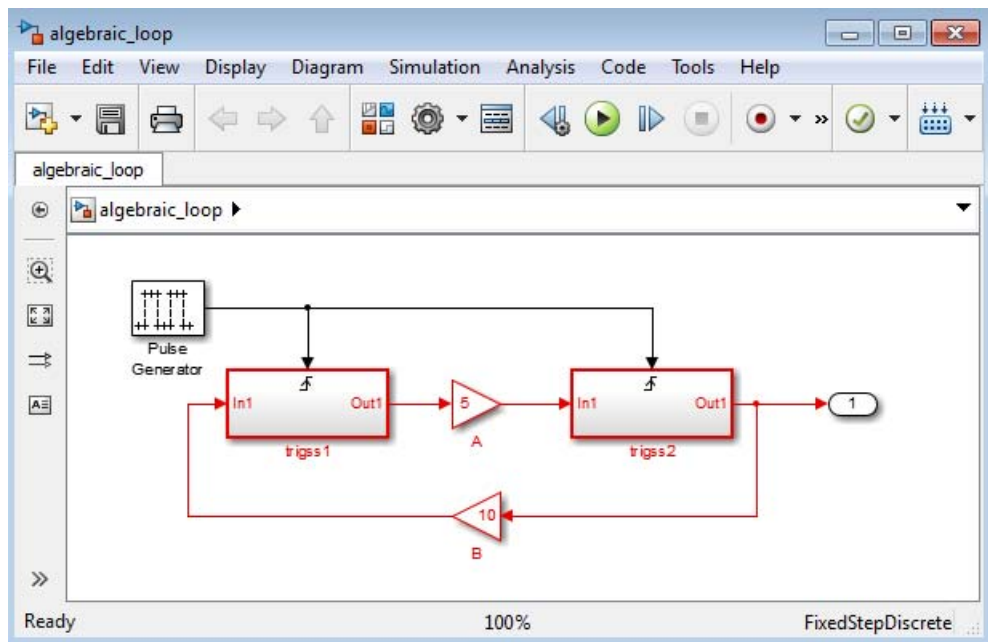
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1(sl_subsys_trigerr1)`
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2(sl_subsys_trigerr2)`

- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call systems/sl_subsys_fcncallerr3 (sl_subsys_fcncallerr3)`

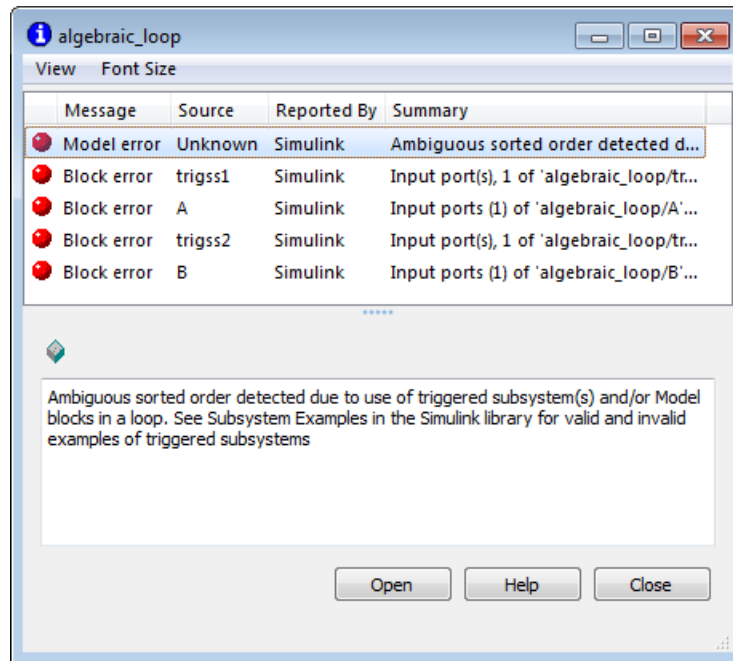
You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update Diagram** from the model's **Simulation** menu. 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 Simulation Diagnostics 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".

The rules Simulink software uses to find Model files are similar to those used by MATLAB software. See "How the Search Path Determines Which Function to Use" in the MATLAB documentation. 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, a block diagram may be loaded without showing its window. If the MATLAB path or the current MATLAB folder changes while block diagrams are in memory, these block diagrams may interfere with the use of other files of the same name. For example, after a change of folder, a loaded but invisible library may be used instead of the one the user expects.

To see an example:

- 1 Enter `sldemo_hydcy14` to open the Simulink model `sldemo_hydcy14`.
- 2 Use the `find_system` command to see which block diagrams are in memory:

```
find_system('type','block_diagram')
```

```
ans =
```

```
    'hydlib'  
    'sldemo_hydcy14'
```

Note that a Simulink library, `hydlib`, has been loaded, but is currently invisible.

- 3 Now close `sldemo_hydcy14`. Run the `find_system` command again, and you will see that the library is still loaded.

If you change to another folder which contains a different library called `hydlib`, and try to run a model in that folder, the library in that folder would not be loaded because the block diagram of the same name in memory takes precedence. This can lead to problems including:

- Simulation errors
- "Bad Link" icons on blocks which are library links
- Wrong results

To prevent these conditions, it is necessary to close the library explicitly as follows:

```
close_system('hydlib')
```

Then, when the Simulink software next needs to use a block in a library called `hydlib` it will use the file called `hydlib.slx` which is highest on the MATLAB path at the time. Alternatively, to make the library visible, enter:

```
open_system('hydlib')
```

Detecting and Fixing Problems

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 a Subsystem” on page 4-37. The Model Browser provides useful information about complex models (see “Model Browser” on page 9-80).

- 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 50-19 and “Annotate Diagrams” on page 4-23.

- Modeling strategies

If several of your models tend to use the same blocks, you might find it easier to save these blocks in a model. Then, when you build new models, just open this model and copy the commonly used blocks from it. You can create a block library by placing a collection of blocks into a system and saving the system. You can then access the system by typing its name in the MATLAB Command Window.

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.

Model a Continuous System

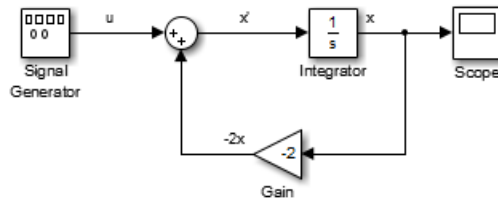
To model the differential equation

$$x' = -2x(t) + u(t),$$

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.

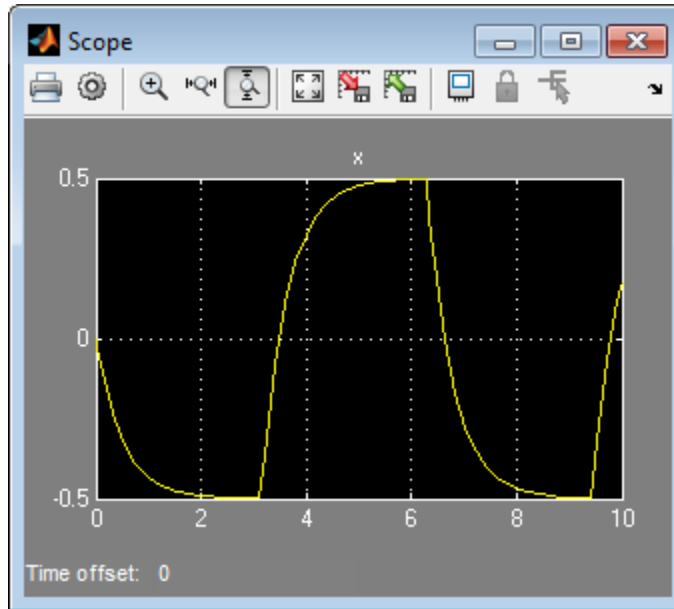
In this model, to reverse the direction of the Gain block, select the block, then use the **Diagram > Rotate & Flip > Flip Block** command. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see “Draw a Branch Line” on page 4-16.

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

$$x = u/(s + 2)$$

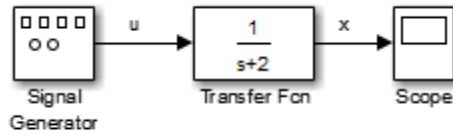
or,

$$x/u = 1/(s + 2).$$

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.

Best-Form Mathematical Models

In this section...

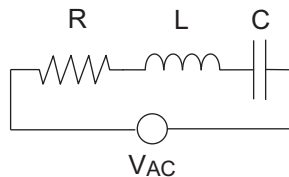
“Series RLC Example” on page 12-11

“Solving Series RLC Using Resistor Voltage” on page 12-12

“Solving Series RLC Using Inductor Voltage” on page 12-13

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

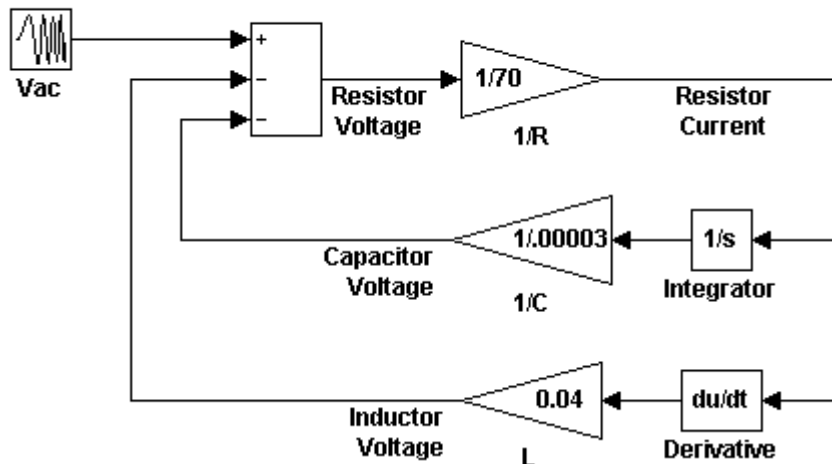
$$V_R = Ri$$

$$Ri = V_{AC} - L \frac{di}{dt} - \frac{1}{C} \int_{-\infty}^t i(t) dt$$

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 .

Series RLC Circuit: Formulated to solve for resistor current



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 Loops" on page 3-39 for more information.

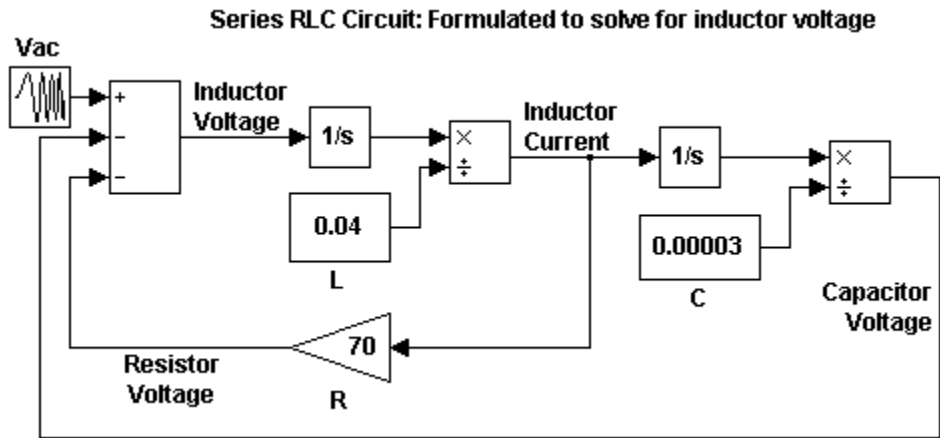
Solving Series RLC Using Inductor Voltage

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.

Model a Simple Equation

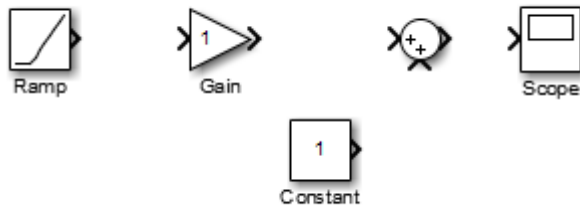
To model the equation that converts Celsius temperature to Fahrenheit

$$T_F = 9/5(T_C) + 32$$

First, consider the blocks needed to build the model:

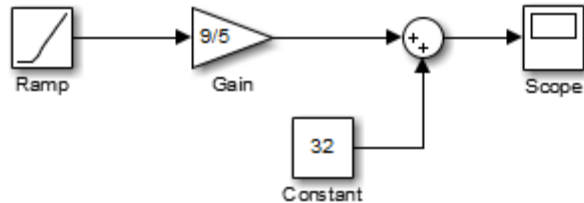
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library
- A Gain block to multiply the input signal by 9/5, from the Math Operations library
- A Sum block to add the two quantities, also from the Math Operations 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.

Componentization Guidelines

Componentization

A component is a piece of your design, a unit level item, or a subassembly, that you can work on without needing the higher level parts of the model.

Componentization involves organizing your model into components. Componentization provides many benefits for organizations that develop large Simulink models that consist of many functional pieces. The benefits include:

- Meeting development process requirements, such as:
 - Component reuse
 - Team-based development
 - Intellectual property protection
 - Unit testing
- Improving performance for:
 - Model loading
 - Simulation speed
 - Memory usage

Componentization Techniques

Key componentization techniques that you can use with Simulink include:

- Subsystems
- Libraries
- Model referencing

These componentization techniques support a wide range of modeling requirements for models that vary in size and complexity. Most large models use a combination of componentization techniques. For example, you can include subsystems in referenced models, and include referenced models in subsystems. As another example, a large model might use model reference

Accelerator blocks at the top level component partitions and blend model reference Accelerator and atomic subsystem libraries at lower levels.

Simulink provides tools to convert from subsystems to model referencing. Because of the differences between subsystems and model referencing, switching from subsystems to model referencing can involve several steps (see *Converting a Subsystem to a Referenced Model*). Consider scalability and support for anticipated future modeling requirements, such as how a model is likely to grow in size and complexity and possible code generation requirements. Designing a scalable architecture can avoid later conversion costs.

General Componentization Guidelines

This table provides high-level guidelines about the kinds of modeling goals and models for which subsystems, libraries, and model referencing are each particularly well suited.

Componentization Technique	Modeling Goals for Which the Technique Is Well Suited
Subsystems	<ul style="list-style-type: none"> • Add hierarchy to organize and visually simplify a model. • Maximize design reuse with inherited attributes for context-dependent behavior.
Libraries	<ul style="list-style-type: none"> • Provide a frequently used, and infrequently changed, modeling utility. • Reuse components repeatedly in a model or in multiple models.
Model referencing	<ul style="list-style-type: none"> • Develop a referenced model independently from the models that use it. • Obscure the contents of a referenced model, allowing you to distribute it without revealing the intellectual property that it embodies. • Reference a model multiple times without having to make redundant copies.

Componentization Technique	Modeling Goals for Which the Technique Is Well Suited
	<ul style="list-style-type: none"> • Facilitate changes by multiple people, with defined interfaces for top-level components. • Improve the overall performance by using incremental model loading, update diagram, simulation, and code generation for large models (for example, a model with 10,000 blocks). • Perform unit testing. • Simplify debugging for large models. • Generate code that reflects the model structure.

For a more detailed comparison of these modeling techniques, see “Summary of Componentization Techniques” on page 12-19.

Summary of Componentization Techniques

This section compares subsystems, libraries, and model referencing. The table includes recommendations and notes about a range of modeling requirements and features.

To see more information about a specific requirement or feature, click a link in a table cell.

Modeling Requirement or Feature	Subsystems	Libraries	Model Referencing
Development Process			
Component reuse	Not supported	Well suited	Well suited
Team-based development	Not supported	Supported, with limitations	Well suited

Modeling Requirement or Feature	Subsystems	Libraries	Model Referencing
Intellectual property protection	Not supported	Not supported	Well suited
Unit testing	Supported, with limitations	Supported, with limitations	Well suited
Performance			
Model loading speed	Supported, with limitations	Well suited	Well suited
Simulation speed for large models	Supported, with limitations	Supported, with limitations	Well suited
Memory	Supported, with limitations	Supported, with limitations	Well suited
Artificial algebraic loop avoidance	Well suited	Well suited	Supported, with limitations
Features			
Signal property inheritance	Well suited	Well suited	Supported, with limitations
State initialization	Well suited	Well suited	Supported, with limitations
Tunability	Well suited	Well suited	Supported, with limitations
Buses	Well suited	Well suited	Supported, with limitations
S-functions	Well suited	Well suited	Supported, with limitations

Modeling Requirement or Feature	Subsystems	Libraries	Model Referencing
Model configuration settings	Well suited	Well suited	Supported, with limitations
Tools	Well suited	Well suited	Supported, with limitations
Code generation	Supported, with limitations	Supported, with limitations	Well suited

For each modeling technique, you can see a summary table that includes the more detailed information included in the links in the above summary table of componentization techniques:

- “Subsystems Summary” on page 12-21
- “Libraries Summary” on page 12-25
- “Model Referencing Summary” on page 12-29

Subsystems Summary

This section provides guidelines for using subsystems for each of the modeling requirements and features highlighted in the “Summary of Componentization Techniques” on page 12-19.

For additional information about subsystems, see:

- Creating Subsystems
- “Conditional Subsystems”

Modeling Requirement or Feature	Guidelines for Subsystems
Development Process	
Component reuse	<p>Not supported</p> <ul style="list-style-type: none"> • Copy a subsystem to reuse it in a model. • Subsystem copies are independent of each other. • Save a subsystem by saving the model that contains the subsystem. • Configuration management for subsystems is difficult.
Team-based development	<p>Not supported</p> <ul style="list-style-type: none"> • For subsystems in a model, Simulink provides no direct interface with source control tools. • To create or change a subsystem, you need to open the parent model's file. This can lead to file contention when multiple people want to work on multiple subsystems in a model.
Intellectual property protection	<p>Not supported</p> <p>Use model referencing protected models instead.</p>
Unit testing	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • For coverage testing, use Signal Builder and source blocks. • Each time a subsystem changes, you need to copy the subsystem to a harness model. • The test harness may have different Simulink sort orders, due to virtual boundaries. • Test harness files require configuration management overhead.
Performance	

Modeling Requirement or Feature	Guidelines for Subsystems
Model loading speed	<p>Supported, with limitations</p> <p>Loading a model loads all subsystems at one time. There is no incremental loading.</p>
Simulation speed for large models	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • To speed simulation, use Accelerator or Rapid Accelerator simulation mode. • Simulation mode applies to the whole model. Model referencing provides a finer level of control for simulation modes.
Memory	<p>Supported, with limitations</p> <p>Memory use for simulation and code generation is comparable for subsystems and libraries. For models with over 500 blocks, model reference Accelerator mode can significantly reduce memory usage for simulation and code generation.</p>
Artificial algebraic loop avoidance	<p>Well suited</p> <ul style="list-style-type: none"> • Virtual subsystems avoid artificial algebraic loops. • For nonvirtual subsystems, consider enabling the Subsystem block parameter Minimize algebraic loop occurrences.
Features	
Signal property inheritance	<p>Well suited</p> <ul style="list-style-type: none"> • Inheriting signal properties from outside of the subsystem boundary avoids your having to specify properties for every signal. • Propagation of signal properties can lead to Simulink using signal properties that you did not anticipate.

Modeling Requirement or Feature	Guidelines for Subsystems
State initialization	<p>Well suited</p> <p>You can initialize states of subsystems.</p>
Tunability	<p>Well suited</p> <ul style="list-style-type: none"> • Tune subsystems using a block parameterization or masked subsystems. • Control tunability using Configuration Parameters > Optimization > Signals and Parameters > Inline parameters.
Buses	<p>Well suited</p> <p>Subsystems do not require the use of bus objects for virtual buses.</p>
S-functions	<p>Well suited</p> <p>Subsystems support inlined or noninlined S-functions.</p>
Model configuration settings	<p>Well suited</p> <p>A subsystem uses the model configuration settings of the model that contains the subsystem.</p>
Tools	<p>Well suited</p> <p>Subsystems provide extensive support for Simulink tools.</p>
Code generation	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • To generate code for a subsystem by itself, right-click the Subsystem block and select a code generation option. • As an optimization, Simulink attempts to recognize identical subsystems. For detected identical subsystems, the generated code includes only one copy of code for the multiple subsystems.

Modeling Requirement or Feature	Guidelines for Subsystems
	<ul style="list-style-type: none"> • For virtual subsystems, you cannot specify file or function code partitions for code generation.

Libraries Summary

This section provides guidelines for using libraries for each of the modeling requirements and features highlighted in the “Summary of Componentization Techniques” on page 12-19.

For additional information about libraries, see “Libraries”.

Modeling Requirement or Feature	Guidelines for Libraries
Development Process	
Component reuse	<p>Well suited</p> <ul style="list-style-type: none"> • Access a collection of well-defined utility blocks. • Create a component once and reuse it in models. • Link to the same library block multiple times without creating multiple copies. • Link to the same library block from multiple models. • Restrict write access to library components. • Maintain one truth: propagate changes from a single library block to all blocks that link to that library. • Disable a link to allow independent changes to a linked block. • Managing library links adds some overhead.

Modeling Requirement or Feature	Guidelines for Libraries
	<ul style="list-style-type: none"> • Save library in a file similar to a Simulink model, but you cannot simulate the file contents.
Team-based development	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • Place library files in source control for version control and configuration management. • Maintain one truth: propagate changes from a single library block to all blocks that link to that library. • To reduce file contention, use one subsystem per library. • Link to the same library block from multiple models. • Restrict write access to library component. • See General Reusability Limitations.
Intellectual property protection	<p>Not supported Use model referencing protected models instead.</p>
Unit testing	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • For coverage testing, use Signal Builder and source blocks. • Test harness may have different Simulink sort orders, due to virtual boundaries. • Test harness files require configuration management overhead.
Performance	

Modeling Requirement or Feature	Guidelines for Libraries
Model loading speed	<p>Well suited</p> <p>Simulink incrementally loads a library at the point needed during editing, updating a diagram, or simulating a model.</p>
Simulation speed for large models	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • To speed simulation, use Accelerator or Rapid Accelerator simulation mode. • Simulation mode applies to the whole model. Model referencing provides a finer level of control for simulation modes.
Memory	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • Simulink incrementally loads libraries at the point needed during editing, updating a diagram, or simulating a model. • Simulink duplicates library block instances during block update. • Memory usage for simulation and code generation is comparable for subsystems and libraries. For models with over 500 blocks, model reference Accelerator mode can significantly reduce memory usage for simulation and code generation.
Artificial algebraic loop avoidance	<p>Well suited</p> <ul style="list-style-type: none"> • Virtual subsystems avoid artificial algebraic loops. • For nonvirtual subsystems, consider enabling the Subsystem block parameter Minimize algebraic loop occurrences.
Features	

Modeling Requirement or Feature	Guidelines for Libraries
Signal property inheritance	<p>Well suited</p> <ul style="list-style-type: none"> • Inheriting signal properties from outside of the library block boundary avoids your having to specify properties for every signal. • Propagation of signal properties can lead to Simulink using signal properties that you did not anticipate.
State initialization	<p>Well suited</p> <p>You can initialize states of library blocks.</p>
Tunability	<p>Well suited</p> <ul style="list-style-type: none"> • Tune library blocks using block parameterization or masked subsystems. • Control tunability using Configuration Parameters > Optimization > Signals and Parameters > Inline parameters.
Buses	<p>Well suited</p> <p>Libraries do not require the use of bus objects for virtual buses.</p>
S-functions	<p>Well suited</p> <p>Libraries support inlined and noninlined S-functions.</p>
Model configuration settings	<p>Well suited</p> <ul style="list-style-type: none"> • Library models do not have model configuration settings. • A referenced library block uses the model configuration setting of the model that contains that block.

Modeling Requirement or Feature	Guidelines for Libraries
Tools	<p>Supported, with limitations</p> <p>There are some limitations for using some Simulink tools, such as the Model Advisor, with libraries.</p>
Code generation	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • As an optimization, Simulink attempts to recognize identical subsystems. For detected identical subsystems, the generated code includes only one copy of code for the multiple subsystems. • For virtual subsystems, you cannot specify file or function code partitions for code generation.

Model Referencing Summary

This section provides guidelines for using model referencing for each of the modeling requirements and features highlighted in the “Summary of Componentization Techniques” on page 12-19.

For additional information about model referencing, see:

- “Model Reference”
- Simulink Model Referencing Requirements
- Model Referencing Limitations

Modeling Requirement or Feature	Guidelines for Model Referencing
Development Process Requirements	
Component reuse	<p>Well suited</p> <ul style="list-style-type: none"> • Create a standalone component once and reuse it in multiple models. • Reference the same model multiple times without creating multiple copies. • Reference the same model from multiple models. • Model referencing uses specified boundaries for preserving component integrity.
Team-based development	<p>Well suited</p> <ul style="list-style-type: none"> • For version control and configuration management, you can place model reference files in a source control system. • Design, create, simulate, and test a referenced model independently from the model that references it. • Link to the same model reference from multiple models. • Changes made to a referenced model apply to all instances of that referenced model. • Simulink does not limit access for changing a model reference. • You save a referenced model in a separate file from the model that references it. Using separate files helps to avoid file contention.

Modeling Requirement or Feature	Guidelines for Model Referencing
Intellectual property protection	<p>Well suited</p> <ul style="list-style-type: none"> • Use the protected model feature to obscure contents of a referenced model in a distributed model. • Creating a protected model feature requires a Simulink Coder license. Using a protected model does <i>not</i> require a Simulink Coder license.
Unit testing	<p>Well suited</p> <ul style="list-style-type: none"> • Test components independently to isolate behaviors, by simulating them standalone. • You can eliminate unit retest for unchanged components. • Use a data-defined test harness, with MATLAB test vectors and direct coverage collection. • For coverage testing, use root inports and outports.
Performance	
Model loading speed	<p>Well suited</p> <ul style="list-style-type: none"> • Simulink incrementally loads a referenced model at the point needed during editing, updating a diagram, or simulating a model. • If a simulation target build is required, first-time loading can be slow.

Modeling Requirement or Feature	Guidelines for Model Referencing
Simulation speed for large models	<p>Well suited</p> <ul style="list-style-type: none"> • Simulate a referenced model standalone. • The Model block has an option for specifying the simulation mode. • You can improve rebuild performance by selecting the appropriate setting for the Configuration Parameters > Model Referencing > Rebuild parameter. • Simulation through code generation can have a slow start-up time, which might be undesirable during prototyping. • See “Limitations on Normal Mode Referenced Models” on page 6-86, “Limitations on Accelerator Mode Referenced Models” on page 6-87, and “Limitations on PIL Mode Referenced Models” on page 6-90.
Memory	<p>Well suited</p> <ul style="list-style-type: none"> • Simulink loads a referenced model at the point that model is needed for navigation during editing, updating a diagram, or simulating a model. • Use model reference Accelerator mode to reduce memory usage, incrementally loading a compiled version of a referenced model.
Artificial algebraic loop avoidance	<p>Supported, with limitations</p> <p>Consider enabling Configuration Parameters > Model Referencing > Minimize algebraic loop occurrences.</p>
Features	

Modeling Requirement or Feature	Guidelines for Model Referencing
Signal property inheritance	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • Inherit sample time when the referenced model is sample-time independent. You cannot propagate a continuous sample time to a Model block that is sample-time independent. • Model block is context-independent, so it cannot inherit signal properties. Explicitly set input and output signal properties. • Use a bus object to define the signal data type of a bus signal that is passed into a referenced model. • Goto and From block lines cannot cross model referencing boundaries. • • See Index Base Limitations.
State initialization	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • You can initialize states from the top model. • Use either the structure format or structure with time format to initialize the states of a top model and the models that it references. • To use the SimState (simulation state) feature with model referencing, simulate all Model blocks in Normal mode. • See “Import and Export State Information for Referenced Models” on page 48-149.

Modeling Requirement or Feature	Guidelines for Model Referencing
Tunability	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • To have each instance of a referenced model use different values, use model arguments in the Model block. • To have each instance of a referenced model use the same values, use <code>Simulink.Parameter</code> objects. • To have each instance of a referenced model use the same values, use <code>Simulink.Parameter</code> objects. By default, all other parameters are inlined.
Buses	<p>Supported, with limitations</p> <p>You must use bus objects for bus signals that cross referenced model boundaries (for example, global data stores, root inports, root outports).</p>
S-functions	<p>Supported, with limitations</p> <p>Model referencing generally supports inlined or noninlined S-functions. See “S-Function Limitations” on page 6-89.</p>
Model configuration settings	<p>Supported, with limitations</p> <ul style="list-style-type: none"> • To apply the same model configuration settings to all models in a model hierarchy, use a referenced configuration set. • Configuration settings for the root model and referenced models must be consistent. However, not all configuration settings need to be the same across the model hierarchy.

Modeling Requirement or Feature	Guidelines for Model Referencing
Tools	<p data-bbox="664 366 1055 395">Supported, with limitations</p> <ul data-bbox="664 430 1292 604" style="list-style-type: none"><li data-bbox="664 430 1292 522">• There are some limitations for using some Simulink tools, such as the Simulink Debugger, with model referencing.<li data-bbox="664 539 1292 604">• For details, see “Simulink Tool Limitations” on page 6-85.
Code generation	<p data-bbox="664 621 825 651">Well suited</p> <ul data-bbox="664 685 1322 1032" style="list-style-type: none"><li data-bbox="664 685 1322 749">• By default, model referencing generates code incrementally.<li data-bbox="664 767 1322 888">• You can improve rebuild performance by selecting the appropriate setting for the Configuration Parameters > Model Referencing > Rebuild parameter.<li data-bbox="664 906 1322 1032">• Model referencing requires the use of bus objects. For information about the impact of bus objects for code generation, in the Embedded Coder documentation, see “Buses”.

Modeling Complex Logic

To model complex logic in a Simulink model, consider using Stateflow software.

Stateflow extends Simulink with a design environment for developing state charts and flow charts. Stateflow provides the language elements required to describe complex logic in a natural, readable, and understandable form. It is tightly integrated with MATLAB and Simulink products, providing an efficient environment for designing embedded systems that contain control, supervisory, and mode logic.

For more information on Stateflow software, see “Stateflow”.

Modeling Physical Systems

To model physical systems in the Simulink environment, consider using Simscape software.

Simscape extends Simulink with tools for modeling systems spanning mechanical, electrical, hydraulic, and other physical domains as physical networks. It provides fundamental building blocks from these domains to let you create models of custom components. The MATLAB based Simscape language enables text-based authoring of physical modeling components, domains, and libraries.

For more information on Simscape software, see “Simscape”.

Modeling Signal Processing Systems

To model signal processing systems in the Simulink environment, consider using DSP System Toolbox™ software.

DSP System Toolbox provides algorithms and tools for the design and simulation of signal processing systems. These capabilities are provided as MATLAB functions, MATLAB System objects, and Simulink blocks. The system toolbox includes design methods for specialized FIR and IIR filters, FFTs, multirate processing, and DSP techniques for processing streaming data and creating real-time prototypes. You can design adaptive and multirate filters, implement filters using computationally efficient architectures, and simulate floating-point digital filters. Tools for signal I/O from files and devices, signal generation, spectral analysis, and interactive visualization enable you to analyze system behavior and performance. For rapid prototyping and embedded system design, the system toolbox supports fixed-point arithmetic and C or HDL code generation.

For more information on DSP System Toolbox software, see “DSP System Toolbox”.

Managing Projects

- “Organize Large Modeling Projects” on page 13-4
- “What Are Simulink Projects?” on page 13-5
- “Try Simulink Project Tools with the Airframe Project” on page 13-7
- “Create a New Project to Manage Existing Files” on page 13-21
- “Add Files to the Project” on page 13-25
- “Create a New Project from an Archived Project” on page 13-27
- “Create a New Project Using Templates” on page 13-28
- “Open Recent Projects” on page 13-32
- “Change the Project Name, Root, Description, and Startup Folder” on page 13-33
- “What Can You Do With Project Shortcuts?” on page 13-35
- “Automate Startup Tasks with Shortcuts” on page 13-36
- “Set Project Path at Startup and Reset at Shutdown” on page 13-39
- “Automate Shutdown Tasks with Shortcuts” on page 13-41
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47
- “Using Templates to Create Standard Project Settings” on page 13-49
- “Create a Template from the Current Project” on page 13-50
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53

- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56
- “Group and Sort File Views” on page 13-57
- “Search and Filter File Views” on page 13-59
- “Work with Project Files” on page 13-60
- “Move Project Files” on page 13-64
- “Back Out Changes” on page 13-65
- “Add Labels to Files” on page 13-66
- “Create Labels” on page 13-67
- “View and Edit Label Data” on page 13-68
- “Create a Batch Function” on page 13-70
- “Create Shortcuts to Batch Job Functions” on page 13-71
- “Run a Simulink Project Batch Job” on page 13-72
- “Upgrade Model Files to SLX and Preserve Revision History” on page 13-74
- “Archive Projects in Zip Files” on page 13-80
- “Automate Project Management Tasks” on page 13-81
- “About Source Control with Projects” on page 13-90
- “Register Model Files with Source Control Tools” on page 13-91
- “Add a Project to Source Control” on page 13-92
- “Add Files to a New Repository” on page 13-98
- “Disable Source Control” on page 13-99
- “Change Source Control” on page 13-100
- “Set Up SVN Source Control” on page 13-101
- “Write a Source Control Adapter with the SDK” on page 13-109
- “Retrieve a Working Copy of a Project from Source Control” on page 13-110
- “Tag and Retrieve Versions of Project Files” on page 13-115
- “Refresh Status of Project Files” on page 13-117

-
- “Check for Modifications” on page 13-121
 - “Update Revisions of Project Files” on page 13-122
 - “Check Out Files” on page 13-125
 - “View Modified Files” on page 13-128
 - “Review Changes” on page 13-130
 - “Precommit Actions” on page 13-132
 - “Commit Modified Files to Source Control” on page 13-134
 - “Revert Local Changes and Release Locks” on page 13-136
 - “Resolve Conflicts” on page 13-137
 - “Work with Derived Files in Projects” on page 13-139
 - “What Is Dependency Analysis?” on page 13-140
 - “Choose Files and Run Dependency Analysis” on page 13-141
 - “Check Dependencies Results and Resolve Problems” on page 13-144
 - “Perform Impact Analysis” on page 13-148
 - “Save and Reload Dependency Analysis Results” on page 13-157
 - “Analyze Model Dependencies” on page 13-159

Organize Large Modeling Projects

You can use Simulink 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 “Try Simulink Project Tools with the Airframe Project” on page 13-7.
- 2** Create a new project. See “Create a New Project to Manage Existing Files” on page 13-21.
- 3** Use the Dependency Analysis view to analyze your project and check required files. See “Choose Files and Run Dependency Analysis” on page 13-141.
- 4** Explore views of your files. See “Work with Project Files” on page 13-60.
- 5** Create shortcuts to save and run frequent tasks. See “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47.
- 6** Run operations on batches of files. See “Run a Simulink Project Batch Job” on page 13-72.
- 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 13-90.

For guidelines on structuring projects, see “Componentization Guidelines” on page 12-17.

What Are Simulink Projects?

You can use Simulink 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 Simulink Projects 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), an external source control tool.

See the Web page

<http://www.mathworks.com/products/simulink/simulink-projects/> 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 “Try Simulink Project Tools with the Airframe Project” on page 13-7.

- 2** Create a new project. See “Create a New Project to Manage Existing Files” on page 13-21.
- 3** Use the Dependency Analysis view to analyze your project and check required files. See “Choose Files and Run Dependency Analysis” on page 13-141.
- 4** Explore views of your files. See “Work with Project Files” on page 13-60.
- 5** Create shortcuts to save and run frequent tasks. See “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47.
- 6** Run operations on batches of files. See “Run a Simulink Project Batch Job” on page 13-72.
- 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 13-90.

For guidelines on structuring projects, see “Componentization Guidelines” on page 12-17.

Try Simulink Project Tools with the Airframe Project

In this section...

- “Explore the Airframe Project” on page 13-7
- “Set Up Project Files and Open Simulink Project” on page 13-8
- “View, Search, and Sort Project Files” on page 13-8
- “Understand Project Startup and Shutdown Tasks” on page 13-10
- “Create a Startup Shortcut” on page 13-11
- “Open and Run Frequently Used Files” on page 13-12
- “Review Changes in Modified Files” on page 13-12
- “Run Project Integrity Checks” on page 13-14
- “Run Dependency Analysis” on page 13-14
- “Commit Modified Files” on page 13-18
- “View Source Control and Project Information” on page 13-19

Explore the Airframe Project

Try an example Simulink 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, data)
- A set of actions to use with your project (run setup code, open models, simulate, build, run shutdown code)
- Working with files under source control (check out, compare revisions, tag or label, check in)

The Airframe example shows how to:

- 1 Set up and browse some example project files under local revision control.
- 2 Examine project shortcuts to run setup and shutdown tasks and 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 Simulink Project

Run this command to create a working copy of the project files and open the project:

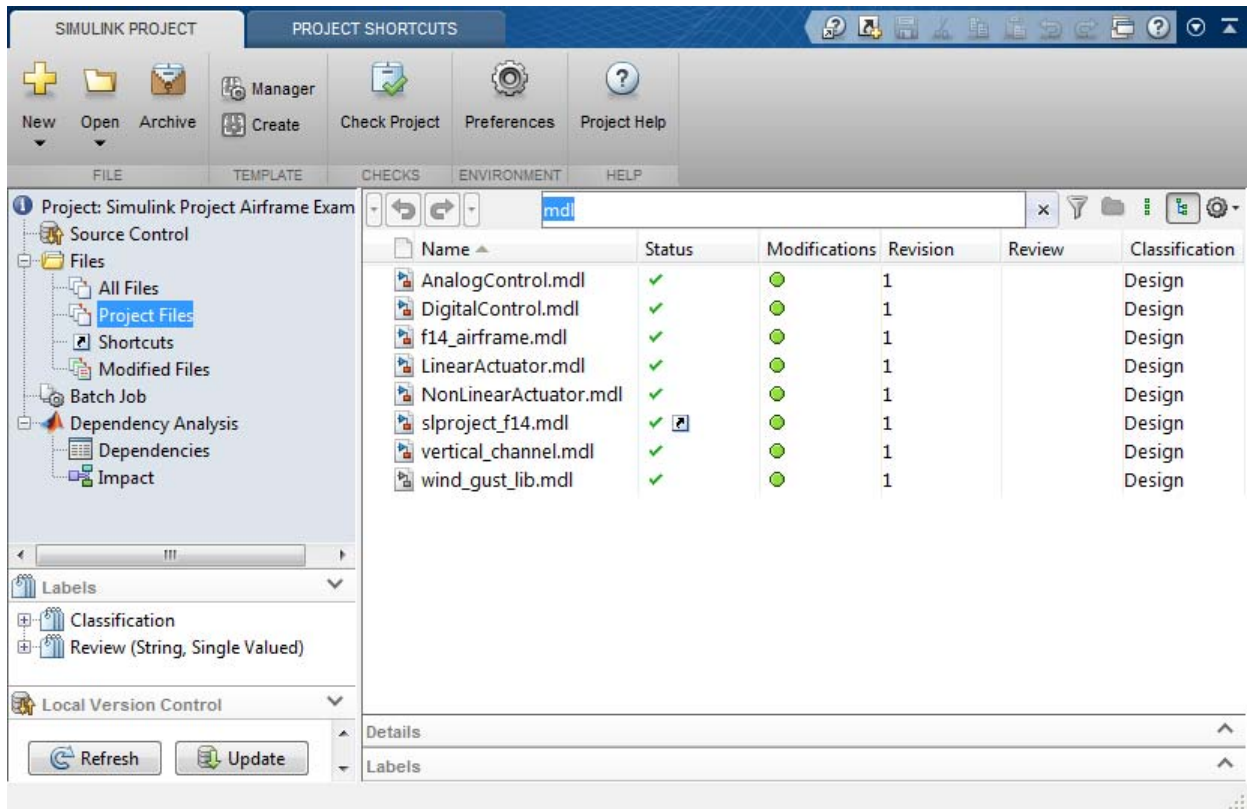
```
sldemo_slproject_airframe
```

The project example copies files to your temporary folder so that you can edit them and put them under local revision control.

The Simulink Project 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.

View, Search, and Sort Project Files

- 1 Click the Project Files view to manage the files within your project. Only the files in your project are shown.
- 2 Click the All Files view to see all the files in your sandbox. 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.



Click the **x** to clear the search.

- 4** To view files as a list instead of a tree, click the List view button.



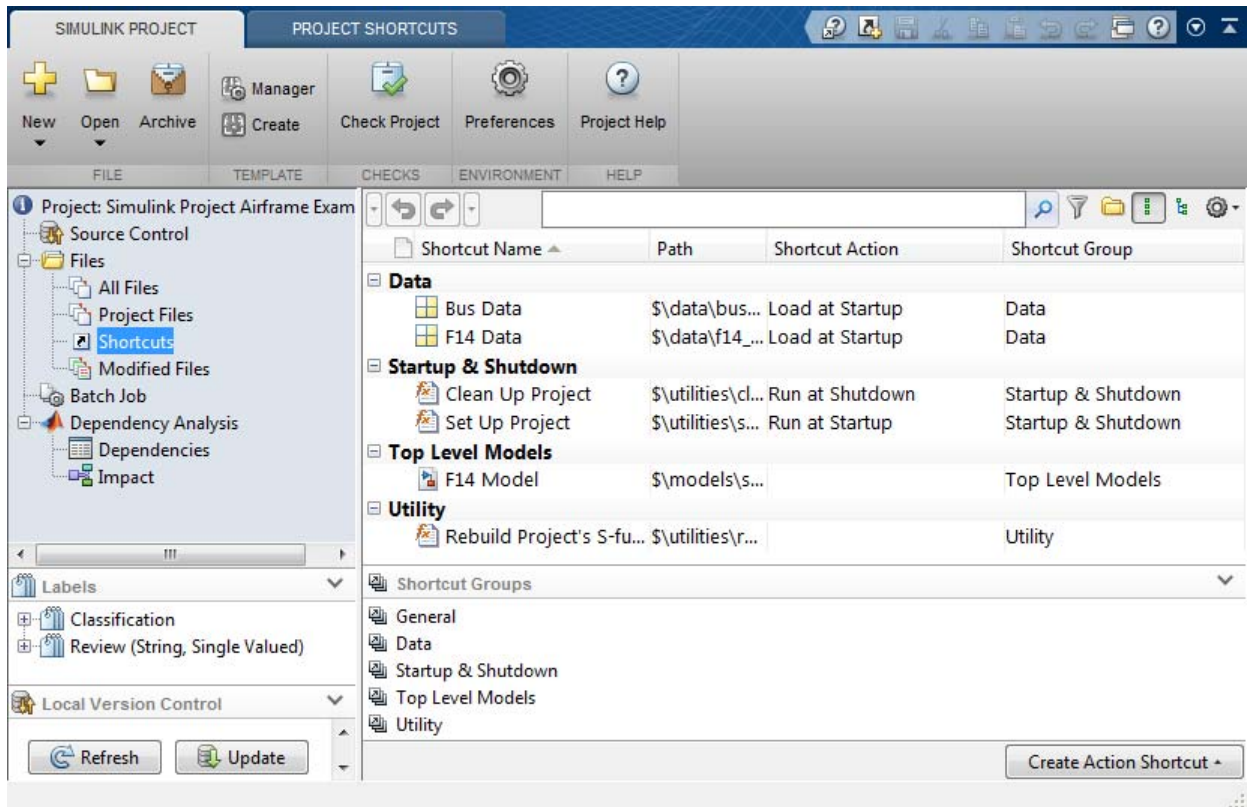
- 5** To sort files and to customize the columns, click the Actions button at the far right of the search box.

- 6 You can dock and undock the Simulink Project into the MATLAB Desktop. If you want to maximize space for viewing your project files, undock the Simulink Project. Drag the title bar to undock it.

Understand Project Startup and Shutdown Tasks

You can use shortcuts to automatically run startup or shutdown tasks, and to easily find files within a large project.

- 1 Click the Shortcuts node to view the startup tasks and other shortcuts.
- 2 Click the List view button to view the shortcuts as a list of files.



In this example, you can see that some files are set as **Run at Startup** shortcuts. Startup shortcut files automatically run, load, or open when you open the project. You can use these shortcuts to set up the environment for your project. The file `set_up_project.m` sets the MATLAB path, and defines where to create the `slprj` folder.

- Open the **Set Up Project** shortcut to understand how it works. Double-click the shortcut or right-click to select **Open**. The file `set_up_project.m` opens in the Editor. The following lines use the project API to get the current project and the root folder:

```
p = simulinkproject;  
projectRoot = p.RootFolder;
```

- The shortcut **Clean Up Project** is set as a **Run at Shutdown** shortcut. This type of shortcut runs before the current project closes. In this example, `clean_up_project.m` resets the environment changes made by `set_up_project.m`.

For details on configuring shortcuts to set your project path, see “Set Project Path at Startup and Reset at Shutdown” on page 13-39.

- Click the **Project Shortcuts** tab in the toolstrip. You can view and run shortcuts from any project view using the toolstrip. The shortcuts are organized into groups that you can specify in the **Shortcuts** view.

Create a Startup Shortcut

Create a new startup shortcut.

- 1 In the **Project Files** view, right-click a file, and select **Create Shortcut**.
- 2 In the **Shortcuts** view, right-click the shortcut you created and select **Set Shortcut Action > Startup** to make the file run, load, or open at startup.

When you open the project, the project performs the default action for startup shortcut files depending on their type:

- Run `.m` files.
- Load `.mat` files.
- Open Simulink models.

For details on shortcuts, see “Automate Startup Tasks with Shortcuts” on page 13-36.

Open and Run Frequently Used Files

You can use shortcuts to make scripts easier to find in a large project. 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 The shortcut file builds a MEX-file. If you have not previously used the `mex` command, run the following command and follow the instructions to choose a compiler: `mex -setup`
- 2 Select the Project Shortcuts tab in the toolstrip, and click the shortcut `Rebuild Project's S-functions`.

Open the `rebuild_s_functions.m` file to explore how it works.

Open the top model.

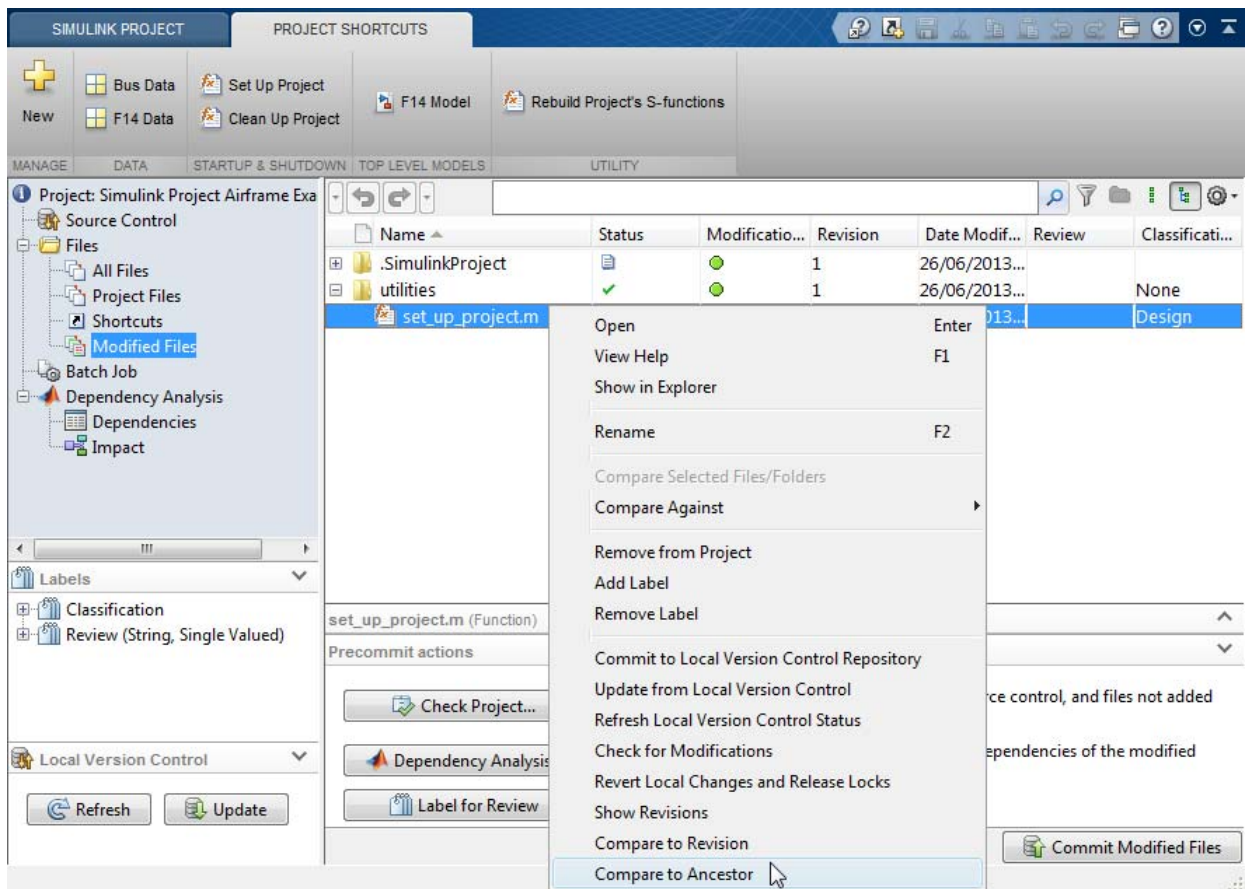
- On the Project Shortcuts tab, click the shortcut `F14 Model` to open the root model for this project. This model runs only after you compile the required S-Function.
- To create new shortcuts to access frequently used files, select the Project Files view, right-click a file, and select **Create Shortcut > General** or any other `shortcutGroupName`. This creates a basic shortcut with no startup or shutdown action.

Review Changes in Modified Files

Open and make changes to files and review changes.

- 1 In the Project Files view, expand the `utilities` folder.
- 2 Either double-click to open the `set_up_project` file for editing from the Simulink 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 Simulink Project, click the Modified Files node. The files you changed appear in the list.
- 5 To review changes, right-click the `set_up_project` file in the Modified Files view and select **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 to Ancestor**, and you have Simulink Report Generator installed, this command runs a Simulink XML comparison.

If you have Simulink Report Generator, try the following example.

- 1 Select the Project Files view and expand the `models` folder.
- 2 Either double-click to open the `AnalogControl` file for editing from the Simulink 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, right-click the file in the Modified Files view and select **Compare to Ancestor**.

The Comparison Tool opens a report.

Run Project Integrity Checks

In the Modified Files view, click **Check Project** to run the project integrity checks to look for missing files, files to add to source control or retrieve from source control, and other issues. The checks dialog can offer automatic fixes to problems found.

When you click a **Fix** button in the Checks dialog box, you can view recommended actions and decide whether to make the changes.

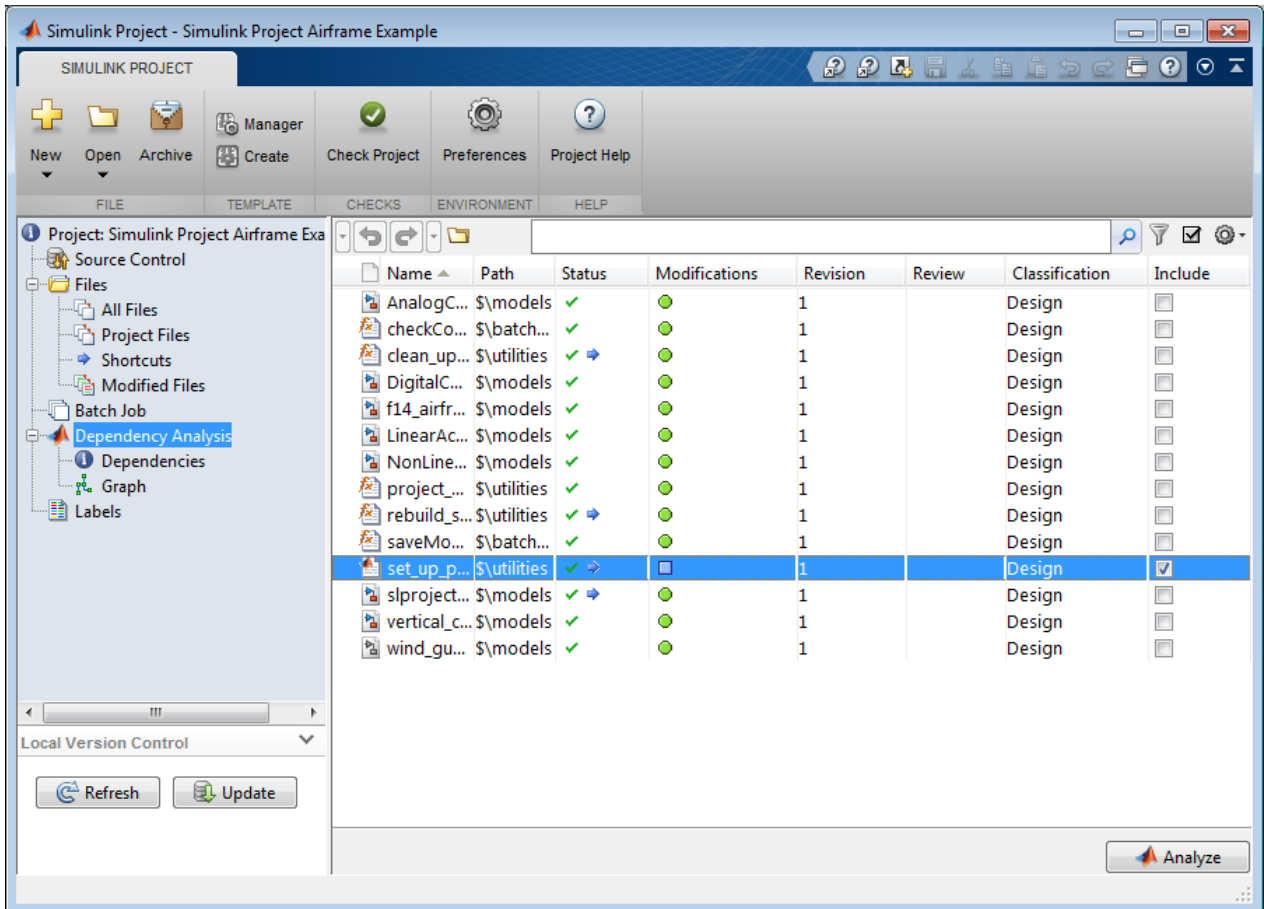
For an example using the project checks to fix issues, see “Upgrade Model Files to SLX and Preserve Revision History” on page 13-74.

Run Dependency Analysis

Run a file dependency analysis on the modified files in your project to check that all the required files were added to the project.

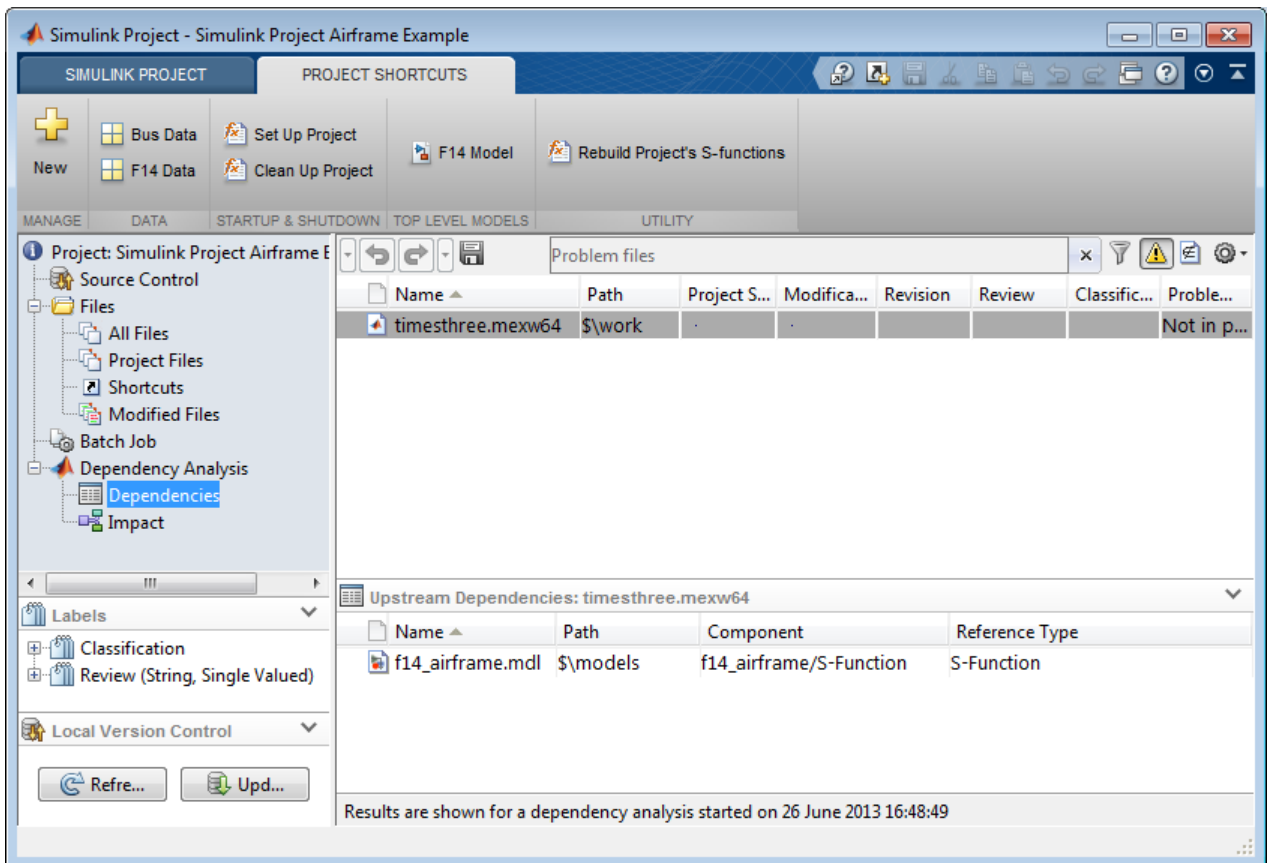
- 1 In the Modified Files view, click **Dependency Analysis** to analyze the modified files.

The Simulink Project opens the Dependency Analysis view with the files you modified selected for analysis.



- 2 Include all files in the analysis, to analyze the whole project, not just the modified files. To quickly include all files, click the list of files, and press **Ctrl+A** to select all the files. Then right-click and select **Include**.
- 3 Click **Analyze**.

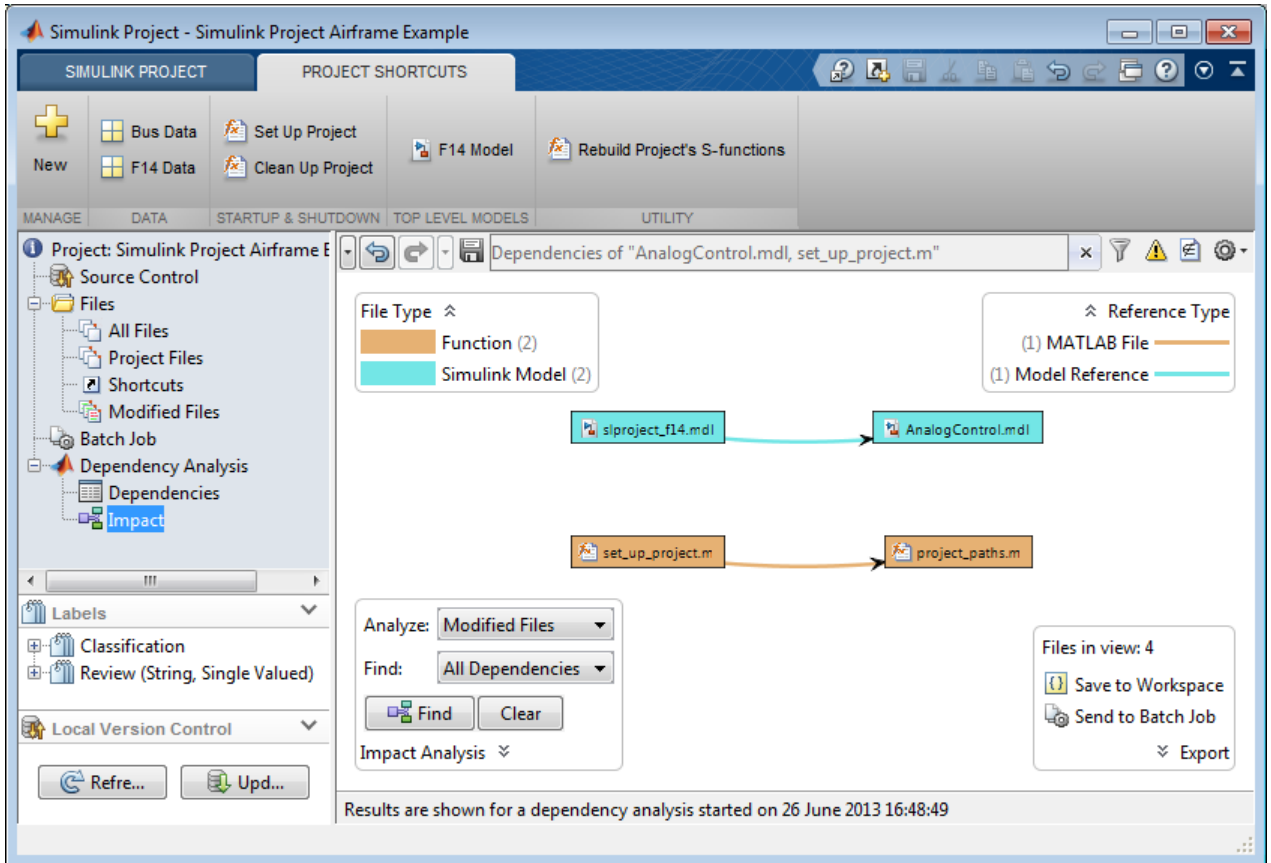
- 4 Review reported problem files. Observe that Problem files automatically appears in the search box for filtering file views.
- 5 Observe that the S-Function binary file, `timesthree.mexw64`, is required by the project but is not currently part of it. The **Problem Description** column reports the file is Not in project.
- 6 Click the problem file to view where it is used. The Used By column in the list at the bottom of the dialog box displays the file location that uses the selected problem file.



You might want to 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.

- 7** Right-click the problem file and select **Add External File**. You remove the file from the problem files list, and the next time you run dependency analysis, this file does not appear as a problem file. This does not add the external file to the project. In this example, you do not want to add the binary file to the project, but instead use the script to regenerate it from the source code in the project. Use **Add External File** to stop such files being marked as problems.
- 8** Click the **Impact** node to view the dependency graph of the project structure.
- 9** Select **Modified Files** in the **Analyze** list.
- 10** Click **Find**.

The graph shows only the modified files and their file dependencies.



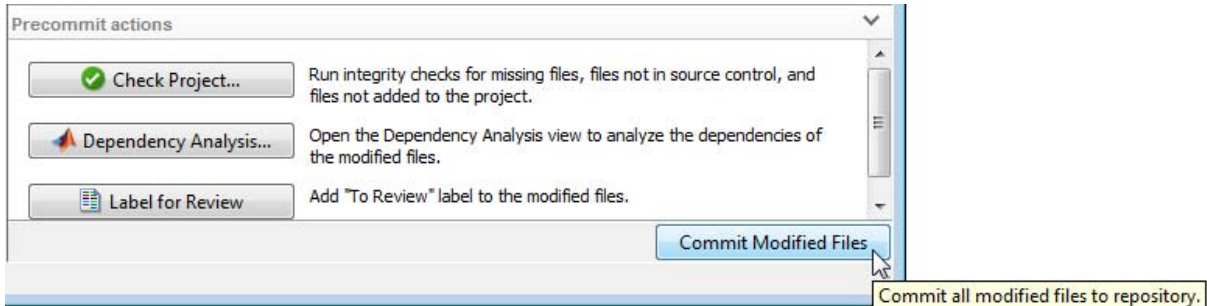
Commit Modified Files

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 Modified Files view, click **Check Project** to make sure your changes are ready to commit.
- 2 Observe the Modified Files list includes a `.SimulinkProject` folder. The files stored in the `.SimulinkProject` folder are internal project definition files generated by your changes. These definition files allow you to add a label to a file without checking it out. You do not need to view the definition

files directly unless you need to merge them, but they are listed so you know about all the files being committed to the source control system. See “Project Definition Files” on page 13-129.

- 3 To commit your changes to source control, click **Commit Modified Files**.



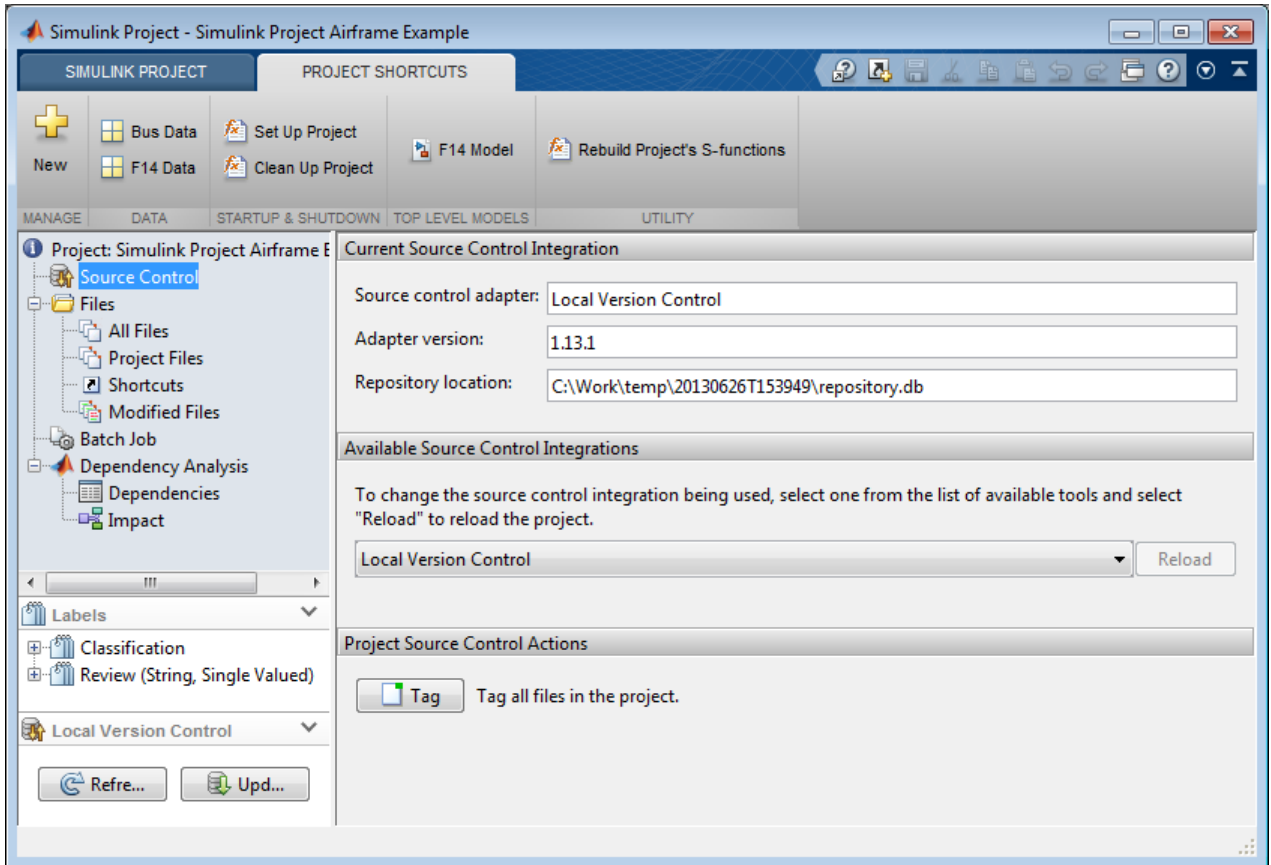
- 4 Enter a comment for your submission, and click **Submit**.

Watch the messages as the example source control commits your changes.

View Source Control and Project Information

- 1 Click the **Source Control** node to see information about the source control tool used by the current project.

The Airframe example project is under the control of the Local Version Control tool.



For source control information on individual files (e.g., modified, checked out), see the Modifications column in the Files views.

- 2 Click the root tree node **Project: Simulink Project Airframe Example** to see information about the currently open project, including a description and the location of the project root folder.

For next steps, see “Simulink Projects”.

Create a New Project to Manage Existing Files

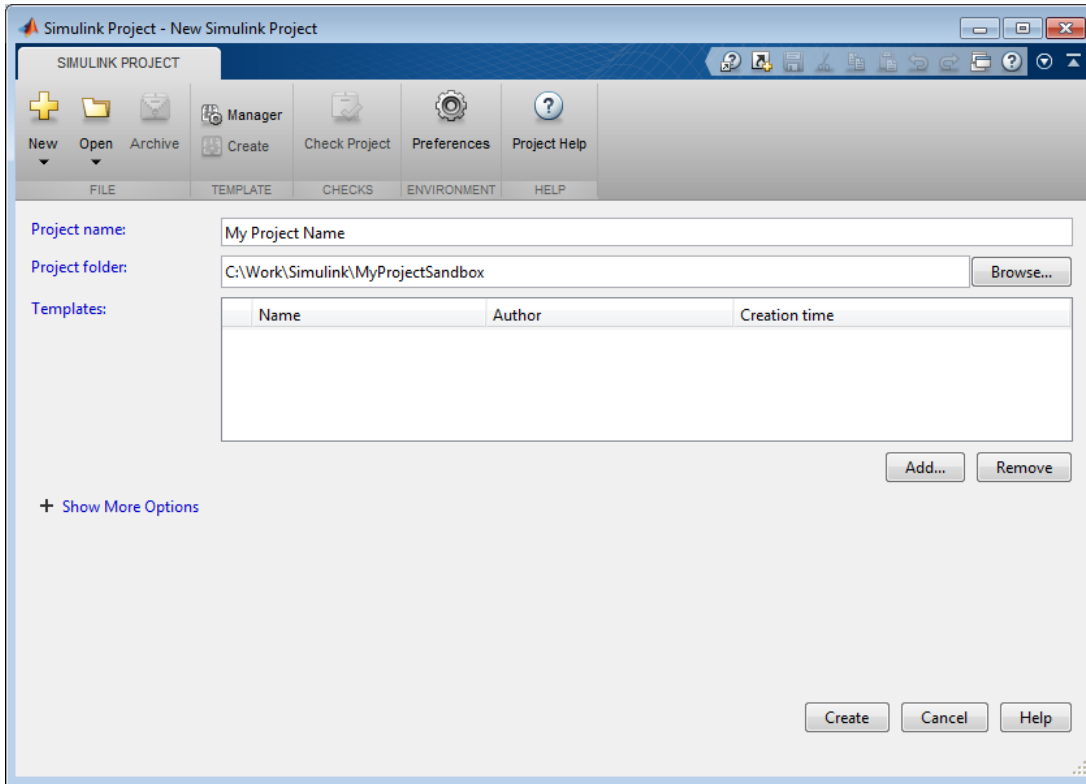
If you have files that you want to organize into a project, with or without source control, use the following steps for a new project.

Note If you want to retrieve your project from a source control repository, see instead “Retrieve a Working Copy of a Project from Source Control” on page 13-110.

To create a new project to manage your files:

- From MATLAB, on the **Home** tab, in the **File** section, select **New > Simulink Project > Blank**.
- From the Model Editor, select **File > New > Simulink Project**.
- From the Simulink Project, on the **Simulink Project** tab, click **New**.

The Create Simulink Project dialog box appears.



- 1 Specify the project name and location. The default project folder is your current folder.
- 2 If you want, click **Add** to use the Select a Template dialog box to add one or more templates to apply to the project. See “Create a New Project Using Templates” on page 13-28. Note: avoid adding templates if you are creating a project in a folder that already contains files, unless you know the template files are compatible. You see a warning if a template will overwrite files.
 - Try the Project Environment template if you are creating a project in a new folder and intend to add files later. The Project Environment template can create a new project with a preconfigured structure, and utilities to configure the path and environment with startup and shutdown shortcuts. For example, the path utilities help set up your

search path to ensure dependency analysis can detect project files. You can modify any of these files, folders, and settings later.

- Try the **Code Generation Example** template to set up a project with settings for production code generation of a plant and controller. This template requires Simulink Coder and Embedded Coder.
- Create your own templates. See “Using Templates to Create Standard Project Settings” on page 13-49.

3 If your project folder is already under SVN source control, click **Show More Options**, then click **Detect** to check the project folder for source control integration options. See below for details.

4 Click **Create** to create the project.

The Simulink Project displays the empty Project Files list for the specified project root. Your project does not yet contain any files. You need to select files to add. For next steps, see “Add Files to the Project” on page 13-25.

If you have existing source control in your project folder, you might need the following options when creating projects. Click **Show More Options** to expand them.

- Click **Detect** to check the project folder for source control integration options. If your selected folder is under source control that the project can recognize, the **Source control integration** field reports the detected source control. You might want to use this option to use a specific version of SVN. See “Set Up SVN Source Control” on page 13-101.

Note You can put the project under source control later by using the Source Control node in the Project tree. See “Add a Project to Source Control” on page 13-92.

- Change project definition files.

Use **multiple project files** is the default and is better for avoiding merging issues on shared projects.

Use a single project file is faster but is likely to cause merge issues when multiple users submit changes in the same project to a source control tool. See “Project Definition Files” on page 13-129.

Related Examples

- “Add Files to the Project” on page 13-25
- “Work with Project Files” on page 13-60
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Add a Project to Source Control” on page 13-92

Concepts

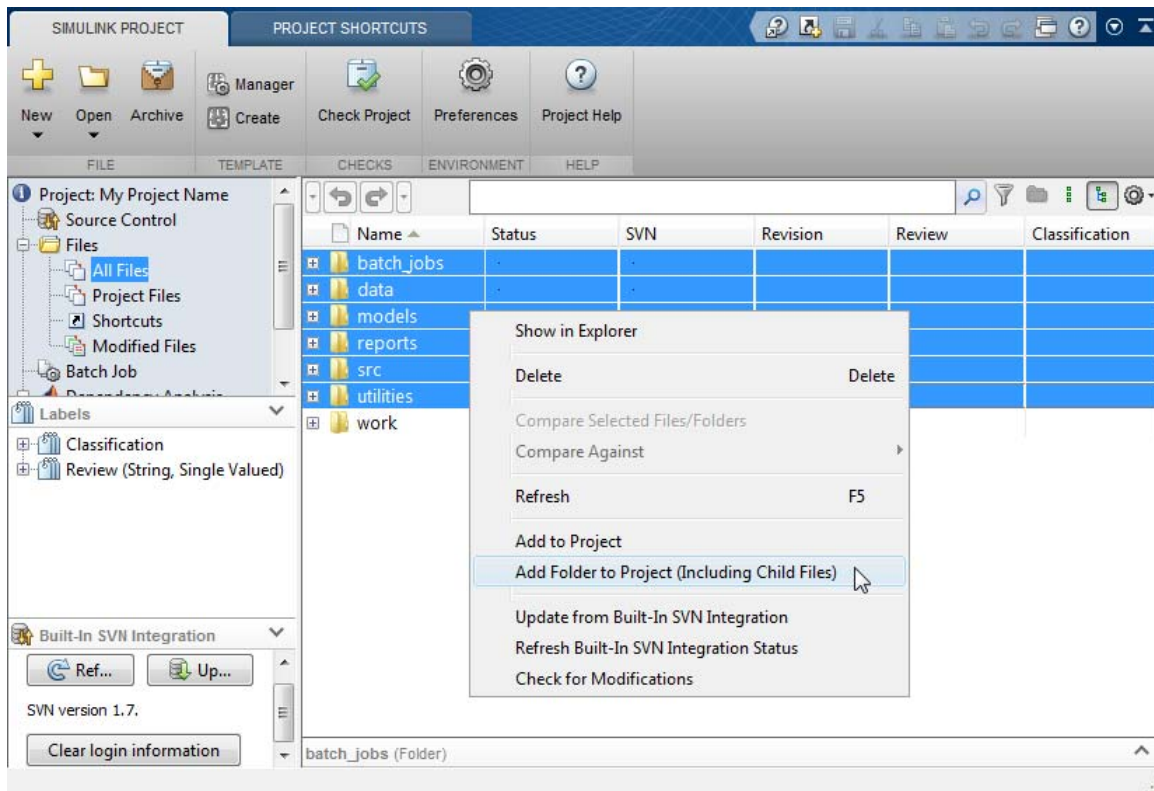
- “What Can You Do With Project Shortcuts?” on page 13-35

Add Files to the Project

After creating a Simulink Project, the All Files node in the Project tree shows all files in your project folder (or projectroot). Files under your chosen project root are not included in your project until you add them. 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 files to your project, use any of these methods:

- In the All Files view, select files or folders, right-click, and select **Add to Project** or **Add to Project (including child files)**.



- Drag and drop files from a file browser or the Current Folder browser onto the Project Files view to add files to your project. If you drag a file from outside the project root, this copies the file into your project. If you drag a file within project root, you move the file.
- Add and remove project files at the command line using `addFile`.

To learn how to set up your project with all required files, see “Choose Files and Run Dependency Analysis” on page 13-141.

To configure your project to automatically run startup and shutdown tasks, see “Automate Startup Tasks with Shortcuts” on page 13-36.

You can access your recent projects direct from MATLAB. See “Open Recent Projects” on page 13-32.

If you want to add source control, see “Add a Project to Source Control” on page 13-92.

Related Examples

- “Work with Project Files” on page 13-60
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Add a Project to Source Control” on page 13-92

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

Create a New Project from an Archived Project

To create a new project from an archived project:

- From MATLAB, on the **Home** tab, in the **File** section, select **New > Simulink Project > From Archive**.
- From the Simulink Project, on the **Simulink Project** tab, select **New > From Archive**

For details, see “Archive Projects in Zip Files” on page 13-80.

Create a New Project Using Templates

In Simulink Project, you can apply one or more templates when creating a project.

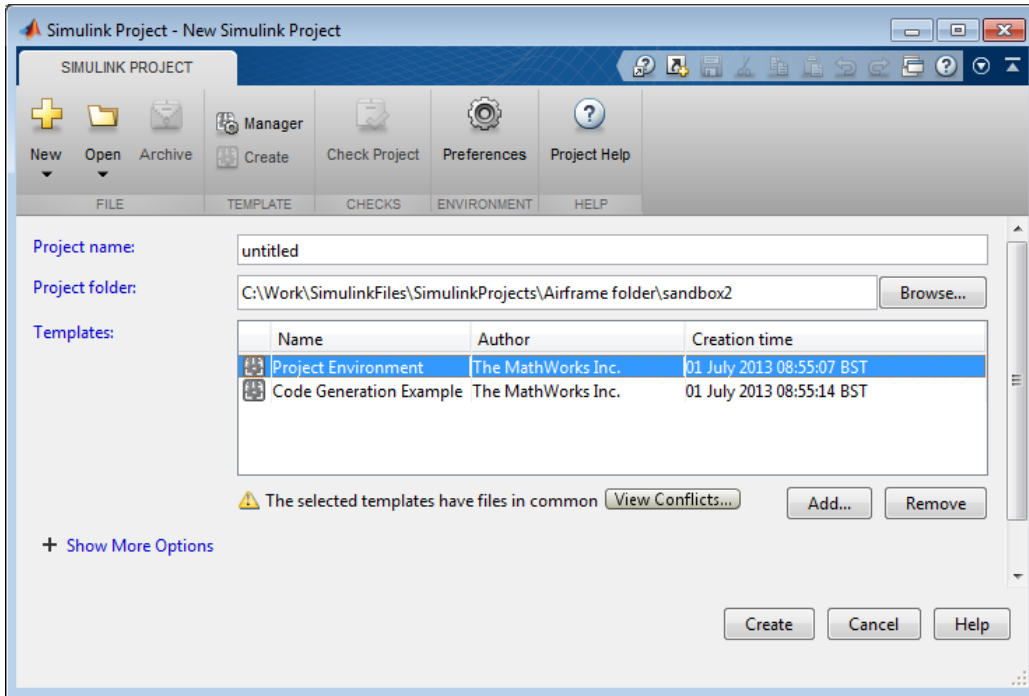
- 1** To browse for templates, on the Simulink Project tab, select **New > Choose Template**.

The Select a Template dialog box opens.

- 2** Click a template in the list to read the description. For example, click **Project Environment**.

Only templates on the template path appear in the dialog. If your templates do not appear, use the Template Manager to add to the template search path. See “View and Validate Templates” on page 13-53.

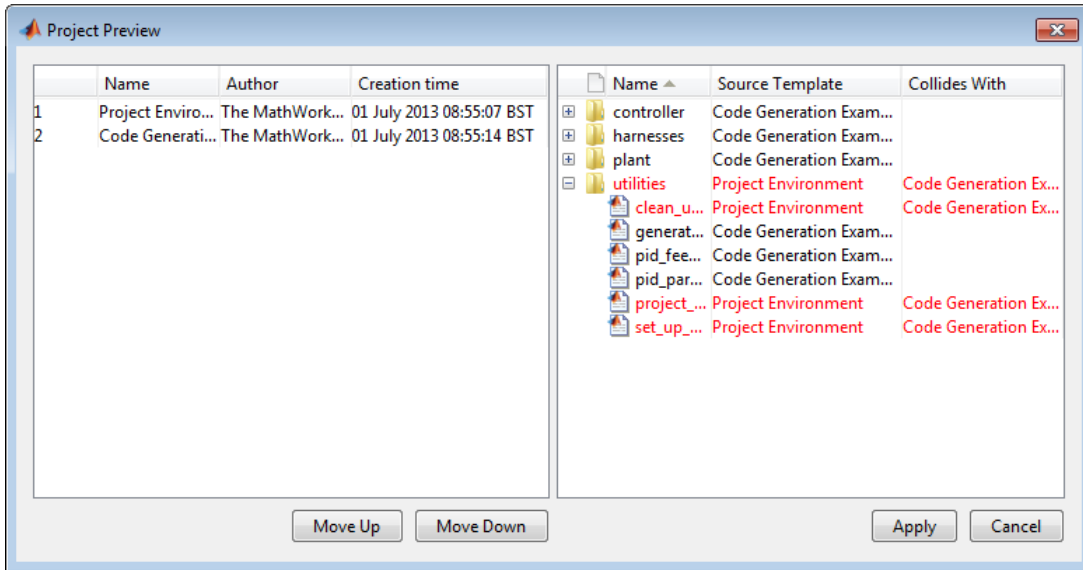
- 3** Click the **Files** tab to check the file names in the template.
- 4** Click **Select** to use the template and return to the New Project dialog box.
- 5** You can click **Add** to choose additional templates to apply to the same project.
- 6** If the templates conflict with each other or with existing files in the selected project folder, a warning message appears in the New Project dialog box, e.g., The selected templates have files in common.



Click **View Conflicts** to examine the conflicts.

The Project Preview dialog box opens.

- 7 Examine the conflicted files, shown in red with a warning icon. If you are applying multiple templates and they conflict, you can choose template priority. To change which template to use when there is a conflict, select a template and click **Move Up** or **Move Down** to change priority. Your choice of template priorities applies to all conflicts.



The figure shows a combination of files from two example templates. In this case, the project paths are not compatible. Ensure you choose compatible files when you combine templates.

- 8 When you are satisfied with the priority order for conflicted files, click **Apply**.
- 9 In the New Project dialog box, specify your project folder and click **Create** to create the new project using the selected templates.

Note If you create a project in a folder that already contains files, a warning appears if there are any conflicts with the template. The template overwrites the existing files only if you choose to continue.

You can select recently used project templates direct from the **New** menu in either MATLAB or the Simulink Project. For example, to use the Project Environment template, from the MATLAB **Home** tab, select **New > Simulink Project > Project Environment**.

On the Simulink Project tab, select a template from the **New > Templates** list.

Related Examples

- “Create a Template from the Current Project” on page 13-50
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

Open Recent Projects

Note You can have one project open at a time, to avoid conflicts. If you open another project, any currently open project closes.

You can use any of these methods to open recent Simulink projects:

- On the **MATLAB Home** tab, click the **Open** arrow and select your project under the **Recent Simulink Projects** list.
- From the Current Folder browser, double-click the `.prj` file.
- On the **Simulink Project** tab, click the **Open** arrow and select your project under the **Recent** list.
- For projects created or saved in Release 2012b or later, select **Open > Open Project File**. Browse and select your project `.prj` file.
- For projects saved in Release 2012a or earlier, select **Open > Open Project by Folder**. Navigate to the folder containing the `.SimulinkProject` folder, and click **OK** to load the project. After you load this project once, then you can use **Open Project File** to open the project.

To open the Simulink Project:

- From the MATLAB command line, enter:

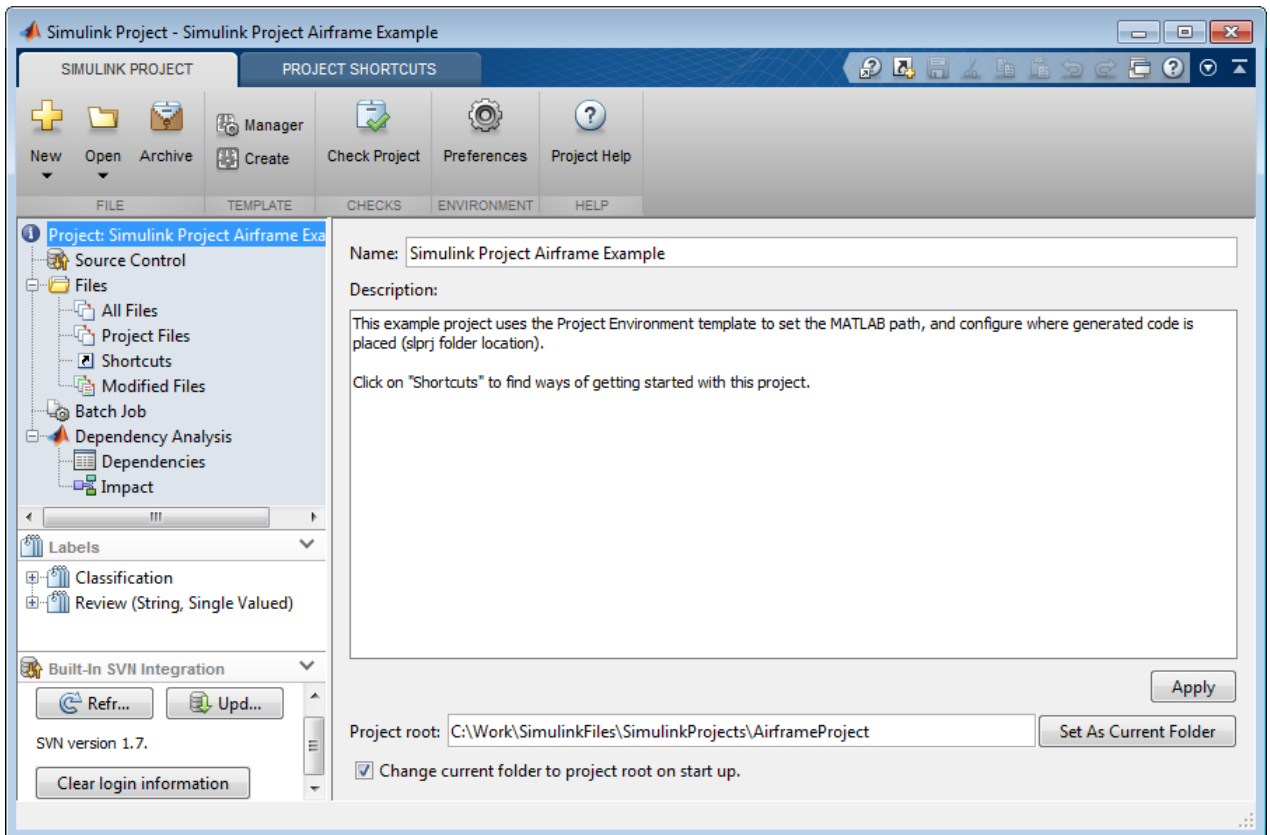
```
simulinkproject
```
- From the Library Browser or Simulink Editor, select **View > Simulink Project**.

Tip Create a MATLAB shortcut for opening or giving focus to the Simulink Project by dragging this command to the toolstrip from the Command History or Command Window:

```
simulinkproject
```

Change the Project Name, Root, Description, and Startup Folder

In Simulink Project, use the Project tree node to edit the project name or add a description. If you edit the name or description, click **Apply** to save your changes.



You can view the project root folder, and click **Set as Current Folder** to change the current working folder to your project root. 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.

The check box option **Change current folder to project root on startup** is selected by default. Clear the check box if you do not want to change to the project root folder on startup.

You can also configure startup shortcut scripts that set the current folder and perform other setup tasks. If you configure startup shortcuts to set the current folder, your shortcut setting takes precedence over the check box at the Project node. To set up shortcuts, see “Automate Startup Tasks with Shortcuts” on page 13-36.

What Can You Do With Project Shortcuts?

In Simulink 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.

You can automate shortcuts to perform startup and shutdown tasks. Startup shortcuts help you set up the environment for your project. Shutdown shortcuts help you clean up the environment for the current project when you close it.

In the Shortcuts view, you can execute, group, annotate, or automate shortcuts. You can specify Startup, Shutdown, or basic shortcuts.

- Startup shortcuts run when you open your project.
- Shutdown shortcuts run when you close your project.
- Basic shortcuts run when you execute them manually from the context menu or Project Shortcuts tab in the toolbar.

Related Examples

- “Automate Startup Tasks with Shortcuts” on page 13-36
- “Set Project Path at Startup and Reset at Shutdown” on page 13-39
- “Automate Shutdown Tasks with Shortcuts” on page 13-41
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47

Automate Startup Tasks with Shortcuts

In Simulink Project, startup shortcuts help you set up the environment for your project.

Startup shortcut files are automatically run (.m files), loaded (.mat files), and opened (Simulink models) when you open the project.

Note Files named `startup.m` on the MATLAB path run when you start MATLAB. If your `startup.m` file calls the project with `simulinkproject`, an error appears because no project is loaded yet. To avoid the error, rename `startup.m` and use it as a project startup shortcut instead.

Projects warn if you have more than one startup shortcut. Startup shortcuts run in alphabetical order. If execution order is important, consider creating one script that calls all the others, and use that script as your only startup shortcut.

Create a new startup shortcut file.

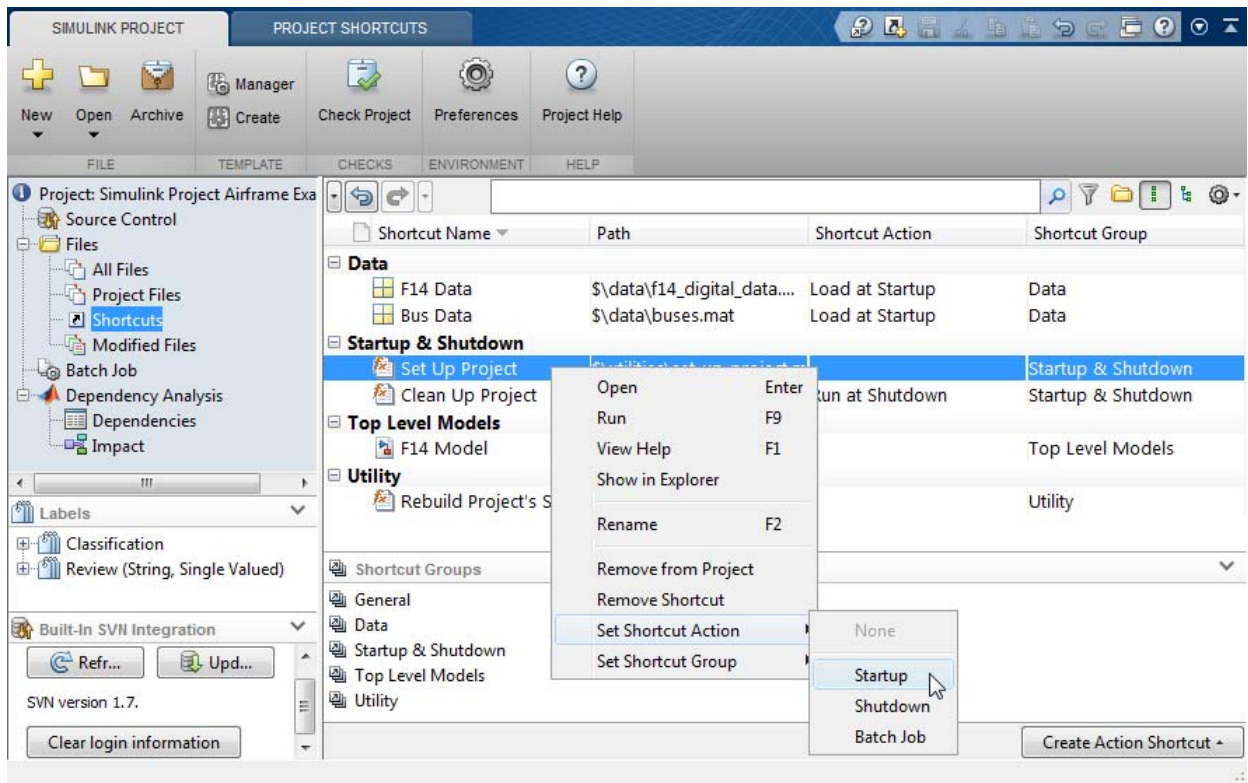
- 1** In the Shortcuts view, click **Create Action Shortcut > Startup Script**.
- 2** Name and save the file. Startup shortcut scripts can have any name. You do not need to use `startup.m`.

The tool adds the new file to the project and sets it to Run at Startup.

- 3** Right-click the file and select **Open**.
- 4** Write the shortcut script using the Editor.
- 5** Click **Save**.

Configure an existing file as a shortcut to run when you open your project.

- 1 Right-click the file in the All Files or Project Files view and select **Create Shortcut > General**, or if you have created a shortcut group you want to use, then select **Create Shortcut > GroupName**. You can change shortcut group later.
- 2 Click Shortcuts.
- 3 Right-click the shortcut file and select **Set Shortcut Action > Startup**.



The Action column displays Run at Start Up.

Note Shortcuts are included when you commit modified files to source control. Any startup shortcuts you create run for all other project users.

To stop a shortcut running at startup, change back to a basic shortcut using **Set Shortcut Action > None**.

Related Examples

- “Set Project Path at Startup and Reset at Shutdown” on page 13-39
- “Automate Shutdown Tasks with Shortcuts” on page 13-41
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

Set Project Path at Startup and Reset at Shutdown

In Simulink Project, you can view an example startup shortcut file in the Airframe example project that sets the MATLAB path, and defines where to create the `slprj` folder. The project also contains a shutdown shortcut to reverse these changes. You can copy and modify these files to set up your own project path.

- 1 Open the Airframe project.

```
sldemo_slproject_airframe
```

- 2 In the Simulink Project, click the Shortcuts node.

- 3 Open `set_up_project.m`.

- 4 In the Editor, observe how the script sets the path for the project.

```
% Set the path for this project.  
folders = project_paths();  
for jj=1:numel(folders)  
    addpath( fullfile(projectRoot, folders{jj}) );  
end
```

This code uses the file `project_paths` to set the path.

- 5 In the Simulink Project, open `clean_up_project.m`.

- 6 Observe how this script removes the paths added at startup.

```
% Remove paths added for this project. Get the single definition  
% of the folders to add to the path:  
folders = project_paths();  
  
% Remove these from the MATLAB path:  
for jj=1:numel(folders)  
    rmpath( fullfile(projectRoot, folders{jj}) );  
end
```

This code uses the same file `project_paths` to define the folders to add and remove from the path in a single place.

- 7 In the Simulink Project, select the Project Files view and expand the utilities folder.
- 8 Open `project_paths.m`.
- 9 Observe how the file defines a list of folders to add and remove from the path at startup and shutdown.

```
folders = { ...  
    'batch_jobs', ...  
    'data', ...  
    'models', ...  
    'utilities', ...  
    'work' ...  
};
```

Edit this list of folders to set up your own project path.

You can use the `Project Environment` template to add these three shortcut files when you create a new project. Then modify them to set up your path as desired. For details, see “Using Templates to Create Standard Project Settings” on page 13-49.

Alternatively, copy the files to an existing project, edit them as needed, add them to the project, and set them to run at startup and shutdown.

Related Examples

- “Automate Startup Tasks with Shortcuts” on page 13-36
- “Automate Shutdown Tasks with Shortcuts” on page 13-41
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

Automate Shutdown Tasks with Shortcuts

In Simulink Project, shutdown shortcuts help you clean up the environment for the current project when you close it. Shutdown shortcuts should undo the settings applied in startup shortcuts. You can view an example in the `sldemo_slproject_airframe` project.

Create a new shutdown shortcut file.

- 1 In the Shortcuts view, click **Create Action Shortcut > Shutdown Script**.

- 2 Name and save the file.

The tool adds the new file to the project and sets it to Run at Shutdown.

- 3 Right-click the file and select **Open**.

- 4 Write the shortcut script using the Editor.

- 5 Click **Save**.

Configure an existing file as a shortcut to run when you close your project.

- 1 Right-click the file in the All Files or Project Files view and select **Create Shortcut > General**, or if you have created a shortcut group you want to use, then select **Create Shortcut > GroupName**. You can change shortcut group later.

- 2 Click the Shortcuts node.

- 3 Right-click the shortcut file and select **Set Shortcut Action > Shutdown**.

The Action column displays Run at Shutdown.

Note Shortcuts are included when you commit modified files to source control. Any shutdown shortcuts you create run for all other project users.

To stop a shortcut running at shutdown, change back to a basic shortcut using **Set Shortcut Action > None**.

Related Examples

- “Automate Startup Tasks with Shortcuts” on page 13-36
- “Set Project Path at Startup and Reset at Shutdown” on page 13-39
- “Create Shortcuts to Frequent Tasks” on page 13-43
- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

Create Shortcuts to Frequent Tasks

In this section...

“Create Shortcuts” on page 13-43

“Group Shortcuts” on page 13-44

“Annotate Shortcuts to Use Meaningful Names” on page 13-45

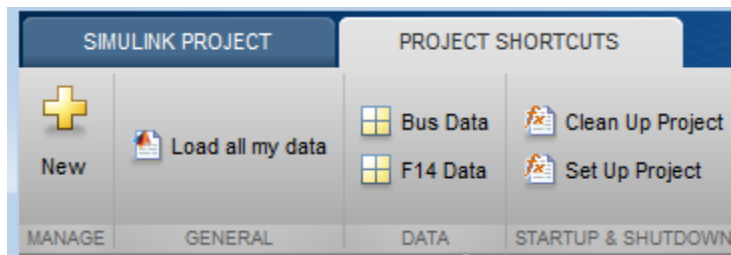
Create Shortcuts

In Simulink 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 startup code (e.g., change the path or load data), simulate models, or run shutdown code.

To create a shortcut to an existing project file, use any of the following methods:

- Right-click the file in the All Files or Project Files view and select **Create Shortcut > General**, or **ShortcutGroupName**.
- Click **New** 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.



You can create new shortcuts interactively only in the Simulink Project, but you can get and view your shortcuts at the command line. For details, see “Query Shortcuts” on page 13-87.

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.

By default, new shortcuts appear in the General group on the Project Shortcuts toolstrip tab and in the Shortcuts view.

Create new shortcut groups to organize your shortcuts:

- Right-click in the Shortcut Groups pane and select **Create New Shortcut Group**. Enter a name for the group and click **OK**.

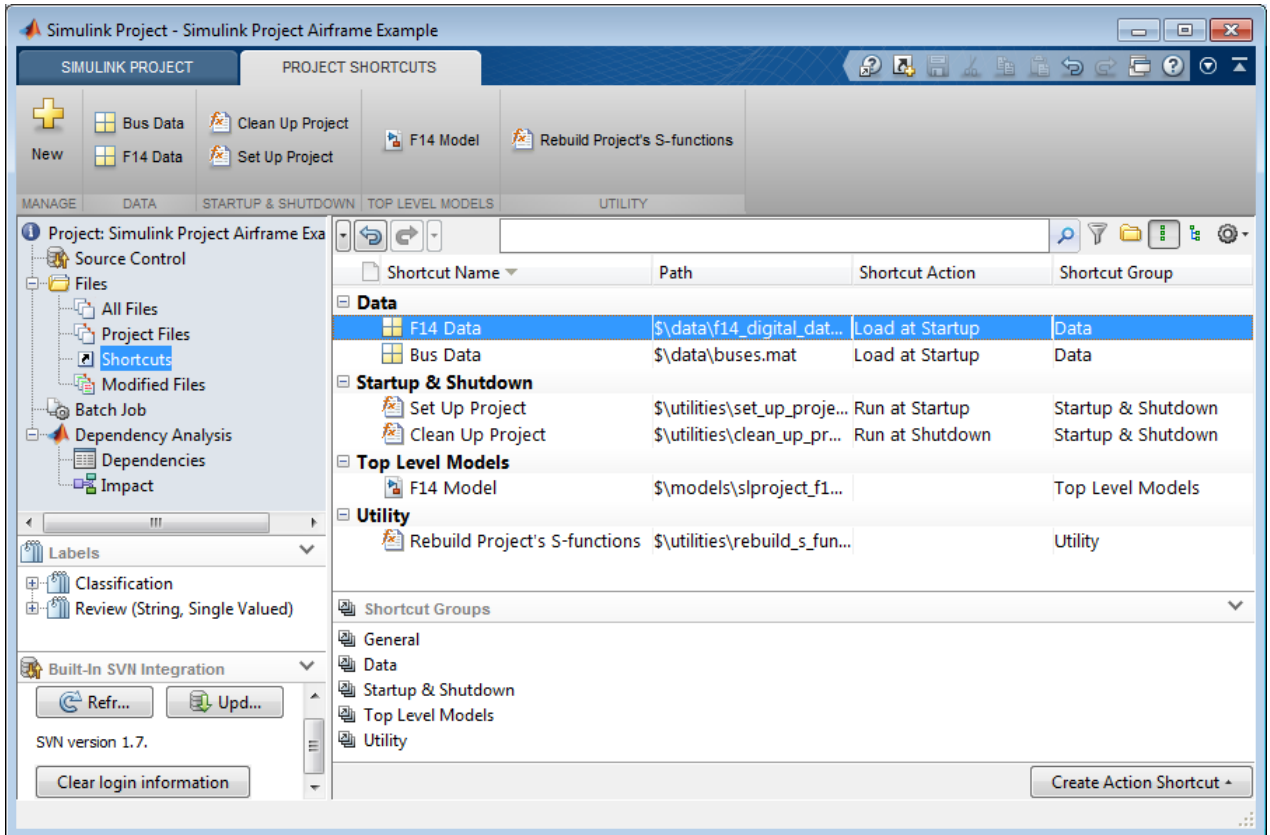
The new shortcut group appears in the Shortcut Groups pane.

- Alternatively, click **New** in the Project Shortcuts tab. Right-click in the Select a Shortcut Group pane to create a new group.

To organize your shortcuts by group, either:

- In the All Files or Project Files views, right-click a file and select **Create Shortcut > ShortcutGroupName**.
- In the Shortcuts view, drag a shortcut group onto a file from the Shortcut Groups pane, or right-click a file and select **Set Shortcut Group > ShortcutGroupName**.

The shortcuts are organized by group in the Project Shortcuts tab and in the Shortcuts view.



To change shortcut group, in the Shortcuts view, drag a different group onto the file, or right-click and select **Set Shortcut Group > GroupName**.

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 useful string for the shortcut name.

In the Shortcuts view, right-click and select **Rename** to edit the **Shortcut Name** column. 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.

Related Examples

- “Use Shortcuts to Find and Run Frequent Tasks” on page 13-47

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

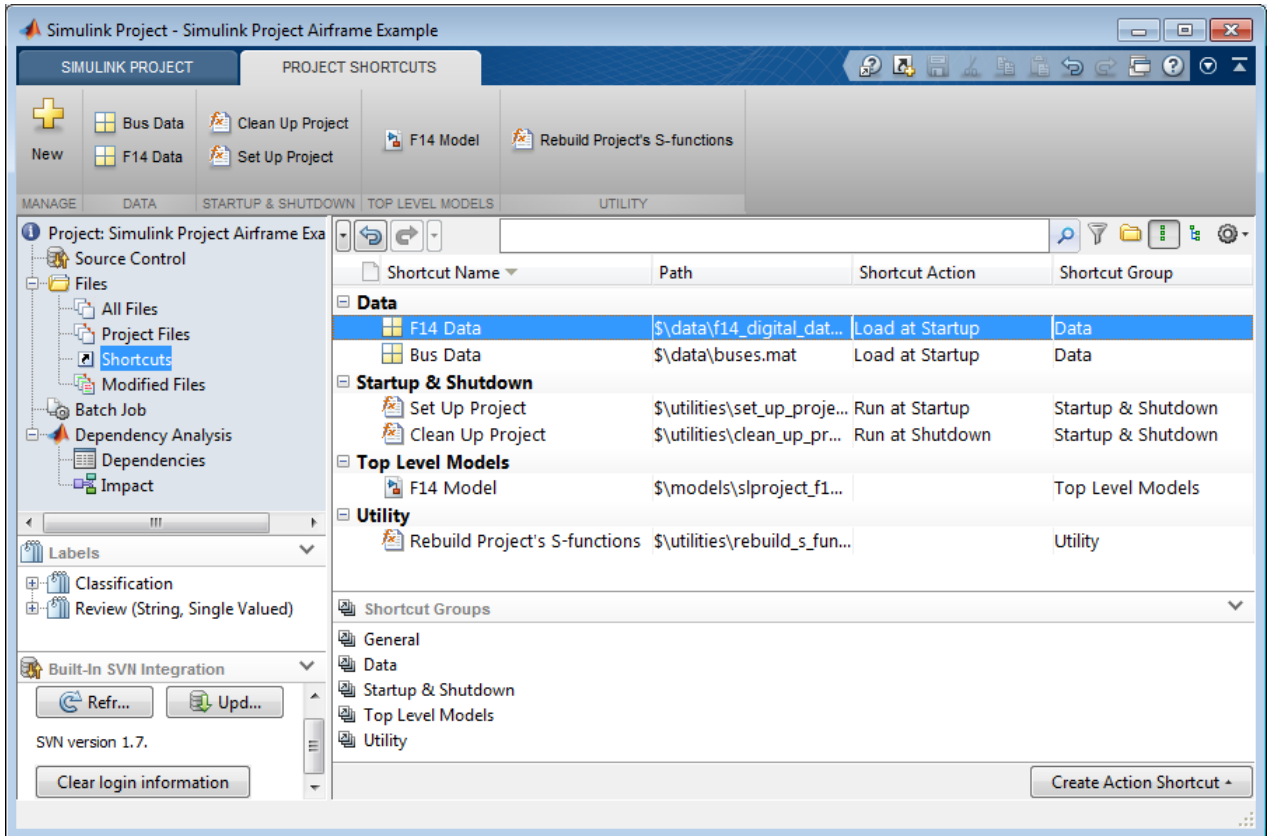
Use Shortcuts to Find and Run Frequent Tasks

In Simulink 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 toolbar.

If your project does not yet contain any shortcuts, see “Create Shortcuts to Frequent Tasks” on page 13-43.

To use shortcuts:

- In the Project Shortcuts tab, click the shortcut. Clicking a shortcut in the toolbar 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.
- Alternatively, select the Shortcuts view, right-click the shortcut file, and select an action, such as **Open** or **Run**.



- To make a shortcut available in the Batch Job list without browsing for the file, select **Set Shortcut Action > Batch Job**.

Related Examples

- “Create Shortcuts” on page 13-43
- “Group Shortcuts” on page 13-44
- “Annotate Shortcuts to Use Meaningful Names” on page 13-45
- “Create Shortcuts to Batch Job Functions” on page 13-71

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35

Using Templates to Create Standard Project Settings

In Simulink 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.

Related Examples

- “Create a Template from the Current Project” on page 13-50
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

Create a Template from the Current Project

In Simulink 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.

- 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 13-51.

- 2 On the **Simulink Project** tab, in the **Template** section, select **Create**.
- 3 On the Create Template dialog box, edit the name, add a description to help template users, choose a file location, then click **Create**.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

Create a Template from a Project Under Version Control

- 1 Get a new working copy of the project. See “Retrieve a Working Copy of a Project from Source Control” on page 13-110.
- 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 Project Files view, right-click unwanted files and select **Remove from Project**.
- 4 On the **Simulink Project** tab, in the **Template** section, click **Create** 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 13-28.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from the Current Project” on page 13-50
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

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 13-28.
- 2 Make the changes.
- 3 On the **Simulink Project** tab, in the **Template** section, click **Create**.

Use the dialog box to create a new template or overwrite the existing one.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from the Current Project” on page 13-50
- “Create a Template from a Project Under Version Control” on page 13-51
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

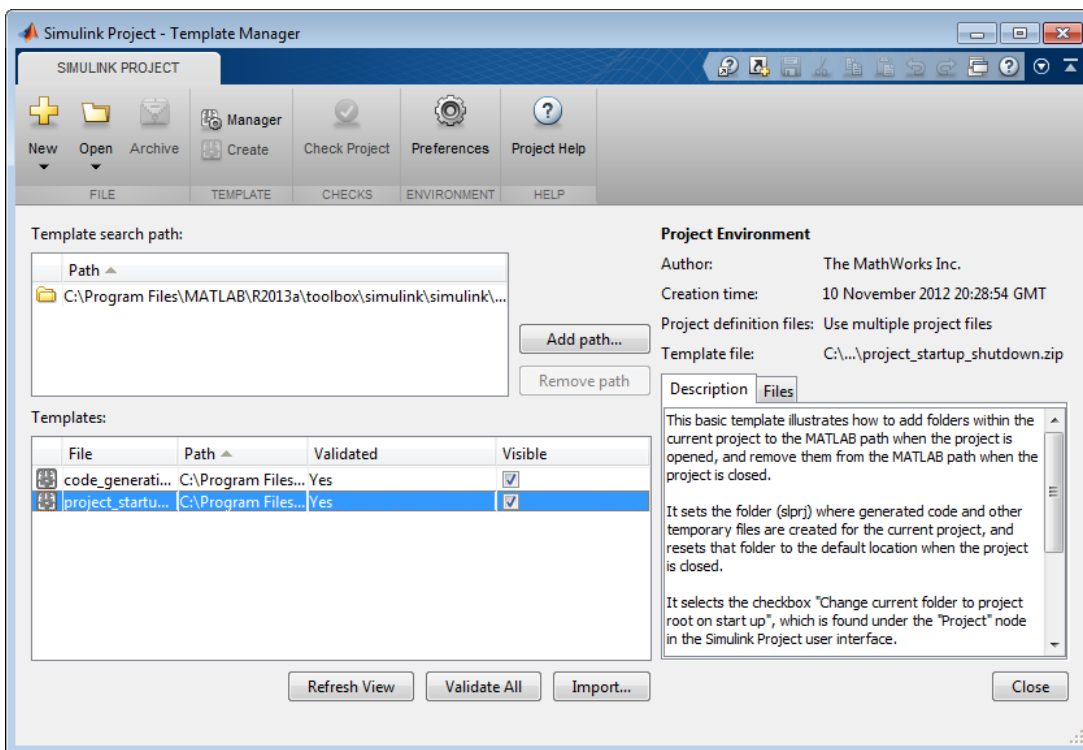
Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

View and Validate Templates

In Simulink Project, use the Template Manager to:

- View templates, including examples supplied with Simulink.
 - Locate template files.
 - Validate that a template is usable to create new projects.
 - Add folders to the template path to make templates accessible from the New Project dialog box.
 - Specify the templates to make visible in the Select a Template dialog box.
- 1 To open the Template Manager, on the **Simulink Project** tab, in the **Template** section, click **Manager**.



- 2 Click a template in the **Templates** pane to view the description of the template functions.
- 3 Click the **Files** tab to inspect files and folders in the template.
- 4 To add folders to the template search path, click **Add path**, then browse and select the folder containing templates. All templates in the folder appear in the list.
- 5 Right-click a template and select **Validate**, or click **Validate All**. The Template Manager displays **Yes** in the **Validated** column if the template is usable to create new projects.
- 6 Select the **Visible** check box to make the template visible in the Select a Template dialog box.

See also “Import Templates” on page 13-55.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “Import Templates” on page 13-55
- “Explore the Example Templates” on page 13-56

Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

Import Templates

Use import to copy a template onto your local template path, to make it available to use in new projects.

- 1** On the **Simulink Project** tab, in the **Template** section, click **Manager**.
- 2** Click **Import**.
- 3** Browse to the zip file that contains the template.
- 4** Select a location on the template path to save the template.
- 5** Click **Import** to validate the template and import it. If the template validates, it imports.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Explore the Example Templates” on page 13-56

Concepts

- “Using Templates to Create Standard Project Settings” on page 13-49

Explore the Example Templates

Example templates are supplied with Simulink Project. You can use these templates as example structures for a new project. The templates are:

- The **Project Environment** template, which shows how to set up a project with a preconfigured structure and utilities to configure the path and environment with startup and shutdown shortcuts. For example, the path utilities help set up your search path to ensure dependency analysis can detect project files. For details, see “Set Project Path at Startup and Reset at Shutdown” on page 13-39.
- The **Code Generation Example** template, which shows how to set up a project with settings for production code generation of a plant and controller. This template requires Simulink Coder and Embedded Coder.

You can explore the templates using the Template Manager.

- 1** On the **Simulink Project** tab, in the **Template** section, click **Manager**.
- 2** Select the template you want to explore.
- 3** Read the description for detailed information about the template.
- 4** Click the **Files** tab and explore the utilities and other files provided.

To try the example templates, select them when creating a new project. See “Create a New Project Using Templates” on page 13-28.

Related Examples

- “Create a New Project Using Templates” on page 13-28
- “Create a Template from a Project Under Version Control” on page 13-51
- “Edit a Template” on page 13-52
- “View and Validate Templates” on page 13-53
- “Import Templates” on page 13-55

Concepts

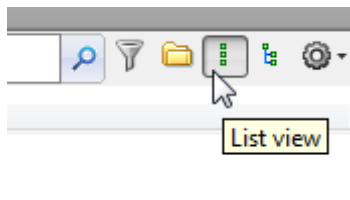
- “Using Templates to Create Standard Project Settings” on page 13-49


Group and Sort File Views

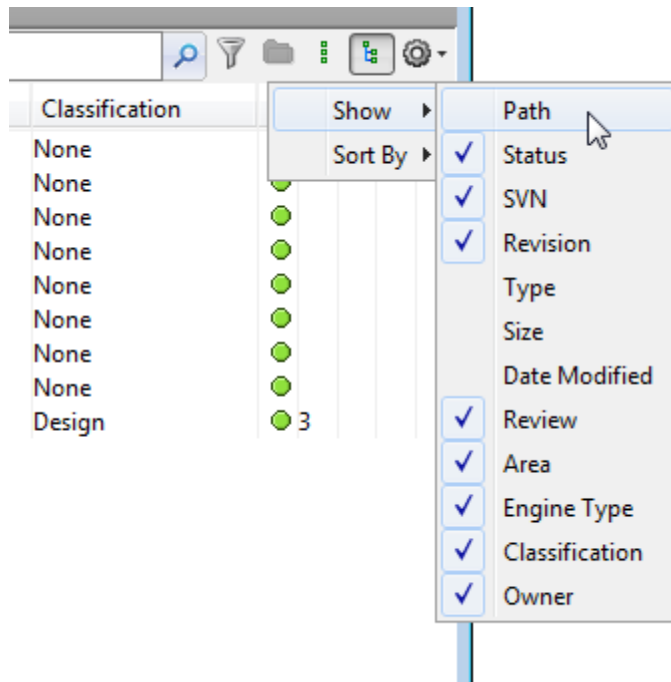
In Simulink Project, to group and sort the views in all Files views:

- Use the List view or Tree view buttons to switch between a flat list of files and a hierarchical tree of files.

In a list view, you can click the **Hide Folders** button if you want to view only files.



- Click the Actions button  to select the columns to show and select groupings, for example, to sort by Review Status labels or any labels you have defined.




Search and Filter File Views

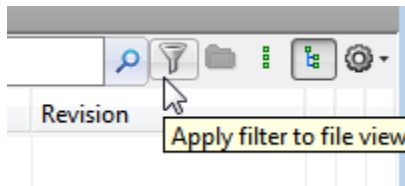
In Simulink Project, use the search box or Filter button to specify the files to display. In all the file views, and in batch job and dependency analysis nodes, you can use the search box and filtering tools.

- Type a search term in the search box, for example, part of a file name, or a file extension. You can use wildcards, e.g., *.m, or *.m*.



Click the x to clear the search.

- Click the filter button  to build a filter for the current view.

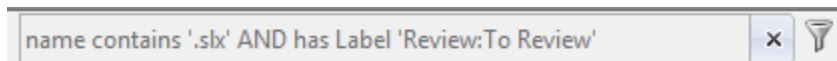


In the Filter Builder dialog box you can select multiple filter criteria to apply using names, file types, project status, and labels. Press **Ctrl** to select multiple labels.

The dialog 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 **OK**, the search box shows the filter you are applying.

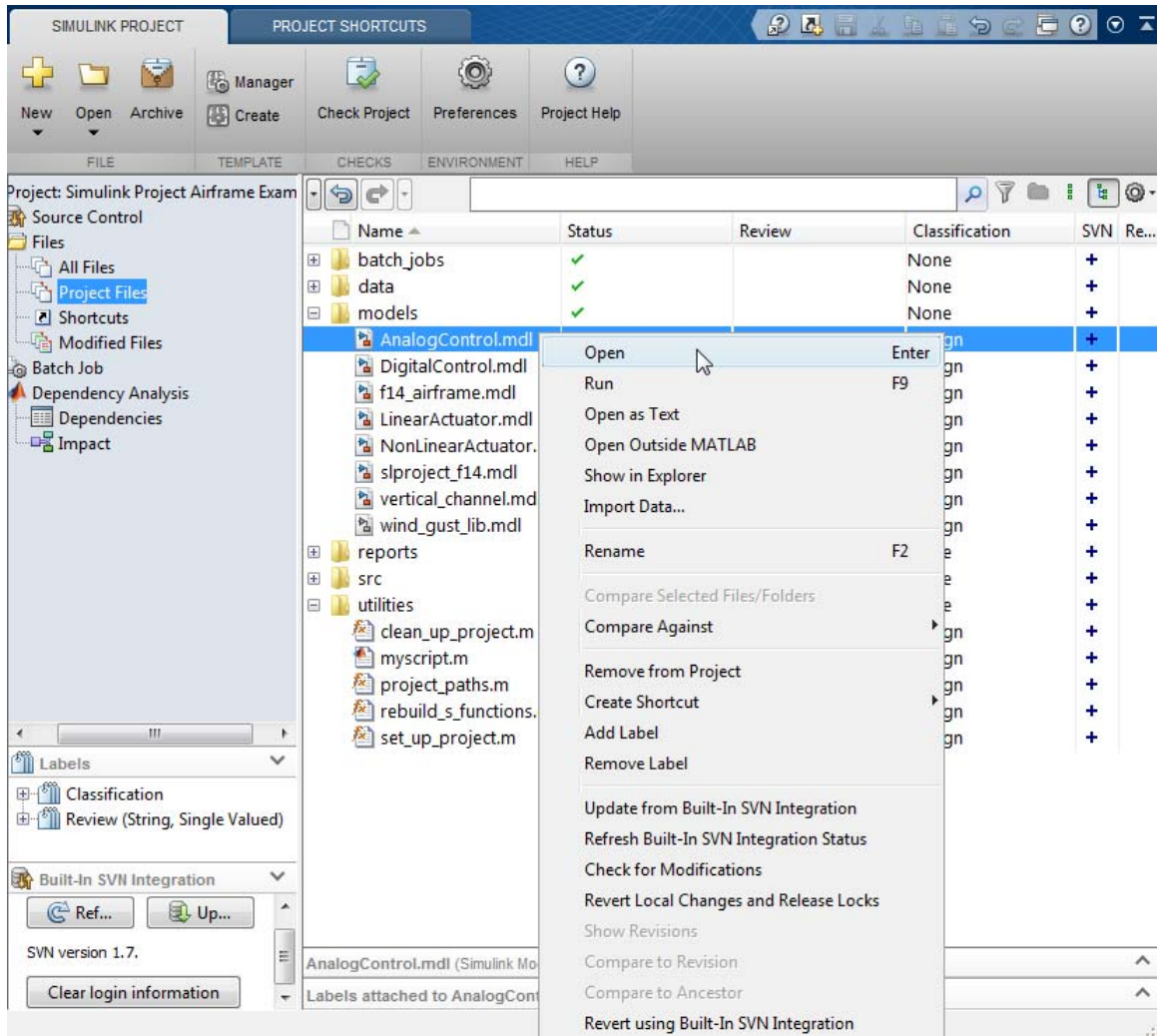


Work with Project Files

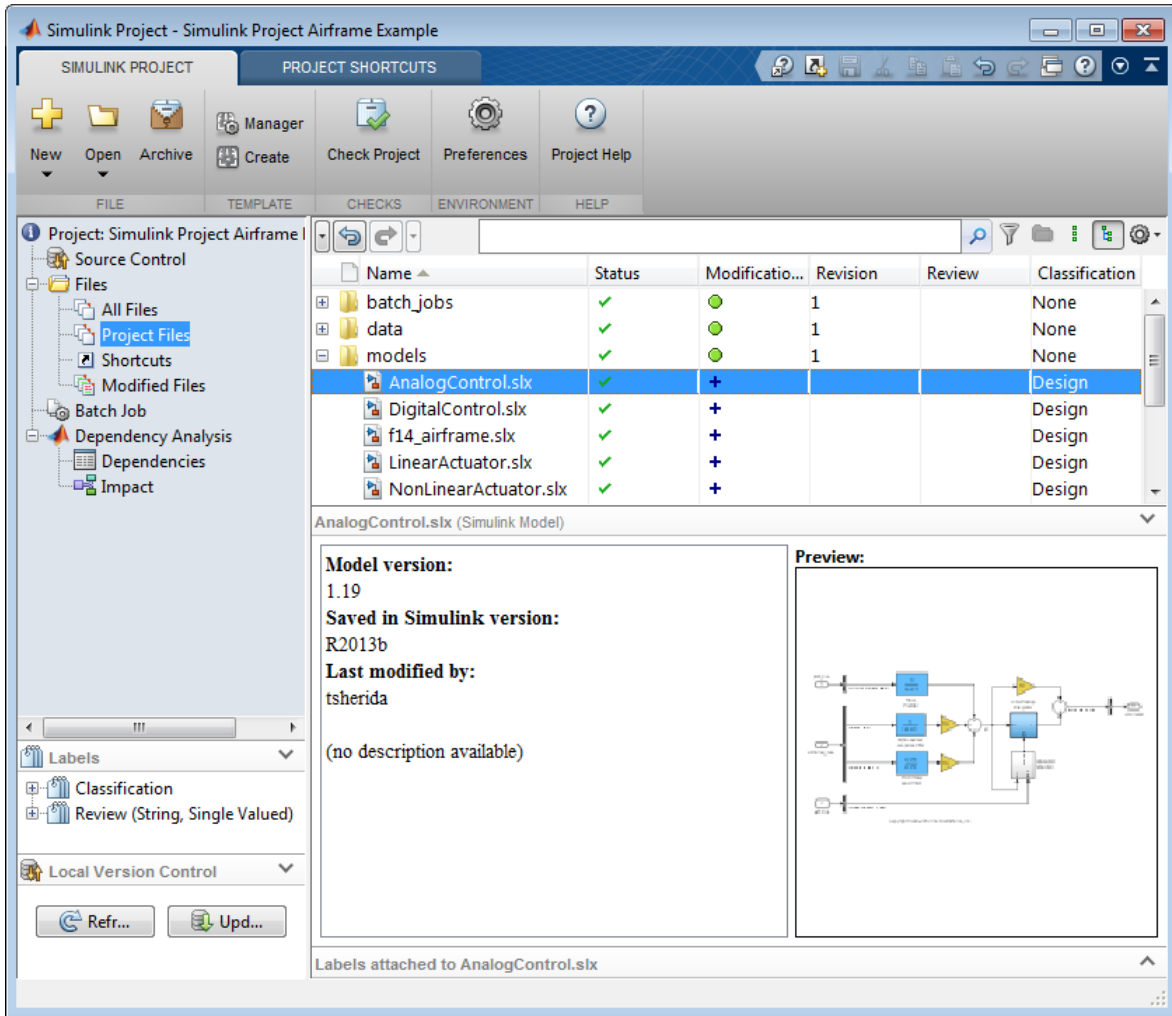
In Simulink Project, in all Files views, use the context menus to perform actions on the files that you are viewing. Right-click a file (or selected multiple files) to perform project options such as:

- Open or run files.
- Add and remove files from the project.
- Add, change and remove labels. See “Add Labels to Files” on page 13-66.
- 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 13-43.
- 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 13-90.



To view file information and preview model files, select a file and expand the lower pane under the file list.



See also .

Related Examples

- “Move Project Files” on page 13-64
- “Back Out Changes” on page 13-65
- “Group and Sort File Views” on page 13-57
- “Search and Filter File Views” on page 13-59
- “Create a Batch Function” on page 13-70

Concepts

- “What Can You Do With Project Shortcuts?” on page 13-35
- “About Source Control with Projects” on page 13-90

Move Project Files

You can drag and drop to move or add project files. Drag files in the Simulink Project to move files within your project. You can drag and drop files from a file browser or the Current Folder browser onto the All Files or Project Files view. If you drop a file in the Project Files view, you add the file to the project.

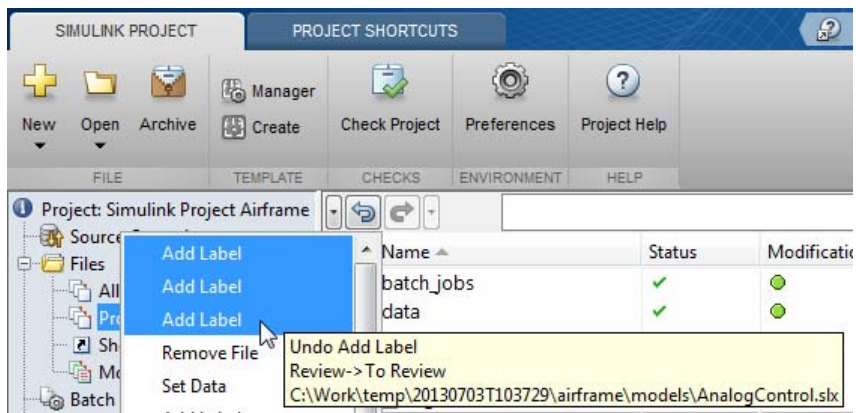
If you drag a file from outside the project root, this copies the file into your project. If you drag a file within project root, you move the file.

See also “Add Files to the Project” on page 13-25.

Back Out Changes

Similar to many applications, Simulink 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.



Add Labels to Files

In Simulink Project, use labels to organize your files. The project automatically applies built-in labels for file categories. You can create your own labels, see “Create Labels” on page 13-67.

To add labels to files, use any of these methods:

- To add a label to a single file, drag and drop the label from the Labels pane onto the file.
- To label multiple files at once in the Files views or Dependency Graph, select the files you want to add labels to, then right-click and select **Add Label**. Use the dialog box to select labels to add and click **OK**.
- To add labels programmatically, for example, in batch job functions, see “Automate Project Management Tasks” on page 13-81.

Note After it is added to a file, a label persists across file revisions.

After you add labels, you can organize files by labels in the Files views and Impact graph. See “Group and Sort File Views” on page 13-57 and “Perform Impact Analysis” on page 13-148.

If the project is under source control, adding tags to all your project files enables you to mark versions. See “Tag and Retrieve Versions of Project Files” on page 13-115.

Related Examples

- “Create Labels” on page 13-67
- “View and Edit Label Data” on page 13-68

Create Labels

In Simulink Project, to add a new category of labels:

- 1** In the Labels pane, right-click **Labels** and select **Create New Category**.
- 2** In the dialog box, enter a name for the new category.
- 3** If you want to attach only one label of the category to a file at a time, select the check box **Single Valued**.

For example, if you want to create a category with labels for Prototype and Release and want to apply only one of those labels to a specific file, select **Single Valued** to ensure only one label can be attached.

- 4** You can also change the **Type** selection to change the data type to store in the label, e.g., **Double**, **Logical**, **String**, **None**, or **Integer**.
- 5** Click **Create**.

To add a new label in any category:

- 1** In the Labels pane, right-click the category name under **Labels** and select **Create New Label**.
- 2** In the dialog box, enter a name for the new label and click **OK**.

To delete labels, right-click user-defined categories and labels and select **Remove**. You cannot delete built-in labels.

To create new labels at the command line, see “Automate Project Management Tasks” on page 13-81.

After you create labels, you can use them to label and group files. See “Add Labels to Files” on page 13-66.

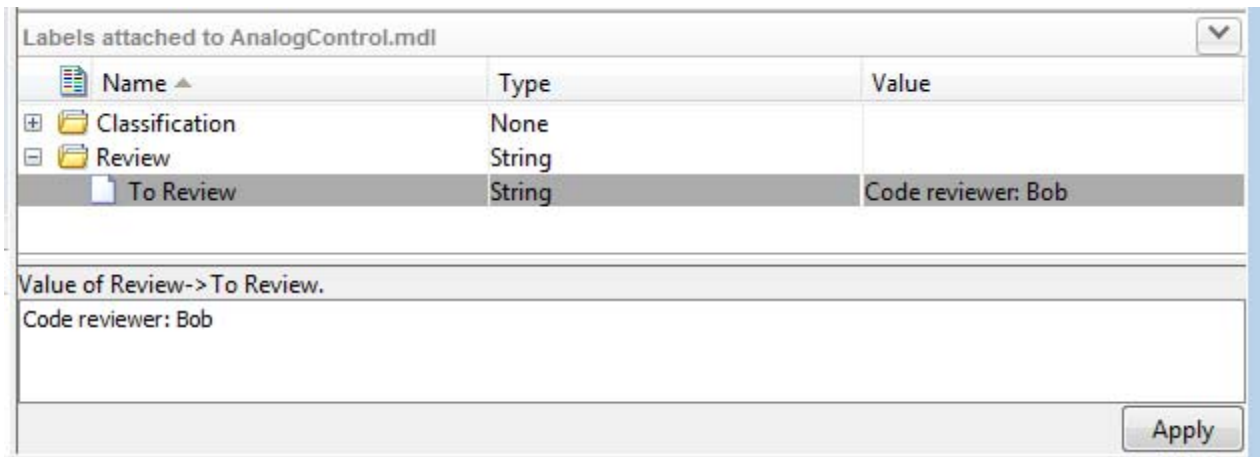
Related Examples

- “Add Labels to Files” on page 13-66
- “View and Edit Label Data” on page 13-68

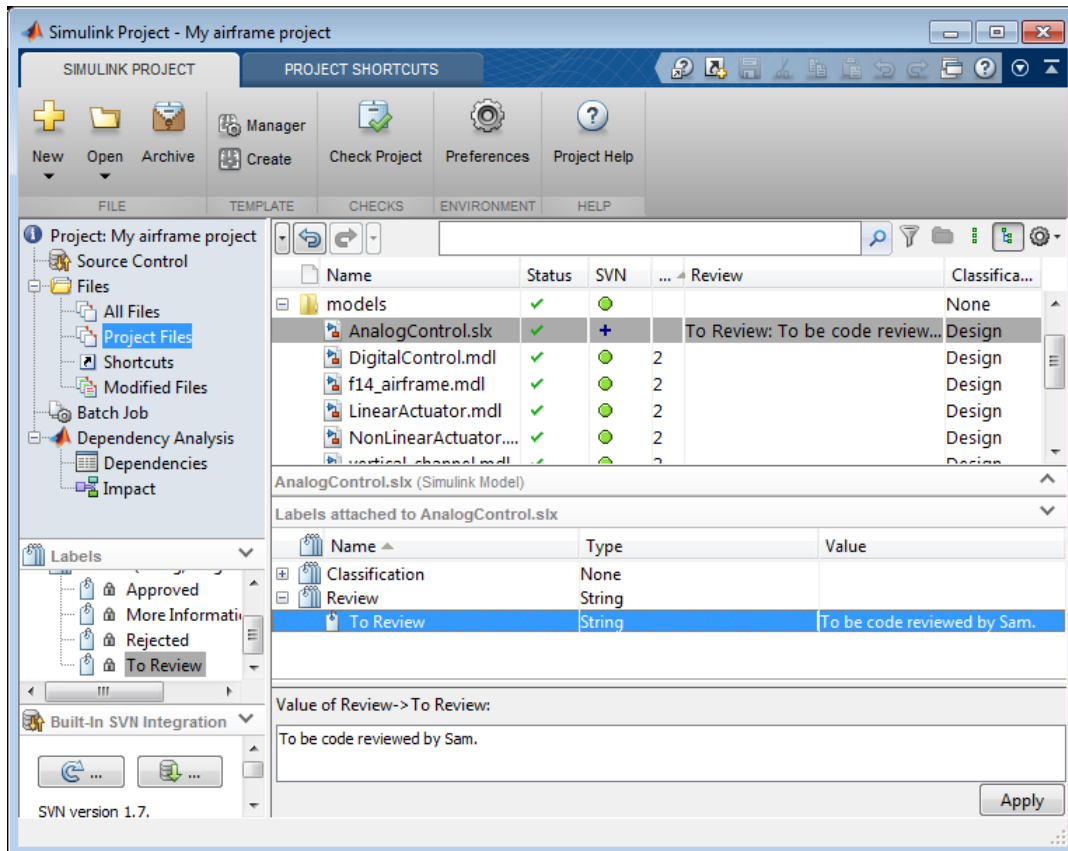
View and Edit Label Data

In Simulink Project, to view and edit label data,

- 1 Select a file in the Project Files view.
- 2 Expand the Labels pane, titled **Labels attached to *selectedFileName***.
- 3 Select a label in the label pane to view or edit any label data in the lower Value pane. After editing values, click **Apply**.



- 4 After you apply label data, you can view the data for each file in the label category column in the File views. For example, if you enter a string value for the label **To Review** for a particular file, the string displays in the **Review** category column.



Related Examples

- “Create Labels” on page 13-67
- “Add Labels to Files” on page 13-66

Create a Batch Function

In Simulink Project, in the Batch Job view, you can create functions and run them on selected project files.

For example batch jobs, see [Running Batch Jobs with a Simulink Project](#).

To create a batch function that is also a shortcut so it is easy to reuse:

- 1 In the Simulink Project, click the Shortcuts node.
- 2 Select **Create Action Shortcut > Batch Job Function**.
- 3 Name and save the file on your MATLAB path.
- 4 Edit the function to perform the desired action on each file. The file contains a simple example batch job for you to edit. The instructions guide you to create a batch job with the correct function signature.
- 5 In the Batch Job view, you can select the shortcut in the Batch Job list without browsing for the file.

To create a batch function that is not a shortcut:

- 1 In the Batch Job view, click **Create**.

The MATLAB Editor opens a new untitled file containing a simple example batch job. The instructions guide you to create a batch job with the correct function signature.

- 2 Edit the function to perform the desired action on each file.
- 3 Save the function file on your MATLAB path.

To make existing batch functions into shortcuts, see “Create Shortcuts to Batch Job Functions” on page 13-71.

Related Examples

- [Running Batch Jobs with a Simulink Project](#)

Create Shortcuts to Batch Job Functions

You can make existing batch functions in the project easier to find and reuse by making them shortcuts. These shortcuts are available in the Batch Job list without browsing for the file.

- 1** In the Project Files view, right-click the batch function and select **Create Shortcut**.
- 2** In the Shortcuts view, right-click the batch function and select **Set Shortcut Action > Batch Job**.
- 3** In the Batch Job view, click **Select** to choose from a list of Batch Job shortcuts and recently used shortcuts.

Related Examples

- “Create a Batch Function” on page 13-70
- “Run a Simulink Project Batch Job” on page 13-72

Run a Simulink Project Batch Job

- 1 In Simulink Project, in the Batch Job view, select the check boxes of project files you want to include in the batch job.

Tip If the batch function can identify the files to operate on, include all files. For example, the batch job 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**.

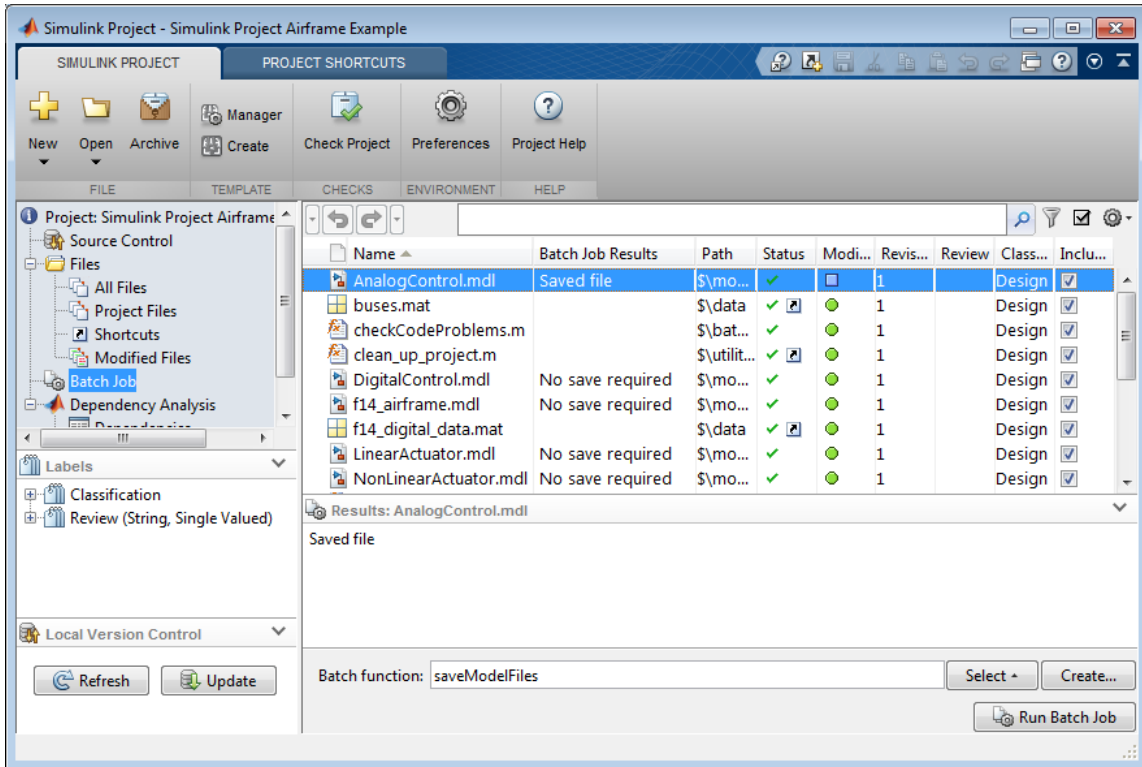
- 2 Specify the batch function to run in the Batch function box. Enter the name, or click **Select** to browse or choose from a list of Batch Job shortcuts and recently used shortcuts.

If your project does not yet contain any batch functions, see “Create a Batch Function” on page 13-70.

- 3 Click **Run Batch Job**.

Simulink Project displays the results in the Batch Job Results column.

- 4 Expand the Results pane to view details of results for the currently selected file.



Related Examples

- Running Batch Jobs with a Simulink Project

Upgrade Model Files to SLX and Preserve Revision History

In this section...

“Project Tools for Migrating Model Files to SLX” on page 13-74

“Upgrade the Model and Commit the Changes” on page 13-74

“Verify Changes After Upgrade to SLX” on page 13-77

Project Tools for Migrating Model Files to SLX

Simulink Project helps you upgrade model files from MDL format to SLX format. You can use the project integrity checks to automatically add the new SLX file to your project, remove the MDL file from the project, and preserve the revision history of your MDL file with the new SLX file. You can then commit your changes to source control and maintain the continuity of your model file history.

Upgrade the Model and Commit the Changes

- 1 Open a new copy of the airframe project.

```
sldemo_slproject_airframe
```

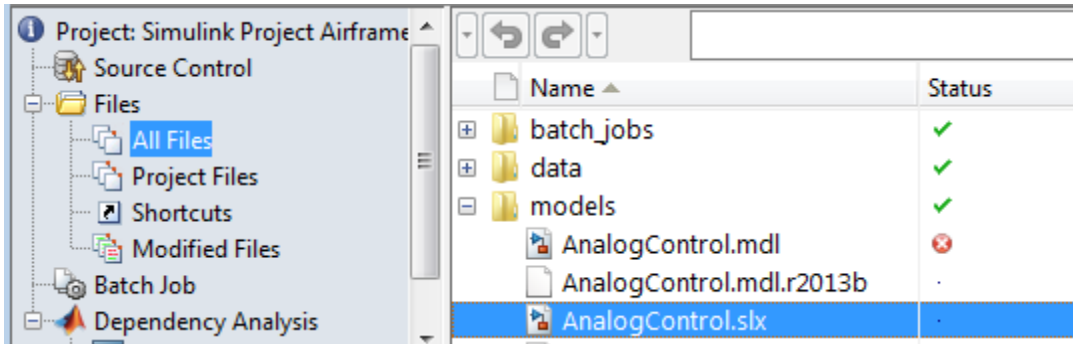
- 2 In the Project Files view, right-click the model file `AnalogControl.mdl`, and select **Open**.

- 3 Select **File > Save As**.

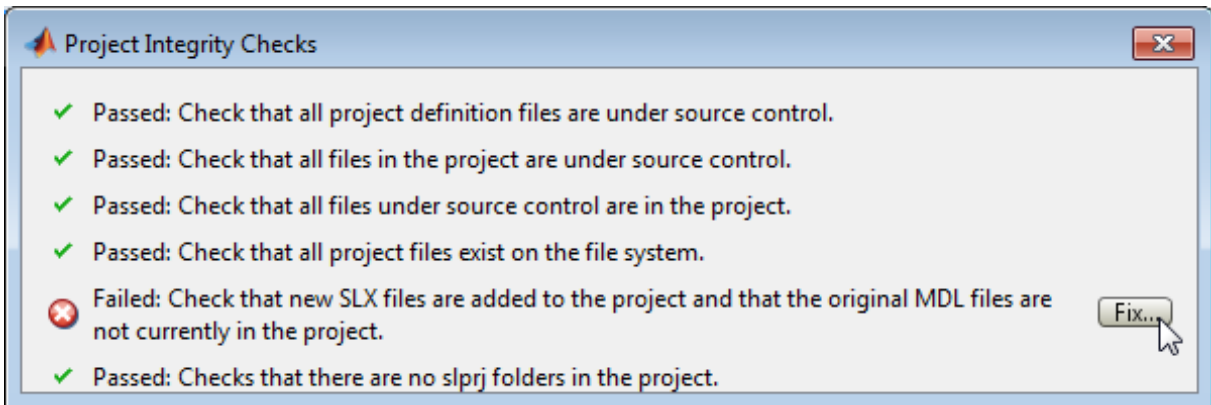
- 4 Ensure that **Save as type** is SLX, and click **Save**. SLX is the default unless you change your preferences.

- 5 Select the All Files view to see the results. Expand the `models` folder.

Simulink saves the model in SLX format, and creates a backup file by renaming the MDL file to `AnalogControl.mdl.releaseName`. The project also reports the original name of the MDL file as missing.

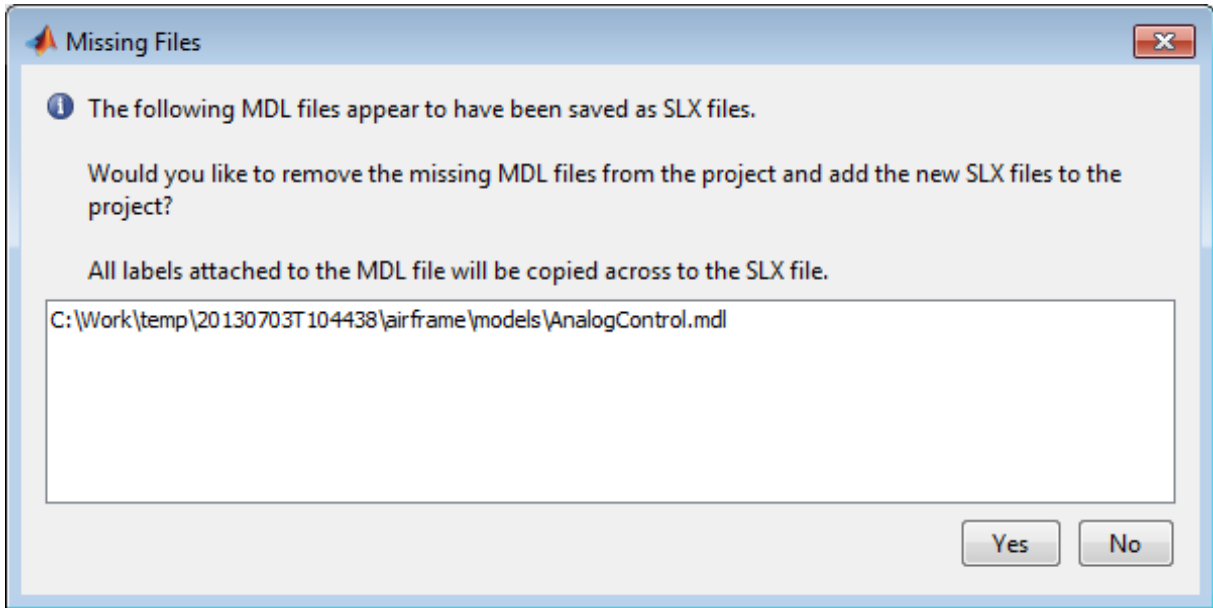


- 6 To resolve these issues, click **Check Project** on the toolbar to run the project integrity checks. The checks look for MDL files converted to SLX, and offer automatic fixes if that check fails.
- 7 Click the **Fix** button to view recommended actions and decide whether to make the changes.

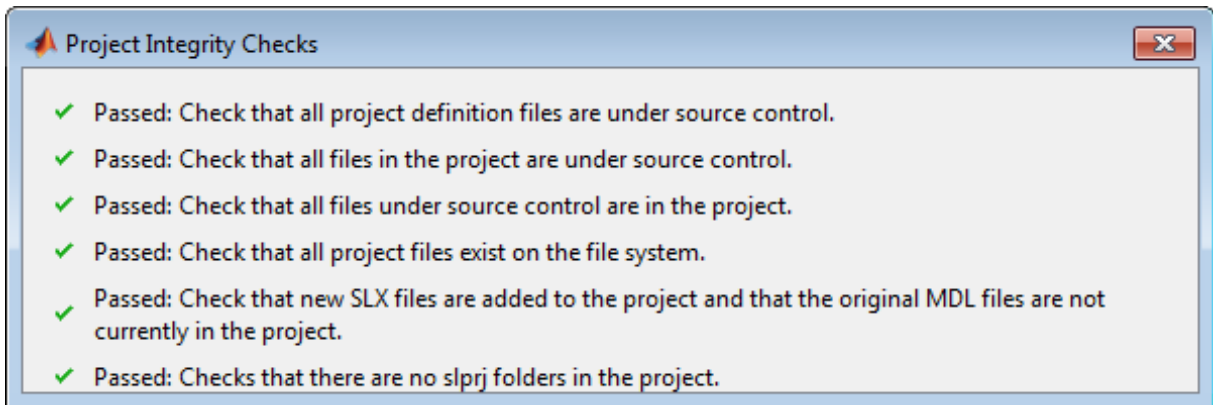


When you click **Fix**, the Missing Files dialog box offers to remove the missing MDL file from the project and add the new SLX file to the project.

- 8 Click **Yes** to perform the fix.

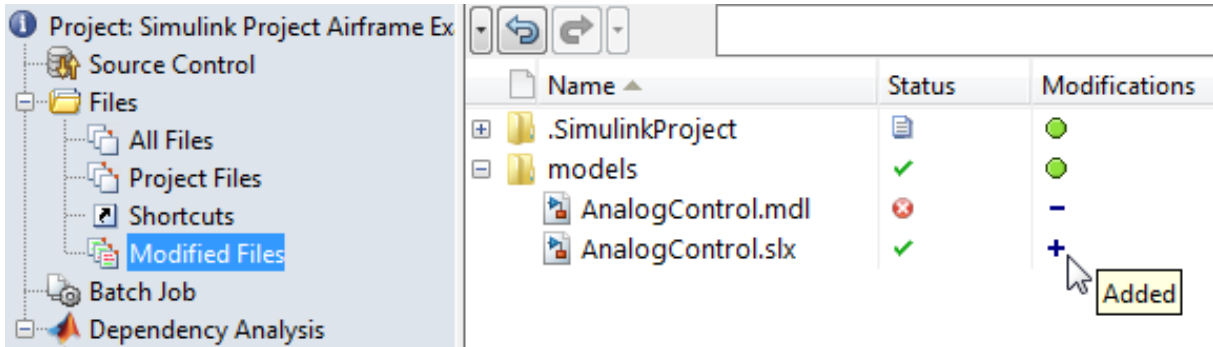


Project checks rerun after you click **Yes** to perform the fix.




Close the Project Integrity Checks dialog box.

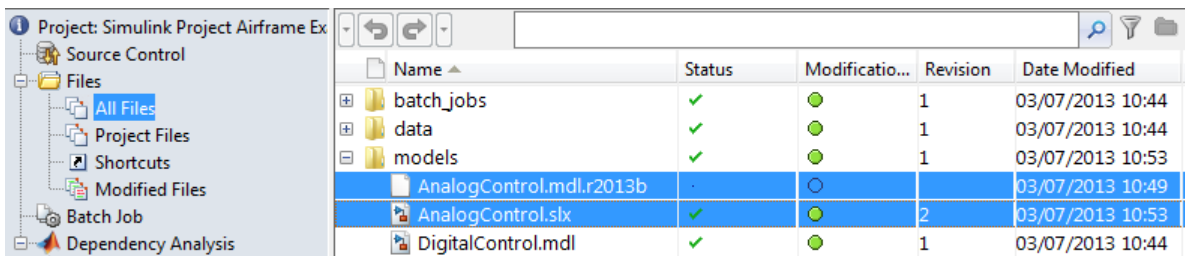
- 9 Select the Modified Files view. Expand the `models` folder and check the Modifications column to see that the newly created SLX file has been added to the project, and the original MDL file is scheduled for removal.



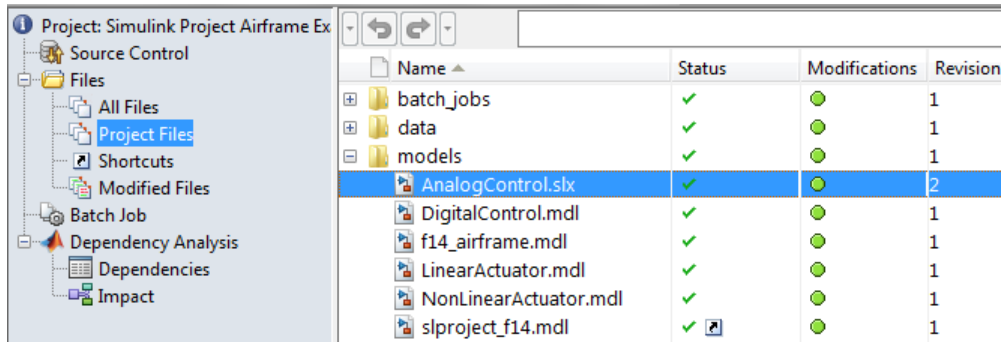
- 10 Click **Commit Modified Files**. Enter a comment for your submission in the dialog, for example, Convert to SLX, and click **Submit**.

Verify Changes After Upgrade to SLX

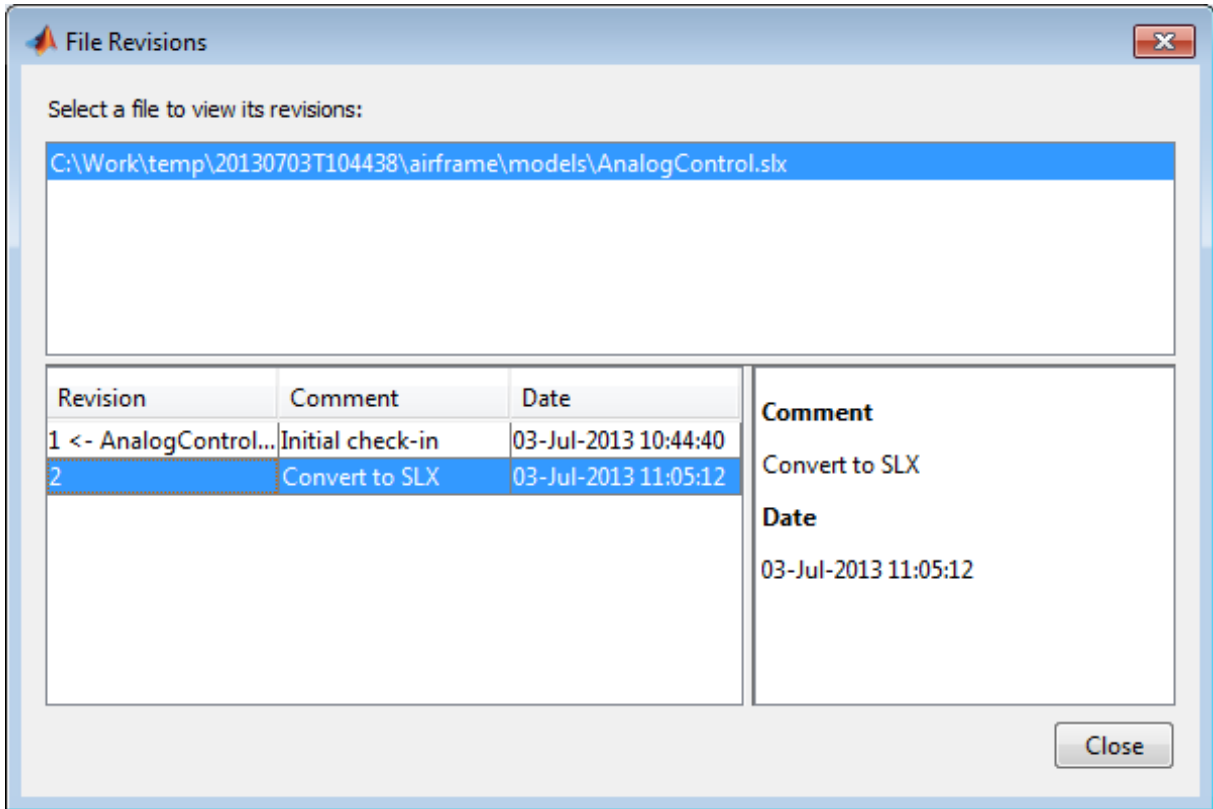
- 1 In the All Files view, check that the backup file AnalogControl1.mdl.r2012b is still present, along with the new SLX file. Click the Actions button  to customize the columns to show, such as **Date Modified**.



- 2 In the Project Files view, check that only the new SLX file is included in the project, and the backup file is not included in the project.



- 3 Right-click the model file AnalogControl.slx and select **Show Revisions**.
- 4 In the File Revisions dialog box, verify that the previous revision is AnalogControl.mdl. The revision history of the previous model file is preserved with the new SLX file.



For an example showing commands to find and upgrade all model files in the project to SLX, see [Converting from MDL to SLX Model File Format in a Simulink Project](#).

Archive Projects in Zip Files

To package and share project files, you can export all project files to a zip file. For example, you can share a zipped project with people who do not have access to the connected source control tool.

- 1** With a project loaded, on the **Simulink Project** tab, select **Archive**.
- 2** Use the file browser and save the file. By default, the file *myprojectname.zip* is created in the current working folder.

The archive command exports a complete project.

See also “Create a New Project from an Archived Project” on page 13-27.

To share projects with other users, it can be useful to create a model dependencies manifest report showing required toolboxes for your top model. You can save and share that information by adding the manifest report to the project. See “Generate Manifests” on page 13-160.

Automate Project Management Tasks

In this section...

“Manipulate a Simulink Project at the Command Line” on page 13-81

“Get Simulink Project at the Command Line” on page 13-81

“Find Project Commands” on page 13-82

“Examine Project Files” on page 13-82

“Label a Project File” on page 13-84

“Attach Data to a Label” on page 13-84

“Create New Category of Project Labels” on page 13-85

“Define a New Label” on page 13-86

“Attach New Labels and Label Data to a File” on page 13-86

“Query Shortcuts” on page 13-87

“Close Project” on page 13-89

“More Project API Examples” on page 13-89

Manipulate a Simulink Project at the Command Line

You can automate some Simulink project tasks using scripts. You can manipulate project files and labels and use commands for scripting operations on project files. This example shows how to use commands to automate Simulink project tasks with files and labels.

To automate *startup* and *shutdown* tasks, see “Automate Startup Tasks with Shortcuts” on page 13-36.

Get Simulink Project at the Command Line

This example shows how to open the Airframe project and use `simulinkproject` to get a project object to manipulate the project at the command line. You must open a project in the Simulink Project to perform command-line operations on it.

```
sldemo_slproject_airframe
```

```
proj = simulinkproject

proj =

    ProjectManager with properties:

        Name: 'Simulink Project Airframe Example'
        Categories: [1x2 slproject.Category]
        Shortcuts: [1x6 slproject.Shortcut]
        Files: [1x24 slproject.ProjectFile]
        RootFolder: [1x61 char]
```

Find Project Commands

This example shows how to find out what you can do with your project.

Examine project commands.

```
methods(proj)
```

```
Methods for class slproject.ProjectManager:
```

```
addFile createCategory findCategory
isLoading removeCategory close
export findFile reload removeFile
```

Examine Project Files

After you get a project object, you can examine project properties.

- 1 Examine the project files.

```
files = proj.Files
```

```
files =
```

```
1x24 ProjectFile array with properties:
```

```
Path
Labels
```


- 2** Use indexing to access files in this list. The following command gets file number 8. Each file has two properties describing its path and attached labels.

```
proj.Files(8)
```

```
ans =
```

```
ProjectFile with properties:
```

```
Path: 'C:\Temp\project1\airframe\models\AnalogControl.mdl'  
Labels: [1x1 slproject.Label]
```

- 3** Examine the labels of the eighth file.

```
proj.Files(8).Labels
```

```
ans =
```

```
Label with properties:
```

```
File: 'C:\Temp\project1\airframe\models\AnalogControl.mdl'  
Data: []  
DataType: 'none'  
Name: 'Design'  
CategoryName: 'Classification'
```

- 4** Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
Path: [1x86 char]  
Labels: [1x1 slproject.Label]
```

- 5** Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class slproject.ProjectFile:
```

```
addLabel  
findLabel  
removeLabel
```

Label a Project File

- 1 Attach a label to the retrieved file, `myfile`.

```
addLabel(myfile, 'Review', 'To Review')
```

This label appears next to the file in the Simulink Project.

Attach Data to a Label

- 1 Get a particular label and attach data (the string `Code reviewer: Bob`) to it.

```
label = findLabel(myfile, 'Review', 'To Review');  
label.Data = 'Code reviewer: Bob'
```

```
label =
```

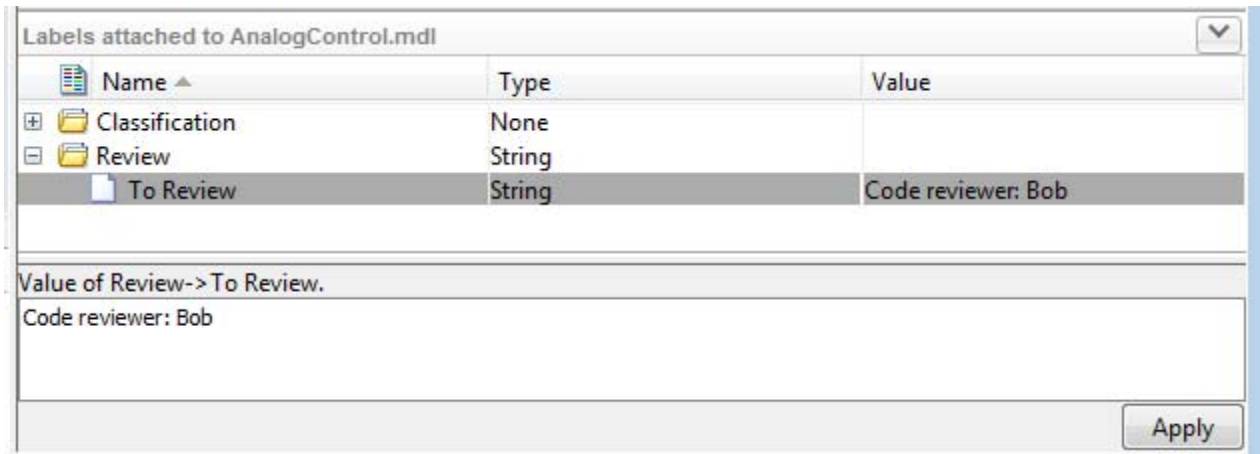
```
Label with properties:
```

```
File: [1x86 char]  
Data: 'Code reviewer: Bob'  
DataType: 'char'  
Name: 'To Review'  
CategoryName: 'Review'
```

Alternatively, you can specify a variable for the label data: `mydata = label.Data`.

- 2 Select the file `AnalogControl` in the Simulink Project. Expand the lower pane, **Labels attached to AnalogControl.mdl**, and select the label `To Review`.

The label data Code reviewer: Bob appears in the **Value** pane. You can edit label data in this window and click **Apply** to save label data changes.



Create New Category of Project Labels

- 1 Create a new category of labels, of type double.

```
createCategory(proj, 'Engineers', 'double')
```

```
category =
```

```
Category with properties:
```

```
    Name: 'Engineers'
    DataType: 'double'
    LabelDefinitions: [ ]
```

- 2 Find out what you can do with the new category.

```
category = findCategory(proj, 'Engineers')
methods(category)
```

```
Methods for class slproject.Category:
```

```
createLabel      findLabel
```

```
removeLabel
```

Define a New Label

- 1 Define a new label in the new category.

```
createLabel(category, 'Sam')
```

- 2 Get a label definition.

```
ld = findLabel(category, 'Sam')
```

```
ld =
```

```
LabelDefinition with properties:
```

```
    Name: 'Sam'  
  CategoryName: 'Engineers'
```

Attach New Labels and Label Data to a File

- 1 Attach your new label to a file.

```
myfile = proj.Files(8)  
addLabel(myfile, 'Engineers', 'Sam')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
    Path: [1x72 char]  
  Labels: [1x2 slproject.Label]  
ans =
```

```
Label with properties:
```

```
    File: [1x86 char]  
    Data: []  
  DataType: 'double'
```

```

        Name: 'Sam'
    CategoryName: 'Engineers'

```

- 2 Attach your new label to a file and assign data 2 to the label.

```
addLabel(myfile, 'Engineers', 'Sam', 2)
```

```
ans =
```

```
Label with properties:
```

```

        File: [1x86 char]
        Data: 2
    DataType: 'double'
        Name: 'Sam'
    CategoryName: 'Engineers'

```

Query Shortcuts

- 1 Examine the project's Shortcuts property.

```
shortcuts = proj.Shortcuts
```

```
shortcuts =
```

```
1x6 Shortcut array with properties:
```

```

    File
    RunAtStartup
    RunAtShutdown

```

- 2 Examine the second shortcut in the array.

```
shortcuts(2)
```

```
ans =
```

```
Shortcut with properties:
```

```

        File: 'C:\Temp\airframe\data\f14_digital_data.mat'
    RunAtStartup: 1

```

```
RunAtShutdown: 0
```

The RunAtStartup property is set to 1, so this shortcut file is set to run at project startup. At the command line you can view but not change the RunAtStartup and RunAtShutdown properties. To set these properties, use the Shortcuts node in the Simulink Project.

- 3** Get the file path of the second shortcut.

```
shortcuts(2).File
```

```
ans =
```

```
C:\Temp\airframe\data\f14_digital_data.mat'
```

- 4** Examine all the files in the shortcuts cell array.

```
{shortcuts.File}'
```

```
ans =
```

```
'C:\Temp\airframe\data\buses.mat'
```

```
'C:\Temp\airframe\data\f14_digital_data.mat'
```

```
'C:\Temp\airframe\models\slproject_f14.mdl'
```

```
'C:\Temp\airframe\utilities\clean_up_project.m'
```

```
'C:\Temp\airframe\utilities\rebuild_s_functions.m'
```

```
'C:\Temp\airframe\utilities\set_up_project.m'
```

- 5** Create a logical array that shows the shortcuts set to run at startup.

```
idx = [shortcuts.RunAtStartup]
```

```
idx =
```

```
1 1 0 0 0 1
```

- 6** Use the logical array to get only the startup shortcuts.

```
startupshortcuts = shortcuts(idx)
```

```
startupshortcuts =
```

```
1x3 Shortcut array with properties:
```

```
File  
RunAtStartup  
RunAtShutdown
```

- 7** Get the path of the third startup shortcut by accessing the File property.

```
startupshortcuts(3).File
```

```
ans =
```

```
C:\Temp\airframe\utilities\set_up_project.m
```

Close Project

1

```
close(proj)
```

Closing the project at the command line is the same as closing the project using the Simulink Project tool. For example, shutdown scripts run.

More Project API Examples

For more examples, see:

- Managing Labels Using the Simulink Project's API
- Examples on specific function pages, e.g., `addLabel`, `createLabel`, `createCategory`.

About Source Control with Projects

You can use Simulink Project to work with source control. You can perform operations such as update, commit, merge changes, and view revision history directly from the Simulink Project environment.

Simulink Project has interfaces to:

- Subversion® (SVN) — See “Set Up SVN Source Control” on page 13-101.
- Local Version Control — Project examples use a lightweight local version control tool so you can try source control operations. The local version control tool is designed only for the examples. See “Try Simulink Project Tools with the Airframe Project” on page 13-7.
- Software Development Kit (SDK) — You can use the SDK to integrate Simulink Projects with third-party source control tools. See “Write a Source Control Adapter with the SDK” on page 13-109.

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 13-92.
- Retrieve files from an existing repository and create a new project. See “Retrieve a Working Copy of a Project from Source Control” on page 13-110.
- Create a new project in a folder already under source control and click **Detect**. See “Create a New Project to Manage Existing Files” on page 13-21.

When your project is under source control, you can:

- “Retrieve a Working Copy of a Project from Source Control” on page 13-110
- “Review Changes” on page 13-130
- “Commit Modified Files to Source Control” on page 13-134

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 automerger. Corruption can occur whether you use the source control tools outside of Simulink or if you try submitting files from Simulink 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, .mdl, .slx, .p, MEX-files, .xlsx, .jpg, .pdf, .docx, etc.

For instructions with SVN, see “Register Model Files with Subversion” on page 13-104. You must register model files if you use SVN, including the Built-In SVN Integration provided by Simulink Project.

Add a Project to Source Control

To find out what you can do with source control in Simulink Project, see “About Source Control with Projects” on page 13-90.

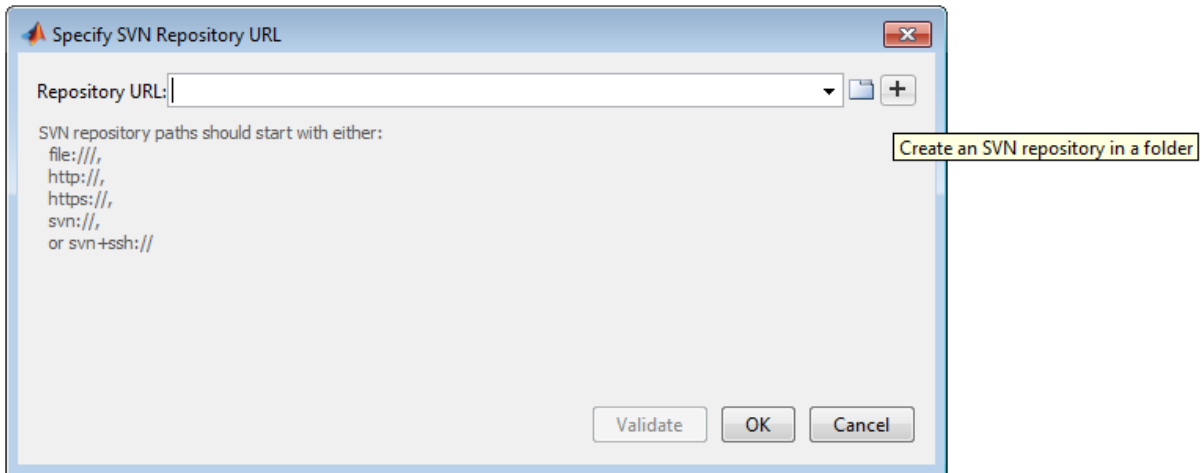
Caution Using a network folder with SVN is slow and unreliable. Before you start, check your sandbox folder is on a local hard disk.



To add a project to source control:

- 1** Select the Source Control node in the project tree.
- 2** Under **Available Source Control Integrations**, click **Add project to source control**.
- 3** In the Add to Source Control dialog box, leave the default **Source control integration** selected to use **Built-In SVN Integration** provided by Simulink Project.

If you want to use a different version of SVN, see “Set Up SVN Source Control” on page 13-101.

- 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.



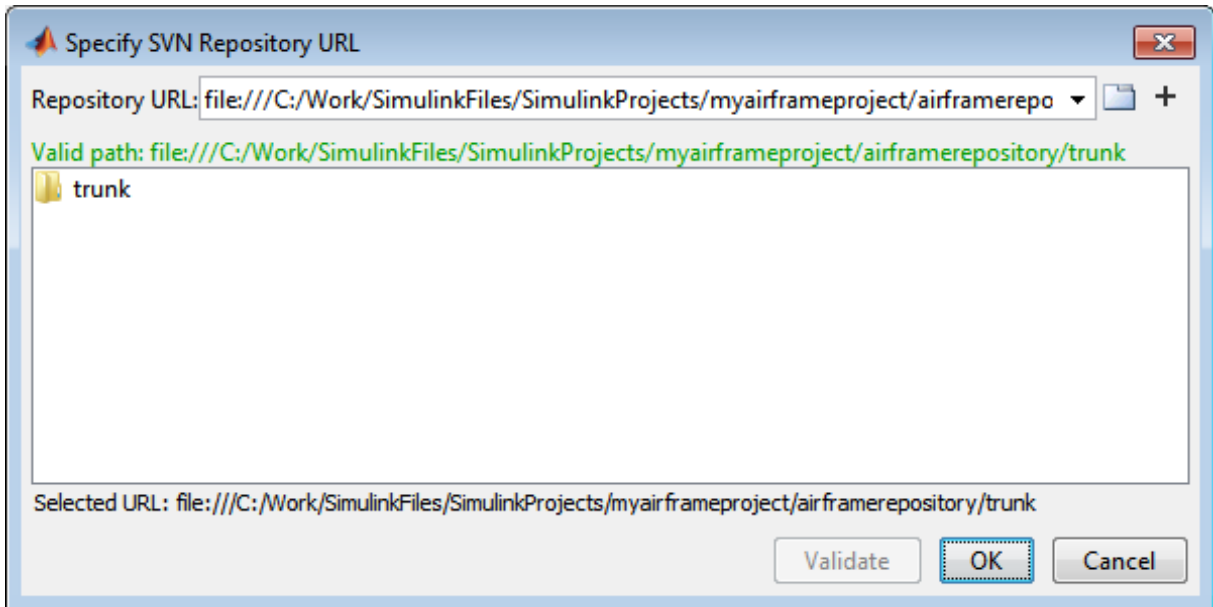
- To specify an existing repository, click Generate URL from folder  to browse for your repository, paste a URL into the box, or use the list to select a recent repository.
- To create a new repository, click Create an SVN repository . Using the file browser, create a folder where you want to create the new repository and click **Select Folder**. Do not place the new repository inside the existing project folder.

Simulink 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 URL** box, and the project automatically selects the trunk folder.

Caution Specifying `file://` URLs and creating new repositories are for single users only. For multiple users, see “Share a Subversion Repository” on page 13-107.

- 6 Click **Validate** to check the path to the selected repository. If you have problems, check the dialog messages for required path styles.

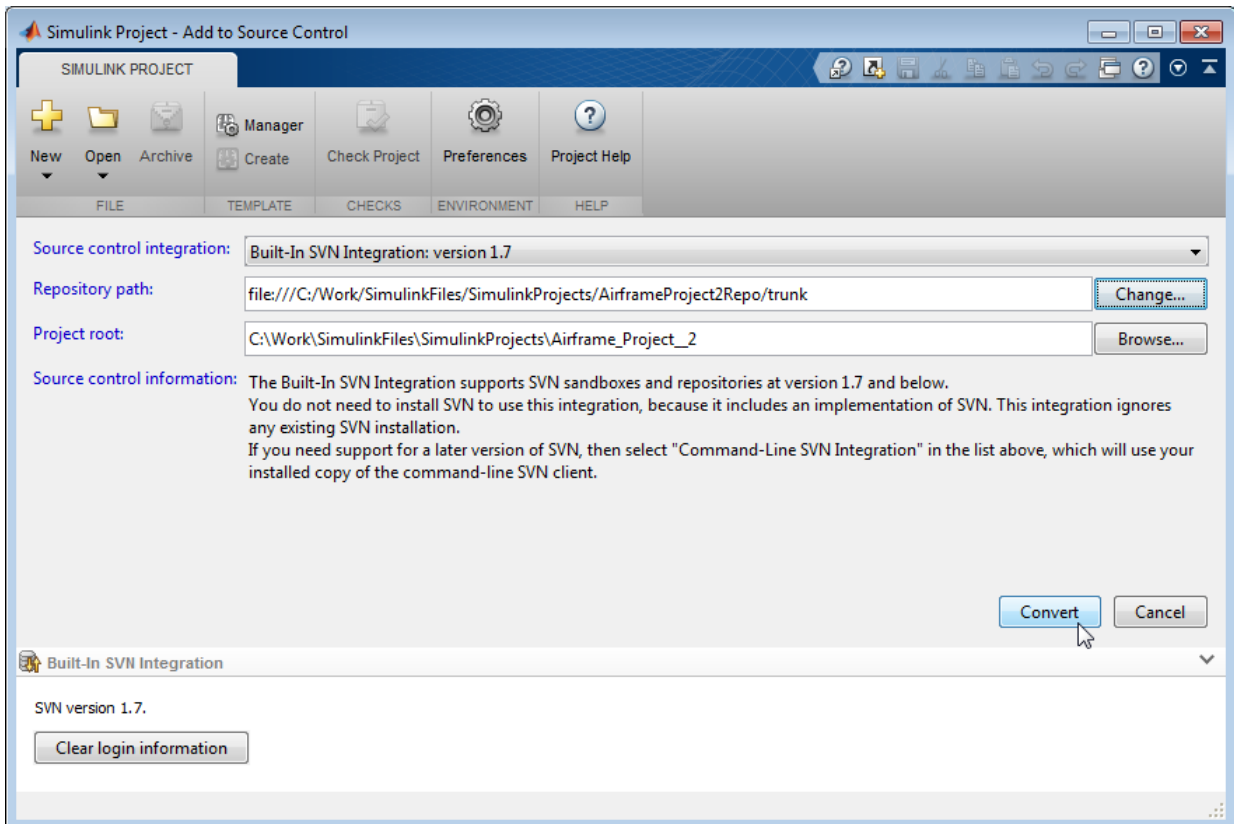
When the path is valid, you can browse the repository folders. Select the trunk folder if needed, and verify the selected URL at the bottom of the dialog box.



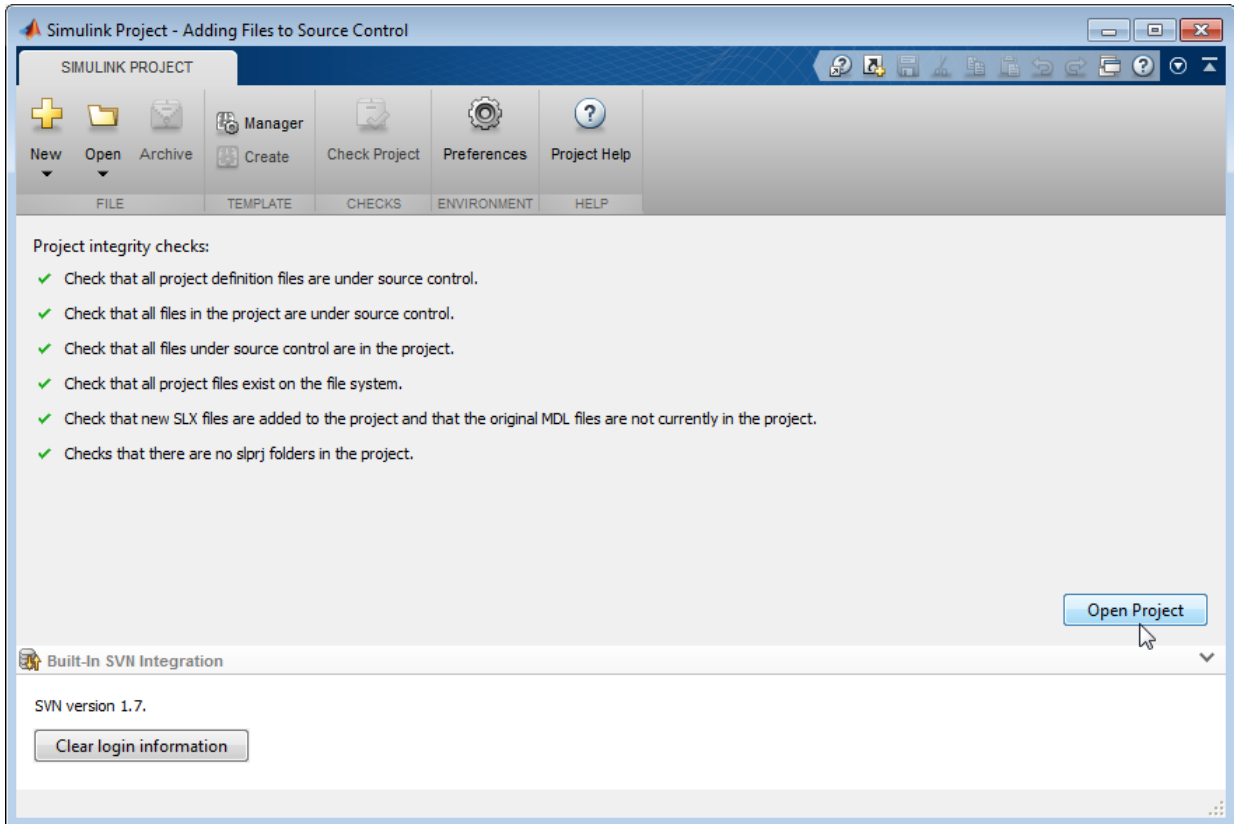
- 7 Click **OK** to return to the Add to Source Control dialog box.

If your repository has a file URL, you see a dialog box to warn that file URLs are for single users, and provide a link to information on safely sharing repositories. Click **OK** to continue.

- 8 Click **Convert** to finish adding the project to source control.



The project runs integrity checks.



9 Click **Open Project** to return to your project.

The Source Control node displays details of the current source control tool, the adapter interfacing to source control in the current project, and the repository location.

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 13-91.

Related Examples

- “Register Model Files with Source Control Tools” on page 13-91
- “Add Files to a New Repository” on page 13-98
- “Check Out Files” on page 13-125
- “Work with Project Files” on page 13-60
- “View Modified Files” on page 13-128
- “Commit Modified Files to Source Control” on page 13-134

Concepts

- “About Source Control with Projects” on page 13-90

Add Files to a New Repository

If you created a new repository as described in “Add a Project to Source Control” on page 13-92, then you need to add your files to it.

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 13-91.

- 1 Select the Modified Files view.
- 2 Click **Check Project** on the Simulink Project tab to find all files not yet in source control.
- 3 Click **Commit Modified Files** to commit the first version of your files to the new repository.
- 4 In the dialog box, enter a comment if you want, and click **Submit**.

Related Examples

- “Register Model Files with Source Control Tools” on page 13-91
- “Check Out Files” on page 13-125
- “Work with Project Files” on page 13-60
- “View Modified Files” on page 13-128
- “Commit Modified Files to Source Control” on page 13-134

Concepts

- “About Source Control with Projects” on page 13-90

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** In the Source Control node, under **Available Source Control Integrations**, select **No source control integration**.
- 2** Click **Reload**.

Note Source control tools create files in the project folders (e.g., 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.

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** On the **Simulink Project** tab, click **Archive** to save a Zip file containing the project without any source control information.
- 3** On the **Simulink Project** tab, select **New > From Archive** to create a new project from the archived project.
- 4** In the Source Control view, click **Add project to source control** to select a new source control. For details, see “Add a Project to Source Control” on page 13-92.

Tip To avoid accidentally committing changes to the previous source control, delete the original sandbox.

Set Up SVN Source Control

In this section...

“Set Up SVN Integration Provided with Simulink Project” on page 13-101

“Set Up SVN Integration for SVN Version Already Installed” on page 13-102

“Set Up SVN Integration for SVN Version Not Yet Provided with Simulink Project” on page 13-103

“Register Model Files with Subversion” on page 13-104

“Share a Subversion Repository” on page 13-107

Set Up SVN Integration Provided with Simulink Project

Simulink Project provides Built-In SVN Integration for use with Subversion (SVN) sandboxes and repositories at version 1.7. 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 Built-In SVN Integration supports secure logins.

To use the version of SVN provided with Simulink Project, when you add a project to source control or retrieve from source control, select **Built-In SVN Integration** in the **Source control integration** list. For instructions, see

- “Add a Project to Source Control” on page 13-92, or
- “Retrieve a Working Copy of a Project from Source Control” on page 13-110.

When you create a new sandbox using the Simulink Project Built-In SVN Integration, the new sandbox uses the latest version of SVN provided by Simulink Project.

When your project is under source control, you can use these project features:

- “Retrieve a Working Copy of a Project from Source Control” on page 13-110
- “Review Changes” on page 13-130
- “Commit Modified Files to Source Control” on page 13-134

Caution Before using source control, you must register model files with the source control tools to avoid corrupting models. See “Register Model Files with Subversion” on page 13-104.

You can checkout from a branch, but the project Built-In SVN Integration 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 an XML comparison report when you perform a diff on model files. See “Compare XML from Models Managed with Subversion”.

Tip You can check for updated source control integration downloads on the Simulink Projects Web page:
<http://www.mathworks.com/products/simulink/simulink-projects/>

Set Up SVN Integration for SVN Version Already Installed

If you want to use Simulink Project with an earlier SVN version you already have installed, create a new project in a folder already under SVN source control and click **Detect** in the New Project dialog box.

For example:

- 1 Create the sandbox using TortoiseSVN from Windows Explorer.
- 2 Use Simulink Project to create a new project in that folder, then click **Detect** to discover 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 13-104.

Set Up SVN Integration for SVN Version Not Yet Provided with Simulink Project

If you need to use a later version of SVN than 1.7, 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 1.7. Otherwise, use Built-In SVN Integration 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. However, 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, e.g., both clients are SVN 1.7.

You can find Subversion clients on this Web page:

<http://subversion.apache.org/packages.html>

- 2 In Simulink 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.

Register Model Files with Subversion

You must register model files if you use SVN, including the Built-In SVN Integration provided by Simulink Project.

If you do not register your model file extension as binary, Subversion 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 Subversion, follow these steps to 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 13-104.
- 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 13-105.

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]
*.mdl = svn:mime-type=application/octet-stream
*.mat = svn:mime-type=application/octet-stream
*.slx = svn:mime-type= application/octet-stream
```

Note 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 `.mat`, `.mdl`, `.slx`, `.p`, MEX-files (`.mexa64`, `.mexmaci64`, `.mexw32`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the config file for each file type you need.

2 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
- In `~/ .subversion` on Linux or Mac OS X.

After creating the SVN config file, SVN treats new model files as binary. If you created a new repository before creating the SVN config file, next see “Add Files to a New Repository” on page 13-98.

If you already have models in repositories, see “Register Models Already in Repositories” on page 13-106.

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 to verify or add the following lines.

2 Locate the `[miscellany]` section, and verify the following line enables `auto-props`:

```
enable-auto-props = yes
```

3 Locate the `[auto-props]` section and add the following lines at the end:

```
*.mdl = svn:mime-type= application/octet-stream  
*.mat = svn:mime-type=application/octet-stream  
*.slx = svn:mime-type= application/octet-stream
```

These lines prevent SVN from adding annotations to Simulink files on conflict and from automerging.

Note 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 `.mat`, `.mdl`, `.slxp`, `.p`, MEX-files (`.mexa64`, `.mexmaci64`, `.mexw32`, `.mexw64`), `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc. Add a line to the config file for each file type you need.

4 Save the config file.

After you create or update the SVN config file, SVN treats new model files as binary.

If you created a new repository before updating the SVN config file, next see “Add Files to a New Repository” on page 13-98.

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 identify models as binary.

To manually identify 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 SVN Integration for SVN Version Not Yet Provided with Simulink Project” on page 13-103.

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 Simulink Projects 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

To enable tagging within projects when using SVN, create your repository with the standard tags, trunk, and branch folders, and check out files from trunk. The Subversion project recommends this structure. See the Web page:

<http://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html>

If you use Simulink Project to create an SVN repository, it creates the standard repository structure.

After you create a repository with this structure, you can click **Tag** in the Project Source Control node to add tags to all your project files. See “Tag and Retrieve Versions of Project Files” on page 13-115.

Concepts

- “About Source Control with Projects” on page 13-90

Write a Source Control Adapter with the SDK

Simulink provides a Software Development Kit (SDK) that you can use to integrate Simulink Projects with third-party source control tools.

The SDK provides instructions for writing an adapter 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 adapter. Build and test your own interfaces using the example as a guide. Then you can use your source control adapter with Simulink Projects.

- 1 Extract the contents of the SDK.

```
run(fullfile(matlabroot, 'toolbox', 'simulink', 'simulink', 'slproject', ...  
    'adapterSDK', 'extractSDK'))
```

Select a folder to place the `cm_adapter_SDK` folder and files in, and click **OK**.

- 2 Locate the new folder `cm_adapter_SDK`, and open the file `cm_adapter_SDK_guide.pdf` for instructions.

After you write a source control adapter, see “Add a Project to Source Control” on page 13-92.

Concepts

- “About Source Control with Projects” on page 13-90

Retrieve a Working Copy of a Project from Source Control

To create a new local copy of a project by retrieving files from source control:

- 1** From MATLAB, on the **Home** tab, in the **File** section, select **New > Simulink Project > From Source Control**.


Alternatively, on the **Simulink Project** tab, in the **File** section, select **New > From Source Control**.

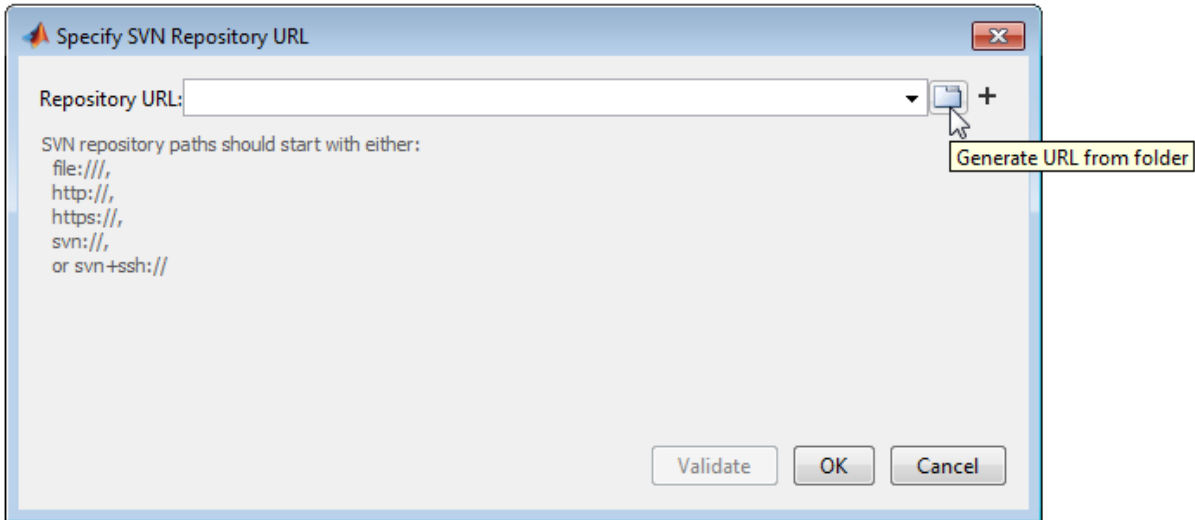
- 2** In the Project Retriever dialog box, select the source control interface from the **Source control integration** list.

Leave the default **Built-In SVN Integration** to use the SVN integration with your project. For more information on SVN support and choosing versions, see “Set Up SVN Source Control” on page 13-101.

- 3** If you know your repository location, you can paste it into the **Repository Path** box. Otherwise click **Change** to browse for and validate the repository path to retrieve files from.

The Specify Repository URL dialog box opens.

- 4** Specify the repository URL using the Generate URL from folder button , the list of recent repositories, or by entering or pasting a URL in the box.



Caution Use `file://` URLs only for single user repositories. For more information, see “Share a Subversion Repository” on page 13-107.

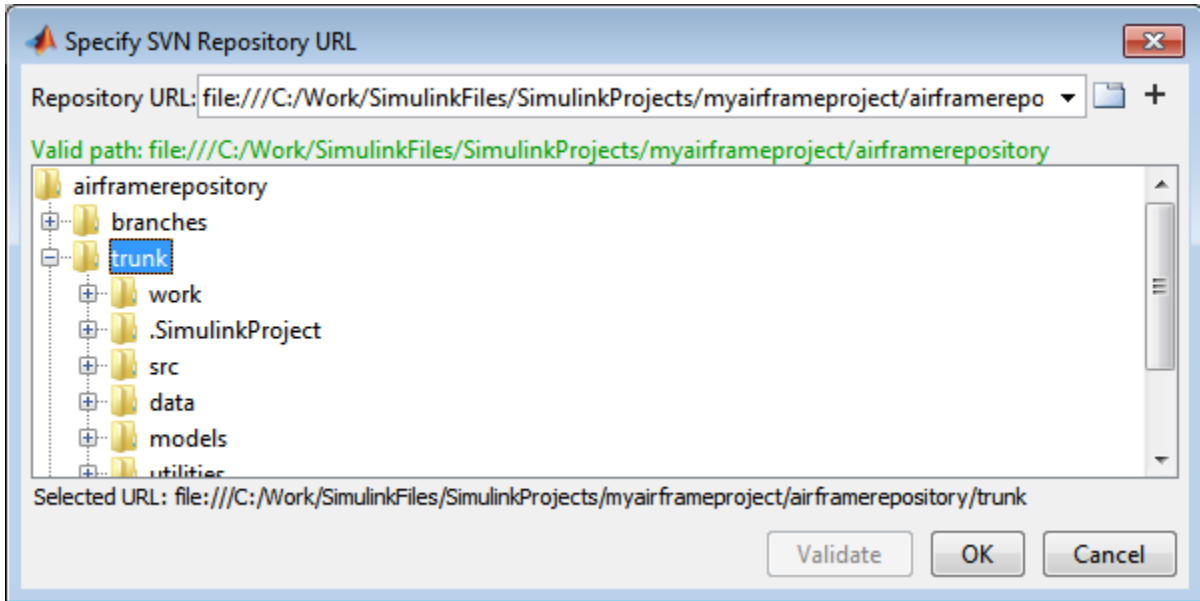
5 Click **Validate** to check the repository path.

If the path is invalid, check the URL against your source control repository browser.

You might see an authentication dialog for your repository. Enter login information to continue.

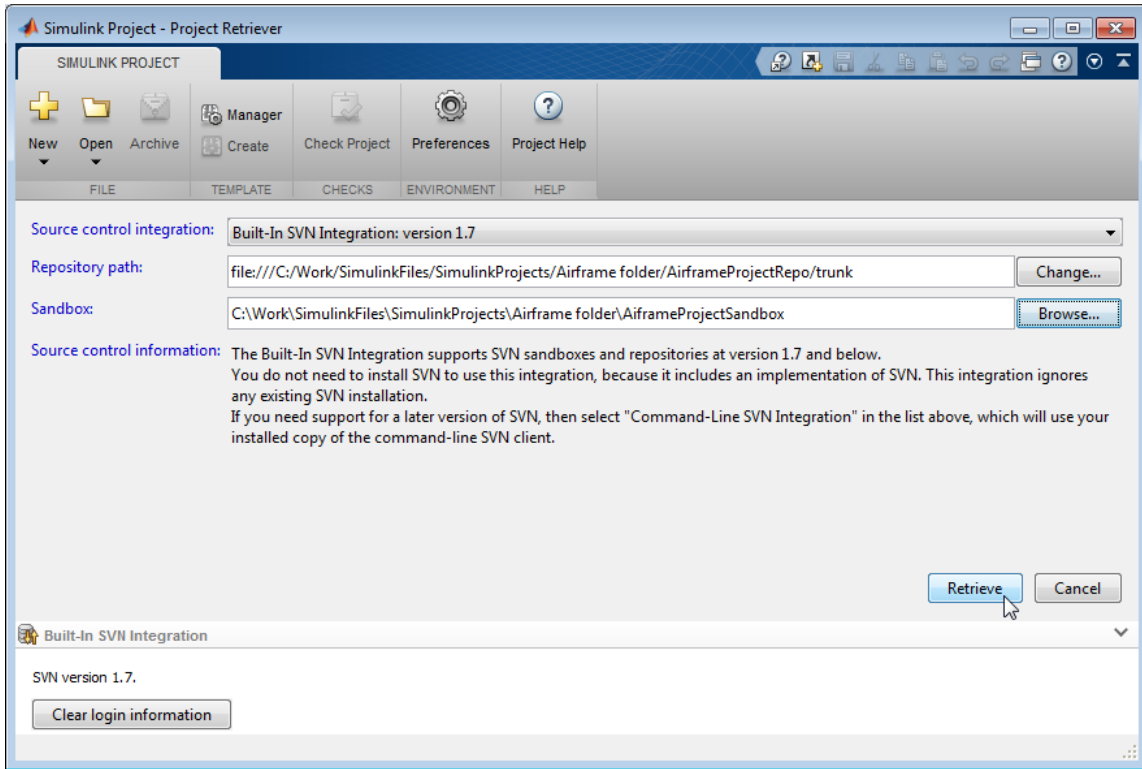
6 If necessary, select a deeper folder in the repository tree. You might want to check out from `trunk` or from a branch folder under `tags`, if your repository contains tagged versions of files. The example shows `trunk` selected, and the **Selected URL** displays at the bottom of the dialog box.

The retriever uses the **Selected URL** when you click **OK**.



- 7 When you have finished specifying the URL path you want to retrieve, click **OK**.
- 8 In the Project Retriever, select the sandbox folder where you want to put the retrieved files for your new project, and click **Retrieve**.

Caution Use local sandbox folders. Avoid using a network folder with SVN because this is slow.



The **Built-In SVN Integration** pane displays messages as the project retrieves the files from source control.

If your repository already contains a Simulink project, the project is ready when the tool finishes retrieving files to your selected sandbox folder.

- 9 If your sandbox doesn't yet contain a Simulink project, then a dialog box prompts you to check whether you want to create a project in the folder. Click **Yes** to continue creating the project.

The new project controls appear.

- a In the new project controls, enter a project name.
- b Click **Create** to finish creating the new project in your new sandbox.

Simulink Project displays the empty Project Files list for the chosen project root. The project does not yet contain any files. For next steps, see “Add Files to the Project” on page 13-25.

Note To update an existing project sandbox from source control, see “Update Revisions of Project Files” on page 13-122.

Related Examples

- “Check Out Files” on page 13-125
- “Work with Project Files” on page 13-60
- “Tag and Retrieve Versions of Project Files” on page 13-115
- “Refresh Status of Project Files” on page 13-117
- “Check for Modifications” on page 13-121
- “Update Revisions of Project Files” on page 13-122
- “View Modified Files” on page 13-128
- “Commit Modified Files to Source Control” on page 13-134

Concepts

- “About Source Control with Projects” on page 13-90

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 13-107.

- 1 In the Source Control node, click **Tag** under Project Source Control Actions.
- 2 Specify the tag text and click Submit. 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 **Simulink Project** tab, select **New > From Source Control**.
- 2 Click **Change** to select the Repository Path that you want to retrieve files from.

The Specify Repository URL dialog box opens.

- a Select a recent repository from the **Repository URL** list, or click the Generate URL from folder 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. Verify there is a **.SimulinkProject** folder under the chosen tag subfolder.
- d Click **OK** to continue and return to the Project Retriever.

3 Select the sandbox folder to receive the tagged files. You must use an empty sandbox folder. (If you try to retrieve tagged files into an existing sandbox, an error appears.)

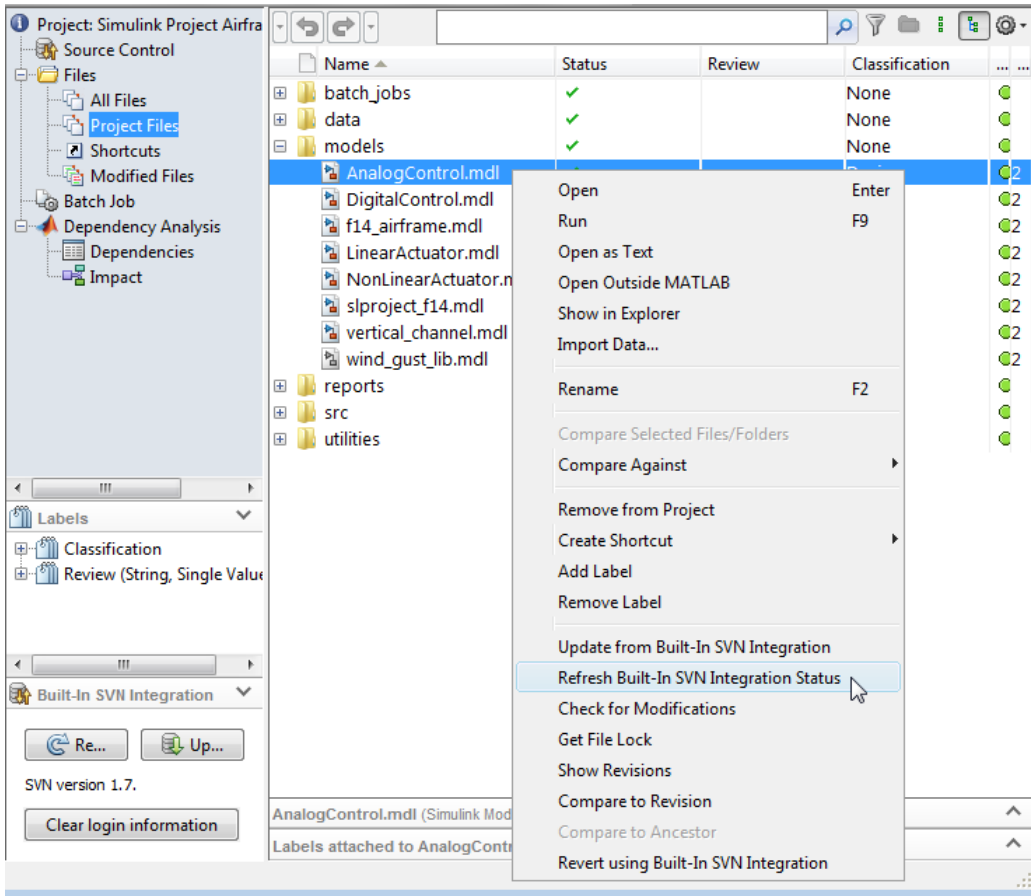
4 Click **Retrieve**.

See also “Add Labels to Files” on page 13-66.

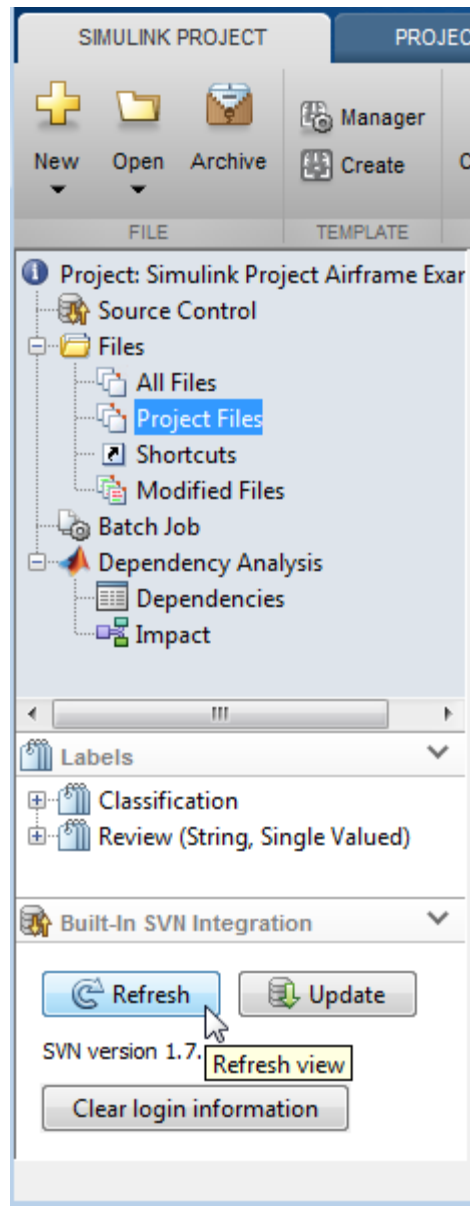
Refresh Status of Project Files

In Simulink Project, to check the status of individual files, right-click files in any view and select the **Refresh** command for the source control system you are using. For example, if you are using SVN, select **Refresh Built-In SVN Integration status**. Refresh queries the local sandbox state and checks for changes made with another tool.

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 13-121 and “Update Revisions of Project Files” on page 13-122.



To check source control status of all project files, click **Refresh** in the source control pane. The source control pane title depends on your source control, for example, **Built-In SVN Integration**. The source control pane reports source control messages, and the buttons in the pane 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 (e.g., **SVN** or **Modifications** column). Hover over the **Modifications** row to see the tooltip showing the source control status of a file, e.g., **Modified (Checked Out)**.

Check for Modifications

To check the status of individual files for modifications, right-click files in Simulink Project and select **Check for Modifications**.

This option contacts the SVN repository to check for external modifications. Simulink 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 Simulink Project displays (not latest) next to the revision number of the local file.

To check for locally modified files, use **Refresh** instead. See “Refresh Status of Project Files” on page 13-117.

To get the latest revisions from the repository, use **Update**. See “Update Revisions of Project Files” on page 13-122.

Update Revisions of Project Files

In Simulink Project, to get the latest revisions of all project files from the source control repository, click the **Update** button in the source control pane.

Use **Update** to get other people's changes from the repository and find out about any conflicts, without losing your local modifications. If you want to back out local changes, use **Revert** instead. See "Revert Local Changes and Release Locks" on page 13-136.

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. See "Register Model Files with Subversion" on page 13-104.

SIMULINK PROJECT PROJECT SHORTCUTS

New Open Archive Manager Create Check Project Preferences Project Help

FILE TEMPLATE CHECKS ENVIRONMENT HELP

Project: Simulink Project Airframe Exam

- Source Control
 - Files
 - All Files
 - Project Files
 - Shortcuts
 - Modified Files
 - Batch Job
 - Dependency Analysis
 - Dependencies
 - Impact

Labels

- Classification
- Review (String, Single Valued)

Built-In SVN Integration

Refresh Update

SVN version 1.7.

Clear login information

Name	Status	Rev
batch_jobs	✓	
data	✓	
models	✓	
reports	✓	
src	✓	
utilities	✓	

Get the latest revisions from Built-In SVN Integration repository

models (Folder)

Labels attached to models

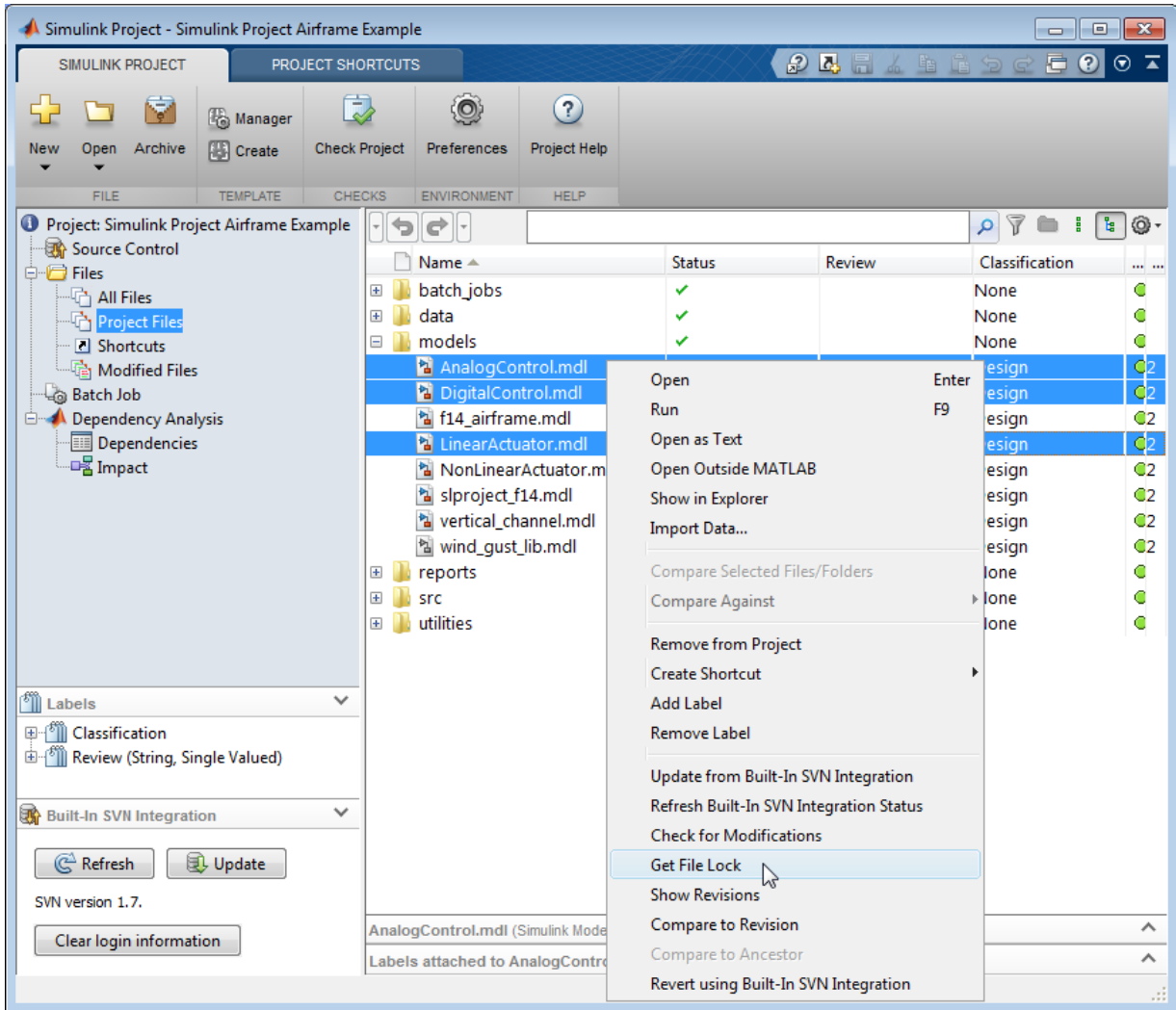
To update selected files, right-click and select the **Update** command for the source control system you are using. For example, if you are using SVN, select **Update from Built-In SVN Integration Repository** to get fresh local copies of the selected files from the repository.

Check Out Files

- 1 In Simulink Project, in any Files view, select the files you want to check out.
- 2 Right-click the selected files and select **Get File Lock**.

The menu wording for source control items is specific to your selected source control. For example, **Get File Lock** is for SVN. This option does not modify the file in your local sandbox.

A lock icon appears in the source control column (e.g., SVN). Other users cannot see the lock icon in their sandboxes, but they cannot get a file lock or check in a change when you have the lock.



Note To get a fresh local copy of the file from the repository, select **Update from Built-In SVN Integration Repository**.

**Related
Examples**

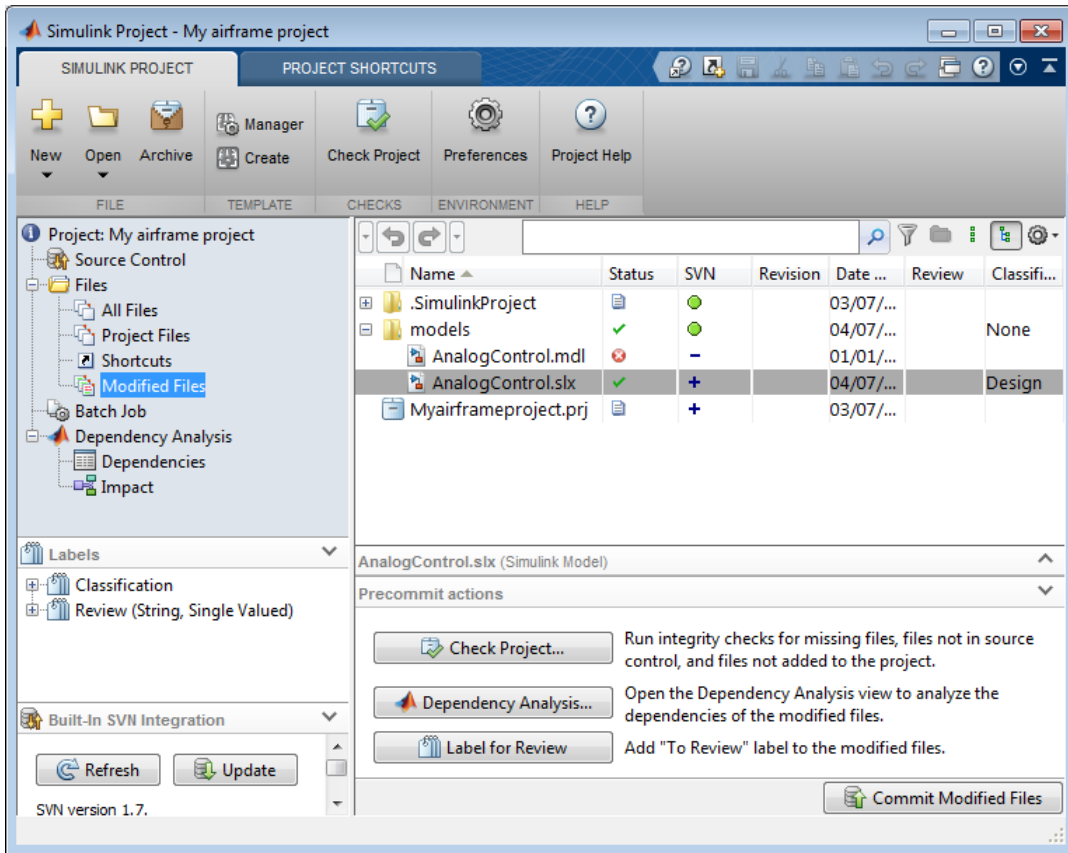
- “Work with Project Files” on page 13-60
- “View Modified Files” on page 13-128
- “Commit Modified Files to Source Control” on page 13-134

Concepts

- “About Source Control with Projects” on page 13-90


View Modified Files

In Simulink Project, select the Modified Files view. The Modified Files node is visible only if you are using source control integration with your project.



If you need to update the modified files list, click **Refresh** in the source control pane.

Use the Modified Files view to review, analyze, label, and commit modified files. 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.

Tip In the Modified Files view, it can be useful to switch to List view by clicking the List button 

Project Definition Files

The files in `.SimulinkProject` are project definition files generated by your changes. The project definition files allow you to add metadata to files without checking them out, 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.

Any changes you make to your project (e.g., to shortcuts, labels, categories, or files in the project) generate changes in the `.SimulinkProject` 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 13-137.

If you want to change project definition file from the type selected when the project was created, see `export`.

Related Examples

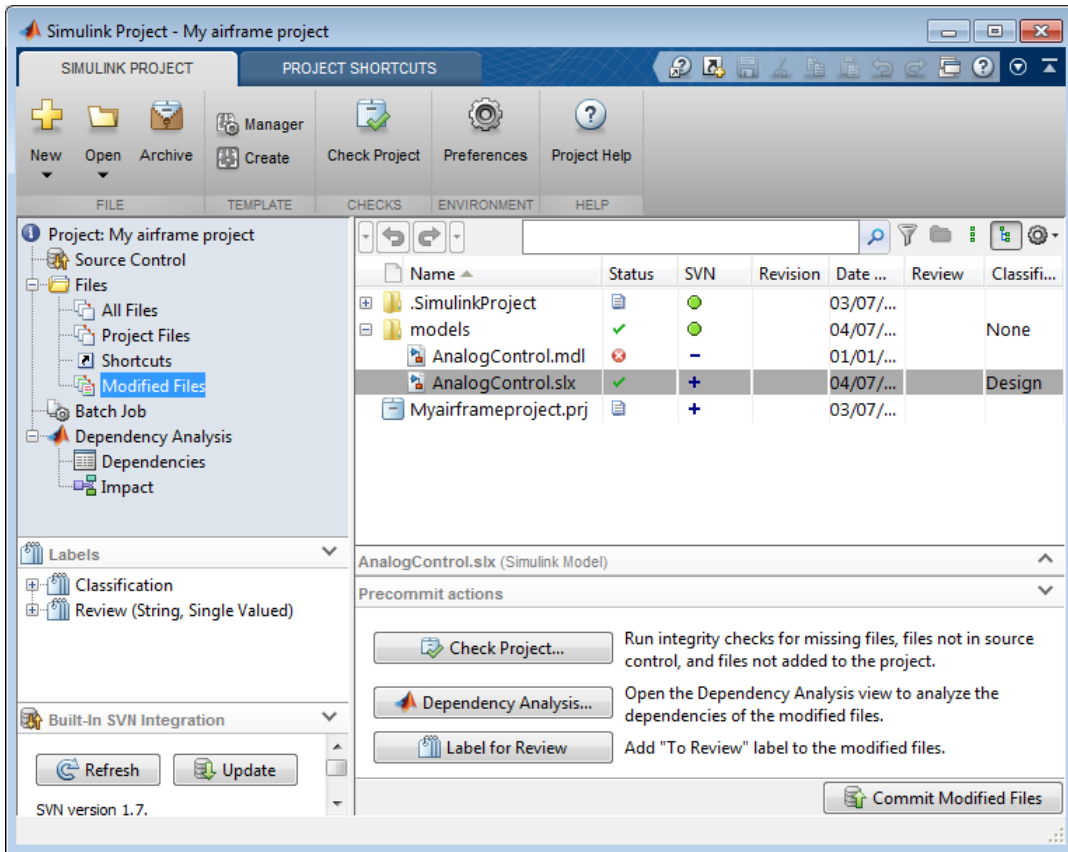
- “Review Changes” on page 13-130
- “Precommit Actions” on page 13-132
- “Refresh Status of Project Files” on page 13-117
- “Check for Modifications” on page 13-121
- “Resolve Conflicts” on page 13-137
- “Revert Local Changes and Release Locks” on page 13-136
- “Commit Modified Files to Source Control” on page 13-134

Concepts

- “About Source Control with Projects” on page 13-90

Review Changes

To view modified files in Simulink Project, select the Modified Files view.



If you need to update the modified files list, click **Refresh** in the source control pane.

To review changes in modified files, right-click selected files in any view in Simulink Project and:

- Select **Show Revisions** to open the File Revisions dialog box and browse the history of a file. You can view SVN information about who previously committed the file, when they committed it, and the log messages.
- Select **Compare to Revision** to open a dialog box where you can select the revision you want to compare. You can select multiple files and select a revision to compare against for each file. Click **Compare** to run comparisons and view reports.
- Select **Compare to Ancestor** to run a comparison with the last checked-out version in the sandbox. The Comparison Tool displays a report.

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, and you have Simulink Report Generator installed, this command runs a Simulink XML comparison.

When reviewing changes, you can merge Simulink models from the Comparison Tool report (requires Simulink Report Generator).

To examine the dependencies of modified files, see “Perform Impact Analysis” on page 13-148.

Related Examples

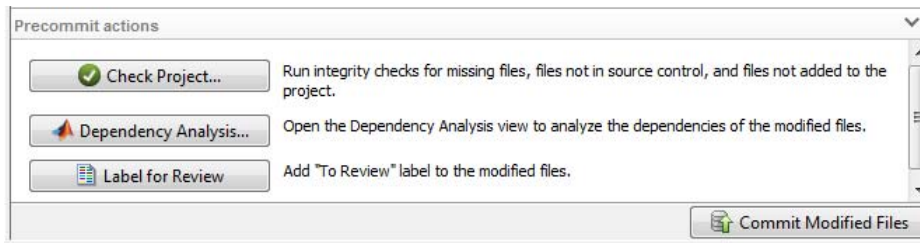
- “Precommit Actions” on page 13-132
- “Commit Modified Files to Source Control” on page 13-134
- “Revert Local Changes and Release Locks” on page 13-136
- “Resolve Conflicts” on page 13-137

Concepts

- “About Source Control with Projects” on page 13-90

Precommit Actions

In Simulink Project, the Precommit actions pane in the Modified Files view contains tools to use before committing your changes to source control.



- 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 “Upgrade Model Files to SLX and Preserve Revision History” on page 13-74.

This command is also in the **Simulink Project** tab (**Check Project**) so you can run the checks from any project view.

For more information on problems the checks can fix, see “Work with Derived Files in Projects” on page 13-139.

- If you want to check for required files, click **Dependency Analysis** to open the Dependency Analysis view. The modified files are automatically selected for analysis. (You can select more or fewer files to analyze by selecting and clearing the check boxes in the Include column.) Click **Analyze** when you are ready to run the analysis.

You can use the dependency tools to analyze the structure of your project. See “What Is Dependency Analysis?” on page 13-140.

- If you want to add the label **To Review** to all files in your changeset, click **Label for Review**.

Note The files in `.SimulinkProject` are project definition files generated by your changes, and these files are not labeled. See “Project Definition Files” on page 13-129.

Related Examples

- “Commit Modified Files to Source Control” on page 13-134

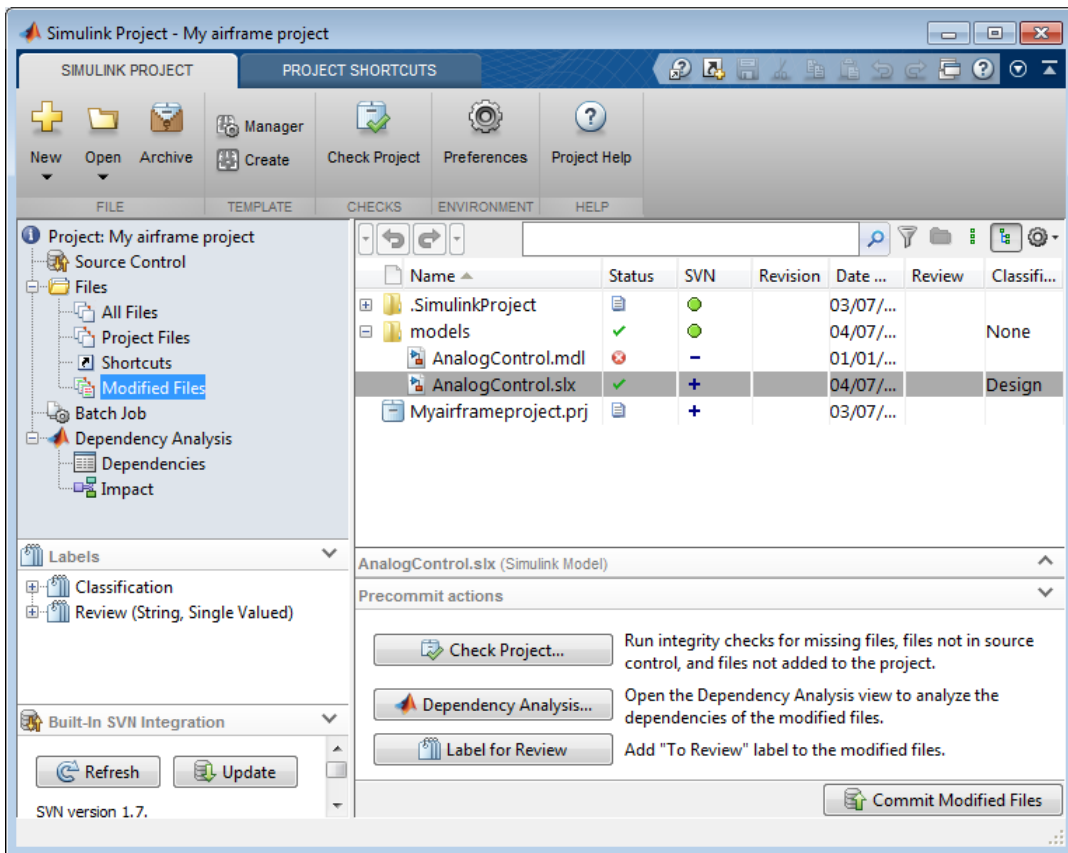
Concepts

- “About Source Control with Projects” on page 13-90

Commit Modified Files to Source Control

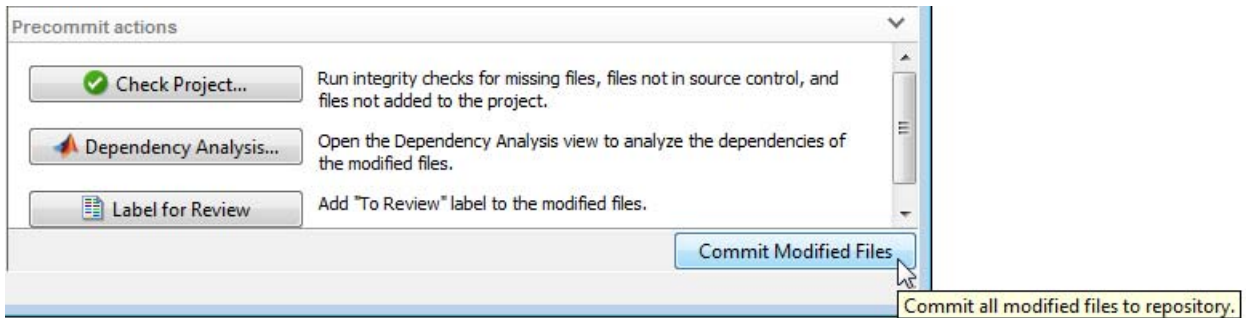
Before you commit modified files, review changes and consider precommit actions. See “Review Changes” on page 13-130 and “Precommit Actions” on page 13-132.

- 1 In Simulink Project, select the Modified Files view.



If you need to update the modified files list, click **Refresh** in the source control pane.

- 2 Click **Commit Modified Files** to check in all files in the modified files list.



3 Enter comments in the dialog box if you want, and click **Submit**.

Related Examples

- “Refresh Status of Project Files” on page 13-117
- “View Modified Files” on page 13-128
- “Precommit Actions” on page 13-132
- “Revert Local Changes and Release Locks” on page 13-136

Concepts

- “About Source Control with Projects” on page 13-90

Revert Local Changes and Release Locks

If you want to roll back local changes, right-click a file and select **Revert Local Changes and Release Locks** to release locks and revert to the version in the last sandbox update (e.g., the last version you synchronized or retrieved from the repository).

To abandon all local changes, select all the files in the Modified Files list, then right-click and select **Revert Local Changes and Release Locks**.


Select **Revert using Built-In SVN Integration Repository** to choose a revision to revert to.

Resolve Conflicts

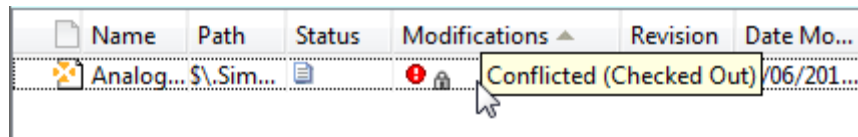
If two users change the same file in different sandboxes, a conflict message appears when you try to commit your modified files. Conflicts can also occur in project definition files. For example, if two users change the labels attached to a file, a conflict occurs in one of the project definition files in the .SimulinkProject folder.

To resolve conflict problems:

- 1 Look for conflicted files in the Modified Files view.

Tip It can be useful to switch to List view .

- 2 Check the source control status column (e.g., the **SVN** or **Modifications** column) for files with a red warning.



- 3 Examine the conflict. Select **Compare to Ancestor** to run a comparison with the last checked out version in the sandbox, or **Compare to Revision** to select revisions for comparison.

In the Comparison Tool, examine the report to see how the files have changed.

- 4 Resolve the conflict. In the Comparison Tool report, you can merge changes between revisions (requires Simulink Report Generator). For example, you can merge project file labels and shortcuts.

Open files from the project to make changes and resolve the conflict. If you need to change labels manually, see “Add Labels to Files” on page 13-66. To change the project description, click the Project node and edit the Description box.

- 5 When you have resolved the changes and want to commit the version in your sandbox, right-click the file and select **Mark Conflict Resolved**. You can then commit the file.

Work with Derived Files in Projects

Best practice is to omit derived and temporary files from your project or exclude them from source control. Use **Check Project** in the Precommit Actions pane or the **Simulink Project** tab to check the integrity of the project. If you add the `s1prj` 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 `s1prj` 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 `s1prj` is under source control and you generate code, most of the files under `s1prj` change and become locked. Other users cannot generate code because of file permission errors. The `s1prj` folder is also used for simulation via code generation (e.g., 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 `s1prj` is often required. However, deleting `s1prj` causes problems such as “not a working copy” errors if the folder is under some source control tools (e.g., 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 `s1prj` 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 `s1prj` 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.

What Is Dependency Analysis?

Project Dependency Analysis

In Simulink Project, you can analyze project structure and discover files required by your project in the Dependency Analysis view.

- You can use dependency analysis to help you set up your project with all required files. For example, you can add a top-level model to your project, analyze the model dependencies, and then add all dependent files to your project. See “Choose Files and Run Dependency Analysis” on page 13-141.
- You can run dependency analysis at any point in your workflow when you want to check that the project has all required files. For example, you can check dependencies before submitting a version of your project to source control. To work with results, see “Check Dependencies Results and Resolve Problems” on page 13-144.
- You can use the Impact graph view of a dependency analysis to analyze the structure of a project visually. You can perform impact analysis to find the impact of changing particular files. From the graph, you can examine your project structure and perform file operations such as adding labels and opening files. See “Perform Impact Analysis” on page 13-148.

Model Dependency Analysis

Simulink Project can analyze file dependencies for your entire project. For detailed dependency analysis of a *specific* model, use the manifest tools to control more options. Use the manifest tools if you want to:

- Save the list of the model dependencies to a manifest file.
- Create a report to identify where dependencies arise.
- Control the scope of dependency analysis.
- Identify required toolboxes.

See “Analyze Model Dependencies” on page 13-159.

Choose Files and Run Dependency Analysis

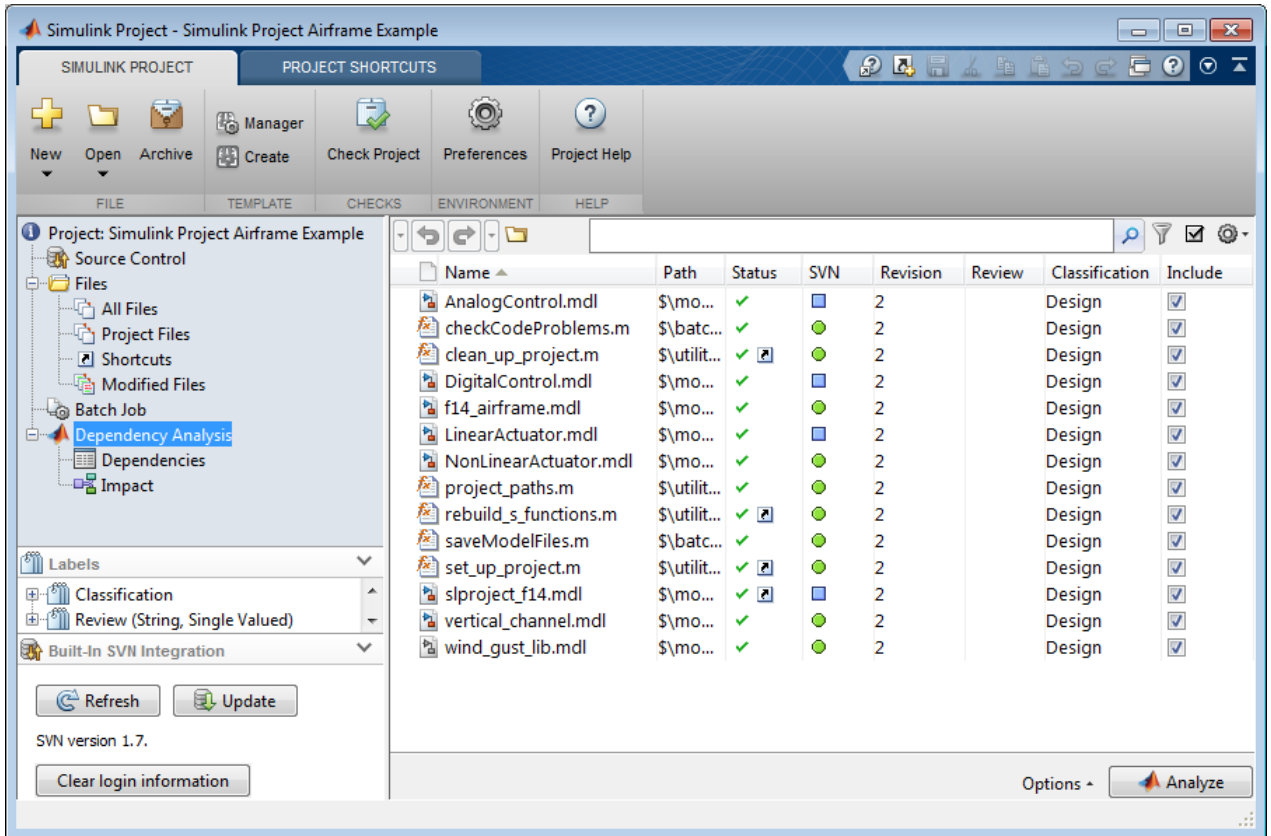
Note You can analyze only files within your project. If your project is new, you must add files to the project before running dependency analysis. See “Add Files to the Project” on page 13-25.

1 In Simulink Project, click Dependency Analysis in the Project tree.

Files in your project are listed.

2 Select the check boxes in the **Include** column for the files you want to analyze.

Tip To include or exclude all files, press **Ctrl+A** to select all files, then right-click and select **Include** or **Exclude**.



Note If you use the search box or Filter button to select the files to display, a message appears if the filter hides files selected for analysis. To show all files selected for analysis, click the Show only included files button .

- 3 If you want to include requirements documents, click **Options** and select the **Find requirements documents** check box.

In Simulink Project, Dependency Analysis searches for requirements documents linked using the Requirements Management Interface. You can view linked requirements documents in Simulink Project and navigate

to and from the linked documents. You can create or edit Requirements Management links only if you have Simulink Verification and Validation.

4 Click Analyze.

The view changes to the Dependencies view and displays the results in list form.

For next steps, see “Check Dependencies Results and Resolve Problems” on page 13-144.

**Related
Examples**

- “Check Dependencies Results and Resolve Problems” on page 13-144
- “Perform Impact Analysis” on page 13-148
- “Save and Reload Dependency Analysis Results” on page 13-157

Check Dependencies Results and Resolve Problems

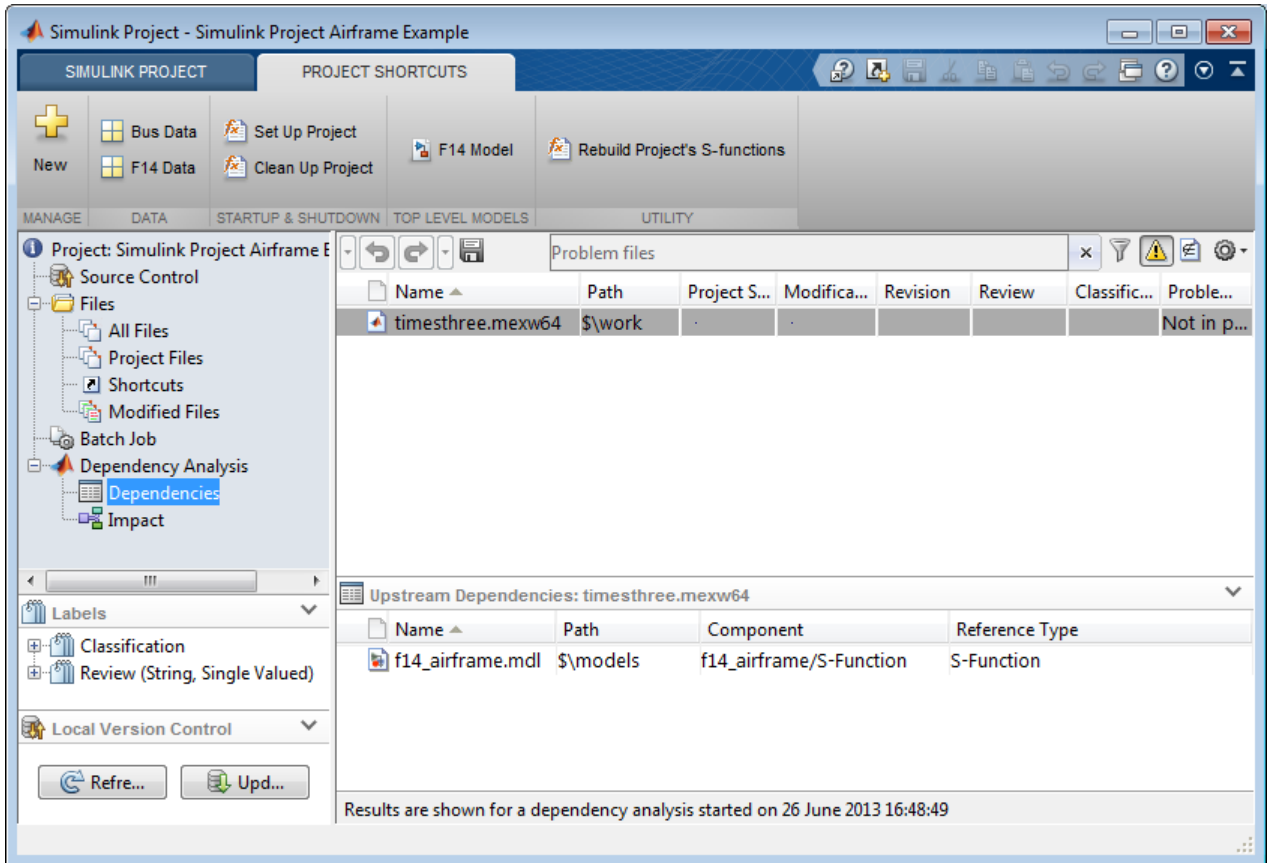
If dependency analysis finds problems, Simulink Project displays the problem files. The search box shows that the list is filtered to show only problem files.



The list reports dependent files to review because they are required by the project but not currently in the project or missing from the file system.

- 1 Click each file in the Problem list.

The lower pane displays the files that use the selected file, under **Upstream Dependencies: *filename***.




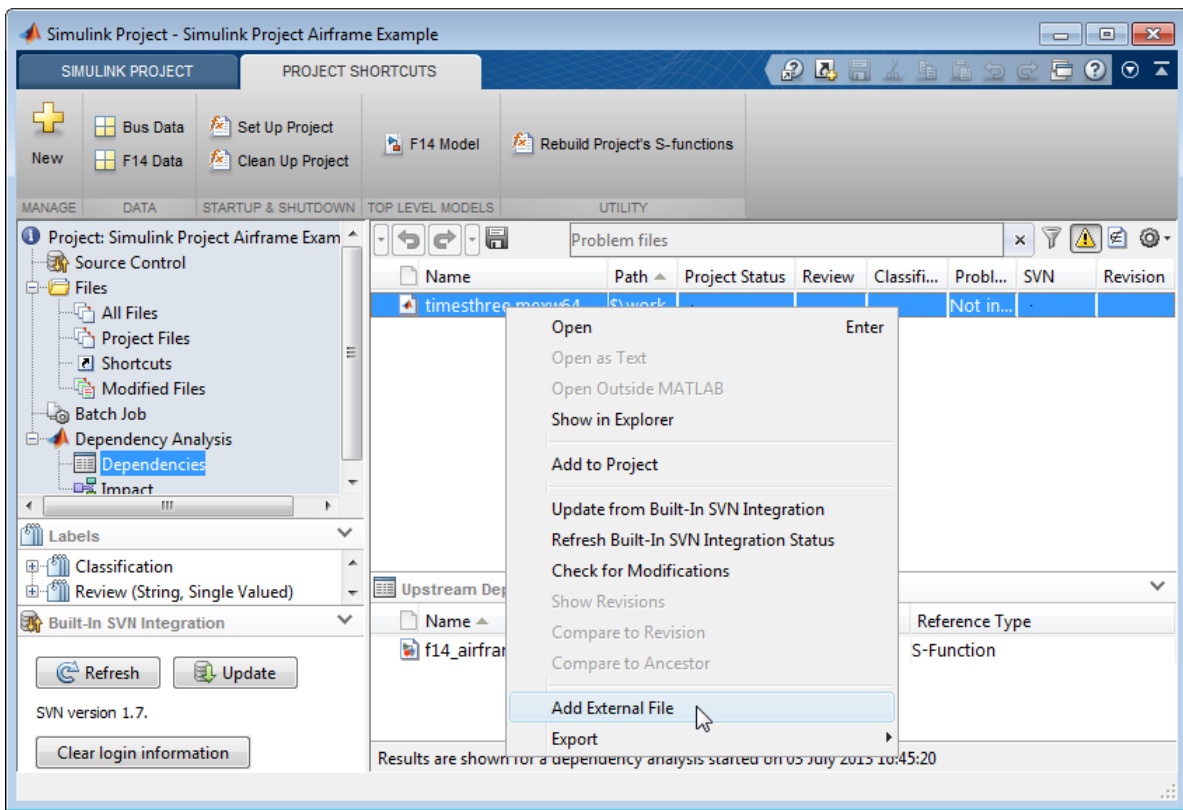
These are the some of the actions you can take to resolve problems after running a dependency analysis:

- In the dependencies table, check the Problem Description and Project Status of dependent files.
- To open the referencing component for editing, right-click a file in the **Upstream Dependencies: filename** table and select **Open**. MATLAB files open in the MATLAB Editor, and Simulink models open in the Model Editor with the block highlighted.
- Check the **Path**, where \$ indicates the project root. Check 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 other resource that is not part of your project. Use dependency analysis to ensure that you understand the design dependencies.

You do not necessarily need to add all required files to the project. For example, you can exclude derived S-Function binary files that the source code in your project generates. See “Work with Derived Files in Projects” on page 13-139.

- To remove a file from the Problem list without adding it to the project, right-click the file and select **Add External File**. The file disappears from the Problem list. Next time you run dependency analysis, this file does not appear in the Problem list. To view all external files, click the Show only external files button .



- If you need a file outside the project root in your project, copy or move it within the project root, and add it the project and the path. Remember to remove the original file location from the path.
- Clear the search box to view all identified dependencies, not just problem files.

Note If you clear the search box, you can return to viewing only problem files or external files by clicking the toolbar buttons to the right of the search box.

- Click the Impact view to investigate your project dependencies graphically. See “Perform Impact Analysis” on page 13-148.

Related Examples

- “Choose Files and Run Dependency Analysis” on page 13-141
- “Perform Impact Analysis” on page 13-148
- “Save and Reload Dependency Analysis Results” on page 13-157

Perform Impact Analysis

In this section...
“About Impact Analysis” on page 13-148
“Find Impact of Selected Files” on page 13-149
“Explore Impact Graph” on page 13-152
“Export Impact Results” on page 13-155

About Impact Analysis

In Simulink Project, use impact analysis to find the impact of changing particular files. You can investigate dependencies visually to explore the structure of your project. You can also analyze selected or modified files to find their required files and the files they impact. Visualize changes and dependencies in the Impact graph.

Impact analysis can show you how a change will impact other files before you make the change. For example:

- 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. This can help you identify design and test files that need modifications and help you find the tests you need to run.

After performing impact analysis, you can:

- Label or open the files
- Export the results to a workspace variable
- Export the results to batch processing
- Save the graph as an image file
- Save the impact results to a `.graphml` file that you can reload in Simulink Project

Exporting the results enables further processing or archiving of impact analysis results. You can add the exported list of files to reports or artifacts that describe the impact of a change.

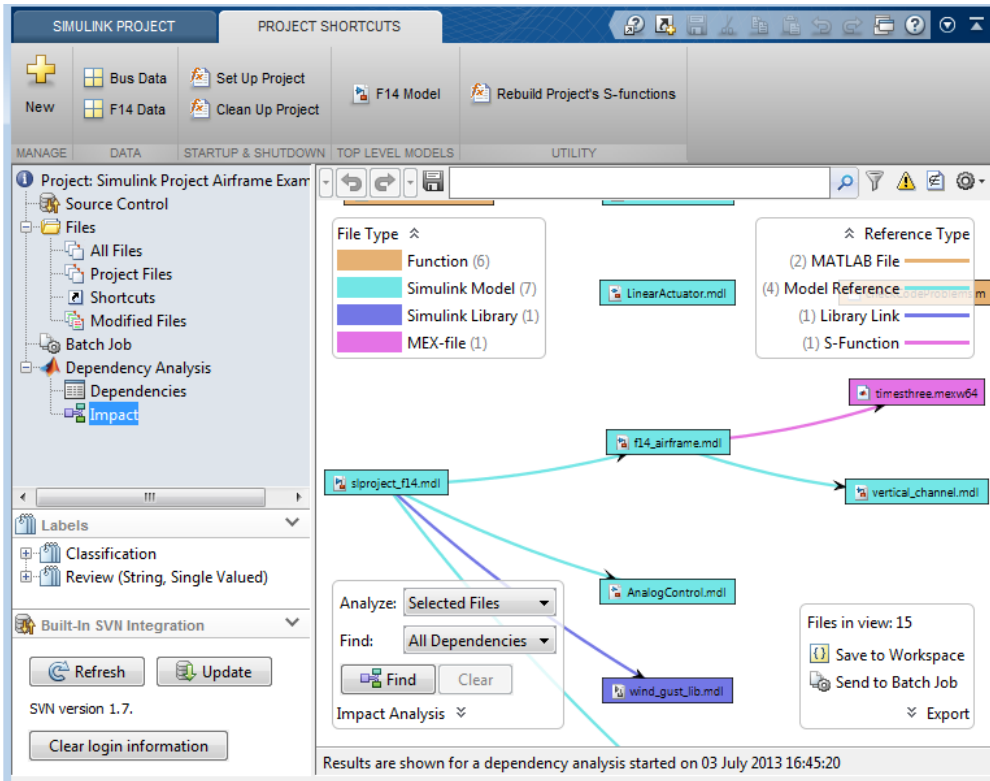
Find Impact of Selected Files

Perform Dependency Analysis

To investigate dependencies visually, first perform a dependency analysis on your project.

- 1** In the Project tree, click Dependency Analysis.
- 2** Include all files.
- 3** If you want to find linked requirements, select **Options > Find requirements documents**.
- 4** Click **Analyze**.
- 5** Click Impact in the Project tree under Dependency Analysis.

The graph displays the structure of all analyzed dependencies in the project. Some project files might not be visible in the graph, because they are not detectable dependencies of the analyzed files.



Analyze the Impact of Selected Files

After performing dependency analysis, to find the impact of particular files, use the Impact Analysis controls in the graph.

- 1 Leave the default **Selected Files** in the Analyze list.
- 2 Use the search and filter controls to modify the graph before selecting files.
- 3 Select files to analyze by clicking the graph or a legend.

For example, select all model files to analyze by clicking **Simulink Model** in the **File Type** legend. Change the legend using the **Highlight Files** context menu, for example to show particular labels.

- 4 Leave the default All Dependencies in the **Find** list to show all results. Or, select Impacted Files or Required Files to find only upstream or downstream dependencies.

- 5 Click **Find**.

The graph shows the selected files and their file dependencies.

The screenshot displays the Simulink Project environment with a dependency analysis window open. The window title is "Dependencies of 'slproject_f14.mdl'". The interface includes a toolbar with icons for search, refresh, and other actions. Below the toolbar, there are two filter panels: "File Type" and "Reference Type".

The "File Type" panel shows the following counts:

- Simulink Model (5)
- Simulink Library (1)
- MEX-file (1)

The "Reference Type" panel shows the following counts:

- (4) Model Reference
- (1) Library Link
- (1) S-Function

The main area of the window displays a dependency graph. The central node is "slproject_f14.mdl", which is highlighted in blue. It has five outgoing arrows pointing to other files: "f14_airframe.mdl", "AnalogControl.mdl", "wind_gust_lib.mdl", "NonLinearActuator.mdl", and "vertical_channel.mdl". The "f14_airframe.mdl" node is highlighted in cyan, and "vertical_channel.mdl" is highlighted in purple. The "AnalogControl.mdl" node is highlighted in cyan, "wind_gust_lib.mdl" is highlighted in blue, and "NonLinearActuator.mdl" is highlighted in cyan.

At the bottom of the window, there are several controls:

- Analyze: Selected Files
- Find: All Dependencies
- Buttons: Find, Clear
- Impact Analysis: (dropdown menu)
- Selected files: 1
- Buttons: Save to Workspace, Send to Batch Job, Export

At the very bottom of the window, a status bar reads: "Results are shown for a dependency analysis started on 03 July 2013 16:45:20".

Analyze the Impact of Modified Files

- 1 Select **Modified Files** in the **Analyze** list.
- 2 Leave the default **All Dependencies** in the **Find** list to show all results. Or, select **Impacted Files** or **Required Files** to find only upstream or downstream dependencies.
- 3 Click **Find**.

The graph shows only the modified files and their file dependencies.
- 4 To see which files are modified in the graph, right-click and select **Highlight Files > Modifications** to highlight the modified files.
- 5 You might want to select **Compare to Ancestor** or **Compare to Revision**. See “Review Changes” on page 13-130.

Tip To reset the graph to show all analyzed dependencies in the project, click **Clear** in the Impact Analysis controls, or clear the filter box.

Explore Impact Graph

These are the some of the actions you can take to investigate the structure of your project in the graph.

To change the graph layout and view:

- Right-click the graph canvas to try different graph layouts and change zoom.
- Zoom using a mouse wheel, or press + and – to zoom in or out.
- Press the space bar to fit to view.
- Click the mouse wheel and drag to pan, or use the arrow keys.

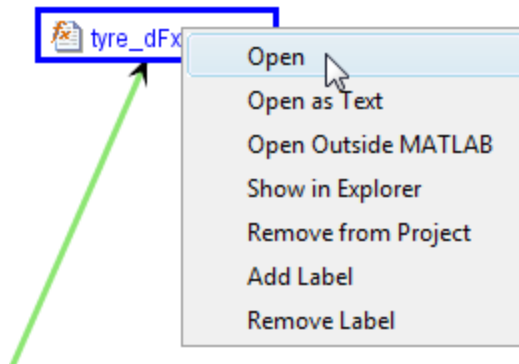
Your graph layout is saved with the project. You can also save and reload graph layouts. See “Save and Reload Dependency Analysis Results” on page 13-157.

- Right-click and select **Show Self-Dependencies** if you want to see self-references. By default the graph hides self-references to present a

simpler view of other graph items. For example, a library that contains multiple blocks that use another block in the same library shows many self-references. Self-referencing files do not introduce any new file dependencies, so you can leave these self-dependencies hidden to simplify the graph. Similarly, you can select **Show Parallel Dependencies** if you want to see multiple lines between the same items. By default, parallel dependencies are consolidated.

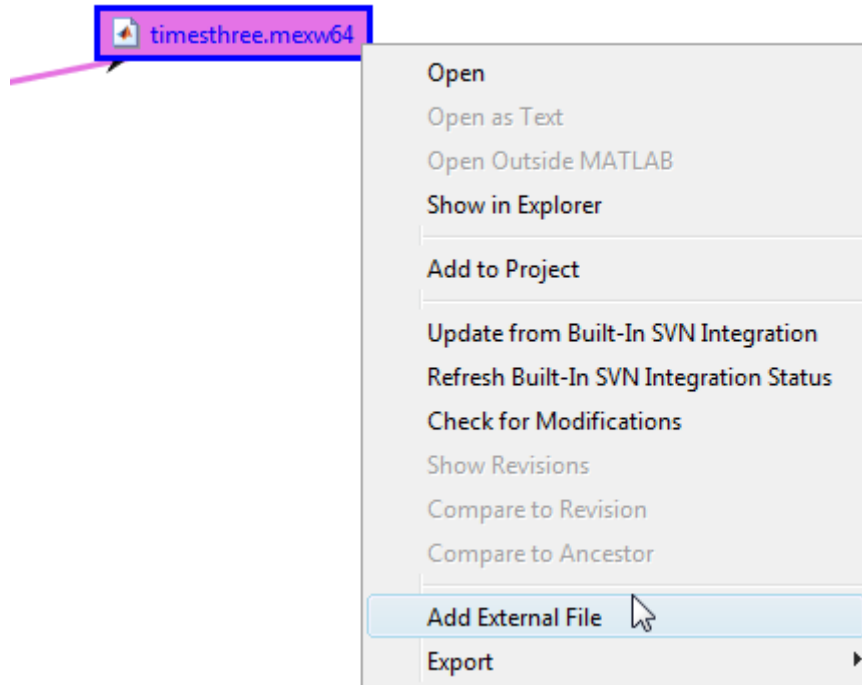
To perform file operations:

- Right-click files in the graph to use commands such as **Open**, **Add Label**, or **Remove from Project**. You can also double-click to open a file.



You can perform file operations on multiple files. To select multiple files, press **Shift** and drag the mouse to enclose the files. Hold down **Ctrl** to multiselect and add to any existing selection. Press **F** to fit the view to the currently selected files.

- You can take actions to resolve problems in the Impact graph in the same way as in the Dependencies table. For example, right-click to **Add to Project** or **Add External File**.



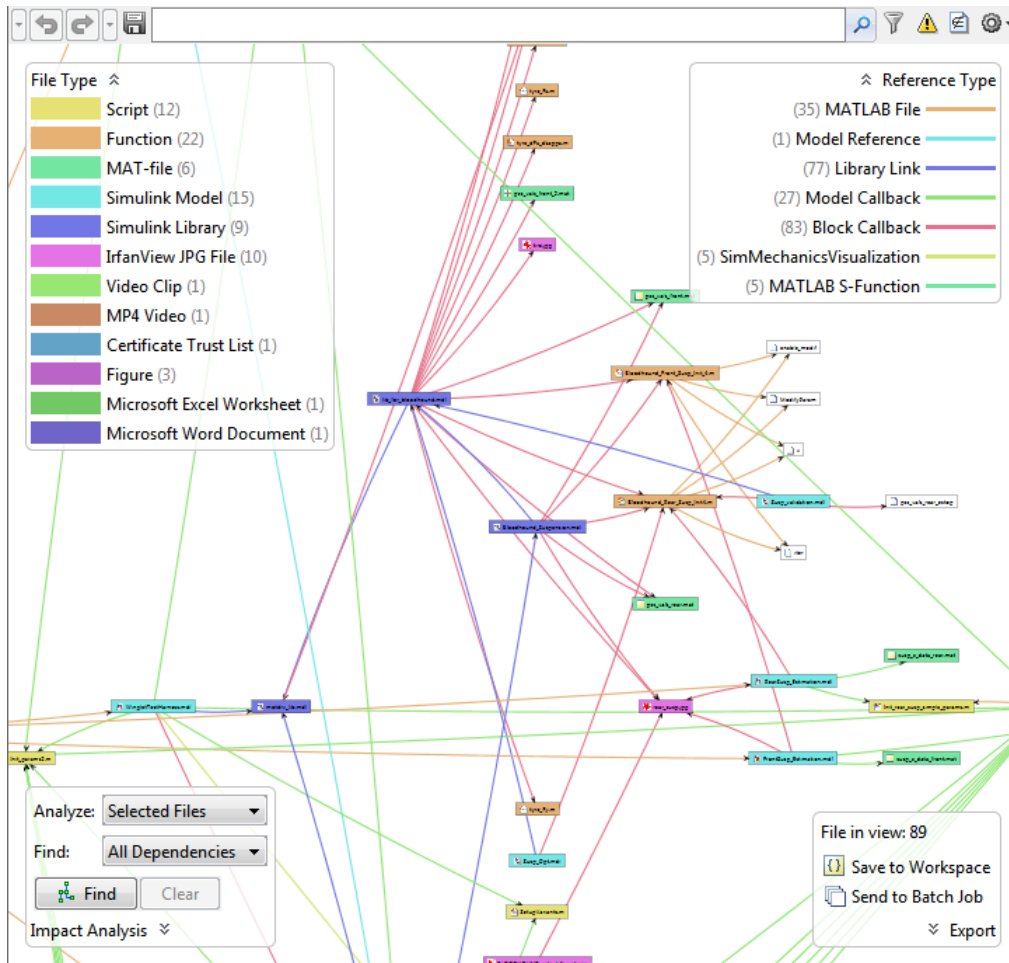
To highlight or select files by labels or status:

- Right-click the graph canvas and select **Highlight Files** and then select a category of labels to use for highlighting. For example, select **Highlight Files > Review** to see which files have labels in the Review category, such as To Review.

The **Review** legend appears and files in the graph are colored to indicate each file's label.

- Click a legend item to select files. For example, In the **Review** legend, click a label to select the files with that label. Selected files display a blue box.
- Use the **Highlight Files** context menu to highlight graph items by file type, problem type, project status, source control status, and other options. For example, if your project is under SVN source control, select **Highlight Files > Modifications** to show modified files.

The following example shows a project impact graph with many file types and reference types highlighted.



Export Impact Results

To export impact analysis results, use the Export controls in the graph, or use the **Export** context menu on selected files.

Select Files to Export

To export all the files in the current view, ensure no files are selected. (Click the graph background to clear the selection on all files). The Export pane displays **Files in view:** *number of files*.

Select a subset of files in the graph to export. The Export pane displays **Selected files:** *number of files*.

Export Files

- Click **Save to Workspace** to export the selected file paths to a variable.
- Click **Send to Batch Job** to switch to the Batch Job view with the files selected for batch job processing.

Export Graph to Image File

To export the impact graph to an image file to share or archive, you can either:

- Save an image file. Right-click the graph and select **Export to Image** to save an image file.
- Copy the image to the clipboard. Right-click the graph and select **Copy**. You can paste the clipboard contents into other documents.

Export Reloadable Results

To export the impact results to a `.graphml` file that you can reload in Simulink Project, see “Save and Reload Dependency Analysis Results” on page 13-157.

Related Examples

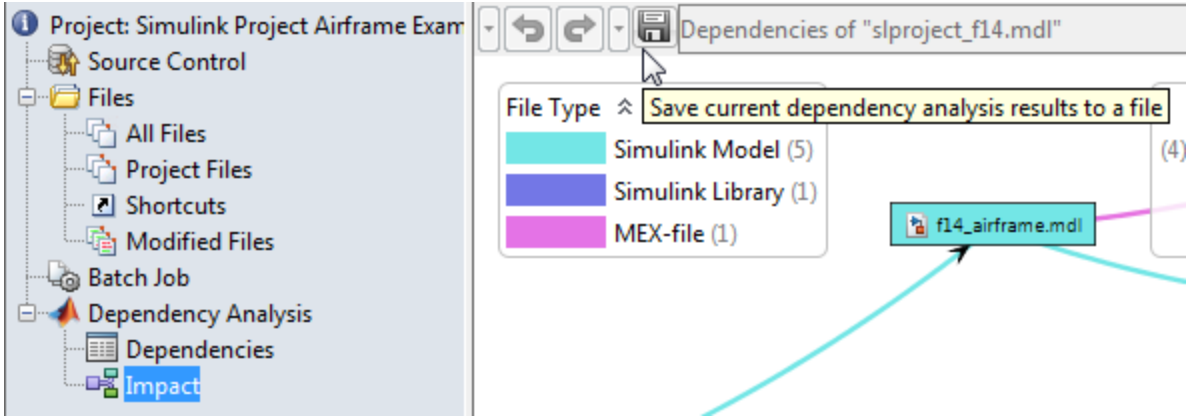
- “Choose Files and Run Dependency Analysis” on page 13-141
- “Check Dependencies Results and Resolve Problems” on page 13-144
- “Save and Reload Dependency Analysis Results” on page 13-157

Save and Reload Dependency Analysis Results

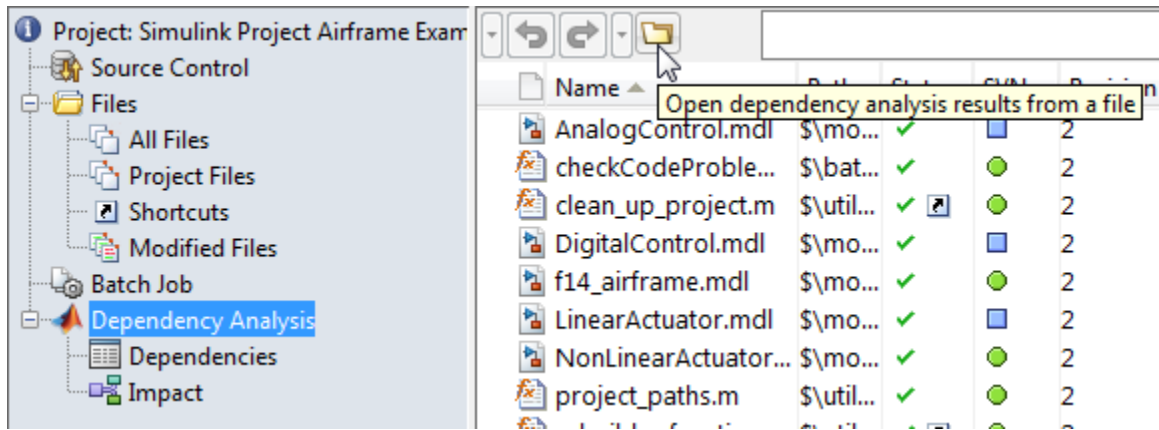
Simulink Project saves the results of previous dependency analysis with your project. You can view previous results without having to run a time-consuming analysis again. The Graph view saves your layout and display selections.

You can also save results separately in named files and reload them.

- To save dependency analysis results, in either the Dependencies or Graph views, click the Save a dependency analysis button.



- You can create or load analyses from the Dependency Analysis view. To load saved dependency analysis results, in the Dependency Analysis view, click the Open a dependency analysis button. The Simulink Project saves your results as .graphml file.



You can save reports with more detailed results by using model dependency analysis. For more information about choosing model or project dependency analysis, see “What Is Dependency Analysis?” on page 13-140.

Analyze Model Dependencies

In this section...
“What Are Model Dependencies?” on page 13-159
“Generate Manifests” on page 13-160
“Command-Line Dependency Analysis” on page 13-167
“Edit Manifests” on page 13-169
“Compare Manifests” on page 13-172
“Export Files in a Manifest” on page 13-173
“Scope of Dependency Analysis” on page 13-175
“Best Practices for Dependency Analysis” on page 13-178
“Use the Model Manifest Report” on page 13-179

What Are Model Dependencies?

Each Simulink model requires a set of files to run successfully. These files can include referenced models, data files, S-functions, and other files the model cannot run without. These required files are called *model dependencies*.

Dependency Analysis Requirements	Tools to Choose
Find required files for an entire project.	Use dependency analysis from the Simulink Project. See “Choose Files and Run Dependency Analysis” on page 13-141.
Perform detailed dependency analysis of a specific model with control of more options.	Use the manifest tools from your model. See “Generate Manifests” on page 13-160. Generate a manifest if you want to: <ul style="list-style-type: none"> • Save the list of the model dependencies to a manifest file. • Create a report to identify where dependencies arise.

Dependency Analysis Requirements	Tools to Choose
	<ul style="list-style-type: none"> • Control the scope of dependency analysis. • Identify required toolboxes.

After you generate a manifest for a model to determine its dependencies, you can:

- View the files required by your model in a manifest file.
- Trace dependencies using the report to understand why a particular file or toolbox is required by a model.
- Package the model with its required files into a zip file to send to another Simulink user.
- Compare older and newer manifests for the same model.
- Save a specific version of the model and its required files in a revision control system.

You can also view the libraries and models referenced by your model in a graphical format using the Model Dependency Viewer. See “Model Dependency Viewer” on page 9-83.

Generate Manifests

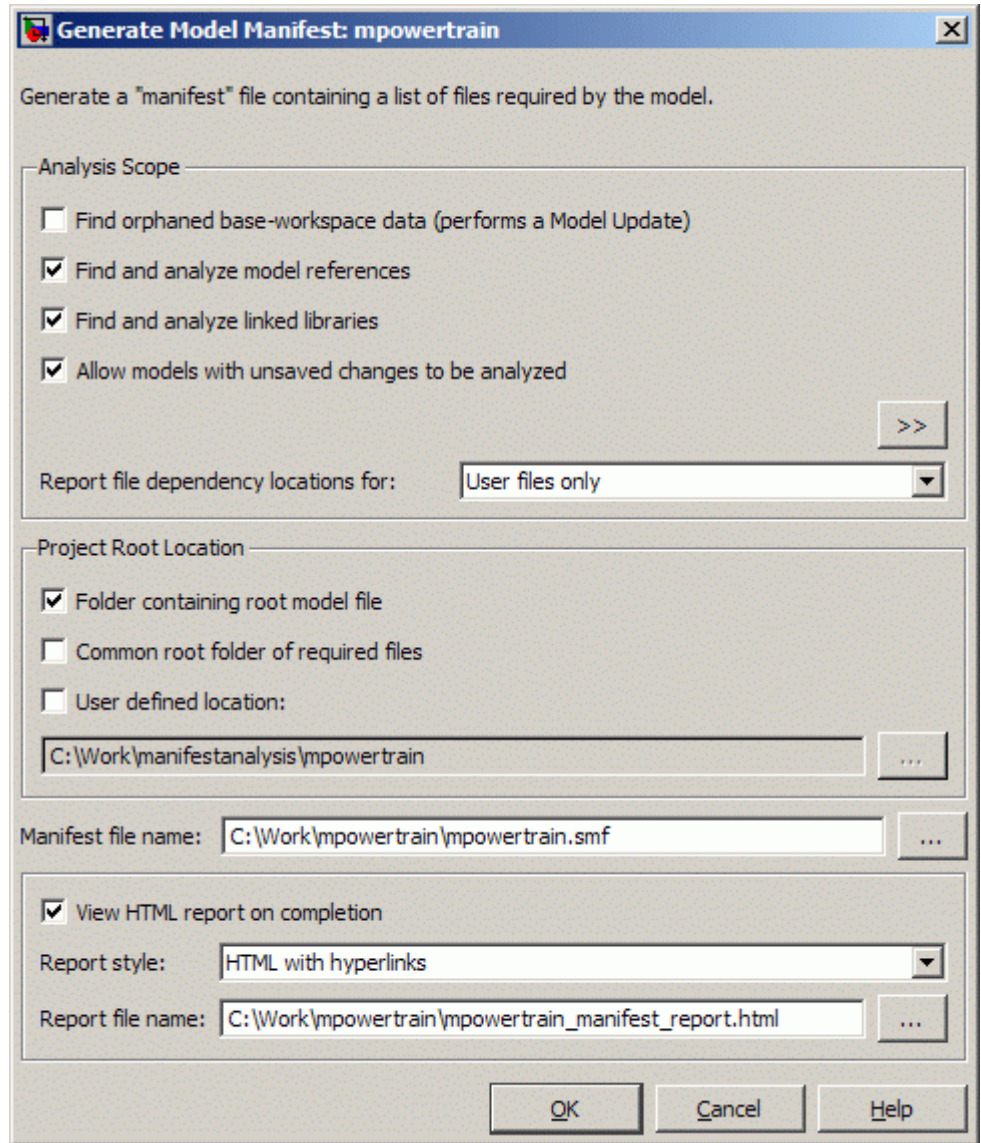
Generating a manifest performs the dependency analysis and saves the list of model dependencies to a manifest file. You must generate the manifest before using any of the other Simulink Manifest Tools.

Note The model dependencies identified in a manifest depend upon the **Analysis Scope** options you specify. For example, performing an analysis without selecting **Find Library Links** might not find all the Simulink blocksets that your model requires, because they are often included in a model as library links. See “Manifest Analysis Scope Options” on page 13-164.

To generate a manifest:

1 Select **Analysis > Model Dependencies > Generate Manifest**.

The Generate Model Manifest dialog box appears.



2 Click **OK** to generate a manifest and report using the default settings.

Alternatively you can first change the following settings:

- Select the **Analysis scope** check boxes to specify the type of dependencies you want to detect (see “Manifest Analysis Scope Options” on page 13-164).
- Control whether to report file dependency locations by selecting **Report file dependency locations for**:
 - **User files only** (default) — only report locations where dependencies are upon user files. Use this option if you want to understand the interdependencies of your own code and do not care about the locations of dependencies on MathWorks products. This option speeds up report creation and streamlines the report.
 - **All files** — report all locations where dependencies are introduced, including all dependencies on MathWorks products. This is the slowest option and the most verbose report. Use this option if you need to trace all dependencies to understand why a particular file or toolbox is required by a model. If you need to analyze many references, it can be helpful to sort the results by clicking the report column headers.
 - **None** — do not report any dependency locations. This is the fastest option and the most streamlined report. Use this option if you want to discover and package required files and do not require all the information about file references.
- If desired, change the **Project Root Location**. Select one of the check box options: **Folder containing root model file** (the default), **Common root folder of required files**, or **User-defined location** — for this option, enter a path in the edit box, or browse to a location.
- If desired, edit the **Manifest file name** and location in which to save the file.
- Use the check box **View HTML report on completion** to specify if you want to generate a report when you generate the manifest. You can edit the **Report file name** or leave the default, *mymodelname_manifest_report.html*. You can set the **Report style** to Plain HTML or HTML with Hyperlinks.

When you click **OK** Simulink generates a manifest file containing a list of the model dependencies. If you selected **View HTML report on completion**, the Model Manifest Report appears after Simulink generates the manifest. See “Use the Model Manifest Report” on page 13-179 for an example.

The manifest is an XML file with the extension `.smf` located (by default) in the same folder as the model itself.

Manifest Analysis Scope Options

The Simulink Manifest Tools allow you to specify the scope of analysis when generating the manifest. The dependencies identified by the analysis depend upon the scope you specify.

The following table describes the Analysis Scope options.

Check Box Option	Description
Find orphaned base workspace data (performs a Model Update)	Searches for base workspace variables the model requires, that are not defined in any file in this Manifest. If Model Update fails you see an error message. Either clear this analysis option to generate a manifest without a Model Update, or try a manual Model Update to find out more about the problem. For example your model may require variables that are not present in the workspace (e.g., if a block parameter defines a variable that you forgot to load manually).
Find and analyze model references	Searches for Model blocks in the model, and identifies any referenced models as dependencies.
Find and analyze linked libraries	Searches for links to library blocks in the model, and identifies any library links as dependencies.
Allow models with unsaved changes to be analyzed	Select this check box only if you want to allow analysis of unsaved changes.
Click the >> button on the right to show the following advanced analysis options.	

Check Box Option	Description
Find S-functions	Searches for S-Function blocks in the model, and identifies S-function files (MATLAB code and C) as dependencies. See the source code item in “Special Cases” on page 13-177.
Analyze model and block callbacks (including Interpreted MATLAB Function blocks)	Searches for file dependencies introduced by the code in Interpreted MATLAB Function blocks, block callbacks, and model callbacks. For more detail on how callbacks are analyzed, see “Code Analysis” on page 13-176.
Find files required for code generation	Searches for file dependencies introduced by Simulink Coder custom code, and Embedded Coder templates. If you do not have a code generation product, this check is off by default, and produces a warning if you select it. This includes analysis of all configuration sets (not just the Active set) and <i>STF_make_rtw_hook</i> functions, and locates system target files and Code Replacement Library definition files (.m or .mat). See also “Required Toolboxes” on page 13-181, and the source code item in “Special Cases” on page 13-177.
Find data files (e.g. in “From File” blocks)	Searches for explicitly referenced data files, such as those in From File blocks, and identifies those files as dependencies. See “Special Cases” on page 13-177.
Analyze Stateflow charts	Searches for file dependencies introduced through the use of syntax such as <code>ml.mymean(myvariable)</code> in models that use Stateflow.

Check Box Option	Description
Analyze code in MATLAB Functions blocks	Searches for MATLAB Function blocks in the model, and identifies any file dependencies (outside toolboxes) introduced in the code. Toolbox dependencies introduced by a MATLAB Function block are not detected.
Find requirements documents	<p>Searches for requirements documents linked using the Requirements Management Interface. Note that requirements links created with IBM Rational DOORS software are not included in manifests. For more information, see “Requirements Management Interface Setup” in the Simulink Verification and Validation documentation.</p> <p>This option is disabled if you do not have a Simulink Verification and Validation license, and Simulink ignores any requirements links in your model.</p>
Analyze files in “user toolboxes”	Searches for file dependencies introduced by files in user-defined toolboxes. See “Special Cases” on page 13-177.
Analyze MATLAB files	Searches for file dependencies introduced by MATLAB files called from the model. For example, if this option is selected and you have a callback to <code>mycallback.m</code> , then the referenced file <code>mycallback.m</code> is also analyzed for further dependencies. See “Code Analysis” on page 13-176.
Store MATLAB code analysis warnings in manifest	Saves any warnings in the manifest.

See also “Scope of Dependency Analysis” on page 13-175 for more information.

Command-Line Dependency Analysis

- “Check File Dependencies” on page 13-167
- “Check Toolbox Dependencies” on page 13-167

Check File Dependencies

To programmatically check for file dependencies, use the function `dependencies.fileDependencyAnalysis` as follows.

```
[files, missing, depfile, manifestfile] =  
dependencies.fileDependencyAnalysis('modelname', 'manifestfile')
```

This returns the following:

- *files* — a cell array of strings containing the full-paths of all existing files referenced by the model *modelname*.
- *missing* — a cell array of strings containing the names all files that are referenced by the model *modelname*, but cannot be found.
- *depfile* — returns the full path of the user dependencies (`.smd`) file, if it exists, that stores the names of any files you manually added or excluded. Simulink uses the `.smd` file to remember your changes the next time you generate a manifest. See “Edit Manifests” on page 13-169.
- *manifestfile* — (optional input) specify the name of the manifest file to create. Note that the suffix `.smf` is always added to the user-specified name.

If you specify the optional input, *manifestfile*, then the command creates a manifest file with the specified name and path *manifestfile*. *manifestfile* can be a full-path or just a file name (in which case the file is created in the current folder).

If you try this analysis on an example model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the example models.

Check Toolbox Dependencies

To check which toolboxes are required, use the function `dependencies.toolboxDependencyAnalysis` as follows:

```
[names,dirs] = dependencies.toolboxDependencyAnalysis(files_in)
```

files_in must be a cell array of strings containing .m or model files on the MATLAB path. Simulink model names (without file extension) are also allowed.

This returns the following:

- *names* — a cell-array of toolbox names required by the files in *files_in*.
- *dirs* — a cell-array of the toolbox folders.

Note The method `toolboxDependencyAnalysis` looks for toolbox dependencies of the files in `files_in` but does *not* analyze any subsequent dependencies.

If you want to find all detectable toolbox dependencies of your model *and* the files it depends on:

- 1 Call `fileDependencyAnalysis` on your model.

For example:

```
[files, missing, depfile, manifestfile] = dependencies.fileDependencyAnalysis('mymodel')

files =
    'C:\Work\manifest\foo.m'
    'C:\Work\manifest\mymodel'
missing =
    []
depfile =
    []
manifestfile =
    []
```

- 2 Call `toolboxDependencyAnalysis` on the files output of step 1.

For example:

```
tbxes = dependencies.toolboxDependencyAnalysis(files)

tbxes =
[1x24 char]    'MATLAB'    'Simulink Coder'    'Simulink'
```

To view long product names examine the `tbxes` cell array as follows:

```
tbxes{:}

ans =
Image Processing Toolbox

ans =
MATLAB

ans =
Simulink Coder

ans =

Simulink
```

For command line dependency analysis, the analysis uses the default settings for analysis scope to determine required toolboxes. For example, if you have code generation products, then the check **Find files required for code generation** is on by default and Simulink Coder is always reported as required. See “Required Toolboxes” on page 13-181 for more examples of how your installed products and analysis scope settings can affect reported toolbox requirements.

Edit Manifests

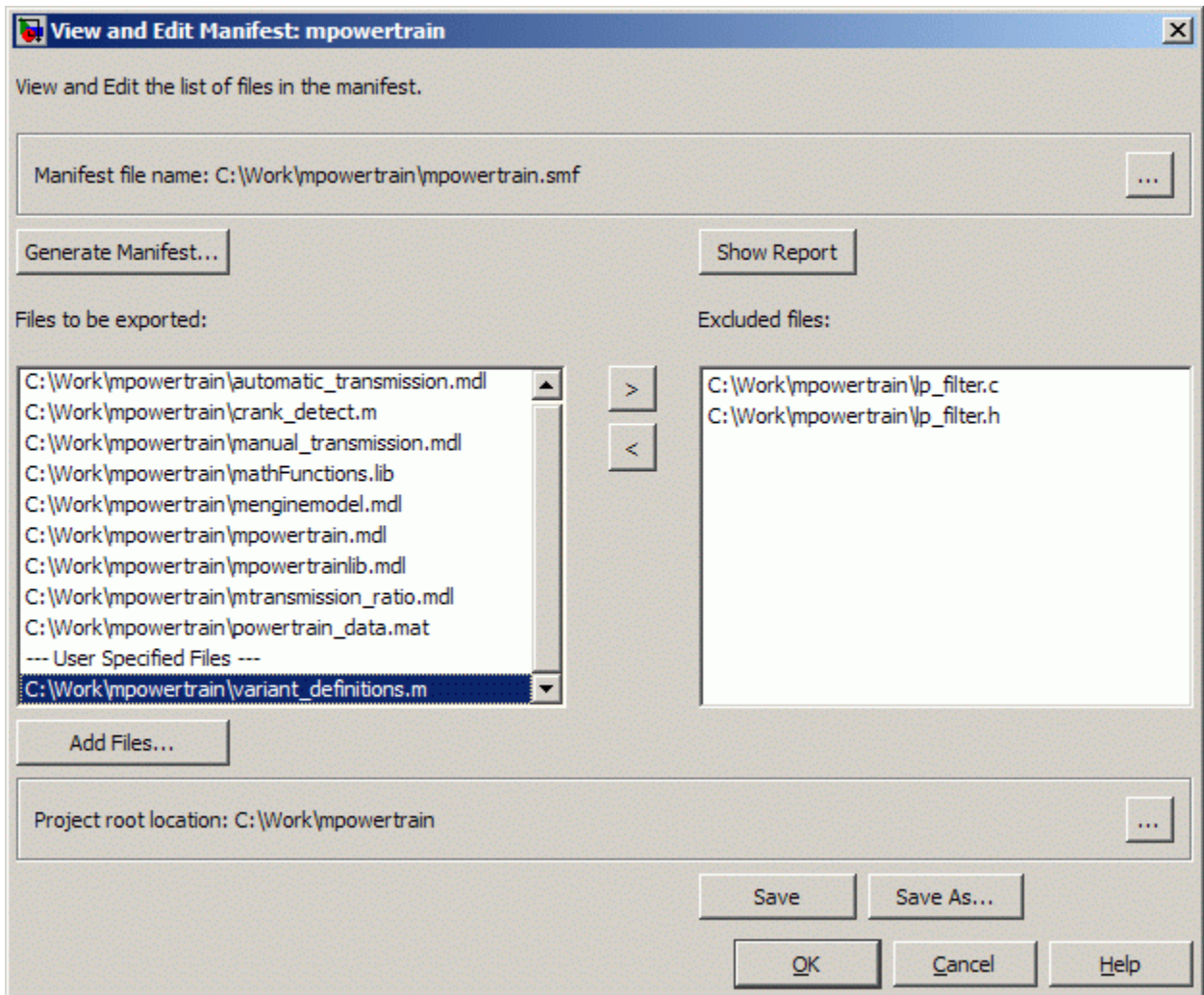
After you generate a manifest, you can view the list of files identified as dependencies, and manually add or delete files from the list.


To edit the list of required files in a manifest:

- 1 Select **Analysis > Model Dependencies > Edit Manifest Contents**.

Alternatively, if you are viewing a manifest report you can click **Edit** in the top **Actions** box, or you can click **View and Edit Manifest** in the Export Manifest dialog box.

The View and Edit Manifest dialog box appears, showing the latest manifest for the current model.



Note You can open a different manifest by clicking the Browse for manifest file button . If you have not generated a manifest, select **Generate Manifest** to open the Generate Model Manifest dialog box (see “Generate Manifests” on page 13-160).

2 Examine the **Files to be exported** list on the left side of the dialog box. This list shows the files identified as dependencies.

3 To add a file to the manifest:

a Click **Add Files**.


The Add Files to Manifest dialog box opens.

b Select the file you want to add, then click **Open**.

The selected file is added to the **Files to be exported** list.

4 To remove a file from the manifest:

a Select the file you want to remove from the **Files to be exported** list.

b Click the Exclude selected files button .

The selected file is moved to the **Excluded files** list.

Note If you add a file to the manifest and then exclude it, that file is removed from the dialog (it is not added to the **Excluded files** list). Only files detected by the Simulink Manifest Tools are included in the Excluded files list.

5 If desired, change the **Project Root Location**.

6 Click **Save** to save your changes to the manifest file.

Simulink saves the manifest (.smf) file, and creates a user dependencies (.smd) file that stores the names of any files you manually added or

excluded. Simulink uses the `.smd` file to remember your changes the next time you generate a manifest, so you do not need to repeat manual editing. For example, you might want to exclude source code or include a copyright document every time you generate a manifest for exporting to a customer. The user dependencies (`.smd`) file has the same name and folder as the model. By default, the user dependencies (`.smd`) file is also included in the manifest.

Note If the user dependencies (`.smd`) file is read-only, a warning is displayed when you save the manifest.

- 7** To view the Model Manifest Report for the updated manifest, click **Show Report**.

An updated Model Manifest Report appears, listing the required files and toolboxes, and details of references to other files. See “Use the Model Manifest Report” on page 13-179 for an example.

- 8** When you are finished editing the manifest, click **OK**.

Compare Manifests

You can compare two manifests to see how the list of model dependencies differs between two models, or between two versions of the same model. You can also compare a manifest with a folder or a ZIP file.

To compare manifests:

- 1** From the Current Folder browser, right-click a manifest file and select **Compare Against > Choose**.

Alternatively, from your model, select **Analysis > Model Dependencies > Compare Manifests**.

The dialog box Select Files or Folders for Comparison appears.

- 2** In the dialog box Select Files or Folders for Comparison, select files to compare, and the comparison type.
 - a** Use the drop-down lists or browse to select manifest files to compare.

- b** Select the **Comparison type**. For two manifests you can select:
 - **Simulink manifest comparison** — Select for a manifest file list comparison reporting new, removed and changed files. The report contains links to open files and compare files that differ. You can use a similar file **List comparison** for comparing a manifest to a folder or a ZIP file.
 - **Simulink manifest comparison (printable)** — Select for a printable **Model Manifest Differences Report** without links. The report provides details about each manifest file, and lists the differences between the files.
- 3** View the report in the Comparison Tool comparing the file names, dates, and sizes stored in the manifests.

Be aware the details stored in the manifest may differ from the files on disk. If you click a “compare” link in the report, you see warnings if there are problems such as size mismatches, or if the tool cannot find those files on disk.

For more information on the Comparison Tool, see “Comparing Files and Folders” in the MATLAB Desktop Tools and Development Environment documentation.

Export Files in a Manifest

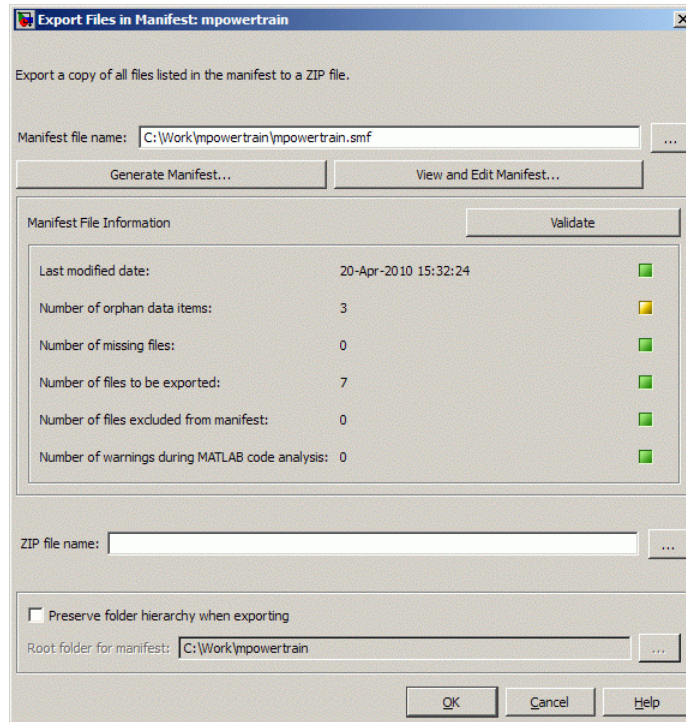
You can export copies of the files listed in the manifest to a ZIP file. Exporting the files allows you to package the model with its required files into a single ZIP file, so you can easily send it to another user or save it in a revision control system.


To export your model with its required files:

- 1** Select **Analysis > Model Dependencies > Export Files in Manifest**.

Alternatively, if you are viewing a manifest report you can click **Export** in the top **Actions** box.

The Export Files in Manifest dialog box appears, showing the latest manifest for the current model.



Note You can export a different manifest by clicking the Browse for manifest file button . If you have not generated a manifest, select **Generate Manifest** to open the Generate Model Manifest dialog box (see “Generate Manifests” on page 13-160).

- 2 If you want to view or edit the manifest before exporting it, click **View and Edit Manifest** to view or change the list of required files. See “Edit Manifests” on page 13-169. When you close the View and Edit Manifest dialog box, you return to the Export Files in Manifest dialog box.
- 3 Click **Validate** to check the manifest. Validation reports information about possible problems such as missing files, warnings, and orphaned base workspace data.

- 4 Enter the ZIP file name to which you want to export the model.
- 5 Select **Preserve folder hierarchy when exporting** if you want to keep folder structure for your exported model and files. Then, select the root folder to use for this structure (usually the same as the **Project Root Location** on the Generate Manifest dialog box).

Note You must select **Preserve folder hierarchy** if you are exporting a model that uses an .m file inside a MATLAB class (to maintain the folder structure of the class), or if the model refers to files in other folders (to ensure the exported files maintain the same relative paths).

- 6 Click **OK**.

The model and its file dependencies are exported to the specified ZIP file.

Scope of Dependency Analysis

The Simulink Manifest Tools identify required files and list them in an XML file called a *manifest*. When Simulink generates a manifest file, it performs a static analysis on your model, which means that the model does not need to be capable of performing an “update diagram” operation (see “Update a Block Diagram” on page 1-32). The only exception to this is when you select the analysis option **Find orphaned base workspace data (performs a Model Update)**.

You can specify the type of dependencies you want to detect when you generate the manifest. See “Manifest Analysis Scope Options” on page 13-164.

For more information on what the tool analyzes, refer to the following sections:

- “Analysis Limitations” on page 13-176
- “Code Analysis” on page 13-176
- “Special Cases” on page 13-177

Analysis Limitations

The analysis might not find all files required by your model (for examples, see “Code Analysis” on page 13-176).

The analysis might not report certain blocksets or toolboxes required by a model. You should be aware of this limitation when sending a model to another user. Blocksets that do not introduce dependence on any files (such as Fixed-Point Designer™) cannot be detected. Some SimEvents blocks do not introduce a detectable dependence on SimEvents.

To include dependencies that the analysis cannot detect, you can add additional file dependencies to a manifest file using the View/Edit Manifest Contents option (see “Edit Manifests” on page 13-169).

Code Analysis

When the Simulink Manifest Tools encounter MATLAB code, for example in a model or block callback, or in a .m file S-function, they attempt to identify the files it references. If those files contain MATLAB code, *and* the analysis scope option **Analyze MATLAB files** is selected, the referenced files are also analyzed. This function is similar to `depfun` but with some enhancements:

- Files that are in MathWorks toolboxes are not analyzed.
- Strings passed into calls to `eval`, `evalc`, and `evalin` are analyzed.
- File names passed to `load`, `fopen`, `xlsread`, `importdata`, `dlmread`, `wk1read`, and `imread` are identified.

File names passed to `load`, etc., are identified only if they are literal strings. For example:

```
load('mydatafile')  
load mydatafile
```

The following example, and anything more complicated, is not identified as a file dependency:

```
str = 'mydatafile';  
load(str);
```

Similarly, arguments to `eval`, etc., are analyzed only if they are literal strings.

The Simulink Manifest Tools look inside MAT-files to find the names of variables to be loaded. This enables them to distinguish reliably between variable names and function names in block callbacks.

If a model depends upon a file for which both `.m` and `.p` files exist, then the manifest reports both, and, if the **Analyze MATLAB files** option is selected, the `.m` file is analyzed.

Special Cases

The following list contains additional information about specific cases:

- If your model references a data class created using MATLAB syntax, for example called *MyPackage.MyClass*, all files inside the folder *MyPackage* and its subfolders are added to the manifest.

Warning The analysis adds all files in the class, which includes any source control files such as `.svn` or `.cvs`. You may want to edit the manifest to remove these files.

- A user-defined toolbox must have a properly configured `Contents.m` file. The Simulink Manifest Tools search user-defined toolboxes as follows:
 - If you have a `Contents.m` file in folder *X*, any file inside a subfolder of *X* is considered part of your toolbox.
 - If you have a `Contents.m` file in folder *X/X*, any file inside all subfolders of the “outer” folder *X* will be considered part of your toolbox.

For more information on the format of a `Contents.m` file, see `ver`.

- If your S-functions require TLC files, these are detected.
- If you have Simscape, your Simscape components are analyzed. See also “Required Toolboxes” on page 13-181 for other effects of your installed products on manifests.
- If you create a UI using GUIDE and add this to a model callback, then the dependency analysis detects the `.m` and `.fig` file dependencies.
- If you have a dependence on source code, such as `.c`, `.h` files, these files are not analyzed at all to find any files that they depend upon. For example, subsequent `#include` calls inside `.h` files are not detected. To make such files detectable, you can add them as dependent files to the “header file”

section of the Custom Code pane of the Simulink Coder section of the Configuration Parameters dialog box (or specify them with `rtwmakecfg`). Alternatively, to include dependencies that the analysis cannot detect, you can add additional file dependencies to a manifest file using the View/Edit Manifest Contents option (see “Edit Manifests” on page 13-169).

- Various blocksets and toolboxes can introduce a dependence on a file through their additional source blocks. If the analysis scope option **Find data files (e.g. in “From File” blocks)** is selected, the analysis detects file dependencies introduced by the following blocks:

Product	Blocks
DSP System Toolbox	From Wave File (Obsolete) block (Microsoft Windows operating system only) From Multimedia File block (Windows only)
Computer Vision System Toolbox™	Image From File block Read Binary File block
Simulink 3D Animation™	VR Sink block

The option **Find data files** also detects dependencies introduced by setting a "Model Workspace" for a model to either `MAT-File` or `MATLAB Code`, and model dependencies specified on the Model Referencing pane of the Configuration Parameters dialog box.

Best Practices for Dependency Analysis

The starting point for dependency analysis is the model itself. Make sure that the model refers to any data files it needs, even if you would normally load these manually. For example, add code to the model's `PreLoadFcn` to load them automatically, e.g.,

```
load mydatafile
load('my_other_data_file.mat')
```

This way, the Simulink Manifest Tools can add them to the manifest. For more detail on callback analysis, see the notes on code analysis (see “Code Analysis” on page 13-176).

More generally, ensure that the model creates or loads any variables it uses, either in model callbacks or in scripts called from model callbacks. This reduces the possibility of the Simulink Manifest Tools confusing variable names with function names when analyzing block callbacks.

If you plan to export the manifest after creating it, ensure that the model does not refer to any files by their absolute paths, for example:

```
load C:\mymodel\mydata\mydatafile.mat
```

Absolute paths can become invalid when you export the model to another machine. If referring to files in other folders, do it by relative path, for example:

```
load mydata\mydatafile.mat
```

Select **Preserve folder hierarchy** when exporting, so that the exported files are in the same locations relative to each other. Also, choose a root folder so that all the files listed in the manifest are inside it. Otherwise, any files outside the root are copied into a new folder called `external` underneath the root, and relative paths to those files become invalid.

If you are exporting a model that uses a `.m` file inside a MATLAB class (in a folder called `@myclass`, for example), you must select the **Preserve folder hierarchy** check box when exporting, to maintain the folder structure of the class.

Always test exported ZIP files by extracting the contents to a new location on your computer and testing the model. Be aware that in some cases required files may be on your path but not in the ZIP file, if your path contains references to folders other than MathWorks toolboxes.

Use the Model Manifest Report

- “Report Sections” on page 13-180
- “Required Toolboxes” on page 13-181
- “Example Model Manifest Report” on page 13-181

Report Sections

If you selected **View HTML report on completion** in the Generate Model Manifest dialog box, the Model Manifest Report appears after Simulink generates the manifest. The report shows:

- Analysis date
- **Actions** panel — Provides links to conveniently regenerate, edit or compare the manifest, and export the files in the manifest to a ZIP file.
- **Model Reference and Library Link Hierarchy** — Links you can click to open models.
- **Files used by this model** — Required files, with paths relative to the projectroot.

You can sort the results by clicking the report column headers.

- **Toolboxes required by this model.** For details see “Required Toolboxes” on page 13-181.
- **References in this model** — This section provides details of references to other files so you can identify where dependencies arise. You control the scope of this section with the **Report file dependency locations** options on the Generate Manifest dialog box. You can choose to include references to user files only, all files or no files. See “Generate Manifests” on page 13-160. Use this section of the report to trace dependencies to understand why a particular file or toolbox is required by a model. If you need to analyze many references, it can be helpful to sort the results by clicking the report column headers.
- **Folders referenced by this model**
- **Orphaned base workspace variables** — If you selected the analysis option **Find orphaned base workspace data**, this section reports any base workspace variables the model requires that are not defined in a file in this manifest.
- **Warnings generated while analyzing MATLAB code** — You can opt out of this section by clearing the **Store MATLAB code analysis warnings in manifest** analysis option.
- **Dependency analysis settings** — Records the details of the analysis scope options.

See the examples shown in “Example Model Manifest Report” on page 13-181.

Required Toolboxes

In the report, the “Toolboxes required by this model” section lists all products required by the model *that the analysis can detect*. Be aware that the analysis might not report certain blocksets or toolboxes required by a model, e.g., blocksets that do not introduce dependence on any files (such as Fixed-Point Designer) cannot be detected. Some MathWorks files under toolbox/shared can report only requiring MATLAB instead of their associated toolbox.

The results reported can be affected by your analysis scope settings and your installed products. For example:

- If you have code generation products and select the scope option “**Find files required for code generation**”, then:
 - Simulink Coder software is always reported as required.
 - If you also have an .ert system target file selected then Embedded Coder software is always reported as required.
- If you clear the **Find library links** option, then the analysis cannot find a dependence on, for example, *someBlockSet*, and so no dependence is reported upon the block set.
- If you clear the **Analyze MATLAB files** option, then the analysis cannot find a dependence upon *fuzzy.m*, and so no dependence is reported upon the Fuzzy Logic Toolbox™.

Example Model Manifest Report

You should always check the **Dependency analysis settings** section in the Model Manifest Report to see the scope of analysis settings used to generate it.

Following are portions of a sample report.

Model Manifest Report:mpowertrain

Actions

- [Re-generate this manifest](#)
- [Edit this manifest](#)
- [Compare this manifest with another one](#)
- [Export the files in this manifest to a ZIP file](#)

Analysis performed:09-Jul-2013 14:30:49

Model Reference and Library Link hierarchy

- [mpowertrain](#)
 - [menginemodel](#)
 - [mtransmission](#)
 - [mtransmission_ratio](#)
 - [mpowertrainlib](#)
 - [mpowertrainlib](#)

Files used by this model

Root folder for this manifest: C:\Work\SimulinkFiles\manifestanalysis\mpowertrain\mpowertrain_1

Click on a column header to sort the table

File Name	Size	Last Modified Date	Will be Exported
\$projectroot/crank_detect.m (open)	11613bytes	2009-06-03 10:26:38	true
\$projectroot/lp_filter.c (open)	17bytes	2006-10-27 17:11:58	true
\$projectroot/lp_filter.h (open)	20bytes	2006-10-27 17:12:00	true
\$projectroot/mathFunctions.lib (open)	4bytes	2006-10-27 17:12:00	true
\$projectroot/menginemodel.mdl (open)	19260bytes	2006-08-08 14:13:10	true
\$projectroot/mpowertrain.mdl (open)	58077bytes	2009-06-03 10:25:14	true
\$projectroot/mpowertrainlib.mdl (open)	34759bytes	2006-08-08 14:13:14	true
\$projectroot/mtransmission.mdl (open)	36071bytes	2009-06-03 10:29:10	true
\$projectroot/mtransmission_ratio.mdl (open)	24761bytes	2009-06-03 10:26:46	true
\$projectroot/powertrain_data.mat	1976bytes	2006-10-27 17:12:02	true

Toolboxes required by this model

- MATLAB (8.2)
- Simulink (8.2)
- Simulink Coder (8.5)
- Stateflow (8.2)

References in this model

Use the table below to determine where in a model a dependence upon a particular file or toolbox originates.

Click on a column header to sort the table

Reference Type	Reference Location	File Name	Toolbox
Model Reference	mpowertrain/engine model (show)	\$projectroot/menginemodel.mdl (open)	(not in a toolbox)
Library Link	mpowertrain/Library Shift Logic (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
Model Reference	mpowertrain/transmission (show)	\$projectroot/mtransmission.mdl (open)	(not in a toolbox)
Model Callback, PreLoadFcn	mpowertrain (show)	\$projectroot/powertrain_data.mat	(not in a toolbox)
Simulink Coder, Configuration, Custom Code	mtransmission (show)	\$projectroot/lp_filter.c (open)	(not in a toolbox)
Simulink Coder, Configuration, Custom Code	mpowertrain/transmission (show)	\$projectroot/lp_filter.h (open)	(not in a toolbox)
Simulink Coder, Configuration, Custom Code	mtransmission (show)	\$projectroot/mathFunctions.lib (open)	(not in a toolbox)
Library Link	mtransmission/Grouped Unit Delay (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
Library Link	mtransmission/Torque Converter/Torque Conversion (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
Model Reference	mtransmission/transmission ratio (show)	\$projectroot/mtransmission_ratio.mdl (open)	(not in a toolbox)
MATLAB S-Function	mtransmission_ratio/CrankSpeedSmoothing (show)	\$projectroot/crank_detect.m (open)	(not in a toolbox)

Orphaned base workspace variables

Use the table below to determine what base workspace variables the model requires, that are not defined in a file in this Manifest

Click on a column header to sort the table

Variable Name	Class	Reference Location
manual	logical	mpowertrain/Model Variants (show)
trans_type_auto	Simulink.Variant	mpowertrain/Model Variants (show)
trans_type_manual	Simulink.Variant	mpowertrain/Model Variants (show)

Warnings generated while analyzing MATLAB code

(No warnings were generated)

Dependency analysis settings:

- Detect orphaned workspace variables: **true**
- Find model references: **true**
- Find library links: **true**
- Allow models with unsaved changes to be analyzed: **false**
- Find S-functions: **true**
- Analyze model and block callbacks: **true**
- Find code-generation files: **true**
- Find data files: **true**
- Analyze Stateflow charts: **true**
- Analyze Embedded MATLAB code: **true**
- Find Requirements documents: **false**
- Analyze files in user-defined toolboxes: **true**
- Analyze MATLAB files: **true**
- Reporting of file dependence locations: **user files only**
- Store warnings: **true**

Simulating Dynamic Systems

- Chapter 14, “Running Simulations”
- Chapter 15, “Running a Simulation Programmatically”
- Chapter 16, “Visualizing and Comparing Simulation Results”
- Chapter 17, “Inspecting and Comparing Logged Signal Data”
- Chapter 18, “Analyzing Simulation Results”
- Chapter 19, “Improving Simulation Performance and Accuracy”
- Chapter 20, “Performance Advisor”
- Chapter 21, “Simulink Debugger”
- Chapter 22, “Accelerating Models”

Running Simulations

- “Simulation Basics” on page 14-2
- “Control Execution of a Simulation” on page 14-3
- “Specify Simulation Start and Stop Time” on page 14-8
- “Choose a Solver” on page 14-9
- “Interact with a Running Simulation” on page 14-31
- “Save and Restore Simulation State as SimState” on page 14-32
- “Diagnose Simulation Errors” on page 14-39

Simulation Basics

You can simulate a model at any time simply by clicking the **Run** button on the Model Editor displaying the model. See “Start a Simulation” on page 14-3.

However, before starting the simulation, you might want to specify various simulation options, such as the simulation’s start and stop time and the type of solver used to solve the model at each simulation time step.

With Simulink software, you can create multiple model configurations, called configuration sets, modify existing configuration sets, and switch configuration sets with a click of a mouse button (see “Configuration Sets” for information on creating and selecting configuration sets).

Once you have defined or selected a model configuration set that meets your needs, you can start the simulation. The simulation runs from the specified start time to the specified stop time. While the simulation is running, you can interact with the simulation in various ways, stop or pause the simulation (see “Pause or Stop a Simulation” on page 14-5), and launch simulations of other models.

If an error occurs during a simulation, Simulink halts the simulation and the Simulation Diagnostics Viewer pops up that helps you to determine the cause of the error.

Control Execution of a Simulation

In this section...

“Start a Simulation” on page 14-3

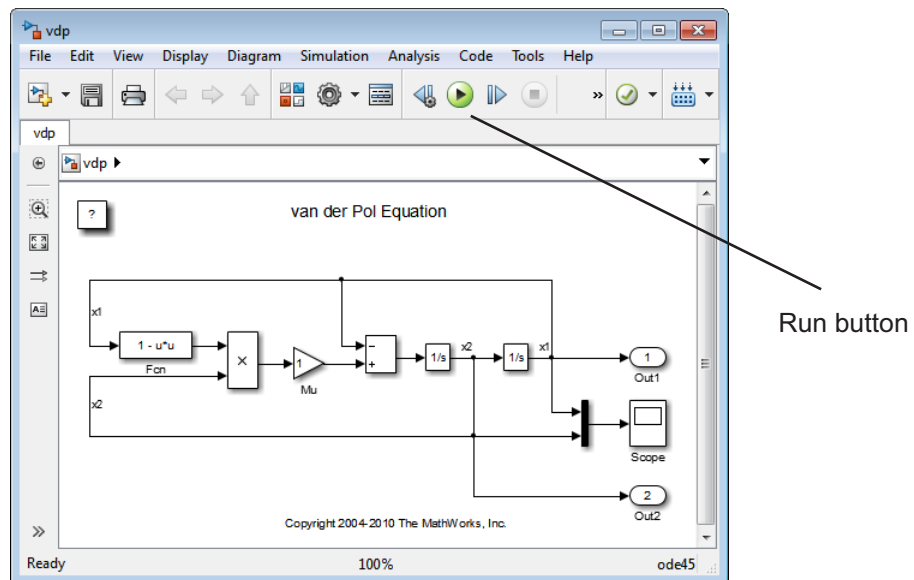
“Pause or Stop a Simulation” on page 14-5

“Use Blocks to Stop or Pause a Simulation” on page 14-5

Start a Simulation

This section explains how to run a simulation interactively using this model. See “Run Simulation Using the `sim` Command” on page 15-3 and “Control Simulation Using the `set_param` Command” on page 15-8 for information on running a simulation from a program, an S-function, or the MATLAB command line.

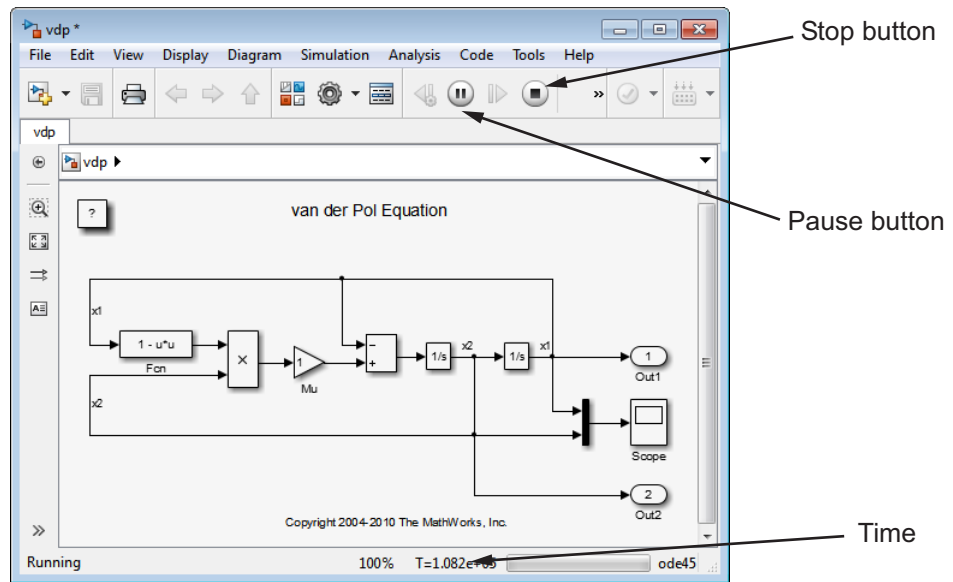
To start the execution of a model, from the **Simulation** menu of the Model Editor, select **Run** or click the **Run** button on the model toolbar.



Note A common mistake is to start a simulation while the Simulink block library is the active window. Make sure that your model window is the active window before starting a simulation.

The model execution begins at the start time that you specify on the Configuration Parameters dialog box. Execution continues until an error occurs, until you pause or terminate the simulation, or until the simulation reaches the stop time as specified on the Configuration Parameters dialog box.

While the simulation is running, a progress bar at the bottom of the model window shows how far the simulation has progressed. A **Pause** command replaces the **Run** command on the **Simulation** menu. A **Pause** command appears on the menu and replaces the **Run** button on the model toolbar.



Your computer beeps to signal the completion of the simulation.

Pause or Stop a Simulation

Select the **Pause** command or button to pause the simulation. Once Simulink completes the execution of the current time step, it suspends the simulation. When you select **Pause**, the menu item and the button change to **Continue**. (The button has the same appearance as the **Run** button). You can resume a suspended simulation at the next time step by choosing **Continue**.

To terminate execution of the model, select the **Stop** button or the **Stop Simulation** menu item. Simulink completes the execution of the current time step and then terminates the simulation. Subsequently selecting the **Run** menu item or button restarts the simulation at the first time step specified on the Configuration Parameters dialog box.

If the model includes any blocks that write output to a file or to the workspace, or if you select output options on the Configuration Parameters dialog box, the Simulink software writes the data when the simulation is terminated or suspended.

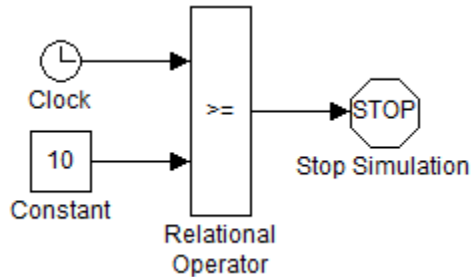
Use Blocks to Stop or Pause a Simulation

Using Stop Blocks

You can use the Stop Simulation block to terminate a simulation when the input to the block is nonzero. To use the Stop Simulation block:

- 1 Drag a copy of the Stop Simulation block from the Sinks library and drop it into 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 input signal reaches 10.



If the block input is a vector, any nonzero element causes the simulation to terminate.

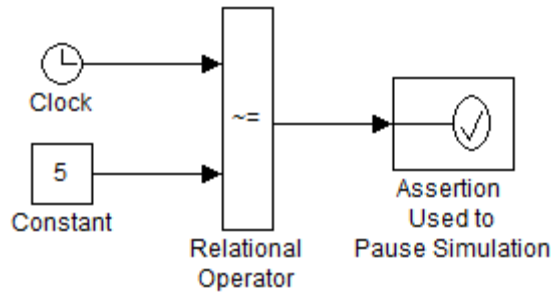
Creating Pause Blocks

You can use an Assertion block to pause the simulation when the input signal to the block is zero. To create a pause block:

- 1 Drag a copy of the Assertion block from the Model Verification library and drop it into your model.
- 2 Connect the Assertion block to a signal whose value becomes zero at the desired pause time.
- 3 Open the Block Parameters dialog box of the Assertion block .
 - Enter the following commands into the **Simulation callback when assertion fails** field:


```
set_param(bdroot, 'SimulationCommand', 'pause'),
disp(sprintf('\nSimulation paused.'))
```
 - Uncheck the **Stop simulation when assertion fails** option.
- 4 Click **OK** to apply the changes and close this dialog box.

The following model uses a similarly configured Assertion block, in conjunction with the Relational Operator block, to pause the simulation when the simulation time reaches 5.



When the simulation pauses, the Assertion block displays the following message at the MATLAB command line.

```
Simulation paused
Warning: Assertion detected in 'assertion_as_pause/
Assertion Used to Pause Simulation' at time 5.000000
```

You can resume the suspended simulation by choosing **Continue** from the **Simulation** menu on the model editor, or by selecting the **Continue** button in the toolbar.

Note The Assertion block uses the `set_param` command to pause the simulation. See “Control Simulation Using the `set_param` Command” on page 15-8 for more information on using the `set_param` command to control the execution of a Simulink model.

Specify Simulation Start and Stop Time

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, you must scale all 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. On computers running the Microsoft Windows operating system, you can also specify the simulation stop time in the **Simulation** menu.

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.

Choose a Solver

In this section...
“What Is a Solver?” on page 14-9
“Choosing a Solver Type” on page 14-10
“Choosing a Fixed-Step Solver” on page 14-13
“Choosing a Variable-Step Solver” on page 14-17
“Choosing a Jacobian Method for an Implicit Solver” on page 14-24

What Is a Solver?

A solver is a component of the Simulink software. The Simulink product provides an extensive library of solvers, each of which determines the time of the next simulation step and applies a numerical method to solve the set of ordinary differential equations that represent the model. In the process of solving this initial value problem, the solver also satisfies the accuracy requirements that you specify. To help you choose the solver best suited for your application, “Choosing a Solver Type” on page 14-10 provides background on the different types of solvers while “Choosing a Fixed-Step Solver” on page 14-13 and “Choosing a Variable-Step Solver” on page 14-17 provide guidance on choosing a specific fixed-step or variable-step solver, respectively.

The following table summarizes the types of solvers in the Simulink library and provides links to specific categories. All of these solvers can work with the algebraic loop solver.

		Discrete	Continuous	Variable-Order
Fixed-Step	Explicit	Not Applicable	“Explicit Fixed-Step Continuous Solvers” on page 14-14	Not Applicable
	Implicit	Not Applicable	“Implicit Fixed-Step Continuous Solvers” on page 14-16	Not Applicable

		Discrete	Continuous	Variable-Order
Variable-Step	Explicit	“Choosing a Variable-Step Solver” on page 14-17	“Explicit Continuous Variable-Step Solvers” on page 14-18	“Variable-Order Solvers” on page 14-13
	Implicit		“Implicit Continuous Variable-Step Solvers” on page 14-19	“Variable-Order Solvers” on page 14-13

Note

- 1** The fixed-step discrete solvers do not solve for discrete states; each block calculates its discrete states independent of the solver. For more information, see “Discrete versus Continuous Solvers” on page 14-12
 - 2** Every solver in the Simulink library can perform on models that contain algebraic loops.
-

For information on tailoring the selected solver to your model, see “Improve Simulation Accuracy” on page 19-11

Choosing a Solver Type

The Simulink library of solvers is divided into two major types in the “Solver Pane”: fixed-step and variable-step. You can further divide the solvers within each of these categories as: discrete or continuous, explicit or implicit, one-step or multistep, and single-order or variable-order.

Fixed-Step versus Variable-Step Solvers

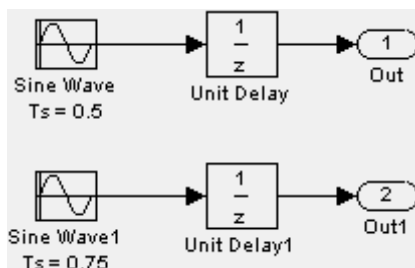
Both fixed-step and variable-step solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the *step size*. With a fixed-step solver, the step size remains constant throughout the simulation. In contrast, with a variable-step solver, the step size can 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 **Type** control on the Simulink **Solver** configuration pane allows you to select either of these two types of solvers.

The choice between the two 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 because 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 savings because, for a given level of accuracy, the solver can dynamically adjust the step size as necessary and thus reduce the number of steps. Whereas the fixed-step solver must use a single step size throughout the simulation based upon the accuracy requirements. To satisfy these requirements throughout the simulation, the fixed-step solver might require a very small step.

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



This 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 needs 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.

If you wish to achieve evenly spaced steps, you must use the format `0.4*[0.0:100.0]` rather than `[0.0:0.4:40]`.

Discrete versus Continuous Solvers

When you set the **Type** control of the **Solver** configuration pane to **fixed-step** or to **variable-step**, the adjacent **Solver** control allows you to choose a specific solver. Both sets of solvers comprise two types: discrete and continuous. 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. Therefore, when choosing a solver, you must first determine whether you need 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 instead your model has continuous states, you must choose a continuous solver from the remaining solver choices based on the dynamics of your model. Otherwise, an error occurs.

Explicit versus Implicit Solvers

While you can apply either an implicit or explicit continuous solver, the implicit solvers are designed specifically for solving stiff problems whereas explicit solvers are used to solve nonstiff problems. A generally accepted definition of a stiff system is a system that has extremely different time scales. Compared to the explicit solvers, the implicit solvers provide greater stability for oscillatory behavior, but they 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 “Choosing a Jacobian Method for an Implicit Solver” on page 14-24 for more information.

One-Step versus Multistep 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 a number of points between t_n and t_{n-1} . These points are *minor steps*.

The 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.

Variable-Order Solvers

Two variable-order solvers, ode15s and ode113, are part of the solver library. These solvers 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. For ode15s, you can limit the highest order applied via the Maximum Order parameter. For more information, see “Maximum Order” on page 14-20.

Choosing a Fixed-Step Solver

About the 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. You can allow the Simulink software to choose the size of the step (the default) 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.

Note If you try to use the fixed-step discrete solver to update or simulate a model that has continuous states, an error message appears. Thus, selecting a fixed-step solver and then updating or simulating a model is a quick way to determine whether the model has continuous states.

About 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. The fixed-step continuous solvers can, therefore, handle models that contain both continuous and discrete states.

Note In theory, a fixed-step continuous solver can handle models that contain no continuous states. However, that would impose an unnecessary computational burden on the simulation. Consequently, 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.

Two types of fixed-step continuous solvers that Simulink provides are: explicit and implicit. (See “Explicit versus Implicit Solvers” on page 14-12 for more information). 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.

Explicit Fixed-Step Continuous 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. Expressed mathematically for a fixed-step explicit solver:

$$x(n + 1) = x(n) + h * Dx(n)$$

where x is the state, Dx is a solver-dependent function that estimates the state derivative, h is the step size, and n indicates the current time step.

Simulink provides a set of explicit fixed-step continuous solvers. The solvers differ in the specific numerical integration technique that they use to compute the state derivatives of the model. The following table lists each solver and the integration technique it uses.

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

The table lists the solvers in order of the computational complexity of the integration methods they use, from the least complex (ode1) to the most complex (ode8).

None of these solvers has an error control mechanism. Therefore, the accuracy and the duration of a simulation depends 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 more computationally complex the solver is, the more accurate are the simulation results.

If you specify a fixed-step solver type for a model, then by default, Simulink selects the ode3 solver, which 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 might need to choose another solver, a different fixed step size, or both to achieve acceptable accuracy and an acceptable simulation time.

Implicit Fixed-Step Continuous Solvers. An implicit fixed-step 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. In other words:

$$x(n+1) - x(n) - h * Dx(n+1) = 0$$

Simulink provides one implicit fixed-step 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.

Process for Choosing a Fixed-Step Continuous Solver

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 generally is not possible, or at least not practical, to decide *a priori* which combination of solver and step size will yield acceptable results for the continuous states in the shortest time. Determining the best solver for a particular model generally requires experimentation.

Following is the most efficient way to choose the best fixed-step solver for your model experimentally.

- 1** Choose error tolerances. For more information, see "Specifying Error Tolerances for Variable-Step Solvers" on page 14-22.
- 2** Use one of the variable-step solvers to simulate your model to the level of accuracy that you desire. Start with `ode45`. If your model runs slowly, your problem might be stiff and need an implicit solver. The results of this

step give a good approximation of the correct simulation results and the appropriate fixed step size.

- 3** Use `ode1` to simulate your model at the default step size for your model. Compare the simulation results for `ode1` with the simulation for the variable-step solver. If the results are the same for the specified level of accuracy, you have found the best fixed-step solver for your model, namely `ode1`. You can draw this conclusion because `ode1` is the simplest of the fixed-step solvers and hence yields the shortest simulation time for the current step size.
- 4** If `ode1` does not give satisfactory results, repeat the preceding steps with each of the other fixed-step solvers until you find the one that gives accurate results with the least computational effort. The most efficient way to perform this task is to use a binary search technique:
 - a** Try `ode3`.
 - b** If `ode3` gives accurate results, try `ode2`. If `ode2` gives accurate results, it is the best solver for your model; otherwise, `ode3` is the best.
 - c** If `ode3` does not give accurate results, try `ode5`. If `ode5` gives accurate results, try `ode4`. If `ode4` gives accurate results, select it as the solver for your model; otherwise, select `ode5`.
 - d** If `ode5` does not give accurate results, reduce the simulation step size and repeat the preceding process. Continue in this way until you find a solver that solves your model accurately with the least computational effort.

Choosing a Variable-Step Solver

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 subset 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. In fact, 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. 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.

About Variable-Step Continuous Solvers

The variable-step solvers in the Simulink product 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 but 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 “Choosing a Solver Type” on page 14-10 for more information.)

Explicit Continuous Variable-Step Solvers

The explicit variable-step solvers are designed for nonstiff problems. Simulink provides three such solvers: ode45, ode23, and ode113.

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
ode113		X	Variable, Low to High	PECE Implementation of Adams-Bashforth-Moulton

ODE Solver	Tips on When to Use
ode45	<p>In general, the ode45 solver is the best to apply as a first try for most problems. For this reason, ode45 is the default solver for models with continuous states. This 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 “free” interpolant, which allows for event location and smoother plots.</p> <p>The ode45 is more accurate and faster than ode23. If the ode45 is slow computationally, your problem may be stiff and thus in need of an implicit solver.</p>
ode23	<p>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 for “free” by applying a cubic Hermite interpolation to the values and slopes computed at the ends of a step.</p>
ode113	<p>For problems with stringent error tolerances or for computationally intensive problems, the Adams-Bashforth-Moulton PECE solver can be more efficient than ode45.</p>

Implicit Continuous Variable-Step Solvers

If your problem is stiff, try using one of the implicit variable-step solvers: ode15s, ode23s, ode23t, or 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

ODE Solver	One-Step Method	Multistep Method	Order of Accuracy	Solver Reset Method	Max. Order	Method
ode23t	X		Low	X		Trapezoidal rule using a “free” interpolant
ode23tb	X		Low	X		TR-BDF2

Solver Reset Method. For three of the stiff solvers — ode15s, ode23t, and ode23tb— a drop-down menu for the Solver reset method appears on the Solver 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. The former setting specifies that the solver does not recompute the Jacobian for a solver reset, whereas the latter setting specifies that the solver does. Consequently, the Fast setting is faster computationally but might use a small step size for certain cases. To test for such cases, run the simulation with each setting and compare the results. If there is no difference, 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 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 relating to the application of variable-step implicit solvers. For an example comparing the behavior of these solvers, see sldemo_solvers.

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.)
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.

Support for Zero-Crossing Detection

Both the variable-step discrete and continuous solvers use zero-crossing detection (see “Zero-Crossing Detection” on page 3-24) to handle continuous signals.

Specifying Error Tolerances for Variable-Step Solvers

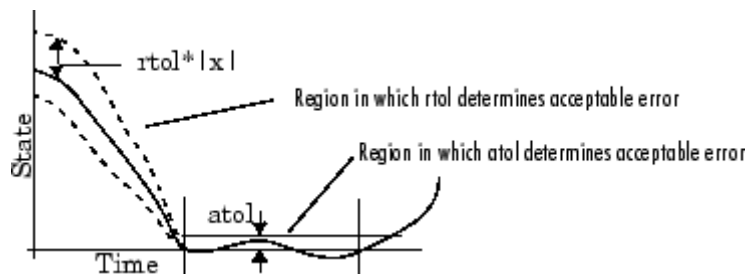
Local Error. The variable-step solvers use standard control techniques to monitor the local error at each time step. During each time step, the solvers compute the state values at the end of the step and determine the *local error*—the estimated error of these state values. They then compare the local 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.

- The *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:

$$e_i \leq \max(\text{rtol} \times |x_i|, \text{atol}_i).$$

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 to $1e-6$. 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 $1e-3$, then by the end of the simulation, `abstol` becomes $1e-3$ also. If a state goes from 0 to 1000, then `abstol` changes to 1.

If the computed setting is not suitable, you can determine an appropriate setting yourself. You might have to run a simulation more than once to determine an appropriate value for the 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 Limited
- 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. The block absolute tolerance can be set to:

- `auto`
- `-1` (same as `auto`)
- `real 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 choose too high of an absolute tolerance, 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.

Choosing a Jacobian Method for an Implicit Solver

About the Solver Jacobian

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:

$$\begin{aligned}\dot{x} &= f(x, t, u) \\ y &= g(x, t, u).\end{aligned}$$

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:

$$\dot{x}_1 = f_1(x_1, x_3)$$

$$\dot{x}_2 = f_2(x_2)$$

$$\dot{x}_3 = f_3(x_2).$$

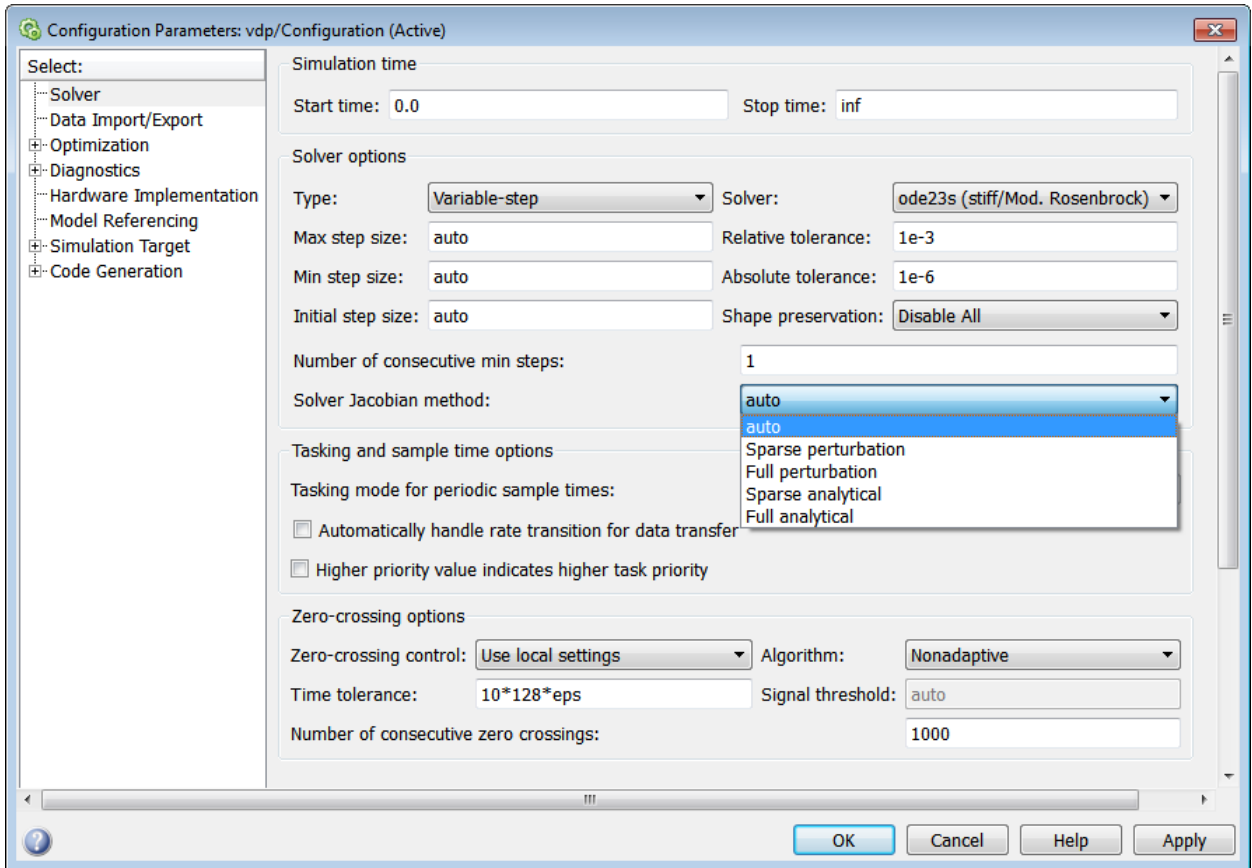
From this system, you can derive a sparsity pattern that reflects the structure of the equations. The pattern, a Boolean matrix, has a 1 for each x_i that appears explicitly on the right-hand side of an equation. You thereby attain:

$$J_{x,pattern} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

As discussed in “Full and Sparse Perturbation Methods” on page 14-28 and “Full and Sparse Analytical Methods” on page 14-30 respectively, the Sparse Perturbation Method and the Sparse Analytical Method may be able to take advantage of this sparsity pattern to reduce the number of computations necessary and thereby 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: **auto**, **Sparse perturbation**, **Full perturbation**, **Sparse analytical**, and **Full analytical**.



Note If you set **Automatic solver parameter selection** to warning or error in the Solver Diagnostics pane, and you choose a different solver method than Simulink, you might receive a warning or an error.

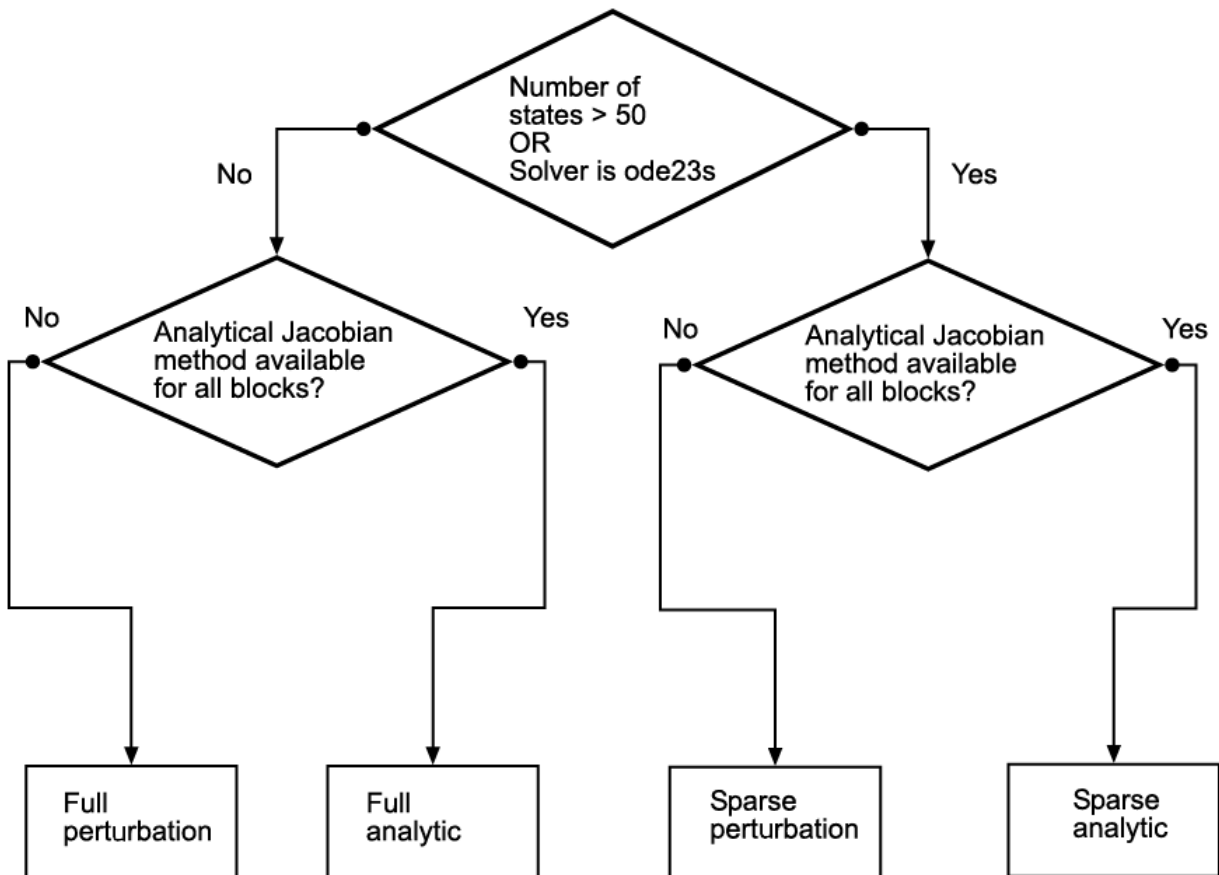
Limitations. The solver Jacobian methods have the following limitations associated with them.

- 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 perform a heuristic to determine which of the remaining four methods best suits your model. This algorithm is depicted in the following flowchart.



Because the sparse methods are beneficial for models having a large number of states, the heuristic chooses a sparse method if more than 50 states exist in your model. The logic also leads to a sparse method if you specify `ode23s` because, unlike other implicit solvers, `ode23s` generates a new Jacobian at every time step. A sparse analytical or a sparse perturbation method is, therefore, highly advantageous. The heuristic 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 was the standard numerical method that Simulink used to solve a system. For this method, Simulink solves the full set of perturbation equations and uses LAPACK for linear algebraic operations. This method is costly from a computational standpoint, but it remains the 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,

$$\begin{aligned}\dot{x}_1 &= f_1(x_1, x_3) \\ \dot{x}_2 &= f_2(x_2) \\ \dot{x}_3 &= f_3(x_2).\end{aligned}$$

The solver Jacobian is:

$$\begin{aligned}
 J_x &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{pmatrix} \\
 &= \begin{pmatrix} \frac{f_1(x_1 + \Delta x_1, x_2, x_3) - f_1}{\Delta x_1} & \frac{f_1(x_1, x_2 + \Delta x_2, x_3) - f_1}{\Delta x_2} & \frac{f_1(x_1, x_2, x_3 + \Delta x_3) - f_1}{\Delta x_3} \\ \frac{f_2(x_1 + \Delta x_1, x_2, x_3) - f_2}{\Delta x_1} & \frac{f_2(x_1, x_2 + \Delta x_2, x_3) - f_2}{\Delta x_2} & \frac{f_2(x_1, x_2, x_3 + \Delta x_3) - f_2}{\Delta x_3} \\ \frac{f_3(x_1 + \Delta x_1, x_2, x_3) - f_3}{\Delta x_1} & \frac{f_3(x_1, x_2 + \Delta x_2, x_3) - f_3}{\Delta x_2} & \frac{f_3(x_1, x_2, x_3 + \Delta x_3) - f_3}{\Delta x_3} \end{pmatrix}
 \end{aligned}$$

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:

$$= \begin{pmatrix} \frac{f_1(x_1 + \Delta x_1, x_2 + \Delta x_2, x_3) - f_1}{\Delta x_1} & 0 & \frac{f_1(x_1, x_2, x_3 + \Delta x_3) - f_1}{\Delta x_3} \\ 0 & \frac{f_2(x_1 + \Delta x_1, x_2 + \Delta x_2, x_3) - f_2}{\Delta x_2} & 0 \\ 0 & \frac{f_3(x_1 + \Delta x_1, x_2 + \Delta x_2, x_3) - f_3}{\Delta x_2} & 0 \end{pmatrix}$$

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.

Sparsity Pattern

For details on how to access and interpret the sparsity pattern in MATLAB, see `sldemo_metro`.

Support for Code Generation

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.

Interact with a Running Simulation

You can perform certain operations interactively while a simulation is running. You can do the following:

- Modify some configuration parameters, including the stop time and the maximum step size
- Click a line to see the signal carried on that line on a floating (unconnected) Scope or Display block
- Modify the parameters of a block, as long as you do not cause a change in the:
 - 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
- Display block data tips (see “Block Tool Tips” on page 23-2)

You cannot make changes to the structure of the model, such as adding or deleting lines or blocks, during a simulation. To make these kinds of changes, stop the simulation, make the change, then start the simulation again to see the results of the change.

Save and Restore Simulation State as SimState

In this section...

“Overview of the SimState” on page 14-32

“Save the SimState” on page 14-33

“Restore the SimState” on page 14-35

“Change the States of a Block within the SimState” on page 14-37

“SimState Interface Checksum Diagnostic” on page 14-37

“Limitations of the SimState” on page 14-38

“Using SimState within S-Functions” on page 14-38

Overview of the SimState

In real-world applications, you simulate a Simulink model repeatedly to analyze the behavior of a system for different input, boundary conditions, or operating conditions. In many applications, a start-up phase with significant dynamic behavior is common to multiple simulations. For example, the cold start take-off of a gas turbine engine occurs before each set of aircraft maneuvers. Ideally, you would simulate this start-up phase once, save the simulation state at the end of the start-up phase, and then use this simulation state or *SimState* as the initial state for each set of conditions or maneuvers.

The Simulink `SimState` feature allows you to save all run-time data necessary for restoring the simulation state of a model. A `SimState` includes both the logged and internal state of every block (e.g., continuous states, discrete states, work vectors, zero-crossing states) and the internal state of the Simulink engine (e.g., the data of the ODE solver).

You can save a `SimState`:

- At the final **Stop time**
- When you interrupt a simulation with the **Pause** or **Stop** button
- When you use a block (e.g., the Stop block) to stop a simulation

At a later time, you can restore the `SimState` and use it as the initial conditions for any number of simulations.

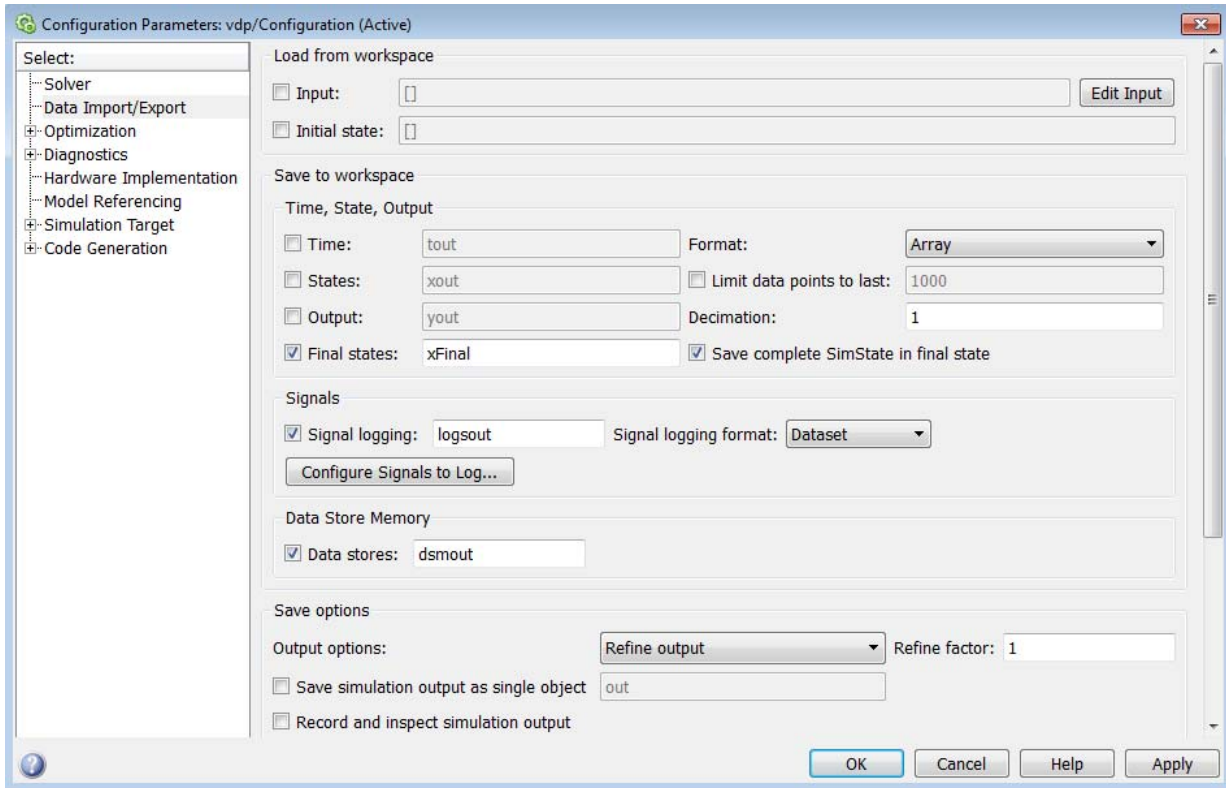
Note The **Final states** option of the **Data Import/Export** pane in Simulink saves only *logged* states—the continuous and discrete states of blocks—which are a subset of the complete simulation state of the model. Hence you cannot use the **Final states** to save and restore the complete simulation state as the initial state of a new simulation.

Save the SimState

Saving the SimState Interactively

To save the complete `SimState` at the beginning of the final step:

- 1** In the Simulink model window, select **Simulation > Configuration Parameters**.
- 2** Navigate to the **Data Import/Export** pane.
- 3** Select the **Final states** check box. The **Save complete SimState in final state** check box becomes active.
- 4** Select the **Save complete SimState in final state** check box.
- 5** In the adjacent field, enter a variable name for the `SimState`.
- 6** Simulate the model by selecting **Simulation > Run**.



Saving the SimState Programmatically

You can save the SimState at the beginning of the final step programmatically using the `sim` command in conjunction with the `set_param` command with the `SaveCompleteFinalSimState` parameter set to `on`:

```
set_param mdl, 'SaveFinalState', 'on', 'FinalStateName', ...
    [mdl 'SimState'], 'SaveCompleteFinalSimState', 'on')
simOut = sim mdl, 'StopTime', tstop)
set_param mdl, 'SaveFinalState', 'off')
```

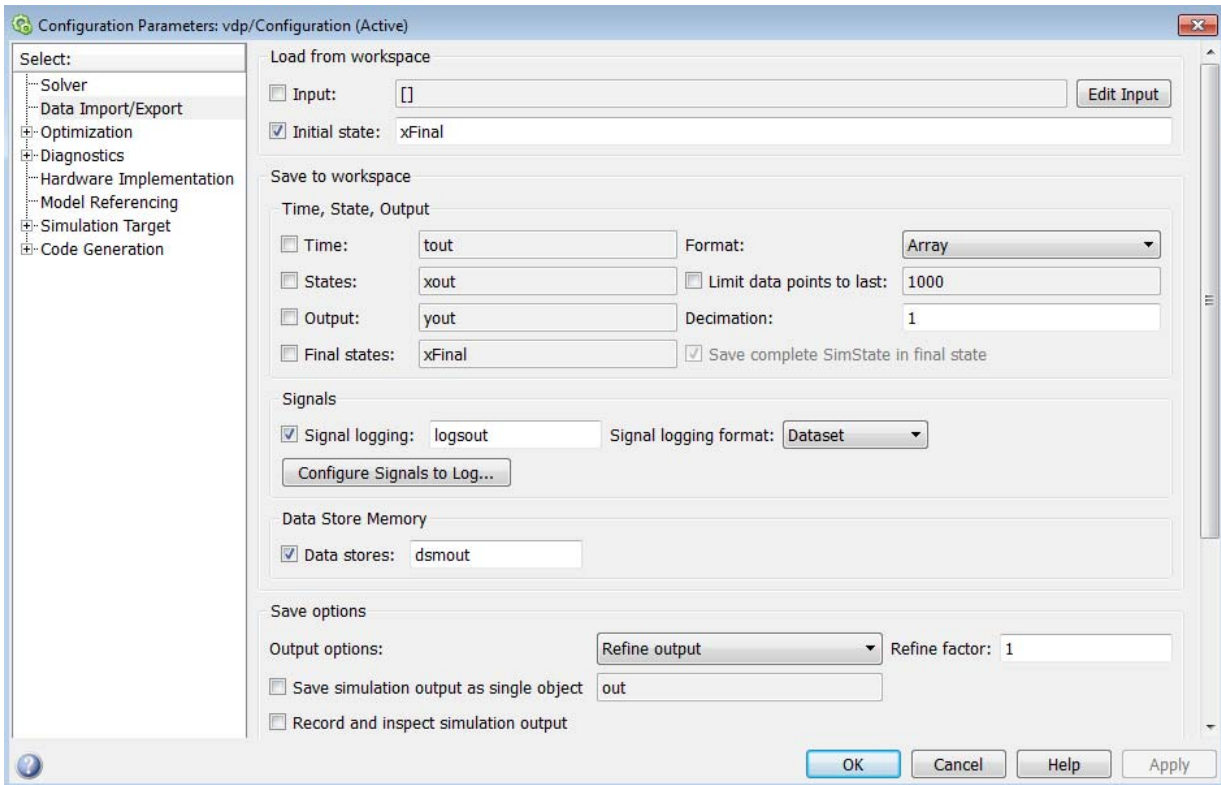
Restore the SimState

Restoring the SimState Interactively

You can restore the SimState interactively.

In the **Data Import/Export** pane:

- 1 Under **Load from workspace**, select the check box next to **Initial state**. The adjacent field becomes active.
- 2 Enter the name of the variable containing the SimState in the field.



In the **Solver** pane of the **Configuration Parameters** window:

- 1 Keep the **Start time** set to its original value.
- 2 Set the **Stop time** to the sum of the original simulation time plus the new additional simulation time.
- 3 Click **OK**.

The **Start time** must maintain its original value because it is a reference value for all time and time-dependent variables in both the original and the current simulations. For example, a block may save and restore the number of sample time hits that occurred since the beginning of simulation as the `SimState`. For clarity, consider a model that you ran from 0 s to 100 s and that you now wish to run from 100 s to 200 s. The **Start time** is 0 s for both the original simulation (0 s to 100 s) and for the current simulation (100 s to 200 s). And 100 s is the initial time of the current simulation. Also, if the block had ten sample time hits during the original simulation, Simulink recognizes that the next sample time hit will be the eleventh relevant to 0 s (not 100 s).

Restoring the `SimState` Programmatically

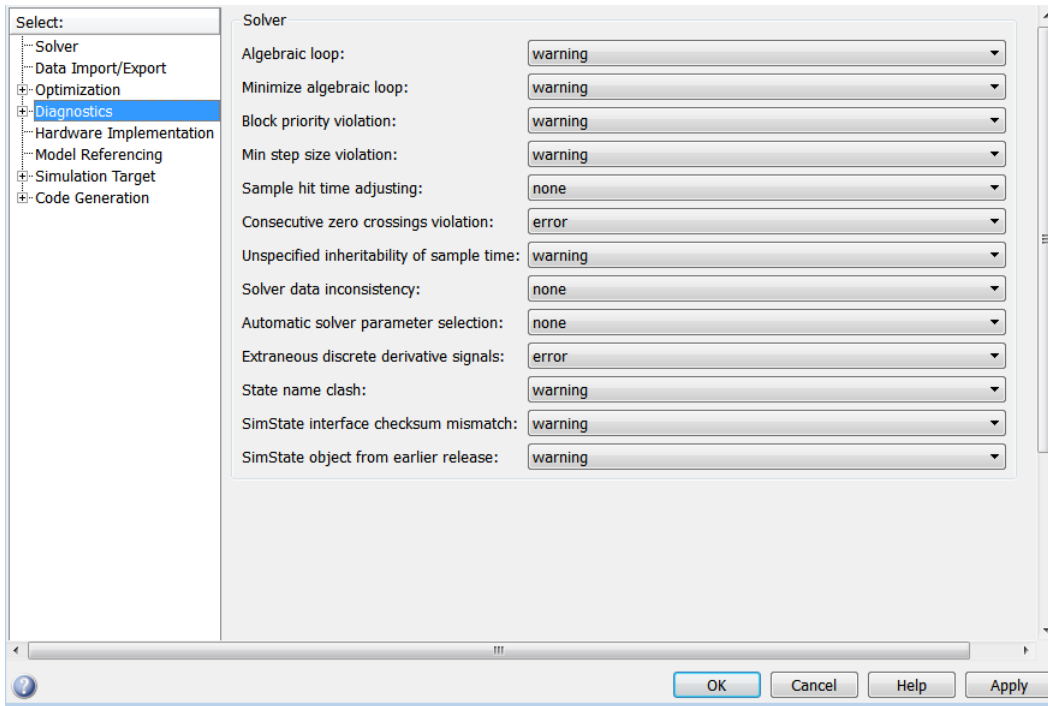
Use the `set_param` command to specify the initial condition as the `SimState`.

```
set_param mdl, 'LoadInitialState', 'on', 'InitialState', ...  
[mdl 'SimState']];  
simOut = sim(mdl, 'StopTime', tstop)
```

Restoring a `SimState` Saved in an Earlier Version of Simulink

You can now use `SimState` objects saved in earlier releases (R2010a or later) to restore the `SimState` of a model. You can see the version of Simulink used to save the `SimState` by examining the `'version'` field of the `SimState` object. Simulink detects if the `SimState` object provided as the initial state was saved in the current release. By default, the Simulink software reports an error message if the `SimState` was not saved in the current release. You may configure the diagnostic to allow Simulink to report the message as a warning and try to restore as many of the values as possible. To enable this best-effort restoration, go to the Diagnostics pane of the Configuration Parameter dialog box and set the message for **SimState object from earlier release** to warning. You can also set the diagnostic programmatically using the `set_param` command.

```
set_param mdl, 'SimStateOlderReleaseMsg', 'warning');
```



Change the States of a Block within the SimState

You can use the `loggedStates` to get or set the `SimState`. The `loggedStates` field has the same structure as `xout.signals` if `xout` is the state log that Simulink exports to the workspace.

SimState Interface Checksum Diagnostic

The `SimState` interface checksum is primarily based upon the configuration settings of your model. A diagnostic, 'SimState interface checksum mismatch', resides on the Diagnostics pane of the Configuration Parameters dialog box. You can set this diagnostic to 'none', 'warning', or 'error' to receive a warning or an error if the interface checksum of the restored `SimState` does not match the current interface checksum of the model. Such mismatches may occur when you try to simulate using a solver that is different from the one that

generated the saved `SimState`. Simulink permits such solver changes. For example, you can use a solver such as `ode15s`, to solve the initial stiff portion of a simulation, save the final `SimState`, and then continue the simulation with the restored `SimState` and using `ode45`. In other words, this diagnostic is purely to serve your own purposes for detecting solver changes.

Limitations of the SimState

Several limitations exist for the `SimState`:

- You can use only the Normal or the Accelerator mode of simulation.
- You cannot save the `SimState` in Normal mode and restore it in Accelerator mode, or vice versa.
- You can save the `SimState` only at the final stop time or at the execution time at which you pause or stop the simulation.
- By design, Simulink does not save user data, run-time parameters, or logs of the model.
- The `SimState` feature does not support code generation, including Model Reference in accelerated modes.
- You cannot make any structural changes to the model between the time at which you save the `SimState` and the time at which you restore the simulation using the `SimState`. For example, you cannot add or remove a block after saving the `SimState` without repeating the simulation and saving the new `SimState`.
- You cannot input the `SimState` to model functions.

Using SimState within S-Functions

Special APIs for C and Level-2 MATLAB S-functions are available, which enable the S-functions to work with the `SimState`. For information on how to implement these APIs within S-functions, see “S-Function Compliance with the `SimState`”.

Diagnose Simulation Errors

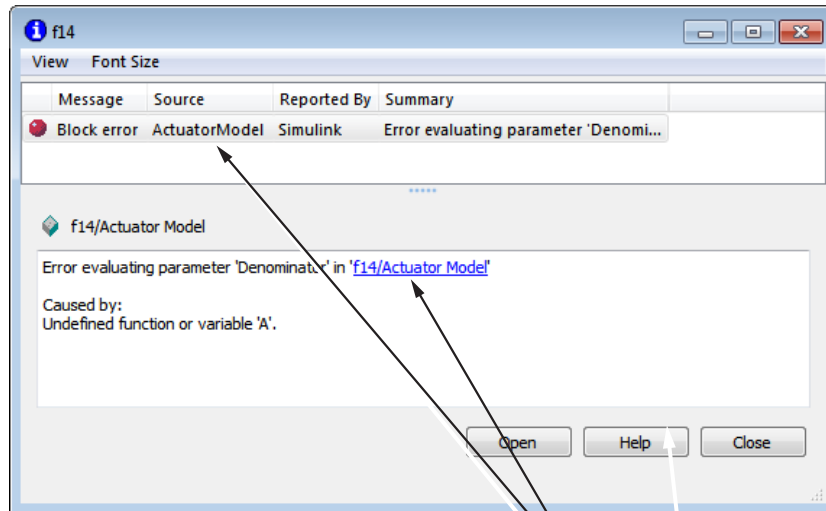
In this section...
“Response to Run-Time Errors” on page 14-39
“Simulation Diagnostics Viewer” on page 14-39
“Create Custom Simulation Error Messages” on page 14-41

Response to Run-Time Errors

If errors occur during a simulation, the Simulink software halts the simulation, opens the subsystems that caused the error (if necessary), and displays the errors in the Simulation Diagnostics Viewer. The following sections explain how to use the viewer to determine the cause of the errors, and how to create custom error messages.

Simulation Diagnostics Viewer

The viewer comprises an Error Summary pane and an Error Message pane.



Click to display error source.

Error Summary Pane

The upper pane lists the errors that caused the simulation to terminate. The pane displays the following information for each error.

Message. Message type (for example, block error, warning, or log)

Source. Name of the model element (for example, a block) that caused the error

Reported By. Component that reported the error (for example, the Simulink product, the Stateflow product, or the Simulink Coder product).

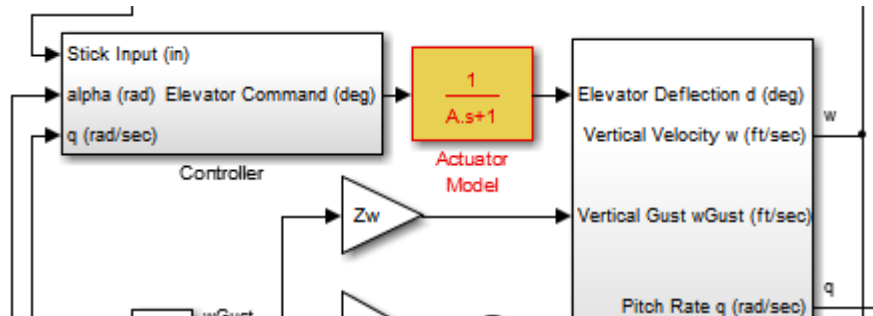
Summary. Error message, abbreviated to fit in the column

You can remove any of these columns of information to make room for other columns. To remove a column, select the **View** menu and uncheck the corresponding item.

Error Message Pane

The lower pane initially contains the contents of the first error message listed in the top pane. You can display the contents of other messages by clicking their entries in the upper pane.

In addition to displaying the viewer, the Simulink software opens (if necessary) the subsystem that contains the first error source and highlights the source.



You can display the sources of other errors by clicking: anywhere in the error message in the upper pane; the name of the error source in the error message (highlighted in blue); or the **Open** button on the viewer.

Changing Font Size

To change the size of the font used to display errors, select **Increase Font Size** or **Decrease Font Size** from the **Font Size** menu of the viewer.

Create Custom Simulation Error Messages

The Simulation Diagnostics Viewer displays the output of any instance of the MATLAB error function executed during a simulation. Such instances include those invoked by block or model callbacks, or by S-functions that you create or that the MATLAB Function block executes.

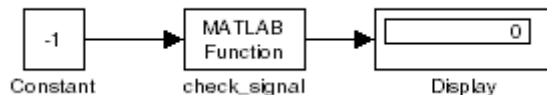
You can use the MATLAB error function in callbacks, S-functions or the MATLAB Function block to create custom error messages specific to your application. Capabilities available for messages include:

- Display the contents of a text string
- Include hyperlinks to an object
- Link to an HTML file

Displaying A Text String

To display the contents of a text string, pass the string enclosed by quotation marks to the error function.

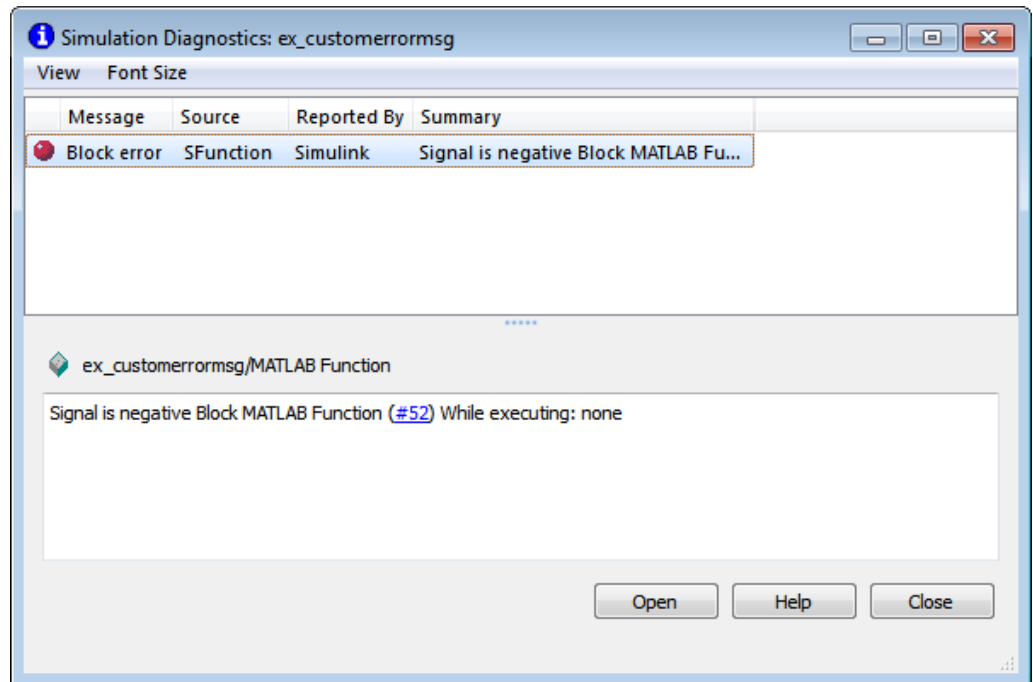
The following example shows how you can make the user-created function `check_signal` display the string `Signal is negative`.



The MATLAB Function block invokes the following function:

```
function y=check_signal(x)
    if x<0
        error('Signal is negative');
    else
        y=x;
    end
end
```

Executing this model causes a runtime error and starts the debugger when you click the **OK** button of the runtime error message. When you quit the debugger, an error message in the Simulation Diagnostics window.



Creating Hyperlinks to Files, Directories, or Blocks

To include a hyperlink to a block, file, or folder in the error message, include the path to the item enclosed in quotation marks.

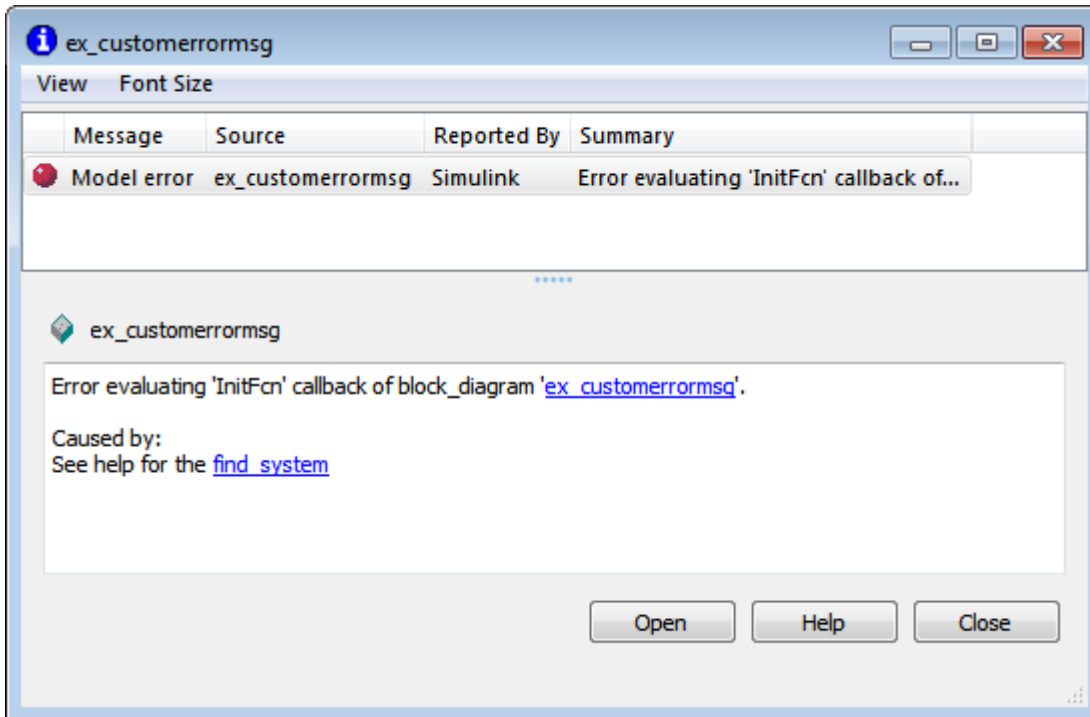
- `error ('Error evaluating parameter in block "mymodel/Mu"')`
displays a text hyperlink to the block Mu in the model “mymodel”. Clicking the hyperlink displays the block in the model window.
- `error ('Error reading data from "c:/work/test.data"')`
displays a text hyperlink to the file test.data in the error message. Clicking the link displays the file in your preferred MATLAB editor.
- `error ('Could not find data in folder "c:/work"')`
displays a text hyperlink to the c:/work folder. Clicking the link opens a system command window (shell) and sets its working folder to c:/work.

Creating Programmatic Hyperlinks

You can create a hyperlink that, when clicked, causes the evaluation of a MATLAB expression. For example, the following model `InitFcn` callback displays an error, when the model starts, with a hyperlink to help for the `find_system` command.

```
error('See help for the <a href="matlab:doc  
find_system">find_system</a>
```

In this example, the Simulation Diagnostics window displays a hyperlink labeled `find_system`. Clicking the link opens the documentation for the `find_system` command in the MATLAB Help browser.



Running a Simulation Programmatically

- “About Programmatic Simulation” on page 15-2
- “Run Simulation Using the sim Command” on page 15-3
- “Control Simulation Using the set_param Command” on page 15-8
- “Run Parallel Simulations” on page 15-11
- “Error Handling in Simulink Using MSLException” on page 15-26

About Programmatic Simulation

Entering simulation commands in the MATLAB Command Window or from a MATLAB file enables you to run unattended simulations. You can perform Monte Carlo analysis by changing the parameters randomly and executing simulations in a loop. You can use either the `sim` command or the `set_param` command to run a simulation programmatically. To run simulations simultaneously, you can call `sim` from within a `parfor` loop under specific conditions.

Run Simulation Using the sim Command

In this section...

“Single-Output Syntax for the sim Command” on page 15-3

“Examples of Implementing the sim Command” on page 15-4

“Calling sim from Within parfor” on page 15-5

“Backwards Compatible Syntax” on page 15-5

Single-Output Syntax for the sim Command

The general form of the command syntax for running a simulation is:

```
SimOut = sim('model', Parameters)
```

where *model* is the name of the block diagram and *Parameters* can be a list of parameter name-value pairs, a structure containing parameter settings, or a configuration set. The `sim` command returns, `SimOut`, a single `Simulink.SimulationOutput` object that contains all of the simulation outputs (logged time, states, and signals). This syntax is the “single-output format” of the `sim` command.

```
SimOut = sim('model', 'Param1', Value1, 'Param2', Value2...);  
SimOut = sim('model', ParameterStruct);  
SimOut = sim('model', ConfigSet);
```

Note The output of the `sim` command will always be returned to `SimOut`, the single simulation output object and not to the workspace.

During simulation, the specified parameters override the values in the block diagram configuration set. The original configuration values are restored at the end of simulation. If you wish to simulate the model without overriding any parameters, and you want the simulation results returned in the single-output format, then you must do one of the following:

- select **Save simulation output as single object** on the Data Import/Export pane of the Configuration Parameters dialog box

- specify the `ReturnWorkspaceOutputs` parameter value as 'on' in the `sim` command:

```
SimOut = sim('model', 'ReturnWorkspaceOutputs', 'on');
```

To log the model time, states, or outputs, use the Configuration Parameters Data Import/Export dialog box. To specify the time span for a simulation, you must specify the `StartTime` and `StopTime` parameters. To log signals, either use a block such as the To Workspace block or the Scope block, or use the **Signal and Scope Manager** to log results directly.

To stop a simulation in progress, press **Ctrl+C**. The `sim` command does not allow you to pause a simulation in progress.

For complete details of the `sim` command syntax, see the `sim` reference page.

Examples of Implementing the `sim` Command

Following are examples that show the application of each of the three formats for specifying parameter values using the single-output format of the `sim` command.

Specifying Parameter Name-Value Pairs

In the following example, the `sim` syntax specifies the model name, `vdp`, followed by consecutive pairs of parameter name and parameter value. For example, the value of the `SimulationMode` parameter is `rapid`.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...  
            'SaveState','on','StateSaveName','xoutNew',...  
            'SaveOutput','on','OutputSaveName','youtNew');  
simOutVars = simOut.who;  
yout = simOut.get('youtNew');
```

Specifying a Parameter Structure

The following example shows how to specify parameter name-value pairs as a structure to the `sim` command.

```
paramNameValStruct.SimulationMode = 'rapid';  
paramNameValStruct.AbsTol         = '1e-5';
```



```

paramNameValStruct.SaveState      = 'on';
paramNameValStruct.StateSaveName = 'xoutNew';
paramNameValStruct.SaveOutput    = 'on';
paramNameValStruct.OutputSaveName = 'youtNew';
simOut = sim('vdp',paramNameValStruct);

```

Specifying a Configuration Set

The following example shows how to create a configuration set and use it with the `sim` syntax.

```

model = 'vdp';
load_system(model)
simMode = get_param(model, 'SimulationMode');
set_param(model, 'SimulationMode', 'rapid')
cs = getActiveConfigSet(model);
model_cs = cs.copy;
set_param(model_cs, 'AbsTol', '1e-5', ...
    'SaveState', 'on', 'StateSaveName', 'xoutNew', ...
    'SaveOutput', 'on', 'OutputSaveName', 'youtNew')
simOut = sim(model, model_cs);
set_param(model, 'SimulationMode', simMode)

```

The block diagram parameter, `SimulationMode`, is not part of the configuration set, but is associated with the model. Therefore, the `set_param` command saves and restores the original simulation mode by passing the model rather than the configuration set.

Calling sim from Within parfor

For information on how to run simultaneous simulations by calling `sim` from within `parfor`, see “Run Parallel Simulations” on page 15-11.

Backwards Compatible Syntax

The following syntax is now obsolete but will be maintained for backwards compatibility with Simulink Versions 7.3 or earlier releases.

```

[T,X,Y] =sim('model',Timespan, Options, UT)
[T,X,Y1,...,Yn] =sim('model',Timespan, Options, UT)

```

If only one right-hand side argument exists, then Simulink automatically saves the time, the state and the output to the specified left-hand side arguments. You can explicitly switch to the single-output format by changing the defaults.

If you do not specify any left-hand side arguments, then Simulink determines what data to log based on the workspace I/O settings of the Data Import/Export pane of the Configuration Parameters dialog box.

<i>T</i>	The time vector returned.
<i>X</i>	The state returned in matrix or structure format. The state matrix contains continuous states followed by discrete states.
<i>Y</i>	The output returned in matrix or structure format. For block diagram models, this variable contains all root-level blocks.
<i>Y1,...,Yn</i>	The outputs, which can only be specified for diagram models. Here <i>n</i> must be the number of root-level blocks. Each output will be returned in the <i>Y1,...,Yn</i> variables.
' <i>model</i> '	The name of a block diagram model.
<i>Timespan</i>	The timespan can be one of the following: TFinal, [TStart TFinal], or [TStart OutputTimes TFinal]. Output times are time points which will be returned in <i>T</i> , but in general <i>T</i> will include additional time points.
<i>Options</i>	Optional simulation parameters created in a structure by the <code>simset</code> command using name-value pairs.
<i>UT</i>	Optional external inputs. For supported expressions, see "Enable Data Import" on page 48-88.

Simulink only requires the first parameter. Simulink takes all defaults from the block diagram, including unspecified options. If you specify any optional arguments, your specified settings override the settings in the block diagram.

Specifying the right-hand side argument of `sim` as the empty matrix, `[]`, causes Simulink to use the default for the argument.

To specify the single-output format for `sim(model, Timespan, Options, UT)`, set the 'ReturnWorkspaceOutputs' option of the options structure to 'on'.

See also `simset` and `simget`.

Control Simulation Using the `set_param` Command

You can use the `set_param` command to

- start
- stop
- pause or continue a simulation
- update a block diagram
- write all data logging variables to the base workspace

Note If you use `matlab -nodisplay` to start a session, you cannot use `set_param` to run your simulation. The `-nodisplay` mode does not support menu simulation.

The format of the `set_param` command is:

```
set_param('sys', 'SimulationCommand', 'cmd')
```

where `'sys'` is the name of the system and `'SimulationCommand'` takes the argument assigned to `'cmd'` (`'start'`, `'stop'`, `'pause'`, `'continue'`, `'update'`, or `'WriteDataLogs'`).

For example, to start a simulation on system `vdp`, use the following command:

```
set_param('vdp', 'SimulationCommand', 'start')
```

When you start a simulation using the `set_param` command and `'start'` argument, you must use the `set_param` command and `'stop'` argument to stop it.

Note You can also use the `sim` command to start a simulation. However, you cannot pause a simulation started with `sim`.

You can update work space variables dynamically when a simulation is running. To do this, type:

```
set_param(bdroot,'SimulationCommand','update')
```

You can also use the `get_param` command to check the status of a simulation. The format of the `get_param` function for this use is

```
get_param('sys','SimulationStatus')
```

The software returns 'stopped', 'initializing', 'running', 'paused', 'updating', 'terminating', or 'external' (used with the Simulink Coder product).

Run Simulation from an S-Function

S-functions can use the `set_param` command to control simulation execution. A C MEX S-function can use the `mexCallMATLAB` macro to call the `set_param` command itself.

Control Simulation Using Block Callbacks

A callback executes when you perform various actions on your model, such as clicking on a block or starting a simulation. You can use callbacks to execute a MATLAB script or other MATLAB commands.

Provide code for the appropriate block callback parameter to specify actions that occur at one of these times:

- At the start of model simulation
- After the simulation pauses
- When the simulation continues
- When the simulation stops

For details, see “Callbacks for Customized Model Behavior” on page 4-56 and “Block Callback Parameters” on page 4-64.

Execute MATLAB Code Before Starting Simulation

You can use the `StartFcn` callback to automatically execute MATLAB code before the simulation starts.

For example, you can make all of the Scope blocks that exist in a model come to the forefront before running the simulation. Create a simple MATLAB script named `openscopes.m` and save it on your MATLAB search path, as shown in the following code.

```
% openscopes.m
% Brings scopes to forefront at beginning of simulation.

blocks = find_system(bdroot,'BlockType','Scope');

% Finds all of the scope blocks on the top level of your
% model to find scopes in subsystems, give 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
```

After you create this MATLAB script, set the `StartFcn` parameter for the model to call the script. For example,

```
set_param('myModel','StartFcn','openscopes')
```

Now every time you run the model, all of the Scope blocks automatically open in the forefront.

Run Parallel Simulations

In this section...

“Overview of Calling `sim` from within `parfor`” on page 15-11

“Simulink and Parallel Computing Toolbox Software” on page 15-15

“Simulink and MATLAB® Distributed Computing Server™ Software” on page 15-16

“`sim` in `parfor` with Normal Mode” on page 15-17

“`sim` in `parfor` with Normal Mode and MATLAB® Distributed Computing Server™ Software” on page 15-18

“`sim` in `parfor` with Rapid Accelerator Mode” on page 15-20

“Workspace Access Issues” on page 15-21

“Resolving Workspace Access Issues” on page 15-21

“Data Concurrency Issues” on page 15-23

“Resolving Data Concurrency Issues” on page 15-23

Overview of Calling `sim` from within `parfor`

The MATLAB `parfor` command allows you to run parallel (simultaneous) Simulink simulations of your model (design). In this context, parallel runs of a model means multiple model simulations at the same time on different workers. Calling `sim` from within a `parfor` loop is often advantageous for performing multiple simulation runs of the same model for different inputs or for different parameter settings. For example, you may save significant 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 22-11 for details on selecting a simulation mode and “Design Your Model for Effective Acceleration” on page 22-16 for optimizing simulation run times.) For other simulations 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 will overwrite the same workspace variables and files, or possibly have collisions trying to write variables and files simultaneously.

For more information on code regeneration and parameter handling in Rapid Accelerator mode, see “Parameter Handling in Rapid Accelerator Mode” on page 22-8.

For details about the `parfor` command, see `parfor`.

Note If you open models inside a `parfor` statement you should close them again using `bdclose all` to avoid leaving temporary files behind.

The following examples show how to use `sim` in `parfor` using the `sldemo_suspn_3dof` model. To access these examples, from Simulink Doc Center, click **Examples > Modeling Features** and navigate to **Simulation Performance**:

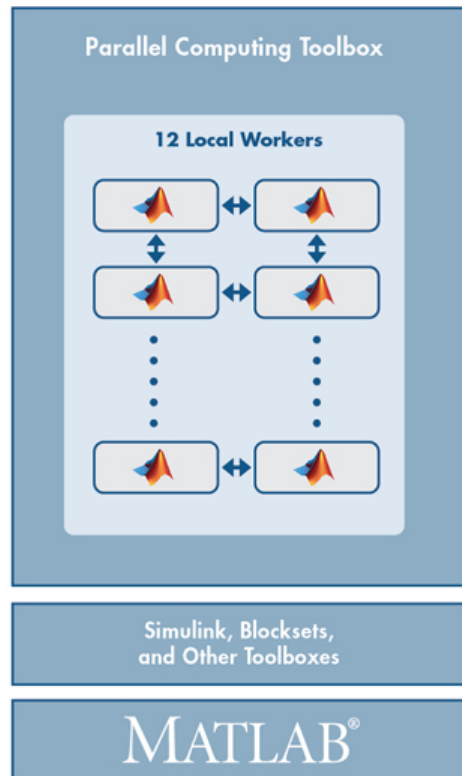
- Parallel Simulations Using Parfor: Test-Case Sweep
- Parallel Simulations Using Parfor: Parameter Sweep in Normal Mode
- Parallel Simulations Using Parfor: Parameter Sweep in Rapid Accelerator Mode

Computationally Intensive Simulations

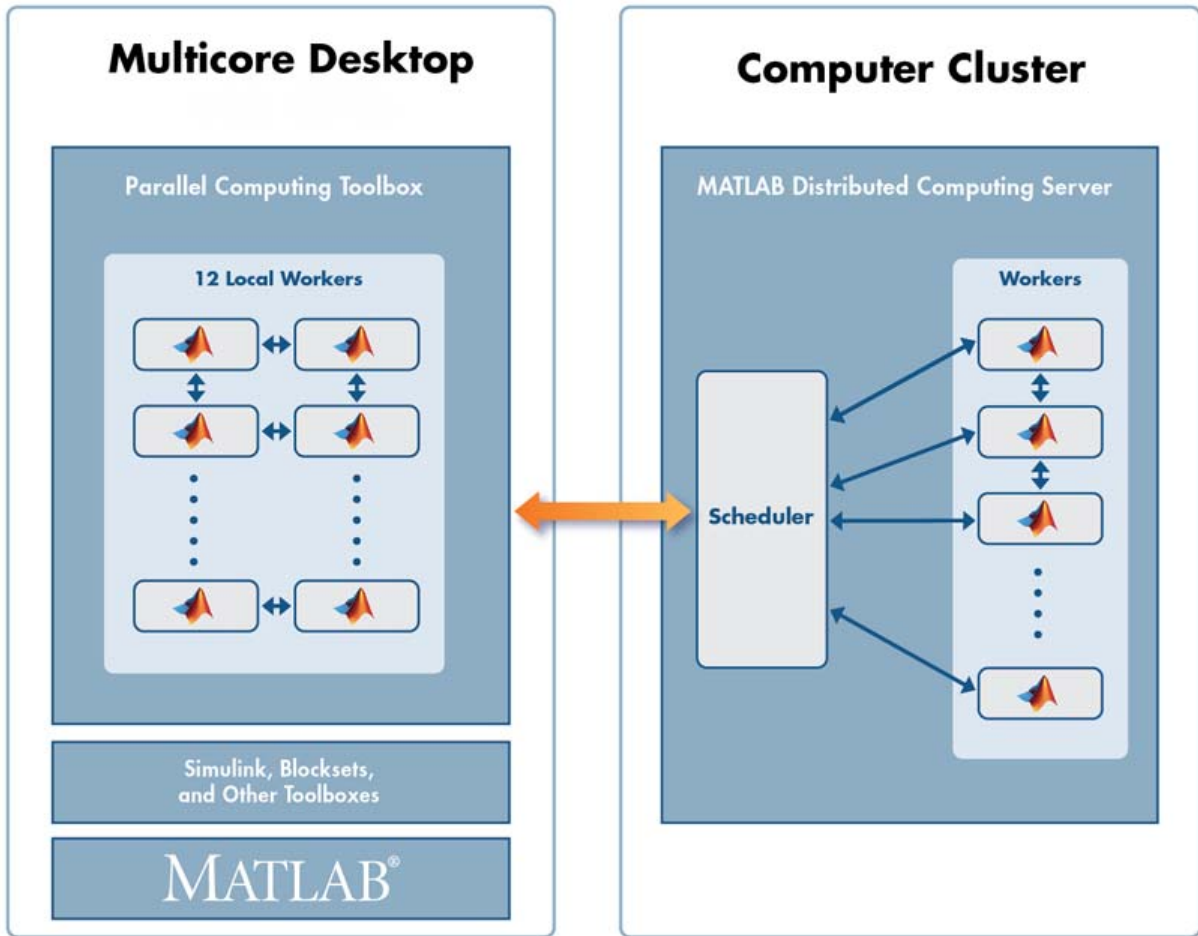
To further improve the performance of multiple simulations of the same model, you can use:

- Multiple designated workers (MATLAB computational engines) using the Parallel Computing Toolbox software

Multicore Desktop



- Multiple computer clusters, clouds, and grids using the MATLAB Distributed Computing Server software



To take advantage of these environments, you can:

- 1 Simulate your model on a single computer.
- 2 When satisfied with single simulation, run multiple simulations in parallel on multiple designated workers on a local multicore desktop.
- 3 When satisfied with multiple simulations on local multicore desktop, run simulations in parallel remotely on computer clusters, clouds, and grids.

Consider the following:

To	Required Software*			
	MATLAB	Simulink	Parallel Computing Toolbox	MATLAB Distributed Computing Server
Run a single simulation.	✓	✓		
Run multiple simulations simultaneously on your computer.	✓	✓	✓	
Run multiple simulations remotely on a server.	✓	✓	✓	✓

* And other required and optional software

Simulink and Parallel Computing Toolbox Software

Before you run simulations in the Parallel Computing Toolbox environment, see “Parallel Computing Toolbox”. You can use the Parallel Computing Toolbox software to run simulations either on a local multicore computer, or on multiple remote computers.

- 1 Have a Parallel Computing Toolbox license.
- 2 Run `sim` in `parfor` for your model.

For an example of using `sim` in `parfor` using Normal mode, see “`sim` in `parfor` with Normal Mode” on page 15-17.

For an example of using `sim` in `parfor` using Rapid Accelerator, see “`sim` in `parfor` with Rapid Accelerator Mode” on page 15-20.

Note Code generation operations for the same model overwrite each other. If you want each worker to generate its own copy of code, attach and distribute the folder to each worker.

You can also use Parallel Computing Toolbox software to run simulations on multiple remote computers or in a nonhomogeneous environment. For these cases, you can use the Parallel Computing Toolbox software with the MATLAB Distributed Computing Server software.

Simulink and MATLAB Distributed Computing Server Software

Before you run simulations in the MATLAB Distributed Computing Server environment, see “MATLAB Distributed Computing Server”.

- 1** Have Parallel Computing Toolbox and MATLAB Distributed Computing Server licenses.
- 2** Select and configure your cluster configuration.
- 3** Use the “Parallel Computing Toolbox” software to create, import, and select a default parallel cluster profile.
- 4** Use the Parallel Computing Toolbox software to run `sim` in `parfor` for your model.

For an example of `sim` in `parfor` with Normal mode using MATLAB Distributed Computing Server software, see “`sim` in `parfor` with Normal Mode and MATLAB® Distributed Computing Server™ Software” on page 15-18.

You can also use `sim` in `parfor` with Rapid Accelerator and MATLAB Distributed Computing Server software (see “`sim` in `parfor` with Rapid Accelerator Mode” on page 15-20). In this example, you build the code once and distribute that generated code to the other local workers. You can adapt this example for a remote cluster environment by:

- Calling `matlabpool` with a cluster name.
- Attaching required files.

For an example of how to adapt to a remote cluster environment, see “sim in parfor with Normal Mode and MATLAB® Distributed Computing Server™ Software” on page 15-18.

sim in parfor with Normal Mode

The following code fragment shows how you can use `sim` and `parfor` in Normal mode.

```
% 1) Load model and initialize the pool.
model = 'sldemo_suspn_3dof';
load_system(model);
matlabpool('open');

% 2) Set up the iterations that we want to compute.
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.
parfor idx=1:numworkers
    % Setup tempdir and cd into it
    addpath(pwd);
    currDir{idx} = pwd;
    addpath(currDir{idx});
    tmpDir{idx} = tempname;
    mkdir(tmpDir{idx});
    cd(tmpDir{idx});
    % 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.
parfor idx=1:numworkers
    cd(currDir{idx});
    rmdir(tmpDir{idx}, 's');
    rmpath(currDir{idx});
end

close_system(model);
matlabpool('close');
```

sim in parfor with Normal Mode and MATLAB Distributed Computing Server Software

The following code fragment shows how you can use `sim` and `parfor` in Normal mode. This code fragment is similar to the one in “sim in parfor with Normal Mode” on page 15-17. The primary difference is:

- In item 1, the `matlabpool` function calls a cluster name:
- In item 3, you need to attach files the model and other required files for distribution to cluster workers on remote machines.

Start your cluster before running the code.

```
% 1) Load model and initialize the pool.
model = 'sldemo_susp_n_3dof';
load_system(model);
matlabpool('open', clustername);

% 2) Setup the iterations that we want to compute.
Cf_sweep      = Cf*(0.05:0.1:0.95);
iterations    = length(Cf_sweep);
simout(iterations) = Simulink.SimulationOutput;

% 3) The model and other required files may not be present on the
% cluster and need to be added as an attached file.
files_attached= { fullfile(which(model)), ...
                 fullfile(which('sldemo_susp_n_3dof_data'))};
apool= gcp;
```

```
for idx=1:length(files_attached)
    if any(strcmp(files_attached{idx}, apool.AttachedFiles))
        % If files are already present in the attached files folder,
        % adding them again will throw an error, instead update.
        apool.updateAttachedFiles();
    else
        % This file is not currently present in the attached
        % files, add it.
        apool.addAttachedFiles(files_attached{idx});
    end
end

% 4) 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.
parfor idx=1:numworkers
    % Setup tempdir and cd into it
    addpath(pwd);
    currDir{idx} = pwd;
    addpath(currDir{idx});
    tmpDir{idx} = tempname;
    mkdir(tmpDir{idx});
    cd(tmpDir{idx});
    % Load the model on the worker
    load_system(model);
end

% 5) 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

% 6) Switch all of the workers back to their original folder.
parfor idx=1:numworkers
    cd(currDir{idx});
    rmdir(tmpDir{idx},'s');
    rmpath(currDir{idx});
end
```

```
end

close_system(model);
matlabpool('close');
```

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, you must:

- Configure the model to run in Rapid Accelerator simulation mode
- 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 only change parameters 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 “Determining If the Simulation Will Rebuild” on page 22-8.

As for the third condition, the following sample code shows how to disable the up-to-date check using the `sim` command.

```
matlabpool open
% 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));  
end  
  
matlabpool close
```

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.

For a detailed example of this method of running parallel simulations, refer to the Rapid Accelerator Simulations Using PARFOR.

Workspace Access Issues

To run `sim` in `parfor`, you must first open a MATLAB pool of workers using the `matlabpool` 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.

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 have tunable parameters in a model workspace. For a detailed discussion on the model workspace, see “Model Workspaces” on page 4-74.

Specifying Parameter Values Using the `sim` Command

Use the `sim` command in the `parfor` loop to set parameters that change with each iteration.

```
% Run parallel simulations of a model that does not
% result in data concurrency issues
model = 'vdp';
paramName = 'StopTime';
paramValue = {'10', '20', '30', '40'};
parfor i=1:4
    simOut{i} = sim(model, ...
                    paramName, paramValue{i}, ...
                    'SaveTime', 'on');
end
```

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', externalInput{i})
    % 'extInp' is the name of the variable in the base
    % workspace which contains the External Input data
    simOut{i} = sim(model, 'ExternalInput', 'extInp');
end
```

For further details, see the Rapid Accelerator Simulations Using PARFOR example.

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 theMentor 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_modelref_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});  
    cd(cwd)  
    rmdir(tmpdir, 's')  
    rmpath(cwd)  
end
```

Note that 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.

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.

```
parfor i=1:4  
    sim(model, ...  
        'ConcurrencyResolvingToFileSuffix', num2str(i),...  
        'SimulationMode', 'rapid');  
end
```

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 “Properties of the `MException` Class”. 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 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 `vdv`, MATLAB would respond to the `getfullname` command with:

```
ans =
```

vdp/Mu

Visualizing and Comparing Simulation Results

- “View Simulation Results” on page 16-2
- “Signal Viewer Tasks” on page 16-6
- “Scope Signal Viewer Tasks” on page 16-11
- “Scope Signal Viewer Characteristics” on page 16-14
- “Signal Generator Tasks” on page 16-21
- “Signal and Scope Manager” on page 16-23
- “Signal Selector” on page 16-27
- “Control Time Scope Programmatically” on page 16-31

View Simulation Results

In this section...

“What Are Scope Blocks, Signal Viewers, Test Points and Data Logging?” on page 16-2

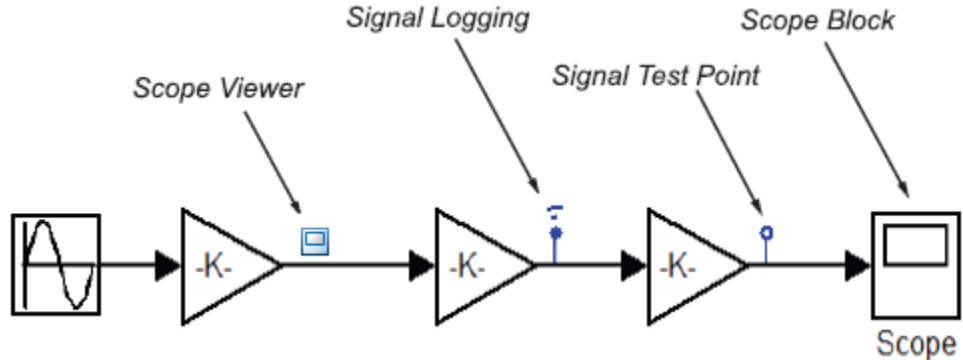
“Scope Block and Scope Signal Viewer Differences” on page 16-3

“Why Use Signal Generators and Signal Viewers Instead of Source Blocks and Scope Blocks?” on page 16-4

What Are Scope Blocks, Signal Viewers, Test Points and Data Logging?

Scope blocks, signal viewers, test points, and data logging provide ways for you to display and capture results from your simulations.

These icons represent the various data display and data capture devices:



- To perform basic signal viewer tasks, see “Signal Viewer Tasks” on page 16-6 and “Scope Signal Viewer Tasks” on page 16-11.
- For detailed information on signal viewers, see “Signal and Scope Manager” on page 16-23.
- For information on signal logging, see “Export Signal Data Using Signal Logging” on page 48-21.

- For information on signal Test Points, see “Test Points” on page 50-59.
- For information on Scope Blocks, see Scope.

Scope Block and Scope Signal Viewer Differences

Use Scope blocks and scope signal viewers to display simulation results, but as shown in this table, they have different characteristics.

Characteristic	Scope Signal Viewer	Scope Block
Interface	Attach to signal using Signal Selector or context menu See “Signal Selector” on page 16-27 See “Scope Signal Viewer Context Menu” on page 16-15	Drag from Library Browser
Scope of Control	All viewers centrally managed from Signal and Scope Manager See “Signal and Scope Manager” on page 16-23.	Each managed individually
Signals per axis	Multiple	One nonbus signal per axis or multiple signals fed through a mux or a bus
Axes per scope	Multiple “Creating a Multiple Axes Scope Signal Viewer” on page 16-11	Multiple
Data handling	Save data to a signal logging object	Save variable data to workspace as structures or arrays

Characteristic	Scope Signal Viewer	Scope Block
Data logging	Log data to model-wide data object See <code>Simulink.SimulationData.Dataset</code> and <code>Simulink.ModelDataLogs</code>	None
Scrolling display data capability	Yes	No
Display	<ul style="list-style-type: none"> • Data markers • Legends • Color and line codes distinguish signals 	Color and line codes distinguish signals
Graph Refresh Period	Adjustable	Fixed
Display of Minor Steps	The viewer does not display minor steps regardless of the value of the Refine parameter setting.	Only displays major time step values. The scope displays additional interpolated points between major time steps if specified by the refine parameter.

Why Use Signal Generators and Signal Viewers Instead of Source Blocks and Scope Blocks?

You should use signal generators and signal viewers instead of Source and Scope blocks when you want to:

- Navigate to and attach signal generators or signal viewers deep within a model hierarchy.
- Centrally manage all signal generators and signal viewers present in your model.

- Use the display features provided by signal viewers that are not available in Scope blocks.
- Reduce clutter in your block diagram. Because signal viewers attach directly to signals, it is not necessary for you to route them to a Scope block. This results in fewer signal routes in your block diagram.
- View data from referenced models. See “Test Points” on page 50-59 for more information.

Signal Viewer Tasks

In this section...

- “About Signal Viewers” on page 16-6
- “Attaching a Signal Viewer” on page 16-6
- “Adding Signals to a Signal Viewer” on page 16-8
- “Displaying a Signal Viewer” on page 16-9
- “Saving Data to MATLAB Workspace” on page 16-10
- “Plotting the Viewer Data” on page 16-10

About Signal Viewers

- Signal viewers are not the same as Scope blocks. For an explanation of the differences, see “Scope Block and Scope Signal Viewer Differences” on page 16-3.
- Signal viewers do not show signal labels on the axis, display the simulation minor time step values, or work with Simulink Report Generator.
- Not all signal viewer features are supported when you simulate your model in Rapid Accelerator mode. See “Behavior of Scopes and Viewers with Rapid Accelerator Mode” on page 22-19.
- The **Help** button for a signal viewer is located on the parameters dialog box for a specific viewer type. For more information, see “Scope Signal Viewer Parameters Dialog Box” on page 16-16 .

Signal viewer

Attaching a Signal Viewer

If you want to,

- Quickly connect or disconnect a scope signal viewer, use the context menu.
- Review all of the signal viewers and the signals connected to them, use the Signal and Scope Manager.

Using the Context Menu

To attach a signal viewer with the signal content menu,

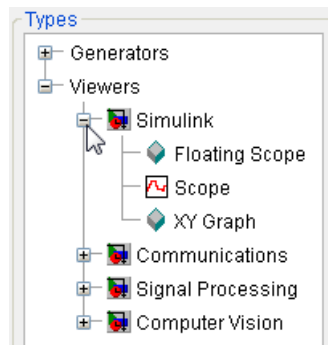
- 1 On your Simulink block diagram, right-click a signal, and then from the context menu, select **Create & Connect Viewer**.
- 2 From the list that appears, select the type of signal viewer you want to attach.

If you attach a scope signal viewer, a viewer window opens. You must run the simulation after you attach the viewer for the information to be plotted.

Using the Signal and Scope Manager

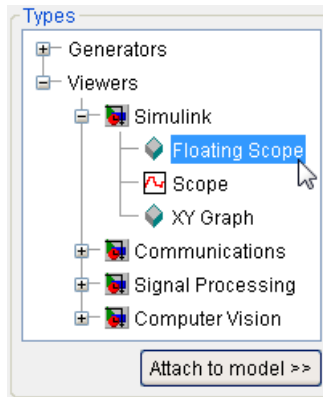
To attach a scope signal viewer using the Scope & Signal Manager,

- 1 On your Simulink block diagram, right-click a signal, and then from the context menu, select **Signal & Scope Manager**. The Signal & Scope Manager dialog box opens.
- 2 In the **Types** pane and under the Viewers node, expand a product node to show the viewers installed and available to you.

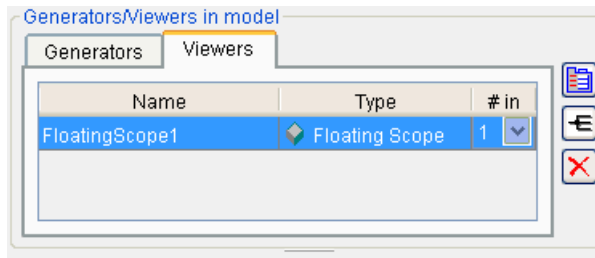


Note The Simulink Scope displayed in the Signal and Scope Manager Types pane is not the same as the Simulink Scope Block. For an explanation of the differences, see “Scope Block and Scope Signal Viewer Differences” on page 16-3.

- Under an expanded product node, select a viewer, and then click the **Attach to model** button.



The viewer is added to a table in the **Viewers** tab in the Generators/Viewers in model pane. The table lists the viewers in your model.




Each row corresponds to a viewer. The columns specify each viewer name, type, and number of inputs. If a viewer accepts a variable number of inputs, the **# in** entry for the viewer contains a pull-down list that displays the range of inputs that the viewer can accept. To change the number of inputs accepted by the viewer, select a value from the pull-down list.

Adding Signals to a Signal Viewer

To add additional traces to an existing signal viewer,

- In the Simulink Editor, right-click a signal.

- 2 From the signal context menu, point to **Create & Connect Viewer**, and then from the submenu, point to a product.
- 3 From the list, select the signal viewer you want to attach to the signal.

A viewer icon  is added to the signal line.

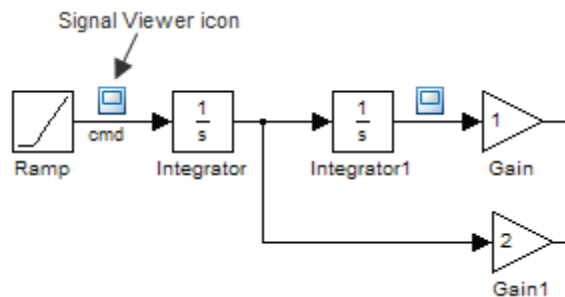
You can also add signals to a signal viewer with the Signal and Scope Manager. See “Signal and Scope Manager” on page 16-23.

Note You must run a simulation for the new signal trace to display in a viewer.

Note If you add or remove signals from a Signal Viewer, the Signal Viewer title and legends update only when you run a simulation, and then open the viewer.

Displaying a Signal Viewer

- 1 Right-click a signal viewer icon. From the context menu, point to **Open Viewer**, and then select a viewer from the list.



Depending on the type of signal viewer, the following happens:


- Scope and Floating Scope — A Viewer: Scope window opens with a graph of the signals attached to the viewer.

- XY Graph — A Sink Block Parameters dialog box opens. After running a simulation, a Viewer: XY Graph window opens with a graph of the X and Y signals attached to the viewer.

2 Customize the signal viewer to better visualize the simulation results.

Saving Data to MATLAB Workspace

To save the data displayed on the viewer to the MATLAB workspace, perform the following steps:

- 1** In the Viewer toolbar, click the parameters icon  .
- 2** Click the **History** tab.
- 3** Select **Save to model signal logging object (logout)**.
- 4** In the **Logging name** field, enter a logging variable name.
- 5** Assign a name to the signal in the block diagram, for example, *x1*.
- 6** Run the simulation by selecting **Simulation > Start**.

You can now view the data in the MATLAB Command Window by entering the following commands:

```
logout.unpack('all')  
data = x1.Data  
x1.time
```

where *x1* is the name of the signal.

Plotting the Viewer Data

You can plot the data interactively or programmatically using the Simulation Data Inspector. Plot the data interactively as explained in “Inspect Signal Data” on page 17-11. The Simulation Data Inspector API allows you to plot and compare signal data programmatically as seen in “Record and Inspect Signal Data Programmatically” on page 17-57.

Scope Signal Viewer Tasks

In this section...
“Adding a Legend to a Scope Signal Viewer” on page 16-11
“Creating a Multiple Axes Scope Signal Viewer” on page 16-11

Adding a Legend to a Scope Signal Viewer

- 1 In a Scope signal viewer window, right-click.
- 2 From the context menu, select **Legends**.

The viewer draws a box and identifies each signal with the name of the signal. If a signal does not have a name, the legend uses the name of the originating block or subsystem.

- 3 To move a legend to another location on the graph, left-click and drag the legend box.

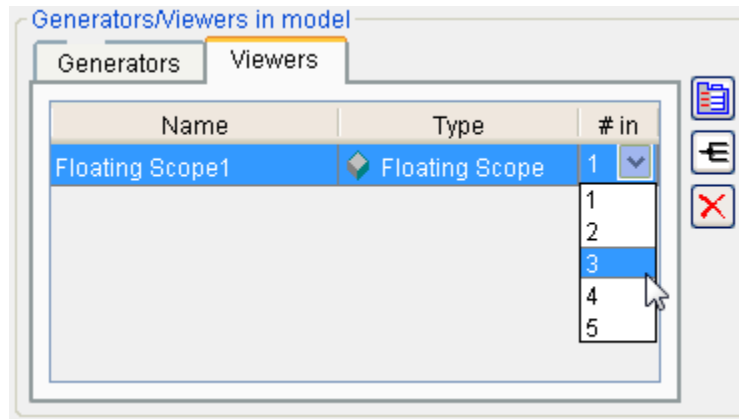
For information about the color and line styles in the Viewer window, see “Line Styles with Scope Signal Viewer” on page 16-19.

Creating a Multiple Axes Scope Signal Viewer

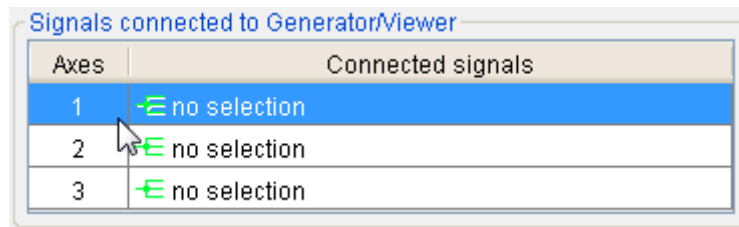
You can add multiple plots (called *axes*) to a scope signal viewer. Each axes can have different *y*-axis settings.



To create a viewer with more than one axes.

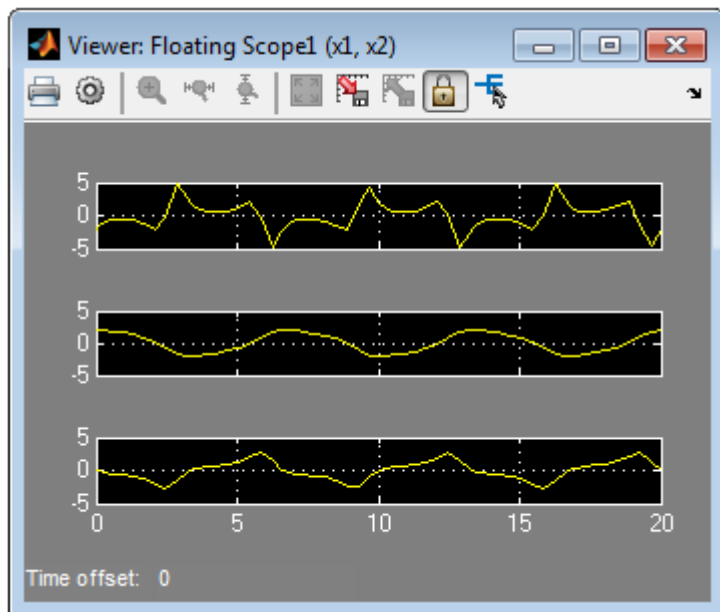
- 1 Open the Signal and Scope manager. Add a signal Scope or Floating Scope signal viewer to your model. See “Attaching a Signal Viewer” on page 16-6.
- 2 From the **# in** drop-down list, select the total number of axes for the graph.



- 3 In the **Signals connected to Generator/Viewer** section, select the first axis.



- 4 Click the **Signal Selector** button . The Signal Selector dialog box opens.
- 5 Select the signal or signals to add to this axis, and then close the dialog.
- 6 Repeat steps 3 to 5 until you have added signals to all of the axes.
- 7 Click the parameters icon  to display the scope signal viewer, and then run the simulation.













Scope Signal Viewer Characteristics

In this section...
“Scope Signal Viewer Toolbar” on page 16-14
“Scope Signal Viewer Context Menu” on page 16-15
“Scope Signal Viewer Parameters Dialog Box” on page 16-16
“Line Styles with Scope Signal Viewer” on page 16-19
“Parameter Settings and Performance with Scope Signal Viewer” on page 16-20

Scope Signal Viewer Toolbar

The signal viewer toolbar is attached to each scope signal viewer. It has the following controls.

Icon	Function
	Opens the Print dialog box so you can print the contents of a Scope Viewer window.
	Opens the Scope parameters dialog for modifying display characteristics. For details, see “Scope Signal Viewer Parameters Dialog Box” on page 16-16.
	Simultaneously zooms in on the x and y axes. The zoom feature is not active while the simulation is running.
	Use this button to zoom in on the x axis only. The zoom feature is not active while the simulation is running.
	Use this button to zoom in on the y axis only. The zoom feature is not active while the simulation is running.
	Automatically scales the axis to fully display all signals.
	Stores the current axis settings so you can apply them to the next simulation.
	Restores the graph setting values saved by the most recent Save axes settings command.

Icon	Function
	Activates the Signal Selector. For more information, see “Signal Selector” on page 16-27.
	Docks and undocks the Scope Viewer. When you dock the Scope Viewer, it is placed within the MATLAB Command Window and automatically resized.

Scope Signal Viewer Context Menu

The scope signal viewer context menu is a convenient way to make simple changes to a viewer without navigating to a Scope or Floating Scope parameters dialog box.


Within a Scope signal viewer window, right-click to display the context menu. It contains the following controls.

Control	Function
Legends	Adds a legend to your scope viewer.
Autoscale	Automatically scales the viewer axis.
Signal selection	Displays the Signal Selector dialog. For information, see “Signal Selector” on page 16-27
Axes properties	Displays the Axis Properties dialog. You can manually set the minimum and maximum range for the y axis here.

Control	Function
Scope parameters	Displays the Scope parameters dialog. For information, see “Scope Signal Viewer Parameters Dialog Box” on page 16-16.
Tick labels	Displays the Tick Labels dialog. From here you can turn on and off various tick options.

Scope Signal Viewer Parameters Dialog Box

To open a Scope or Floating Scope parameters dialog box,

- On the scope toolbar, click 

There are three tabs on the dialog box:

- **General** — set the axis characteristics and the sampling decimation value
- **History** — control the amount of stored and displayed data
- **Performance** — control the scope refresh rate.

General Tab

With this tab you control the number of axes, the time range, and the appearance of your graph.

Number of axes. Set the number of axes in this data field. Each axis is displayed as a separate graph within a single Scope Viewer.

An example of this is shown in “Adding Signals to a Signal Viewer” on page 16-8.

Time Range. Change the x-axis limits by entering a number or auto in the **Time range** field.

Entering a number of seconds causes each screen to display the amount of data that corresponds to that number of seconds. Enter `auto` to set the x -axis to the duration of the simulation.

Note Do not enter variable names in these fields.

Tick labels. Specifies whether to label axes ticks. The options are:

Option	Effect
<code>all</code>	Places ticks on the outside of all axes
<code>inside</code>	Places tick labels inside all axes (available only on signal viewers)
<code>bottom axis only</code>	Places tick labels outside the bottom axes

Scroll. When you select this option, the scope continuously scrolls the displayed signals to the left to keep as much data in view as will fit on the screen at any one time.

In contrast, when this option is not selected, the scope draws a screen full of data from left to right until the screen is full, erases the screen, and draws the next screen full of data. This loop is repeated until the end of simulation time. The effects of this option are discernible only when drawing is slow, for example, when the model is very large or has a very small step size.

Note In some cases, the simulation slows when the simulation runs with the scroll option selected. See “Parameter Settings and Performance with Scope Signal Viewer” on page 16-20.

Data markers. Displays a marker at each data point on the Scope Viewer screen.

Legends. Displays a legend on the scope that indicates the line style used to display each signal.

Decimation. Logs every Nth data point, where N is the number entered in the edit field.

For example, suppose that the input signal to your Scope block has a fixed sample time of 0.1 s. If you enter a value of 2, data points for this viewer will be recorded at times 0.0, 0.2, 0.4....

History Tab

With this tab you control the amount of data that the Scope signal viewer stores, displays and stores to the workspace. The values that appear in these fields are the values that are used in the next simulation.

Limit data points to last. Limits the number of data points saved to the workspace. Select the **Limit data points to last** check box and enter a value in its data field.

The Scope relies on its data history for zooming and autoscaling operations. If the number of data points is limited to 1,000 and the simulation generates 2,000 data points, only the last 1,000 are available for regenerating the display.

Save to model signal logging object. At the end of the simulation, this option saves the data displayed on the Scope Viewer . The data is saved in the `Simulink.ModelDataLogs` object used to log data for the model (see “Export Signal Data Using Signal Logging” on page 48-21 for more information).

For this option to take effect, you must also enable signal logging for the model as a whole. To do this, check the **Signal logging** option on the **Data Import/Export** pane of the **Model Configuration Parameters** dialog box.

For new models, use the Dataset logging format. If you use the Dataset format for signal logging, then Simulink does not log the signals configured to be logged in the Signal Viewer. Explicitly mark the signal for signal logging by using the Signal Properties dialog box. To access the Signal Properties dialog box, right-click a signal and from the context menu, select **Properties**.

Logging Name. Specifies the name under which to store the viewer's data in the model's `Simulink.ModelDataLogs` object. The name must be different from the log names specified by other signal viewers or for other signals, subsystems, or model references logged in the model's `Simulink.ModelDataLogs` object.

Performance Tab

Controls how frequently the Scope signal viewer is refreshed. Reducing the refresh rate can speed up the simulation in some cases.

Note For information about additional scope signal viewer parameters that can affect performance, see “Parameter Settings and Performance with Scope Signal Viewer” on page 16-20.

This tab contains the following controls.

Refresh Period. Select the units in which the refresh period is expressed. Options are either seconds or frames, where a frame is the width of the scope's screen in seconds. This is the value of the scope's **Time range** parameter.

Refresh Slider. Sets the refresh rate.

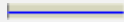
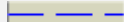


Drag the slider button to the right to increase the refresh period and hence decrease the refresh rate.

Freeze Button. Controls refresh.

Click the button to freeze (stop refreshing) or unfreeze the Scope Viewer.

Line Styles with Scope Signal Viewer

If a signal contains multiple elements such as a vector or matrix, the viewer distinguishes the elements with different line styles. If a signal has more than four elements, the viewer cycles through the line styles. The line styles retain the color of the signal.

Signal Element	Scope Viewer
1	
2	
3	
4	

Parameter Settings and Performance with Scope Signal Viewer

In some cases, when a Scope signal viewer needs to display a large number of data points, the simulation slows down. When this happens, you can improve simulation performance by adjusting the settings of some of the viewer parameters. Try one or a combination of the following until you are satisfied with the simulation performance.

- Turn off scroll mode.
- Reduce the time range.
- Use decimation to reduce the number of data points.
- Increase the refresh period to decrease the refresh rate.
- Limit the number of data points that the viewer saves to the workspace.

Signal Generator Tasks

In this section...
“Attaching a Signal Generator.” on page 16-21
“Removing a Generator” on page 16-22

Attaching a Signal Generator.

If you want to,

- Quickly connect or disconnect a Signal Generator, use the context menu.
- Review all of the Signal Generators and their signals, use the Signal and Scope Manager.

Using the Context Menu

To add additional traces to an existing signal viewer,

- 1 In the Simulink Editor, right-click the input to a block.
- 2 From the context menu, point to **Create & Connect Generator**, and then from the submenu, point to a product.
- 3 From the list, select the generator you want as input to the block.

The name of the generator you choose appears in a box connected to the block input.

- 4 Double-click the name to edit the generator name.
- 5 Right-click the name and select **Generator Parameters** to display a dialog box where you can change the generator parameters.
- 6 Right-click the name and select **Properties** to display a dialog box where you can change the signal properties.

Using the Signal and Scope Manager

To attach a Signal Generator using the Signal and Scope Manager,

- 1 Right-click the input to a block, and then from the signal context menu select **Signal & Scope Manager**.
- 2 In the **Types** pane and under the Generators node, expand a product node to show the generator installed and available to you.
- 3 Under an expanded product node, select a generator, and then click the **Attach to model** button.

The viewer is added to a table in the **Generators** tab in the Generators/Viewers in model pane. The table lists the generators in your model.

Each row corresponds to a generator. The columns specify each generator name and type.

Clicking on the name of a generator displays the connected signals. For instance, the constant is shown connected to the second input of the sum block.

Removing a Generator

To remove the generator from the block diagram.

- 1 Right-click a generator.
- 2 From the context menu, select **Disconnect Generator**.

Signal and Scope Manager

In this section...
“About the Signal & Scope Manager” on page 16-23
“Opening the Signal and Scope Manager” on page 16-24
“Changing Generator or Viewer Parameters” on page 16-25
“Adding Signals to a Viewer” on page 16-25
“Removing a Generator or Viewer from a Simulink Model” on page 16-25
“Viewing Test Point Data” on page 16-26

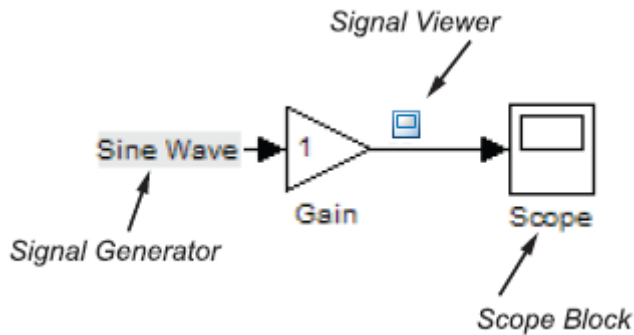
About the Signal & Scope Manager

The Signal & Scope Manager is a user interface to Signal Generator and Viewer objects. Using the Signal and Scope Manager you can manage from a central place all signal generators and viewers.

Note The Signal and Scope Manager requires that you have Java enabled when you start MATLAB. This is the default.

Signal Generator and Viewer Objects

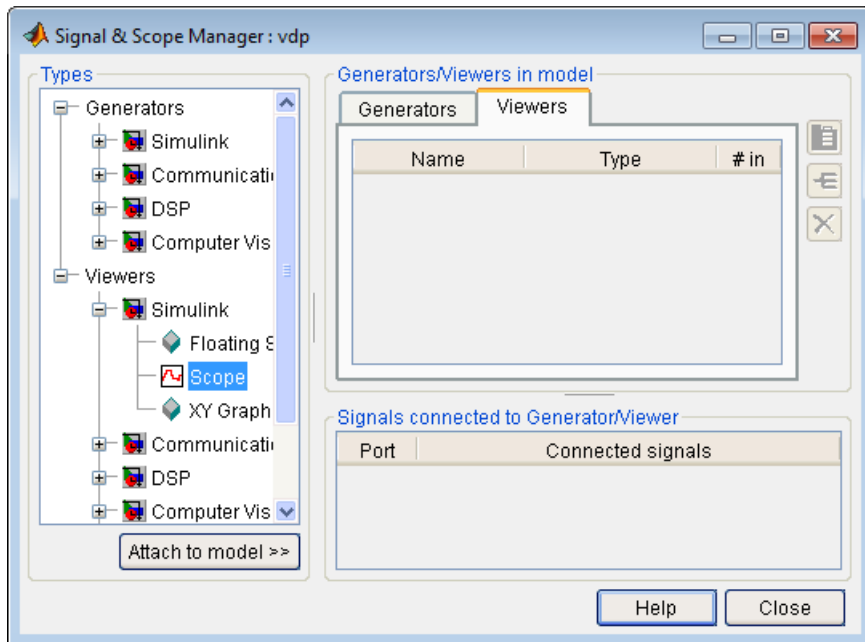
The small icons identifying a viewer or generator are called Viewer and Generator Objects. These objects are not the same as scope blocks or signal blocks. Generator and Viewer Objects are managed by the Signal and Scope Manager, and are placed on signals. Blocks are dragged from the Library browser and are not managed by the Signal and Scope manager.



Opening the Signal and Scope Manager



- 1 In the Simulink Editor window, select **Diagram > Signals & Ports > Signal & Scope Manager**.

The Signal & Scope Manager window opens.




Alternatively within your model, right click a signal line and from the context menu, select **Signal & Scope Manager**.

Changing Generator or Viewer Parameters

- 1 Open the Signal & Scope Manager window.
- 2 To the right side of the Generators and Viewers pane, click the parameters button .
 - For a generator, the Generator Parameters dialog opens.
 - For a viewer, a Viewer window opens. From the viewer toolbar, select the parameters button . The Viewer parameters dialog opens.
- 3 Review and change parameters.

Adding Signals to a Viewer

Use the Signal Selector to add signals to a Viewer.

- 1 To the right side of the Generators/Viewers pane, click the Signal Selector button .


The Signal Selector window opens. Use the Signal Selector to connect and disconnect generators and viewers.

- 2 From the signals pane, select the check boxes for the signals you want to display in the selected viewer.

Tip After adding the new signals, run a simulation to make them visible.

Removing a Generator or Viewer from a Simulink Model

- 1 In the Generators/Viewers pane, select either a listed generator for viewer.

- 2 On the right side, click the Delete button 

The selected generator or viewer is removed from the table.

Viewing Test Point Data

You can use a Signal Viewer available from the Signal and Scope Manager to view any signal that is defined as a test point in a submodel. A test point is a signal that is guaranteed to be observable when using a signal viewer in a model.

Note With some signal viewers (for example, XY Graph, To Video Display, Matrix Viewer, Spectrum Scope, and Vector Scope), you cannot use the Signal Selector to select signals with test points in referenced models.

For more information, see “Test Points” on page 50-59.

Signal Selector

In this section...

“About the Signal Selector” on page 16-27

“Select signals for object” on page 16-28

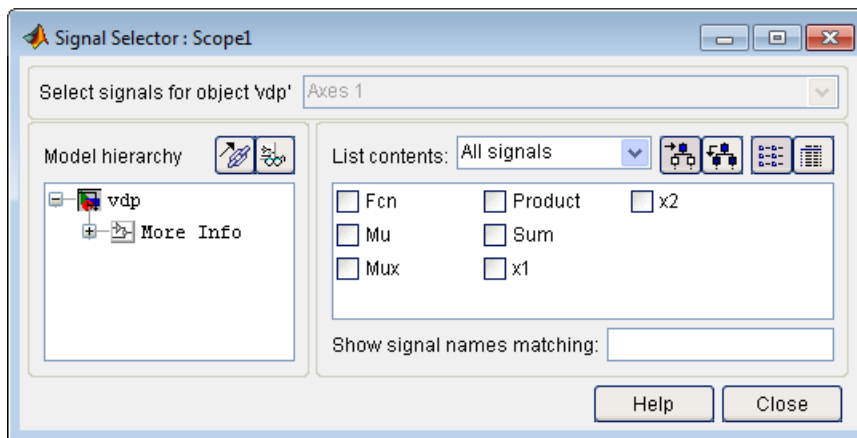
“Model Hierarchy” on page 16-28

“Inputs/Signals List” on page 16-28

About the Signal Selector

The Signal Selector allows you to connect a generator or viewer object (see “Attaching a Signal Viewer” on page 16-6 and “Attaching a Signal Generator.” on page 16-21) or a Floating Scope block to other block inputs and outputs. It appears when you click the **Signal selection** button for a generator or viewer object in the Signal & Scope Manager or on the toolbar of the Floating Scope window.

The Signal Selector that appears when you click the **Signal selection** button applies only to the currently selected generator or viewer object (or the Floating Scope). If you want to connect blocks to another generator or viewer object, you must select the object in the Signal & Scope Manager and launch another instance of the Signal Selector. The object used to launch a particular instance of the Signal Selector is called that instance’s owner.



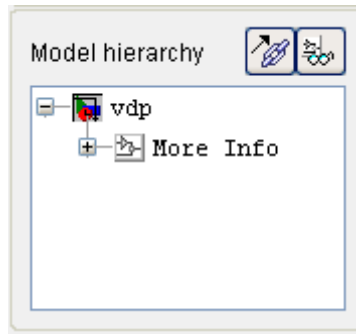
Select signals for object

This list box allows you to select the owner output port (in the case of signal generators) or display axis (in the case of signal viewers) to which you want to connect blocks in your model.

The list box is enabled only if the signal generator has multiple outputs or the signal viewer has multiple axes.

Model Hierarchy

This tree-structured list lets you select any subsystem in your model.

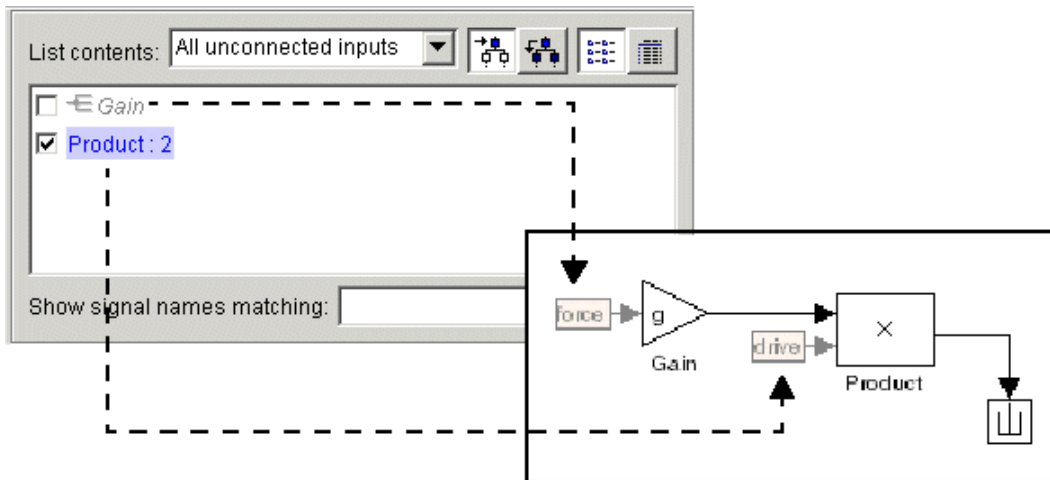


Selecting a subsystem causes the adjacent port list to display the ports available for connection in the selected subsystem. To display subsystems included as library links in your model, click the **Follow links** button at the top of the **Model hierarchy** control. To display subsystems contained by masked subsystems, click the **Look under masks** button at the top of the panel.

Inputs/Signals List

The contents of this panel displays input ports available for connection to the Signal Selector's owner if the owner is a signal generator or signals available for connection to the owner if the owner is a signal viewer.

If the Signal Selector's owner is a signal generator, the inputs/signals list by default lists each input port in the system selected in the model hierarchy tree that is either unconnected or connected to a signal generator.



The label for each entry indicates the name of the block of which the port is an input. If the block has more than one input, the label indicates the number of the displayed port. A greyed label indicates that the port is connected to a signal generator other than the Signal Selector's owner. Selecting the check box next to a port's entry in the list connects the Signal Selector's owner to the port, replacing, if necessary, the signal generator previously connected to the port.

To display more information on each signal, click the **Detailed view** button at the top of the pane. The detailed view shows the path and data type of each signal and whether the signal is a test point. The controls at the top and bottom of the panel let you restrict the amount of information shown in the ports list.

- To show named signals only, select **Named signals only** from the **List contents** control at the top of the pane.
- To show only signals selected in the Signal Selector, select **Selected signals only** from the **List contents** control.
- To show test point signals only, select **Testpointed/Logged signals only** from the **List contents** control.

- To show only signals whose signals match a specified string of characters, enter the characters in the **Show signals matching** control at the bottom of the **Signals** pane and press the **Enter** key.
- To show the selected types of signals for all subsystems below the currently selected subsystem in the model hierarchy, click the **Current and Below** button at the top of the **Signals** pane.

To select or deselect a signal in the **Signals** pane, click its entry or use the arrow keys to move the selection highlight to the signal entry and press the **Enter** key. You can also move the selection highlight to a signal entry by typing the first few characters of its name (enough to uniquely identify it).

Note You can continue to select and deselect signals on the block diagram with the Signal Selector open. For example, shift-clicking a line in the block diagram adds the corresponding signal to the set of signals that you previously selected with the Signal Selector. If the simulation is running when you open and use the Signal Selector, Simulink updates the Signal Selector to reflect signal selection changes you have made on the block diagram. However, the changes do not appear until you select the Signal Selector window itself. You can also use the Signal Selector before running a model. If no simulation is running, selecting a signal in the model does not change the Signal Selector.

Control Time Scope Programmatically

In this section...

“Use Simulink.scopes.TimeScopeConfiguration” on page 16-31

“Time Scope Configuration Parameters” on page 16-32

Use Simulink.scopes.TimeScopeConfiguration

Use instances of Simulink.scopes.TimeScopeConfiguration to programmatically affect the Time Scope display.

- 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

In this example, the variable htsc stores the mask object obtained using get_param.

```
sysname=char(randi(26,1,7)+96);
new_system(sysname);
add_block('built-in/TimeScope',[sysname,'/TimeScope'])
htsc = get_param([sysname,'/TimeScope'],'ScopeConfiguration')
```

ans =

TimeScopeConfiguration with properties:

```

                Name: 'TimeScope'
    NumInputPorts: '1'
    BufferLength: '50000'
  OpenAtSimulationStart: 0
            Position: [755 450 410 300]
    MaximizeAxes: 'Auto'
    AxesScaling: 'Manual'
            TimeSpan: 'Auto'
  TimeSpanOverrunAction: 'Wrap'
```

```
        TimeUnits: 'none'  
    TimeDisplayOffset: '0'  
        TimeAxisLabels: 'Bottom'  
    LayoutDimensions: [1 1]  
        ActiveDisplay: 1  
            Title: ''  
        ShowLegend: 0  
        ShowGrid: 1  
    PlotAsMagnitudePhase: 0  
        YLimits: [-10 10]  
        YLabel: 'Amplitude'
```

Time Scope Configuration Parameters

The following predefined parameters are associated with a Time Scope Configuration. See `Simulink.scopes.TimeScopeConfiguration` for a detailed description of each property.

ActiveDisplay

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display should have its axes colors, line properties, marker properties, and visibility changed. This property is Tunable.

AxesScaling

Specify how axes should be scaled when plotting data

BufferLength

Specify number of data points in buffer

LayoutDimensions

Specify the layout grid dimensions as a 2-element vector: `[numberOfRows, numberOfColumns]`. You can use no more than four rows or four columns. This property is Tunable.

MaximizeAxes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- **Auto** — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- **On** — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- **Off** — In this mode, none of the axes appear maximized.

This property is Tunable.

Name

Specify as a string the caption to display on the scope window. This property is Tunable.

NumInputPorts

Specify the number of input signals to display on the scope as a positive integer. You must invoke the `step` method with the same number of inputs as the value of this property.

OpenAtSimulationStart

Open the Time Scope window when starting the simulation

PlotAsMagnitudePhase

When you set this property to `true`, the scope plots the magnitude and phase of the input signal on two separate axes within the same active display. When you set this property to `false`, the scope plots the real and imaginary parts of the input signal on two separate axes within the same active display. This property is particularly useful for complex-valued input signals. Selecting this check box affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When

the amplitude of the input signal is negative, the phase is 180 degrees. This property is Tunable.

Position

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form, [`left bottom width height`]. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable.

ShowGrid

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid is hidden. This property is Tunable.

ShowLegend

When you set this property to `true`, the scope displays a legend with automatic string labels for each input channel. When you set this property to `false`, the scope does not display a legend. This property applies only when you set the `SpectrumType` property to `'Power'` or `'Power density'`. This property is Tunable.

TimeAxisLabels

Specify how *time*-axis labels should appear in the scope displays as one of `'All'`, `'Bottom'`, or `'None'`.

- When you set this property to `'All'`, *time*-axis labels appear in all displays.
- When you set this property to `'Bottom'`, *time*-axis labels appear in the bottom display of each column.
- When you set this property to `'None'`, there are no labels in any displays.

This property is Tunable.

TimeDisplayOffset

Specify the offset, in seconds, to apply to the *time*-axis. This property can be either a numeric scalar or a vector of length equal to the number of input channels. If you specify this property as a scalar, then that value is the time

display offset for all channels. If you specify a vector, each vector element is the time offset for the corresponding channel. For vectors with length less than the number of input channels, the time display offsets for the remaining channels are set to 0. If a vector has a length greater than the number of input channels, the extra vector elements are ignored. This property is Tunable.

TimeSpan

Specify the time span, in seconds, as a positive, numeric scalar value. This property applies when `FrameBasedProcessing` is `false`. This property also applies when `FrameBasedProcessing` is `true` and `TimeSpanSource` is `Property`. The *time*-axis limits are calculated as follows.

- Minimum *time*-axis limit = `min(TimeDisplayOffset)`
- Maximum *time*-axis limit = `max(TimeDisplayOffset) + TimeSpan`

where `TimeDisplayOffset` and `TimeSpan` are the values of their respective properties. This property is Tunable.

TimeSpanOverrunAction

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- **Wrap** — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *time*-axis limit.
- **Scroll** — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This property is Tunable.

TimeUnits

Specify the units used to describe the *time*-axis. You can select one of the following options:

- **Metric** — In this mode, the scope converts the times on the *time*-axis to the most appropriate measurement units. These can include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.
- **Seconds** — In this mode, the scope always displays the units on the *time*-axis as seconds.
- **None** — In this mode, the scope does not display any units on the *time*-axis. The scope only shows the word **Time** on the *time*-axis.

This property is Tunable.

Title

Specify the display title as a string. Enter %<SignalLabel> to use the signal labels in the Simulink Model as the axes titles. This property is Tunable.

YLabel

Specify as a string the text for the scope to display to the left of the *y*-axis. This property is Tunable.

YLimits

Specify the *y*-axis limits as a 2-element numeric vector, [ymin ymax]. This property is Tunable.

Inspecting and Comparing Logged Signal Data

- “Inspect Signal Data with Simulation Data Inspector” on page 17-2
- “Requirements for Recording Data” on page 17-4
- “Record Simulation Data” on page 17-5
- “Import Logged Signal Data” on page 17-7
- “Load Previously Recorded Data from a MAT-file” on page 17-10
- “Inspect Signal Data” on page 17-11
- “Compare Signal Data” on page 17-23
- “Comparison of One Signal From Multiple Simulations” on page 17-25
- “Compare All Logged Signal Data From Multiple Simulations” on page 17-28
- “Create Simulation Data Inspector Report” on page 17-32
- “Export Results in the Simulation Data Inspector Tool” on page 17-34
- “How the Simulation Data Inspector Tool Aligns Signals” on page 17-35
- “How the Simulation Data Inspector Tool Compares Time Series Data” on page 17-37
- “Run Management Configuration” on page 17-38
- “Customize the Simulation Data Inspector Interface” on page 17-41
- “Limitations of the Simulation Data Inspector” on page 17-56
- “Record and Inspect Signal Data Programmatically” on page 17-57

Inspect Signal Data with Simulation Data Inspector

The Simulation Data Inspector software provides the capability to inspect and compare time series data at several stages of your workflow:

- Model design: Inspect and compare simulation data after making changes to the model diagram or its configuration.
- Testing your model: Compare simulation data with different input data
- Code generation: Compare simulation data and generated code output of your model

You can compare variable-step data, fixed-step solver data from Simulink and Simulink Coder, and fixed-step output with external data. “How the Simulation Data Inspector Tool Aligns Signals” on page 17-35 describes how signals are aligned between runs. “How the Simulation Data Inspector Tool Compares Time Series Data” on page 17-37 describes how aligned signal data are compared.

A typical workflow for inspecting and comparing signal data is:

- 1** Set up your model to log signal data, as in “Export Signal Data Using Signal Logging” on page 48-21.
- 2** Open the Simulation Data Inspector tool, as described in “Open the Simulation Data Inspector Tool” on page 17-41.
- 3** Simulate your model and record simulation data or import logged signal data, as described in “Record Simulation Data” on page 17-5 and “Import Logged Signal Data” on page 17-7.
- 4** Configure the Signal Browser table and specify how you want to graph the data, as described in “Customize the Simulation Data Inspector Interface” on page 17-41.
- 5** Inspect signals to quickly determine if the run satisfies requirements, as described in “Inspect Signal Data” on page 17-11.
- 6** If the run is unsatisfactory, delete it. Repeat steps 3 and 4 to collect the desired simulation runs for comparing data.

- 7** Optionally assign tolerances to signals and graphically inspect the applied tolerances, as described in “Customize the Simulation Data Inspector Interface” on page 17-41.
- 8** Compare individual signals in the same run, or from different runs, as described in “Compare Signal Data” on page 17-23.
- 9** Compare all of the imported signal data from multiple runs, as described in “Compare All Logged Signal Data From Multiple Simulations” on page 17-28.
- 10** Determine which signals have discrepancies within the specified tolerances. Plot and analyze the discrepancies of any two signals.
- 11** Save the imported signal data and comparison results, as described in “Export Results in the Simulation Data Inspector Tool” on page 17-34.

The Simulation Data Inspector software provides a command-line interface. For more information, see “Record and Inspect Signal Data Programmatically” on page 17-57

Requirements for Recording Data

The Simulation Data Inspector tool records the following data configured on the “Data Import/Export Pane” of the Configuration Parameter dialog box:

- **States** and **Output**, if **Format** is **Structure** with **time**, or if **Format** is **Array** or **Structure** and **Time** is logged.
- **Signal logging**
- **Data stores**

The Simulation Data Inspector tool supports the following data imported from MAT-files and the MATLAB base workspace:

- Simulink.Timeseries and MATLAB timeseries objects
- Simulink.SimulationData.Dataset or Simulink.ModelDataLogs objects
- Data in **Structure** with **time** format

The Simulation Data Inspector supports output from the following blocks:

- Scope (**Structure** with **time** and **Array** format)
- To File (**Timeseries** format)
- To Workspace (**Timeseries** or **Structure With Time** format)

Record Simulation Data

You can configure your model and the Simulation Data Inspector to automatically recorded data from a simulation run.

Configure Model for Recording Data

- 1 Set up your model to log signal data, as in “Export Signal Data Using Signal Logging” on page 48-21.
- 2 Select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 3 In the **Data Import/Export** pane:
 - Select the **Signal logging** check box.
 - Select the **Record and inspect simulation output** check box.
- 4 Click **Apply** and **OK**.

Configure Simulation Data Inspector

- 1 To open the Simulation Data Inspector, on the Simulink Editor toolbar, click the **Record** button arrow.



From the menu, select Simulation Data Inspector.

- 2 To specify run configuration options, in the Simulation Data Inspector, click the **Run Configuration** button.



- 3 To specify that new runs are added to the top of the run list, in the Run Configuration dialog box, select **Add new runs at top**.
- 4 To specify the rule for the run name, enter available tokens from the list and any other characters in the **Run naming rule** box.

- 5 Click **OK**.

For more information, see “Run Management Configuration” on page 17-38.

Simulate Model and Record a Run

- 1 On the Simulink Editor toolbar, click the **Record** button on. You can click the **Record** button to toggle on and off recording data for a simulation.

- 2 Simulate the model.

When the simulation is done, a notification bar appears in the Simulink Editor.

- 3 In the notification bar, click the link to open the Simulation Data Inspector.

The simulation data automatically appears in the Signal Browser table.

- 4 To record another simulation run, leave the **Record** button on, and simulate the model again.

When the simulation is done, the data appears as another new run. To configure the Simulation Data Inspector to overwrite a run, see “Overwrite a Run” on page 17-39.

- 5 When you are done importing data for your simulations, click the **Record** button off.

Note If the **Record** button is not on, the configuration parameter **Data Import/Export > Record and inspect simulation output** is no longer selected.

Related Examples


- “Append New Runs” on page 17-38
- “Specify a Run Naming Rule” on page 17-38
- “Overwrite a Run” on page 17-39

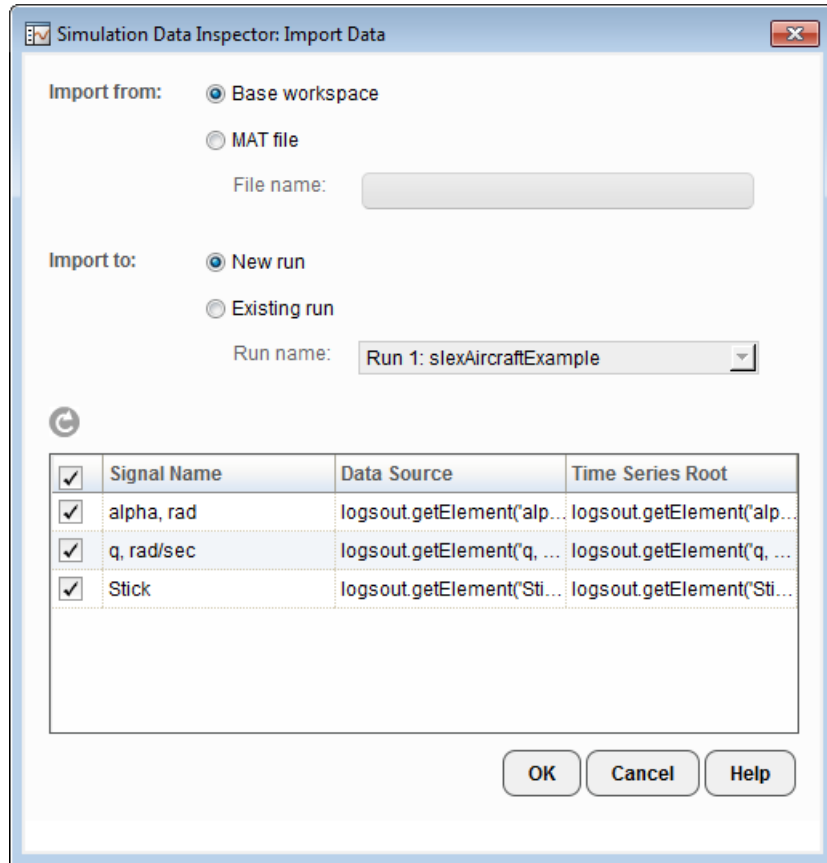
Import Logged Signal Data

Before importing data into the Simulation Data Inspector tool, you must have previously logged signal data to the base workspace or to a MAT-file. For information on how to log signal data, see “Export Signal Data Using Signal Logging” on page 48-21.

Import Signal Data from the Base Workspace

Using the Import Data tool, you can import data from the base workspace.

- 1 Open the Import Data tool by selecting the **Import Data** button  or selecting **File > Import Data**.
- 2 Select **Base Workspace**. Data from logouts appear in the table.




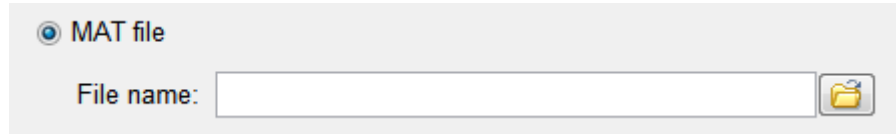
3 Select the **Import** check box to import a signal. Clear the **Import** check box for signals that you do not want. You can also use the **Select All** and **Clear All** buttons for mass selection or clearing of all signals.

4 Click **OK**. The selected signals are displayed in the Signal Browser table.

Import Signal Data from a MAT-File

With the Import Data tool, you can select a subset of signals from a MAT-file to import into the Simulation Data Inspector. Follow the steps to import a MAT-file.

- 1 Open the Import Data tool by selecting the **Import Data** button  or selecting **File > Import Data**.
- 2 For **Import From**, select **MAT file**. The **File name** parameter is enabled.




The screenshot shows a dialog box with a radio button selected next to the text "MAT file". Below this, there is a text input field labeled "File name:" followed by a folder icon button.

- 3 The Import Data tool locates a MAT-file in the current directory. Alternatively, click the **Open folder** button to browse for your MAT-file. The data from the MAT-file populates the data table.
- 4 Specify **Import To** by selecting **New Run** or **Existing Run**. If you select **Existing Run**, select a run from the **Run Name** list.
- 5 Select the **Import** check box to import signals. Clear the **Import** check box for signals that you do not want. You can also use the **Select All** and **Clear All** buttons for selection or clearing of all the signals.
- 6 Click **OK**.

Load Previously Recorded Data from a MAT-file

To load signal data from a MAT-file, which was created from the Simulation Data Inspector tool:

- 1 On the Simulation Data Inspector toolbar, click the **Open** button  or select **File > Open**.
- 2 Select the name of the MAT-file.
- 3 Click **OK**. Data stored in the MAT-file is displayed in the Simulation Data Inspector tool.

Inspect Signal Data

Overview

On the **Inspect Signals** tab you can inspect logged signal data. On the left pane you can open the Signal Browser table to view and configure signal and run properties (see “Customize the Simulation Data Inspector Interface” on page 17-41). On the right pane you can create different plot views. With a view of multiple plots, you can group signals to better view your data. For example, you can group the same signal from different simulation runs, group signals with a similar range of values, or normalize a subset of your signal data. The following examples use the `slexAircraftExample` model.

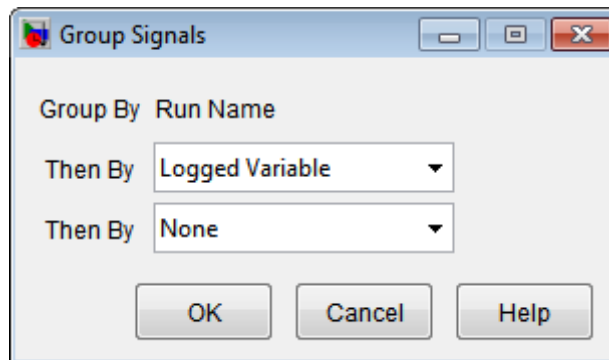
- “View Signal Data” on page 17-11
- “Create a View of Multiple Plots” on page 17-13
- “Explore Signal Data in a Multiple Plot View” on page 17-16
- “Replace a Run in a View” on page 17-17
- “Copy a View in the Inspect Signals Tab” on page 17-20
- “Delete a View in the Inspect Signals Tab” on page 17-21
- “Export a View in the Inspect Signals Tab” on page 17-21
- “Import a View in the Inspect Signals Tab” on page 17-22

View Signal Data

- 1** Configure a model to log signal data. This example uses the `slexAircraftExample` model. To log signals `q`, `rad/sec`, `Stick`, and `alpha`, `rad`, right-click each signal and select **Properties**. Select the **Log signal data** check box, and click **OK**.
- 2** Double-click the Pilot signal generator block. Set the **Wave form** to `sine`, and click **OK**.
- 3** In the Simulink Editor, click the **Record** button arrow, and select **Simulation Data Inspector**.

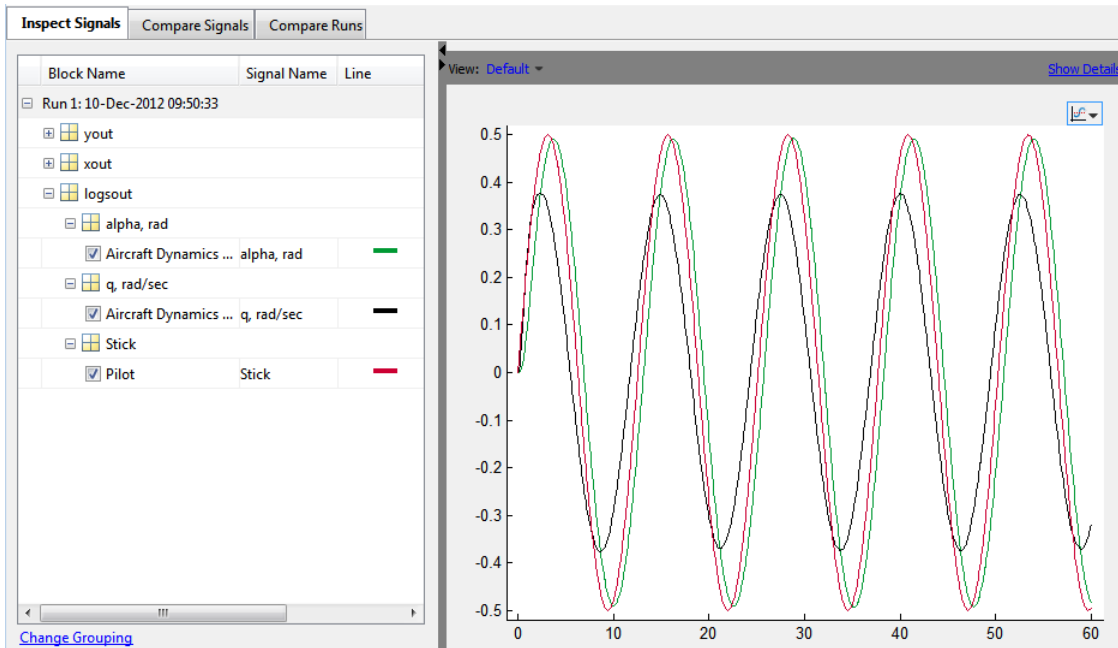


- 4 To record simulation data, click the **Record** button on.
- 5 Simulate the model. The Simulation Data Inspector tool records and displays the simulation data. Instead of recording data from a simulation, you can import signal data from previous simulation runs, see “Import Logged Signal Data” on page 17-7.
- 6 By default, the **Inspect Signals** tab displays the Signal Browser table, which contains a row for each signal, organized by simulation runs. You can expand or collapse any of the runs to view the logged signals in a run. For more information, see “Customize the Simulation Data Inspector Interface” on page 17-41.
- 7 Set up the columns of information to view about each signal. Right-click the Signal Browser table title bar and select **Columns** from the context menu. Select **Block Name**, **Signal Name**, and **Line**, clearing all other options.
- 8 Right-click the Signal Browser table title bar, and select **Group Signals**. In the Group Signals dialog box, set the first **Then By** parameter to **Logged Variable**, and then click **OK**.



- 9 Expand logouts and each signal in that group to view the logged signals.

- 10** Select the check box next to each signal you want to plot. On the right pane of the Simulation Data Inspector tool, the signal data is displayed in the graph.



- 11** To create a report of the results, see “Create Simulation Data Inspector Report” on page 17-32.

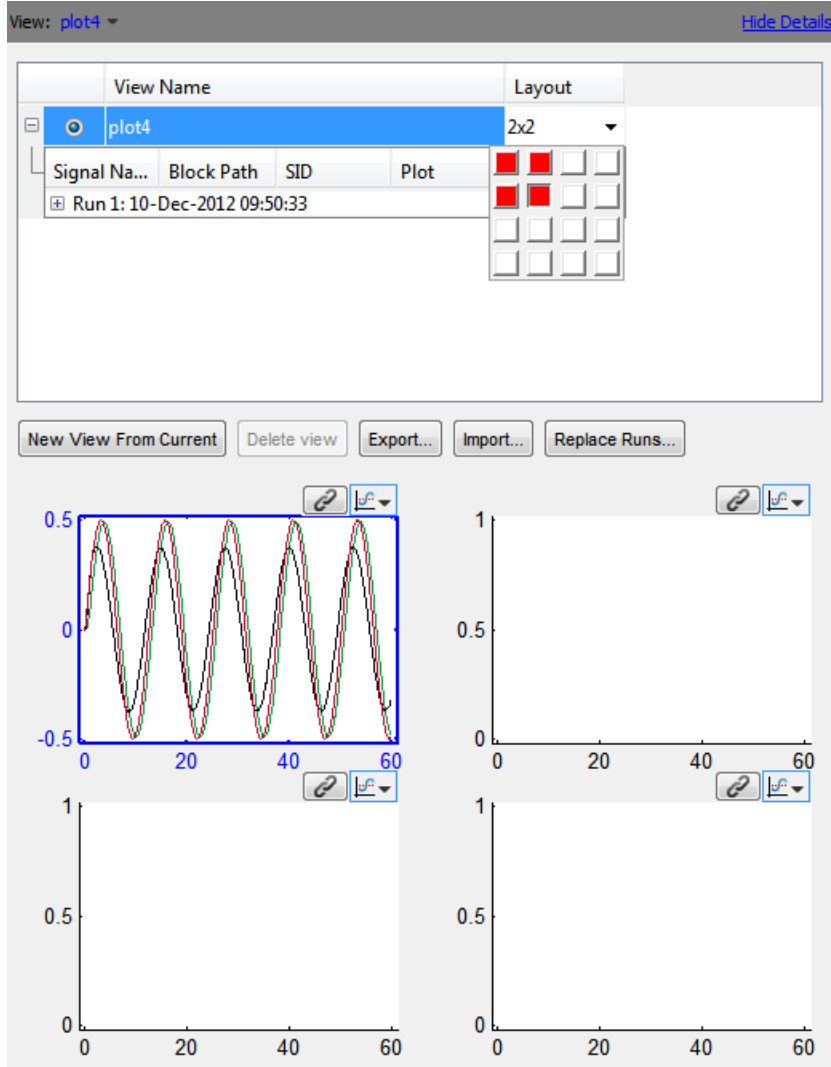
Create a View of Multiple Plots

Before creating a view of multiple plots, the Simulation Data Inspector tool must contain signal data. You can obtain signal data by either recording simulation data or importing data into the Simulation Data Inspector tool. For more information, see “Record Simulation Data” on page 17-5 or “Import Logged Signal Data” on page 17-7.

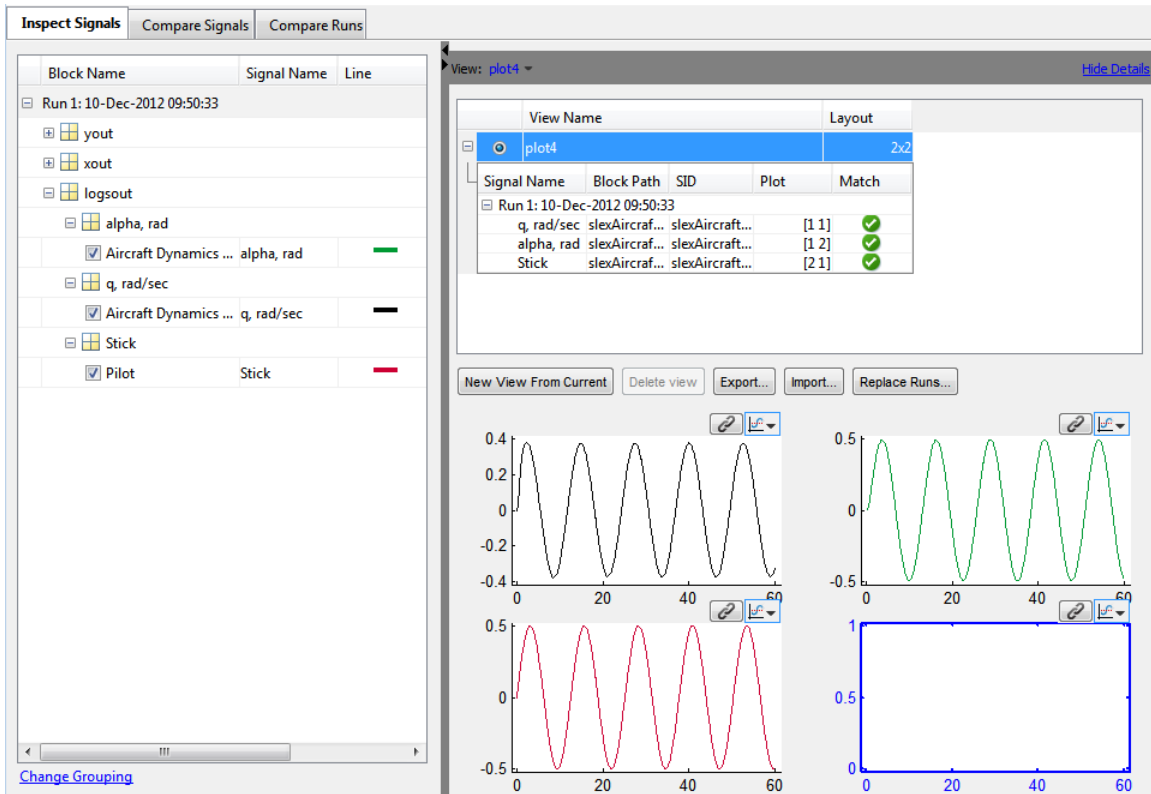
- 1** On the right pane, on the **View** toolbar, click **Show Details**.



- 2 In the **View Name** column, click the field which displays **Default**. Type in a new name for the view. In this example, the name is plot4.
- 3 In the **Layout** column, click the field that displays 1x1. A plot matrix opens. To create a view of 4 plots, select the 2x2 matrix.



- 4 To select a plot, click it. The selected plot is outlined in blue.
- 5 Once a plot is selected, on the left pane in the Signal Browser table, select a signal in the **Plot** column to add to the selected plot. In this example, a signal from Run 1 is displayed in each plot in the plot4 view.









For the selected plot [2,2], the signal or signals are shaded in the **Inspect Signals** table to indicate which signals are in the selected plot.

Explore Signal Data in a Multiple Plot View

Linked Plots in a View

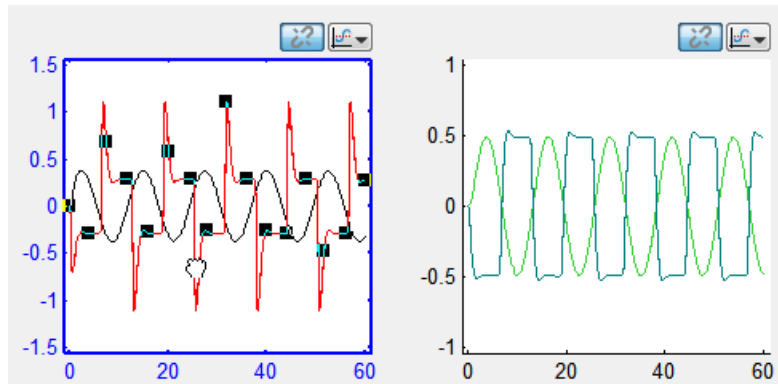
Plots within a view can be linked together to enable synchronized panning and zooming. When you create a view, by default, plots in a view are linked together. The following toolbar operations synchronize across linked plots:

- Pan along the time axis: 
- Zoom on the time axis: 
- Zoom on the time and data value axis: 
- Zoom out: 

To determine if a plot in a view is linked, at the top of each plot, locate the link icon . If a plot is unlinked, the broken link icon  is displayed and the plot pans and zooms independently.

Move Signals Between Plots in a View

To move a signal from one plot to another, click and hold a signal (markers appear).



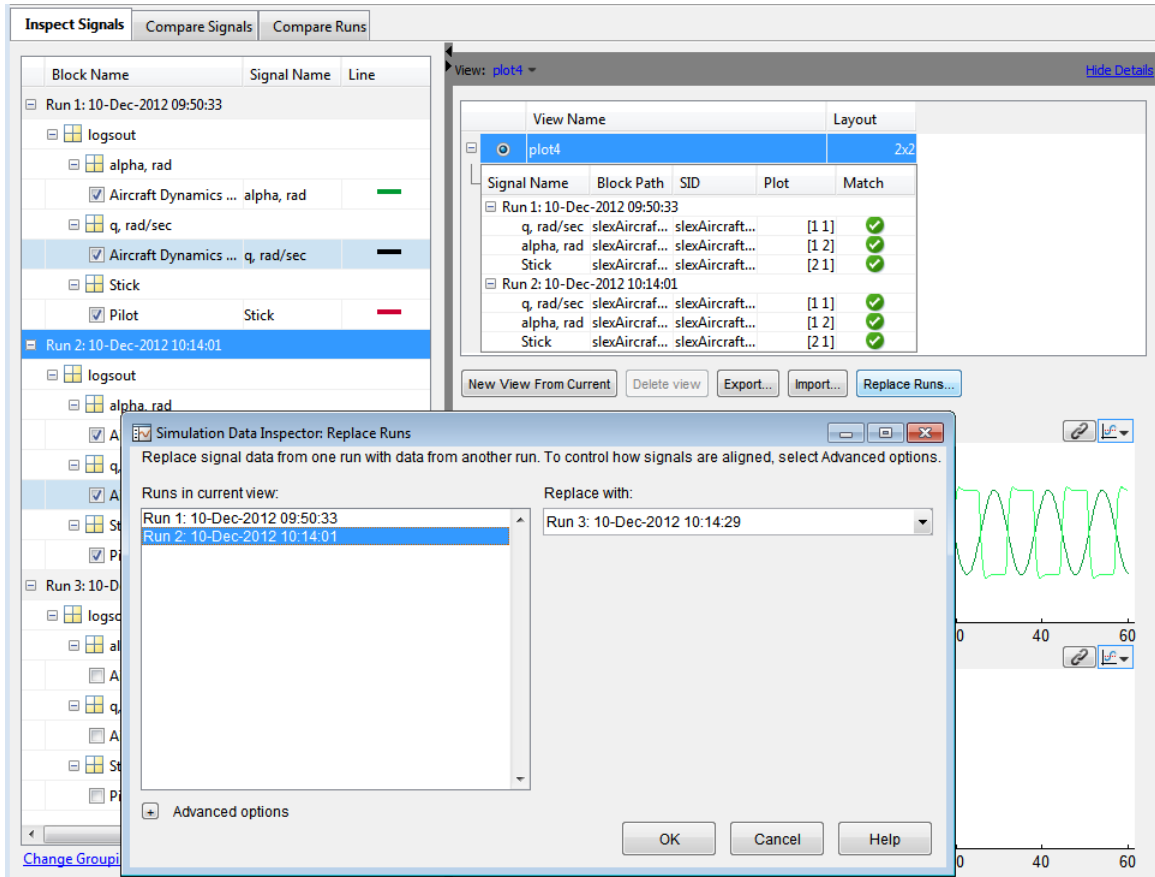
Drag the signal to another plot, and the signal is now displayed in the other plot.

For more information on working with plots, see “Modify a Plot in the Simulation Data Inspector Tool” on page 17-51.

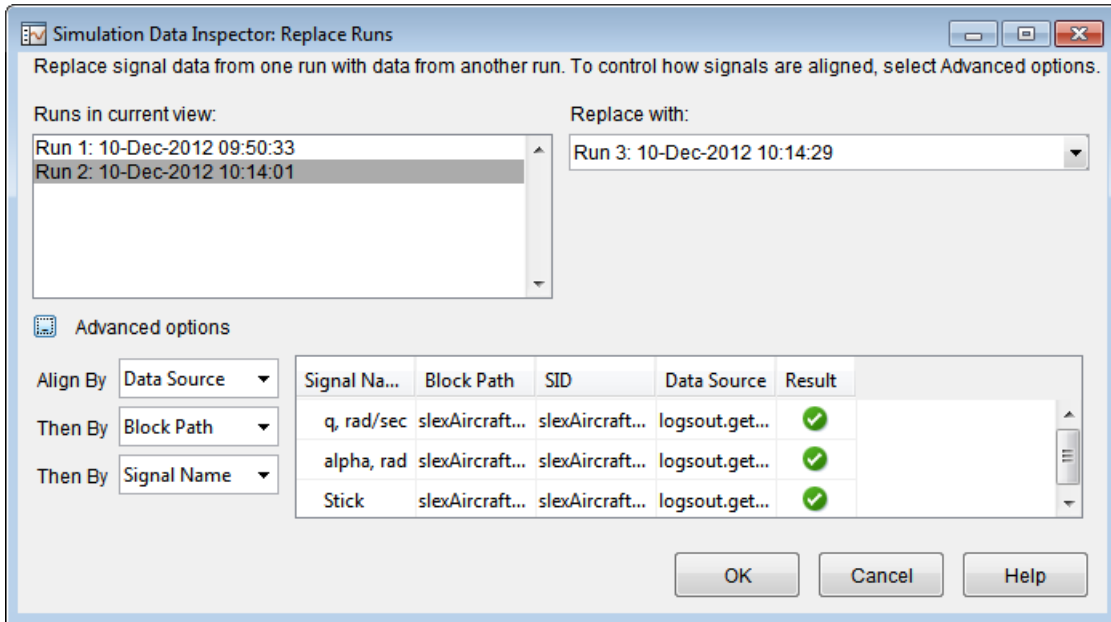
Replace a Run in a View

In this example, the Simulation Data Inspector tool contains three runs of data. To quickly swap out the data from one run with data from another run, do the following:

- 1** Select a view. In this example, the view is plot4.
- 2** Click **Replace runs...**, which opens the Replace runs dialog box.



- 3 To replace the signal data in Run 2 with the signal data from another run, in the **Runs in current view** list, select Run 2. Only the signals in the view are replaced.
- 4 In the **Replace with** list, select Run 3.
- 5 To specify how the signal data in Run 2 is replaced with Run 3, expand **Advanced options**. In this example, the Simulation Data Inspector first attempts to align the signals by Data Source, then by Block Path, and lastly by Signal Name. For more information on aligning signals, see “How the Simulation Data Inspector Tool Aligns Signals” on page 17-35.



6 Click **OK**. The data from Run 2 is replaced with data from Run 3.

The screenshot displays the 'Inspect Signals' tab. On the left, a tree view shows signal data for three runs. The right pane shows the 'View Details' table for 'plot4'.

View Name	Layout
plot4	2x2

Signal Name	Block Path	SID	Plot	Match
Run 1: 10-Dec-2012 09:50:33				
alpha, rad	slexAircraf...	slexAircraf...	[1 2]	✓
Stick	slexAircraf...	slexAircraf...	[2 1]	✓
q, rad/sec	slexAircraf...	slexAircraf...	[1 1]	✓
Run 3: 10-Dec-2012 10:14:29				
q, rad/sec	slexAircraf...	slexAircraf...	[1 1]	✓
alpha, rad	slexAircraf...	slexAircraf...	[1 2]	✓
Stick	slexAircraf...	slexAircraf...	[2 1]	✓

Below the table are four plots showing signal waveforms. The top-left plot shows alpha (rad) vs time (0-60). The top-right plot shows q (rad/sec) vs time (0-60). The bottom-left plot shows Stick vs time (0-60). The bottom-right plot shows Pilot vs time (0-60).

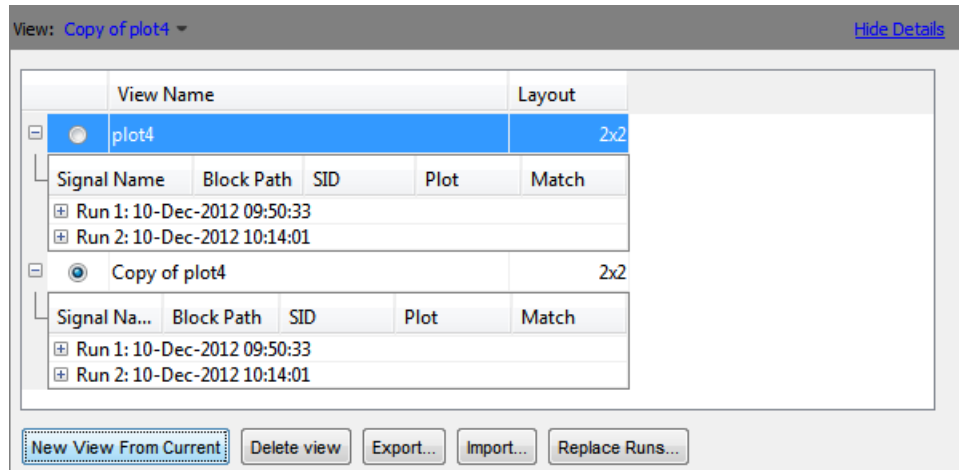
The View Details table now includes Run 3 instead of Run 2. Run 3 lists signals from Run 2. The **Match** column displays a ✓ for a signal that aligned with a signal in Run 3. If a signal did not align, the **Match** column displays a ✗.

Copy a View in the Inspect Signals Tab

To copy a view:

- 1 On the right pane, in the View toolbar, click **Show Details** to open the View Details table.
- 2 Select the view to copy. Be sure to select the option button next to the **View Name** in the table.

- 3 Click the **New View From Current** button. A new view is displayed in the View Details table



- 4 Select the new view, Copy of plot4, and rename it.

Delete a View in the Inspect Signals Tab

To delete a view:

- 1 On the right pane, in the View toolbar, click **Show Details** to open the View Details table.
- 2 In the View Details table, select the view to delete by selecting the corresponding option button.
- 3 Click the **Delete View** button.

Note The View Details table always includes one view.

Export a View in the Inspect Signals Tab

To export a view to a MAT-file:

- 1 On the right pane, in the View toolbar, click **Show Details** to open the View Details table.
- 2 Click the **Export** button and specify a file.
- 3 This saves the view, but not the data, into a MAT-file. To save the data and the view, see “Export Results in the Simulation Data Inspector Tool” on page 17-34.

Import a View in the Inspect Signals Tab

To reload a saved view from a MAT-file:

- 1 On the right pane, in the View toolbar, click **Show Details** to open the View Details table.
- 2 Click the **Import** button and specify a file.

Note This only imports the configuration of the view, which is how the signal data is plotted among the multiple plots. The actual signal data is not imported.

- 3 To include signal data in the Simulation Data Inspector tool, you can record simulation data (see “Record Simulation Data” on page 17-5), or import data (see “Import Logged Signal Data” on page 17-7).

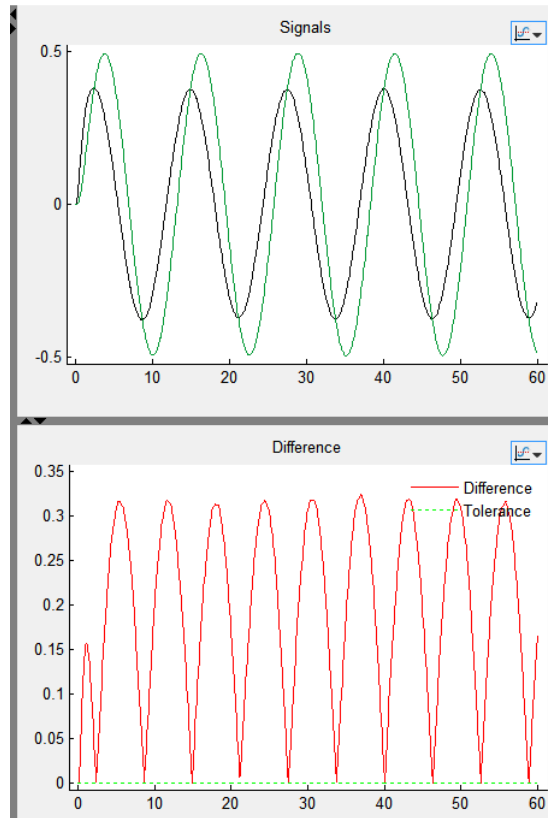
Compare Signal Data

In the Simulation Data Inspector tool, you can use the **Compare Signals** tab of the Signal Browser table for comparing two signals. To compare two signals:

- 1 Open the Simulation Data Inspector tool. See “Open the Simulation Data Inspector Tool” on page 17-41.
- 2 Record simulation data or import data into the Simulation Data Inspector tool. See “Record Simulation Data” on page 17-5 or “Import Logged Signal Data” on page 17-7.
- 3 If it is not already selected, select the **Compare Signals** tab. By default, Signal Browser table on the **Compare Signals** tab displays a row for each signal data, organized by simulation runs. For more information, see “Customize the Simulation Data Inspector Interface” on page 17-41.
- 4 “Specify the Line Configuration” on page 17-47 for the signals that you are comparing.
- 5 In the **Sig 1** column, click one signal for plotting. In this column, you can select only one signal.
- 6 In the **Sig 2** column, click one signal for plotting. In this column, you can select only one signal.

Inspect Signals	Compare Signals	Compare Runs	
	Sig 1	Sig 2	Line
	Run 1: 10-Dec-2012 09:50:33		
	logout		
	alpha, rad		
	<input checked="" type="radio"/>	<input type="radio"/>	—
	q, rad/sec		
	<input type="radio"/>	<input checked="" type="radio"/>	—
	Stick		
	<input type="radio"/>	<input type="radio"/>	—

The **Signals** graph displays the signal data and the **Difference** graph displays the difference and tolerance values.



For information on manipulating the graphs, see “Customize the Simulation Data Inspector Interface” on page 17-41.

- 7 To create a report of the results, see “Create Simulation Data Inspector Report” on page 17-32.

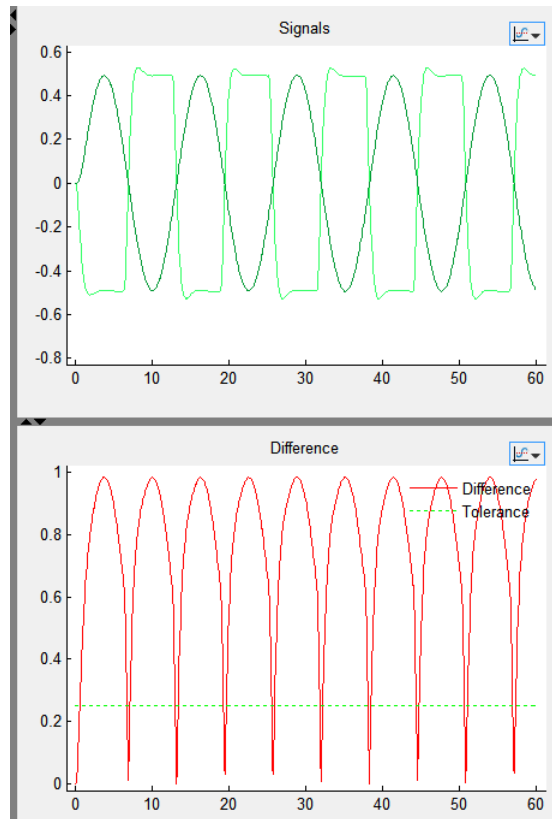
Comparison of One Signal From Multiple Simulations

In the Simulation Data Inspector tool, you can use the **Compare Signals** tab to compare the same signal from two different simulation runs. To compare two data runs for a signal:

- 1** Open the Simulation Data Inspector tool. See “Open the Simulation Data Inspector Tool” on page 17-41.
- 2** Record simulation data or import data for two simulation runs. For more information, see “Record Simulation Data” on page 17-5 or “Import Logged Signal Data” on page 17-7.
- 3** Select the **Compare Signals** tab. By default, the Signal Browser table on the **Compare Signals** tab displays a row for each signal data, organized by simulation runs. For more information on modifying the Signal Browser table, see “Customize the Simulation Data Inspector Interface” on page 17-41.
- 4** “Specify the Line Configuration” on page 17-47 for the signals that you are comparing.
- 5** To change the relative tolerance and absolute tolerance, add the **Rel Tol** and **Abs Tol** columns to the Signal Browser table. Double-click the corresponding fields for a signal and type in a value.
- 6** In the **Sig 1** column, click one signal from the first simulation run for plotting. In this column, you can select only one signal.
- 7** In the **Sig 2** column, click a signal from another simulation run for plotting. In this column, you can select only one signal.

Inspect Signals		Compare Signals	Compare Runs		
Sig 1	Sig 2	Line	Abs Tol	Rel Tol	
[-] Run 1: 10-Dec-2012 09:50:33					
[-] [grid] logout					
[-] [grid] alpha, rad					
<input checked="" type="radio"/>	<input type="radio"/>	—	.25	.05	
[-] [grid] q, rad/sec					
<input type="radio"/>	<input type="radio"/>	—	0	0	
[-] [grid] Stick					
<input type="radio"/>	<input type="radio"/>	—	0	0	
[-] Run 2: 10-Dec-2012 10:14:01					
[-] [grid] logout					
[-] [grid] alpha, rad					
<input type="radio"/>	<input checked="" type="radio"/>	—	.25	.05	
[-] [grid] q, rad/sec					
<input type="radio"/>	<input type="radio"/>	—	0	0	
[-] [grid] Stick					
<input type="radio"/>	<input type="radio"/>	—	0	0	

Once two signals are selected, the **Signals** graph displays the signal data and the **Difference** graph displays the difference and tolerance values.



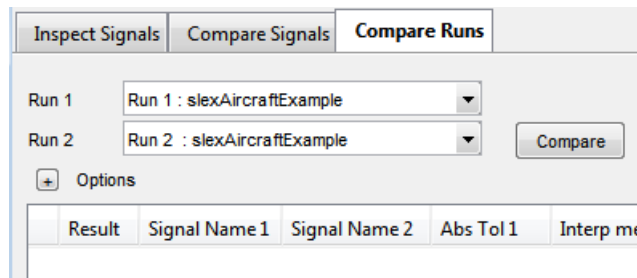
For information on manipulating the graphs, see “Customize the Simulation Data Inspector Interface” on page 17-41.

- 8 To create a report of the results, see “Create Simulation Data Inspector Report” on page 17-32.

Compare All Logged Signal Data From Multiple Simulations

In the Simulation Data Inspector tool, you can use the **Compare Runs** tab to compare all signal data from two different simulation runs. To compare two runs:

- 1 Open the Simulation Data Inspector tool. See “Open the Simulation Data Inspector Tool” on page 17-41.
- 2 Record simulation data or import data from multiple simulation runs, for more information, see “Record Simulation Data” on page 17-5 or “Import Logged Signal Data” on page 17-7.
- 3 Select the **Compare Runs** tab.
- 4 From the **Run 1** and **Run 2** drop-down lists, select two different runs.



Note The number in a column name indicates the run number. For example, **Abs Tol 1** refers to the absolute tolerance values for signals of the run specified for **Run 1**. **Abs Tol 2** refers to the absolute tolerance values for signals of the run specified for **Run 2**.

- 5 Select the **Options** button to specify the signal alignment.

Options

Align By

Then By

Then By

The Simulation Data Inspector aligns signals according to **Align By** and **Then By** parameters. For more information on signal alignment, see “How the Simulation Data Inspector Tool Aligns Signals” on page 17-35.

- Click the **Compare** button. The Comparison Results table lists all signals from **Run 1** with a result. In this example, the comparison results of the aligned signals did not match.

Inspect Signals Compare Signals **Compare Runs**

Run 1

Run 2

Options

Align By

Then By

Then By

Result	Signal Name 1	Signal Name 2	Abs Tol 1	Interp met...	Aligned By	Plot
	alpha, rad	alpha, rad		0.0linear	Data Source	<input type="radio"/>
	q, rad/sec	q, rad/sec		0.0linear	Data Source	<input type="radio"/>
	Stick	Stick		0.0linear	Data Source	<input type="radio"/>

The Simulation Data Inspector only compares signals from **Run 1** that are aligned with a signal from **Run 2**. If a signal from **Run 1** did not align with a signal from **Run 2**, then the **Run 1** signal is listed in the Comparison Results table with a warning .

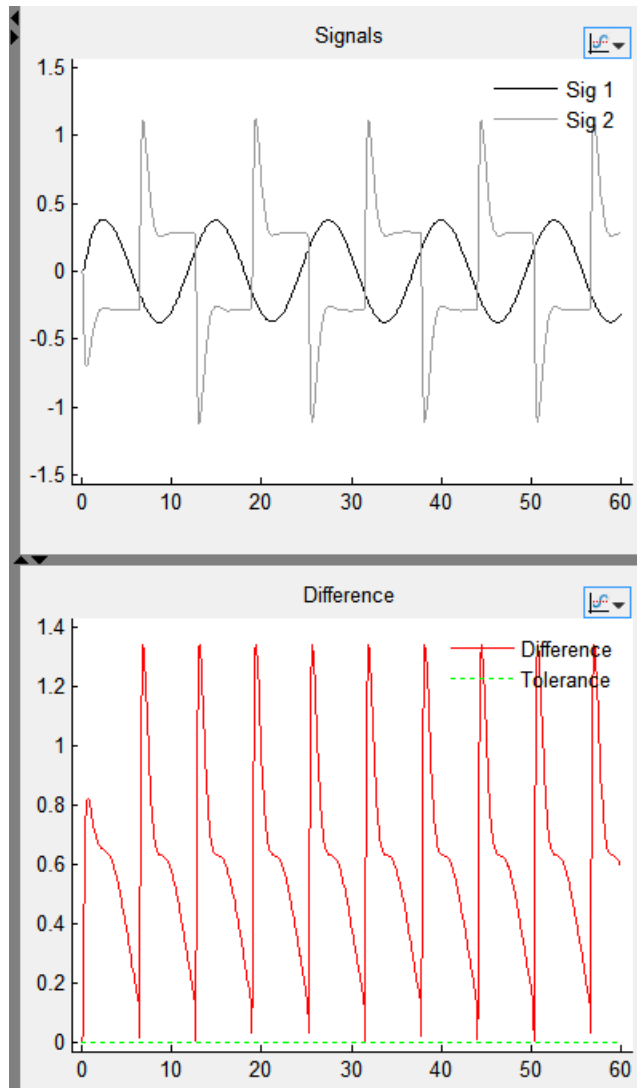
- To modify the relative tolerance and absolute tolerance, add the **Abs Tol 1** column to the Signal Browser table. Double-click the corresponding fields for a signal and type in a value.

Note Tolerances specified in the other tab views do not carry over to the **Compare Runs** tab.

- To see a signal that is within the acceptable tolerance across two simulation runs, set the absolute tolerance in column **Abs Tol 1** to 1.0.

Result	Signal Name 1	Signal Name 2	Abs Tol 1	Interp met...	Aligned By	Plot
✓	alpha, rad	alpha, rad	1.0	linear	Data Source	<input type="radio"/>
✗	q, rad/sec	q, rad/sec	0.0	linear	Data Source	<input type="radio"/>
✗	Stick	Stick	0.0	linear	Data Source	<input type="radio"/>

- To plot a signal, in the plot column select the option button for **q, rad/sec**. The **Signals** graph displays the signal data for the two runs. The **Difference** graph displays the difference and tolerance values.



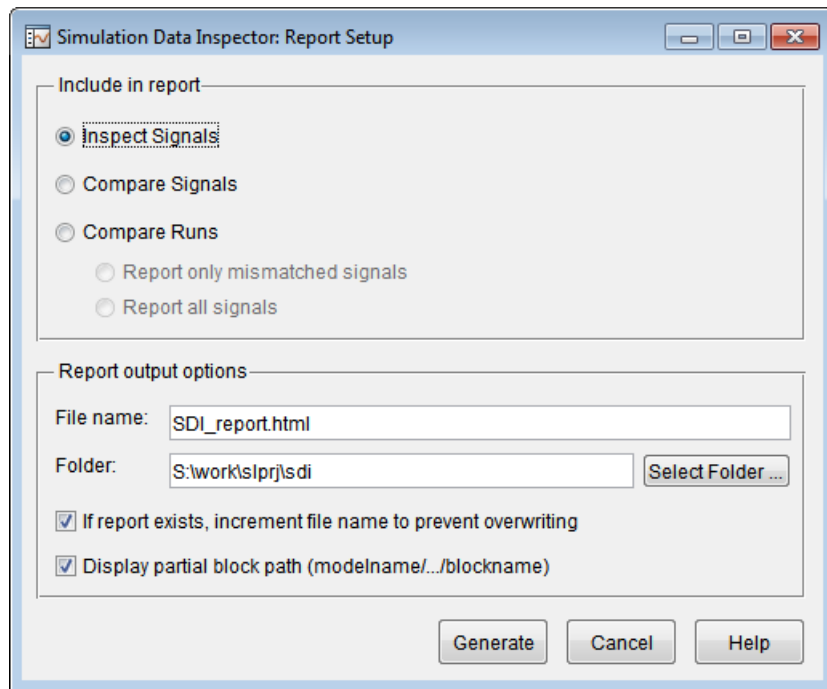
For information on manipulating the graph, see “Customize the Simulation Data Inspector Interface” on page 17-41.

- 10** To create a report of the results, see “Create Simulation Data Inspector Report” on page 17-32.

Create Simulation Data Inspector Report

To create a report of the current view and data in the Simulation Data Inspector:

- 1 In the Menu Bar, select **File > Generate Report**. The Report Setup dialog box opens.



- 2 In the **Include in report** section, specify which tab to include in the report. For the **Compare Runs** tab, you can select **Report only mismatched signals only** to shorten the report or select **Report all signals** which includes all logged signals in the designated runs.
- 3 Specify the **File name** and **Folder**, or use the default.

- 4** Optionally, select **If report exists, increment file name to prevent overwriting**, which appends and increments a number to the file name of the report to preserve earlier versions of the report file.
- 5** Optionally, select **Display partial block path (modelname/.../blockname)**, which shortens the block path string generated for the report.
- 6** Click **Generate**. The report opens automatically.

Export Results in the Simulation Data Inspector Tool

The Simulation Data Inspector tool provides the capability to save data collected by the tool to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

Save Data to a MAT-File

To save signal data and views to a MAT-File, do the following:

- 1** Click **Save** or select **File > Save**.
- 2** Type the name of the file.
- 3** Click **OK**. Your data and views are now saved in a MAT-file that you can load back into the Simulation Data Inspector.

How the Simulation Data Inspector Tool Aligns Signals

When the Simulation Data Inspector tool aligns signals across simulation runs, it attempts to match signals between runs by using metadata stored for each signal. You can specify the metadata used to align signals. The metadata options are: **Data Source**, **Path**, **SID**, and **Signal Name**. By default, the Simulation Data Inspector tool is configured to first align signals by **Data Source**, then by **Path**, and then by **Signal Name**.



To modify the signal alignment specifications see “Inspect Signals: Aligning Signals for Replacing a Run” on page 17-35 or “Compare Runs: Aligning Signals for Comparing Signal Data” on page 17-36.

Inspect Signals: Aligning Signals for Replacing a Run

To modify how signals are aligned on the **Inspect Signals** tab, do the following:

- 1** Click the **Inspect Signals** tab.
- 2** On the right pane, click **Show details**.
- 3** Click the **Replace runs...** button. The Replace Runs dialog opens.
- 4** Select **Advanced options**, which displays the **Align By** and **Then By** parameters.
- 5** Specify the **Align By** and **Then By** parameters from the drop-down lists.

When replacing a run, the Simulation Data Inspector tool attempts to align only the signals selected in the current view. Once those signals are aligned, in the view, the aligned signal data from the first run is replaced with signal data from another run. The View Details table displays the signals from the replaced run as follows:




Status	Result
	<ul style="list-style-type: none"> • Signal aligned • Signal data is replaced in the view
	Signal not aligned

Compare Runs: Aligning Signals for Comparing Signal Data

To modify how signals are aligned on the **Compare Runs** tab, do the following:

- 1 Click the **Compare Runs** tab.
- 2 Select the **Options** button, which displays the **Align By** and **Then By** parameters.
- 3 Specify the **Align By** and **Then By** parameters from the drop-down lists.
- 4 Click **Compare**, to align the signal data and compare the aligned signal data.

After signals are aligned from **Run 1** with signals from **Run 2**, the Simulation Data Inspector tool compares only the aligned signals. The Comparison Results table displays the results of all signals from **Run 1** as follows:

Status	Comparison Result
	<ul style="list-style-type: none"> • Signal aligned • Signal data from two runs matched within the tolerance
	<ul style="list-style-type: none"> • Signal aligned • Signal data from two runs did not match within the tolerance
	Signal from Run 1 did not align with a signal from Run 2

How the Simulation Data Inspector Tool Compares Time Series Data

To compare time series data, the Simulation Data Inspector:

- 1** Converts Simulink time series data to MATLAB time series data.
- 2** Aligns the time vectors using the default synchronization method union. To change the synchronization method for a signal, add the **Sync Method** column to the Signal Browser table and choose a method.
- 3** Aligns the data vectors using the default interpolation method, zoh (zero-order hold). To change the interpolation method for a signal, add the **Interpolation Method** column to the Signal Browser table and choose a method.
- 4** Differences the data.
- 5** Applies the specified tolerances for plotting the **Difference** . For more information, see “How Tolerances Are Applied” on page 17-37.

How Tolerances Are Applied

The default values for the relative tolerance and absolute tolerance for a signal is 0. If you specify tolerances, then the Simulation Data Inspector tool calculates the tolerances as follows:

```
tolerance = max(absoluteTolerance, relativeTolerance*abs(baselineData));
```

To change the relative tolerance and absolute tolerance, add the **Rel Tol** and **Abs Tol** columns to the Signal Browser table. Double-click the corresponding fields for a signal and type in a value.

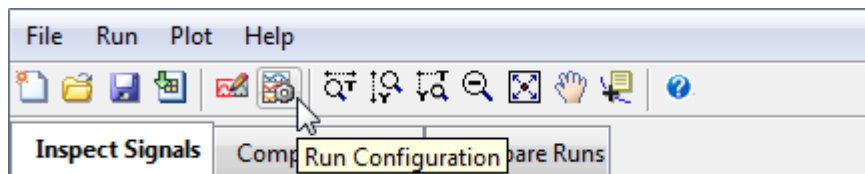
Run Management Configuration

You can configure the Simulation Data Inspector to:

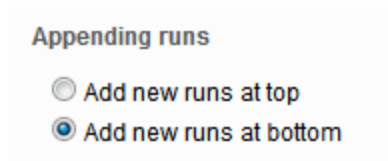
- “Append New Runs” on page 17-38
- “Specify a Run Naming Rule” on page 17-38
- “Overwrite a Run” on page 17-39

Append New Runs

You can specify whether to add new runs in the Signal Browser table at the top or bottom of the run list. To open the Run Configuration dialog box, click the **Run Configuration** button.



In the Run Configuration dialog box, the default is set to add new runs to the bottom of the run list.



To append new runs at the top of the list, click **Add new runs at top**.

Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click the **Run Configuration** button. In the Run Configuration dialog box, the default value for the **Run naming rule** is `Run <run_index> : <model_name>`.

Naming rules

Run naming rule: Restore Defaults

List of tokens:

- <run_index> Sequential number of each run
- <model_name> Name of Simulink model
- <time_stamp> Run creation time
- <sim_mode> Simulation mode for run

To change the run name, enter available tokens from the list and any other regular characters. For example, to include all four tokens, enter the following in the **Run naming rule** box:

```
Run <run_index> : <model_name> : <time_stamp> : <sim_mode>
```

For model `slexAircraftExample`, the run name appears as follows:

```
Run 1 : slexAircraftExample : 17-Dec-2012 12:35:52 : normal
```

Note If you are overwriting a run, the name of the previous run is used, not the **Run naming rule** specification. However, if the `<time_stamp>` token was used in the previous run, the value of this token updates with the new time.

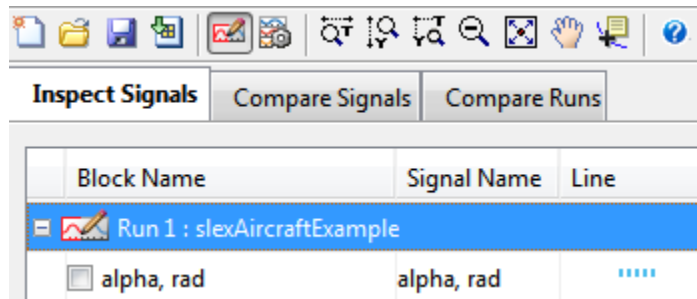
Overwrite a Run

Overwrite the simulation data for a run in the next recorded simulation.

- 1 In the Signal Browser table, click the run name you want to overwrite.
- 2 Click the **Overwrite Run** button.



The **Overwrite Run** icon appears on the run to be overwritten.



3 In the Simulink Editor, click the **Record** button on.

4 Simulate the model.

The data for the selected run is replaced with the new simulation run.

Customize the Simulation Data Inspector Interface

Overview

The Simulation Data Inspector tool provides the capability to inspect and compare time series data. There are several methods for launching the Simulation Data Inspector tool, see “Open the Simulation Data Inspector Tool” on page 17-41 to choose the method that best supports your workflow. The Simulation Data Inspector window includes the following elements:

- Menu Bar
- “Toolbar” on page 17-53
- Three tabs: **Inspect Signals**, **Compare Signals**, and **Compare Runs**

The Signal Browser table appears on each of the three tabs. You can customize the information displayed in the Signal Browser table by performing the following tasks:

- “Add/Delete a Column in the Signal Browser Table” on page 17-42
- “Modify Grouping in Signal Browser Table” on page 17-45
- “Select a Run or Signal Option in the Signal Browser Table” on page 17-48
- “Display Run Properties” on page 17-51

The Plot View displays in the right pane of the Simulation Data Inspector tool. To modify a plot, refer to the following:

- “Specify the Line Configuration” on page 17-47
- “Modify a Plot in the Simulation Data Inspector Tool” on page 17-51

Open the Simulation Data Inspector Tool

To launch the Simulation Data Inspector tool, choose one of the following methods:

- **MATLAB command-line:** Enter

```
Simulink.sdi.view
```

- **Simulink Editor:** Click the **Record** button arrow.



Select **Simulation Data Inspector** from the list.

Why Is the Simulation Data Inspector Tool Empty?

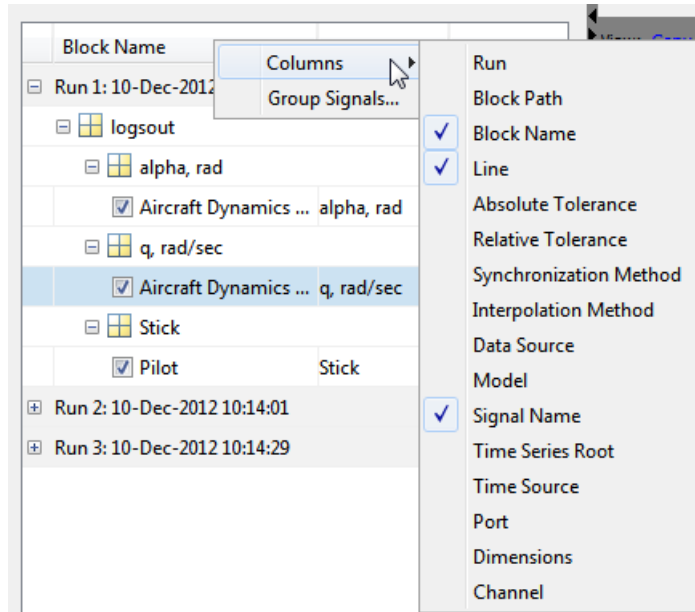
There are several methods for populating the Simulation Data Inspector with data.

- If you are using the record button to simulate and record data, you must specify your model to export signals. For more information, see:
 - “Export Simulation Data” on page 48-4
 - “Export Signal Data Using Signal Logging” on page 48-21After your model is configured to export signal data, see “Record Simulation Data” on page 17-5.
- If you want to view previously logged signal data, see “Import Logged Signal Data” on page 17-7.

For a list of Simulink data export formats that are not supported in the Simulation Data Inspector, see “Limitations of the Simulation Data Inspector” on page 17-56.

Add/Delete a Column in the Signal Browser Table

To add or remove a column, right-click the Signal Browser table title bar. Select **Columns** and click an option on the list. The column is displayed in the table.



Column Options for the Inspect Signals and Compare Signals Tabs

Column Option	Value
Run	Name of a simulation run
Block Path	Path to the source block for the signal
Block Name	Name of the source block
Line	Line style
Absolute Tolerance	Positive number (user-specified)
Relative Tolerance	Positive number (user-specified)
Synchronization Method	Method to align time vector: union, intersection, uniform (user-specified)
Interpolation Method	Method to align data: zoh, linear (user-specified)

Column Options for the Inspect Signals and Compare Signals Tabs (Continued)

Column Option	Value
Data Source	Name for the data (logout.Stick.Data)
Model	Model name for the signal data
Signal Name	Signal name for the data (Stick)
Time Series Root	String signifying the name of the Simulink.Timeseries object (logout.Stick.Time)
Time Source	String signifying the array containing the time data (logout.Stick.Time)
Port	Index of the output port that emits the signal logged
Dimensions	Number of dimensions of the signal
Channel	Channel of matrix data

Column Options for Compare Runs Tab

Column Option	Value
Result	Result of the comparison for the signal across the specified runs
Block Path	<ul style="list-style-type: none"> • Run 1: Block Path • Run 2: Block Path
Data Source	String identifying the data source <ul style="list-style-type: none"> • Run 1: Data Source • Run 2: Data Source

Column Options for Compare Runs Tab (Continued)

Column Option	Value
SID	“Simulink Identifier” <ul style="list-style-type: none"> • Run 1: SID • Run 2: SID
Signal Name	Signal names from two runs for comparison: <ul style="list-style-type: none"> • Run 1: Signal Name • Run 2: Signal Name
Absolute Tolerance	Positive number (user-specified)
Relative Tolerance	Positive number (user-specified)
Synchronization Method	Method to align time vector: union, intersection, uniform (user-specified)
Interpolation Method	Method to align data: zoh, linear (user-specified)
Channel 1	Channel of matrix data

After selecting a column option, the new column is added to the table in the order that it appears in the options list.

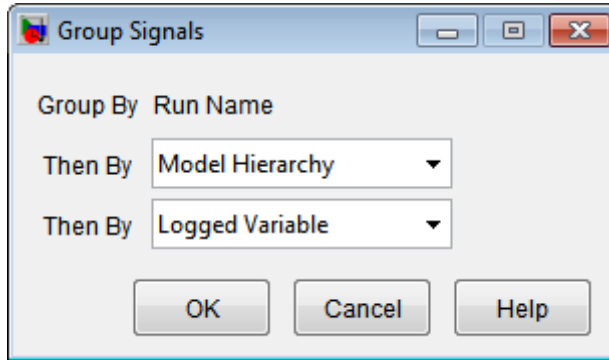
Modify Grouping in Signal Browser Table

You can customize the organization of your logged data in the Signal Browser table. By default, data is first organized by run. You can then organize your data by model hierarchy, logged variable, or no hierarchy.

If your model contains referenced models to view, you can group your data by model hierarchy and then by the logged variables. To change the grouping in the Signal Browser table:

- 1 At the bottom of the Signal Browser table, click the Change Grouping link.

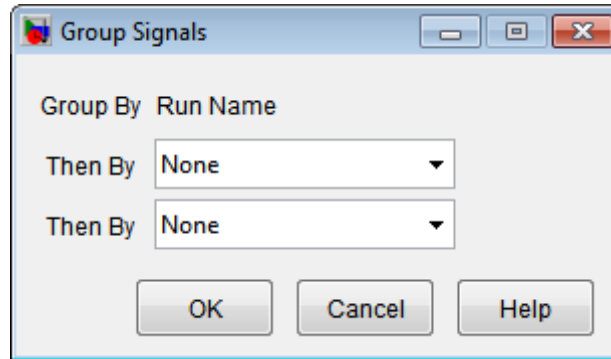
- 2** In the Group Signals dialog box, in the first **Then By** list, select Model Hierarchy.
- 3** In the second **Then By** list, select Logged Variable.



- 4** Click **OK**. The Signal Browser table groups the signal data by model and then by the logged variables.

Block Name	Signal Name	Line
Run 1: sldemo_mdref_dsm		
sldemo_mdref_dsm		
yout		
Out1		—
Out1		—
logsmout		
A	bot_out	—
A1	bot2_out	—
Switch	swich_out	—
sldemo_mdref_dsm_bot		
dsmout		
DSM	RefSignalVal	—

To remove the hierarchy and display a simple list of logged signals, you can select None in the Group Signals dialog box.



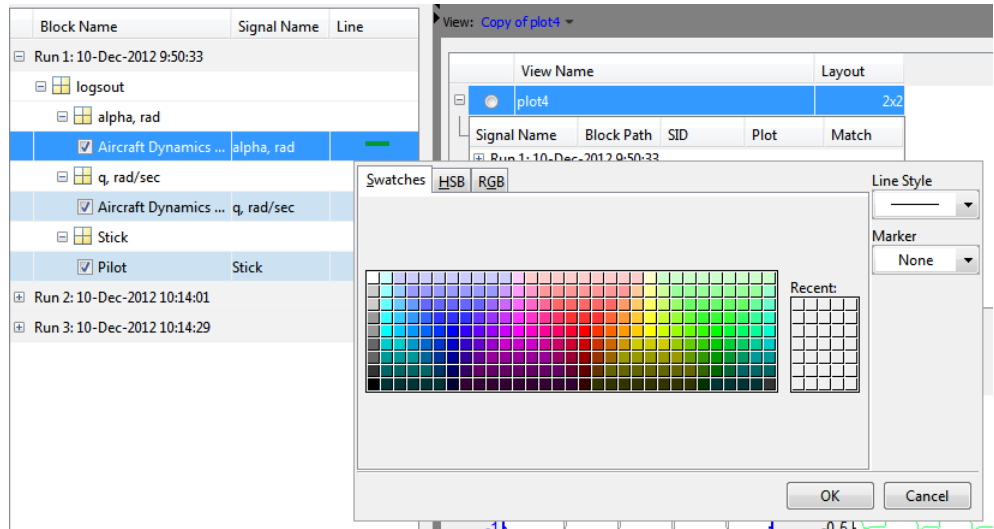
Rename a Run

To rename a run name:

- 1 If the Signal Browser table is not grouped by run, right-click the table and in the menu select **Group By > Run**.
- 2 Double-click the Run row.
- 3 Type the new run name and press **Enter**.

Specify the Line Configuration

- 1 Click in the **Line** column of a signal.

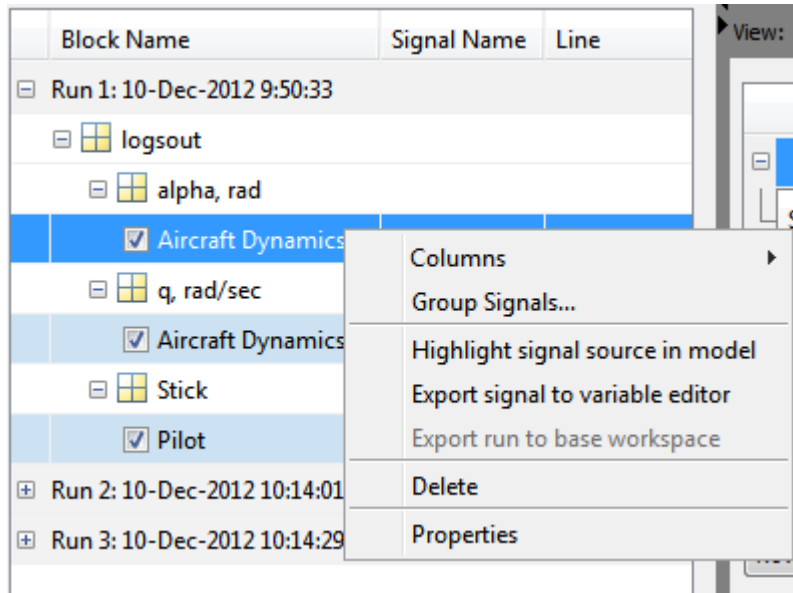


2 Specify the color, **Line Style**, and **Marker** for the signal.

3 Click **OK**.

Select a Run or Signal Option in the Signal Browser Table

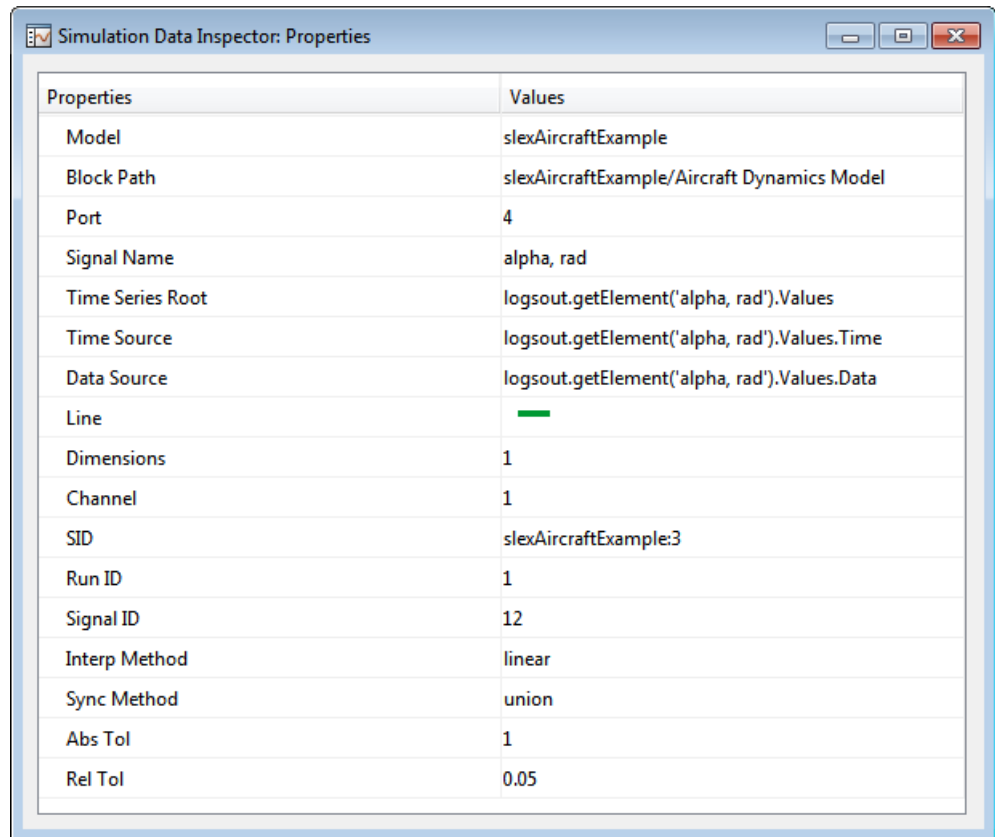
In the Signal Browser table, right-click a run or signal for a menu list of options.



To...	Select...
Display the list of Column options. See Column Options for the Inspect Signals and Compare Signals Tabs on page 17-43	Columns
Group signal data by the specified options: Run, Block Path, Data Source, Model, and Signal Name (this option does not appear in the Compare Runs tab)	Group By
Highlight the source of the selected signal in the model diagram	Highlight signal source in model
Open the Variable Editor and display the selected signal data	Export signal to variable editor
Save signal data in the run to the base workspace	Export run to base workspace

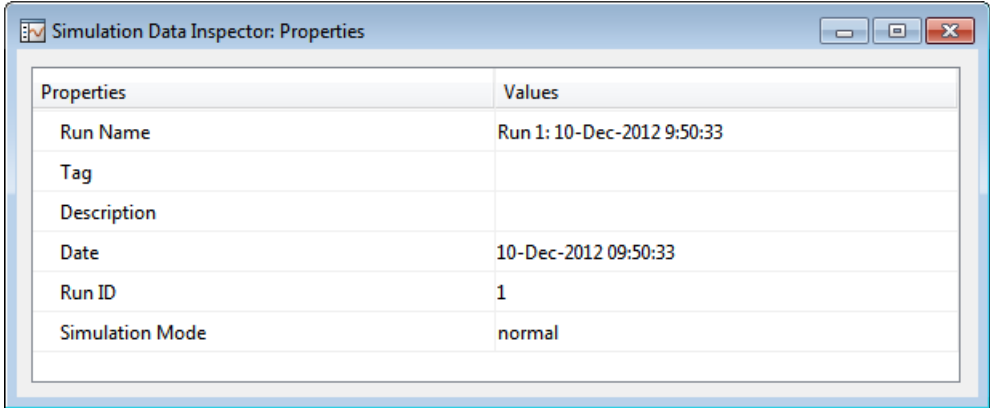
To...	Select...
Delete the highlighted signal in the Signal Browser table (this option does not appear in the Compare Runs tab)	Delete
Display the signal properties of the selected signal	Properties

The Properties dialog box displays the following signal information.




Display Run Properties

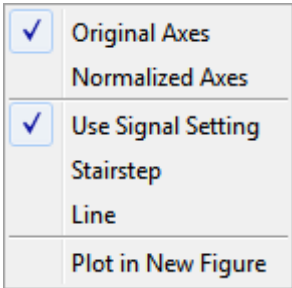
In the Signal Browser table, right-click a run name to view a list of options. To open the Run Properties dialog box, from the options list, select **Properties**.




Modify a Plot in the Simulation Data Inspector Tool

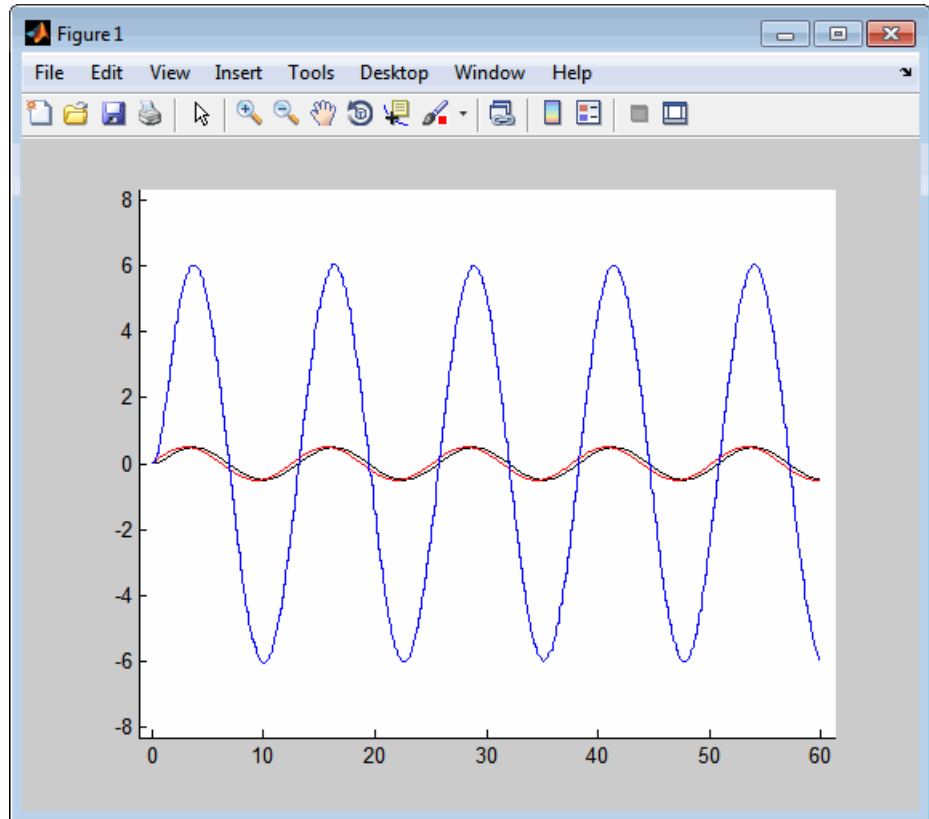
To modify a plot, you can:

- **Toolbar:** Select icons to zoom into data, move the plot, or add a data point to the plot. For more information on these icons, see the “Toolbar” on page 17-53 section.
- **Plot options:** Click  on the plot to select alternatives for plotting the data.



To...	Simulation Data Inspector...
Original Axes	Plots the data according to the maximum and minimum values of the data points. (default)
Normalized Axes	Normalizes the data for each signal from -1 to 1 along the y-axis. If the data is already within that range, the data is displayed as a constant at 1.
Stairstep	Plots the data as a stairstep plot. (default)
Line	Interpolates the data points and produces a linear plot.
Plot in New Figure	Launches the graph in a MATLAB figure window.

- **New figure:** Click  on the plot and select **Plot in New Figure**. The plot opens in a new figure window.
















For more information on plotting and customizing your data plots using the GUI tools, see “Figures, Plots, and Graphs”.

To view an example, see “Create, Save, and Print a Figure”.

Toolbar

The toolbar contains the following command buttons.

To...	Click...
Clear the Simulation Data Inspector of all data.	New 
Open a MAT-file previously saved from the Simulation Data Inspector.	Open 
Save data in Simulation Data Inspector to a MAT-file.	Save 
Open the Import Data tool.	Import Data 
Mark a run to overwrite on the next simulation/record.	Overwrite Run 
Configure appending new runs to the top or bottom of the run list. Specify a run naming rule.	Run Configuration 
Zoom in along the time axis. After selecting the icon, on the graph, click and hold the left mouse button and drag the mouse to select an area to enlarge.	Zoom in T 
Zoom in along the data value axis. After selecting the icon, on the graph, click and hold the left mouse button and drag the mouse to select an area to enlarge.	Zoom in Y 
Zoom in on a section of the graph along both axes. After selecting the icon, on the graph, click and hold the left mouse button, and drag the mouse to select an area to enlarge.	Zoom in T and Y 

To...	Click...
Zoom out from the graph. After selecting the icon, click the graph to incrementally zoom out.	Zoom Out 
Fit the plot to the graph. After selecting the icon, click the graph to enlarge the plot to fill the graph.	Fit to View 
Move the plot in the graph up, down, left, or right. After selecting the icon, on the graph, click and hold the left mouse button and move the mouse to the area of the graph that you want to view.	Pan 
Display the T and Y values of a data point in the plot. After selecting the icon, click a point on the line to view a data point.	Data Cursor 

Limitations of the Simulation Data Inspector

- The following Simulink data export formats are not supported if the time is not logged:
 - Structure
 - Array
- The following Simulink data types are not supported:
 - Complex data, int64, and uint64
 - Enumerated data types
- The Simulation Data Inspector tool supports up to 3-dimensional time series data.
- When you simulate and record data from the Simulink Editor, after you open the Simulation Data Inspector, the recorded simulation data automatically appears in the Signal Browser table. However, when you simulate a model at the command line using `sim`, you must use the `Simulink.sdi.createRun` function to view the simulation data in the Simulation Data Inspector. For more information, see “Create a Run in the Simulation Data Inspector” on page 17-60.

Record and Inspect Signal Data Programmatically

Overview

Using the Simulation Data Inspector API, you can plot signal data, compare two signals, and compare two simulation runs of data. You can use the `Simulink.sdi.createRun` function to add simulation output data to the Simulation Data Inspector. Once the Simulation Data Inspector contains signal data, you can perform the following tasks:

To...	Use...
View signal data, open the Simulation Data Inspector tool	<code>Simulink.sdi.view</code>
Compare the data of two signals	<code>Simulink.sdi.compareSignals</code>
Compare the output of two simulation runs	<code>Simulink.sdi.compareRuns</code>

Run Management

Simulation Data Inspector software creates and manages a list of simulation runs. Each run is an instance of a `Simulink.sdi.Run` object. This object contains all of the simulation data and metadata for that simulation run.

To...	Use...
Get the number of runs currently in the Simulation Data Inspector	<code>Simulink.sdi.getRunCount</code>
Get the run ID from the list of simulation runs in the Simulation Data Inspector	<code>Simulink.sdi.getRunIDByIndex</code>
Determine if a run ID corresponds to a run currently in the Simulation Data Inspector	<code>Simulink.sdi.isValidRunID</code>
Add more data to a run currently in the Simulation Data Inspector	<code>Simulink.sdi.addToRun</code>

To...	Use...
Make a copy of a run currently in the Simulation Data Inspector	<code>Simulink.sdi.copyRun</code>
Delete a run from the Simulation Data Inspector	<code>Simulink.sdi.deleteRun</code>
Get simulation data for a run in the Simulation Data Inspector	<code>Simulink.sdi.getRun</code>
Specify a run naming rule using tokens and regular characters as a template	<code>Simulink.sdi.setRunNamingRule</code>
Mark a run in the Simulation Data Inspector for overwriting on the next recorded simulation	<code>Simulink.sdi.setRunOverwrite</code>
Manage output signal data and metadata of a simulation run	<code>Simulink.sdi.Run</code> class
Get the <code>Simulink.sdi.Signal</code> object corresponding to the given signal ID	<code>Simulink.sdi.Run.getSignal</code> method
Get the <code>Simulink.sdi.Signal</code> object corresponding to the index into the array signals in the run	<code>Simulink.sdi.Run.getSignalByIndex</code> method
Get the signal ID corresponding to the index into the array signals in the run	<code>Simulink.sdi.Run.getSignalIDByIndex</code> method
Determine if a signal ID corresponds to a signal currently in the run	<code>Simulink.sdi.Run.isValidSignalID</code> method

Signal Management

Each `Simulink.sdi.Run` object contains a `Simulink.sdi.Signal` object for each output signal data. This object contains all of the simulation data for the signal and its metadata.

To...	Use...
Get the simulation data and metadata for a signal from one simulation run.	<code>Simulink.sdi.getSignal</code>
Manages a signal's time series data and metadata for one simulation run.	<code>Simulink.sdi.Signal</code> class

Import/Export Data

To...	Use...
Save signal data currently in the Simulation Data Inspector	<code>Simulink.sdi.save</code>
Load previously saved Simulation Data Inspector session	<code>Simulink.sdi.load</code>
Clear all data from the Simulation Data Inspector	<code>Simulink.sdi.clear</code>
Close the Simulation Data Inspector and save data	<code>Simulink.sdi.close</code>

Comparison Results

To...	Use...
Manage the results of comparing two runs (<code>Simulink.sdi.compareRuns</code> creates the <code>Simulink.sdi.DiffRunResult</code> object)	<code>Simulink.sdi.DiffRunResult</code> class
Manage the results of comparing two signals (<code>Simulink.sdi.compareSignals</code> creates the <code>Simulink.sdi.DiffSignalResult</code> object)	<code>Simulink.sdi.DiffSignalResult</code> class

Create a Run in the Simulation Data Inspector

To populate the Simulation Data Inspector with runs of simulation data, you must first simulate your model and then call `Simulink.sdi.createRun`. When using the API, the Simulation Data Inspector does not automatically record simulation data. To create a run of simulation data, you can use the following code:

```
% Open the model 'slexAircraftExample'
load_system('slexAircraftExample');

% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');
% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run',...
            'namevalue',{'MyData'},{simOut});
```

Compare Signal Data

To compare the simulation data for two signals, you can use the following code:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run and get signal IDs
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', ...
            'namevalue', {'MyData'}, {simOut});

sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns the results in an instance
% of Simulink.sdi.diffSignalResult
diff = Simulink.sdi.compareSignals(sig1, sig2);

% Find if the signal data match
match = diff.match;
```



```
% Get the tolerance used in Simulink.sdi.compareSignals
tolerance = diff.tol;
```

Compare Runs of Simulation Data

To compare the signal data between two simulation runs, you can use the following code:

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from
% simOut in the base workspace
runID1 = Simulink.sdi.createRun('First Run','namevalue', ...
    {'simOut'},{simOut});

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue', ...
    {'simOut'},{simOut});

% Compare two runs, the result is stored in a
% Simulink.sdi.DiffRunResult object
difference = Simulink.sdi.compareRuns(runID1, runID2);

% Number of comparisons in result
numComparisons = difference.count;

% Iterate through each result element
for i = 1:numComparisons
    % Get signal result at index i
    signalResult = difference.getResultByIndex(i);
```

```
% Get signal IDs for each comparison result
sig1 = signalResult.signalID1;
sig2 = signalResult.signalID2;

% Display if signals match or not
displayStr = 'Signals with IDs %d and %d %s \n';
if signalResult.match
    fprintf(displayStr, sig1, sig2, 'match');
else
    fprintf(displayStr, sig1, sig2, 'do not match');
end
end
```

Specify Signal Tolerances

To control relative and absolute signal tolerances, you can use the following code:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run', ...
            'base',{'simOut'});

% Get the Simulink.sdi.Run object corresponding to the new run ID
runObj = Simulink.sdi.getRun(runID);

% Get the number of signals in the run
numSignals = runObj.signalCount;

% Get the Simulink.sdi.Signal objects for each signal in the run
% Specify the absolute and relative tolerance for each signal
for i = 1:numSignals
    signalObjs(i) = runObj.getSignal(signalIDs(i));
    signalObjs(i).absTol = 0.5;
    signalObjs(i).relTol = 0.005;
end
```

Record Data During Parallel Simulations

This example shows how to run multiple simulations in a `parfor` loop and record each run in the Simulation Data Inspector tool. Use the `Simulink.sdi.getSource`, `Simulink.sdi.setSource`, and `Simulink.sdi.refresh` methods to get and set the location of the Simulation Data Inspector repository. To record data during parallel simulations, you can use the following code:

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'slexAircraftExample';  
load_system(mdl);
```

Get the location of the simulation data repository.

```
src = Simulink.sdi.getSource();
```

Start a parallel pool with 4 workers.

```
parpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor i=1:4  
    % Set the location of the simulation data repository of this  
    % worker to be the same for aggregating the data  
    Simulink.sdi.setSource(src);  
    % Run the simulation  
    simOut = sim(mdl,'SaveOutput','on',...  
                'SaveFormat','StructureWithTime',...  
                'ReturnWorkspaceOutputs','on');  
    % Create a simulation run in the Simulation Data Inspector  
    Simulink.sdi.createRun(['Run' num2str(i)],'namevalue',...  
                          {'simout'},{simOut});  
end
```

Delete the current parallel pool and close all of the models.

```
delete(gcf);  
bdclose('all');
```

Refresh the Simulation Data Inspector.

```
Simulink.sdi.refresh();
```

See Also

[parfor](#) | [Simulink.sdi.getSource](#) | [Simulink.sdi.setSource](#) | [Simulink.sdi.refresh](#)

Analyzing Simulation Results

- “Viewing Output Trajectories” on page 18-2
- “Linearizing Models” on page 18-5
- “Finding Steady-State Points” on page 18-11

Viewing Output Trajectories

In this section...

“Viewing and Exporting Simulation Data” on page 18-2

“Using the Scope Block” on page 18-2

“Using Return Variables” on page 18-3

“Using the To Workspace Block” on page 18-3

“Using the Simulation Data Inspector Tool” on page 18-4

Viewing and Exporting Simulation Data

You can use several approaches to view output trajectories. Some approaches display output trajectories during simulation. Other approaches export signal values to the MATLAB workspace during simulation for later retrieval and analysis.

The following sections describe several approaches for viewing output trajectories. For additional information about exporting simulation data, see “Export Simulation Data” on page 48-4.

Using the Scope Block

You can display output trajectories on a Scope block during simulation as illustrated by the following model.



The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

The XY Graph block enables you to plot one signal against another.

Using Return Variables

By returning time and output histories, you can use the plotting commands provided in the MATLAB software to display and annotate the output trajectories.



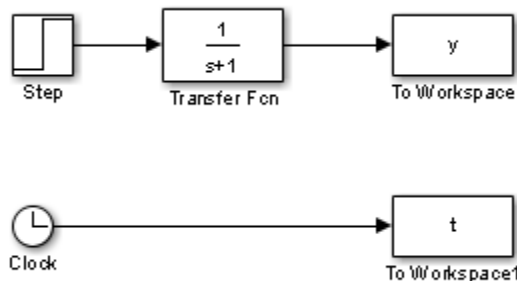
The block labeled Out is an Output block from the Ports & Subsystems library. The output trajectory, `yout`, is returned by the integration solver. For more information, see “Data Import/Export Pane”.

You can also run this simulation from the **Simulation** menu by specifying variables for the time, output, and states on the **Data Import/Export** pane of the Configuration Parameters dialog box. You can then plot these results using

```
plot(tout,yout)
```

Using the To Workspace Block

The To Workspace block can be used to return output trajectories to the workspace. The following model illustrates this use:



The variables `y` and `t` appear in the workspace when the simulation is complete. You store the time vector by feeding a Clock block into a To

Workspace block. You can also acquire the time vector by entering a variable name for the time on the **Data Import/Export** pane of the Configuration Parameters dialog box, for menu-driven simulations, or by returning it using the `sim` command (see “Data Import/Export Pane” for more information).

The To Workspace block can accept an array input, with each input element’s trajectory stored in the resulting workspace variable.

Using the Simulation Data Inspector Tool

By configuring your model to log signal data to the base workspace, you can view the output in the Simulation Data Inspector tool. Open the **Configuration Parameters > Data Import/Export** pane. Set the following parameters:

- **Time:** enable
- **States:** enable
- **Output:** enable
- **Signal logging:** enable
- **Format:** Structure with time

For more information on the Simulation Data Inspector tool, see “Inspect Signal Data with Simulation Data Inspector” on page 17-2.

Linearizing Models

In this section...

“About Linearizing Models” on page 18-5

“Linearization with Referenced Models” on page 18-7

“Linearization Using the 'v5' Algorithm” on page 18-9

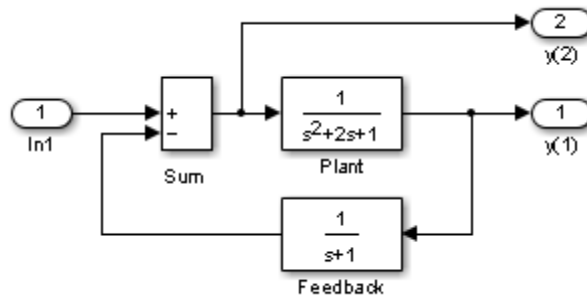
About Linearizing Models

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)
```

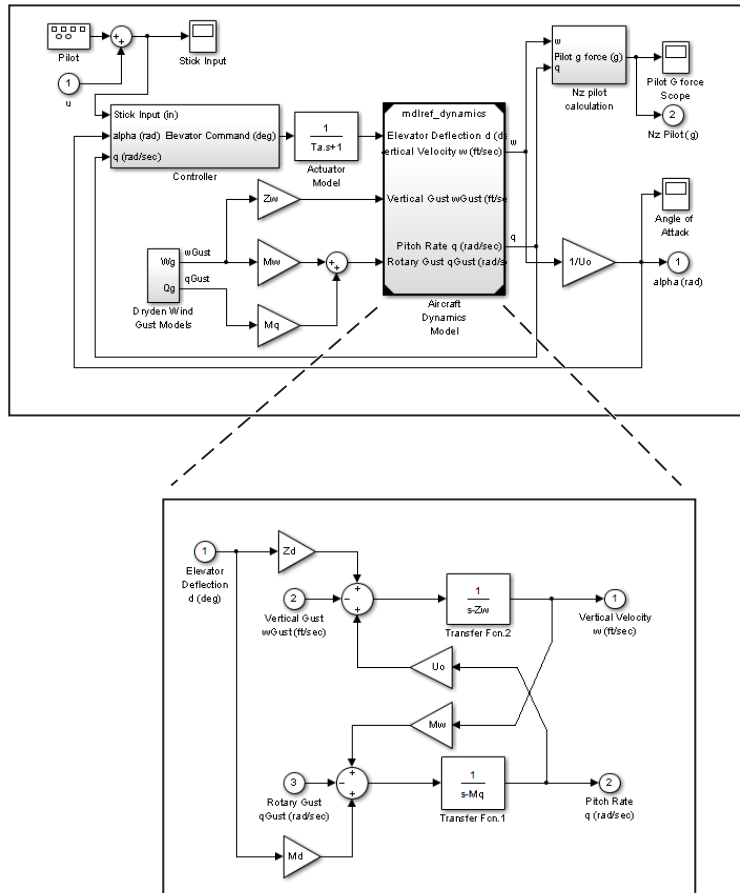
For discrete systems or mixed continuous and discrete systems, use the function `dlinmod` for linearization. This function has the same calling syntax as `linmod` except that the second right-hand argument must contain a sample time at which to perform the linearization.

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, consider the f14 model `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.

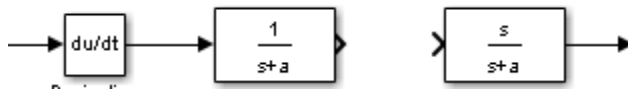
You access the Extras library by opening the Blocksets & Toolboxes icon:

- For the Derivative block, use the Switched derivative for linearization.
- For the Transport Delay block, use the Switched transport delay for linearization. (Using this block requires that you have the Control System Toolbox product.)

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

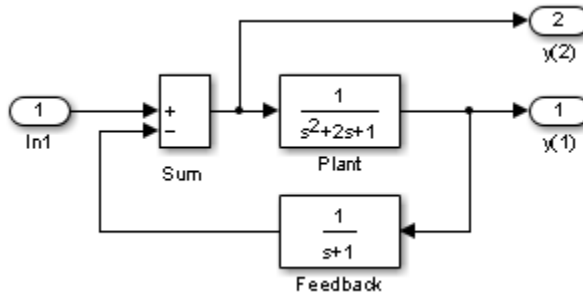
$$\frac{s}{s+a}$$

In this example, the blocks on the left of this figure can be replaced by the block on the right.



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_1mod`.



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`.

Improving Simulation Performance and Accuracy

- “About Improving Performance and Accuracy” on page 19-2
- “Speed Up Simulation” on page 19-3
- “Comparing Performance” on page 19-5
- “Improve Acceleration Mode Performance” on page 19-9
- “Improve Simulation Accuracy” on page 19-11

About Improving Performance and Accuracy

Simulation performance and accuracy can be affected by many things, including the model design and choice of configuration parameters.

The solvers handle most model simulations accurately and efficiently with their default parameter values. However, some models yield better results if you adjust solver parameters. Also, if you know information about your model's behavior, your simulation results can be improved if you provide this information to the solver.

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”.

Speed Up Simulation

Slow simulation speed can have many causes. Here are a few:

- Your model includes an Interpreted MATLAB Function block. When a model includes an Interpreted MATLAB Function block, the MATLAB interpreter is called at each time step, drastically slowing down the simulation. Use the built-in Fcn block or the Math Function block whenever possible.
- Your model includes a MATLAB file S-function. MATLAB file S-functions also cause the MATLAB interpreter to be called at each time step. Consider either converting the S-function 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 be 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).
- Did you ask for too much accuracy? 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 error in “Maximum order”.
- The time scale might be too long. Reduce the time interval.
- The problem might be stiff, but you are using a nonstiff solver. Try using ode15s.
- 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 Loops” on page 3-39.
- 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 a large number of data points. Try adjusting the viewer parameter settings that can affect performance. For more information, see “Parameter Settings and Performance with Scope Signal Viewer” on page 16-20.

Comparing Performance

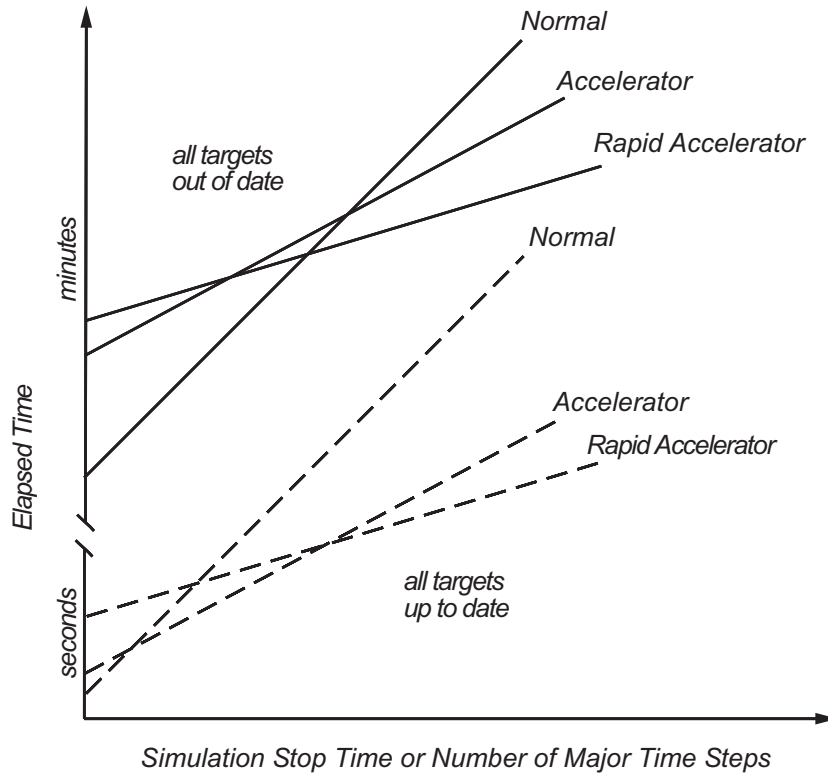
In this section...
“Performance of the Simulation Modes” on page 19-5
“Measure Performance” on page 19-7

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 22-16) 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.

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 From the **Simulation > Mode** menu, select **Normal**.
- 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 Select either **Accelerator** or **Rapid Accelerator** from the **Simulation > Mode** menu, and build an executable for the model by clicking the **Run** button. The acceleration modes use this executable in subsequent simulations as long as the model remains structurally unchanged. “Code Regeneration in Accelerated Models” on page 22-7 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.

Improve Acceleration Mode Performance

In this section...
“Techniques” on page 19-9
“C Compilers” on page 19-10

Techniques

To get the best performance when accelerating your models:

- Verify that the Configuration Parameters dialog box settings are as follows:

On this pane...	Set...	To...
Solver Diagnostics	Solver data inconsistency	none
Data Validity Diagnostics	Array bounds exceeded	none
Optimization	Signal storage reuse	selected

- Disable Stateflow debugging and animation.
- Inline user-written S-functions (these are TLC files that direct the Simulink Coder software to create C code for the S-function). See “Control S-Function Execution” on page 22-17 for a discussion on how the Accelerator mode and Rapid Accelerator mode work with inlined S-functions.

For information on how to inline S-functions, consult “Insert S-Function Code”.

- 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 22-23.

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:

http://www.mathworks.com/support/compilers/current_release/

Improve Simulation Accuracy

To check your simulation accuracy, run the simulation over a reasonable time span. Then, either reduce the relative tolerance to $1e-4$ (the default is $1e-3$) or reduce the absolute tolerance and run it again. Compare the results of both simulations. If the results are not significantly different, you can feel confident that the solution has converged.

If the simulation misses significant behavior at its start, reduce the initial step size to ensure that the simulation does not step over the significant behavior.

If the simulation results become unstable over time,

- Your system might be unstable.
- If you are using `ode15s`, you might need to restrict the maximum order to 2 (the maximum order for which the solver is A-stable) or try using the `ode23s` solver.

If the simulation results do not appear to be 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 or adjust it for individual states in the Integrator dialog box.
- If reducing the absolute tolerances does not sufficiently improve the accuracy, reduce the size of the relative tolerance parameter to reduce the acceptable error and force smaller step sizes and more steps.

Certain modeling constructs can also produce unexpected or inaccurate simulation results.

- A Source block that inherits its 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 5-33).
- A Derivative block found in an algebraic loop can result in a loss in solver accuracy.

Performance Advisor

- “How Performance Advisor Improves Simulation Performance” on page 20-2
- “Performance Advisor Workflow” on page 20-3
- “Get Started with Performance Advisor” on page 20-4
- “Performance Advisor Window” on page 20-5
- “Prepare a Model for Performance Advisor” on page 20-9
- “Run Performance Advisor Checks” on page 20-12
- “Use Performance Advisor Reports” on page 20-14
- “Operate on Performance Advisor Results” on page 20-17

How Performance Advisor Improves Simulation Performance

Whatever the level of complexity of the model, you might want to improve simulation performance. Performance Advisor checks a model for conditions and configuration settings that can result in slower simulation of the system that the model represents. It produces a report that lists the suboptimal conditions or settings it finds and suggests better configuration settings where appropriate. Performance Advisor can fix some of these suboptimal conditions automatically or you can fix them manually.

To learn about faster simulation using acceleration modes, see “Acceleration”.

To learn about faster simulation using modeling techniques, see “Model Execution Profiling”.

Related Examples

- “Performance Advisor Workflow” on page 20-3
- “Get Started with Performance Advisor” on page 20-4

Performance Advisor Workflow

When the performance of a model is slower than expected, use Performance Advisor to help identify and resolve bottlenecks.

Caution Performance Advisor does not automatically save your model after it makes changes. When you are satisfied with the changes to your model from Performance Advisor, save the model

- 1 Create a baseline to compare measurements against.
- 2 Select the checks you want to run.
- 3 Run Performance Advisor with the selected checks and see recommended changes.
- 4 Make changes to your model. Performance Advisor can automatically apply the changes it recommends or you can manually select the ones to apply.
- 5 After applying changes, validate your 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.
- 6 Repeat the workflow using other checks as needed.

Note Use Performance Advisor on top models. Performance Advisor does not traverse referenced models or library links.

Related Examples

- “Get Started with Performance Advisor” on page 20-4
- “Prepare a Model for Performance Advisor” on page 20-9

Concepts

- “How Performance Advisor Improves Simulation Performance” on page 20-2

Get Started with Performance Advisor

In this section...

“Prepare to Use Performance Advisor” on page 20-4

“Start Performance Advisor” on page 20-4


Prepare to Use Performance Advisor

Before you use Performance Advisor, complete the following steps:

- Make a backup of your model.
- Check that the model can simulate without error.
- 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 change the performance of model simulation and the ability of Performance Advisor to measure accurately.

Start Performance Advisor

Start Performance Advisor using one of these methods:

- In the Simulink Editor, select **Analysis > Performance Tools > Performance Advisor**.
- From the Simulink Editor toolbar, on the  list, select Performance Advisor.
- In the MATLAB Command Window, type `performanceadvisor('model_name')`. For example:

`performanceadvisor('vdp')`

Related Examples

- “Prepare a Model for Performance Advisor” on page 20-9
- “Performance Advisor Workflow” on page 20-3

Concepts

- “Performance Advisor Window” on page 20-5
- “How Performance Advisor Improves Simulation Performance” on page 20-2

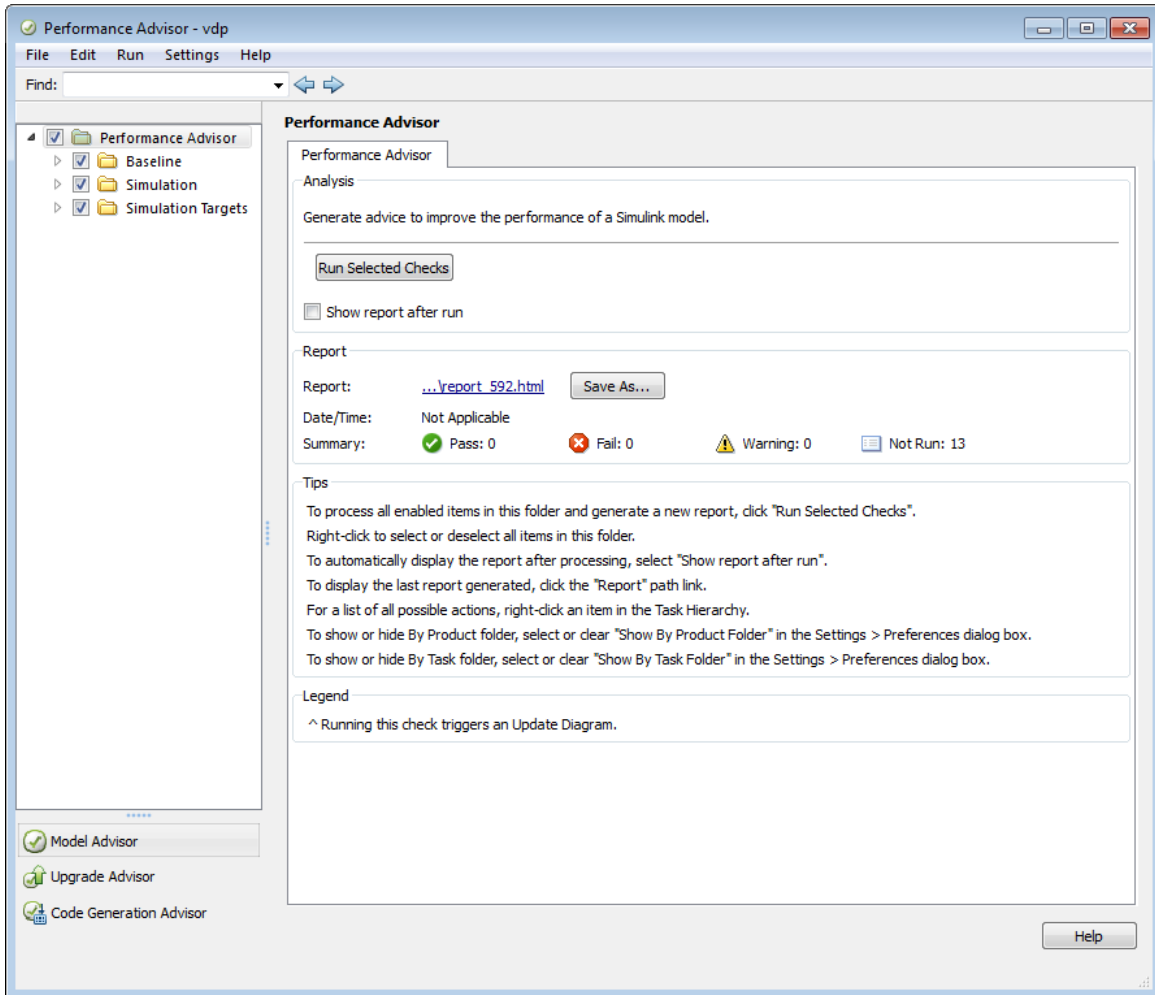
Performance Advisor Window

When you open Performance Advisor, it displays two panes.

In the left pane, the performance checks you can run are stored in folders. Expand the folders to see the checks within and select checks to run. You can search for folders and checks using **Find** above the pane.

Use the right pane to:

- Learn more about each check
- Specify input parameters
- Run checks
- View and save reports
- View Performance Advisor results



After you run a check, Performance Advisor displays the results in the right pane. The right pane changes depending on the check you have selected.

Selection	Right Pane Display
Folder	<p data-bbox="615 300 957 331">Analysis pane, containing:</p> <ul data-bbox="615 361 1328 539" style="list-style-type: none"> <li data-bbox="615 361 1328 427">• Run Selected Checks button — Click to run the selected checks in the folder and its subfolders. <li data-bbox="615 444 1328 539">• Show report after run check box — Select to display an HTML report of the check results in a separate window. <p data-bbox="615 574 936 605">Report pane, containing:</p> <ul data-bbox="615 635 1328 748" style="list-style-type: none"> <li data-bbox="615 635 1328 666">• Link to HTML report of check results <li data-bbox="615 683 1328 748">• Save As button — Click to save the HTML report in a specific location <p data-bbox="615 782 1292 848">Tips and Legend panes, containing brief descriptions on using the checks.</p>
Check	<p data-bbox="615 864 957 895">Analysis pane, containing:</p> <ul data-bbox="615 925 1298 1104" style="list-style-type: none"> <li data-bbox="615 925 1298 1052">• Input Parameters section — Before running checks, specify how you want checks to run (for more information, see “Run Performance Advisor Checks” on page 20-12). <li data-bbox="615 1069 1298 1104">• Result section — Display results after a check runs. <p data-bbox="615 1138 931 1170">Action pane, containing:</p> <ul data-bbox="615 1199 1273 1402" style="list-style-type: none"> <li data-bbox="615 1199 1273 1326">• Modify button — After the check runs, Performance Advisor suggests actions to take to improve performance. Click here to accept the recommendations and modify your model. <li data-bbox="615 1343 1273 1402">• Result section — Display results after performing recommended actions.

From Performance Advisor, you can also run:

- **Model Advisor** — Check a model or subsystem for conditions that result in inaccurate or inefficient simulation of the system. See “Consult the Model Advisor” on page 4-88.
- **Upgrade Advisor** — Upgrade and improve models with the current release. See “Consult the Upgrade Advisor” on page 4-113.
- **Code Generation Advisor** — Configure your model to meet code generation objectives. See “Application Objectives”.

Related Examples

- “Prepare a Model for Performance Advisor” on page 20-9
- “Performance Advisor Workflow” on page 20-3

Concepts

- “How Performance Advisor Improves Simulation Performance” on page 20-2

Prepare a Model for Performance Advisor

Before running checks using Performance Advisor, complete the steps in these sections:

In this section...
“Enable Data Logging for your Model” on page 20-9
“Create a Performance Advisor Baseline Measurement” on page 20-10

Enable Data Logging for your Model

Before creating the baseline, make sure the model configuration parameters are set to enable data logging using the **Structured with time** format.

- 1** In the model, select **Simulation > Model Configuration Parameters**.
- 2** In the Configuration Parameters dialog box, click **Data Import/Export** in the left pane.
- 3** Set **Format** to **Structure with time**.
- 4** 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.
- 5** 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.

- 6** Click **OK** in the Configuration Parameters dialog box.
- 7** Run the model once to make sure that the simulation is successful.

After you have enabled data logging, you can create a baseline measurement.

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, ensure that you “Enable Data Logging for your Model” on page 20-9.

- 1** In the model, select **Analysis > 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.

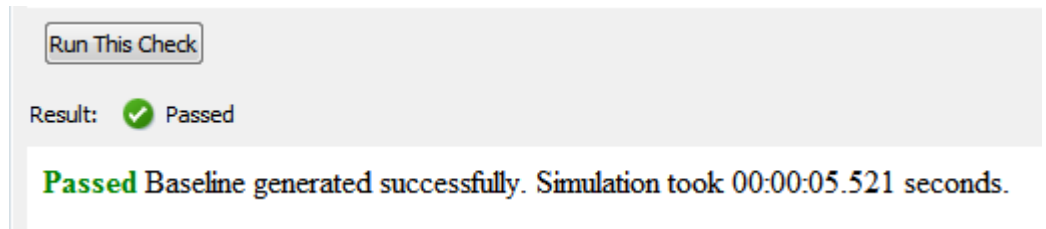
When you enter a Stop Time value in Performance Advisor, this overrides the value set in the model. A large stop time can create a simulation that runs longer.

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 **Click 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.

Related Examples

- “Run Performance Advisor Checks” on page 20-12

Concepts

- “How Performance Advisor Improves Simulation Performance” on page 20-2

Run Performance Advisor Checks

Before running checks using Performance Advisor, make sure that you have completed the steps in “Prepare a Model for Performance Advisor” on page 20-9.

- 1 After you have created a baseline measurement, select checks to run.

By default, when you open Performance Advisor for the first time, all checks are selected. You can select individual checks to run if you want.

- 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 For each check, in the right pane, specify input parameters.

Input Parameter	Description
Take action based on advice	<p>automatically — Allow Performance Advisor to automatically make the change for you.</p> <p>manually — Review the changes first. Then manually make the changes or accept Performance Advisor recommendations.</p>
Validate and revert changes if time of simulation increases	<p>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.</p>

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 Click **Run This Check**.

Performance Advisor displays the results in the right pane and 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.

Related Examples

- “Use Performance Advisor Reports” on page 20-14

Concepts

- “Operate on Performance Advisor Results” on page 20-17
- “How Performance Advisor Improves Simulation Performance” on page 20-2

Use Performance Advisor Reports

In this section...

“View Performance Advisor Reports” on page 20-14

“Save Performance Advisor Reports” on page 20-15

View Performance Advisor Reports

When Performance Advisor runs checks, it generates an HTML report of check results. To view a report, select a folder in the right pane and click the link in the **Report** box in the left 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.

Simulink version: 8.2
 System: vdp
 1 item with a timestamp different than 09-May-2013 17:51:48
 Treat as Referenced Model: off

Simulink Performance Advisor Report - vdp.slx

Model version: 1.6
 Current run: 09-May-2013 17:51:48

Performance Advisor

Baseline

✔ **Create baseline** (09-May-2013 17:50:34)

Passed Baseline generated successfully. Simulation took 00:00:05.521 seconds.

Input Parameters Selection

Name	Value
Stop Time	10
Check to view baseline signals and set their tolerances	false

Simulation

Checks Occurring Before Update

⚠ **Identify resource-intensive diagnostic settings**

Some diagnostics incur run-time overhead during simulation. Review the following parameters in the Model Configuration of model 'vdp' and the suggested changes for these parameters.

Performance Advisor has taken the necessary actions. For details, see the Action Results section.

Severity	Diagnostics checked	Original Value	New Value
✔	Diagnostics > Solver data inconsistency	none	none
⚠	Diagnostics > Data Validity > Signal resolution	Explicit and warn implicit	Explicit only
✔	Diagnostics > Data Validity > Division by singular matrix	none	none
✔	Diagnostics > Data Validity > Inf or nan block output	none	none
✔	Diagnostics > Data Validity > Simulation range checking	none	none
✔	Diagnostics > Data Validity > Array bounds exceeded	none	none
⚠	Diagnostics > Data Validity > Detect read before write	UseLocalSettings	DisableAll
⚠	Diagnostics > Data Validity > Detect write after read	UseLocalSettings	DisableAll

Use the check boxes under **Run Summary** 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 **Pass** and **Not Run** check boxes.

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.

Identify resource-intensive diagnostic settings

Some diagnostics incur run-time overhead during simulation. Review the following parameters in the Model Configuration window.

Performance Advisor has taken the necessary actions. For details, see the Action Results section.

	Severity	Diagnostics checked
Solver	✓	Diagnostics > Solver data inconsistency
Signals	⚠	Diagnostics > Data Validity > Signal resolution
	✓	Diagnostics > Data Validity > Division by singular matrix

▼ More (8 rows)

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

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.

Related Examples

- “Run Performance Advisor Checks” on page 20-12

Concepts

- “Operate on Performance Advisor Results” on page 20-17
- “How Performance Advisor Improves Simulation Performance” on page 20-2

Operate on Performance Advisor Results

In this section...

“View Results” on page 20-17

“Respond to Results” on page 20-18

“Review the Actions Taken” on page 20-19

View Results

After you run checks with Performance Advisor, the right pane shows the results:

Analysis →

Identify simulation target settings

Analysis

Disabling simulation target settings, such as 'Echo expression without semicolons', can improve simulation speed.

Input Parameters

Take action based on advice: automatically

Validate and revert changes if time of simulation increases

Validate and revert changes if degree of accuracy is greater than tolerance

Result: Warning

The simulation target setting 'Echo expressions without semicolons' is enabled. Disabling this setting might improve simulation speed. Review the following settings in the Model Configuration of model 'vdp'.

Performance Advisor has taken the necessary actions. For details, see the Action Results section.

Severity	Diagnostics checked	Original Value	New Value
Warning	Simulation Target > Echo expressions without semicolons	on	off

Action

Review the action results

Result:

Summary of performance validations			
	Before this check	After this check	Improvement
Performance			✓
Accuracy	Within given tolerance	Within given tolerance Click to view	✓
Simulation Time	00:00:34.970	00:00:02.036	94.18%

Action →

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 the Actions Taken

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.

The screenshot shows a web interface titled "Action" with a sub-header "Review the action results". Below this is a button labeled "Modify all and Validate". Underneath, the word "Result:" is followed by a table titled "Summary of performance validations". The table has four columns: "Before this check", "After this check", and "Improvement". The rows are: "Performance" (Before: Within given tolerance, After: Within given tolerance, Improvement: ✓), "Accuracy" (Before: Within given tolerance, After: Within given tolerance, Improvement: ✓), and "Simulation Time" (Before: 00:00:04.276, After: 00:00:02.412, Improvement: 43.59%). Below the table is a section titled "Summary of actions" with a sub-header "Actions:".

	Before this check	After this check	Improvement
Performance	Within given tolerance	Within given tolerance	✓
Accuracy	Within given tolerance	Within given tolerance	✓
Simulation Time	00:00:04.276	00:00:02.412	43.59%

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.

Related Examples

- “Run Performance Advisor Checks” on page 20-12
- “Use Performance Advisor Reports” on page 20-14

Concepts

- “How Performance Advisor Improves Simulation Performance” on page 20-2

Simulink Debugger

- “Introduction to the Debugger” on page 21-2
- “Debugger Graphical User Interface” on page 21-3
- “Debugger Command-Line Interface” on page 21-10
- “Debugger Online Help” on page 21-12
- “Start the Simulink Debugger” on page 21-13
- “Start a Simulation” on page 21-15
- “Run a Simulation Step by Step” on page 21-17
- “Set Breakpoints” on page 21-22
- “Display Information About the Simulation” on page 21-28
- “Display Information About the Model” on page 21-33

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, then the command-line interface.

Debugger Graphical User Interface

In this section...

“Displaying the Graphical Interface” on page 21-3

“Toolbar” on page 21-4

“Breakpoints Pane” on page 21-5

“Simulation Loop Pane” on page 21-6

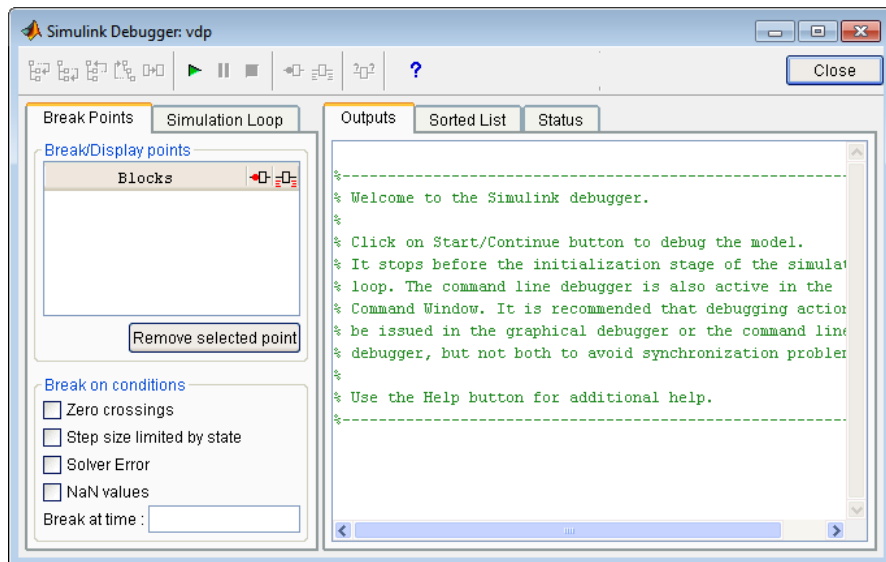
“Outputs Pane” on page 21-7

“Sorted List Pane” on page 21-8

“Status Pane” on page 21-9

Displaying the Graphical Interface

Select **Debug Model** from a model window **Simulation > Debug** menu 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 21-31 and “Display Solver Information” on page 21-32.





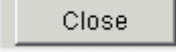
Toolbar

The debugger toolbar appears at the top of the debugger window.



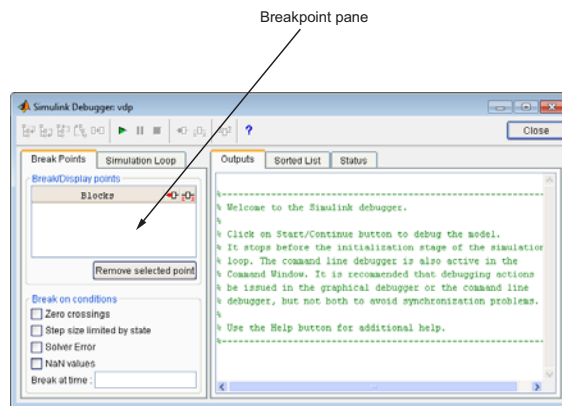
From left to right, the toolbar contains the following command buttons:

Button	Purpose
	Step into the next method (see “Stepping Commands” on page 21-19 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.

Button	Purpose
	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

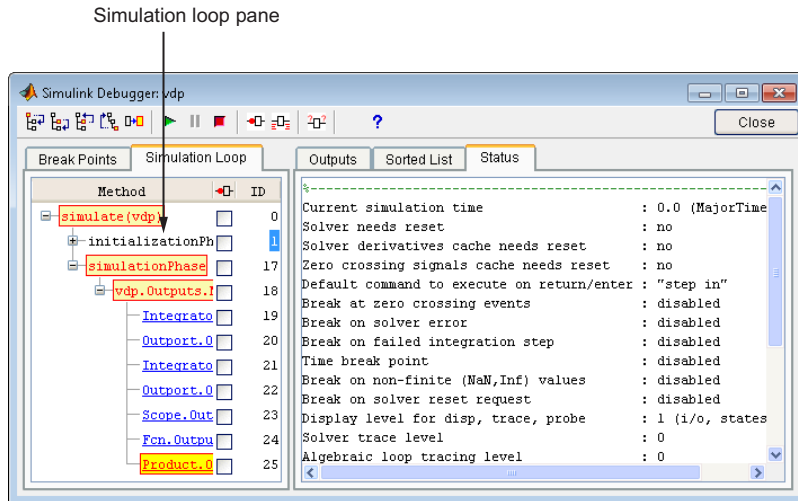
To display the **Breakpoints** pane, select the **Break Points** tab on the debugger window.



The **Breakpoints** pane allows you to specify block methods or conditions at which to stop a simulation. See “Set Breakpoints” on page 21-22 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

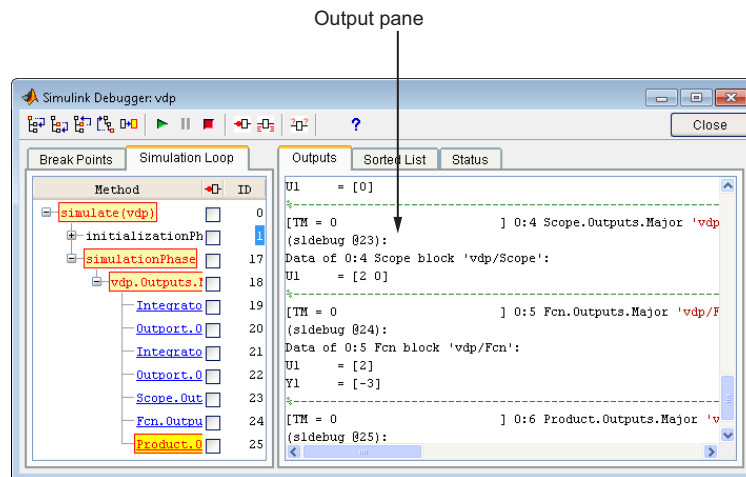
The breakpoints column consists of check boxes. Selecting a check box sets a breakpoint at the method whose name appears to the left of the check box. See “Setting Breakpoints from the Simulation Loop Pane” on page 21-24 for more information.

ID Column

The ID column lists the IDs of the methods listed in the **Methods** column. See “Method ID” on page 21-10 for more information.

Outputs Pane

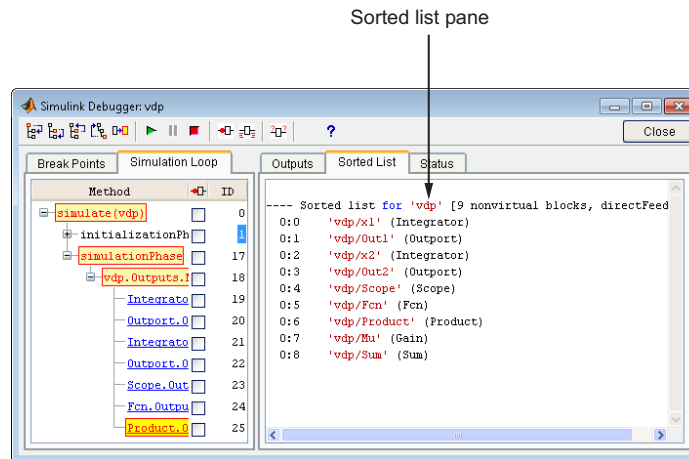
To display the **Outputs** pane, select the **Outputs** tab on the debugger window.



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 21-18). 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 21-10).

Sorted List Pane

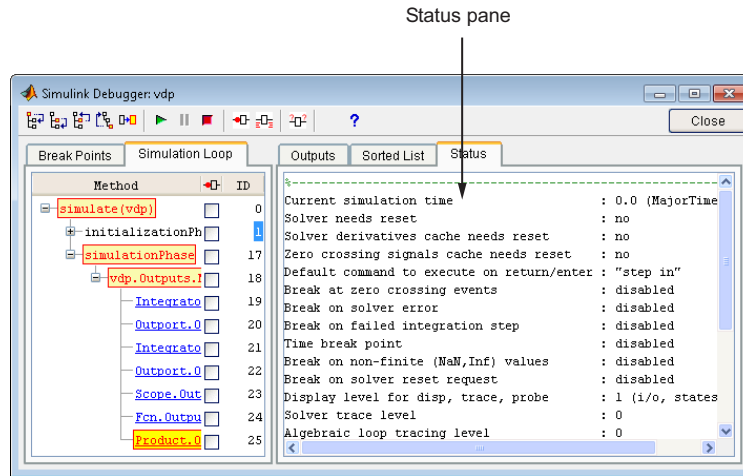
To display the **Sorted List** pane, select the **Sorted List** tab on the debugger window.



The **Sorted List** pane displays the sorted lists for the model being debugged. See “Display Model’s Sorted Lists” on page 21-33 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.

Debugger Command-Line Interface

In this section...

“Controlling the Debugger” on page 21-10

“Method ID” on page 21-10

“Block ID” on page 21-10

“Accessing the MATLAB Workspace” on page 21-11

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 “Debugging”.

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 indexes 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 `ID0:1` 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 `s(tep)` the simulation.

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.

Start the Simulink Debugger

You can start the debugger from either a Simulink model window or from the MATLAB Command Window.

In this section...
“Starting from a Model Window” on page 21-13
“Starting from the Command Window” on page 21-13

Starting from a Model Window

- 1 In a model window, select **Simulation > Debug > Debug Model**.

The debugger graphical user interface opens. See “Debugger Graphical User Interface” on page 21-3.

- 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 21-15.

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')
```

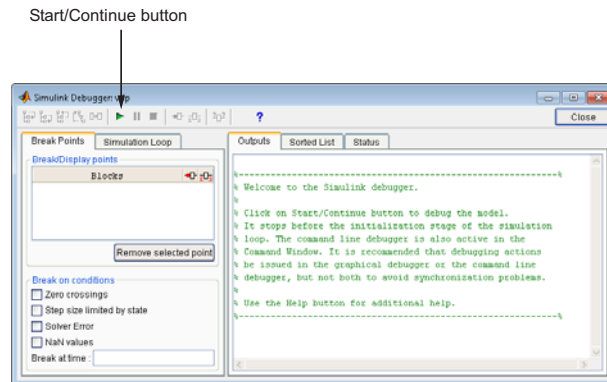
- or the `sldebug` command. For example, enter
`sldebug 'vdp'`

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.

Start a Simulation

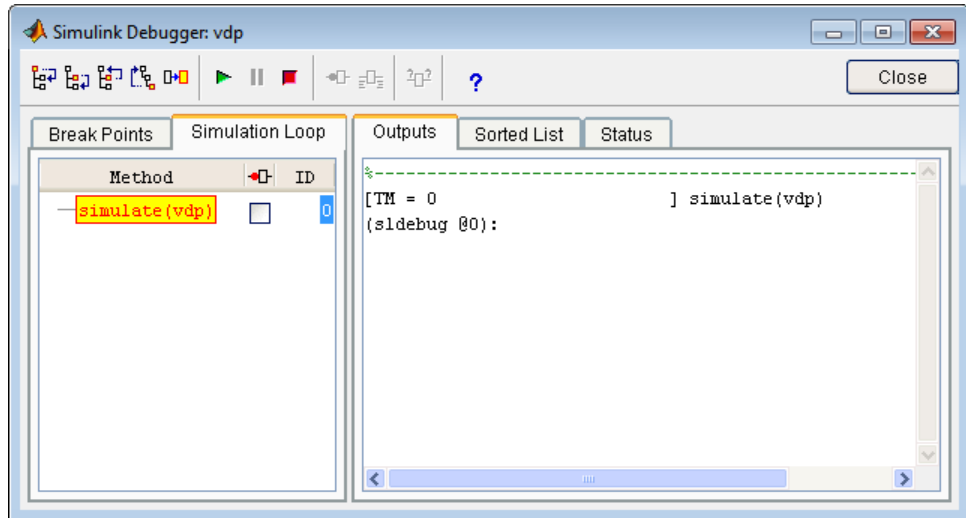
To start the simulation, click the **Start/Continue** button on the debugger toolbar.



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.

Run a Simulation Step by Step

In this section...

“Introduction” on page 21-17

“Block Data Output” on page 21-18

“Stepping Commands” on page 21-19

“Continuing a Simulation” on page 21-20

“Running a Simulation Nonstop” on page 21-20

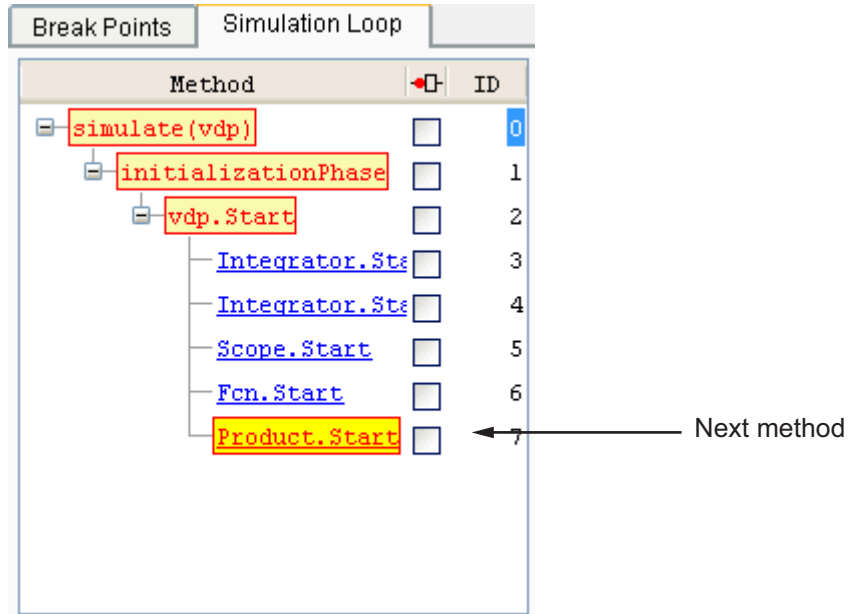
Introduction

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 21-19). 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.

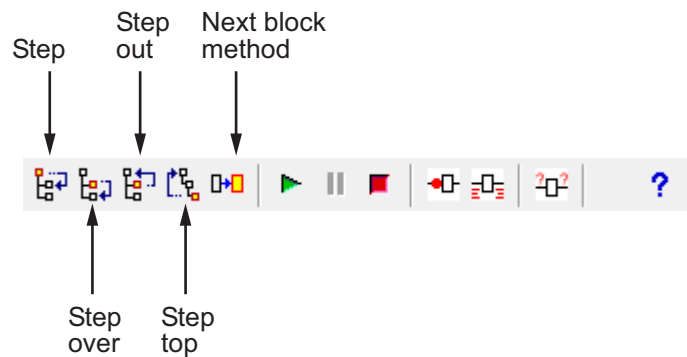
Block Data Output

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:

- $U_n = v$
where v is the current value of the block's n th input.
- $Y_n = v$
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.
- $DSTATE = v$

This command...	Advances the simulation...
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 21-22) 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.

Set Breakpoints

In this section...
“About Breakpoints” on page 21-22
“Setting Unconditional Breakpoints” on page 21-22
“Setting Conditional Breakpoints” on page 21-25

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.

Setting Unconditional Breakpoints

You can set unconditional breakpoints from the:

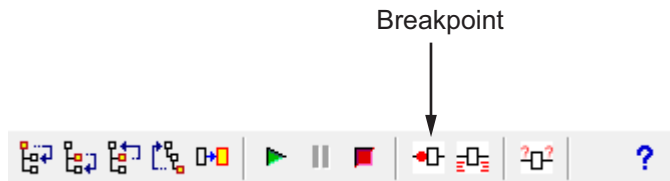
- Debugger toolbar
- **Simulation Loop** pane
- MATLAB product Command Window (command-line mode only)

Setting Breakpoints from the Debugger Toolbar

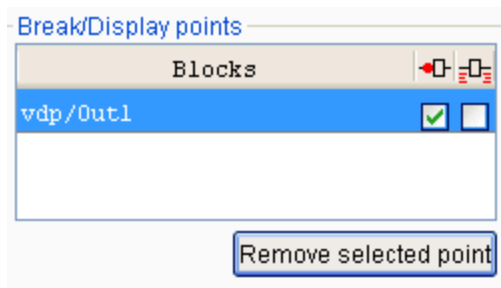
The **Breakpoint** button is disabled by default. In order to enable the button, perform the following steps:

- 1 Start the simulation.

- 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.

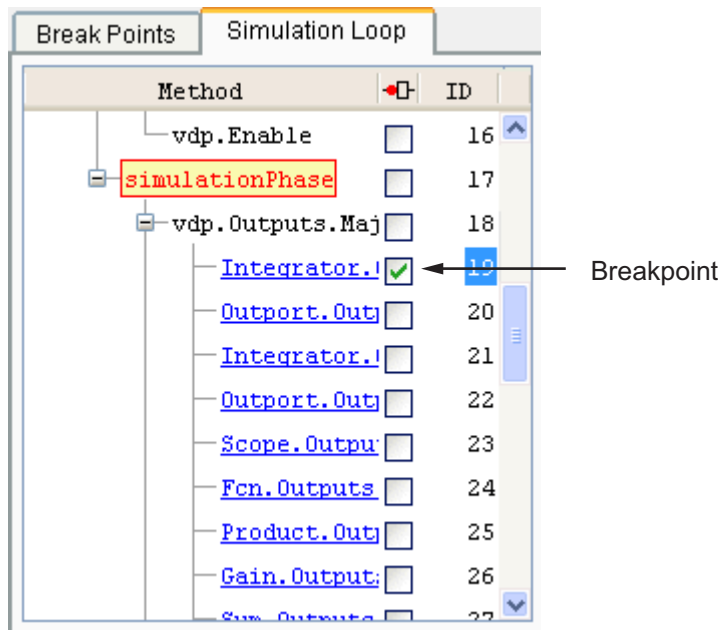
You can temporarily disable the breakpoints on a block by deselecting the check box in the breakpoints column of the panel. To clear the breakpoints on a block and remove its entry from the panel,

- 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 21-35).

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.



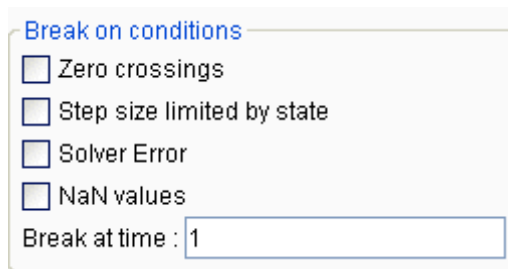
To clear the breakpoint, deselect the check box.

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...
<code>tbreak [t]</code>	At a simulation time step
<code>ebreak</code>	At a recoverable error in the model
<code>nanbreak</code>	At the occurrence of an underflow or overflow (NaN) or infinite (Inf) value
<code>xbreak</code>	When the simulation reaches the state that determines the simulation step size
<code>zcbreak</code>	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 21-10).

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.

Display Information About the Simulation

In this section...

“Display Block I/O” on page 21-28

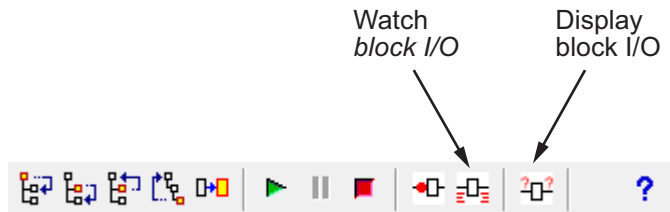
“Display Algebraic Loop Information” on page 21-30

“Display System States” on page 21-31

“Display Solver Information” on page 21-32



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 Loops" on page 3-39) 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...
<code>atrace 0</code>	No information
<code>atrace 1</code>	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
<code>atrace 2</code>	Same as level 1
<code>atrace 3</code>	Level 2 plus the Jacobian matrix used to solve the loop
<code>atrace 4</code>	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, clear the **Block reduction** check box on the Optimization pane, and clear the **Signal storage reuse** check box on the Optimization > Signals and Parameters pane.

```

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.

Display Information About the Model

In this section...

“Display Model’s Sorted Lists” on page 21-33

“Display a Block” on page 21-34

Display Model’s Sorted Lists

In GUI mode, the debugger **Sorted List** pane displays lists of blocks for a model’s 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' [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)
```

These displays include the block index for each command. You can use them to determine the block IDs of the model’s 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]
```

You can use the debugger `ashow` command to highlight the blocks and lines that make up an algebraic loop. See “Displaying Algebraic Loops” on page 21-36 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 21-33) 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
Window reuse                      : not supported
Execution Mode                   : Normal
Display level for etrace         : 0 (disabled)
Break points                     : none installed
Display points                   : none installed
Trace points                     : none installed
```


Accelerating Models

- “What Is Acceleration?” on page 22-2
- “How Acceleration Modes Work” on page 22-3
- “Code Regeneration in Accelerated Models” on page 22-7
- “Choosing a Simulation Mode” on page 22-11
- “Design Your Model for Effective Acceleration” on page 22-16
- “Perform Acceleration” on page 22-23
- “Interact with the Acceleration Modes Programmatically” on page 22-27
- “Run Accelerator Mode with the Simulink Debugger” on page 22-30
- “Capture Performance Data” on page 22-32

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 assist in debugging and determining relative performance of various parts of your model. For more information, see “Run Accelerator Mode with the Simulink Debugger” on page 22-30 and “Capture Performance Data” on page 22-32.

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 19-5.

To optimize your model and achieve faster simulation automatically using Performance Advisor, see “Automated Performance Optimization”.

To find out more about modeling techniques to achieve faster simulation, see “Model Execution Profiling”.

How Acceleration Modes Work

In this section...
“Overview” on page 22-3
“Normal Mode” on page 22-3
“Accelerator Mode” on page 22-4
“Rapid Accelerator Mode” on page 22-5

Overview

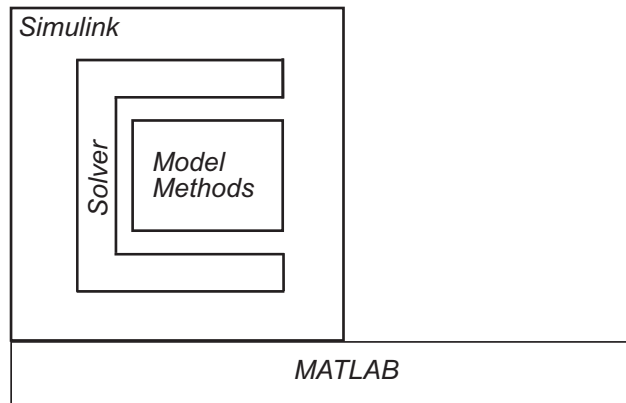
The Accelerator and Rapid Accelerator modes use portions of the Simulink Coder product to create an executable. These modes replace the interpreted code normally used in Simulink simulations, shortening model run time.

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. You must use the Simulink Coder product if you want 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.



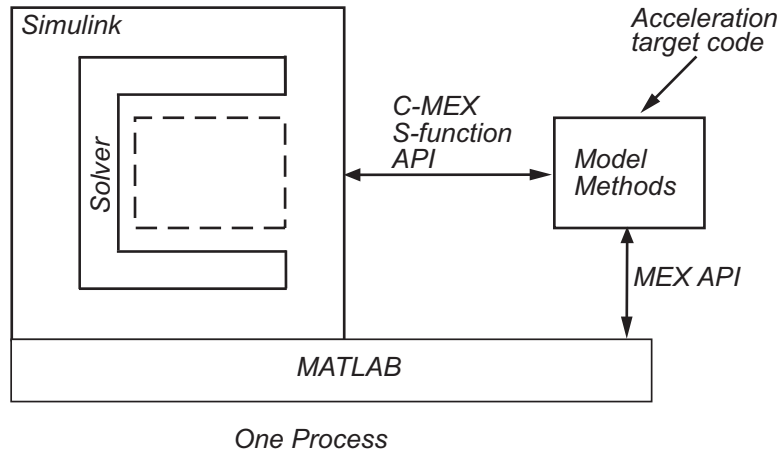
One Process

Accelerator Mode

The Accelerator mode generates and links code into a C-MEX S-function. Simulink uses this *acceleration target code* to perform the simulation, and the code remains available for use in later simulations.

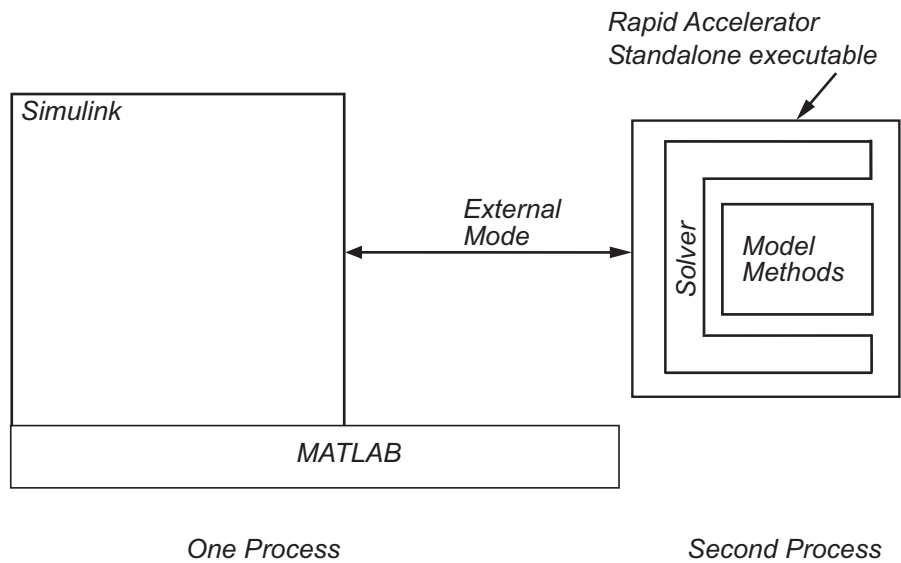
Simulink checks that the acceleration target code is up to date before reusing it. As explained in “Code Regeneration in Accelerated Models” on page 22-7, the target code regenerates if it is not up to date.

In Accelerator mode, the model methods are separate from the Simulink software and are part of the Acceleration target code. A C-MEX S-function API communicates with the Simulink software, and a MEX API communicates with MATLAB. 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 “Host/Target Communication”) to communicate with Simulink.



MATLAB and Simulink run in one process, and if a second processing core is available, the standalone executable runs there.

Code Regeneration in Accelerated Models

In this section...
“Structural Changes That Cause Rebuilds” on page 22-7
“Determining If the Simulation Will Rebuild” on page 22-8

Structural Changes That Cause Rebuilds

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.

Examples of model structure changes that result in a rebuild include:

- Changing the solver type, for example from `Variable-step` to `Fixed-step`
- Adding or deleting blocks or connections between blocks
- Changing the values of nontunable block parameters, for example, the **Seed** parameter of the Random Number block
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states in the model
- Selecting a different function in the Trigonometric Function block
- Changing signs used in a Sum block
- Adding a Target Language Compiler (TLC) file to inline an S-function
- Changing the `sim` command output argument when using Rapid Accelerator mode
- Changing solver parameters such as `stop time` or `rel tol` when using Rapid Accelerator mode
- Modifying the signal logging configuration, using the Signal Logging Selector (in Rapid Accelerator mode)

Determining 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.

Use the `Simulink.BlockDiagram.getChecksum` command to obtain the checksum for your model. For example:

```
cs1 = Simulink.BlockDiagram.getChecksum('myModel');
```

Obtain a second checksum after you have altered your model. The code regenerates if the new checksum does not match the previous checksum. You can 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 Handling in Rapid Accelerator Mode

In model rebuilds, Rapid Accelerator Mode handles block diagram and run-time parameters differently than other parameters. You can change some block diagram parameters during simulation without causing a rebuild. These block diagram parameters include the following:

Block Diagram Parameters That Do Not Require Rapid Accelerator Rebuild	
Solver Parameters	Loading and Logging Parameters
AbsTol	ConsistencyChecking
ConsecutiveZCsStepRelTol	Decimation
ExtrapolationOrder	FinalStateName
InitialStep	InitialState
MaxConsecutiveMinStep	LimitDataPoints
MaxConsecutiveZCs	LoadExternalInput
MaxNumMinSteps	LoadInitialState
MaxOrder	MaxConsecutiveZCsMsg

Block Diagram Parameters That Do Not Require Rapid Accelerator Rebuild	
MaxStep	MaxDataPoints
MinStep	MinStepSizeMsg
NumberNwtonIterations	OutputOption
OutputTimes	OutputSaveName
Refine	SaveFinalState
RelTol	SaveFormat
SolverName	SaveOutput
StartTime	SaveState
StopTime	SaveTime
ZCDetectionTol	StateSaveName
	TimeSaveName

For run-time parameters that you change graphically via the block diagram or programmatically with the `set_param` command, you will need to recompile the model. Recomilation might cause parts of the model to rebuild. Alternatively, you can programmatically change tunable run-time parameters without rebuilding parts of the model:

- 1** Collect the run-time parameters in a run-time 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.

For more information, see “sim in parfor with Rapid Accelerator Mode” on page 15-20.

All other parameter changes might necessitate a rebuild of the model.

Parameter Changes:	Passed Directly to sim command	Passed Graphically via Block Diagram or via set_param command
Run-time	Rebuild <i>not</i> required	Rebuild might be required
Block diagram (solver and logging parameters)	Rebuild <i>not</i> required	Rebuild <i>not</i> required
Other	Rebuild might be required	Rebuild might be required

Choosing a Simulation Mode

In this section...

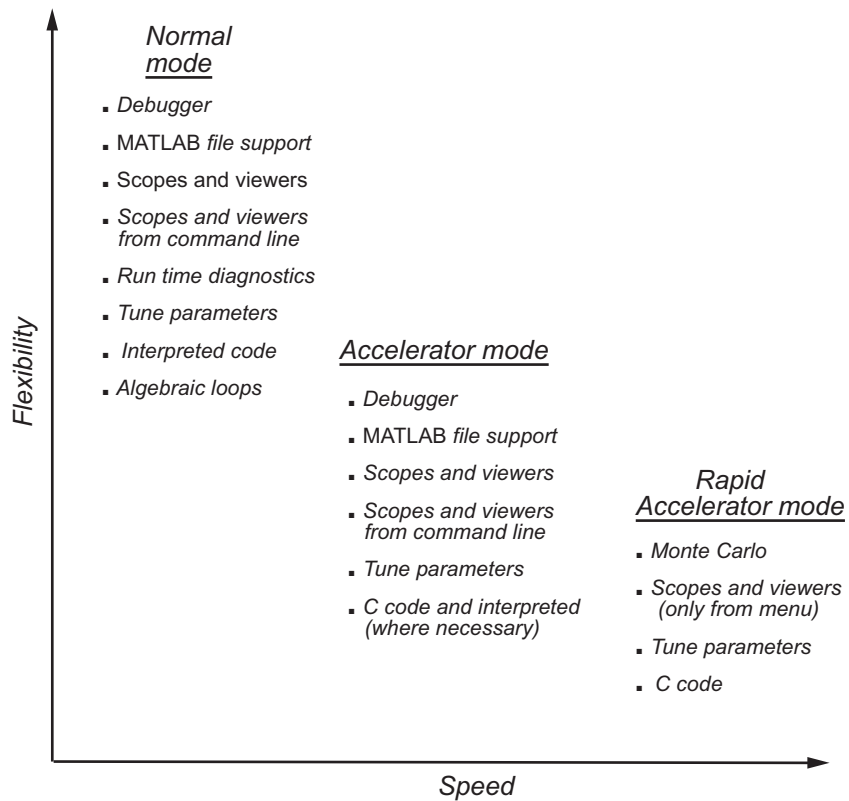
“Simulation Mode Tradeoffs” on page 22-11

“Comparing Modes” on page 22-12

“Decision Tree” on page 22-14

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. 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 is available for all of the blocks in the model. In addition, Rapid Accelerator mode does not support 3-D signals. If your model has 3-D signals, use Normal or Accelerator mode instead. Accelerator mode lies between these two in performance and in interaction with your 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 “Referenced Model Simulation Modes” on page 6-21.

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 22-7			
Model Requirement			
Accelerate your model even if C code is 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			

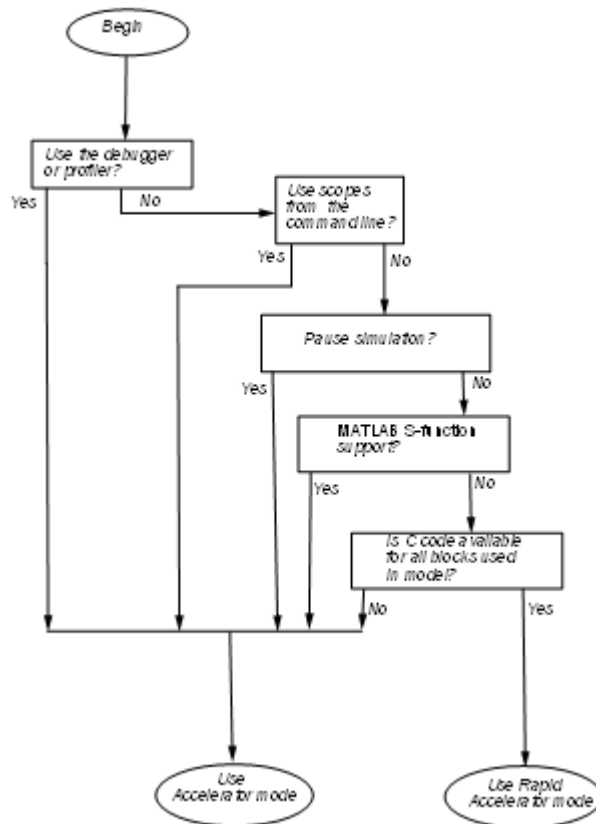
If you want to...	Then use this mode...		
	Normal	Accelerator	Rapid Accelerator
Use scopes and signal viewers	✓	✓	See “Behavior of Scopes and Viewers with Rapid Accelerator Mode” on page 22-19
Use scopes and signal viewers when running your model from the command line	✓	✓	

Note Scopes and viewers do not update if you run your model from the command line in Rapid Accelerator mode.

Decision Tree

The following decision tree can help you select between Normal mode, Accelerator mode, or Rapid Accelerator mode.

See “Comparing Performance” on page 19-5 to understand how effective the accelerator modes will be in improving the performance of your model.



Related Examples

- “Design Your Model for Effective Acceleration” on page 22-16
- “Interact with the Acceleration Modes Programmatically” on page 22-27

Concepts

- “Code Regeneration in Accelerated Models” on page 22-7

Design Your Model for Effective Acceleration

In this section...
“Select Blocks for Accelerator Mode” on page 22-16
“Select Blocks for Rapid Accelerator Mode” on page 22-17
“Control S-Function Execution” on page 22-17
“Accelerator and Rapid Accelerator Mode Data Type Considerations” on page 22-18
“Behavior of Scopes and Viewers with Rapid Accelerator Mode” on page 22-19
“Factors Inhibiting Acceleration” on page 22-20

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
- MATLAB Function
- Outport (root level only)
- Scope
- To File
- To Workspace
- Transport Delay

- Variable Transport Delay
- 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.

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 (such as vxWorks), 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 xPC Target 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

Inlining S-functions using the Target Language Compiler increases performance with the Accelerator mode by eliminating unnecessary calls to the Simulink application program interface (API). By default, however, the 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 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);
}
```

Accelerator and Rapid Accelerator Mode Data Type Considerations

- Accelerator mode supports fixed-point signals and vectors up to 128 bits.
- Rapid Accelerator mode does not support fixed-point signals or vectors greater than 32 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.

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	<ul style="list-style-type: none"> 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.	Supported, with limitations listed in Signal Logging in Rapid Accelerator Mode.
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 System 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 use the menu. “Run Acceleration Mode from the User Interface” on page 22-24 shows how to run Rapid Accelerator mode from the menu. “Interact with the Acceleration Modes Programmatically” on page 22-27 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.

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). For more information, see “Write Fully Inlined S-Functions”.
 - 3-D signals
 - Debugger or Profiler
- 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.
- For models that contain S-functions, ensure that the source files (.h, .c, and .cpp) for the S-function are in the same folder as the S-function

MEX-file. See *Implicit Build Support* for more information. 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*.

- 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 22-7 for more information.
- Rapid Accelerator does not support the variable-step solver on Microsoft Windows platforms when you use the Watcom C/C++ compiler.

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”. Additional keywords that apply only to the Accelerator and Rapid accelerator modes are:

<code>muDoubleScalarAbs</code>	<code>muDoubleScalarCos</code>	<code>muDoubleScalarMod</code>
<code>muDoubleScalarAcos</code>	<code>muDoubleScalarCosh</code>	<code>muDoubleScalarPower</code>
<code>muDoubleScalarAcosh</code>	<code>muDoubleScalarExp</code>	<code>muDoubleScalarRound</code>
<code>muDoubleScalarAsin</code>	<code>muDoubleScalarFloor</code>	<code>muDoubleScalarSign</code>
<code>muDoubleScalarAsinh</code>	<code>muDoubleScalarHypot</code>	<code>muDoubleScalarSin</code>
<code>muDoubleScalarAtan,</code>	<code>muDoubleScalarLog</code>	<code>muDoubleScalarSinh</code>
<code>muDoubleScalarAtan2</code>	<code>muDoubleScalarLog10</code>	<code>muDoubleScalarSqrt</code>

muDoubleScalarAtanh

muDoubleScalarMax

muDoubleScalarTan

muDoubleScalarCeil

muDoubleScalarMin

muDoubleScalarTanh

Perform Acceleration

In this section...

“Customize the Build Process” on page 22-23

“Run Acceleration Mode from the User Interface” on page 22-24

“Making Run-Time Changes” on page 22-26

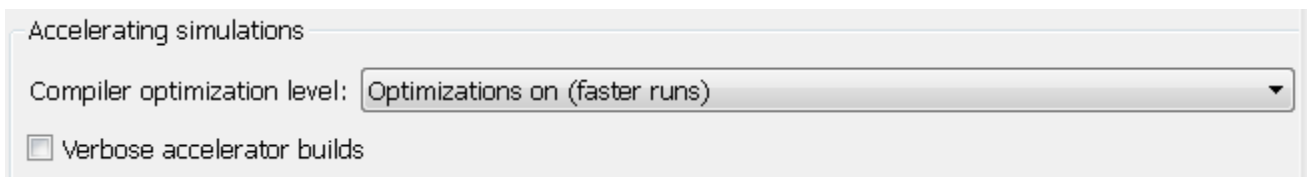
Customize the Build Process

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 left pane of the Configuration Parameters dialog box, select **Optimization**, and then 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 Name of the Generated File Location

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`), and places a compiled MEX-file in the current working folder. To change the name of the folder into which the Accelerator Mode writes generated code:

- 1 In the Simulink editor window, select **File > 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**.

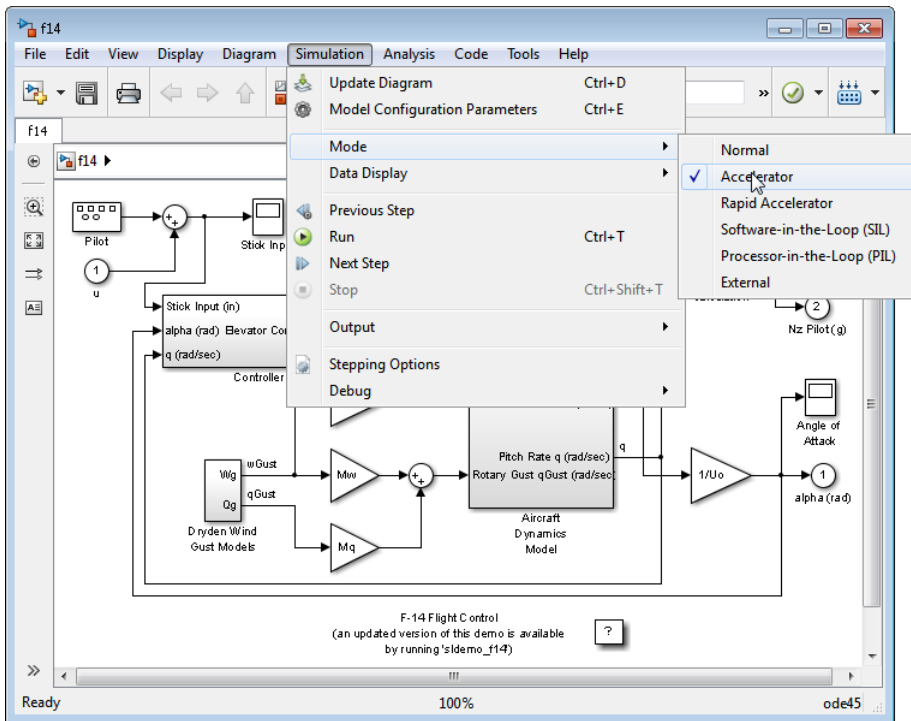
Run Acceleration Mode from the User Interface

To accelerate a model, first open it, and then from the **Simulation > Mode** menu, select either **Accelerator** or **Rapid Accelerator**. Then start the simulation.

The following example shows how to accelerate the already opened f14 model using the Accelerator mode:

- 1 From the **Simulation > Mode** menu, select **Accelerator**.

Alternatively, you can select Accelerator from the Simulink Editor toolbar.



2 From the **Simulation** menu, select **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 22-7.

The Accelerator mode places the generated code in a subfolder of the working folder called `s1prj/accel/modelname` (for example, `s1prj/accel/f14`), and places a compiled MEX-file in the current working folder. If you want to change this path, see “Changing the Name of the Generated File Location” on page 22-23.

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. For instance, changing the stop time in Rapid Accelerator mode causes code to regenerate, but does not cause Accelerator mode to regenerate code.

Interact with the Acceleration Modes Programmatically

In this section...

“Why Interact Programmatically?” on page 22-27

“Build Accelerator Mode MEX-files” on page 22-27

“Control Simulation” on page 22-27

“Simulate Your Model” on page 22-28

“Customize the Acceleration Build Process” on page 22-29

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 Accelerator Mode MEX-files

With the `accelbuild` command, you can build the Accelerator mode MEX-file 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 22-7 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 resimulate 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 22-23 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')
```

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')
```

Run Accelerator Mode with the Simulink Debugger

In this section...
“Advantages of Using Accelerator Mode with the Debugger” on page 22-30
“How to Run the Debugger” on page 22-30
“When to Switch Back to Normal Mode” on page 22-31

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 22-4.

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 From the **Simulation > Mode** menu, select **Accelerator**.

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

Capture Performance Data

In this section...

“What Is the Profiler?” on page 22-32

“How the Profiler Works” on page 22-32

“Enabling the Profiler” on page 22-34

“How to Save Simulink Profiler Results” on page 22-37

What Is the Profiler?

The profiler captures data while your model runs and identifies the parts of your model requiring the most time to simulate. You use this information to decide where to focus your model optimization efforts.

Note You cannot use the Rapid Accelerator mode with the Profiler.

Performance data showing the time spent executing each function in your model is placed in a report called the *simulation profile*.

How the Profiler Works

The following pseudocode summarizes the execution model on which the Profiler is based.

```
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, your model is executed by invoking the following functions zero, one, or more 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
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 a block's state at the current time step.	Block
Integrate	Compute a block's continuous states by integrating the state derivatives at the current time step.	Block
MinorOutput	Compute a block's output at a minor time step.	Block

Function	Purpose	Level
MinorDeriv	Compute a block's state derivatives at a minor time step.	Block
MinorZeroCrossings	Compute a block's zero-crossing values 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

The Profiler measures the time required to execute each invocation of these functions and generates a report at the end of the model that describes how much time was spent in each function.

Enabling the Profiler

To profile a model, open the model and select **Show Profiler Report** from the **Analysis > Performance Tools** menu. Then start the simulation. When the simulation finishes, the Simulink code generates and displays the simulation profile for the model in the Help browser.

Simulink Profiler Report

[Summary](#) | [Function Details](#) | [Simulink Profiler Help](#) | [Clear Highlighted Blocks](#)

Simulink Profile Report: Summary

Report generated 08-May-2012 19:37:16

Total recorded time: 5.85 s
 Number of Block Methods: 13
 Number of Internal Methods: 5
 Number of Model Methods: 4
 Clock precision: 0.00000004 s
 Clock Speed: 2400 MHz

To write this data as vdpProfileData in the base workspace [click here](#)

Function List

Name	Time	Calls	Time/call	Self time	Location (must use MATLAB Web Browser to view)
sim	5.85003750	100.0%	1 5.85003750000	0.00000000	vdp
Initialize	5.39763460	92.3%	1 5.39763460000	5.39763460	vdp
Terminate	0.28080180	4.8%	1 0.28080180000	0.28080180	vdp
...	0.17160110	2.9%	1 0.17160110000	0.00000000	vdp

Summary Section

The summary file displays the following performance totals.

Item	Description
Total Recorded Time	Total time required to simulate the model
Number of Block Methods	Total number of invocations of block-level functions (e.g., <code>Output()</code>)
Number of Internal Methods	Total number of invocations of system-level functions (e.g., <code>ModelExecute</code>)
Number of Nonvirtual Subsystem Methods	Total number of invocations of nonvirtual subsystem functions
Clock Precision	Precision of the profiler's time measurement

The summary section then shows summary profiles for each function invoked to simulate the model. For each function listed, the summary profile specifies the following information.

Item	Description
Name	Name of function. This item is a hyperlink. Clicking it displays a detailed profile of this function.
Time	Total time spent executing all invocations of this function as an absolute value and as a percentage of the total simulation time
Calls	Number of times this function was invoked
Time/Call	Average time required for each invocation of this function, including the time spent in functions invoked by this function

Item	Description
Self Time	Average time required to execute this function, excluding time spent in functions called by this function
Location	Specifies the block or model executed for which this function is invoked. This item is a hyperlink. Clicking it highlights the corresponding icon in the model diagram. The link works only if you are viewing the profile in the Help browser.

Detailed Profile Section

This section contains detailed profiles for each function invoked to simulate the model. Each detailed profile contains all the information shown in the summary profile for the function. In addition, the detailed profile displays the function (parent function) that invoked the profiled function and the functions (child functions) invoked by the profiled function. Clicking the name of the parent or a child function takes you to the detailed profile for that function.

Note Enabling the Profiler on a parent model does not enable profiling for referenced models. You must enable profiling separately for each referenced model. Profiling occurs only if the referenced model executes in Normal mode. See “Normal Mode” on page 6-21 for more information.

How to Save Simulink Profiler Results

You can save the Profiler report to a variable in the MATLAB workspace, and subsequently, to a `mat` file. At a later time, you can regenerate and review the report.

To save the Profiler report for a model `vdp` to the variable `profile1` and to the data file `report1.mat`, complete the following steps:

- 1 In the **Simulink Profiler Report** window, click **click here**. Simulink saves the report data to the variable `vdpProfileData`.
- 2 Navigate to the MATLAB command window.

3 To review the report, at the command line enter:

```
siprofreport(vdpProfileData)
```

4 To save the data to a variable named *profile1* in the base workspace, enter:

```
profile1 = vdpProfileData;
```

5 To save the data to a mat file named *report1*, enter:

```
save report1 profile1
```

To view the report at a later time, from the MATLAB command window, enter:

```
% Load the mat file and recreate the profile1 object  
load report1  
% Recreate the html report from the object  
siprofreport(profile1);
```

Managing Blocks

- Chapter 23, “Working with Blocks”
- Chapter 24, “Working with Block Parameters”
- Chapter 25, “Working with Lookup Tables”
- Chapter 26, “Working with Block Masks”
- Chapter 27, “Creating Custom Blocks”
- Chapter 28, “Working with Block Libraries”
- Chapter 29, “Using the MATLAB Function Block”
- Chapter 30, “System Objects”
- Chapter 31, “Define New System Objects”
- Chapter 32, “System Objects in Simulink”
- Chapter 33, “Design Considerations for C/C++ Code Generation”
- Chapter 34, “Functions Supported for Code Generation”
- Chapter 35, “System Objects Supported for Code Generation”
- Chapter 36, “Defining MATLAB Variables for C/C++ Code Generation”
- Chapter 37, “Defining Data for Code Generation”
- Chapter 38, “Code Generation for Variable-Size Data”
- Chapter 39, “Code Generation for MATLAB Structures”
- Chapter 40, “Code Generation for Enumerated Data”
- Chapter 41, “Code Generation for MATLAB Classes”
- Chapter 42, “Code Generation for Function Handles”
- Chapter 43, “Defining Functions for Code Generation”
- Chapter 44, “Calling Functions for Code Generation”

- Chapter 45, “Generate Efficient and Reusable Code”

Working with Blocks

- “About Blocks” on page 23-2
- “Add Blocks” on page 23-4
- “Edit Blocks” on page 23-9
- “Set Block Properties” on page 23-15
- “Change the Appearance of a Block” on page 23-23
- “Display Port Values” on page 23-30
- “Control and Display the Sorted Order” on page 23-36
- “Access Block Data During Simulation” on page 23-56
- “Configure a Block for Code Generation” on page 23-60

About Blocks

In this section...
“What Are Blocks?” on page 23-2
“Block Tool Tips” on page 23-2
“Virtual Blocks” on page 23-2

What Are Blocks?

Blocks are the elements from which the Simulink software builds models. You can model virtually any dynamic system by creating and interconnecting blocks in appropriate ways. This section discusses how to use blocks to build models of dynamic systems. Most blocks contain fields called *block parameters* that you can use to enter values that customize the behavior of the block. Be careful not to confuse block parameters with Simulink parameters, which are objects of type `simulink.parameter` that exist in the base workspace. See “About Block Parameters” on page 24-2 and “Set Block Parameters” on page 24-4 for information about setting and changing block parameters.

Block Tool Tips

Information about a block is displayed in a tool tip when you hover the mouse pointer over the block in the diagram view. To disable this feature, or control what information the tool tip displays, select **Display > Blocks > Tool Tip Options** in the Simulink Editor.

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 following 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 Output block.
From	Always virtual.
Goto	Always virtual.
Goto Tag Visibility	Always virtual.
Ground	Always virtual.
Inport	Virtual <i>unless</i> the block resides in a conditionally executed or atomic subsystem <i>and</i> has a direct connection to an Output block.
Mux	Always virtual.
Output	Virtual when the block resides within any subsystem block (conditional or not), and does <i>not</i> 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 <code>IsSubsystemVirtual</code> block property. See “Block-Specific Parameters”.
Terminator	Always virtual.
Trigger	Virtual when the output port is <i>not</i> present.

Add Blocks

In this section...
“Ways to Add Blocks” on page 23-4
“Add Blocks by Browsing or Searching with the Library Browser” on page 23-5
“Copy Blocks from a Model” on page 23-5
“Add Frequently Used Blocks” on page 23-6
“Add Blocks Programmatically” on page 23-8

Ways to Add Blocks

You can add blocks to a model in several ways.

Method	When to Use
Browse or search libraries with the Library Browser	<ul style="list-style-type: none"> You are not sure which block to add. You do not have a familiar model from which to copy blocks.
Copy blocks from a model	<ul style="list-style-type: none"> You know where in a model a block is that you want to copy. You want to replicate many of the parameter settings of an existing block.

Method	When to Use
Use the Most Frequently Used Blocks pane or context menu	<ul style="list-style-type: none"> • You want to add a block that you have used frequently and recently. • You are working on multiple models that share several of the same blocks. • You do not have a familiar model from which to copy similar blocks.
Add blocks programmatically with the <code>add_block</code> function	<ul style="list-style-type: none"> • You want to replicate most of the parameter settings of a block. • You are working on multiple models that share several of the same blocks.

Add Blocks by Browsing or Searching with the Library Browser

To browse or search for a block from a block library installed on your system, use the Library Browser. You can browse a list of block libraries or search for blocks whose names include the search string you specify.

When you find the block you want, select that block in the Library Browser and drag the block into your model. See “Populate a Model” on page 4-4 for more information.

Copy Blocks from a Model

To copy a block from a model in the Simulink Editor:

- 1** Select the block that you want to copy from a model.
- 2** Choose **Edit > Copy**.
- 3** In the model window in which you want to place the copied block, choose **Edit > Paste**.

You can copy a block:

- Within the same model window
- Between multiple systems open in one Simulink Editor instance
- Between systems open in multiple Simulink Editor instances

When you copy a block, the parameters use default values.

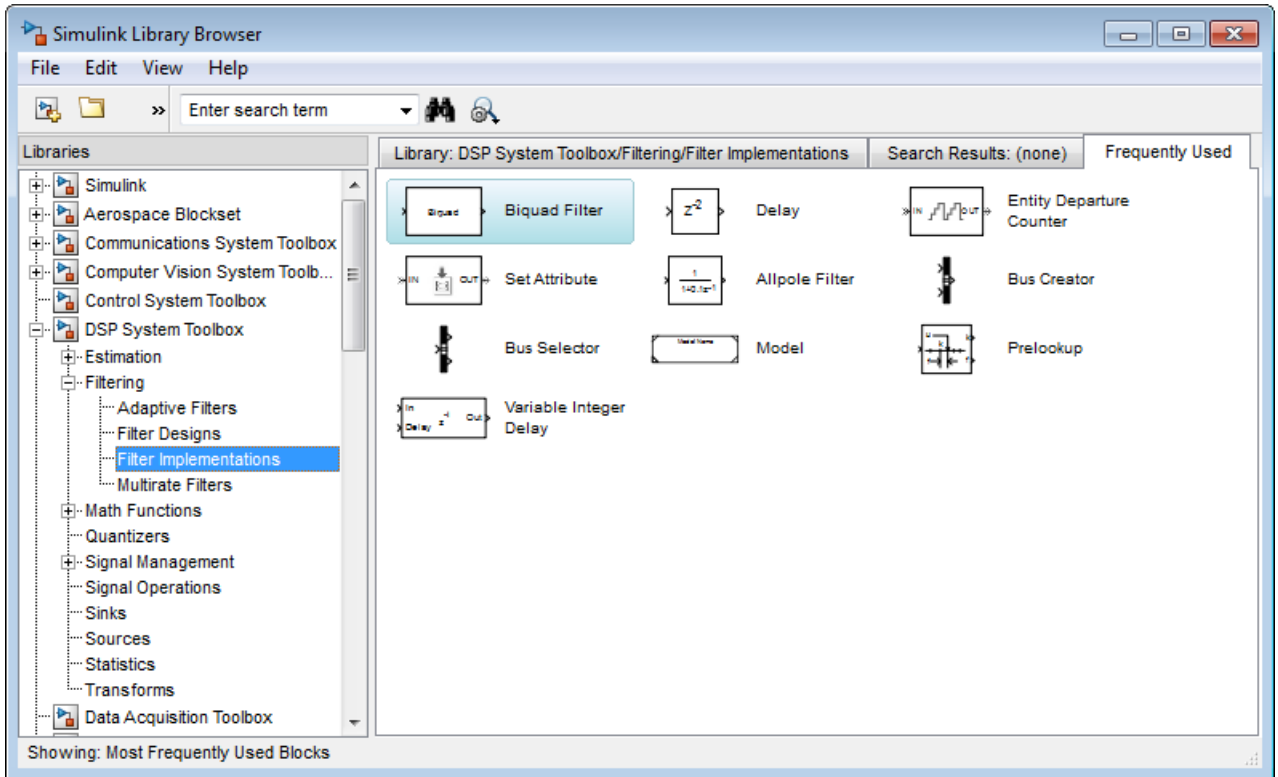
For details, see “Copy Blocks Between Windows” on page 23-10.

Add Frequently Used Blocks

When using the same block repeatedly, you can save time by using the **Frequently Used** feature in the Library Browser or Model Editor.

Selecting Your Frequently Used Blocks From the Library Browser

Open the **Frequently Used** pane with the third tab in the Library Browser.



To add a block to a model, select the block and do one of the following:

- Drag the block into your model.
- Right-click the block, and from the context menu select **Add to <model_name>**.

Select Frequently Used Blocks from the Model Editor

In the Simulink Editor, you can view a list of your most frequently used blocks. Right-click anywhere in the model except with the cursor directly on a block or signal. In the context menu, select **Most Frequently Used Blocks**.

What Blocks Appear on the Frequently Used Tab?

The Frequently Used tab lists blocks that you have added most often, with the most frequently used block at the top. The list reflects your ongoing modeling activity. For example, if you add several Gain blocks, the Gain block appears in the list if the list:

- Does not already include the Gain block
- Has less than 25 blocks
- Has 25 blocks, but you added more Gain blocks than the number of instances of the least frequently used block in the list

The list displays blocks that you rename as instances of the library block. For example, if you rename several Gain blocks to be MyGain1, MyGain2, and so on, those blocks count as Gain blocks.

When you close and reopen the Library Browser, the list reflects the blocks that you added up through the previous session. The list updates only when the Library Browser is open.

The list does *not* reflect blocks that you:

- Add programmatically
- Include in subsystems

If you add a subsystem that uses a specific type of block repeatedly, the list does not reflect that activity.

The list in the Model Editor is the same as the Library Browser list, except that the Model Editor list includes only five blocks.

Add Blocks Programmatically

To add a block programmatically, use the `add_block` function.

The `add_block` function copies the parameter values of the source block to the new block. You can use `add_block` to specify values for parameters of the new block.

Edit Blocks

In this section...

- “Copy Blocks in a Model” on page 23-9
- “Copy Blocks Between Windows” on page 23-10
- “Move Blocks” on page 23-11
- “Delete Blocks” on page 23-13
- “Comment Blocks” on page 23-14

Copy Blocks in a Model

To copy a blocks in a model:

- 1** In the Simulink Editor, select the block.
- 2** Press right mouse button.
- 3** Drag the block to a new location and release the mouse button.

You can also do this by using the **Ctrl** key:

- 1** In the Simulink Editor, press and hold the **Ctrl** key.
- 2** Select the block with the left mouse button.
- 3** Drag the block to a new location and release the mouse button.

Copies of blocks have the same parameter values as the original blocks. Sequence numbers are added to the new block names.

Note Simulink sorts block names alphabetically when generating names for copies pasted into a model. This action can cause the names of pasted blocks to be out of order. For example, suppose you copy a row of 16 gain blocks named Gain, Gain1, Gain2...Gain15 and paste them into the model. The names of the pasted blocks occur in the following order: Gain16, Gain17, Gain18...Gain31.

Copy Blocks Between Windows

As you build your model, you often copy blocks from Simulink block libraries or other libraries or models into your model window. To do this:

- 1** Open the appropriate block library or model window.
- 2** Drag the block to copy into the target model window. To drag a block, position the cursor over the block, then press and hold down the mouse button. Move the cursor into the target window, then release the mouse button.

You can also drag blocks from the Simulink Library Browser into a model window. See “Browse Block Libraries” on page 4-4 for more information.

Note The names of Sum, Mux, Demux, Bus Creator, and Bus Selector blocks are hidden when you copy them from the Simulink block library to a model. This is done to avoid unnecessarily cluttering the model diagram. (The shapes of these blocks clearly indicate their respective functions.)

You can also copy blocks by using the Simulink Editor:

- 1** Select the block you want to copy.
- 2** Select **Edit > Copy**.
- 3** Make the target model window the active window.
- 4** Choose **Edit > Paste**.

Simulink assigns a name to each copied block. If it is the first block of its type in the model, its name is the same as its name in the source window. For example, if you copy the Gain block from the Math library into your model window, the name of the new block is Gain. If your model already contains a block named Gain, Simulink adds a sequence number to the block name (for example, Gain1, Gain2). You can rename blocks; see “Manipulate Block Names” on page 23-27.

When you copy a block, the new block inherits all the original block's parameter values.

For more ways to add blocks, see “Add Blocks” on page 23-4.

add blocks

Move Blocks

To move a single block from one place to another in a model window, drag the block to a new location. Simulink automatically repositions lines connected to the moved block.

To move more than one block, including connecting lines:

- 1 Select the blocks and lines. For information about how to select more than one block, see “Select Multiple Objects” on page 4-6.
- 2 Drag the objects to their new location and release the mouse button.

To move a block, disconnecting lines:

- 1 Select the block.
- 2 Press the **Shift** key, then drag the block to its new location and release the mouse button.

You can also move a block by selecting the block and pressing the arrow keys.

You cannot move a block between:

- Multiple systems open in one Simulink Editor instance
- Systems open in multiple Simulink Editor instances

If you drag a block between Simulink Editor instances, the block is copied to the target Simulink Editor model window.

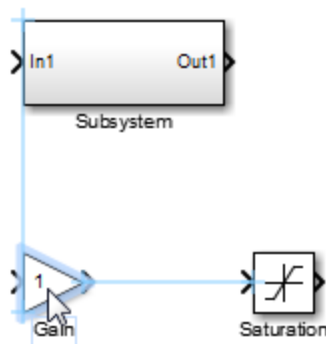
Align Blocks

You can manually align blocks. You can also use commands that align a group of blocks automatically (see “Align, Distribute, and Resize Groups of Blocks” on page 4-22 for details).

Smart Guides. You can enable smart guides to help you align blocks and lines. When you move a block, smart guides appear to indicate when the block ports, center, or edges are aligned with the ports, centers, and edges of other blocks in the same diagram.

To display smart guides, in the Simulink Editor, select **View > Smart Guides**. The setting applies to all open instances of the Simulink Editor.

For example, the following figure shows a snapshot of a Gain block that you drag from one position in a diagram to another. The blue smart guides indicate that if you drop the Gain block at this position, its left edge is aligned with the left edge of the Subsystem block and its output port is aligned with the input port of the Saturation block.



When you drag a block, one of its alignment features, for example, a port, may match more than one alignment feature of another block. In this case, Simulink displays a line for one of the features, using the following precedence order: ports, centers, edges. For example, in the following drag-and-drop snapshot, the Gain block center aligns with the Subsystem block center and the Gain block output port aligns with the Saturation block input port.

However, because ports take precedence over centers, Simulink draws a guide only for the ports.



Position Blocks Programmatically

You can position (and resize) a block programmatically, using its Position parameter. For example, the following command:

```
set_param(gcf, 'Position', [10 20 30 50]);
```

moves the currently selected block to a position where the top left corner of the block is at 10 20, and the bottom right corner is at 30 50.

Note The maximum size of a block diagram's height and width is 32767 points. An error message appears if you try to move or resize a block to a position that exceeds the diagram's boundaries.

Delete Blocks

To delete one or more blocks, select the blocks to be deleted and press the **Delete** or **Backspace** key.

As an alternative, you can select **Edit > Clear** or **Edit > Cut**.

The **Cut** command writes the blocks into the clipboard, which enables you to paste them into a model. Using the **Delete** or **Backspace** key or the **Clear** command does not enable you to paste the block later.

To replace a deleted block, use the **Edit > Undo** command.

Comment Blocks

Comment out blocks in your model if you want to exclude them during simulation. To exclude a block, right-click the selected block and select **Comment out**. You can comment through blocks in a model, if you want them to be a pass through. Right-click the block and select **Comment Through**. You can only comment through blocks that have the same number of inports and outports.

Commenting blocks may be useful for several tasks, such as:

- Incrementally testing parts of a model under development.
- Debugging a model without having to delete and restore blocks between simulation runs.
- Testing and verifying the effects of certain model blocks on simulation results.
- Improving simulation performance.

You can comment Model Variant and Variant Subsystem blocks. However, you cannot comment the following:

- Variant choices within a Variant Subsystem block.
- Port Blocks, Iterator blocks, Data Store Memory, and Goto Tag Visibility blocks.

Set Block Properties

For each block in a model, you can set general block properties, such as:

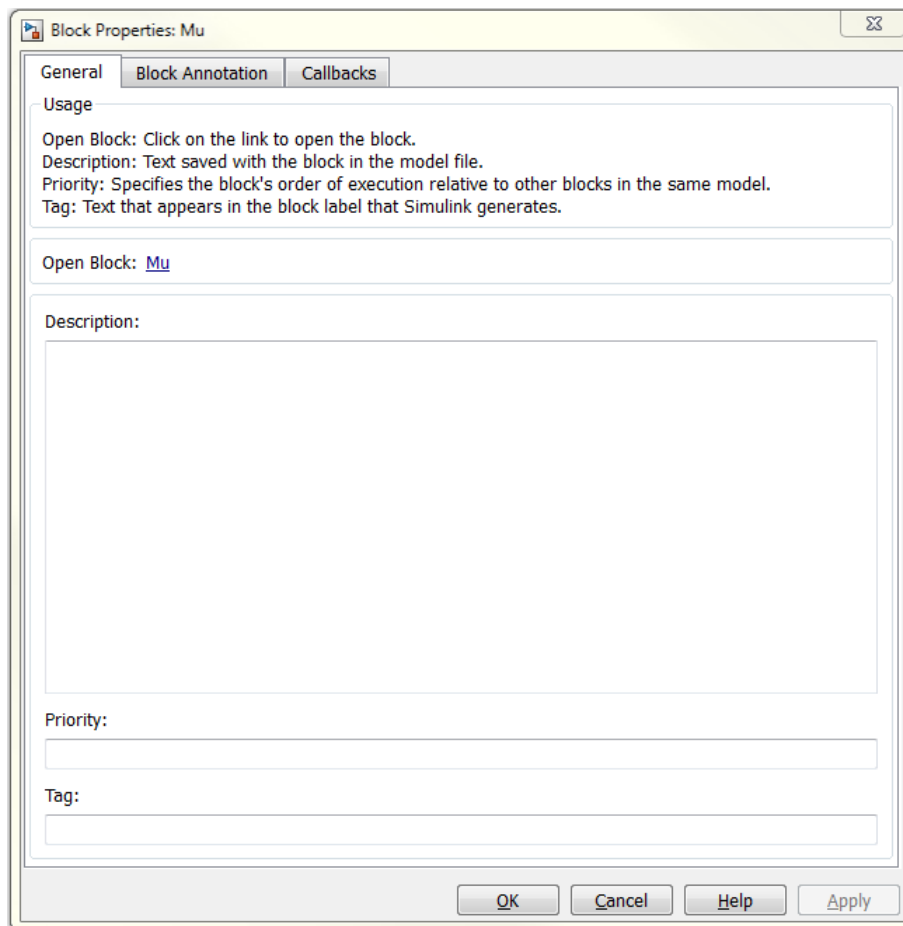
- A description of the block
- The block's order of execution
- A block annotation
- Block callback functions

Block Properties Dialog Box

To set these block properties, open the Block Properties dialog box.

- 1** In the Simulink Editor, select the block.
- 2** Select **Diagram > Properties**.

The Block Properties dialog box opens, with the **General** tab open. For example:



To open the Block Parameters dialog box for the block, in the General tab of the Block Properties dialog box, click the **Open Block** link. Use the Block Parameters dialog box to specify values for attributes that are specific each block.

Note Some blocks, such as Scope blocks, do not have a Block Parameters dialog box.

Using the Block Properties dialog box **Open Block** link to open the Block Parameters dialog box works for all blocks that have parameter dialog boxes, except for Subsystem and Model blocks. Use the Simulink Editor **Diagram** menu or the block context menu to open the Block Parameters dialog box for Subsystem and Model blocks.

General Block Properties

Description

Enter a brief description of the purpose of the block or any other descriptive information.

Priority

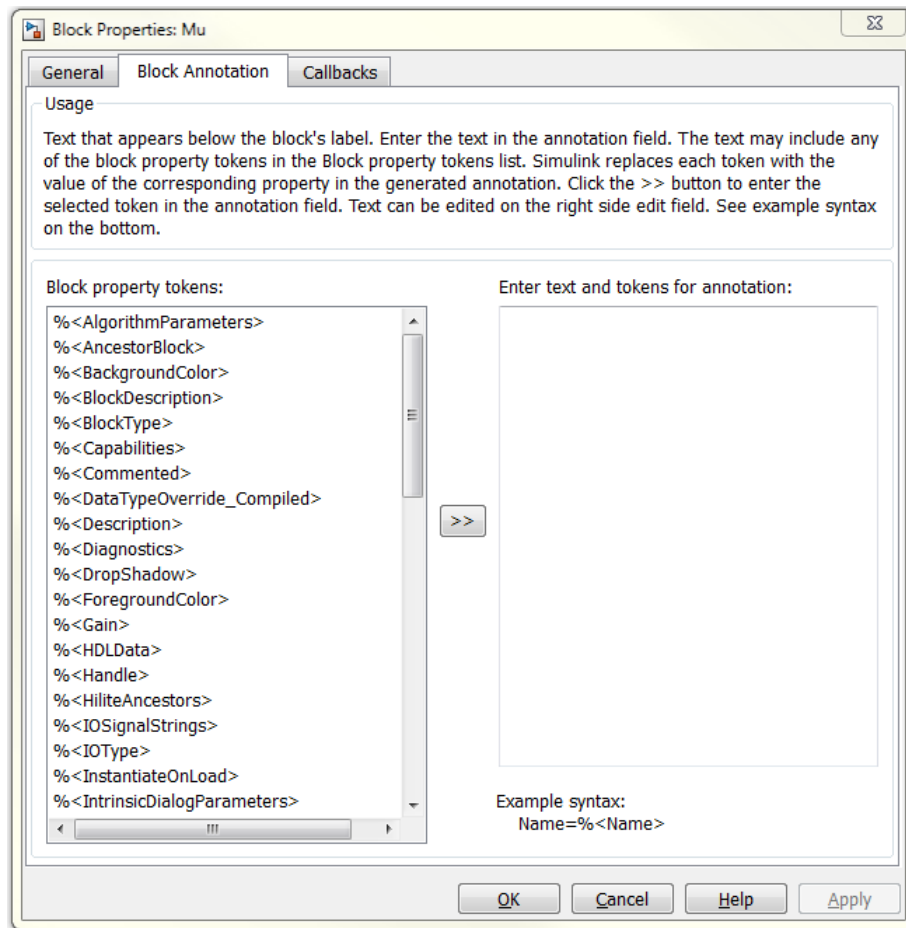
Specify the execution priority of this block relative to other blocks in the model. For more information, see “Assign Block Priorities” on page 23-50.

Tag

You can use a tag to create your own block-specific label for a block. Specify text that Simulink assigns to the block’s Tag parameter and saves with the block in the model.

Block Annotation Properties

Use the **Block Annotation** tab to display the values of selected block parameters in an annotation that appears beneath the block’s icon.



Enter the text of the annotation in the text field that appears on the right side of the pane. The text can include any of the block property tokens that appear in the list on the left side of the pane. A block property token is simply the name of a block parameter preceded by %< and followed by >. When displaying the annotation, Simulink replaces the tokens with the values of the corresponding block parameters. For example, suppose that you enter the following text and tokens for a Product block:

```
Multiplication = %<Multiplication>
Sample time = %<SampleTime>
```


In the Simulink Editor model window, the annotation appears as follows:

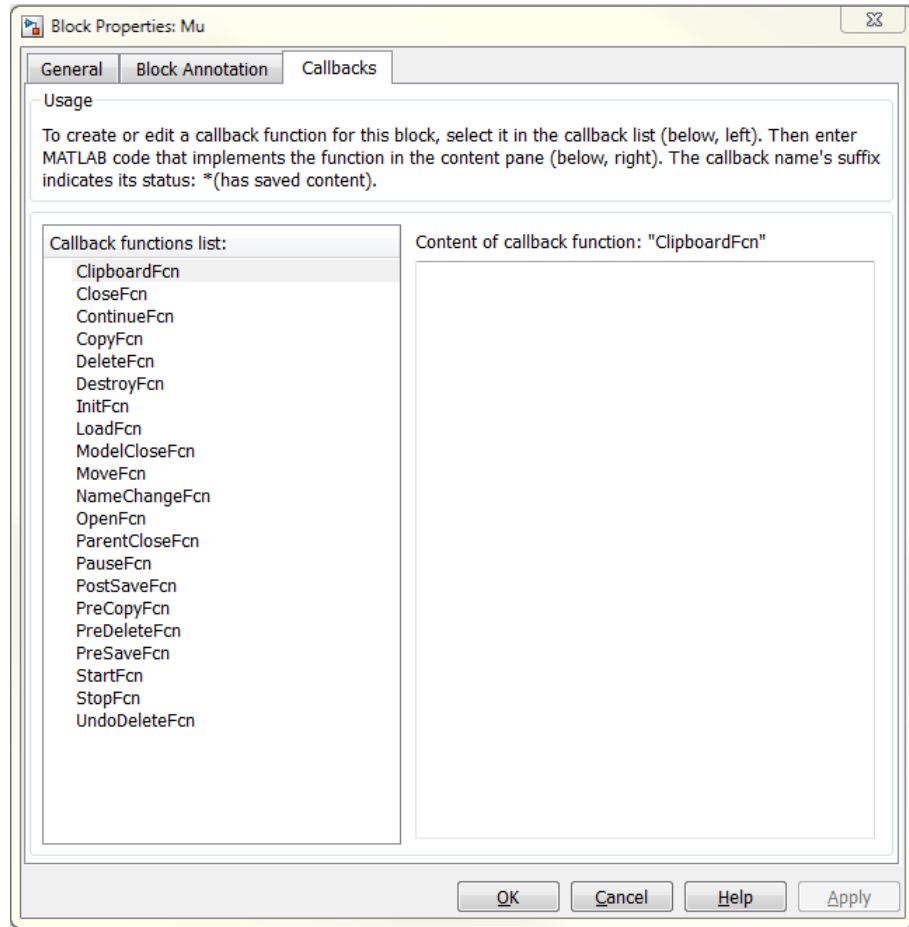


The block property token list on the left side of the pane lists all the parameters that are valid for the currently selected block (see “Common Block Properties” and “Block-Specific Parameters”). To add one of the listed tokens to the text field on the right side of the pane, select the token and then click the button between the list and the text field.

You can also create block annotations programmatically. See “Create Block Annotations Programmatically” on page 23-21.

Block Callbacks

Use the **Callbacks** tab to specify implementations for a block’s callbacks (see “Callbacks for Customized Model Behavior” on page 4-56).



To specify an implementation for a callback, select the callback in the callback list on the left side of the pane. Then enter MATLAB commands that implement the callback in the right-hand field. Click **OK** or **Apply** to save the change. Simulink appends an asterisk to the name of the saved callback to indicate that it has been implemented.

Modify Behavior for Opening a Block

You can use the `OpenFcn` callback to automatically execute MATLAB scripts when you double-click a block. MATLAB scripts can perform many

different tasks, such as defining variables for a block, making a call to MATLAB that brings up a plot of simulated data, or generating a GUI.

The `OpenFcn` overrides the normal behavior which occurs when opening a block (its parameter dialog box is displayed or a subsystem is opened).

To create block callbacks interactively, open the block's Block Properties dialog box and use the **Callbacks** tab to edit callbacks (see “Callbacks for Customized Model Behavior” on page 4-56).

To create the `OpenFcn` callback programmatically, click the block to which you want to add this property, then enter the following at the MATLAB command prompt:

```
set_param(gcf, 'OpenFcn', 'expression')
```

where `expression` is a valid MATLAB command or a MATLAB script that exists in your MATLAB search path.

The following example shows how to set up the callback to execute a MATLAB script called `myfunction.m` when double clicking a subsystem called `mysubsystem`.

```
set_param('mymodelName/mysubsystem', 'OpenFcn', 'myfunction')
```

Create Block Annotations Programmatically

You can use a block's `AttributesFormatString` parameter to display selected block parameters beneath the block as an “attributes format string,” which is a string that specifies values of the block's attributes (parameters). “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 desired attributes format string.

The attributes format string can be any text string 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 string beneath the block's icon, replacing each parameter name with the corresponding parameter value. You can use line-feed characters (`\n`) to display each parameter on a separate line. For example, enter the following at the MATLAB command prompt:

```
set_param(gcb, 'AttributesFormatString', 'pri=%<priority>\ngain=%<Gain>')
```

The Gain block displays the following block annotation:



If a parameter's value is not a string or an integer, Simulink displays N/S (not supported) for the parameter's value. If the parameter name is invalid, Simulink displays ??? as the parameter value.

Change the Appearance of a Block

In this section...

“Change a Block Orientation” on page 23-23

“Resize a Block” on page 23-25

“Displaying Parameters Beneath a Block” on page 23-26

“Drop Shadows” on page 23-26

“Manipulate Block Names” on page 23-27

“Specify Block Color” on page 23-29

Change a Block Orientation

By default, a block is oriented so that its input ports are on the left, and its output ports are on the right. You can change the orientation of a block by rotating it 90 degrees around its center or by flipping it 180 degrees around its horizontal or vertical axis.

- “How to Rotate a Block” on page 23-23
- “How to Flip a Block” on page 23-25

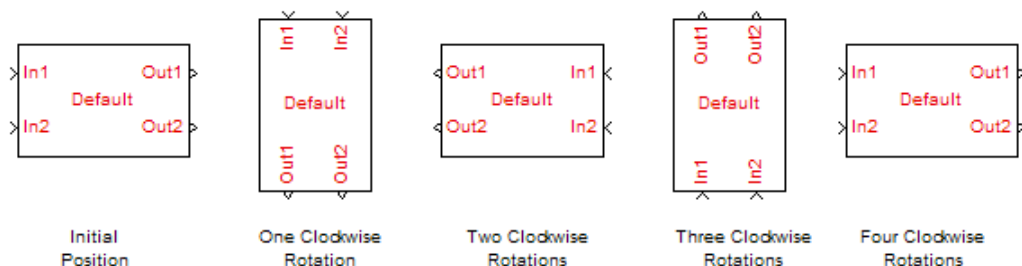
How to Rotate a Block

You can rotate a block 90 degrees by selecting one of these commands from the **Diagram** menu:

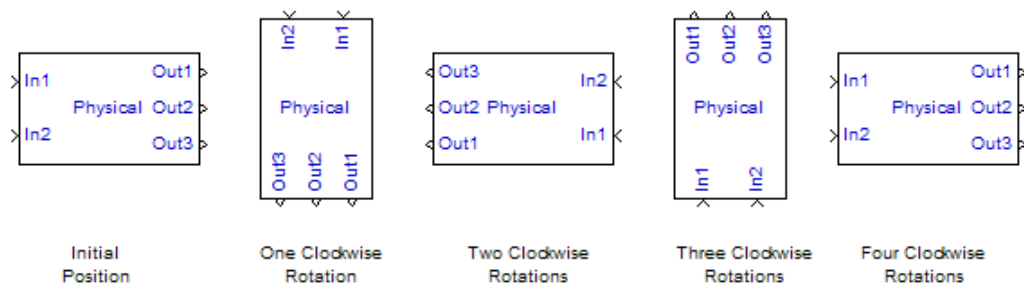
- **Rotate & Flip > Clockwise** (or **Ctrl+R**)
- **Rotate & Flip > Counterclockwise**

A rotation command effectively moves a block’s ports from its sides to its top and bottom or from its top and bottom to its size, depending on the initial orientation of the block. The final positions of the block ports depend on the block’s *port rotation type*.

Port Rotation Type. After rotating a block clockwise, Simulink may, depending on the block, reposition the block's ports to maintain a left-to-right port numbering order for ports along the top and bottom of the block and a top-to-bottom port numbering order for ports along the left and right sides of the block. A block whose ports are reordered after a clockwise rotation is said to have a *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. All nonmasked blocks and all masked blocks by default have the default rotation policy. The following figure shows the effect of using the **Rotate & Flip > Clockwise** command on a block with the default rotation policy.



A masked block can optionally specify that its ports not be reordered after a clockwise rotation (see "Port rotation"). Such a block is said to have a *physical port rotation type*. This policy facilitates layout of diagrams in mechanical and hydraulic systems modeling and other applications where diagrams do not have a preferred orientation. The following figure shows the effect of clockwise rotation on a block with a physical port rotation type

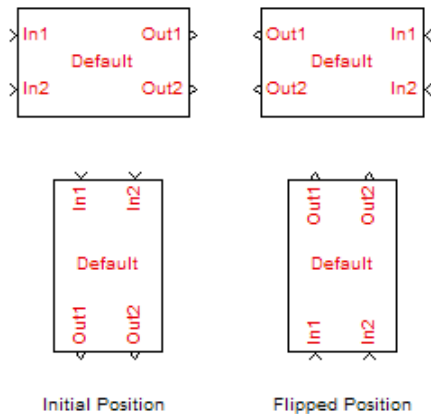


How to Flip a Block

Simulink provides a set of commands that allow you to flip a block 180 degrees about its horizontal or vertical axis. The commands effectively move a block's input and output ports to opposite sides of the block or reverse the ordering of the ports, depending on the block's port rotation type.

A block with the default rotation type has one flip command:

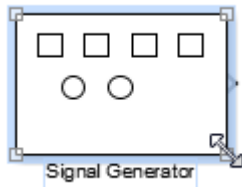
Diagram > Rotate & Flip > Flip Block (Ctrl+I). This command effectively moves the block's input and output ports to the side of the block opposite to the side on which they are initially located, i.e., from the left to the right side or from the top to the bottom side.



Resize a Block

To change the size of a block, select it, then drag any of its selection handles. While you hold down the mouse button, a dotted rectangle shows the new block size. When you release the mouse button, the block is resized.

For example, the following figure below shows a Signal Generator block being resized. The lower-right handle was selected and dragged to the cursor position. When the mouse button is released, the block takes its new size.



Tip Use the model editor's **resize blocks** commands to make one block the same size as another (see "Align, Distribute, and Resize Groups of Blocks" on page 4-22).

Displaying Parameters Beneath a Block

You can cause Simulink to display one or more of a block's parameters beneath the block. Specify the parameters to be displayed by using one of the following approaches:

- Enter an attributes format string in the **Attributes format string** field of the block's **Properties** dialog box (see "Set Block Properties" on page 23-15)
- Set the value of the block's `AttributesFormatString` property to the format string, using `set_param`

Drop Shadows

By default, blocks appear with a drop shadow.

To increase the depth of a block drop shadow:

- 1 Select the block.
- 2 Select **Diagram > Format > Block Shadow**.

For example, in this model, the `Constant1` block has the **Block Shadow** option enabled, and the `Constant2` block uses the default drop shadow.



To remove the default drop shadows for all blocks, select **File > Simulink Preferences > Editor Defaults > Use classic diagram theme**.

Manipulate Block Names

All block names in a model must be unique and must contain at least one character. By default, block names appear below blocks whose ports are on the sides, and to the left of blocks whose ports are on the top and bottom, as the following figure shows:



Note Simulink commands interprets a forward slash (/) as a block path delimiter. For example, the path vdp/Mu designates a block named Mu in the model named vdp. Therefore, avoid using forward slashes (/) in block names to avoid causing Simulink to interpret the names as paths.

Change Block Names

You can edit a block name in one of these ways:

- To replace the block name, click the block name, select the entire name, and then enter the new name.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To apply the block name edit, click the cursor anywhere else in the model, take any other action, or press **Esc**. If you try to change the name of a block to a name that already exists, Simulink displays an error message.

Note If you change the name of a library block, all links to that block become unresolved.

Change Font of Block Name

To modify the font used in a block name by selecting the block, then choosing the **Font Style** menu item from the **Diagram > Format** menu.

- 1 Select the block name.
- 2 Select **Diagram > Format > Font Style**.
The Select Font dialog box opens.
- 3 Select a font and specify other font characteristics, such as the font size.

Note Changing the block name font also changes the font of any text that appears inside the block.

This procedure also changes the font of any text that appears inside the block.

Change the Location of a Block Name

To change the location of the name of a selected block, use one of these approaches:

- Drag the block name to the opposite side of the block.
- Choose **Diagram > Rotate & Flip > Flip Block Name**. This command changes the location of the block name to the opposite side of the block.

For more information about block orientation, see “How to Rotate a Block” on page 23-23.

Hide a Block Name

By default the Simulink Editor displays the names of blocks (except for a few blocks, such as the Bus Creator block). To hide the name of a selected block, clear the **Diagram > Format > Show Block Name** menu option.

Specify Block Color

See “Specify Block Diagram Colors” on page 4-8 for information on how to set the color of a block.

Display Port Values

In this section...

“Port Value Data Tips” on page 23-30

“Display Value for a Port” on page 23-31

“Control the Port Value Display” on page 23-31

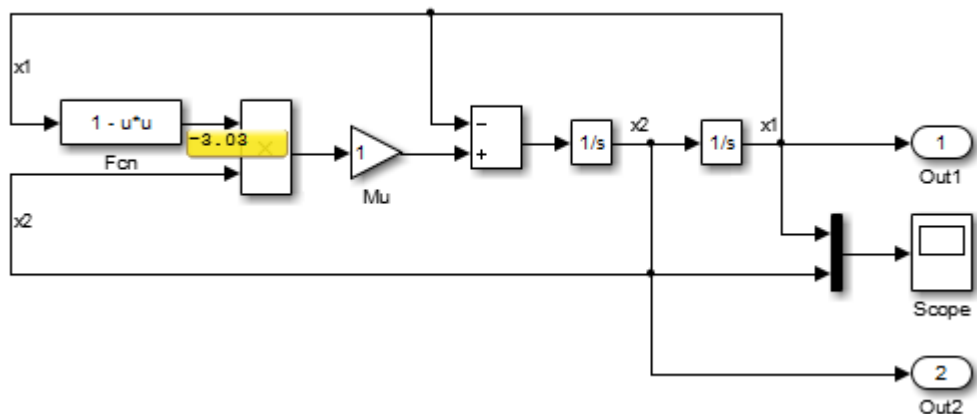
“Displayed Value When No Data Is Available” on page 23-33

“Port Value Display Limitations” on page 23-34

Port Value Data Tips

Signals can be data signals or non-data signals. For many blocks with data signals, Simulink can display port values (block output) as data tips on the block diagram while a simulation is running. The following model shows a port value data tip for the Fcn block, displaying an output value of -3.03 .

Displaying port value data tips can be helpful during interactive debugging of a model. The data tips do not persist when you save a model, so do not rely on data tips as a form of built-in documentation for the model.



In some cases, data signals are not accessible for display. In those cases, an empty label might be displayed, or action might be displayed for an action

subsystem. For more information, see “Port Value Display Limitations” on page 23-34.

Display Value for a Port

To display a data tip that shows the value of a port:

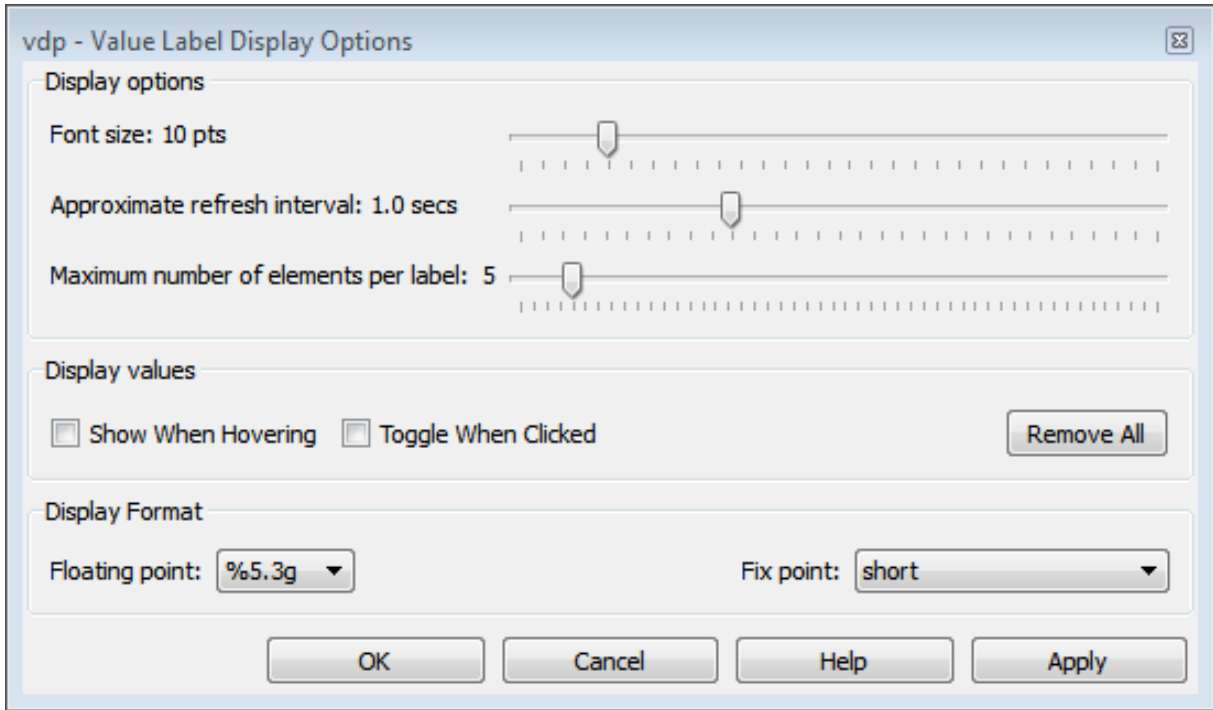
- 1 In the Simulink Editor model window, select a port.
- 2 Specify when the port value appears.
 - To display the port value when you click on a port, select **Display > Data Display in Simulation > Show Value Label of Selected Port**. The data tip remains until you remove it.
 - To display the port value when you hover the cursor over the port, select **Display > Data Display in Simulation > Show Value Labels When Hovering**. When you move the cursor away from the port, the data tip disappears.
 - To toggle between displaying and hiding the port value, select **Display > Data Display in Simulation > Toggle Value Labels When Clicked**. Reselecting that block turns off the display. If you select a block with this port value display method, then change the display method to **Show When Hovering**, the port value for that block continues to display.

To remove the all displayed port value data tips, select **Display > Data Display in Simulation > Remove All Value Labels**.

Control the Port Value Display

To specify port value display formatting and how frequently the display updates:

- 1 In the Simulink Editor model window, select a port.
- 2 Select **Display > Data Display in Simulation > Options**. The Value Label Display Options dialog box opens.



3 In the Value Label Display Options dialog box, select the appropriate option.

Port Value Data Tip Display Characteristic	What You Specify
When to display the port value	Choose Show When Hovering , Toggle When Clicked , or Remove All . For details, see “Display Value for a Port” on page 23-31.
Text size	To increase the size of the output display text, move the Font size slider to the right.
Floating point data format	Choose a floating point format from the Floating point list.

Port Value Data Tip Display Characteristic	What You Specify
Fixed-point data format	Choose a fixed-point format from the Fix point list.
Refresh interval (rate at which Simulink updates the output display) The refresh rate is approximate and is independent of simulation time.	To increase the interval, move the Approximate refresh interval: 1.0 slider to the right.
Number of elements in port value display	To change the number of elements to display in the port value display, move the Maximum number of elements per label: 3 slider to the left or right. This setting affects the number of elements displayed for vector or matrix signals with signal widths greater than 1. This setting does not affect scalar signals.

Displayed Value When No Data Is Available

Simulink displays an empty box when you toggle or hover on a block, indicating that no port value is available. You see the empty box if you did not run the simulation.

Also, if you toggle or hover on a block that Simulink optimizes out of a simulation (such as a virtual subsystem block), the model displays a wait string before displaying the port value during simulation.

To prevent the optimization delay of the port value display, designate the signal as a test point. This enables the port value display to update immediately.

Port Value Display Limitations

Performance

Enabling the hovering option, or toggling at least one block, slows down the simulation.

Accelerated Modes

Port values display as follows.

Accelerated Mode	Port Values...
Accelerator	Display as in Normal mode. See “Port Value Data Tips” on page 23-30.
Rapid Accelerator	Do not display. The limitation exists whether the model itself specifies accelerated simulation or the model is subordinate to a model that specifies accelerated simulation. For more information, see “Rapid Simulations”.

Signal Storage Reuse

Invoking the port value displays the text optimized when the software cannot determine the signal value, if you:

- Select **Show When Hovering** or **Toggle When Clicked**.
- Enable the **Configuration Parameters > Optimization > Signals and Parameters > Signal storage reuse** parameter.

Disabling signal storage reuse increases the amount of 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. However, Simulink does not display data for signals with some complex data types, such as bus

signals. In addition, the software shows the floating and fixed-point formats for only non-complex signal value displays.

Inline Parameters

For models that use inline parameters (models that have **Configuration Parameters > Optimization > Signals and Parameters > Inline parameters** enabled), port values are not displayed for signals such as:

- Constant time input to an Enable block
- Outputs of an Enable block that is never enabled

Subsystems

- You cannot add conditional breakpoints on inputs to, outputs from, or signals inside variant subsystems.
- 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.

Simulation Stepper

Port values displays do not always update 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 > Refine factor**) because Simulink updates port value displays only during major time steps.

Update Rates

Port value displays may update at a faster rate than the rate of the signal if the signal is driven by a block with a faster rate.

Command Line Simulations

For efficiency, port value displays are not supported during a command line simulation.

Control and Display the Sorted Order

In this section...
“What Is Sorted Order?” on page 23-36
“Display the Sorted Order” on page 23-36
“Sorted Order Notation” on page 23-37
“How Simulink Determines the Sorted Order” on page 23-47
“Assign Block Priorities” on page 23-50
“Rules for Block Priorities” on page 23-51
“Block Priority Violations” on page 23-54

What Is Sorted Order?

During the updating phase of simulation, Simulink determines the order in which to invoke the block methods during simulation. This block invocation ordering is the *sorted order*.

You cannot set this order, but you can assign priorities to nonvirtual blocks to indicate to Simulink their execution order relative to other blocks. Simulink tries to honor block priority settings, unless there is a conflict with data dependencies. To confirm the results of priorities that you have set, or to debug your model, display and review the sorted order of your nonvirtual blocks and subsystems.

Note For more information about block methods and execution, see:

- “Block Methods” on page 3-16
 - “Conditional Execution Behavior” on page 7-50
-

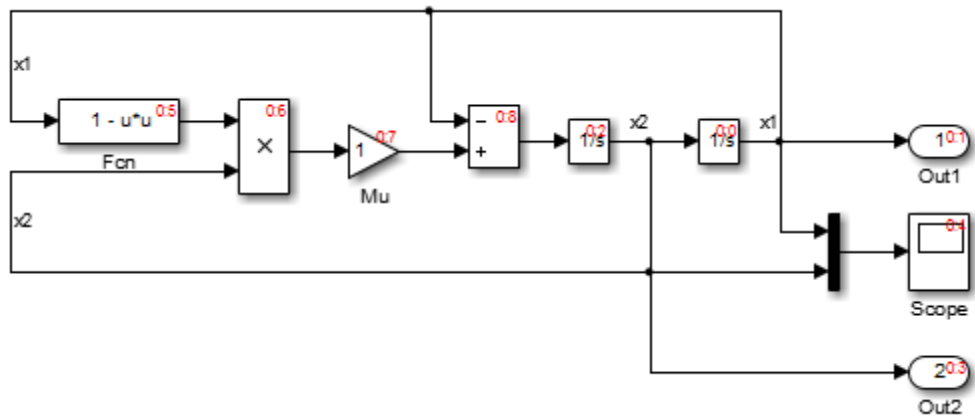
Display the Sorted Order

To display the sorted order of the vdp model:

1 Open the van der Pol equation model:

vdp

2 In the model window, select **Display > Blocks > Sorted Execution Order**.



Simulink displays a notation in the top-right corner of each nonvirtual block and each nonvirtual subsystem. These numbers indicate the order in which the blocks execute. The first block to execute has a sorted order of 0.

For example, in the van der Pol equation model, the Integrator block with the sorted order 0:0 executes first. The Out1 block, with the sorted order 0:1, executes second. Similarly, the remaining blocks execute in numeric order from 0:2 to 0:8.

You can save the sorted order setting with your model. To display the sorted order when you reopen the model, select **Simulation > Update diagram**.

Sorted Order Notation

The sorted order notation varies depending on the type of block. The following table summarizes the different formats of sorted order notation. Each format is described in detail in the sections that follows the table.

Block Type	Sorted Order Notation	Description
“Nonvirtual Blocks” on page 23-41	$s:b$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b specifies the block position within the sorted order for the designated execution context.
“Nonvirtual Subsystems” on page 23-41 (not including function-call and action subsystems)	$s:b$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem. • b specifies the block position within the sorted order for the designated execution context.
“Virtual Blocks and Subsystems” on page 23-43	Not applicable	Virtual blocks do not execute.
Action Subsystems	$s:b'$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b' is the block index of the action block (but not of the action subsystem).
“Function-Call Subsystems” on page 23-44 and Function-Call models	One non-grounded initiator: $s:b^i$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b^{in} is the block index of the n-th root initiator in the subsystem’s hierarchy.
	Two or more initiators: <ul style="list-style-type: none"> • $s:b^{i1}, s:b^{i2}, \dots, s:b^{in}$ where n is the number of non-grounded initiators. 	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b^i is the block index of the root initiator in the subsystem’s hierarchy.

Block Type	Sorted Order Notation	Description
	Initiators are either Ground blocks or are unconnected: <ul style="list-style-type: none"> • $s:G$ 	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • G indicates that all function-call initiators are grounded.
Branched Function-Call Subsystems and Branched Function-Call Models	$s:b^i[B_k]$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b^i is the block index of the root initiator in the subsystem's hierarchy. • B_k indicates that it is a branched function-call subsystem with branch index k.
Function-Call Feedback Latch Blocks	$s:b^i[B_m]$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • b^i is the block index of the root initiator in the subsystem's hierarchy. • B_m indicates that it is a branched subsystem with branch index m.
Blocks in a model with asynchronous function-call inputs	Function-call root-level Inport and Outport blocks: F_i, F_j, \dots	<ul style="list-style-type: none"> • F indicates that it is executed in a function-call context. • i is the function-call index.
	Root-level data Inport blocks: If they drive a function-call subsystem, it is the function-call index of the subsystem. If they are part of the model that is not driven by asynchronous function-call inputs, then no function-call index is displayed.	

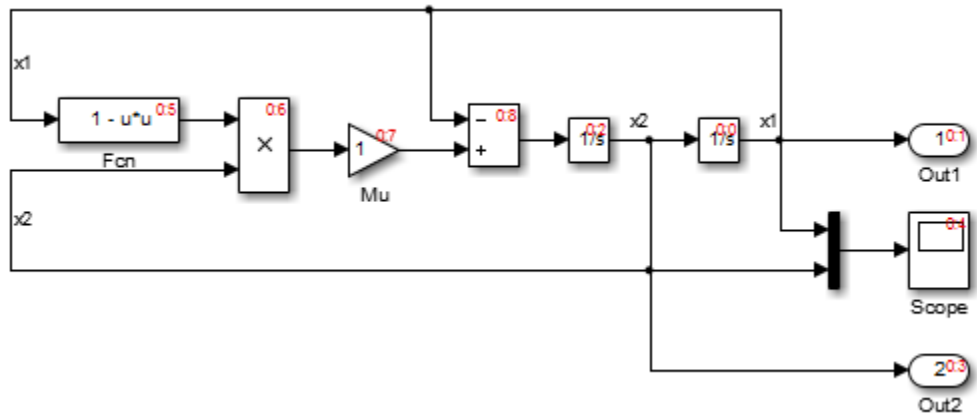
Block Type	Sorted Order Notation	Description
	Merge and Data Store Memory blocks: F_i, F_j, \dots	<ul style="list-style-type: none"> • F indicates that it is executed in a function-call context. • i is the function-call index.
	Root-level function-call blocks: F_i Root-level branched function-call blocks: $F_i[B_k]$	<ul style="list-style-type: none"> • F indicates that it is executed in a function-call context. • i is the function-call index.
Blocks inside Export Function Models	Function-call root-level Inport and Outport blocks: F_i	<ul style="list-style-type: none"> • F indicates that it is a function-call block. • i is the execution order for the function-call root-level Inport or Outport block in normal simulation mode.
	Root-level function-call subsystems: F_i Root-level branched function-call subsystems: $F_i[B_k]$	<ul style="list-style-type: none"> • F indicates that it is a function-call block. • i is the execution order for the function-call root-level Inport block in normal simulation mode. • B_k indicates a branched function-call subsystem with index k.
	Merge and Data Store Memory blocks: F_i, F_j, \dots	<ul style="list-style-type: none"> • F indicates that it is a function-call block. • i is the block execution index.
	Root-level data Inport blocks: Same execution order as the function-call subsystems they are driving.	
“Bus-Capable Blocks” on page 23-47	$s:B$	<ul style="list-style-type: none"> • s is the system index of the model or subsystem the block resides in. For root-level models, s is always 0. • B indicates a bus-capable block.

- “Nonvirtual Blocks” on page 23-41
- “Nonvirtual Subsystems” on page 23-41
- “Virtual Blocks and Subsystems” on page 23-43

- “Function-Call Subsystems” on page 23-44
- “Function-Call Split Blocks” on page 23-46
- “Export-Function Models” on page 23-47
- “Bus-Capable Blocks” on page 23-47

Nonvirtual Blocks

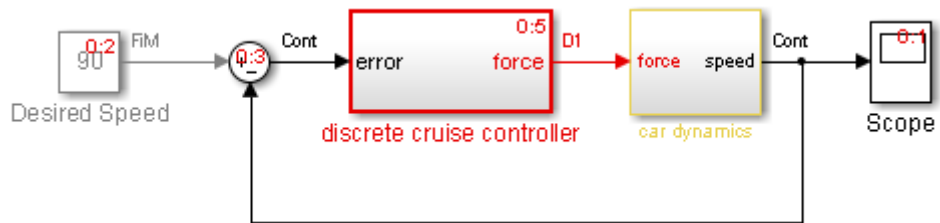
In the van der Pol equation model, all the nonvirtual blocks in the model have a sorted order. The system index for the top-level model is 0, and the block execution order ranges from 0 to 8.



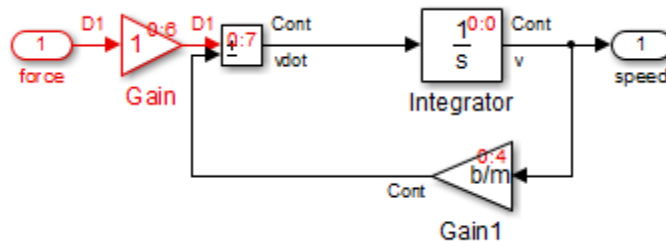
Nonvirtual Subsystems

The following model contains an atomic, nonvirtual subsystem named Discrete Cruise Controller.

When you enable the sorted order display for the root-level system, Simulink displays the sorted order of the blocks.



The Scope block in this model has the lowest sorted order, but its input depends on the output of the Car Dynamics subsystem. The Car Dynamics subsystem is virtual, so it does not have a sorted order and does not execute as an atomic unit. However, the blocks within the subsystem execute at the root level, so the Integrator block in the Car Dynamics subsystem executes first. The Integrator block sends its output to the Scope block in the root-level model, which executes second.



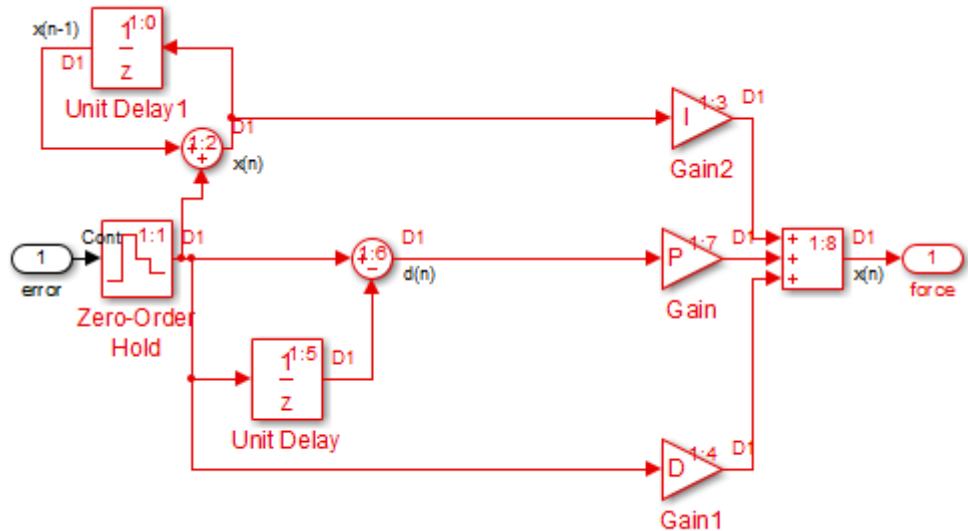
The Discrete Cruise Controller subsystem has a sorted order of 0:5:

- 0 indicates that this atomic subsystem is part of the root level of the hierarchal system comprised of the primary system and the two subsystems.
- 5 indicates that the atomic subsystem is the sixth block that Simulink executes relative to the blocks within the root level.

The sorted order of each block inside the Discrete Cruise Controller subsystem has the form 1:b, where:

- 1 is the system index for that subsystem.

- b is the block position in the execution order. In the Discrete Cruise Controller subsystem, the sorted order ranges from 0 to 8.

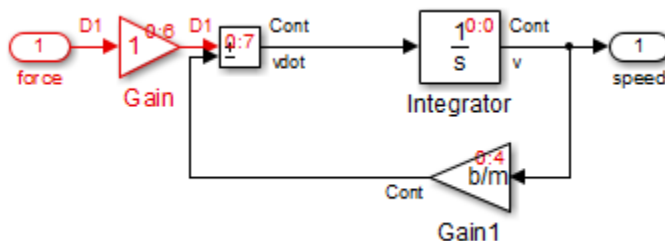


Virtual Blocks and Subsystems

Virtual blocks, such as the Mux block, exist only graphically and do not execute. Consequently, they are not part of the sorted order and do not display any sorted order notation.

Virtual subsystems do not execute as a unit, and like a virtual block, are not part of the sorted order. The blocks inside the virtual subsystem are part of the root-level system sorted order, and therefore share the system index.

In the model in “Nonvirtual Subsystems” on page 23-41, the virtual subsystem Car Dynamics does not have a sorted order. However, the blocks inside the subsystem have a sorted order in the execution context of the root-level model. The blocks have the same system index as the root-level model. The Integrator block inside the Car Dynamics subsystem has a sorted order of 0:0, indicating that the Integrator block is the first block executed in the context of the top-level model.

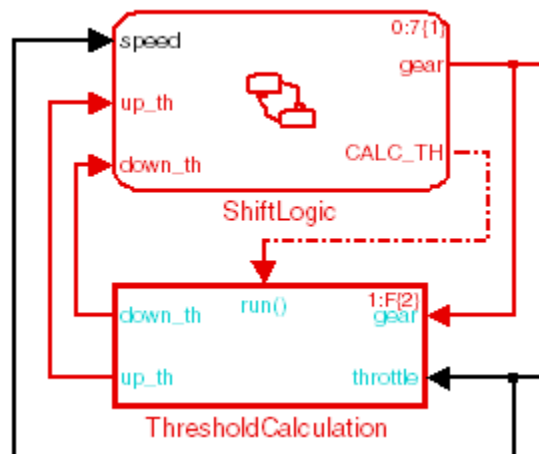


Function-Call Subsystems

Single Initiator. For each function-call subsystem or model attached to a single initiator, Simulink assigns the block position F . The function-call subsystem (or model) executes when the initiator invokes the function-call subsystem (or model) and, therefore, does not have a sorted order independent of its initiator. Specifically, for a subsystem that connects to one initiator, Simulink uses the notation $s:F\{s_i\}$, where s is the index of the system that contains the initiator.

For example, the sorted order for the ThresholdCalculation subsystem is $1:F\{2\}$:

- 1 is the index of the system that contains the initiator, in this case, the ShiftLogic Stateflow chart.
- F indicates that this function-call subsystem connects to one initiator.
- 2 is the system index for the ThresholdCalculation subsystem.

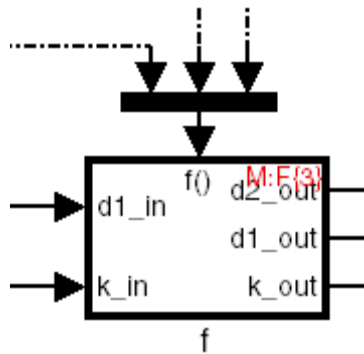


Multiple Initiators. For a function-call subsystem that connects to more than one initiator, the sorted order notation depends on the execution context of the initiators:

- If the initiators all execute at the same level of the model hierarchy, the notation is $s:F\{s_i\}$.
- If the initiators execute at different levels of the model hierarchy, the notation is $M:F\{s_i\}$

For example, open the `s1_subsys_fcncal16` model. The `f` subsystem has three initiators, two from the Stateflow chart, `Chart1`, and one from the Stateflow chart, `Chart`.

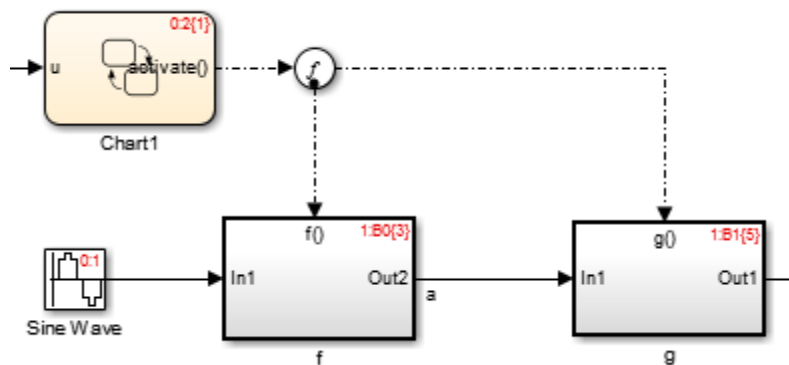
Because `Chart1` has a system index of 2 and `Chart` has a system index of 1, the initiators execute at different levels in the model hierarchy. The Function-Call Subsystem `f` has a sorted order notation of $M:F\{3\}$.



Function-Call Split Blocks

When a function-call signal is branched using a Function-Call Split block, Simulink displays the order in which subsystems (or models) that connect to the branches execute when the initiator invokes the function call. Simulink uses the notation $s : Bb$ to indicate this order, where B stands for branch. The block position b ranges from 0 to one less than the total number of subsystems (or models) that connect to branches (B0, B1, etc.). When the function-call is invoked, the subsystems execute in ascending order based on this number.

For example, open the `s1_subsys_fnccall111` model and display the sorted order. The sorted order indicates that the subsystem `f` (B0) executes before the subsystem `g` (B1).



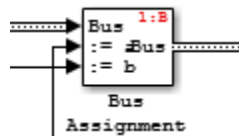
Export-Function Models

The execution order for root-level data Inport blocks is determined by the execution order of the Inport blocks that drive the function-call subsystems that the data blocks are feeding into. In the example below, the Inport blocks fcIn0 and fcIn1 drive the subsystems Function-Call Subsystem1 and Function-Call Subsystem 4, both of which are being driven by root-level data Inport block In1. Hence, it's execution order is F0,F1.

Bus-Capable Blocks

A bus-capable block does not execute as a unit and therefore does not have a unique sorted order. Such a block displays its sorted order as s:B where B stands for bus.

For example, open the `sldemo_bus_arrays` model and display the sorted order. Open the For Each Subsystem to see that the sorted order for the Bus Assignment block appears as 1:B.



For more information, see “Bus-Capable Blocks” on page 51-22.

How Simulink Determines the Sorted Order

Direct-Feedthrough Ports Impact on Sorted Order

To ensure that the sorted 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:

- 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 Sorting Blocks

To sort blocks, Simulink uses the following rules:

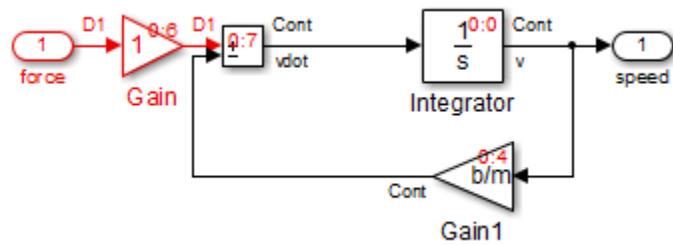
- If a block drives the direct-feedthrough port of another block, the block must appear in the sorted order ahead of the block that it drives.
This rule ensures that the direct-feedthrough inputs to blocks are valid when Simulink invokes block methods that require current inputs.
- Blocks that do not have direct-feedthrough inputs can appear anywhere in the sorted 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 sorted order satisfies this rule. This arrangement allows Simulink to ignore these blocks during the sorting process.

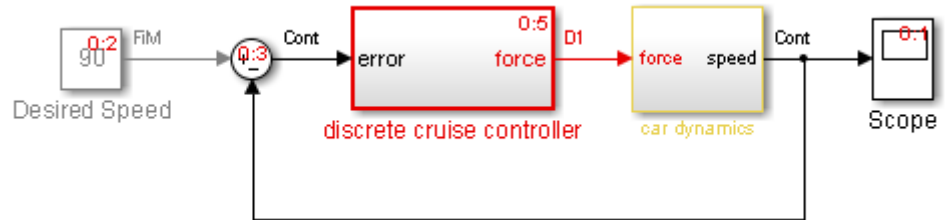
Applying these rules results in the sorted order. Blocks without direct-feedthrough ports appear at the beginning of the list in no particular order. These blocks are followed by blocks with direct-feedthrough ports arranged such that they can supply valid inputs to the blocks which they drive.

The following model, from “Nonvirtual Subsystems” on page 23-41, illustrates this result. The following blocks do not have direct-feedthrough and therefore appear at the beginning of the sorted order of the root-level system:

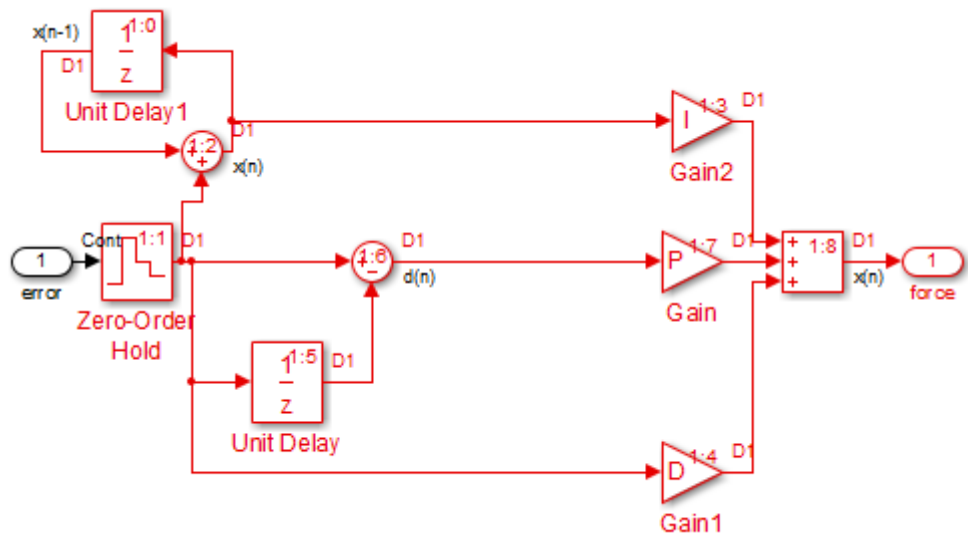
- Integrator block in the Car Dynamics virtual subsystem



- 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.



Assign Block Priorities

You can assign a priority to a nonvirtual block or to an entire subsystem. Higher priority blocks appear before lower priority blocks in the sorted order. The lower the number, the higher the priority.

- “Assigning Block Priorities Programmatically” on page 23-50
- “Assigning Block Priorities Interactively” on page 23-51

Assigning Block Priorities Programmatically

To set priorities programmatically, use the command:

```
set_param(b, 'Priority', 'n')
```

where

- *b* is the block path.
- *n* is any valid integer. (Negative integers and 0 are valid priority values.)

Assigning Block Priorities Interactively

To set the priority of a block or subsystem interactively:

- 1 Right-click the block and select **Properties**.
- 2 On the **General** tab, in the **Priority** field, enter the priority.

Rules for Block Priorities

Simulink honors the block priorities that you specify unless they violate data dependencies. (“Block Priority Violations” on page 23-54 describes situations that cause block property violations.)

In assessing priority assignments, Simulink attempts to create a sorted order such that the priorities for the individual blocks within the root-level system or within a nonvirtual subsystem are honored relative to one another.

Three rules pertain to priorities:

- “Priorities Are Relative” on page 23-51
- “Priorities Are Hierarchical” on page 23-52
- “Lack of Priority May Not Result in Low Priority” on page 23-53

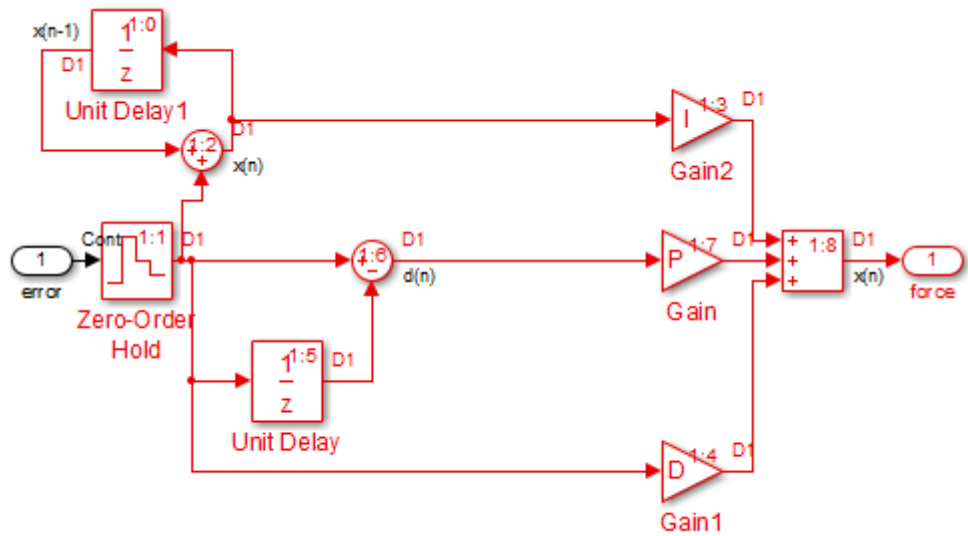
Priorities Are Relative

Priorities are relative; the priority of a block is relative to the priority of the blocks within the same system or subsystem.

For example, suppose you set the following priorities in the Discrete Cruise Controller subsystem in the model in “Nonvirtual Subsystems” on page 23-41.

Block	Priority
Gain	3
Gain1	2
Gain2	1

After updating the diagram, the sorted order for the Gain blocks is as follows.



The sorted order values of the Gain, Gain1, and Gain2 blocks reflect the respective priorities assigned: Gain2 has highest priority and executes before Gain1 and Gain; Gain1 has second priority and executes after Gain2; and Gain executes after Gain1. Simulink takes into account the assigned priorities relative to the other blocks in that subsystem.

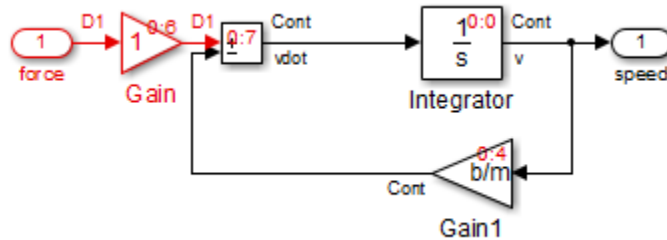
The Gain blocks are not the first, second, and third blocks to execute. Nor do they have consecutive sorted orders. The sorted order values do not necessarily correspond to the priority values. Simulink arranges the blocks so that their priorities are honored relative to each other.

Priorities Are Hierarchical

In the Car Dynamics virtual subsystem, suppose you set the priorities of the Gain blocks as follows.

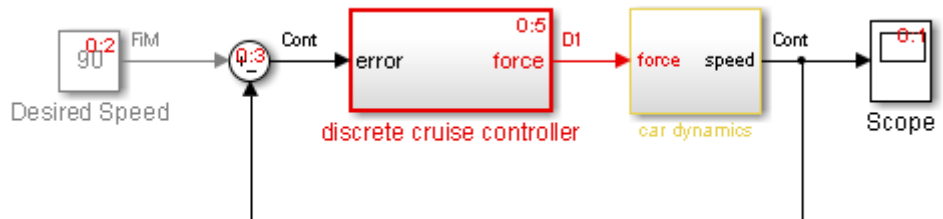
Block	Priority
Gain	2
Gain1	1

After updating the diagram, the sorted order for the Gain blocks is as illustrated. With these priorities, Gain1 always executes before Gain.



You can set a priority of 1 to one block in each of the two subsystems because of the hierarchical nature of the subsystems within a model. Simulink never compares the priorities of the blocks in one subsystem to the priorities of blocks in any other subsystem.

For example, consider this model again.



The blocks within the Car Dynamics virtual subsystem are part of the root-level system hierarchy and are part of the root-level sorted order. The Discrete Cruise Controller subsystem has an independent sorted order with the blocks arranged consecutively from 1:0 to 1:7.

Lack of Priority May Not Result in Low Priority

A lack of priority does not necessarily result in a low priority (higher sorting order) for a given block. Blocks that do not have direct-feedthrough ports execute before blocks that have direct-feedthrough ports, regardless of their priority.

If a model has two atomic subsystems, A and B, you can assign priorities of 1 and 2 respectively to A and B. This priority causes all the blocks in A to execute before any of the blocks in B. The blocks within an atomic subsystem execute as a single unit, so the subsystem has its own system index and its own sorted order.

Block Priority Violations

Simulink software honors the block priorities that you specify unless they violate data dependencies. If Simulink is unable to honor a block priority, it displays a **Block Priority Violation** diagnostic message.

As an example:

- 1 Open the `sldemo_bounce` model.

Notice that the output of the Memory block provides the input to the Coefficient of Restitution Gain block.

- 2 Set the priority of the Coefficient of Restitution block to 1, and set the priority of the Memory block to 2.

Setting these priorities specifies that the Coefficient of Restitution block execute before the Memory block. However, the Coefficient of Restitution block depends on the output of the Memory block, so the priorities you just set violate the data dependencies.

- 3 In the model window, enable sorted order by selecting **Format > Block displays > Sorted Order**.

- 4 Select **Simulation > Update Diagram**.

The block priority violation warning appears in the MATLAB Command Window. The warning includes the priority for the respective blocks:

```
Warning: Unable to honor user-specified priorities.  
'sldemo_bounce/Memory' (pri=[2]) has to execute  
before 'sldemo_bounce/Coefficient of Restitution'  
(pri=[1]) to satisfy data dependencies
```

- 5** Remove the priorities from the Coefficient of Restitution and Memory blocks and update the diagram again to see the correct sorted order.

Access Block Data During Simulation

In this section...

“About Block Run-Time Objects” on page 23-56

“Access a Run-Time Object” on page 23-56

“Listen for Method Execution Events” on page 23-57

“Synchronizing Run-Time Objects and Simulink Execution” on page 23-58

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 46-35) 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 block’s run-time object. 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 “Virtual Blocks” on page 23-2) 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.

```
% Get the full path to the S-function block  
blk = 'sldemo_msfcn_lms/LMS Adaptive';
```

```
% Attach the event-listener function to the S-function
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, eventData)

% The figure's handle is stored in the block's UserData
hFig = get_param(block.BlockHandle, 'UserData');
tAxis = findobj(hFig, 'Type', 'axes');

tAxis = tAxis(2);
tLines = findobj(tAxis, 'Type', 'Line');

% The filter coefficients are stored in the block run-time
% object's second DWork vector.
est = block.Dwork(2).Data;

set(tLines(3), 'YData', est);
```

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(gcf, '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 on the **Optimization > Signals and Parameters** pane (see “Signal storage reuse”).

Configure a Block for Code Generation

Use the **State Attributes** pane of a Block Parameters dialog box to specify Simulink Coder code generation options for blocks with discrete states. See “States” in the Simulink Coder documentation.

Working with Block Parameters

- “About Block Parameters” on page 24-2
- “Set Block Parameters” on page 24-4
- “Specify Parameter Values” on page 24-6
- “Check Parameter Values” on page 24-10
- “Tunable Parameters” on page 24-14
- “Inline Parameters” on page 24-15
- “Structure Parameters” on page 24-17

About Block Parameters

Most Simulink blocks have attributes whose values you can specify to customize the block. Some attributes are common to all Simulink blocks, such as a block's name and foreground color. Other attributes are specific to a block, such as the gain of a Gain block. Simulink associates a *block parameter* with each user-specifiable attribute of a block. Block parameters fall into two categories. A *mathematical parameter* is a parameter used to compute the value of a block's output, such as a Gain block's **Gain** parameter. All other parameters are *configuration parameters*, such as a Gain block's **Name** parameter.

A parameterized block effectively represents a family of similar blocks. For example, when creating a model, you can set the **Constant value** parameter of each instance of the Constant block separately so that each instance behaves differently. Because it allows each standard block to represent a family of blocks, block parameterization greatly increases the modeling power of the standard Simulink libraries.

You specify a block attribute value by setting its associated block parameter. For example, to set the foreground color of a block to red, you set the value of its foreground color parameter to the string 'red'. You can set the value programmatically, using the `set_param` command, or by using the block parameters dialog box. See “Common Block Properties” and “Block-Specific Parameters” for the names, usages, and valid settings for Simulink block parameters.

You can use MATLAB expressions to specify block parameter values. The expressions can include the names of workspace variables. Simulink evaluates the expressions before running a simulation, resolving any names to values as described in “Symbol Resolution” on page 4-83. Do not confuse Simulink block parameters with Simulink parameter objects. See “Block Parameters” on page 3-9 and `Simulink.Parameter` for more information.

A *tunable parameter* is a block parameter whose value can be changed during simulation without recompiling the model. In general, you can change the values of mathematical parameters, but not configuration parameters. If a parameter is not tunable and simulation is running, the dialog box control that sets the parameter value is disabled.

If you change a tunable parameter value programmatically while simulation is running, Simulink pauses the simulation, makes the change, then resumes the simulation after the change is complete. Tunable parameter changes take effect at the start of the next time step after simulation resumes.

You can use the **Inline parameters** check box on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box to specify that all parameters in your model are nontunable except for those that you specify. This action can speed up execution of large models and enable generation of faster code from your model. See “Optimization Pane: Signals and Parameters” for more information.

You can separately define each MATLAB variable intended for use in block parameter expressions. However, this technique has several disadvantages, such as cluttering the base workspace and providing no convenient way to group related parameters. To avoid these disadvantages, you can combine numeric base workspace variables into a MATLAB structure, then:

- Dereference the structure fields to provide values used in block parameter expressions
- Pass a whole structure as an argument to a masked subsystem or a referenced model
- Set the structure to be tunable or nontunable as you could a separately defined variable

A structure used for any of these purposes can contain only numeric data. See “Structure Parameters” on page 24-17 for more information.

Set Block Parameters

You can use the Simulink `set_param` command to set the value of any Simulink block parameter. In addition, you can set many block parameters via Simulink dialog boxes and menus. These include:

- **Format** menu

The Simulink Editor's **Diagram > Format** menu allows you to specify attributes of the currently selected block that are visible on the model's block diagram, such as the block's name and color (see "Change the Appearance of a Block" on page 23-23 for more information).

- **Block Properties** dialog box

Specifies various attributes that are common to all blocks (see "Set Block Properties" on page 23-15 for more information).

- **Block Parameter** dialog box

Every block has a dialog box that allows you to specify values for attributes that are specific to that type of block. See "Display a Block Parameter Dialog Box" on page 24-4 for information on displaying a block's parameter dialog box.

- Model Explorer

The Model Explorer allows you to quickly find one or more blocks and set their properties, thus facilitating global changes to a model, for example, changing the gain of all of a model's Gain blocks. See "Model Explorer Overview" on page 9-2 for more information.

Display a Block Parameter Dialog Box

To open a block parameter dialog box, in the Simulink Editor, use one of these techniques:

- Select the block and choose **Block Parameters** from the **Diagram** menu or from the block's context (right-click) menu.
- Double-click a block.
- From the block's context menu, select **Block Properties**. In the Block Properties dialog box, in the **General** tab, click **Open Block**.

Note Some blocks, such as Scope blocks, do not have a Block Parameters dialog box.

Double-clicking a block or using the Block Properties dialog box **Open Block** link to open the Block Parameters dialog box works for all blocks that have parameter dialog boxes, except for Subsystem and Model blocks. Use the Simulink Editor **Diagram** menu or the block context menu to open the Block Parameters dialog box for Subsystem and Model blocks.

Specify Parameter Values

In this section...

“About Parameter Values” on page 24-6

“Use Workspace Variables in Parameter Expressions” on page 24-6

“Resolve Variable References in Block Parameter Expressions” on page 24-7

“Use Parameter Objects to Specify Parameter Values” on page 24-7

“Convert Numeric Variable into Simulink.Parameter Object” on page 24-8

“Determine Parameter Data Types” on page 24-8

About Parameter Values

Many block parameters, including mathematical parameters, accept MATLAB expression strings as values. When Simulink compiles a model, for example, at the start of a simulation or when you update the model, Simulink sets the compiled values of the parameters to the result of evaluating the expressions.

Use Workspace Variables in Parameter Expressions

Block parameter expressions can include variables defined in the model’s mask and model workspaces and in the MATLAB workspace. Using a workspace variable facilitates updating a model that sets multiple block parameters to the same value, i.e., it allows you to update multiple parameters by setting the value of a single workspace variable. For more information, see “Symbol Resolution” on page 4-83 and “Numeric Values with Symbols” on page 4-85.

Using a workspace variable also allows you to change the value of a parameter during simulation without having to open a block’s parameter dialog box. For more information, see “Tunable Parameters” on page 24-14.

Note If you have a Simulink Coder license, and you plan to generate code from a model, you can use workspace variables to specify the name, data type, scope, volatility, tunability, and other attributes of variables used to represent the parameter in the generated code. For more information, see “Parameters” in the Simulink Coder documentation.

Resolve Variable References in Block Parameter Expressions

When evaluating a block parameter expression that contains a variable, Simulink by default searches the workspace hierarchy. If the variable is not defined in any workspace, Simulink halts compilation of the model and displays an error message. See “Symbol Resolution” on page 4-83 and “Numeric Values with Symbols” on page 4-85 for more information.

Use Parameter Objects to Specify Parameter Values

You can use `Simulink.Parameter` objects in parameter expressions to specify parameter values. For example, `K` and `2*K` are both valid parameter expressions where `K` is a workspace variable that references a `Simulink.Parameter` object. In both cases, Simulink uses the parameter object’s `Value` property as the value of `K`.

You cannot use arrays of `Simulink.Parameter` objects as block parameters.

See “Symbol Resolution” on page 4-83 and “Numeric Values with Symbols” on page 4-85 for more information.

Using parameter objects to specify parameters can facilitate tuning parameters in some applications. See “Using a Parameter Object to Specify a Parameter As Nonlinear” on page 24-16 and “Parameterize Model References” on page 6-54 for more information.

Note Do not use expressions of the form `p.Value` where `p` is a parameter object in block parameter expressions. Such expressions cause evaluation errors when Simulink compiles the model.

Convert Numeric Variable into Simulink.Parameter Object

You can convert a numeric variable into a `Simulink.Parameter` object as follows.

```
myVar = 5; /* Define numerical variable in base workspace
myObject = Simulink.Parameter; /* Create data object
myObject.Value = myVar; /* Assign variable value to data object value
```

Determine Parameter Data Types

When Simulink compiles a model, each of the model's blocks determines a data type for storing the values of its parameters whose values are specified by MATLAB parameter expressions.

Most blocks use internal rules to determine the data type assigned to a specific parameter. Exceptions include the Gain block, whose parameter dialog box allows you to specify the data type assigned to the compiled value of its Gain parameter. You can configure your model to check whether the data type assigned to a parameter can accommodate the parameter value specified by the model (see “Data Validity Diagnostics Overview”).

Obtain Parameter Information

You can use `get_param` to find the system and block parameter values for your model. See “Model Parameters” and “Common Block Properties” for arguments `get_param` accepts.

The model's signal attributes and parameter expressions must be evaluated before some parameters are properly reported. This evaluation occurs during the simulation compilation phase. Alternatively, you can compile your model without first running it, and then obtain parameter information. For instance, to access the port width, data types, and dimensions of the blocks in your model, enter the following at the command prompt:

```
modelName([],[],[],'compile')
q=get_param(gcf,'PortHandles');
get_param(q.Inport,'CompiledPortDataType')
get_param(q.Inport,'CompiledPortWidth')
get_param(q.Inport,'CompiledPortDimensions')
```

```
modelName([],[],[],'term')
```

Check Parameter Values

In this section...
“About Value Checking” on page 24-10
“Blocks That Perform Parameter Range Checking” on page 24-10
“Specify Ranges for Parameters” on page 24-11
“Perform Parameter Range Checking” on page 24-12

About Value Checking

Many blocks perform range checking of their mathematical parameters. Generally, blocks that allow you to enter minimum and maximum values check to ensure that the values of applicable parameters lie within the specified range.

Blocks That Perform Parameter Range Checking

The following blocks perform range checking for their parameters:

Block	Parameters Checked
Constant	Constant value
Data Store Memory	Initial value
Gain	Gain
Interpolation Using Prelookup	Table data
1-D Lookup Table	Table data
2-D Lookup Table	Table data
n-D Lookup Table	Table data
Relay	Output when on Output when off
Repeating Sequence Interpolated	Vector of output values

Block	Parameters Checked
Repeating Sequence Stair	Vector of output values
Saturation	Upper limit Lower limit

Specify Ranges for Parameters

In general, use the **Output minimum** and **Output maximum** parameters that appear on a block parameter dialog box to specify a range of valid values for the block parameters. The following exceptions apply:

- For the Gain block, use the **Parameter minimum** and **Parameter maximum** fields to specify a range for the **Gain** parameter.
- For the Data Store Memory block, use the **Minimum** and **Maximum** fields to specify a range for the **Initial value** parameter.

When specifying minimum and maximum values that constitute a range, enter only expressions that evaluate to a finite, scalar, real number with `double` data type. The default values for the minimum and maximum are `[]` (unspecified). The scalar values that you specify are subject to expansion, for example, when the block parameters that Simulink checks are nonscalar (see “Scalar Expansion of Inputs and Parameters” on page 50-38).

Note You cannot specify the minimum or maximum value as `NaN`, `inf`, or `-inf`.

Specifying Ranges for Complex Numbers

When you specify a minimum or maximum value for a parameter that is a complex number, the specified minimum and maximum 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 $(\sqrt{a^2+b^2})$

Perform Parameter Range Checking

You can initiate parameter range checking in the following ways:

- When you click the **OK** or **Apply** button on a block parameter dialog box, the block performs range checking for its parameters. However, the block checks only the parameters that it can readily evaluate. For example, the block does not check parameters that use an undefined workspace variable.
- In the Simulink Editor, when you start a simulation or select **Simulation > Update Diagram**, Simulink performs parameter range checking for all blocks in that model.

Simulink performs parameter range checking by comparing the values of applicable block parameters with both the specified range (see “Specify Ranges for Parameters” on page 24-11) and the block data type. That is, Simulink performs the following check:

$$\text{DataTypeMin} \quad \text{MinValue} \quad \text{VALUE} \quad \text{MaxValue} \quad \text{DataTypeMax}$$

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 numeric value of a block parameter.
- **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.

When Simulink detects a parameter value that violates the check, it displays an error message. For example, consider a model that contains a Constant block whose

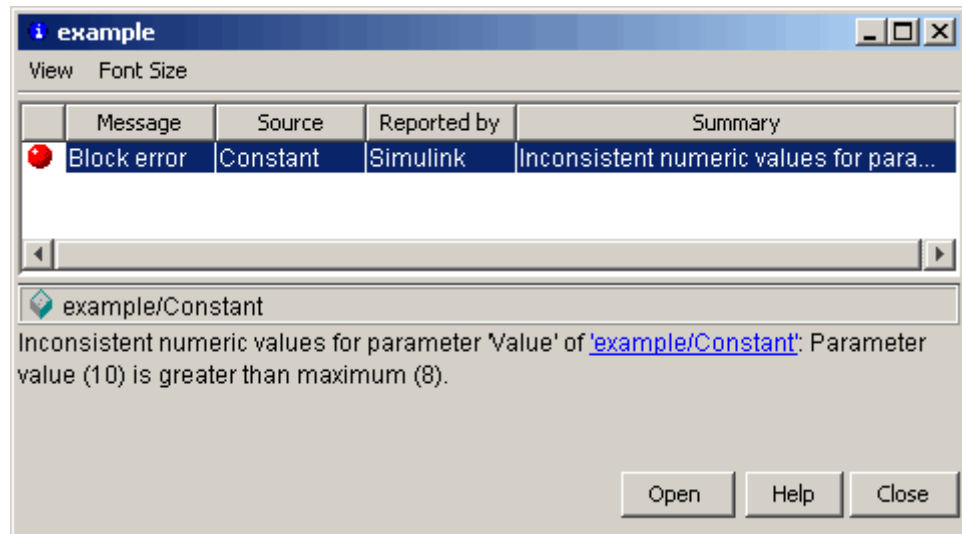
- **Constant value** parameter specifies the variable `const`, which you have yet to define in a workspace.
- **Output minimum** and **Output maximum** parameters are set to 2 and 8, respectively.

- **Output data type** parameter is set to `uint8`.

In this situation, Simulink does not perform parameter range checking when you click the **OK** button on the Constant block dialog box because the variable `const` is undefined. But suppose you define its value by entering

```
const = 10
```

at the MATLAB prompt, and then you update the diagram (see “Update a Block Diagram” on page 1-32). Simulink displays the following error message:



Tunable Parameters

In this section...
“About Tunable Parameters” on page 24-14
“Tune a Block Parameter” on page 24-14

About Tunable Parameters

Simulink lets you change the values of many block parameters during simulation. Such parameters are called *tunable parameters*. In general, only parameters that represent mathematical variables, such as the Gain parameter of the Gain block, are tunable. Parameters that specify the appearance or structure of a block, e.g., the number of inputs of a Sum block, or when it is evaluated, e.g., a block’s sample time or priority, are not tunable.

You can tell whether a particular parameter is tunable by examining its edit control in the block’s dialog box or Model Explorer during simulation. If the control is disabled, the parameter is nontunable. You cannot tune inline parameters. See “Inline Parameters” on page 24-15 for more information.

Tune a Block Parameter

You can use a block’s dialog box or the Model Explorer to modify the tunable parameters of any block. To use the block’s parameter dialog box, open the block’s parameter dialog box, change the value displayed in the dialog box, and click the dialog box’s **OK** or **Apply** button.

You can also tune a parameter at the MATLAB command line, using either the `set_param` command or by assigning a new value to the MATLAB workspace variable that specifies the parameter’s value. In either case, you must update the model’s block diagram for the change to take effect (see “Update a Block Diagram” on page 1-32).

Inline Parameters

In this section...
“About Inline Parameters” on page 24-15
“Specify Some Parameters as Noninline” on page 24-15

About Inline Parameters

The **Inline parameters** option (see “Inline parameters”) controls how mathematical block parameters appear in code generated from the model. When this optimization is off (the default), a model’s mathematical block parameters appear as variables in the generated code. As a result, you can tune the parameters both during simulation and when executing the code.

When **Inline parameters** is selected, the parameters appear in the generated code as inline numeric constants. This reduces the generated code’s memory and processing requirements. However, because the inline parameters appear as constants in the generated code, you cannot tune them during code execution. To ensure that simulation and generated code execution fully correspond, Simulink prevents you from changing the values of block parameters during simulation when **Inline parameters** is selected.

Note Simulink ignores tunable parameter specifications in the Model Parameter Configuration dialog box if the model is a referenced model or contains any Model blocks. Do not use this dialog box to override the inline parameters optimization for selected parameters to permit them to be tuned. Instead, see “Parameterize Model References” on page 6-54 for alternate techniques. If you define tunable parameters using `Simulink.Parameter` objects, you can tune the top model and reference model parameters.

Specify Some Parameters as Noninline

Suppose that you want to take advantage of the **Inline parameters** optimization while retaining the ability to tune some of your model parameters. You can do this by declaring some parameters as *noninline*, using either the “Model Parameter Configuration Dialog Box” or a

Simulink.Parameter object. In either case, you must use a workspace variable to specify the value of the parameter.

If you have a Simulink Coder license, when compiling a model with the inline parameters option on, Simulink checks to ensure that the data types of the workspace variables used to specify the model's nonlinear parameters are compatible with code generation. If not, Simulink halts the compilation and displays an error. For more information, see “Tunable Workspace Parameter Data Type Considerations”.

Note The documentation for the Simulink Coder refers to workspace variables used to specify the value of nonlinear parameters as *tunable workspace parameters*. In this context, the term *parameter* refers to a workspace variable used to specify a parameter as opposed to the parameter itself.

Using a Parameter Object to Specify a Parameter As Nonlinear

If you use a parameter object to specify a parameter's value (see “Use Parameter Objects to Specify Parameter Values” on page 24-7), you can also use the object to specify the parameter as nonlinear. To do this, set the parameter object's `CoderInfo.StorageClass` property to any value but 'Auto' (the default).

```
K=Simulink.Parameter;  
K.CoderInfo.StorageClass = 'SimulinkGlobal';
```

If you set the `CoderInfo.StorageClass` property to any value other than Auto, you should not include the parameter in the tunable parameters table in the **Model Parameter Configuration** dialog box.

Note Simulink halts model compilation and displays an error message if it detects a conflict between the properties of a parameter as specified by a parameter object and the properties of the parameter as specified in the **Model Parameter Configuration** dialog box.

Structure Parameters

In this section...

“About Structure Parameters” on page 24-17

“Define Structure Parameters” on page 24-18

“Referencing Structure Parameters” on page 24-18

“Structure Parameter Arguments” on page 24-19

“Tunable Structure Parameters” on page 24-20

“Parameter Structure Limitations” on page 24-21

About Structure Parameters

Separately defining all base workspace variables used in block parameter expressions can clutter the base workspace and result in very long lists of arguments to subsystems and referenced models. The technique provides no way to conveniently group related base workspace variables, or to configure generated code to reflect the variables' relationships.

To minimize the disadvantages of separately defining workspace variables used by block parameters, you can group numeric variables by specifying their names and values as the fields of a MATLAB structure in the base workspace. A MATLAB structure that Simulink uses in block parameter expressions is called a *structure parameter*. You can use structure parameters to:

- Simplify and modularize the base workspace by using multiple structures to group related variables and to prevent name conflicts
- Dereference the structure in block parameter expressions to provide values from structure fields rather than separate variables
- Pass all the fields in a structure to a subsystem or referenced model with a single argument.
- Improve generated code to use structures rather multiple separate variables

For information about creating and using MATLAB structures, see Structures in the MATLAB documentation. You can use all the techniques described

there to manipulate structure parameters. This section assumes that you know those techniques, and provides only information that is specific to Simulink.

For information on structure parameters in the context of generated code for a model, see “Structure Parameters and Generated Code”. For an example of how to convert a model that uses unstructured workspace variables to a model that uses structure parameters, see `sldemo_applyVarStruct`.

Define Structure Parameters

Defining a structure parameter is syntactically the same as defining any MATLAB structure, as described in Structures. Every field in a MATLAB structure that functions as a structure parameter must have a numeric data type, even if Simulink never uses the field. Different fields can have different numeric types.

In structure parameters, numeric types include enumerated types, by virtue of their underlying integers. The value of a structure parameter field, can be a real or complex scalar, vector, or multidimensional array. However, a structure that contains any multidimensional array cannot be tuned. See “Tunable Structure Parameters” on page 24-20.

MATLAB structures, including those used as structure parameters, can have substructures to any depth. Structures and substructures at any level behave identically, so the following instructions refer only to structures unless substructures are specifically the point.

Referencing Structure Parameters

You can use MATLAB syntax, as described in Structures, to dereference a structure parameter field anywhere in a block parameter expression that a MATLAB variable can appear. You cannot specify a structure name in a mathematical block parameter expression, because that would pass a structure rather than a number. For example, suppose you have defined the following parameter structure:

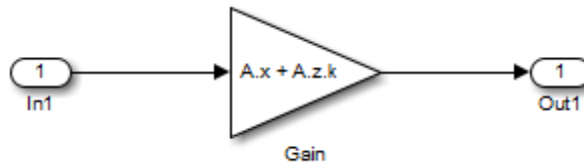
```
A          /* Root structure
|__x      /* Numeric field
|__y      /* Numeric field
```

```

|__z          /* Substructure
|__ m          /* Numeric field
|__ n          /* Numeric field
|__ k          /* Numeric field

```

Given this structure, you can specify an individual field, such as $A.x$, in a block parameter expression, thereby passing only x to the block. The effect is exactly the same as if x were a separate base workspace variable whose value was the same as the value of $A.x$. Similarly, you could reference $A.z.m$, $A.z.n$, etc. The next figure shows an example that uses a Gain block:



The Gain block's **Gain** parameter is the value of $A.x + A.z.k$, a numeric expression. You could not reference A or $A.z$ to provide a **Gain** parameter value, because neither resolves to a numeric value.

Structure Parameter Arguments

You can use a parameter structure field as a masked subsystem or model reference argument by referencing the field, as described in the previous section, in Subsystem block mask, or Model block. For example, suppose you have defined the parameter structure used in the previous example:

```

A          /* Root structure
|__x          /* Numeric field
|__y          /* Numeric field
|__z          /* Substructure
|__ m          /* Numeric field
|__ n          /* Numeric field
|__ k          /* Numeric field

```

You could then:

- 1 Use a whole structure parameter as a masked subsystem argument or a referenced model argument by referencing the structure's name
- 2 Dereference the structure as needed in the subsystem mask code, the subsystem itself, or the referenced model.

For example, you could pass `A`, providing access to everything in the root structure, or `A.z`, providing access only to that substructure. The dereferencing syntax for arguments is the same as in any other context, as described in Structures.

When you pass a structure parameter to a referenced model, the structure definitions must be identical in the parent model and the submodel, including any unused fields. See “Systems and Subsystems” on page 3-11, “Masking”, and “Using Model Arguments” on page 6-55 more information about passing and using arguments.

Tunable Structure Parameters

Declare a structure parameter to be tunable using one of the following techniques.

- Clear **Model Configuration Parameters > Optimization > Signals and Parameters > Inline parameters**. See “Inline parameters” for more information.
- Set **Inline parameters**, and then specify the parameter structure as tunable in the Model Parameter Configuration Dialog Box.
- Associate a `Simulink.Parameter` object with a structure parameter, and specify the object's storage class as anything other than `Auto`.

```
myStruct = Simulink.Parameter;  
myStruct.Value = struct('number',1,'units',24);  
myStruct.CoderInfo.StorageClass = 'ExportedGlobal';
```

A tunable structure parameter can contain a nontunable numeric field (like a multidimensional array) without affecting the tunability of the rest of the structure. You cannot define individual substructures or fields within a structure parameter to be tunable. Only the name of the root level of the

structure appears in the Model Configuration Parameter dialog box, and only the root can have a `Simulink.Parameter` object assigned to it.

For more information about tunability, see “Inline Parameters” on page 24-15 and “Tunable Parameters” on page 24-14. For simplicity, those sections mention only separately defined base workspace variables, but all of the information applies without change to tunable structure parameters.

Parameter Structure Limitations

- You cannot define individual substructures or fields within a structure parameter as tunable.
- Tunable structure parameters do not support context sensitivity.
- You cannot tune structure parameters that contain one or more fields that are arrays. For example, consider the structure parameter `pstruct`. This structure is tunable if its fields are defined as follows:

```
pstruct.gains1.k = 3  
pstruct.gains2.k = 4
```

However, this structure is not tunable if one or more of its fields are arrays.

```
pstruct.gains(1).k = 3  
pstruct.gains(2).k = 4
```


Working with Lookup Tables

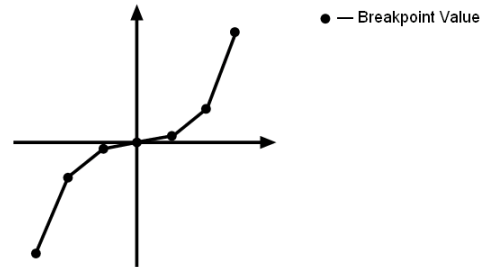
- “About Lookup Table Blocks” on page 25-2
- “Anatomy of a Lookup Table” on page 25-4
- “Lookup Tables Block Library” on page 25-5
- “Guidelines for Choosing a Lookup Table” on page 25-7
- “Enter Breakpoints and Table Data” on page 25-11
- “Characteristics of Lookup Table Data” on page 25-18
- “Methods for Estimating Missing Points” on page 25-23
- “Edit Existing LookupTables” on page 25-28
- “Create a Logarithm Lookup Table” on page 25-64
- “Prelookup and Interpolation Blocks” on page 25-67
- “Optimize Generated Code for Lookup Table Blocks” on page 25-68
- “Update Lookup Table Blocks to New Versions” on page 25-72
- “Lookup Table Glossary” on page 25-78

About Lookup Table Blocks

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.

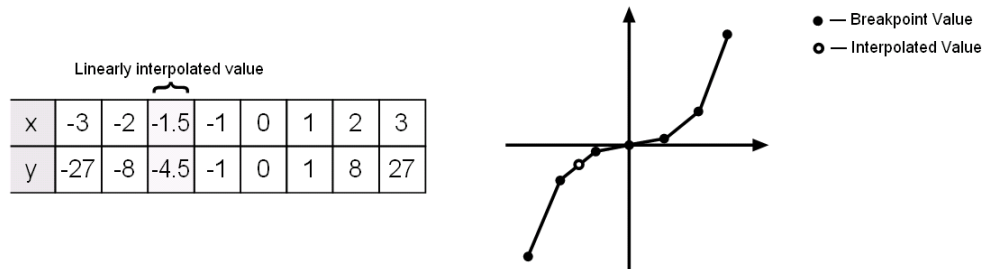
The following example illustrates a one-dimensional lookup table that approximates the function $y = x^3$. The lookup table defines its output (y) data discretely over the input (x) range $[-3, 3]$. The following table and graph illustrate the input/output relationship:

x	-3	-2	-1	0	1	2	3
y	-27	-8	-1	0	1	8	27

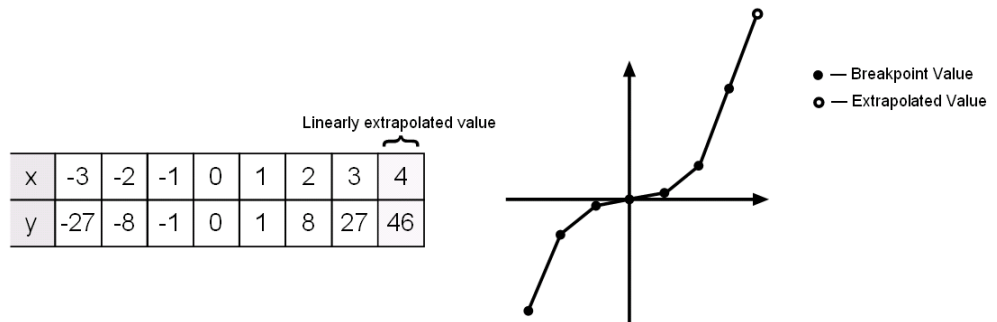


An input of -2 enables the table to look up and retrieve the corresponding output value (-8). Likewise, the lookup table outputs 27 in response to an input of 3.

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 data points (-2, -8) and (-1, -1) to estimate and return a value of -4.5.



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.



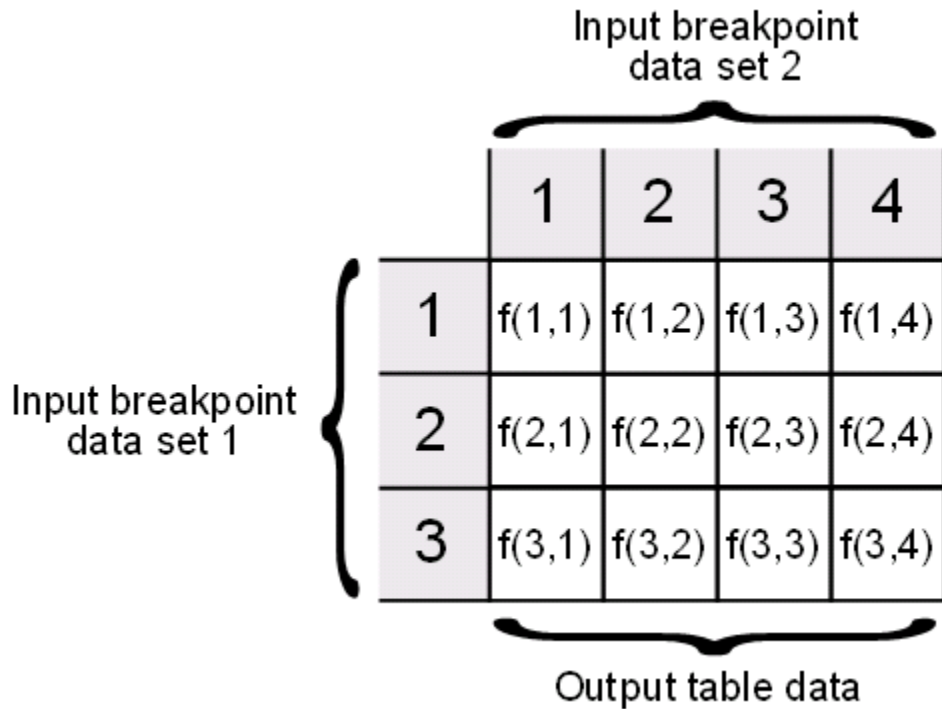
Since table lookups and simple estimations can be faster than mathematical function evaluations, using lookup table blocks might result in speed gains when simulating a model. Consider using lookup tables in lieu of mathematical function evaluations when:

- 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.

Anatomy of a Lookup Table

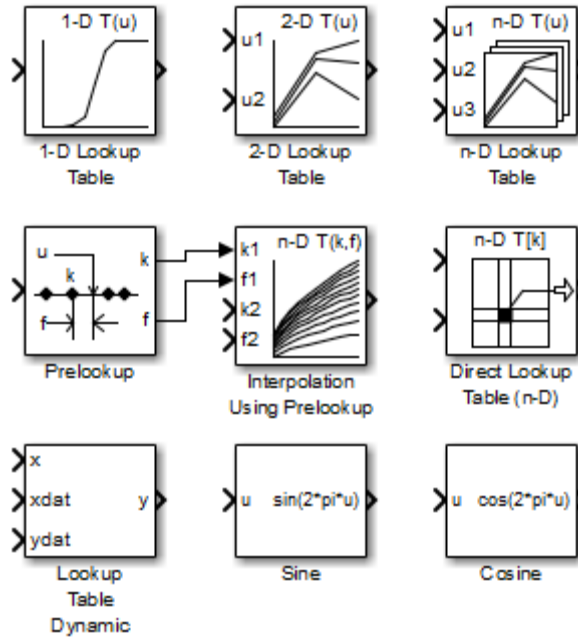
The following figure illustrates the anatomy of a two-dimensional lookup table. Vectors or *breakpoint data sets* and an array, referred to as *table data*, constitute the lookup table.



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.

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.

Guidelines for Choosing a Lookup Table

In this section...

- “Data Set Dimensionality” on page 25-7
- “Data Set Numeric and Data Types” on page 25-7
- “Data Accuracy and Smoothness” on page 25-8
- “Dynamics of Table Inputs” on page 25-8
- “Efficiency of Performance” on page 25-8
- “Summary of Lookup Table Block Features” on page 25-10

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 or cubic 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	Interp. Using Prelookup
Interpolation Methods							
Flat	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline	•	•		•			
Extrapolation Methods							
Clip	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline	•	•		•			
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	•	•		•	•	•	•

Enter Breakpoints and Table Data

In this section...

“Entering Data in a Block Parameter Dialog Box” on page 25-11

“Entering Data in the Lookup Table Editor” on page 25-13

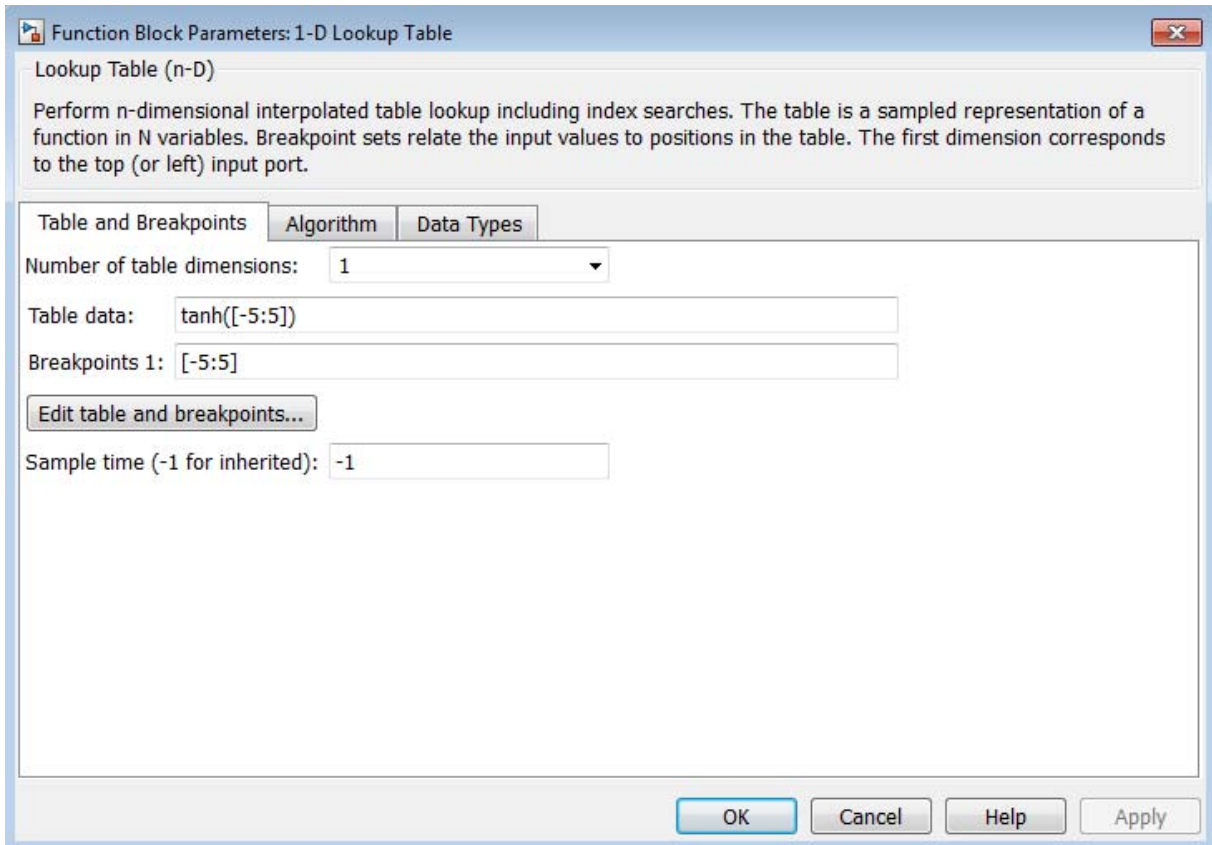
“Entering Data Using Inports of the Lookup Table Dynamic Block” on page 25-16

Entering Data in a Block Parameter Dialog Box

Use the following procedure to populate a 1-D Lookup Table block using the parameter dialog box. In this example, the lookup table 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 model window, double-click the 1-D Lookup Table block.

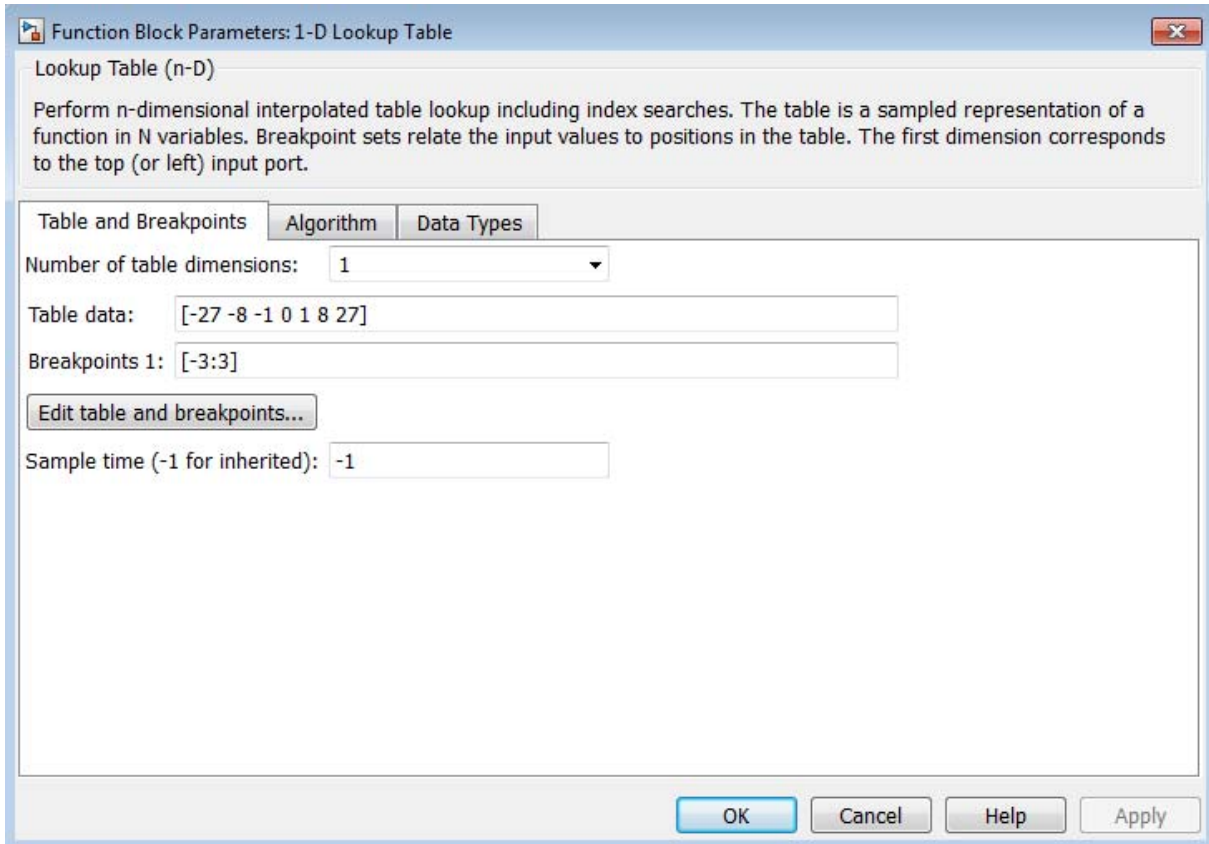
The block parameter dialog box appears.



The dialog box displays the default values for the block.

- 3 Enter the table dimensions, table data, and breakpoint data set in the specified fields of the dialog box:
 - In the **Number of table dimensions** field, enter 1.
 - In the **Table data** field, enter [-27 -8 -1 0 1 8 27].
 - In the **Breakpoints 1** field, enter [-3:3].
 - Click **Apply**.

The block dialog box looks something like this:



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

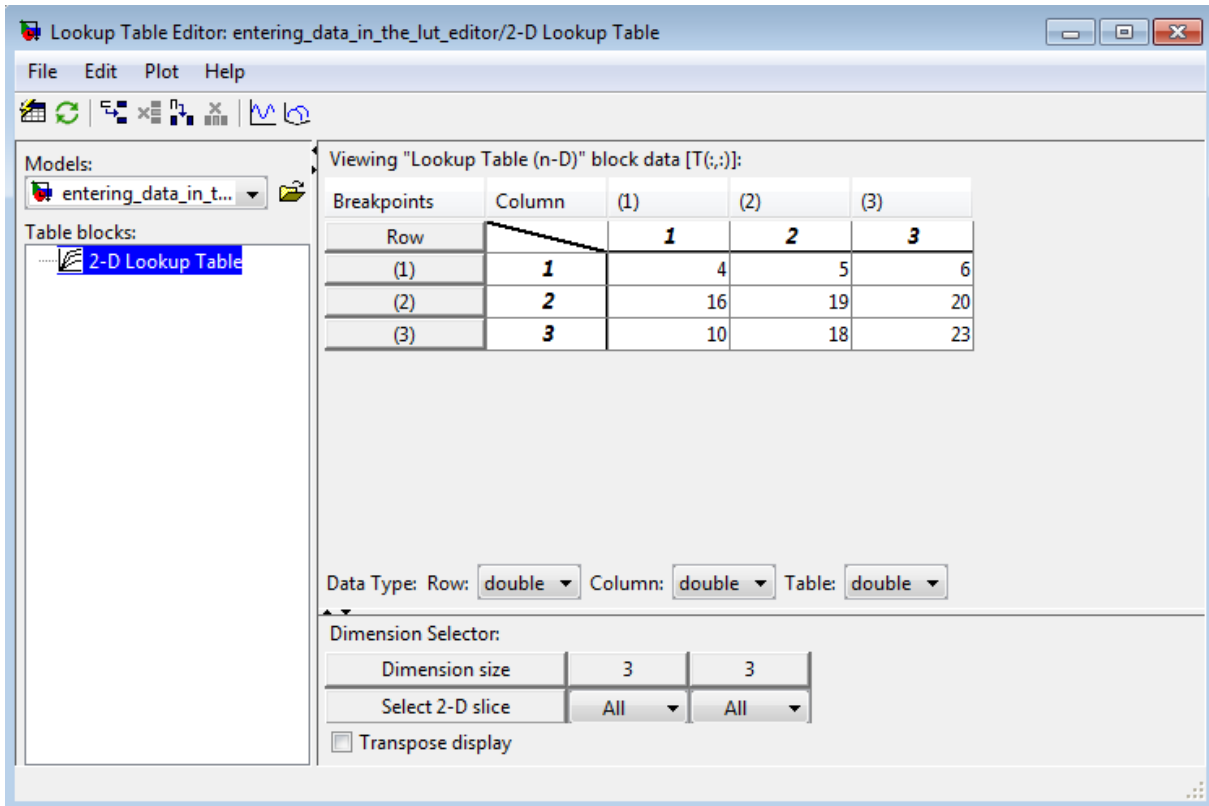
Entering Data in the Lookup Table Editor

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 Simulink **Edit** menu or by clicking **Edit table and breakpoints** on the dialog box of the 2-D Lookup Table block.

The Lookup Table Editor appears.

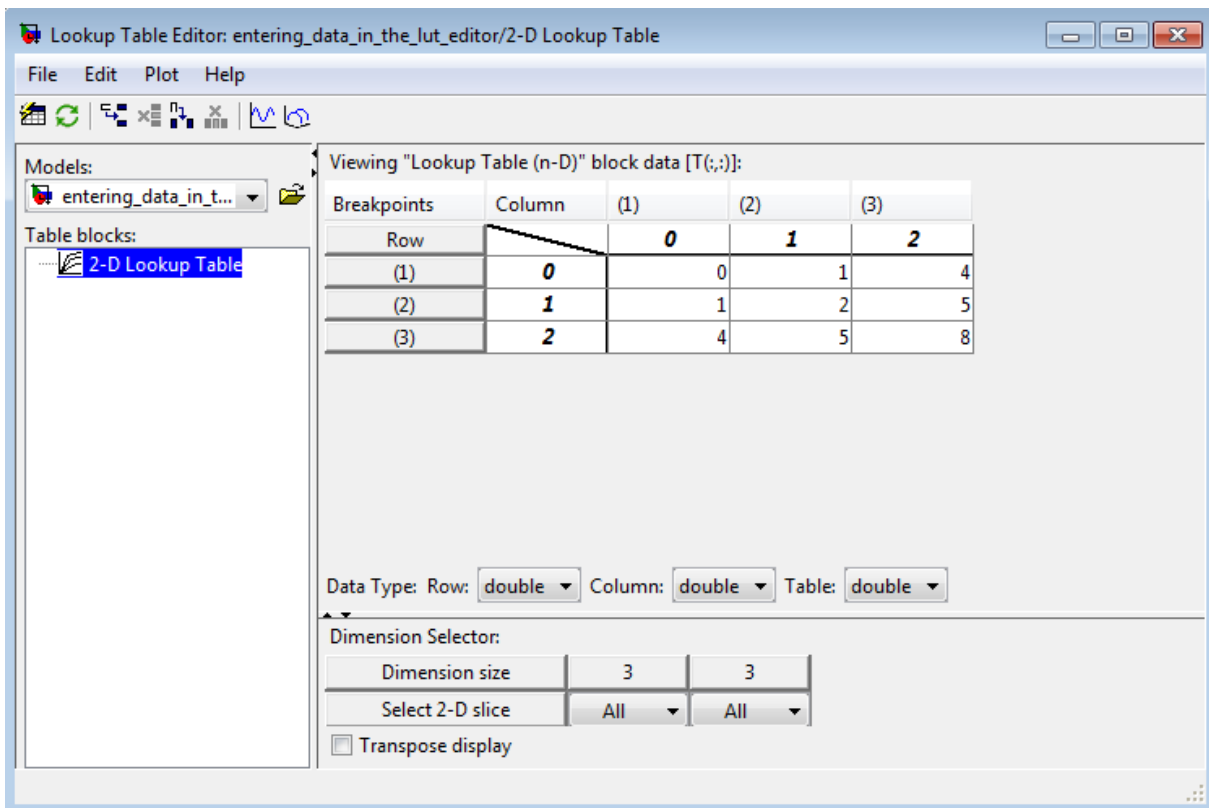


It displays the default data for the 2-D Lookup Table block.

- 3 Under **Viewing "Lookup Table (n-D)" block data**, enter the breakpoint data sets and table data in the appropriate cells. To change the default data, double-click a cell, enter the new value, and then press **Enter** or click outside the field to confirm the change:

- 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 should look 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

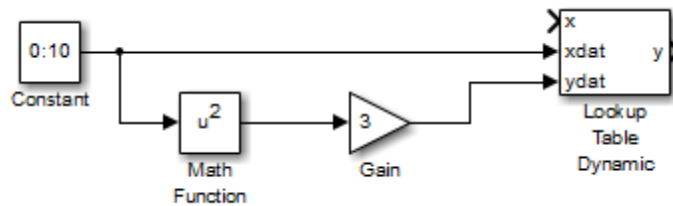
Use the following procedure to populate a Lookup Table Dynamic block using that block's inports. In this example, the lookup table approximates the function $y = 3x^2$ over the range $[0, 10]$.

- 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
Math Function	Function	square
Gain	Gain	3

- 4 Input the breakpoint data set to the Lookup Table Dynamic block by connecting the output of the Constant block to the inport 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 inport of the Lookup Table Dynamic block labeled **ydat**. This signal is the table data for y .

The model should look something like this:



Characteristics of Lookup Table Data

In this section...

“Sizes of Breakpoint Data Sets and Table Data” on page 25-18

“Monotonicity of Breakpoint Data Sets” on page 25-19

“Representation of Discontinuities in Lookup Tables” on page 25-20

“Formulation of Evenly Spaced Breakpoints” on page 25-22

Sizes of Breakpoint Data Sets and Table Data

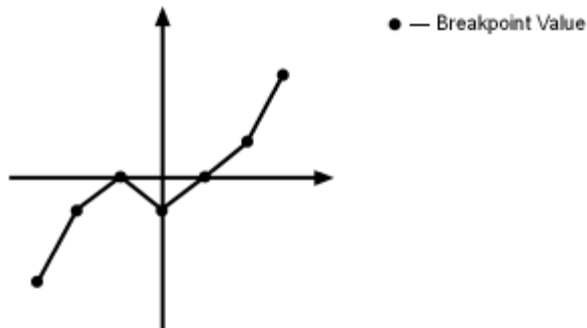
The following constraints apply to the sizes of breakpoint data sets and table data associated with lookup table blocks:

- 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 *monotonically increasing*, that is, each successive element is equal to or greater than its preceding element. For example, the vector

$$A = [0 \quad 0.5 \quad 1 \quad 1.9 \quad 2 \quad 2 \quad 2 \quad 2.1 \quad 3]$$

repeats the value 2 while all other elements are increasingly larger than their predecessors; hence, A is monotonically increasing.

For lookup tables with data types other than `double` or `single`, the search algorithm requires an additional constraint due to quantization effects. In such cases, the input breakpoint data sets must be *strictly monotonically increasing*, that is, each successive element must be greater than its preceding element. Consider the vector

$$B = [0 \quad 0.5 \quad 1 \quad 1.9 \quad 2 \quad 2.1 \quad 2.17 \quad 3]$$

in which each successive element is greater than its preceding element, making **B** strictly monotonically increasing.

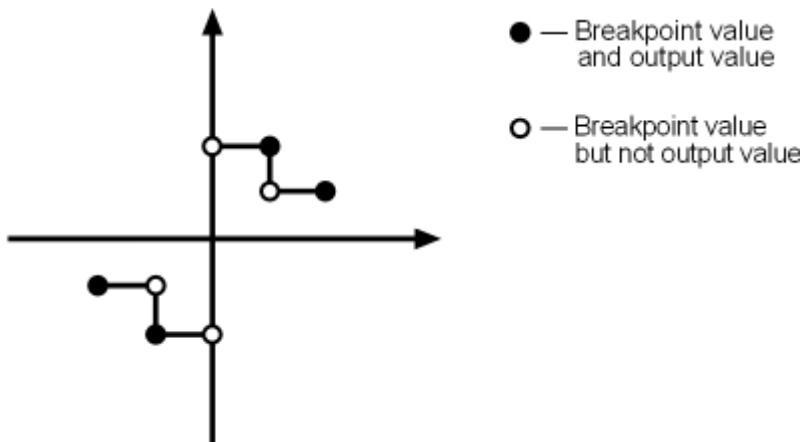
Note Although a breakpoint data set is strictly monotonic in double format, it might not be so after conversion to a fixed-point data type.

Representation of Discontinuities in Lookup Tables

You can represent discontinuities in lookup tables that have monotonically increasing breakpoint data sets. To create a discontinuity, repeat an input value in the breakpoint data set with different output values in the table data. For example, these vectors of input (x) and output (y) values associated with a 1-D lookup table create the step transitions depicted in the plot that follows.

Vector of input values: [-2 -1 -1 0 0 1 1 2]

Vector of output values: [-1 -1 -2 -2 2 2 1 1]



This example has discontinuities at $x = -1$, 0 , and $+1$.

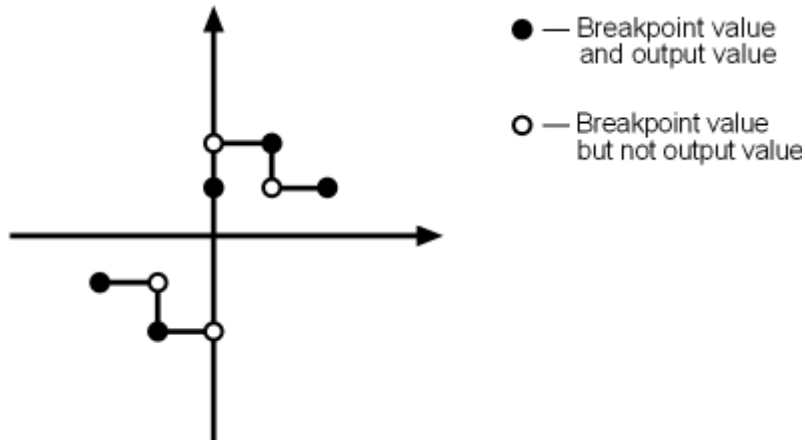
When there are two output values for a given input value, the block chooses the output according to these rules:

- If the input signal is less than zero, the block returns the output value associated with the last occurrence of the input value in the breakpoint data set. In this example, if the input is -1 , y is -2 , marked with a solid circle.
- If the input signal is greater than zero, the block returns the output value associated with the first occurrence of the input value in the breakpoint data set. In this example, if the input is 1 , y is 2 , marked with a solid circle.
- If the input signal is zero and there are two output values specified at the origin, the block returns the average of those output values. In this example, if the input is 0 , y is 0 , the average of the two output values -2 and 2 specified at $x = 0$.

When there are three points specified at the origin, the block generates the output associated with the middle point. The following example demonstrates this special rule.

Vector of input values: $[-2 -1 -1 0 0 1 1 2]$

Vector of output values: $[-1 -1 -2 -2 1 2 2 1 1]$



In this example, three points define the discontinuity at the origin. When the input is 0 , y is 1 , the value of the middle point.

You can apply this same method to create discontinuities in breakpoint data sets associated with multidimensional lookup tables.

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
<code>[first_value:spacing:last_value]</code>	<code>[10:10:200]</code>	The lookup table does <i>not</i> use double or single.
<code>first_value + spacing * [0:(last_value-first_value)/spacing]</code>	<code>1 + (0.02 * [0:450])</code>	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 25-10.

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” in the Fixed-Point Designer documentation
- “Identify questionable fixed-point operations” in the Simulink Coder documentation

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.

Methods for Estimating Missing Points

In this section...

“About Estimating Missing Points” on page 25-23

“Interpolation Methods” on page 25-23

“Extrapolation Methods” on page 25-24

“Rounding Methods” on page 25-25

“Example Output for Lookup Methods” on page 25-26

About Estimating Missing Points

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 25-25.
- **Linear** — Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input.
- **Cubic spline** — Fits a cubic spline to the adjacent breakpoints, and returns the point on that spline corresponding to the input.

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 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.
- **Linear** — Fits a line between 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 line corresponding to the input.
- **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.

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.

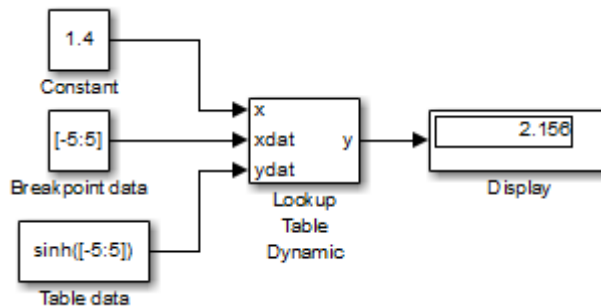
Rounding Methods

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 $\sinh(1.0)$.

Edit Existing Lookup Tables

In this section...

“When to Use the Lookup Table Editor” on page 25-28

“Layout of the Lookup Table Editor” on page 25-28

“Browsing Lookup Table Blocks” on page 25-29

“Editing Table Values” on page 25-30

“Working with Table Data of Standard Format” on page 25-32

“Working with Table Data of Nonstandard Format” on page 25-37

“Importing Data from an Excel Spreadsheet” on page 25-54

“Adding and Removing Rows and Columns in a Table” on page 25-56

“Displaying N-Dimensional Tables in the Editor” on page 25-57

“Plotting Lookup Tables” on page 25-58

“Editing Custom Lookup Table Blocks” on page 25-62

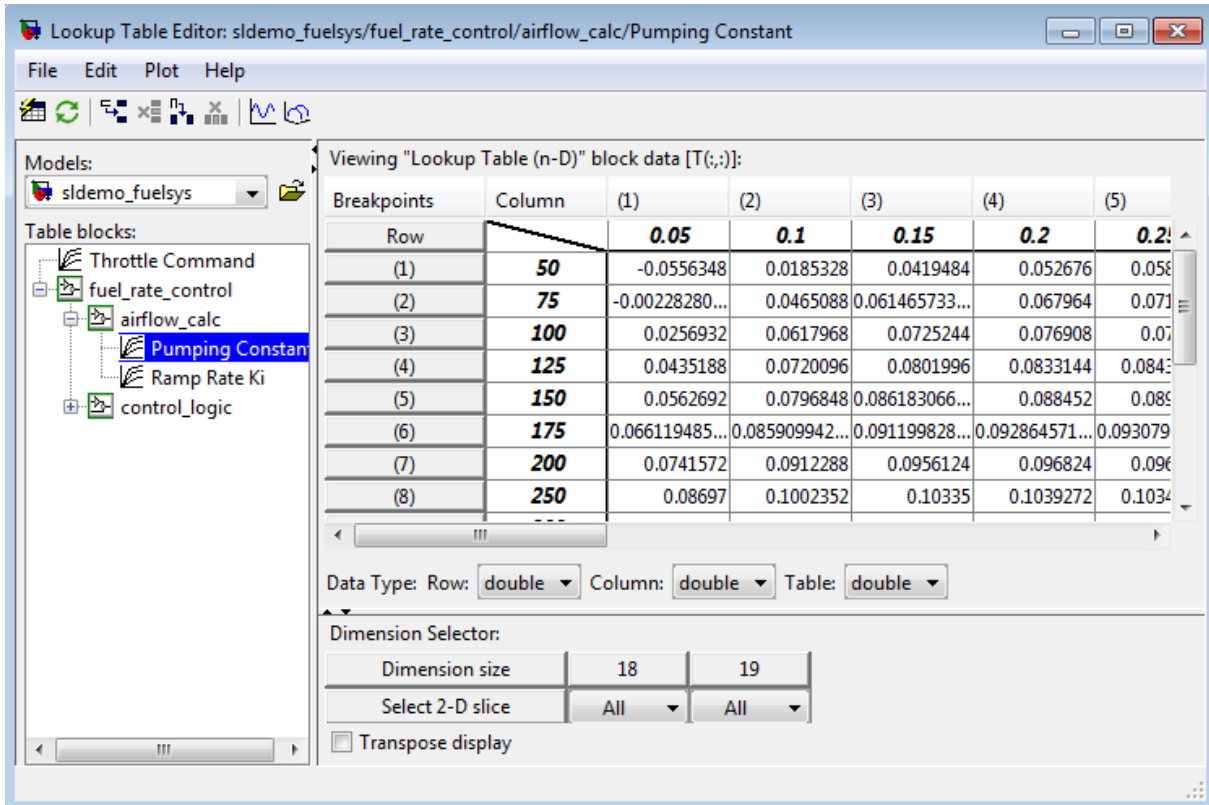
When to Use the Lookup Table Editor

Use the Lookup Table Editor to inspect and change the table elements of any lookup table block in a model, including custom blocks that you create using the Simulink Mask Editor (see “Editing Custom Lookup Table Blocks” on page 25-62). You can also use a block parameter dialog box to edit a table. However, you must open the subsystem containing the block first and then its parameter dialog box. With the Lookup Table Editor, you can skip these steps.

Tip 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.

Layout of the Lookup Table Editor

To open the editor, select **Lookup Table Editor** from the Simulink **Edit** menu. The editor appears.



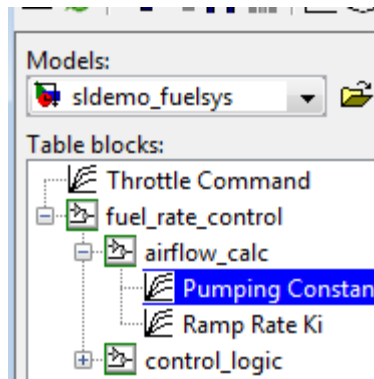
The editor contains two panes and a toolbar.

- Use the left pane to browse and select lookup table blocks in any open model (see “Browsing Lookup Table Blocks” on page 25-29).
- Use the right pane to edit the lookup table of the selected block (see “Editing Table Values” on page 25-30).
- Use the toolbar for one-click access to frequently-used commands in the editor. Each toolbar button has a tooltip that explains its function.

Browsing Lookup Table Blocks

The **Models** list in the upper-left corner of the Lookup Table Editor lists the names of all models open in the current MATLAB session. To browse

lookup table blocks for any open models, select the model name from the list. A tree-structured view of lookup table blocks for the selected model appears in the **Table blocks** field beneath the **Models** list.



The tree view initially lists all lookup table blocks that reside at the model root level. It also displays any subsystems that contain lookup table blocks. Clicking the expand button (+) to the left of the subsystem name expands the tree to show lookup table blocks in that subsystem. The expanded view also shows any subsystems in the expanded subsystem. You can continue expanding subsystem nodes to display lookup table blocks at any level in the model hierarchy.

Clicking any lookup table block in the tree view displays the lookup table for that block in the right pane, so that you can edit the table (see “Editing Table Values” on page 25-30).

Note If you want to browse the lookup table blocks in a model that is not currently open, you can tell the Lookup Table Editor to open the model. To do this, select **File > Open Model** in the editor.

Editing Table Values

In the Viewing “Lookup Table (n-D)” block data table view of the Lookup Table Editor, you can edit the lookup table of the block currently selected in the adjacent tree view.

Viewing "Lookup Table (n-D)" block data [T(:,:)]:

Breakpoints	Column	(1)	(2)	(3)
Row		0.05	0.1	0.15
(1)	50	-0.0556348	0.0185328	0.0419484
(2)	75	-0.00228280...	0.0465088	0.061465733..
(3)	100	0.0256932	0.0617968	0.0725244
(4)	125	0.0435188	0.0720096	0.0801996
(5)	150	0.0562692	0.0796848	0.086183066..
(6)	175	0.066119485...	0.085909942...	0.091199828..

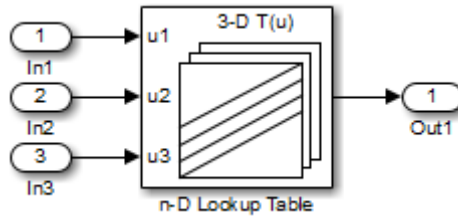
The table view displays the entire table if it is one- or two-dimensional or a two-dimensional slice of the table if the table has more than two dimensions (see “Displaying N-Dimensional Tables in the Editor” on page 25-57). To change any value that appears, double-click the value. The Lookup Table Editor replaces the value with an edit field containing the value. Edit the value and then press **Enter** or click outside the field to confirm the change.

In the **Data Type** below the table, you can specify the data type by row or column, or for the entire table. By default, the data type is **double**. To change the data type, select the pop-up index list for the table element for which you want to change the data type.

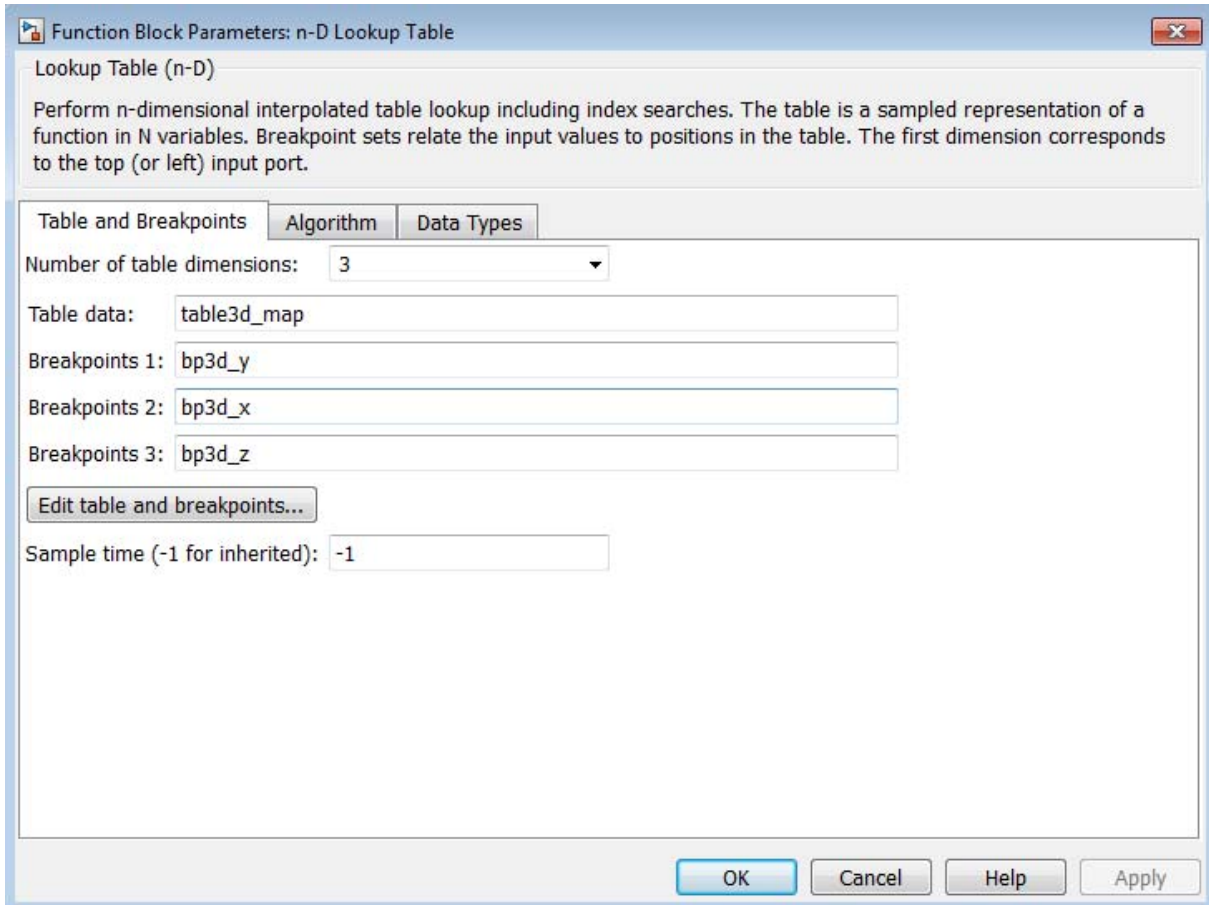
The Lookup Table Editor records your changes by maintaining a copy of the table. To update the copy that the lookup table block maintains, select **File > Update Block Data** in the Lookup Table Editor. To restore the Lookup Table Editor’s copy to the values stored in the block, select **File > Reload Block Data**.

Working with Table Data of Standard Format

Suppose that you specify a 3-D lookup table in your n-D Lookup Table block.



You use workspace variables to define the breakpoint and table data:



The table data uses the default Simulink format:

```
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
```

The breakpoint sets have the following values:

```
bp3d_y =
```

```
    400   6400
```

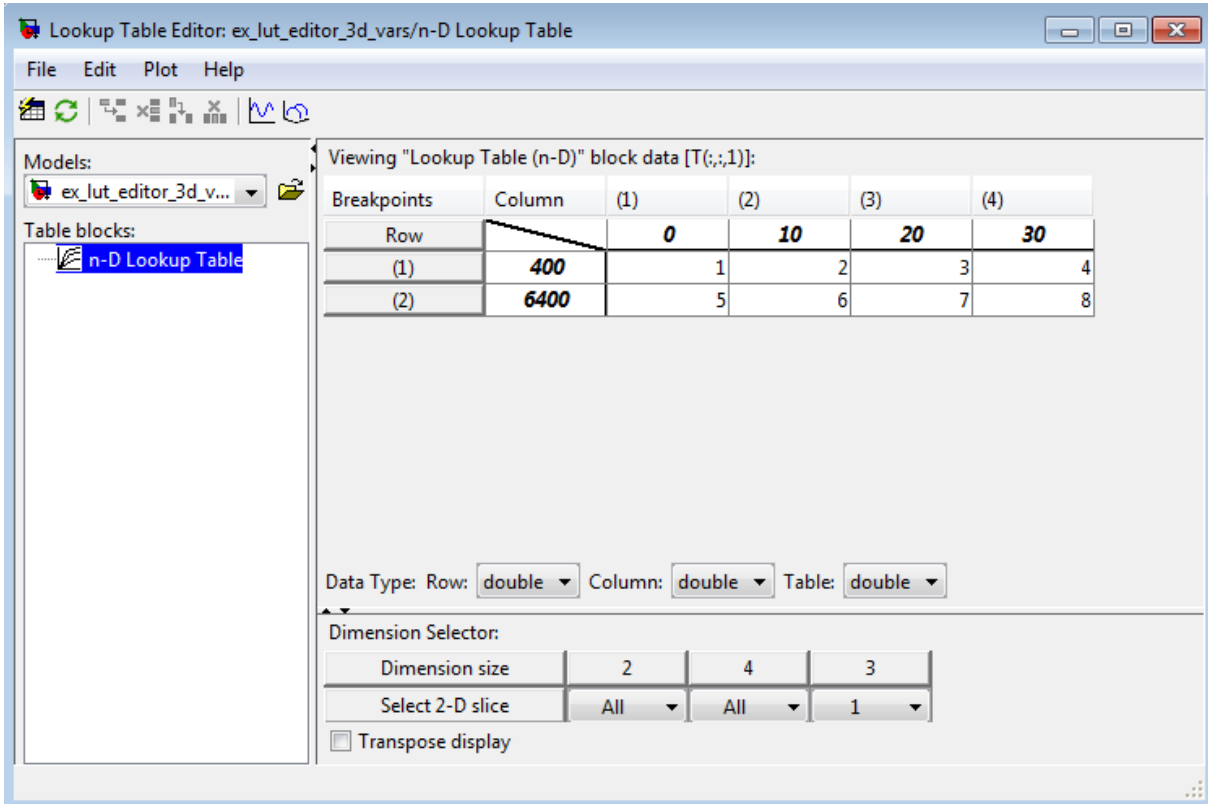
```
bp3d_x =
```

```
    0    10    20    30
```

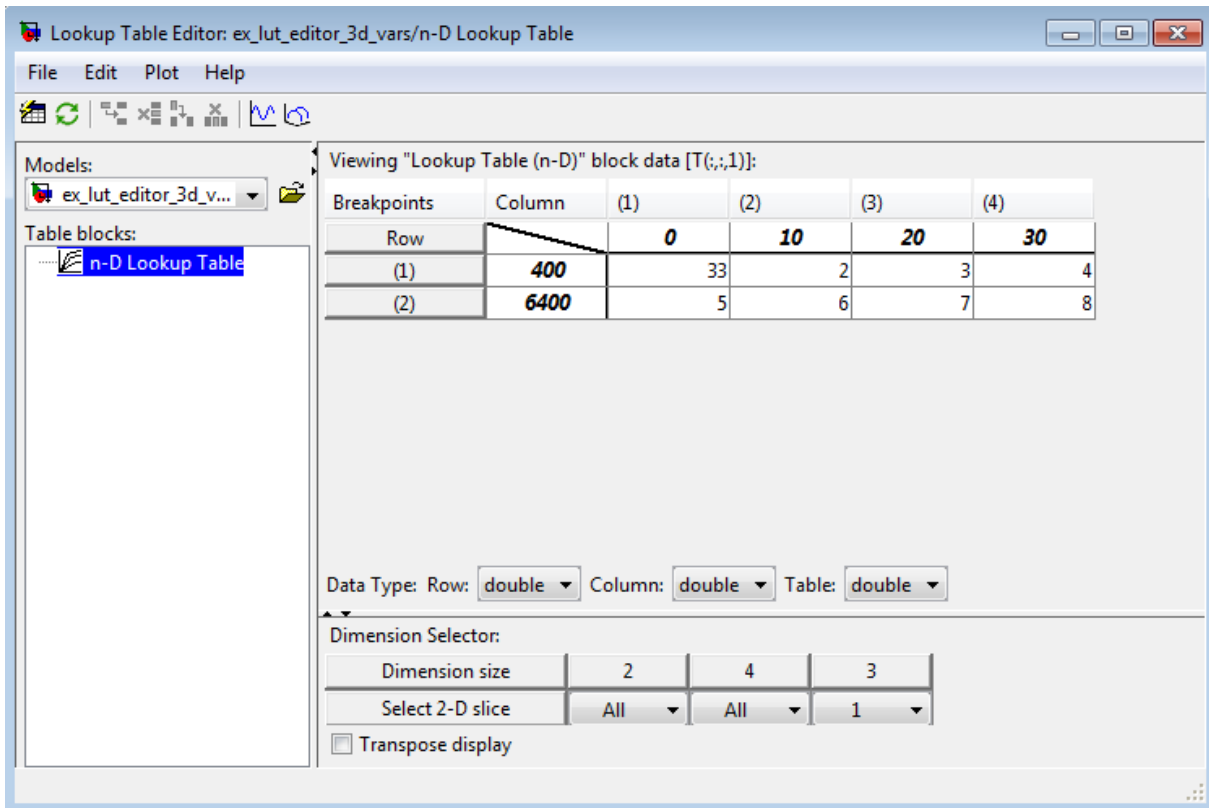
```
bp3d_z =
```

```
    0    10    20
```

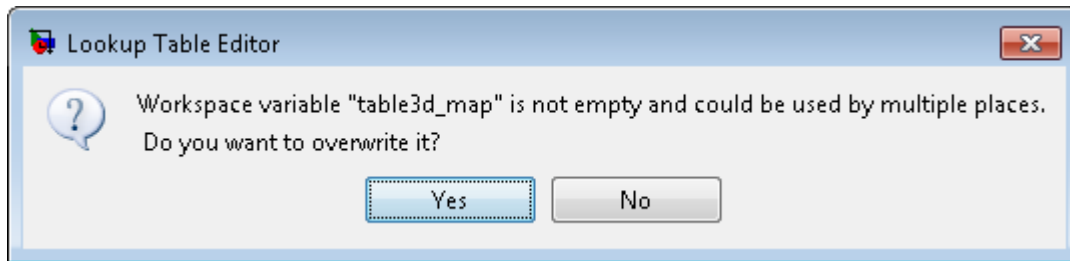
When you click **Edit table and breakpoints** to open the Lookup Table Editor, you see:



Suppose that you change a value in the Lookup Table Editor as follows:

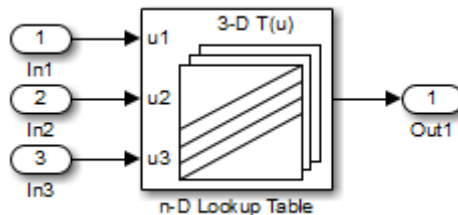


When you select **File > Update Block Data**, you can propagate the change to `table3d_map`, the workspace variable that contains the table data for the n-D Lookup Table block. To propagate the change, click **Yes** in the pop-up window that asks if you want to overwrite the workspace variable:

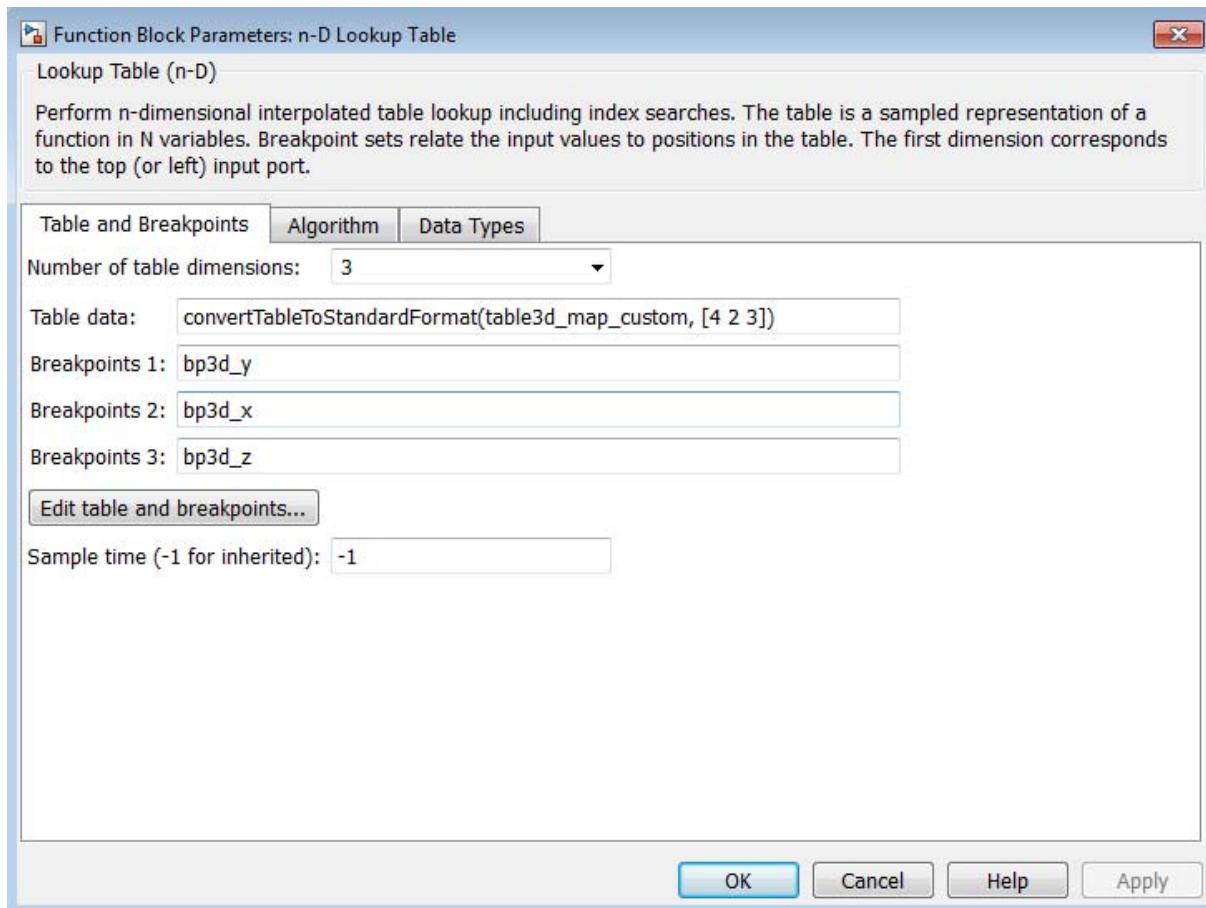


Working with Table Data of Nonstandard Format

Suppose that you specify a 3-D lookup table in your n-D Lookup Table block.



You use workspace variables to define the breakpoint and table data:



The table data uses a nonstandard format for representing 3-D table data that differs from the default Simulink format:

```
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
```

The expression in the **Table data** field converts the table data from a nonstandard format to the default Simulink format.

The breakpoint sets have the following values:

```
bp3d_y =
```

```
    400   6400
```

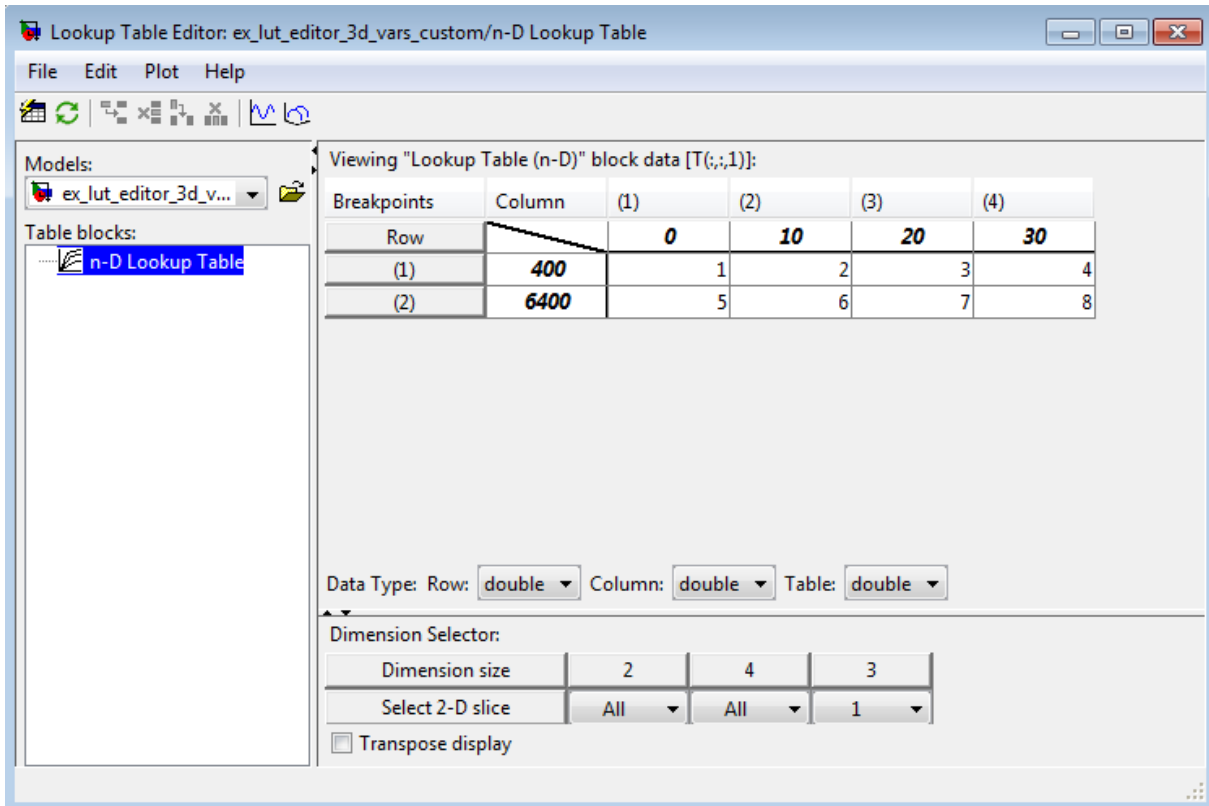
```
bp3d_x =
```

```
    0    10    20    30
```

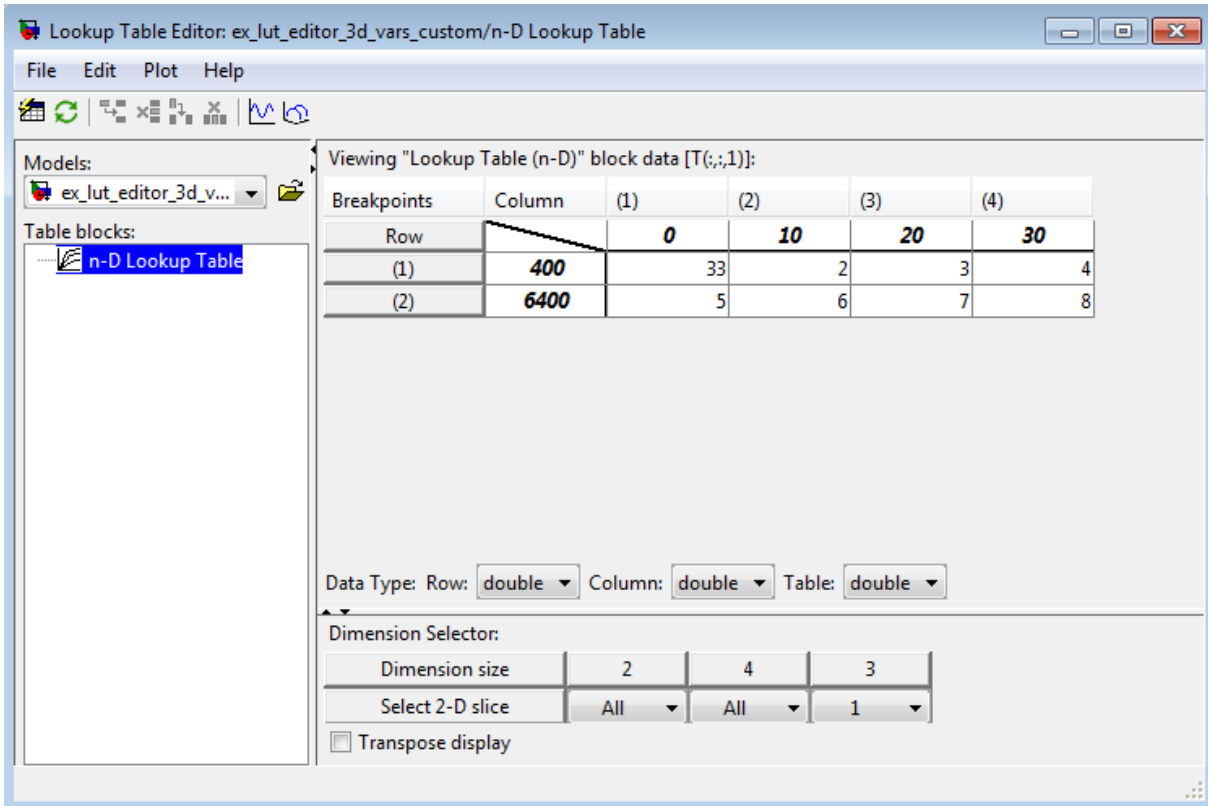
```
bp3d_z =
```

```
    0    10    20
```

When you click **Edit table and breakpoints** to open the Lookup Table Editor, you see:



Suppose that you change a value in the Lookup Table Editor as follows:



When you select **File > Update Block Data**, 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 “Example of Propagating a Change for Table Data of Nonstandard Format” on page 25-46.

How to Propagate Changes in the Lookup Table Editor to Workspace Variables of Nonstandard Format

Step	Task	Reference
1	Register a customization function for the Lookup Table Editor.	“How to Register a Customization Function for the Lookup Table Editor” on page 25-42
2	Select the appropriate responses when prompted in the Lookup Table Editor.	“How to Respond to Prompts in the Lookup Table Editor” on page 25-46

How to Register a Customization Function for the Lookup Table Editor

To register a customization function for the Lookup Table Editor:

- 1 In MATLAB, create a file named `sl_customization.m` to register your customization:

```
function sl_customization(cm)

cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle{end+1} = ...
@myConversionInfoFunction;

end
```

The string `myConversionInfoFunction` represents a function name of your choice.

- 2 In your `sl_customization.m` file, define a function that returns a `blkInfo` object.

```
function blkInfo = myConversionInfoFunction(blk, tableStr)

blkInfo.allowTableConvertLocal = false;
blkInfo.tableWorkspaceVarName = '';
blkInfo.tableConvertFcnHandle = [];
```

```

% Directions for converting table data of block type 1
if strcmp(get_param(blk, 'BlockType'), 'block_type_1')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkSpaceVarName = 'variable_name_1';
end

% Directions for converting table data of block type 2
if strcmp(get_param(blk, 'BlockType'), 'block_type_2')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkSpaceVarName = 'variable_name_2';
end

.....

% Directions for converting table data of block type N
if strcmp(get_param(blk, 'BlockType'), 'block_type_N')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkSpaceVarName = 'variable_name_N';
end

if blkInfo.allowTableConvertLocal
    blkInfo.tableConvertFcnHandle = @myConvertTableDataFcn;
end

end

```

For each type of lookup table block that contains nonstandard table data, specify the following three properties:

- `allowTableConvertLocal` — tells the Lookup Table Editor if table data conversion is allowed 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

Tip You can specify more restrictive if conditions in this function. For example, to specify that table data conversion occur for a block with a given name, you can use:

```
if strcmp(strrep(blk,sprintf('\n'), ' '), 'full_block_path')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkspaceVarName = 'variable_name';
end
```

When the `allowTableConvertLocal` property is true:

- The table data for that block can be converted to the nonstandard format of the workspace variable whose name matches `tableWorkspaceVarName`.
- The function that converts table data corresponds to the handle that `tableConvertFcnHandle` specifies.

You can use any alphanumeric name for the conversion function. The string `myConvertTableDataFcn` represents a function name of your choice.

- 3** In your `sl_customization.m` file, define the function that converts table data from the Lookup Table Editor format to the nonstandard format of your workspace variable.
- 4** Verify that your `sl_customization.m` file is complete.
- 5** Put `sl_customization.m` on the MATLAB search path.

Note You can have multiple files with the name `sl_customization.m` on the search path.

- 6** At the MATLAB prompt, enter the following command to refresh all Simulink customizations.

```
sl_refresh_customizations
```

What Happens When Multiple Customization Functions Exist

If you have multiple `sl_customization.m` files on the search path, the following happens:

- At the start of a MATLAB session, Simulink loads each customization file on the path and executes the function at the top. 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...	Changes in the Lookup Table Editor...
Is empty	Cannot propagate to workspace variables with nonstandard format
Contains one or more function handles	Can propagate to workspace variables with nonstandard format, depending on the value of the <code>allowTableConvertLocal</code> property

- The `allowTableConvertLocal` property controls whether or not table data for a block is converted from the standard format in the Lookup Table Editor to a nonstandard format in a workspace variable.

If a customization function specifies this block property to be...	Then table data is...
true	Converted to the nonstandard format in the workspace variable
false	Not converted to the nonstandard format in the workspace variable
true, but another customization function specifies the property to be false	Not converted and the Lookup Table Editor issues an error

How to Respond to Prompts in the Lookup Table Editor

When you select **File > Update Block Data** in the Lookup Table Editor, several prompts appear. Click **Yes** when asked if you want to:

- Overwrite workspace variables for breakpoint data
- Overwrite workspace variables for table data

Example of Propagating a Change for Table Data of Nonstandard Format

Suppose that you want to propagate a change from the Lookup Table Editor to the workspace variable `table3d_map_custom`, which uses a nonstandard format. Follow these steps:

- 1 Create a file named `s1_customization.m` with the following contents:

```
function s1_customization(cm)

cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle{end+1} = ...
@myGetTableConvertInfoFcn;

end
```

In this file:

- The `s1_customization` function takes a single argument `cm`, which 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. In this example, the function name is `myGetTableConvertInfoFcn`.

- 2 In your `s1_customization.m` file, define the `myGetTableConvertInfoFcn` function.

```
function blkInfo = myGetTableConvertInfoFcn(blk, tableStr)
```

```

blkInfo.allowTableConvertLocal = false;
blkInfo.tableWorkspaceVarName = '';
blkInfo.tableConvertFcnHandle = [];

% Convert table data for n-D Lookup Table blocks
if strcmp(get_param(blk, 'BlockType'), 'Lookup_n-D')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkspaceVarName = 'table3d_map_custom';
end

% Convert table data for Interpolation Using Prelookup blocks
if strcmp(get_param(blk, 'BlockType'), 'Interpolation_n-D')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkspaceVarName = 'table4d_map_custom';
end

% Convert table data for Direct Lookup Table (n-D) blocks
if strcmp(get_param(blk, 'BlockType'), 'LookupNDDirect')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkspaceVarName = 'table4d_map_custom';
end

if blkInfo.allowTableConvertLocal
    blkInfo.tableConvertFcnHandle = @myConvertTableFcn;
end

end

```

The `myGetTableConvertInfoFcn` function returns the `blkInfo` object, which contains three fields:

- `allowTableConvertLocal` — tells the Lookup Table Editor if table data conversion is allowed 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 the `allowTableConvertLocal` property is true:

- The table data for that block is converted to the nonstandard format of the workspace variable whose name matches `tableWorkspaceVarName`.
- The function that converts table data corresponds to the handle that `tableConvertFcnHandle` specifies.

You can use any alphanumeric name for the conversion function. In this example, the function name is `myConvertTableFcn`.

- 3 In your `sl_customization.m` file, define the `myConvertTableFcn` function, which includes a separate function call to `convertTable_3D`. An example of an implementation is shown below:

```
% This function converts LUT block table from Simulink format to
% nonstandard format used in workspace variable (for 1D to 4D table)
function cMap = myConvertTableFcn(data)

    mapDim = size(data);

    numDim = length(mapDim);

    if numDim < 3
        cMap = data;
    elseif numDim == 3
        cMap = convertTable_3D(data);
    else % numDim == 4 , currently only supports up to 4D
        for i = 1:mapDim(numDim)
            dataSub = data(:,:,i);
            iStart = (i-1)*mapDim(1)*mapDim(3) + 1;
            iEnd = i*mapDim(1)*mapDim(3);
            cMap(iStart:iEnd,:) = convertTable_3D(dataSub);
        end
    end
end

% This function converts LUT block table from Simulink format to
% nonstandard format used in workspace variable (for 3D table)
function cMap = convertTable_3D(data)

    % figure out 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);

for i = 1:mapDim(3)
    rowBegin = (i-1)*mapDim(1)+1;
    rowEnd = i*mapDim(1);
    cMap(rowBegin:rowEnd, :) = data(:, :, i);
end
end

```

4 Verify that your `sl_customization.m` file is complete:

```

function sl_customization(cm)

cm.LookupTableEditorCustomizer.getTableConvertToCustomInfoFcnHandle{end+1} = ...
@myGetTableConvertInfoFcn;

% This function converts LUT block table from Simulink format to
% nonstandard format used in workspace variable (for 1D to 4D table)
function cMap = myConvertTableFcn(data)

    mapDim = size(data);

    numDim = length(mapDim);

    if numDim < 3
        cMap = data;
    elseif numDim == 3
        cMap = convertTable_3D(data);
    else % numDim == 4 , currently only supports up to 4D
        for i = 1:mapDim(numDim)
            dataSub = data(:, :, :, i);
            iStart = (i-1)*mapDim(1)*mapDim(3) + 1;
            iEnd = i*mapDim(1)*mapDim(3);
            cMap(iStart:iEnd, :) = convertTable_3D(dataSub);
        end
    end
end

```

```

end

% This function converts LUT block table from Simulink format to
% nonstandard format used in workspace variable (for 3D table)
function cMap = convertTable_3D(data)

    % figure out the row and column number of the 3D table data
    mapDim = size(data);

    numRows = mapDim(2);
    numCol = mapDim(1)*mapDim(3);

    cMap = zeros(numRow, numCol);

    for i = 1:mapDim(3)
        rowBegin = (i-1)*mapDim(1)+1;
        rowEnd = i*mapDim(1);
        cMap(rowBegin:rowEnd, :) = data(:, :, i);
    end
end

function blkInfo = myGetTableConvertInfoFcn(blk, tableStr)

    blkInfo.allowTableConvertLocal = false;
    blkInfo.tableWorkSpaceVarName = '';
    blkInfo.tableConvertFcnHandle = [];

    % Convert table data for n-D Lookup Table blocks
    if strcmp(get_param(blk, 'BlockType'), 'Lookup_n-D')
        blkInfo.allowTableConvertLocal = true;
        blkInfo.tableWorkSpaceVarName = 'table3d_map_custom';
    end

    % Convert table data for Interpolation Using Prelookup blocks
    if strcmp(get_param(blk, 'BlockType'), 'Interpolation_n-D')
        blkInfo.allowTableConvertLocal = true;
        blkInfo.tableWorkSpaceVarName = 'table4d_map_custom';
    end

    % Convert table data for Direct Lookup Table (n-D) blocks

```

```
if strcmp(get_param(blk, 'BlockType'), 'LookupNDDirect')
    blkInfo.allowTableConvertLocal = true;
    blkInfo.tableWorkspaceVarName = 'table4d_map_custom';
end

if blkInfo.allowTableConvertLocal
    blkInfo.tableConvertFcnHandle = @myConvertTableFcn;
end

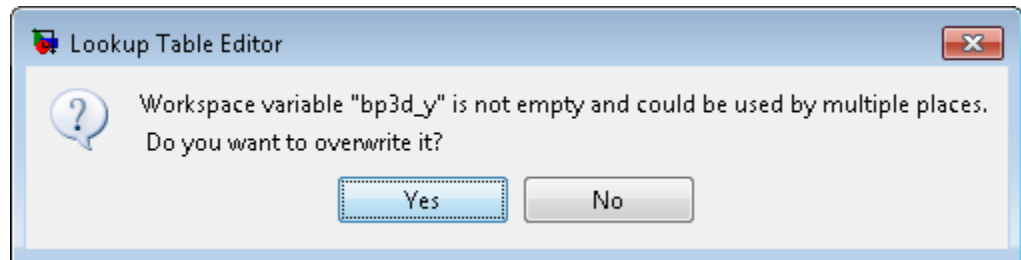
end

end
```

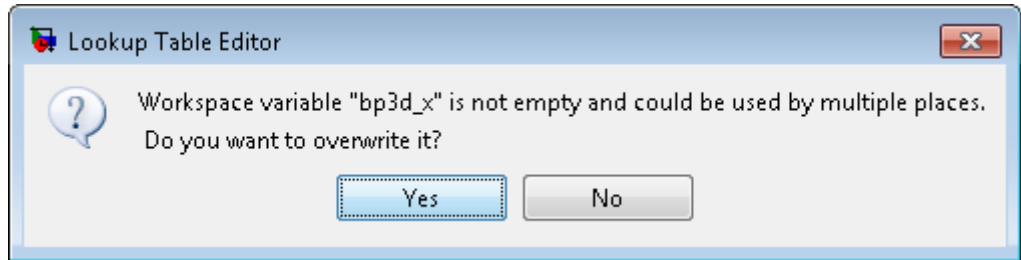
- 5 Put `sl_customization.m` on the MATLAB search path.
- 6 At the MATLAB prompt, enter the following command to refresh all Simulink customizations.

```
sl_refresh_customizations
```

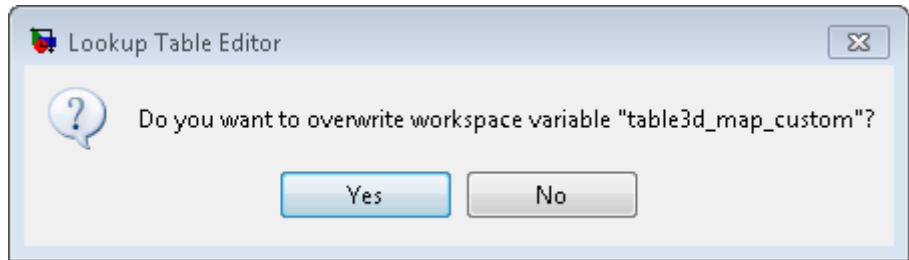
- 7 In the Lookup Table Editor, select **File > Update Block Data**. Several pop-up windows appear:
 - a Click **Yes** to overwrite the workspace variable `bp3d_y`.



- b Click **Yes** to overwrite the workspace variable `bp3d_x`.



- c Click **Yes** to overwrite the workspace variable `table3d_map_custom`.

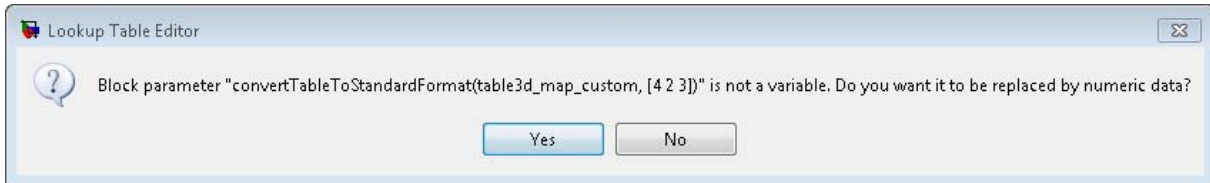


- 8 If you check the value of `table3d_map_custom` in the base workspace, you now see:

```
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 that you make to the table data in the Lookup Table Editor propagates to the workspace variable.

Note If you click **No** to the question of whether to overwrite the workspace variable `table3d_map_custom`, you get another question:



If you click...	Then...
<p>Yes</p>	<p>The block dialog box replaces the expression in the Table data field with numeric data.</p>
<p>No</p>	<p>Your Lookup Table Editor changes for the table data do not appear in the block dialog box.</p>

Importing Data from an Excel Spreadsheet

If you have table data in a Microsoft Excel spreadsheet, you can import the data into the Lookup Table Editor.

How to Import Data from the Spreadsheet

To import data into the Lookup Table Editor:

- 1 Save the Excel file in a folder on the MATLAB path.
- 2 Open the model that contains a lookup table block.
- 3 Save the model with a different name.
- 4 Open the Model Properties dialog box, for example, by selecting **File > Model Properties > Model Properties**.
- 5 Click the **Callbacks** tab and enter code for the model post-load function.

For a 2-D lookup table, the commands look something like this:

```
%%% BEGIN NEW CODE %%%  
% Import the data from Excel for a lookup table  
data = xlsread('name_of_spreadsheet','sheet_name');  
% 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);  
%%% END NEW CODE %%%
```

For more information about using `xlsread`, see the MATLAB documentation.

- 6 Save the model again.

Example of Importing Data from an Excel Spreadsheet

Suppose that you have table data in an Excel spreadsheet named `airflow_tables.xls` and you want to import the data into the Lookup Table Editor.

- 1 Save `airflow_tables.xls` in a folder on the MATLAB path. Enter the following command at the MATLAB prompt and replace *destination* with a location that is on the search path:

```
copyfile((docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'airflow_tables.xls'))), ...
'destination', 'f')
```

For example, `C:\MyFiles\airflow_tables.xls` might be a valid value for *destination*.

- 2 Open the `sldemo_fuelsys` model, which contains a lookup table block named Pumping Constant.
- 3 Save the model with a different name, such as `my_fuelsys`.
- 4 Open the Model Properties dialog box, for example, by selecting **File > Model Properties**.
- 5 Click the **Callbacks** tab and add the following commands under the existing code for the post-load function:

```
%%% BEGIN NEW CODE %%%
% Import the data from Excel for a lookup table
data = xlsread('airflow_tables', 'Pumping Constant');
% Row indices for Pumping Constant data
SpeedVect = data(2:end,1)';
% Column indices for Pumping Constant data
PressVect = data(1,2:end);
% Output values for Pumping Constant data
PumpCon = data(2:end,2:end);
%%% END NEW CODE %%%
```

- 6 Save the model again.





Whenever you open `my_fuelsys`, Simulink imports the table data for the Pumping Constant block from the Excel spreadsheet.

Adding and Removing Rows and Columns in a Table

In the Lookup Table Editor, you can add and remove rows or columns of a table in the following cases:

- Tables that are one- or two-dimensional
- Tables defined only by breakpoints (that are inherently one-dimensional)

In those cases, follow these instructions to add or remove columns of a table in the Lookup Table Editor.

To perform this action:	Use one of these methods:
Add a row or column to a table that appears in the table view	<ul style="list-style-type: none"> • Select Edit > Add Row or Edit > Add Column in the editor. • Click the Add Row button  or the Add Column button  in the toolbar.
Remove a row or column from the table that appears in the table view	<ul style="list-style-type: none"> • Highlight the row or column to remove and then select Edit > Remove Row(s) or Edit > Remove Column(s) in the editor. • Highlight the row or column to remove and then click the Remove Row button  or the Remove Column button  in the toolbar.

The menu items and toolbar buttons for adding and removing rows and columns are not available for any other cases. To add or remove a row or column for a table with more than two dimensions, you must use the block parameter dialog box.

Displaying N-Dimensional Tables in the Editor

If the lookup table of the block currently selected in the Lookup Table Editor's tree view has more than two dimensions, the table view displays a two-dimensional slice of the lookup table.

Viewing "Lookup Table (n-D)" block data [T(:,:),1]:

Breakpoints	Column	(1)	(2)	(3)	(4)	(5)
Row		1	2	3	4	5
(1)	0	1	11	21	31	
(2)	10	2	12	22	32	
(3)	20	3	13	23	33	
(4)	30	4	14	24	34	
(5)	40	5	15	25	35	
(6)	50	6	16	26	36	
(7)	60	7	17	27	37	
(8)	70	8	18	28	38	

Data Type: Row: Column: Table:

Dimension Selector:

Dimension size	10	6	5
Select 2-D slice	All	All	1

Transpose display

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.

Dimension Selector:			
Dimension size	10	6	5
Select 2-D slice	All ▼	All ▼	1 ▼

Transpose display

To transpose the table display, select the **Transpose display** check box.

Plotting Lookup Tables

To display a linear or mesh plot of the table or table slice in the Lookup Table Editor, select **Plot > Linear** or **Plot > Mesh**.

Suppose that you have the following lookup table open:

The screenshot shows the 'Lookup Table Editor' window for the 'Ramp Rate Ki' block. The window title is 'Lookup Table Editor: sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki'. The interface includes a menu bar (File, Edit, Plot, Help), a toolbar with various icons, and a left-hand pane showing the 'Models' tree with 'Ramp Rate Ki' selected. The main area displays the table data for the 'Ramp Rate Ki' block.

Viewing "Lookup Table (n-D)" block data [T(:,,:)]:

Breakpoints	Column	(1)	(2)	(3)	(4)	(5)
Row		0	0.2	0.4	0.6	0.8
(1)	100	0.012	0.024	0.036000000...	0.048	0.060
(2)	200	0.024	0.048	0.072000000...	0.096	0.120
(3)	300	0.036000000...	0.072000000...	0.108	0.144000000...	0.180
(4)	400	0.048	0.096	0.144000000...	0.192	0.240
(5)	500	0.060	0.120	0.180	0.240	0.300
(6)	600	0.072000000...	0.144000000...	0.216	0.288000000...	0.360

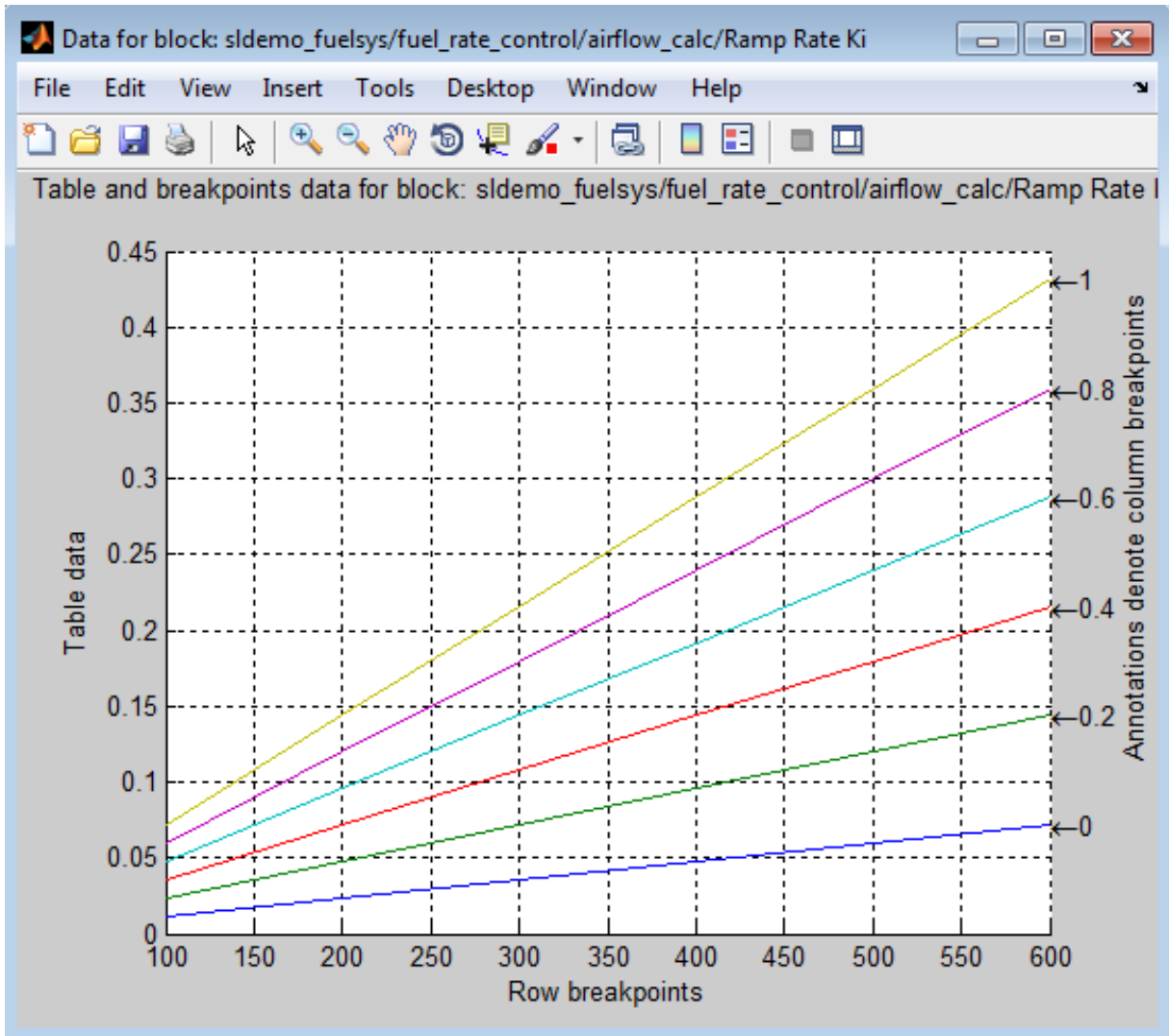
Data Type: Row: Column: Table:

Dimension Selector:

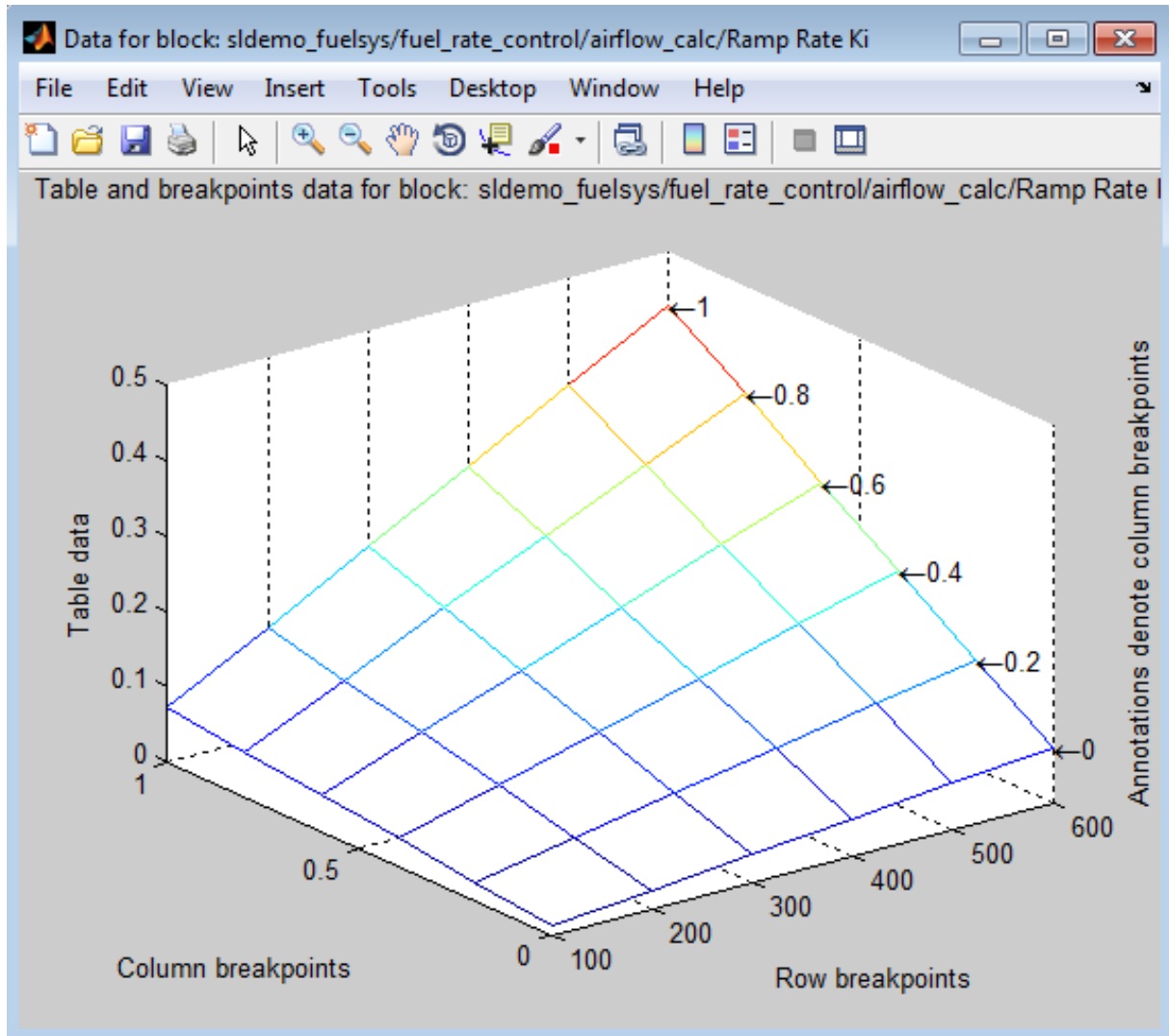
Dimension size	6	6
Select 2-D slice	All	All

Transpose display

A linear plot of the table looks something like this:



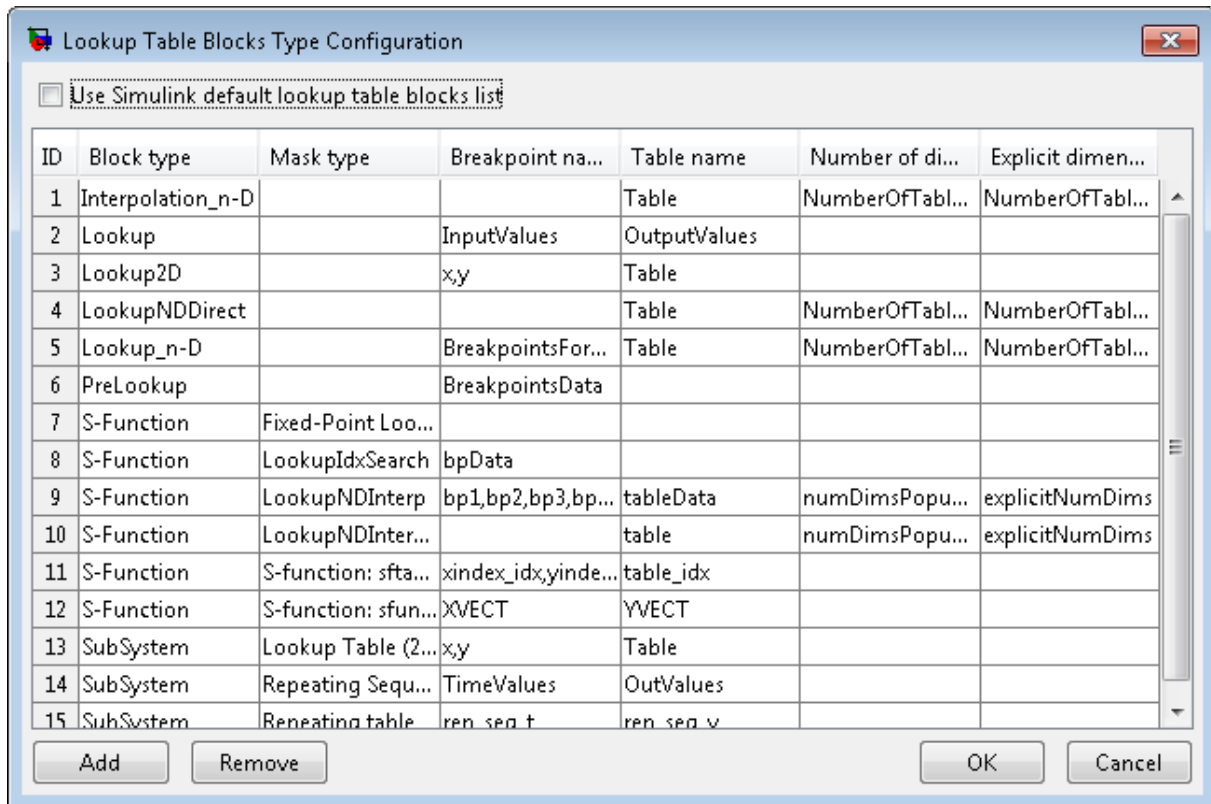
A mesh plot of the table looks something like this:



Editing Custom Lookup Table Blocks

You can use the Lookup Table Editor to edit custom lookup table blocks that you or others have created. To do this, you must first configure the Lookup Table Editor to recognize the custom lookup table blocks in your model. After you configure the Lookup Table Editor to recognize custom blocks, you can edit them as if they were standard blocks.

To configure the Lookup Table Editor to recognize custom blocks, select **File > Configure**. The Lookup Table Blocks Type Configuration dialog box appears.



By default, 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.

Adding a Custom Lookup Table Block Type

To add a custom block to the list of recognized types:

- 1 Click **Add** on the dialog box.

A new row appears at the bottom of the block type table.

- 2 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 <code>BlockType</code> parameter.
Mask type	Mask type of the custom block. The mask type is the value of the block's <code>MaskType</code> 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.

- 3 Click **OK**.

Removing Custom Lookup Table Block Types

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. Then click **Remove**.

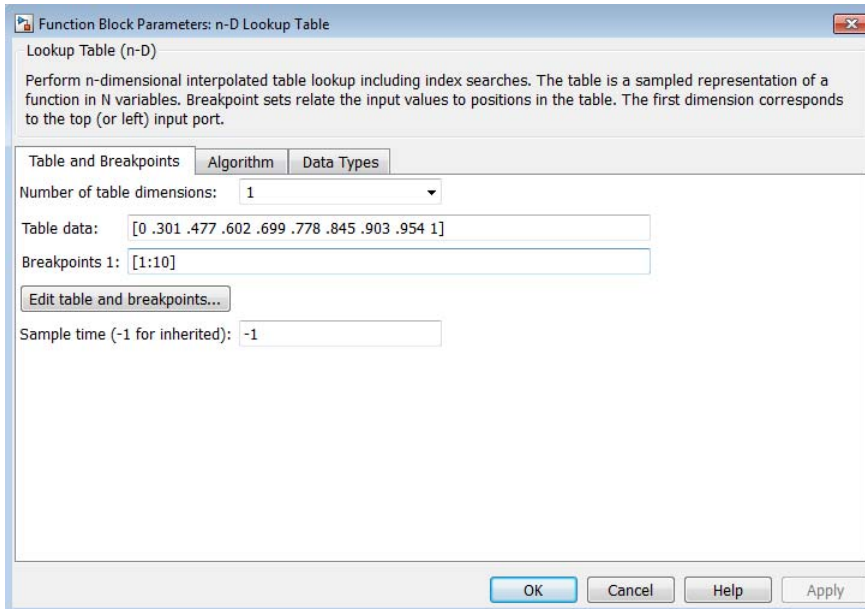
To remove all custom lookup table block types, select the **Use Simulink default lookup table blocks list** check box at the top of the dialog box.

Create a Logarithm Lookup Table

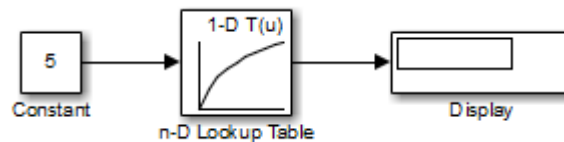
Suppose you want to approximate the common logarithm (base 10) over the input range [1, 10] without performing an expensive computation. You can perform this approximation using a lookup table block as described in the following procedure.

- 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.

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	To open the model, type <code>sldemo_bpcheck</code> at the command prompt.
Set breakpoint and table data types explicitly	Lowers memory required to store: <ul style="list-style-type: none"> • Breakpoint data that uses a smaller type than the input signal • Table data that uses a smaller type than the output signal 	To open the model, type <code>sldemo_interp_memory</code> at the command prompt.
	Provides easier sharing of: <ul style="list-style-type: none"> • Breakpoint data among Prelookup blocks • Table data among Interpolation Using Prelookup blocks 	To open the model, type <code>fxpdemo_lookup_shared_param</code> at the command prompt.
	Enables reuse of utility functions in the generated code	To open the model, type <code>fxpdemo_prelookup_utilfcn</code> at the command prompt.
Set the data type for intermediate results explicitly	Enables use of higher precision for internal computations than for table data or output data	To open the model, type <code>fxpdemo_interp_precision</code> at the command prompt.

Optimize Generated Code for Lookup Table Blocks

In this section...

“Remove Code That Checks for Out-of-Range Inputs” on page 25-68

“Optimize Breakpoint Spacing in Lookup Tables” on page 25-70

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 Verification and Validation > 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 “Consult the Model Advisor” on page 4-88 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 “Consult the Model Advisor” on page 4-88 in the Simulink documentation.

Update Lookup Table Blocks to New Versions

In this section...

“Comparison of Blocks with Current Versions” on page 25-72

“Compatibility of Models with Older Versions of Lookup Table Blocks” on page 25-73

“How to Update Your Model” on page 25-74

“What to Expect from the Model Advisor Check” on page 25-75

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	<ul style="list-style-type: none"> Block renamed as 1-D Lookup Table Icon changed 	<ul style="list-style-type: none"> Default integer rounding mode changed from Floor to Simplest Support for the following features: <ul style="list-style-type: none"> 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)	<ul style="list-style-type: none"> • Block renamed as 2-D Lookup Table • Icon changed 	<ul style="list-style-type: none"> • Default integer rounding mode changed from Floor to Simplest • Support for the following features: <ul style="list-style-type: none"> ▪ 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)	<ul style="list-style-type: none"> • Block renamed as n-D Lookup Table • Icon changed 	<ul style="list-style-type: none"> • 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 this Model Advisor check: By Product > Simulink > Check model, local libraries, and referenced models for known upgrade issues requiring compile time information.	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: <ul style="list-style-type: none"> 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 Model Advisor check.	Ensure that block replacement works for the entire model.
4	Run the <code>slookup</code> function on your model.	Perform block replacement with the 1-D Lookup Table and 2-D Lookup Table blocks.

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 Model Advisor, see “Consult the Model Advisor” on page 4-88 in the Simulink documentation.

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
<p>The Lookup Method is Use Input Nearest or Use Input Above. The replacement block does not support these lookup methods.</p>	<p>Change the lookup method to one of the following options:</p> <ul style="list-style-type: none"> • Interpolation - Extrapolation • Interpolation - Use End Values • Use Input Below 	<p>The Lookup Method changes to Interpolation - Use End Values.</p> <p>In the replacement block, this setting corresponds to:</p> <ul style="list-style-type: none"> • Interpolation set to Linear • Extrapolation set to Clip
<p>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.</p>	<p>Change the extrapolation method or the port data types of the block.</p>	<p>You also see a message that explains possible numerical differences.</p>
<p>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.</p>	<p>None</p>	<p>You see a message that explains possible numerical differences.</p>

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.

Lookup Table Glossary

The following table summarizes the terminology used to describe lookup tables in the Simulink user interface and documentation.

Term	Meaning
breakpoint	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.
breakpoint data set	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.
extrapolation	A process for estimating values that lie beyond the range of known data points.
interpolation	A process for estimating values that lie between known data points.
lookup table	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.
monotonically increasing	The elements of a set are ordered such that each successive element is greater than or equal to its preceding element.

Term	Meaning
rounding	A process for approximating a value by altering its digits according to a known rule.
strictly monotonically increasing	The elements of a set are ordered such that each successive element is greater than its preceding element.
table data	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.

Working with Block Masks

- “Block Masks” on page 26-3
- “How Mask Parameters Work” on page 26-5
- “Mask Code Execution” on page 26-7
- “Mask Terminology” on page 26-11
- “Mask a Block” on page 26-12
- “Draw Mask Icon” on page 26-15
- “Create Mask Documentation” on page 26-19
- “Initialize Mask” on page 26-21
- “Best Practices for Masking” on page 26-24
- “Considerations for Masking Model Blocks” on page 26-25
- “Masks on Blocks in User Libraries” on page 26-27
- “Promote Underlying Block Parameters to Mask” on page 26-29
- “Create Custom Interface for Simulink Blocks” on page 26-32
- “Rules for Promoting Parameters” on page 26-35
- “Mask Blocks and Promote Parameters” on page 26-37
- “Operate on Existing Masks” on page 26-41
- “Calculate Values Used Under the Mask” on page 26-44
- “Control Masks Programmatically” on page 26-47
- “Create Dynamic Mask Dialog Boxes” on page 26-50
- “Create Dynamic Masked Subsystems” on page 26-54
- “Debug Masks That Use MATLAB Code” on page 26-60

- “Masking Linked Blocks” on page 26-61
- “Mask a Linked Block” on page 26-64

Block Masks

In this section...
“What Are Masks?” on page 26-3
“When to Use Masks?” on page 26-3

What Are Masks?

Masks are custom interfaces you can apply to Simulink blocks. A mask hides the user interface of the block, and instead displays a custom dialog control for specific parameters of the masked block.

When you mask a block, you change only the interface to the block, not its underlying characteristics. Masking a non atomic subsystem does not make it act as an atomic subsystem, and masking a virtual block does not convert it to a non virtual block.

Note You cannot save a mask separately from the block that it masks. Also you cannot create an isolated mask definition and apply it to more than one block.

The mask icon and mask dialog box are analogous to the block icon and block dialog box, respectively.

You can use the mask parameters to control the mask display and change underlying subsystem content dynamically (add or delete blocks and set the parameters of those blocks). However, you can change subsystem content dynamically only if the subsystem is part of a library.

When to Use Masks?

Masks are useful for customizing block interfaces, encapsulating logic, and providing restricted access to data.

Masks are comparable to subsystems, in that they both simplify the graphical appearance of a model. However, subsystems do not offer an interface for users to interact with underlying block parameters.

Consider masking a Simulink block when you want to:

- display a meaningful dynamic icon that reflects values within a block
- define customized parameters whose names reflect the purpose of a block
- provide a dialog box that lets users access only select parameters of the underlying blocks
- provide users customized documentation that is specific to the masked block

If you use a mask only to represent the contents of a subsystem, consider using content preview instead. For that usage, content preview has these advantages, compared to using a mask:

- Automatically update changes in the subsystem (for a masked block, you need to manually update the mask image that represents the content of the item).
- Eliminates the setup tasks for icons for masked blocks.

Masked blocks do not support content preview.

For details, see “Preview Content of Hierarchical Items” on page 1-28.

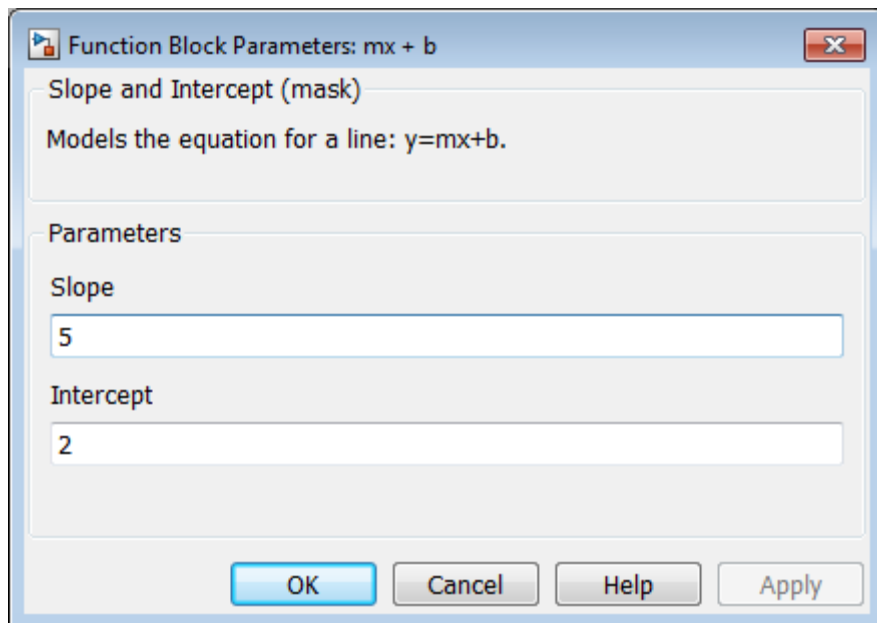
How Mask Parameters Work

A masked block is a custom interface to underlying blocks that are governed by block parameters. Mask parameters are the links to underlying block parameters.

Mask parameters are defined in the mask workspace, while block parameters are defined in the model or base workspace.

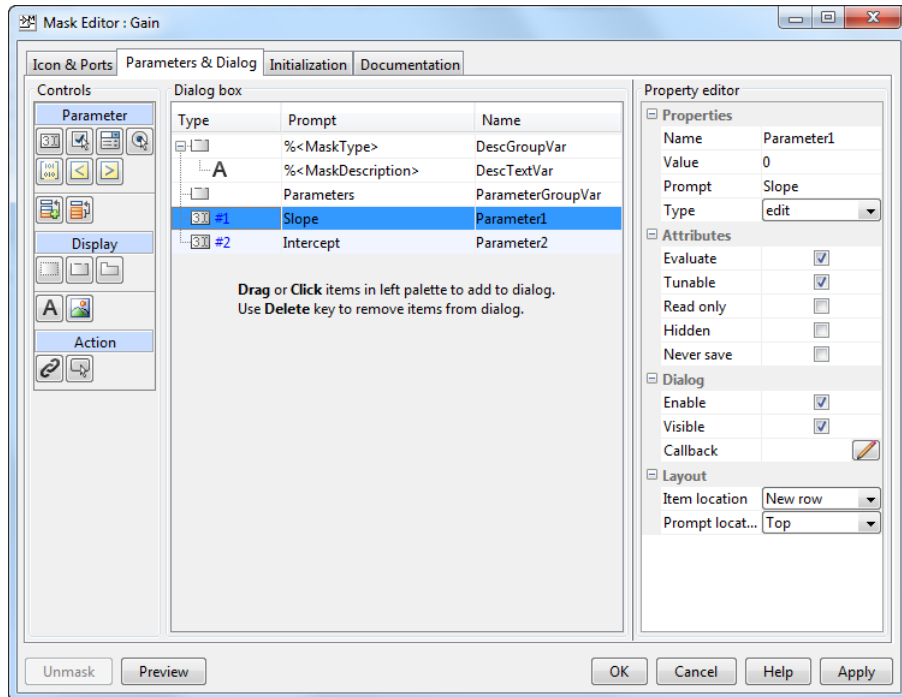
You can provide access to one or more underlying block parameters by defining the corresponding number of mask parameters. Mask parameters appear in the **Mask Parameters** dialog box as fields that can be edited. Simulink applies the value of a mask parameter to the value of the corresponding block parameter during simulation.

Consider the **Mask Parameters** dialog box of the model `masking_example`.



This dialog contains fields for mask parameters **Slope** and **Intercept**, both defined in the Mask Editor.

Slope corresponds to mask workspace variable m , and **Intercept**, to mask workspace variable b . Moreover, names m and b correspond to the **Gain** and **Constant value** parameters of the underlying blocks.



In the mask dialog box, when you set **Slope** and **Intercept** to 5 and 2, respectively, Simulink assigns these values to m and b .

Before simulation begins, Simulink searches the workspace hierarchy, looking in the mask workspace first, for values to resolve the **Gain** parameter m and **Constant value** parameter b . Since variables m and b are defined in the mask workspace, Simulink applies their values to the block parameters.

Mask Code Execution

In this section...
“Mask Code Placement” on page 26-7
“Drawing Command Execution” on page 26-7
“Initialization Command Execution” on page 26-8
“Callback Code Execution” on page 26-9

Mask Code Placement

You can use MATLAB code to initialize a mask as well as to draw mask icons. Since the location of code affects model performance, place your code to reflect the functionality you need.

Purpose	Placement 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

Drawing Command Execution

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 in the following cases:

- The drawing commands are dependent on mask parameters and the values of these mask parameters change.
- The block appearance is altered due to rotation or other changes.

Note If you place MATLAB code for drawing mask icons in the **Initialization** pane, model performance is affected, because Simulink redraws the icon each time the masked block is evaluated in the model.

Initialization Command Execution

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.

- The masked subsystem belongs to a library and has the **Allow library block to modify its contents** parameter enabled.

Simulink initializes masked blocks that are not initially visible when you open the model that contains these blocks.

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 1-8 and `load_system` for information about loading a model without displaying it.

Initialization commands for all masked blocks in a model that have drawing commands run when you:

- Update the diagram
- Start simulation
- Start code generation

Initialization commands for an individual masked block run when you:

- 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.

Callback Code Execution

Simulink executes the callback commands in the following cases:

- 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's focus (that is, you press the **Tab** key or click into another field in the dialog box).

Note When you modify the parameter value by using the `set_param` command, the callback commands do not 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 (See “Initialization Pane”).
- You hover over a masked block to see the data tip for the block, when the data tip contains parameter names and values. The callback executes again when the block data tip disappears.

Note Callback commands do not execute if the mask dialog box is open when the block data tip appears.

- Update a diagram (for example, by pressing **Ctrl+D** or by selecting **Simulation > Update diagram** in the Simulink Editor).

Mask Terminology

Terminology Table

Term	Description
Mask icon	The masked block icon generated using drawing commands. This icon may be static or change dynamically with underlying block parameter
Mask parameters	Parameters defined in the Mask Editor that link to underlying block parameters. Setting a mask parameter sets the corresponding block parameter.
Mask initialization code	MATLAB code that initializes a masked block or reflects current parameter values.
Mask callback code	MATLAB code that runs when the value of a mask parameter changes. Use callback code to modify a mask dialog box to reflect current parameter values.
Mask documentation	Description and usage information for a masked block defined in the Mask Editor.
Mask dialog	A dialog box that contains fields for setting mask parameter values and provides mask documentation.
Mask workspace	Masks that define mask parameters or contain initialization code have a mask workspace. This workspace stores mask parameters and temporary values used by the mask.

Mask a Block

This example shows how to create a block mask and define its parameters.

In this section...
“Create mask” on page 26-12
“Define mask parameters” on page 26-12
“Set mask parameter values” on page 26-13

Create mask

- 1 Open the model `subsystem_example`. Alternately, execute the following command in MATLAB:

```
open_system([docroot ' /toolbox/simulink/ug/examples/masking..  
/subsystem_example'])
```

This model contains a Subsystem block that models the equation for a line: $y = mx + b$.

- 2 Double-click the subsystem block to open it.

Notice that this subsystem contains the following blocks that are controlled by parameters.

- Gain block, named *Slope*
- Constant block, named *Intercept*

- 3 Return to the top-level model, right-click the subsystem block, and select **Mask > Create Mask**.

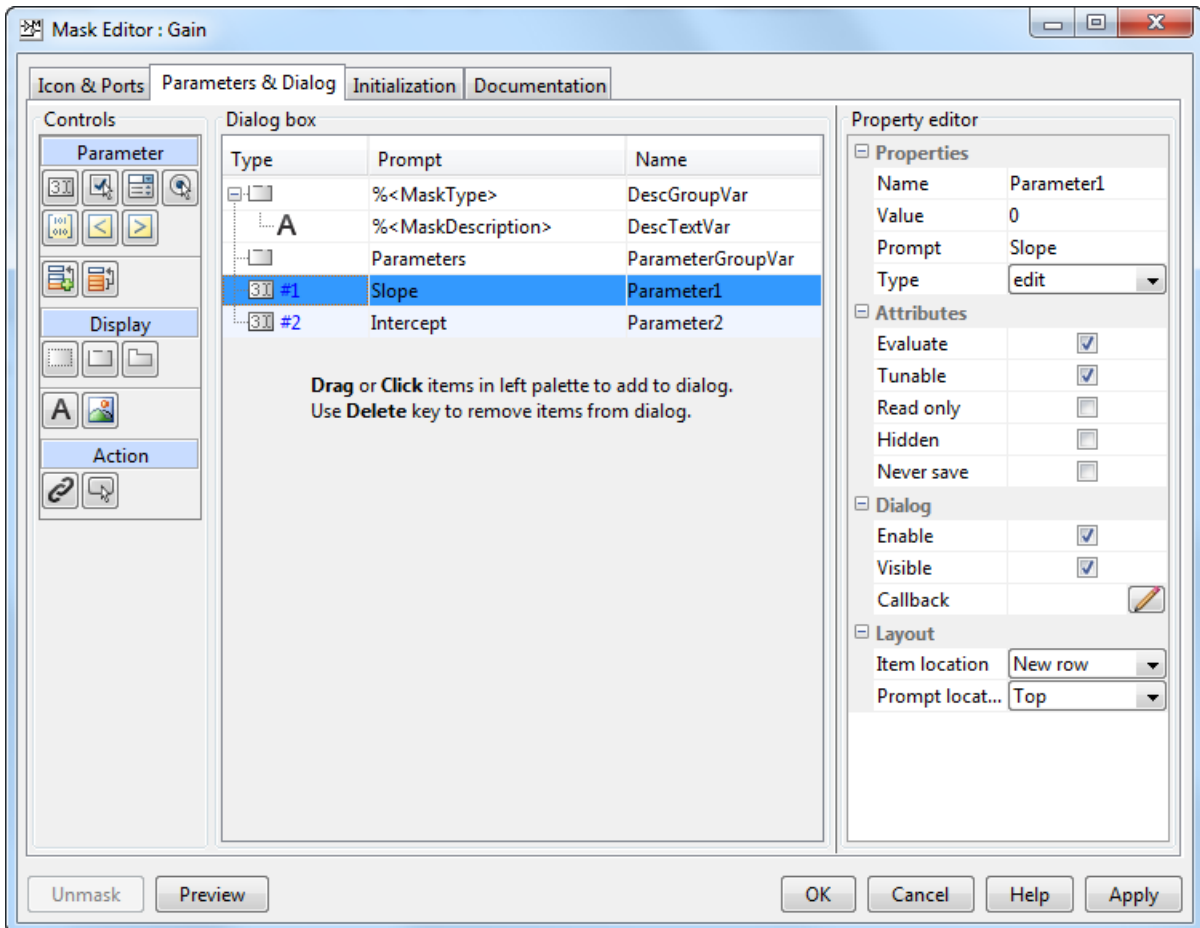
The Mask Editor appears.

Define mask parameters

Define parameters to control the underlying blocks.

- 1 In the Mask Editor, click the **Parameters & Dialog** tab.

- 2 Double click the **Edit** parameter icon and add two rows.
- 3 In the rows that appear, specify the parameters as follows.



- 4 Click **Apply**.

Set mask parameter values

Provide values to the parameters.

1 Double-click the mask to view the mask dialog box.

2 Set **Slope** and **Intercept** as 5 and 2, respectively.

To control the underlying blocks, change these parameters.

3 Click **OK**.

Draw Mask Icon

This example shows how to use drawing commands to create a mask icon. You can create icons that update when you change the mask parameters, thereby reflecting the purpose of the block.

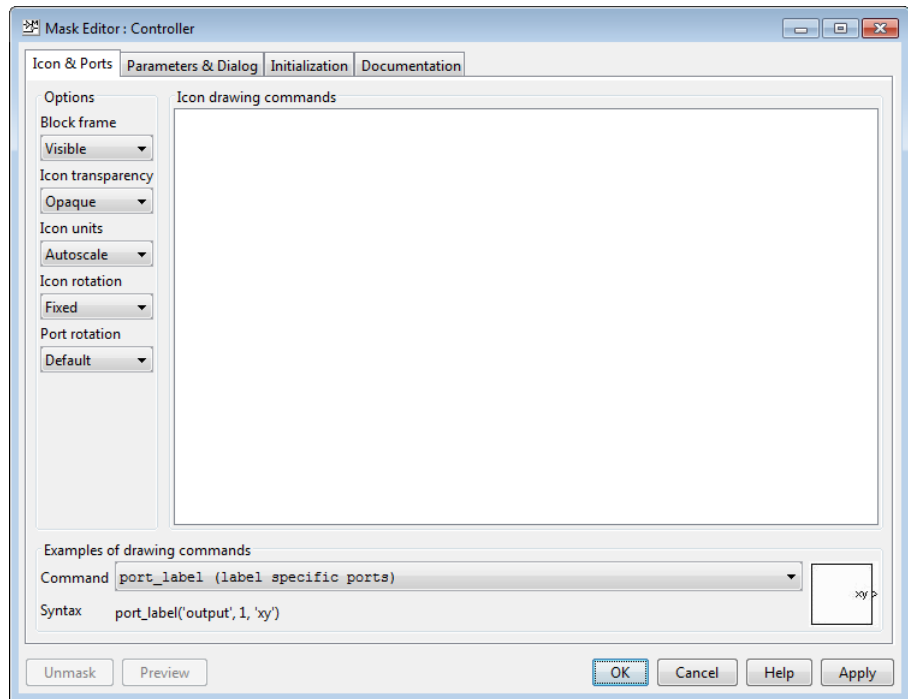
In this section...
“Draw static icon” on page 26-15
“Draw dynamic icon” on page 26-17
“Additional examples” on page 26-18

Draw static icon

A static mask icon remains unchanged, independent of the value of the mask parameters.

- 1 Right-click the masked block that requires the icon and select **Mask > Edit Mask**.

The Mask Editor appears.



2 In the **Icons & Ports** tab, enter the following 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. You can use images in .svg format as block mask image.

For more examples of drawing command syntax, explore the **Command** drop-down list in the **Examples of drawing commands** pane.

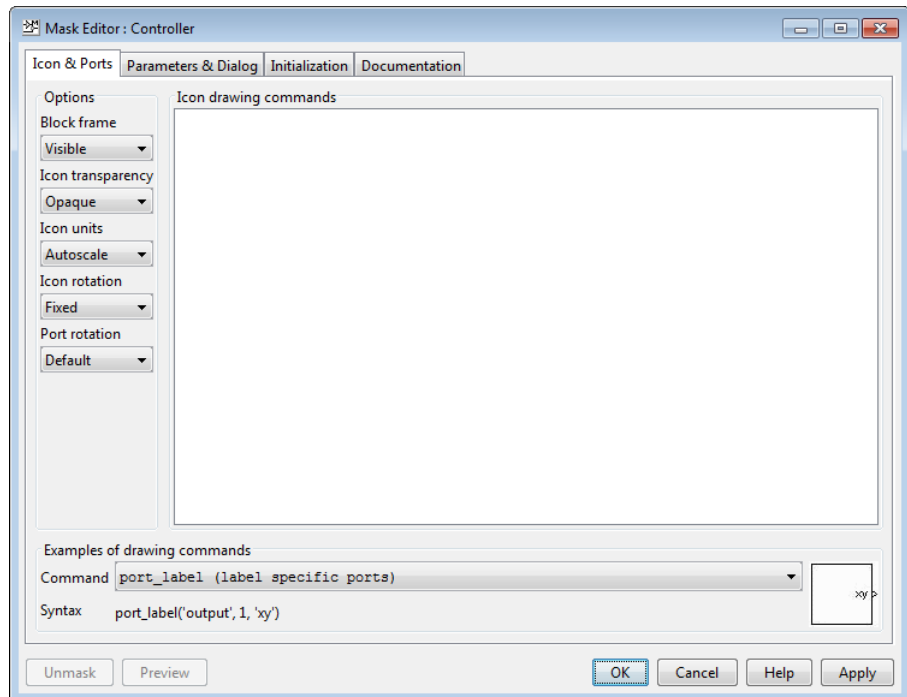
Images in formats: .cur, .hdf4, .ico, .pcx, .ras, .xwd cannot be used as block mask images. However, you can use images in these formats if you wrap the filename in the `imread()` function and use the RGB triplet. Using the `imread()` function is not efficient, however it is still supported for backward 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 Right-click the masked block that requires the icon and select **Mask > Edit Mask**.

The Mask Editor appears.



- 2 In the **Icons & Ports** tab, enter the following command in the **Icon Drawing commands** pane:

```
pos = get_param(gcb, 'Position');
width = pos(3) - pos(1);
x = [0, width];
y = m*x + b;
plot(x,y)
```

3 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.

4 Click **Apply**. If Simulink cannot evaluate all commands in the **Icon Drawing commands** pane to generate an icon, three question marks (? ? ?) appear on the mask. See `model_masking_example` to view the icon generated.

Additional examples

See model `slexMaskDisplayAndInitializationExample` for more examples of icon drawing commands. This model shows how to draw:

- a static mask
- a dynamic shape mask
- a dynamic text mask
- an image mask

Create Mask Documentation

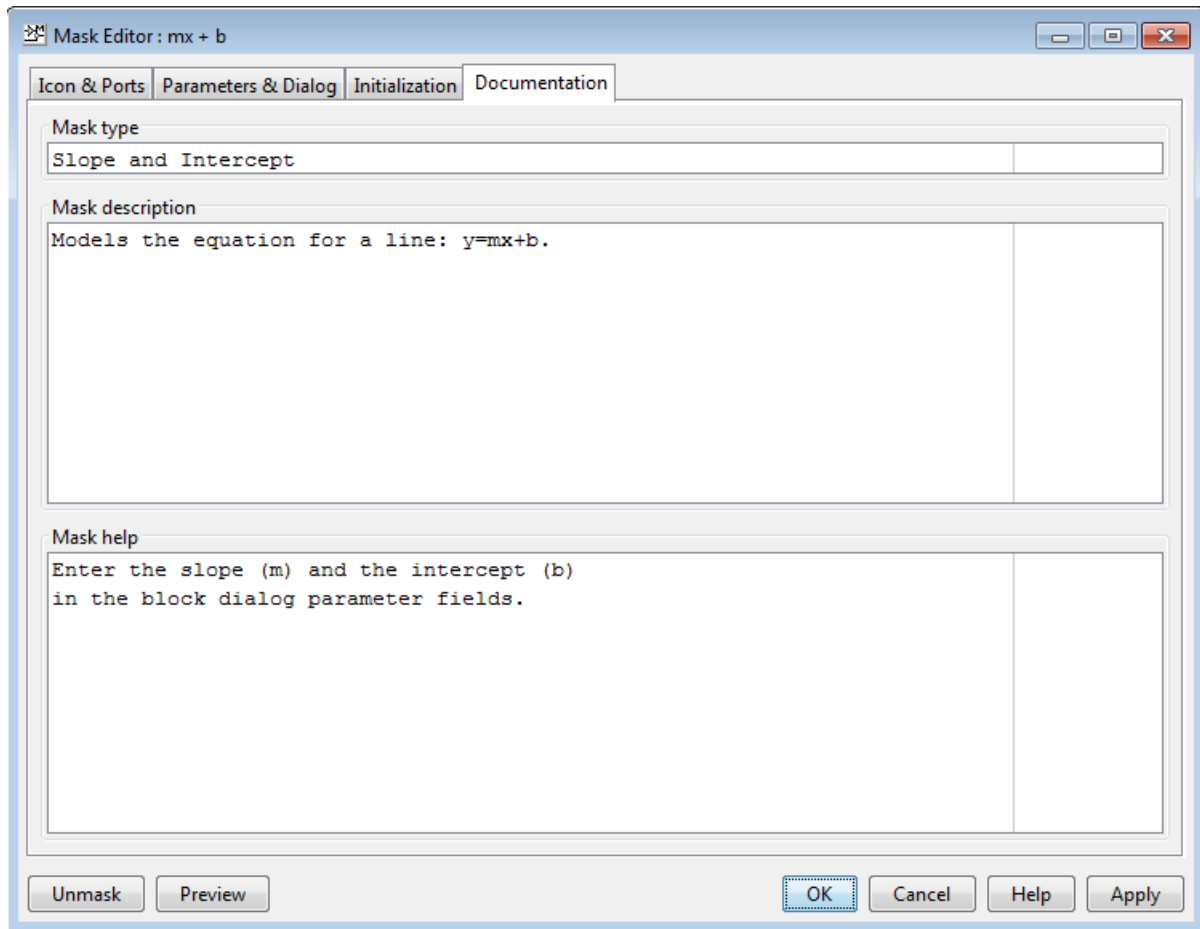
This example shows how to create mask documentation for display in the mask dialog box.

- 1 Right-click the masked block to document and select **Mask > Edit Mask**.

The Mask Editor appears.

- 2 In the **Documentation** tab, enter the following information:

- **Mask type:** The name of the mask. This name appears at the top of the mask dialog box. New lines are not permitted.
- **Mask description:** A summary of what the mask does. This description appears below the mask name, and it contain new lines as well as spaces.
- **Mask help:** Additional mask information that appears when you click **Help** in the mask dialog box. You can use plain text, HTML and graphics, URLs, and web or eval commands.



Initialize Mask

The initialization code is MATLAB code that you specify and that Simulink runs to initialize the masked subsystem at critical times, such as model loading and the start of a simulation run (see “Initialization Command Execution” on page 26-8). You can use the initialization code to set the initial values of the mask parameters.

The masked subsystem initialization code can refer only to variables in its local workspace.

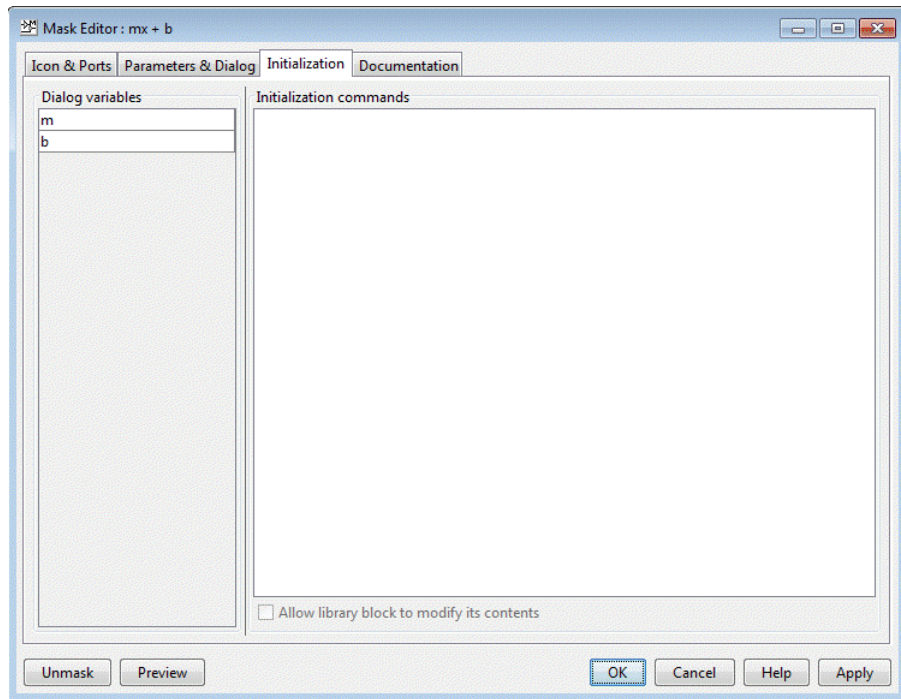
When you reference the block within, or copy the block into, a model, the mask dialog box displays the specified default values. You cannot use mask initialization code to change mask parameter default values in a library block or any other block.

Mask Editor Initialization Pane

Use the Mask Editor **Initialization** pane to enter MATLAB commands that initialize a masked block. Reference information about the **Initialization** pane appears in “Initialization Pane”.

The **Initialization** pane has two sections:

- **Dialog variables** list
- **Initialization commands** edit area



Dialog variables

The **Dialog variables** list displays the names of the variables associated with the mask parameters of the masked block (that is, the parameters defined in the **Parameters** pane).

You can copy the name of a parameter from this list and paste it into the adjacent **Initialization commands** field, using the Simulink keyboard copy and paste commands.

You can also use the list to change the names of mask parameter variables. To change a name, double-click the name in the list. An edit field containing the existing name appears. Edit the existing name and press **Enter** or click outside the edit field to confirm your changes.

Initialization Commands

Enter the initialization commands in this field. You can enter any valid MATLAB expression, consisting of MATLAB functions and scripts, operators, and variables defined in the mask workspace. Initialization commands cannot access base workspace variables.

Terminate initialization commands with a semicolon to avoid echoing results to the MATLAB Command Window.

For information on debugging initialization commands, see “Initialization Command Limitations” on page 26-23 and “Debug Masks That Use MATLAB Code” on page 26-60.

Initialization Command Limitations

Mask initialization commands must observe the following rules:

- Do not use initialization code to create dynamic mask dialog boxes (that is, dialog boxes whose appearance or control settings change depending on changes made to other control settings). Instead, use the mask callbacks that are specifically for this purpose. For more information, see “Create Dynamic Mask Dialog Boxes” on page 26-50.
- Avoid using `set_param` commands on blocks residing 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 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 the following command:

```
set_param([gcb '/B/C'], 'SampleTime', '-1');
```

Simulating or updating a model containing A causes an unresolved symbol error.

Best Practices for Masking

These examples show best practices for masking Simulink blocks. There are also some examples that show practices to avoid.

In this section...
“Use These Best Practices” on page 26-24
“Avoid These Practices” on page 26-24

Use These Best Practices

- Set mask parameters
- Group parameters under tabs
- Promote mask parameters
- Sequence mask callbacks
- Define mask display and initialization
- Create dynamic mask dialog boxes
- Use self-modifying library masks
- Use handle graphics in masking

Avoid These Practices

- Unsafe mask callbacks
- Unsafe nested mask callbacks

Considerations for Masking Model Blocks

In this section...
“Referenced Model Name” on page 26-25
“Variable Workspace” on page 26-26

Referenced Model Name

You can use a mask parameter to specify the name of a model referenced by

- a masked Model block, or
- a Model block in a masked subsystem

In these cases, the mask parameter should receive the name of the reference model literally, without being evaluated, because Simulink updates model reference targets before mask parameters.

Use one of the following approaches to obtain the literal name of the referenced model:

- **Restricted model names:** In the **Parameters & Dialog** pane of the **Mask Editor**, select the parameter that stores the referenced model name. Set its **Type** to **popup** and clear the check box for **Evaluate**.

With this approach, users can only select a model name from a drop-down list in the mask dialog box. Further, since the **Evaluate** option is cleared, the name is provided literally and not numerically evaluated.

- **Unrestricted model names:** In the **Parameters & Dialog** pane of the **Mask Editor**, select the parameter that stores the referenced model name. Set its **Type** to **edit** and clear the check box for **Evaluate**.

With this approach, users can type the model name in the mask dialog box. However, since the **Evaluate** option is cleared, the name is provided literally and not numerically evaluated.

See “Parameters & Dialog Pane” for more information about **Pop-Up** and **Edit** controls.

Variable Workspace

When you mask a model block that references another model, the referenced model cannot access the mask workspace of the model block.

Therefore, variables used by the referenced model must resolve either to workspaces defined by the referenced model or to the base workspace.

Masks on Blocks in User Libraries

In this section...

“About Masks and User-Defined Libraries” on page 26-27

“Masking a Block for Inclusion in a User Library” on page 26-27

“Masking a Block that Resides in a User Library” on page 26-27

“Masking a Block Copied from a User Library” on page 26-28

About Masks and User-Defined Libraries

You can mask a block that will be included in a user library or already resides in a user library, or you can mask an instance of a user library block that you have copied into a model. For example, a user library block might provide the capabilities that a model needs, but its native interface might be inappropriate or unhelpful in the context of the particular model. Masking the block could give it a more appropriate user interface.

Masking a Block for Inclusion in a User Library

You can create a custom block by encapsulating a block diagram that defines the block’s behavior in a masked subsystem and then placing the masked subsystem in a library. You can also apply a mask to any other type of block that supports masking, then include the block in a library.

Masking a block that will later be included in a library requires no special provisions. Create the block and its mask as described in this chapter, and include the block in the library as described in “Create Block Libraries” on page 28-20.

Masking a Block that Resides in a User Library

Creating or changing a library block mask immediately 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 already exist as separate copies.

To apply or change a library block mask, open the library that contains the block. Apply, change, or remove a mask as you could if the block did not reside

in a library. In addition, you can specify non-default values for block mask parameters. When the block is referenced within or copied into a model, the specified default values appear on the block's mask dialog box. By default, edit fields have a value of zero, check boxes are cleared, and drop-down lists select the first item in the list. To change the default for any field:

- 1** Fill in the desired default values or change check box or drop-down list settings
- 2** Click **Apply** or **OK** to save the changed values into the library block mask.

Be sure to save the library after changing the mask of any block that it contains. Additional information relating to masked library blocks appears in “Create Block Libraries” on page 28-20.

Masking a Block Copied from a User Library

A block that was copied from a user library, as distinct from a block accessed by using a library reference, has no special status with respect to masking. You can add a mask to the copied block, or change or remove any mask that it already has.

Promote Underlying Block Parameters to Mask

This example shows to promote the parameters of underlying blocks to the mask. See model `slexMaskParameterPromotionExample` for more examples of parameter promotion.

- 1 Right-click a block or subsystem in your model and select **Mask > Create Mask**.

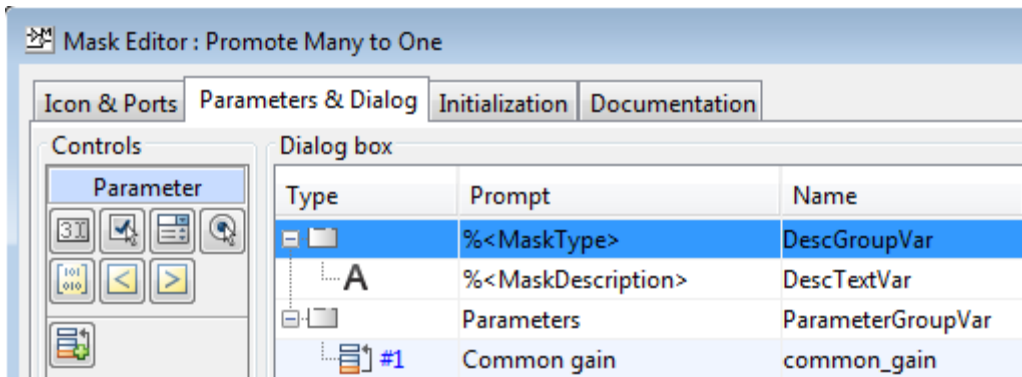
The Mask Editor appears.

- 2 Select the **Parameters & Dialog** pane.

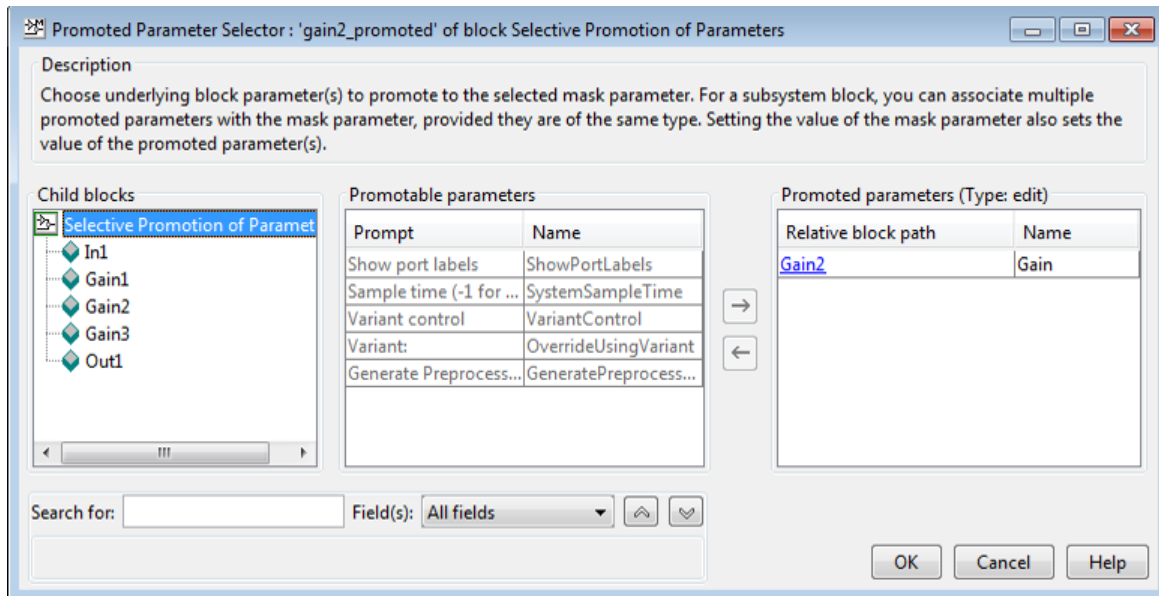
- 3 Choose one or more parameters for promotion.

- a Double click any of the icons in the **Parameter** palette.

- b Click the **Promote** button.



The **Promoted parameter selector** dialog box opens.



- c Choose underlying block parameters to promote to the currently selected mask parameter.
 - Select a parameter in the **Promotable parameters** table and click the right arrow button to add to the **Promoted parameters** list.

View the tool tips on promotable parameters to see key fields such as **Type**.
 - When masking a subsystem, you can use the **Child blocks** list or the **Search** box to find underlying block parameters to promote.
 - (Optional) When masking a subsystem, you can associate multiple promoted parameters with the currently selected mask parameter, provided they are of the same type.

For example, you can promote multiple Gain parameters in a subsystem to a single prompt on your mask.
 - d Click **OK** to return to the Mask Editor.
- 4 Edit the prompt names for the mask, if desired. You cannot edit the variable names of built-in block parameters. See “Rules for Promoting Parameters” on page 26-35.

- 5 Repeat step 3 to add mask parameters and add or edit additional promoted parameters.

If you want to promote many parameters from a built-in block, click

Promote All .

The **Promote All** button is only available for block masks, not subsystem masks.

All the parameters appear in the Mask Editor. To remove any unwanted parameters, use the **Delete Parameter** button.

- 6 Click **OK** to finish editing the mask. Now whenever you set a mask parameter, you also set the value of the underlying promoted parameters.

Create Custom Interface for Simulink Blocks

This example shows how to mask the Gain block and promote only the Gain parameter to the mask, while hiding the other options and adding custom parameters.

- 1 Right-click a Gain block in your model and select **Mask > Create Mask**.

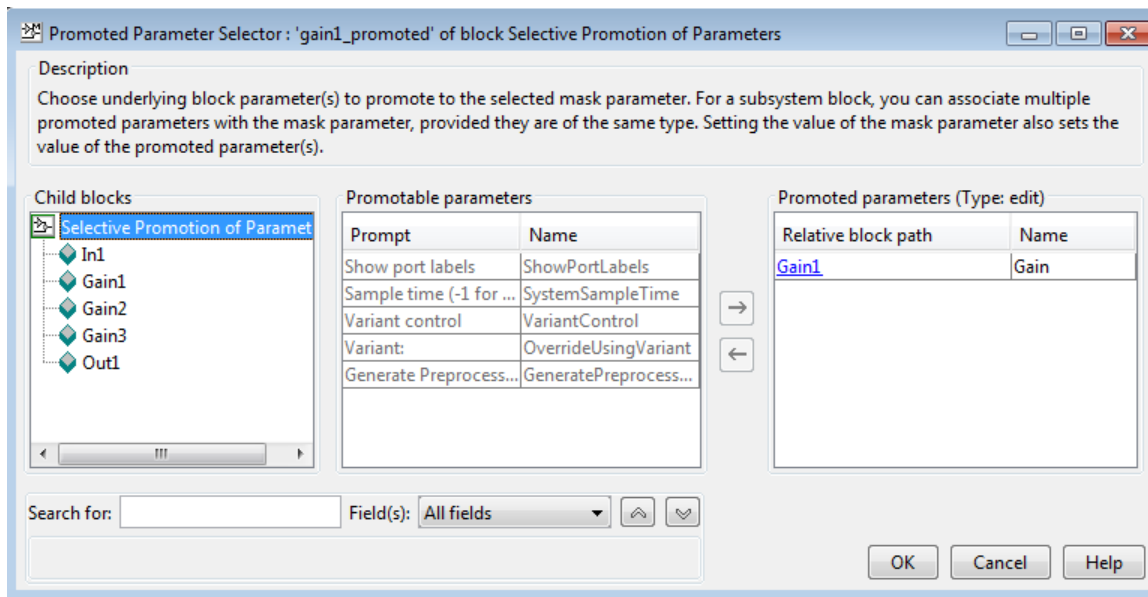
The Mask Editor appears.

- 2 Select the **Parameters & Dialog** pane.

- 3 Click the **Promote** button. .

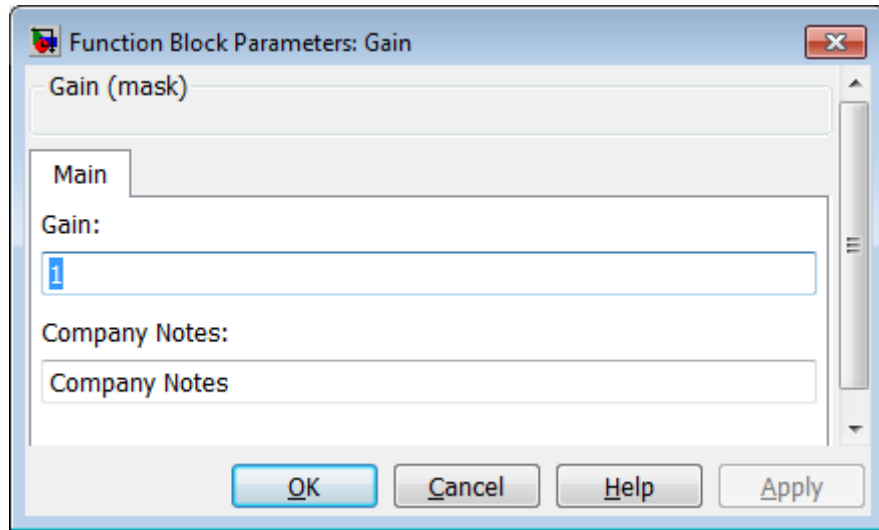
The **Promoted parameter selector** dialog box opens.

- 4 Select the Gain parameter in the **Promotable parameters** table and click the right-arrow button to add to the **Promoted parameters** list.

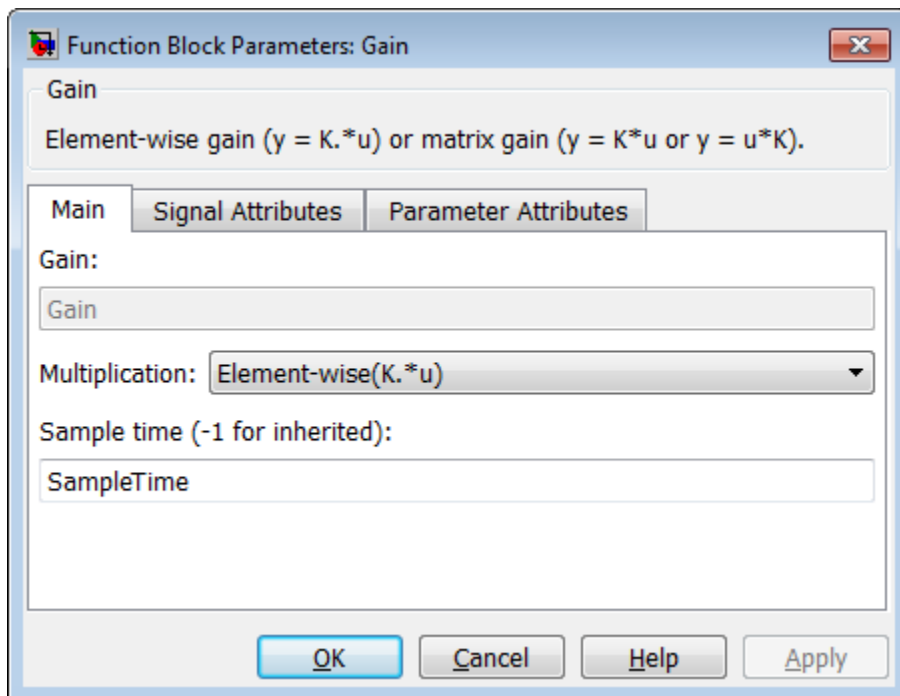


- 5 Click **OK** to return to the Mask Editor.

- 6 Double-click the Gain block. The new mask opens, showing only the promoted Gain parameter and custom parameter.



- 7 Look under the mask to see the underlying Gain block dialog box. You can see that the Gain parameter is now disabled because you have promoted it to the mask.



Rules for Promoting Parameters

In this section...

“General Rules” on page 26-35

“Promotion from directly masked block” on page 26-35

“Promotion from child blocks within subsystems” on page 26-36

“Links created from masked blocks” on page 26-36

General Rules

- Some parameters with certain attributes cannot be promoted, and if you try, an error message appears.
- After you promote a parameter to the mask, the following rules apply.
 - The parameter is disabled on the block dialog box. If you try to directly edit promoted parameters, either in the dialog or at the command line, an error message appears.
 - You cannot promote the same parameter again to a different mask parameter.
 - The mask parameter inherits the Evaluate property from the parameter promoted to it, and you cannot change it. Any mismatch between the Evaluate property of the mask parameter and the promoted parameter results in an error message.

Promotion from directly masked block

- The mask parameter must have the same variable name as that of the parameter promoted to it. You cannot change the promoted mask parameter variable names because they are strictly inherited from the underlying built-in block parameter.
- The mask tries to retain the dynamic dialog behavior, if any, of the promoted parameter. You can specify your own dynamic dialog behavior of the mask parameter. If you do, the mask first executes the dynamic dialog callback of the promoted parameter, followed by the user-specified dynamic dialog callback of the mask parameter.

- You cannot promote multiple parameters to a single mask parameter.

Promotion from child blocks within subsystems

- You can associate multiple parameters provided they are of the same type. If the parameter is of type `popup` or `DataType` then the options must also be the same for them to be promoted together. The `Evaluate` property among the parameters to be promoted together must be similar.

View the tool tips on promotable parameters to see key fields such as `Type` and `Evaluate`.

- If a child block is masked, you cannot promote the underlying block dialog parameters.
- For child blocks, you cannot view or promote parameters from inside a masked block.
- For child blocks, you cannot view or promote parameters from inside a linked block.

Links created from masked blocks

- The underneath block dialog opens completely disabled for the links of masked blocks. You can edit values only from the mask dialog box.
- If the mask author decides not to promote a parameter, its value in the linked blocks is tied to the library value for that parameter as specified in the library.

Mask Blocks and Promote Parameters

In this section...

“Mask Built-In Blocks Directly and Within Subsystems” on page 26-37

“Create Custom Interface for Multiple Parameters in Subsystem” on page 26-37

Mask Built-In Blocks Directly and Within Subsystems

You can directly mask built-in blocks with the Mask Editor to provide custom icons and dialog boxes. In the Mask Editor, you can choose to *promote* any underlying parameter of any block to the mask. For subsystems, you can choose to promote parameters from any child blocks. For a subsystem block, you can associate a single mask parameter with multiple promoted parameters if they are of the same type. Changing the value of the mask parameter also sets the value of the associated promoted parameters.

You can use masking in the following use cases:

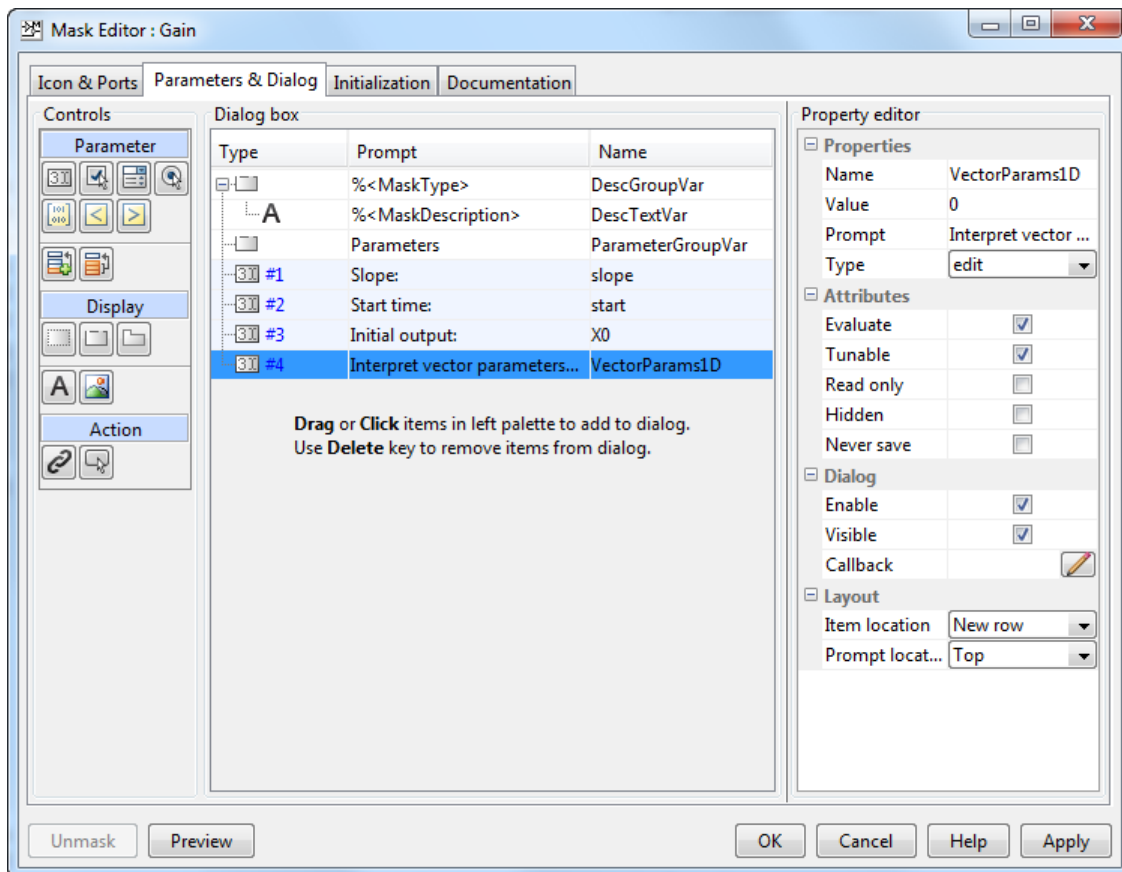
- You can directly mask a built-in block to simplify the interface and add custom parameters.
- You can promote multiple parameters from child blocks to a single mask parameter for a subsystem block.
- You can promote all parameters for a directly masked built-in block, and then choose a subset of the parameters to keep in the mask.

Create Custom Interface for Multiple Parameters in Subsystem

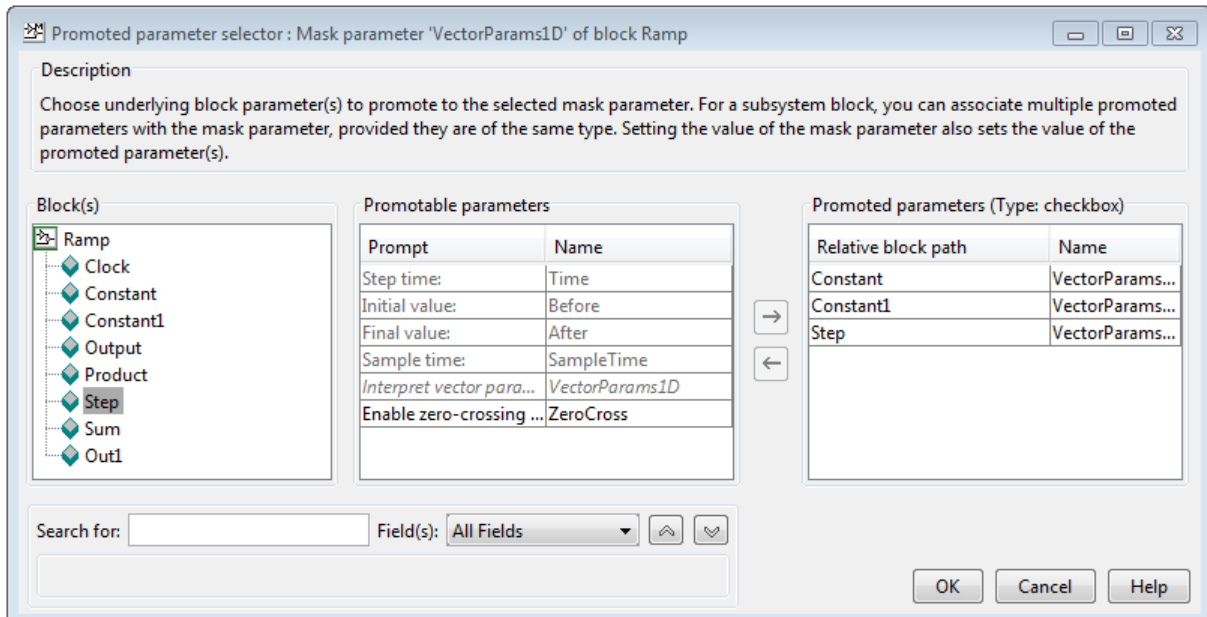
You can promote multiple parameters from child blocks to a single mask parameter for a subsystem block. This enables you to use a simplified single setting in the mask to set multiple child block parameters (of the same type) in the subsystem.

The following example demonstrates this capability. It recreates the functions of the Ramp block by using other blocks in a subsystem, and then masking the subsystem to create a simplified custom interface.

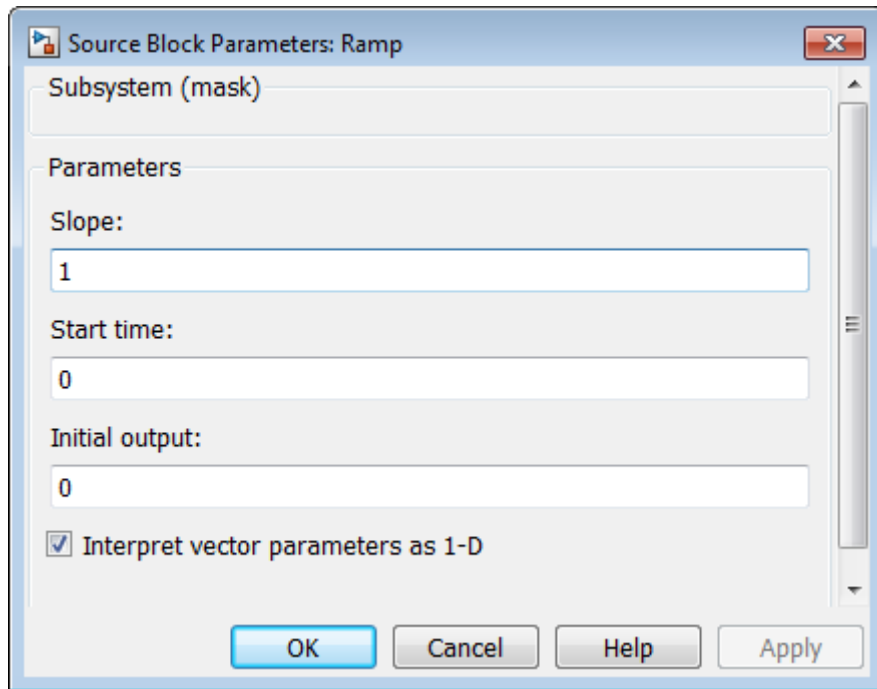
The **Mask Editor** contains a selection of promoted parameters from underlying blocks within the subsystem. The selected parameter has three underlying parameters of the same type promoted to the same mask parameter.



The Promoted parameter selector dialog shown next demonstrates how to specify this setup. Three parameters of the same type, from different child blocks, are added to the **Promoted parameters** list to promote to the currently selected mask parameter.



The mask shows the four parameters specified in the **Mask Editor**.



When you select the **Interpret vector parameters as 1-D** check box on the mask, you also set the underlying promoted parameters in the three child blocks.

Operate on Existing Masks

In this section...

- “Change a Block Mask” on page 26-41
- “View Mask Parameters” on page 26-41
- “Look Under Block Mask” on page 26-41
- “Remove and Cache Mask” on page 26-42
- “Restore Cached Mask” on page 26-43
- “Permanently Delete Mask” on page 26-43

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 Select **Mask > Edit Mask**.

The Mask Editor reopens, showing the existing mask definition. Change the mask as needed. After you change a mask, be sure to save the model before closing it, or the changes will be lost.

View Mask Parameters

To display a mask dialog box, double-click the block. Alternatively, right-click the block and select **Mask > Mask Parameters**.

To display the block dialog box that double-clicking would display if no mask existed, right-click the masked block and select **Block Parameters (BlockType)**.

Look Under Block Mask

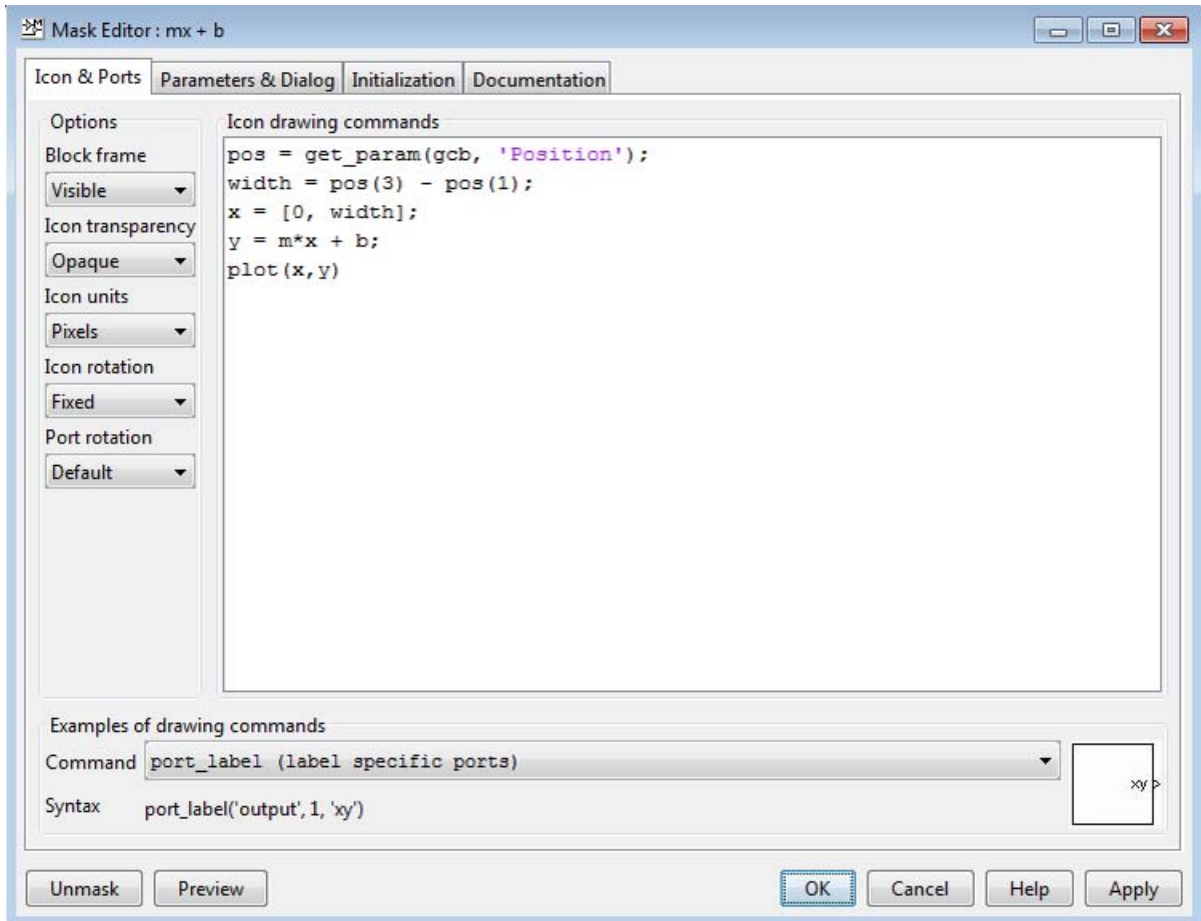
To see the block diagram under a masked Subsystem block, built-in block, or the model referenced by a masked model block, right-click the block and select **Mask > Look Under Mask**.

Remove and Cache Mask

To remove a mask from a block and cache it for possible restoration later:

- 1 Right-click the block.
- 2 Select **Mask > 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, saves the mask in a cache for possible restoration, then closes. The editor caches masks separately for each block, so removing a mask from one block has no effect on a mask cached for any other block. Closing the Mask Editor has no effect on cached masks.

When you have removed and cached a mask, you can later restore it, as described in “Restore Cached Mask” on page 26-43, or delete it, as described in “Permanently Delete Mask” on page 26-43. The removed cached mask has no further effect unless you restore it.

Restore Cached Mask

As long as a model remains open, you can restore a mask that you removed as described in “Remove and Cache Mask” on page 26-42.

- 1 Right-click the block.
- 2 Select **Mask > Create Mask**.

The Mask Editor reopens, showing the cached masked definition.

- 3 Modify the definition if needed, using the techniques in “Mask a Block” on page 26-12
- 4 Click **Apply** or **OK** to restore the mask, including any changes that you made.

If you made any changes, be sure to save the model before closing it, or the changes will be lost.

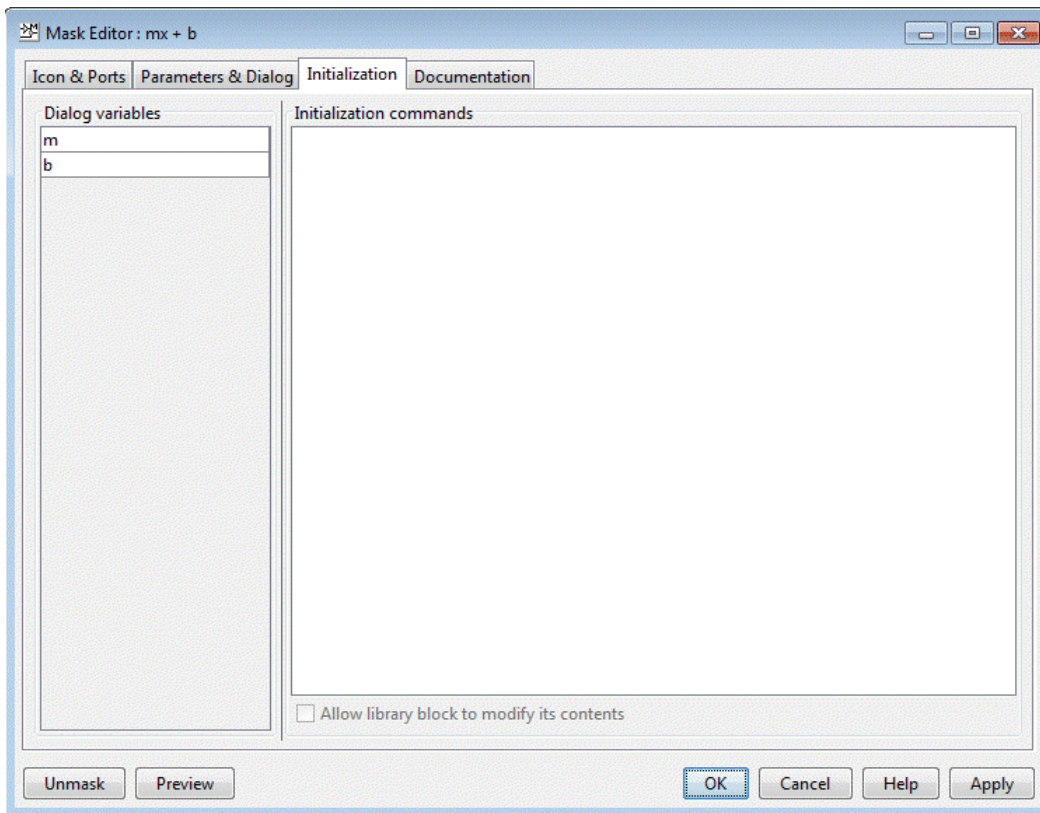
Permanently Delete Mask

To delete a mask permanently, first remove it as described in “Remove and Cache Mask” on page 26-42, then save and close the model. You do not need to close the model immediately after removing a mask that you intend to delete. The removed mask remains in the cache and has no further effect unless you restore it.

Calculate Values Used Under the Mask

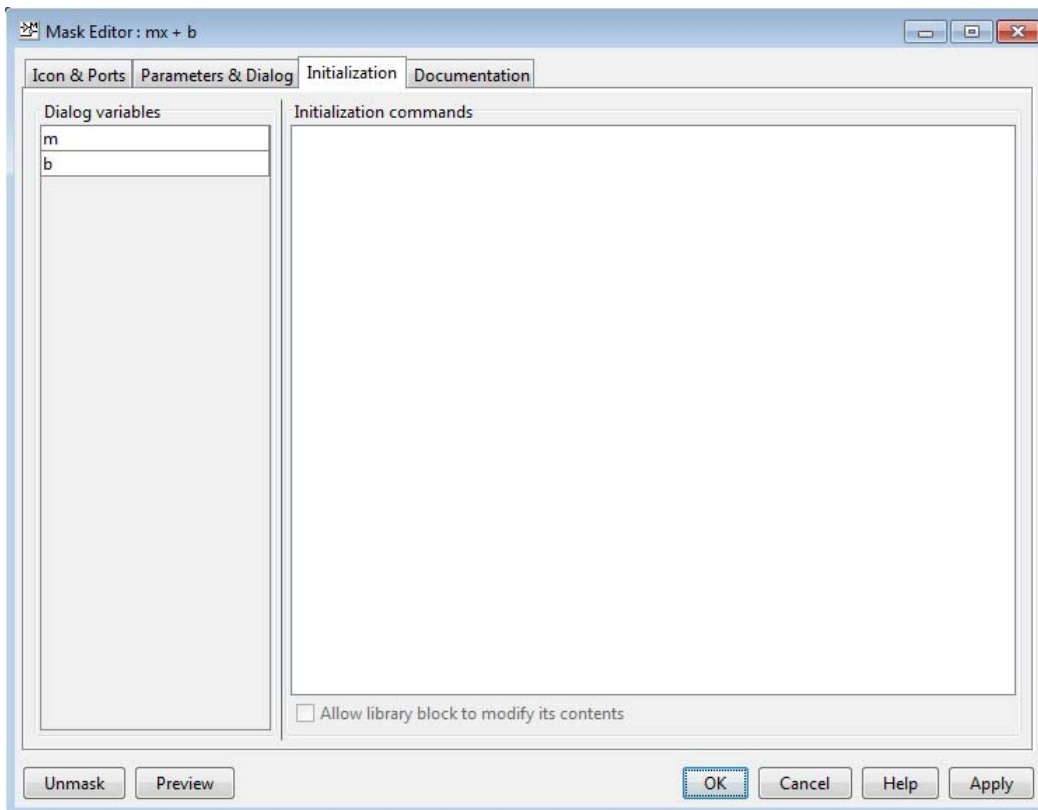
The `masking_example` assigns the values input using the mask dialog box directly to block parameters underneath the mask, as described in “How Mask Parameters Work” on page 26-5. The assignment occurs because the block parameter and the mask parameter have the same name, so the search that always occurs when a block parameter needs a value finds the mask parameter value automatically, as described in “Symbol Resolution” on page 4-83.

You can use the Mask Editor to insert any desired calculation between a value in the mask dialog box and an underlying block parameter:



See the “Initialization Pane” reference for reference information about all **Initialization** pane capabilities. This section shows you how to use it for calculating block parameter values.

To calculate a value for a block parameter, first break the link between the mask and block parameters by giving them different names. To facilitate such changes, the Dialog variables subpane lists all mask parameters. The **Initialization** pane looks like this:



You cannot use mask initialization code to change mask parameter default values in a library block or any other block.

You can use the initialization code for a masked block to link mask parameters indirectly to block parameters. In this approach, the initialization code creates variables in the mask workspace whose values are functions of the mask parameters and that appear in expressions that set the values of parameters of blocks concealed by the mask.

If you need both the string entered and the evaluated value, clear the **Evaluate** option. To get the value of a base workspace variable entered as the literal value of the mask parameter, use the MATLAB `evalin` command in the mask initialization code. For example, suppose the user enters the string 'gain' as the literal value of the mask parameter `k` where `gain` is the name of a base workspace variable. To obtain the value of the base workspace variable, use the following command in the initialization code for the mask:

```
value = evalin('base', k)
```

These values are stored in variables in the *mask workspace*. A masked block can access variables in its mask workspace. A workspace is associated with each masked subsystem that you create. The current values of the subsystem's parameters are stored in the workspace as well as any variables created by the block's initialization code and parameter callbacks.

To use a masked subsystem in a referenced model that uses model arguments, do not create in the mask workspace a variable that derives its value from a mask parameter. Instead, use blocks under the masked subsystem to perform the calculations for the mask workspace variable.

Control Masks Programmatically

In this section...

“Use `Simulink.Mask` and `Simulink.MaskParameter`” on page 26-47

“Use `get_param` and `set_param`” on page 26-49

Simulink defines a set of parameters that help in setting and editing masks. To set and edit mask from the MATLAB command line, you can use instances of the `Simulink.Mask` and `Simulink.MaskParameter` classes. You can also use the `get_param` and `set_param` functions to set and edit masks. However, since these functions use de-limiters that do not support Unicode (Non English) characters you must use instances of the `Simulink.Mask` and `Simulink.MaskParameter` classes to control masks.

Use `Simulink.Mask` and `Simulink.MaskParameter`

Use instances 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

I In this example the `Simulink.Mask.create` method is used to create a block mask:

```
maskObj = Simulink.Mask.create(gcf);
```

```
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(gcf)
```

```

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(gcf, 'MaskStyleString', 'edit,edit',...
'MaskVariables', 'maskparameter1=@1;maskparameter2=&2;',...
'MaskPromptString', 'Mask Parameter 1:|Mask Parameter 2:',...
'MaskValues', {'1', '2'});

get_param(gcf, 'MaskStyleString');

set_param(gcf, 'MaskStyles', {'edit', 'edit'}, 'MaskVariables',...
'maskparameter1=@1;maskparameter2=&2;', 'MaskPrompts',...
{'Mask Parameter 1:', 'Mask Parameter 2:'},...
'MaskValueString', '1|2');

get_param(gcf, 'MaskStyles');
```

where

- | separates individual string values for the mask parameters
- @ indicates that the parameter field is evaluated
- & indicates that the parameter field is not evaluated but assigned as a string

See [Mask Parameters](#), for detailed information on the mask parameters.

Create Dynamic Mask Dialog Boxes

In this section...

“About Dynamic Masked Dialog Boxes” on page 26-50

“Show parameter” on page 26-51

“Enable parameter” on page 26-51

“Setting Masked Block Dialog Box Parameters” on page 26-51

“Setting Nested Masked Block Parameters” on page 26-53

About Dynamic Masked Dialog Boxes

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.

Creating a dynamic masked dialog box entails using the Mask Editor in combination 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 “Callback Code Execution” on page 26-9). 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 “Setting Masked Block Dialog Box Parameters” on page 26-51). Finally, you save the model or library containing the masked subsystem to complete the creation of the dynamic masked dialog box.

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.

Setting Masked Block Dialog Box Parameters

The following example creates a mask dialog box with two parameters. The first parameter is a pop-up menu that selects one of three gain values: 2, 5, or User-defined. The selection in this popup menu determines the visibility of an edit field for specifying the gain.

- 1** Mask a subsystem: Right-click the block and select **Mask > Create Mask**.
- 2** Select the **Parameters & Dialog** pane on the Mask Editor.
- 3** Add a parameter.
 - Enter **Gain:** in the **Prompt** field
 - Enter **gain** in the **Name** field
 - Uncheck **Evaluate** to use the literal value from the popup menu options.
- 4** Enter the following three values in the **Popups (one per line)** field:

```
2  
5  
User-defined
```
- 5** Enter the following code in the **Dialog callback** field:

```
% Get the mask parameter values. This is a cell
```

```

% array of strings.
maskStr = get_param(gcb, 'MaskValues');

% The pop-up menu is the first mask parameter.
% Check the value selected in the pop-up
if strcmp(maskStr{1}(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 string in the Values field equal to the
    % string selected in the Gain pop-up menu.

    maskStr{2}=maskStr{1};
    set_param(gcb, 'MaskValues', maskStr);
end

```

6 Add a second parameter.

- Enter **Value:** in the **Prompt** field
- Enter **val** in the **Variable** field
- Uncheck **Visible** property in the **Properties** pane. This step turns the visibility of this parameter off, by default.

7 Select **Apply** on the Mask Editor.

Double-clicking on the new masked subsystem opens the mask dialog box. Selecting 2 or 5 for the **Gain** parameter hides the **Value** parameter, while selecting **User-defined** makes the **Value** parameter visible. Note that any blocks in the masked subsystem that need the gain value should reference

the mask variable `val` as the `set_param` in the `else` code assures that `val` contains the current value of the gain when 2 or 5 is selected in the popup.

Setting Nested Masked Block Parameters

Avoid using `set_param` commands to set parameters of blocks residing in masked subsystems that reside in the masked subsystem being initialized. 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 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 the command

```
set_param([gcb '/B/C'], 'SampleTime', '-1');
```

Simulating or updating a model containing A causes an unresolved symbol error.

Create Dynamic Masked Subsystems

In this section...

“Allow library block to modify its contents” on page 26-54

“Create Self-Modifying Masks for Library Blocks” on page 26-54

“Evaluate Blocks Under Self-Modifying Mask” on page 26-58

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(gcf, '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.

Creating Self-Modifying Masks Using the Mask Editor

To create a self-modifying mask using the Mask Editor:

- 1 Unlock the library (see “Modify and Lock Libraries” on page 28-21).

- 2 Right-click the block in the library.
- 3 Select **Mask > 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 “Mask Code Placement” on page 26-7). Doing so triggers an error when a user edits the parameter.

- 6 Click **Apply** to apply the change or **OK** to apply the change and dismiss 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')
```

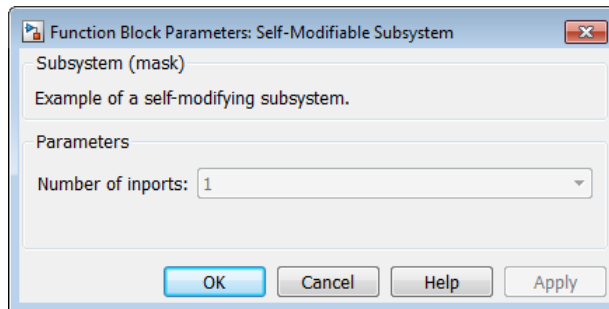
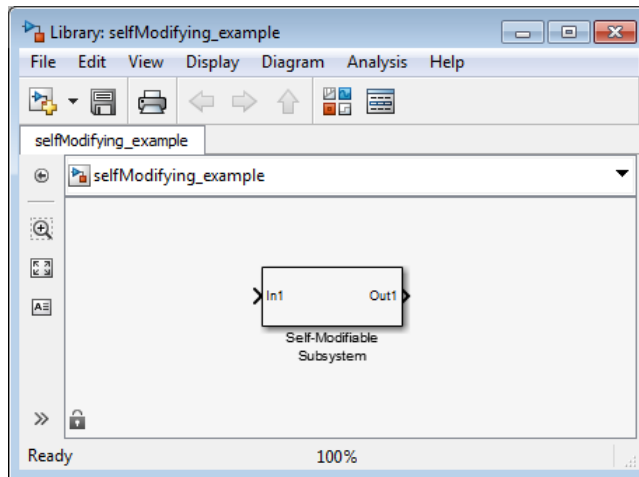
- 2 Specify that the block is self-modifying by using the following command:

```
set_param(block_name, 'MaskSelfModifiable', 'on')
```

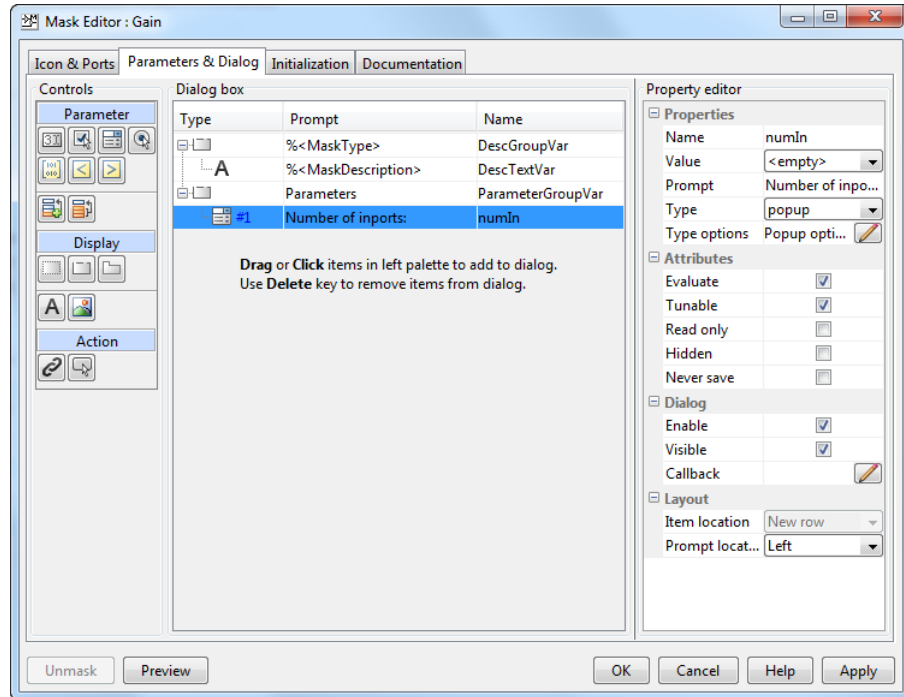
where `block_name` is the full path to the block in the library.

Self-Modifying Mask Example

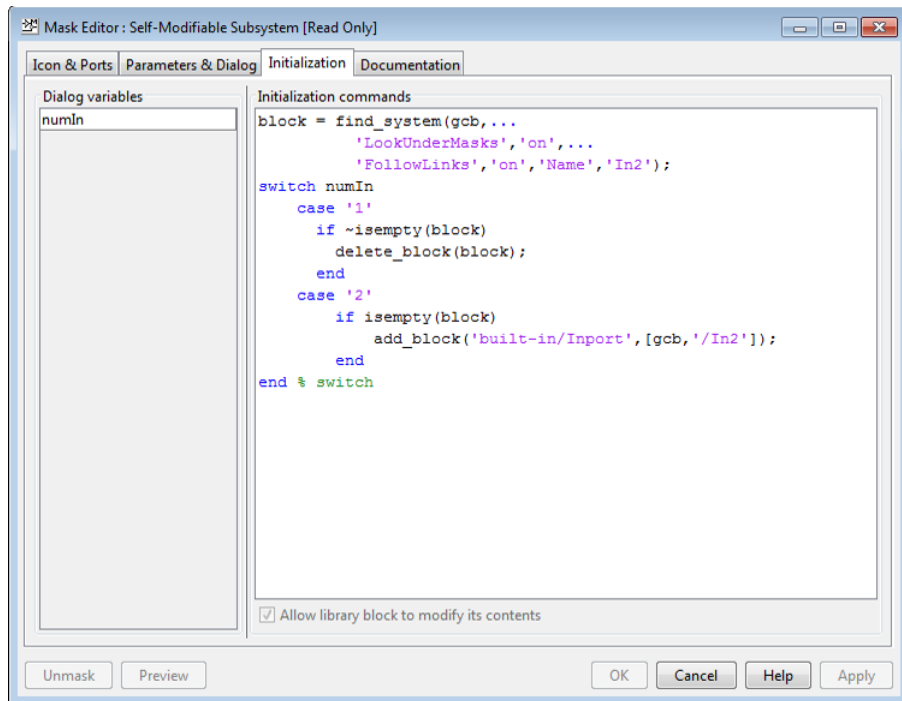
The library `selfModifying_example` contains a masked subsystem that modifies its number of input ports based on a selection made in the subsystem mask dialog box.



Right-click the subsystem then select **Mask > Edit Mask**. The Mask Editor opens. 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.



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 were not selected, copies of the library block could not modify their structural contents and changing the selection in the **Number of inports** list would produce an error.



Evaluate Blocks Under Self-Modifying Mask

This example shows how to force Simulink to evaluate blocks inside self-modifying masks.

Simulink evaluates elements of models containing masks in the following order:

- 1 Mask dialog box
- 2 Mask initialization code
- 3 Blocks or masked subsystems under the mask

Consider the following case:

A block named `myBlock` inside subsystem `mySubsys` masked by a self-modifying mask depends on mask parameter `myParam` to update itself.

`myParam` is exposed to the user through the **Mask Parameters** dialog box. `mySubsys` is updated through MATLAB code written in the **Mask Initialization** pane.

In this model, the sequence of updates is as follows:

- 1** User modifies `myParam` through the mask dialog box.
- 2** The mask initialization code receives this change and modifies `mySubsys` under the mask.
- 3** `myBlock`, which lies under `mySubsys`, modifies itself based on the change to `myParam`.

In this sequence, `myBlock`, which lies under `mySubsys`, will not be evaluated when the mask initialization code executes. Instead, only the masked subsystem `mySubsys` is evaluated at that time and gets updated. Meanwhile, `myBlock`, which needs to change, remains unmodified.

You can force Simulink to evaluate such blocks earlier by using the `Simulink.Mask.eval` method in the masked subsystem's initialization code:

```
Simulink.Block.eval(mySubsys/myBlock);
```

Debug Masks That Use MATLAB Code

In this section...
“Code Written in Mask Editor” on page 26-60
“Code Written Using MATLAB Editor/Debugger” on page 26-60

Code Written in Mask Editor

Debug initialization commands and parameter callbacks entered directly into the Mask Editor in one of the following ways:

- Remove the terminating semicolon from a command to echo its results to the MATLAB Command Window.
- Place a keyboard command in the code to stop execution and give control to the keyboard.

Code Written Using MATLAB Editor/Debugger

Note You cannot debug icon drawing commands using the MATLAB Editor/Debugger. Use the syntax examples provided in the Mask Editor **Icons & Ports** pane to help solve errors in the 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`.

Masking Linked Blocks

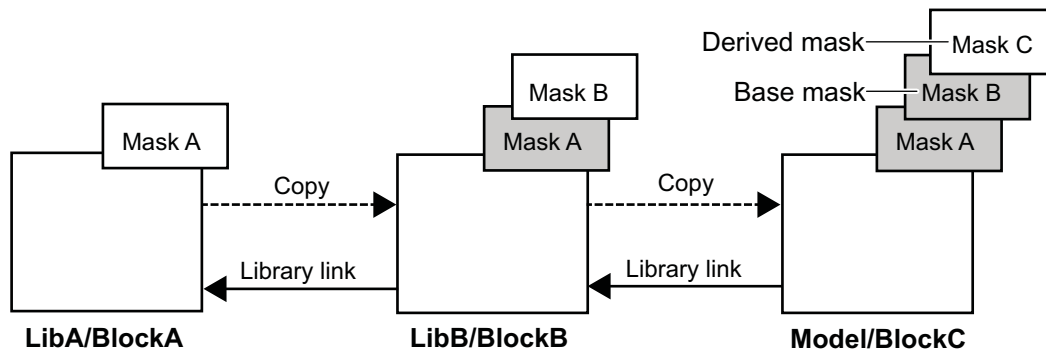
In this section...
“Guidelines for Mask Parameters” on page 26-62
“Mask Behavior for Masked, Linked Blocks” on page 26-63

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 new 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.

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 Library B to Block C. Block C now has Mask A, Mask B, and Mask C. Mask C is the derived mask and Mask B is the base mask.



Note that 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.

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.
- 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`.
- 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 **File > Export Model to > Previous Version...** , 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

In this section...

“Create a Custom Library With Mask on Link Block” on page 26-64

“Add a Mask to the Masked, Link Block” on page 26-64

“View Masks Below the Top Mask” on page 26-65

This example shows the steps for creating masks on linked blocks. To view the example model, click Mask Linked Blocks.


Create a Custom Library With Mask on Link Block

- 1 From the **Simulink Library Browser**, select **File > New > Library** .
- 2 Drag the Ramp block to the Library editor window.
- 3 Right-click the Ramp block and select **Mask > Create Mask**.

The Mask Editor opens.

- 4 In the **Icons & Ports** tab, enter the following command in the **Icon drawing commands** pane:

```
plot ([0:10],[0,1:10])
```

- 5 Click the  **Promote underlying block parameter(s) to this mask parameter** button.

Add `Slope` and `Initial Output` to the promoted parameters list, and click **OK**. Add a custom parameter `Company Notes`.

- 6 Rename the block to `Derived Ramp Block`.

Add a Mask to the Masked, Link Block

- 1 From the Simulink Library Browser, select **File > New > Model**.

- 2 Drag the Derived Ramp Block to the Model.

The Derived Ramp Block in the model has multiple masks on it. You can set parameters of the derived mask.

View Masks Below the Top Mask

- 1 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.

Creating Custom Blocks

- “When to Create Custom Blocks” on page 27-2
- “Types of Custom Blocks” on page 27-3
- “Comparison of Custom Block Functionality” on page 27-7
- “Expanding Custom Block Functionality” on page 27-17
- “Create a Custom Block” on page 27-18
- “Custom Block Examples” on page 27-44

When to Create Custom Blocks

Custom blocks expand the modeling functionality provided with the Simulink product. Use a custom block to:

- Model behaviors that are not provided with a Simulink built-in solution.
- Build more advanced models.
- Encapsulate model components into a library block that you can copy into multiple models.
- Provide custom graphical user interfaces or analysis routines.

Types of Custom Blocks

In this section...
“MATLAB Function Blocks” on page 27-3
“Subsystem Blocks” on page 27-4
“S-Function Blocks” on page 27-4

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 Fcn block allows you to use a MATLAB expression to define a single-input, single-output (SISO) block.
- 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 while you cannot generate code for models containing an Fcn block.

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 hand-written 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.
- 2 Placing the block in a library to prohibit modifications and allow for easily updating copies of the block.

For more information, see “Libraries” and “Masking”.

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 need to model 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 “C/C++ S-Functions”).
- The S-Function Builder block assists you in creating a new C MEX S-function or a wrapper function to incorporate legacy C or C++ code. (See “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 “Generated S-Function Block” in the Simulink Coder User’s Guide for details and limitations on using the S-function target.

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 while MATLAB functions for code generation are a subset of the MATLAB language. For a list of supported functions for code generation, see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2.
- MATLAB S-functions can model discrete and continuous state dynamics while MATLAB functions for code generation cannot model state dynamics.

Using S-Function Blocks to Incorporate Legacy Code

Each S-function block allows you to incorporate legacy code into your model, as follows.

- A MATLAB S-function accesses legacy code through its TLC-file. Therefore, the legacy code is available only in the generated code, not during simulation.
- A C MEX S-functions directly calls legacy C or C++ code.
- The S-Function Builder generates a wrapper function that calls the legacy C or C++ code.
- The Legacy Code Tool generates a C MEX S-function to call the legacy C or C++ code, which is optimized for embedded systems. See “Integrate C Functions Using Legacy Code Tool” for more information.

See “Integration Options” in the Simulink Coder User’s Guide for more information.

See “S-Functions Incorporate Legacy C Code” in the Simulink Developing S-Functions for an example.

Comparison of Custom Block Functionality

In this section...

“Custom Block Considerations” on page 27-7

“Modeling Requirements” on page 27-10

“Speed and Code Generation Requirements” on page 27-13

Custom Block Considerations

When creating a custom block, you may want to consider the following.

- Does the custom block need multiple input and output ports?
- Does the block need to model continuous or discrete state behavior?
- Will the block inputs and outputs have various data attributes, such as data types or complexity?
- How important is the effect of the custom block on the speed of updating the Simulink diagram or simulating the Simulink model?
- Do you need to generate code for a model containing the custom block?

The following two tables provide an overview of how each custom block type addresses the previous questions. More detailed information for each consideration follows these two tables.

Modeling Requirements

Custom Block Type	Supports Multiple Inputs and Outputs	Models State Dynamics	Supports Various Data Attributes
Subsystem	Yes, including bus signals.	Yes.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.
Fcn	No. Must have a single vector input and scalar output.	No.	Supports only real scalar signals with a data type of double or single.

Modeling Requirements (Continued)

Custom Block Type	Supports Multiple Inputs and Outputs	Models State Dynamics	Supports Various Data Attributes
Interpreted MATLAB Function	No. Must have a single vector input and output.	No.	Supports only n-D, real, or complex signals with a data type of double.
MATLAB Function	Yes, including bus signals.	No.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.
Level-2 MATLAB S-function	Yes.	Yes, including limited access to other S-function APIs.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.
C MEX S-function	Yes, including bus signals if using the Legacy Code Tool to generate the S-function.	Yes, including full access to all S-function APIs.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.

Speed and Code Generation Requirements

Custom Block Type	Speed of Updating the Diagram	Simulation Overhead	Code Generation Support
Subsystem	Proportional to the complexity of the subsystem. For library blocks, can be slower the first	Proportional to the complexity of the subsystem. Library	Natively supported.

Speed and Code Generation Requirements (Continued)

Custom Block Type	Speed of Updating the Diagram	Simulation Overhead	Code Generation Support
	time the library is loaded.	blocks introduce no additional overhead.	
Fcn	Very fast.	Minimal, but these blocks also provide limited functionality.	Natively supported.
Interpreted MATLAB Function	Fast.	High and incurred when calling out to the MATLAB interpreter. These calls add overhead that should be avoided if simulation speed is a concern.	Not supported.
MATLAB Function	Can be slower if code must be generated to update the diagram.	Minimal if the MATLAB interpreter is not called. Simulation speed is equivalent to C MEX S-functions when the MATLAB interpreter is not called.	Natively supported, with exceptions. See “Code Generation” on page 27-16 for more information.
Level-2 MATLAB S-function	Can be slower if the S-function overrides methods executed when updating the diagram.	Higher than for Interpreted MATLAB Function blocks because the MATLAB interpreter is called for every S-function method used. Very flexible, but very costly.	MATLAB S-functions initialized as a <code>SimViewingDevice</code> do not generate code. Otherwise, MATLAB S-functions require a TLC-file for code generation.
C MEX S-function	Can be slower if the S-function overrides methods executed	Minimal, but proportional to the complexity of the	Might require a TLC-file.

Speed and Code Generation Requirements (Continued)

Custom Block Type	Speed of Updating the Diagram	Simulation Overhead	Code Generation Support
	when updating the diagram.	algorithm and the efficiency of the code.	

Modeling Requirements

Multiple Input and Output Ports

The following types of custom blocks support multiple input and output ports.

Custom Block Type	Multiple Input and Output Port Support
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-Modifying Linked Subsystems” on page 28-7 for more information.
Fcn, Interpreted MATLAB Function	Supports only a single input and a single output port. You must use a Mux block to combine the inputs and a Demux block to separate the outputs if you need to pass multiple signals into or out of these blocks.
MATLAB Function	Supports multiple input and output ports, including bus signals. See “How Structure Inputs and Outputs Interface with Bus Signals” on page 29-83 for more information.
S-function (MATLAB or C MEX)	Supports multiple input and output ports. In addition, you can modify the number of input and output ports based on user-defined parameters. S-functions generated using the Legacy Code Tool also accept Simulink bus signals. See “Integrate C Functions Using Legacy Code Tool” for more information.

State Behavior and the S-Function API

Simulink blocks communicate with the Simulink engine through the S-function API, a set of methods that fully specifies the behavior of blocks. Each custom block type accesses a different sets of the S-function APIs, as follows.

Custom Block Type	S-Function API Support
Subsystem	Communicates directly with the engine. You can model state behaviors using appropriate blocks from the Continuous and Discrete Simulink block libraries.
Fcn, Interpreted MATLAB Function, MATLAB Function	All create an <code>mdlOutputs</code> method to calculate the value of the outputs given the value of the inputs. You cannot access any other S-function API methods using one of these blocks and, therefore, cannot model state behavior.
MATLAB S-function	Accesses a larger subset of the S-function APIs, including the methods needed to model continuous and discrete states. For a list of supported methods, see “Level-2 MATLAB S-Function Callback Methods” in “Writing S-Functions”.
C MEX S-function	Accesses the complete set of S-function APIs.

Data Attribute Support

All custom block types support real scalar inputs and outputs with a data type of double.

Custom Block Type	Data Attribute Support
Subsystem	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.
Fcn	Supports only double or single data types. In addition, the input and output cannot be complex and the output must be a scalar signal. Does not support frame-based signals.
Interpreted MATLAB Function	Supports 2-D, n-D, and complex signals, but the signal must have a data type of double. Does not support frame-based signals.
MATLAB Function	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.
S-function (MATLAB or C MEX)	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.

Speed and Code Generation Requirements

Updating the Simulink Diagram

The Simulink software updates the diagram before every simulation and whenever requested by the user. Every block introduces some overhead into the “update diagram” process.

Custom Block Type	Speed of Updating the Diagram
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 the Simulink software loads any unloaded libraries the first time the diagram is updated or readied for simulation. If all referenced library blocks remain unchanged, the Simulink software does not subsequently reload the library and compiling the model becomes faster than if the model did not use libraries.
Fcn, Interpreted MATLAB Function	Does not incur greater update cost than other Simulink blocks.
MATLAB Function	Performs simulation through code generation, so these blocks might take a significant amount of time when first updated. However, because code generation is incremental, if the block and the signals connected to it have not changed, Simulink does not repeatedly update the block.
S-function (MATLAB or C MEX)	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, the Simulink software invokes all relevant methods in the model initialization phase up to, but not including, <code>mdlStart</code> .

Simulation Overhead

For most applications, any of the custom block types provide acceptable simulation performance. Use the Simulink profiler to obtain an indication of the actual performance. See “Capture Performance Data” on page 22-32 for more information.

You can break simulation performance into two categories. The interface cost is the time it takes to move data from the Simulink engine into the block. The algorithm cost is the time needed to perform the algorithm that the block implements.

Custom Block Type	Simulation Overhead
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.
Fcn	Has the least simulation overhead. The block is tightly integrated with the Simulink engine and implements a rudimentary expression language that is efficiently interpreted.
Interpreted MATLAB Function	<p>Has a higher interface cost than most blocks and the same algorithm cost as a MATLAB function.</p> <p>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, this overhead can be substantial.</p> <p>Once the data has been converted, the MATLAB interpreter executes the algorithm. As a result, the algorithm cost is the same as for MATLAB function. Efficient code can be competitive with C code if MATLAB is able to optimize it, or if the code uses the highly optimized MATLAB library functions.</p>

Custom Block Type	Simulation Overhead
MATLAB Function	<p>Performs simulation through code generation and so incurs the same interface cost as standard blocks.</p> <p>The algorithm cost of this block is harder to analyze because of the block's implementation. On average, a function for this block and a MATLAB function run at about the same speed. To further reduce the algorithm cost, you can disable debugging for all the MATLAB Function blocks in your model.</p> <p>If the MATLAB Function block uses simulation-only capabilities to call out to the MATLAB interpreter, it incurs all the costs that a MATLAB S-function or Interpreted MATLAB Function block incur. Calling out to the MATLAB interpreter from a MATLAB Function block produces a warning to prevent you from doing so unintentionally.</p>
MATLAB S-function	<p>Incurs the same algorithm costs as the Interpreted MATLAB Function block, but with a slightly higher interface cost. Because 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 interpreter for each block method you implement whereas for the Interpreted MATLAB Function block, it calls the MATLAB interpreter only for the <code>mdlOutputs</code> method.</p>
C MEX S-function	<p>Simulates via the compiled code and so incurs the same interface cost as standard blocks. The algorithm cost depends on the complexity of the S-function.</p>

Code Generation

Not all custom block types support code generation with Simulink Coder.

Custom Block Type	Code Generation Support
Subsystem	Supports code generation.
Fcn	Supports code generation.
Interpreted MATLAB Function	Does not support code generation.
MATLAB Function	Supports code generation. However, if your MATLAB Function block calls out to the MATLAB interpreter, it will build with the Simulink Coder product only if the calls to the MATLAB interpreter 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.
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 interpretive mode by calling back to the MATLAB interpreter without implementing the algorithm in TLC. If the MATLAB S-function is a <code>SimViewingDevice</code> , the Simulink Coder product automatically omits the block during code generation.
C MEX S-function	Supports code generation. For noninlined S-functions, the Simulink Coder product uses the C MEX function during code generation. However, you must write a TLC-file for the S-function if you need to either inline the S-function or create a wrapper for hand-written code. See “Insert S-Function Code” in the Simulink Coder User’s Guide for more information.

Expanding Custom Block Functionality

You can expand the functionality of any custom block using callbacks and Handle 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 “Create Block Callbacks” on page 4-64.

GUIDE, the MATLAB graphical user interface development environment, provides tools for easily creating custom user interfaces. See “GUI Building” for more information on using GUIDE.

Create a Custom Block

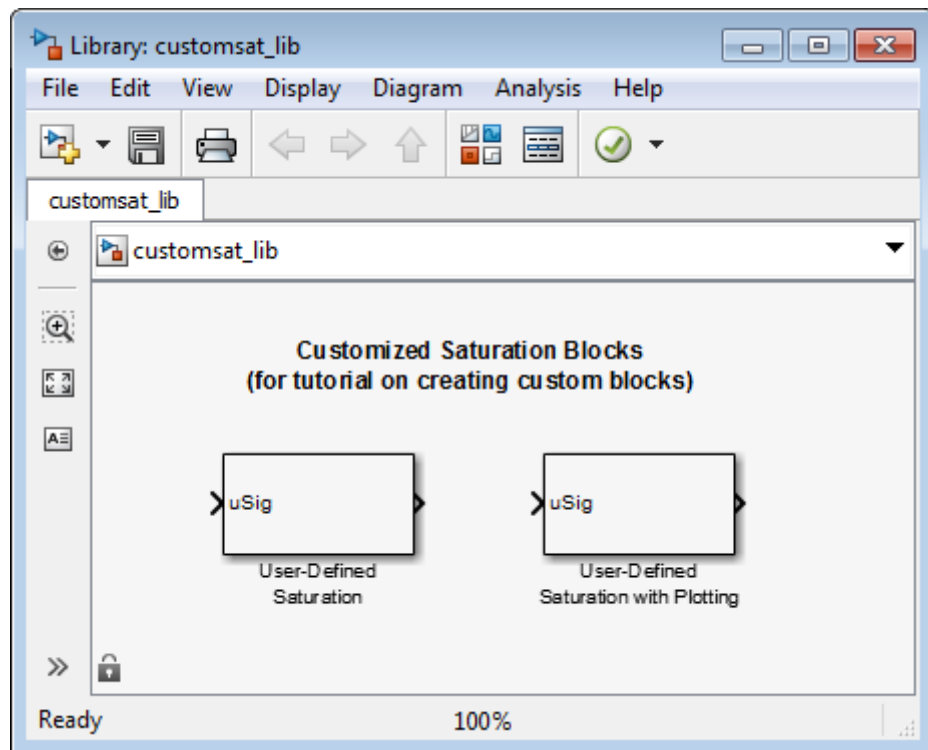
In this section...
“How to Design a Custom Block” on page 27-18
“Defining Custom Block Behavior” on page 27-20
“Deciding on a Custom Block Type” on page 27-21
“Placing Custom Blocks in a Library” on page 27-27
“Adding a User Interface to a Custom Block” on page 27-29
“Adding Block Functionality Using Block Callbacks” on page 27-38

How to Design a Custom Block

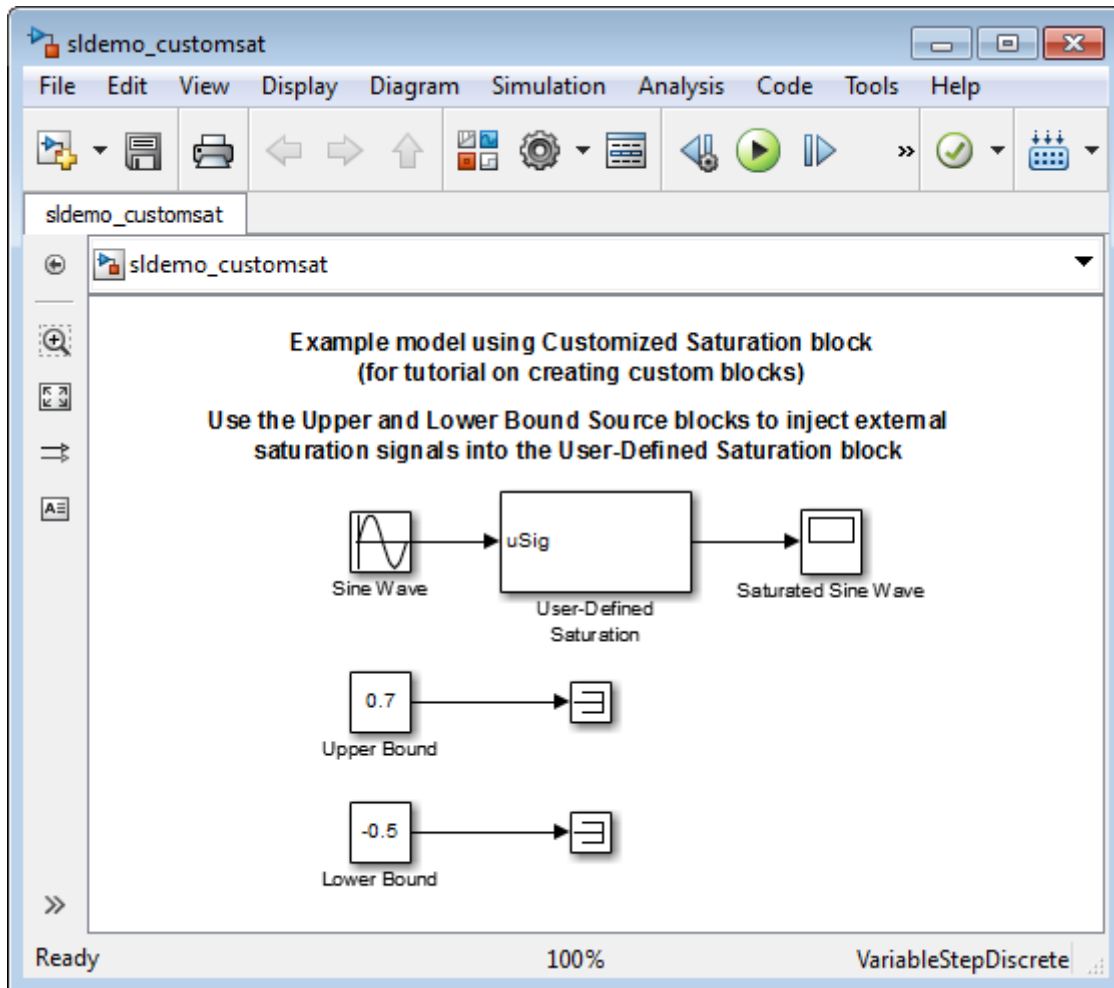
In general, use the following process to design a custom block:

- 1 “Defining Custom Block Behavior” on page 27-20
- 2 “Deciding on a Custom Block Type” on page 27-21
- 3 “Placing Custom Blocks in a Library” on page 27-27
- 4 “Adding a User Interface to a Custom Block” on page 27-29

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 `customsat_lib` contains the two versions of the customized saturation block.



The example model `s1demo_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 27-7 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
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

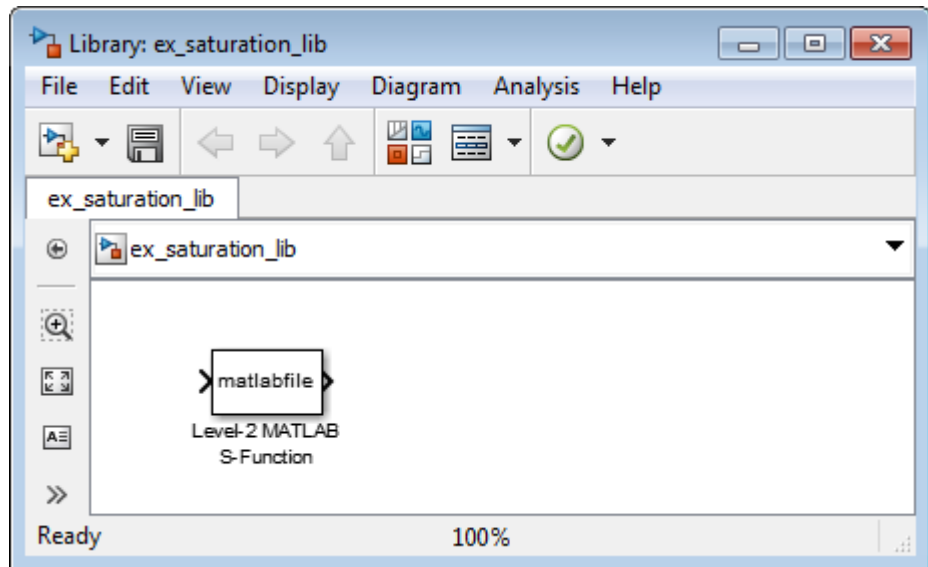
block.OutputPort(1).Data = sigVal;
```

```
%end Outputs function
```

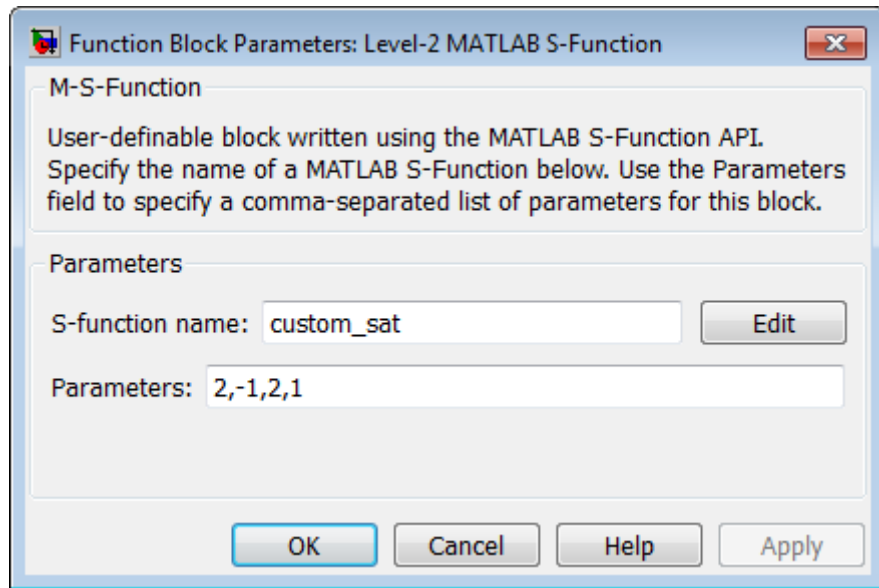
Placing Custom Blocks in a Library

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 Library Browser, select **File > New > Library**.
- 2 From the User-Defined Functions library, drag a Level-2 MATLAB S-Function block into your new library.

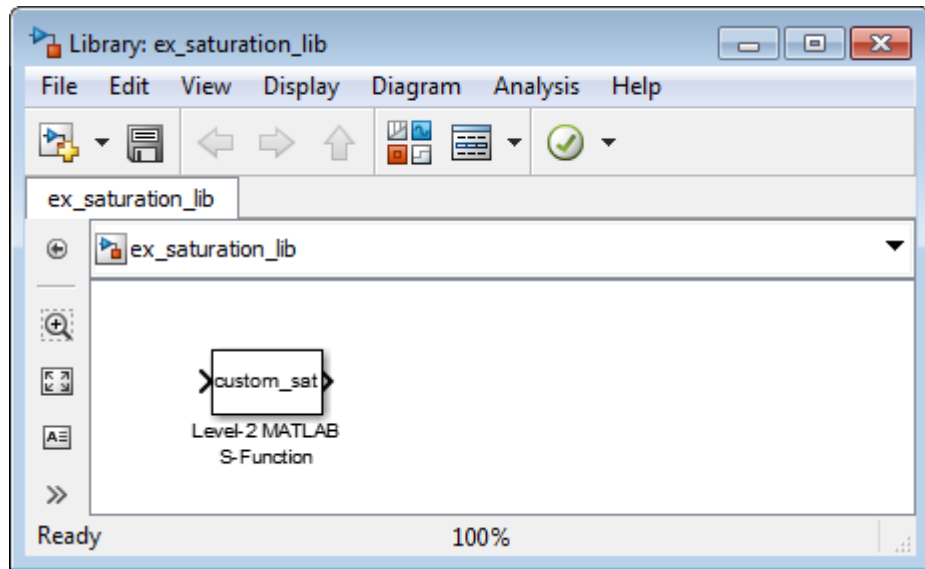


- 3 Save your library with the filename `saturation_lib`.
- 4 Double-click the block to open its Function Block Parameters dialog box.
- 5 In the **S-function name** field, enter the name of the S-function. For example, enter `custom_sat`. In the **Parameters** field enter `2, -1, 2, 1`.



6 Click **OK**.

You have created a custom saturation block that you can share with other users.



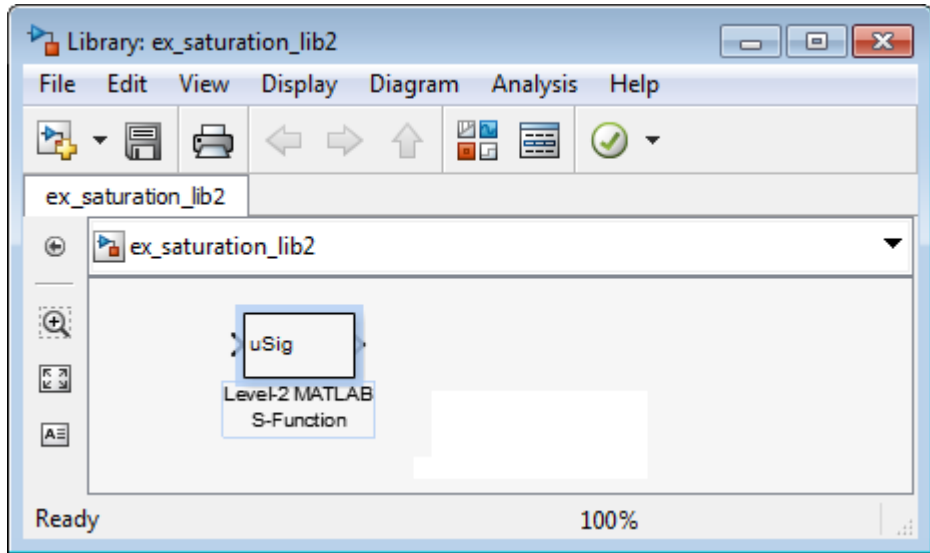
You can make the block easier to use by adding a customized user interface.

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.



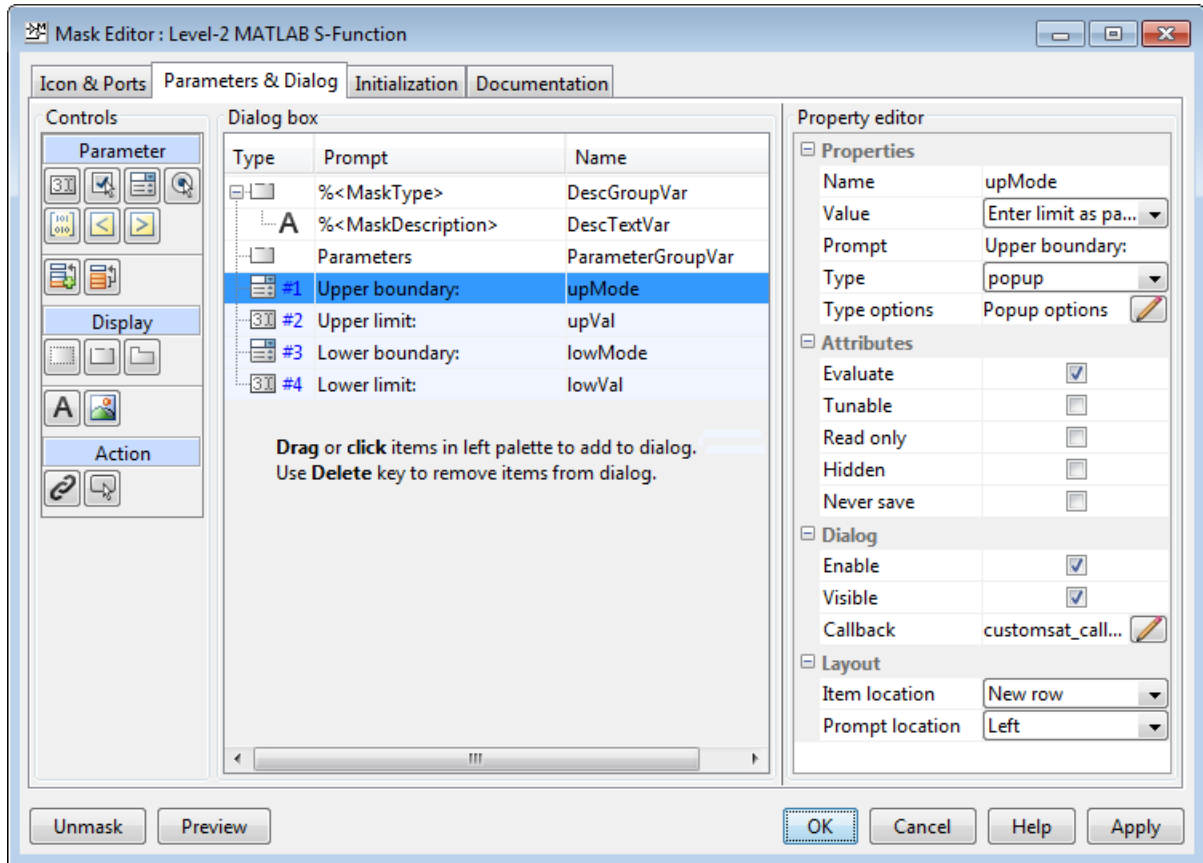
4 In the **Parameters & Dialog** pane, add four parameters corresponding to the four S-Function parameters. For each new parameter, drag a popup or edit control to the **Dialog box** section, as shown in the table. Drag each parameter into the Parameters group.

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	customsat_callback('lowerbound_callback_gcb')
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 string 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.

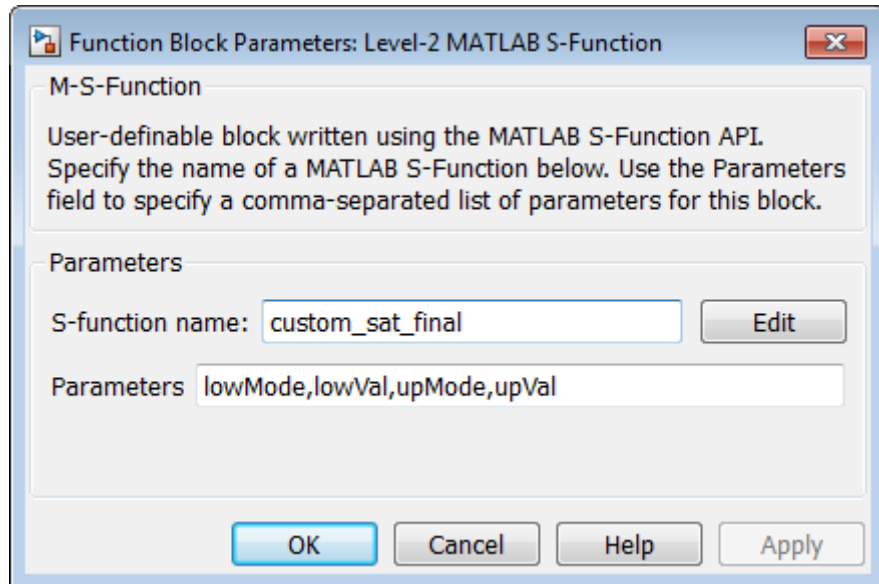


- 5 In the **Initialization** pane, select the **Allow library block to modify its contents** check box. This setting allows the S-function to change the number of ports on the block.
- 6 In the **Documentation** pane:
 - In the **Mask type** field, enter
Customized Saturation
 - In the **Mask description** field, enter
Limit the input signal to an upper and lower saturation value

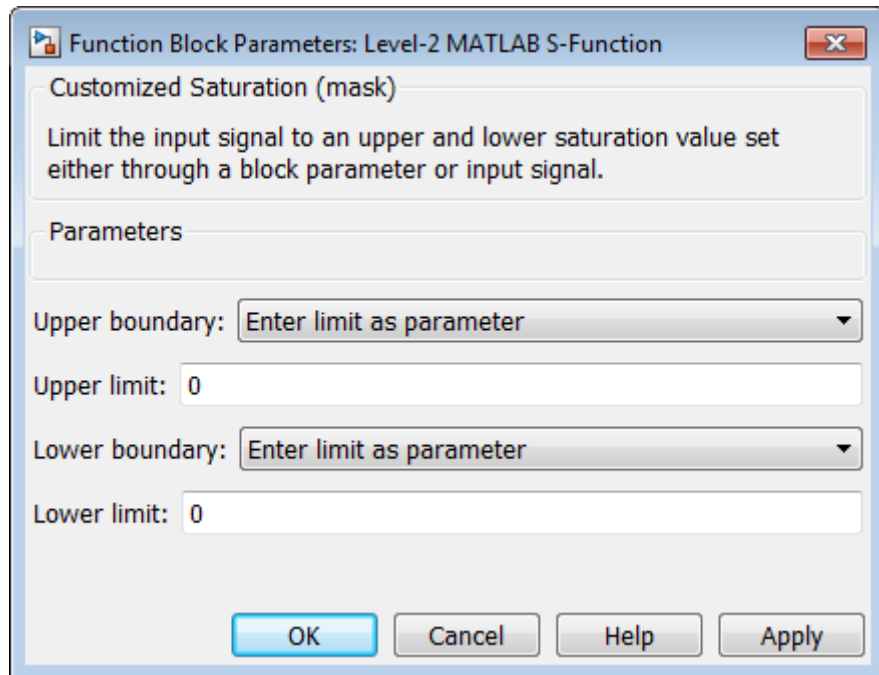
set either through a block parameter or input signal.

- 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 Handle Graphics user interface on top of the masked block. The block `OpenFcn` invokes the Handle 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 `MaskVisibilities`. This call returns a cell array of strings 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 string.

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

```

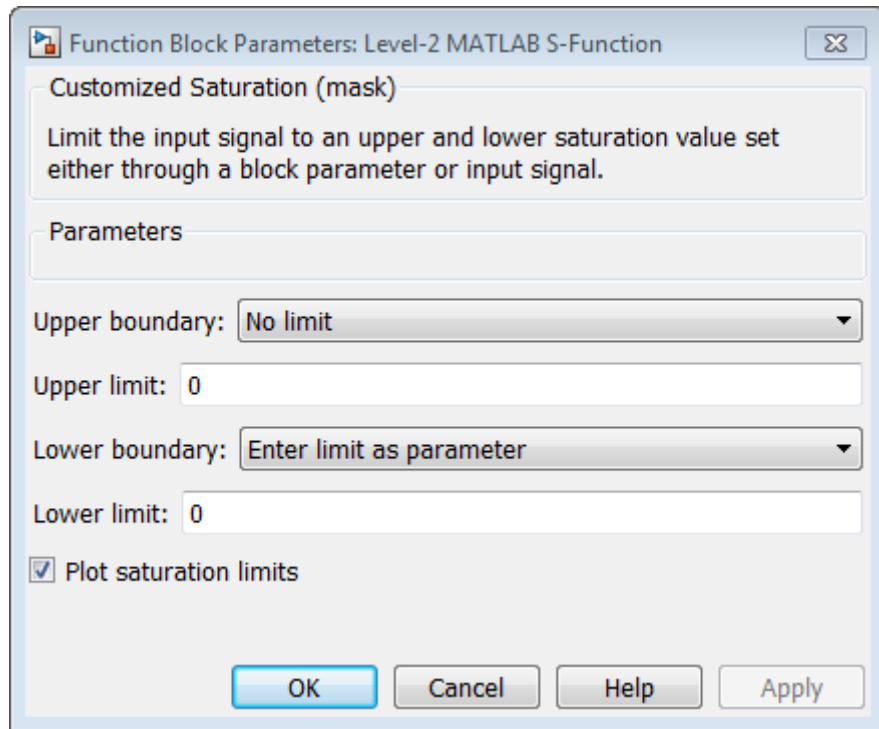
Adding Block Functionality Using Block Callbacks

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	checkbox	No	NA	customsat_callback('plotsaturation')

- 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(gcf, '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.

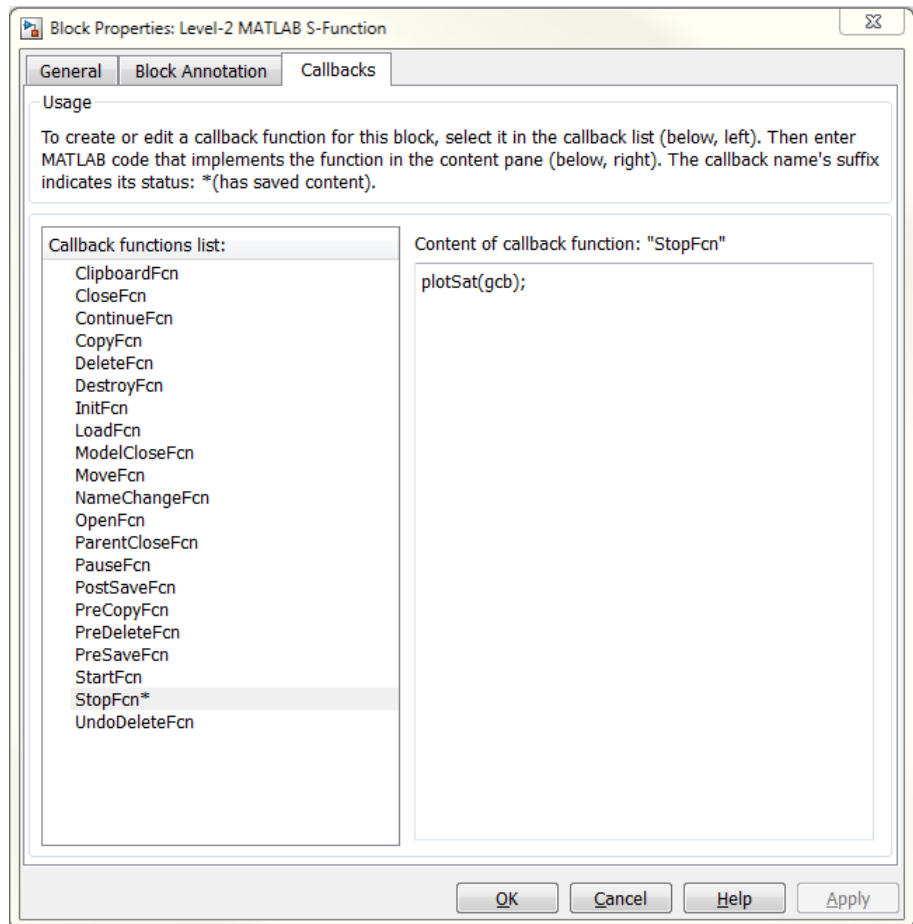
ud = get_param(block,'UserData');

```

```
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**.



Custom Block Examples

In this section...

“Creating Custom Blocks from Masked Library Blocks” on page 27-44

“Creating Custom Blocks from MATLAB Functions” on page 27-44

“Creating Custom Blocks from S-Functions” on page 27-45

Creating Custom Blocks from Masked Library Blocks

The Additional Math and Discrete Simulink library is a group of custom blocks created by extending the functionality of built-in Simulink blocks. The Additional Discrete library contains a number of masked blocks that extend the functionality of the standard Unit Delay block. See “Libraries” for more general information on Simulink libraries.

Creating Custom Blocks from MATLAB Functions

The Simulink product provides a number of examples that show how to incorporate MATLAB functions into a custom block.

- The Single Hydraulic Cylinder Simulation, `sldemo_hydcyl`, uses a Fcn block to model the control valve flow. In addition, the Control Valve Flow block is a library link to one of a number of custom blocks in the library `hydlib`.
- The Radar Tracking Model, `sldemo_radar`, uses an Interpreted MATLAB Function block to model an extended Kalman filter. The MATLAB function `aero_extkalman.m` implements the Kalman filter found inside the Radar Kalman Filter subsystem. In this example, the MATLAB function requires three inputs, which are bundled together using a Mux block in the Simulink model.
- The Spiral Galaxy Formation example, `sldemo_eml_galaxy`, uses several MATLAB Function blocks to construct two galaxies and calculate the effects of gravity as these two galaxies nearly collide. The example also uses MATLAB Function blocks to plot the simulation results using a subset of MATLAB functions not supported for code generation. However, because these MATLAB Function blocks have no outputs, the Simulink Coder product optimizes them away during code generation.

Creating Custom Blocks from S-Functions

The Simulink model `sfundemos` contains various examples of MATLAB and C MEX S-functions. For more information on writing MATLAB S-functions, see “Write Level-2 MATLAB S-Functions”. For more information on writing C MEX S-functions, see “C/C++ S-Functions”. For a list of available S-function examples, see “S-Function Examples” in Writing S-Functions.

Working with Block Libraries

- “About Block Libraries and Linked Blocks” on page 28-2
- “Create and Work with Linked Blocks” on page 28-4
- “Work with Library Links” on page 28-8
- “Create Block Libraries” on page 28-20
- “Add Libraries to the Library Browser” on page 28-32

About Block Libraries and Linked Blocks

Block Libraries

A *block library* is a collection of blocks that serve as prototypes for instances of blocks in a Simulink model.

Simulink comes with two built-in block libraries: the Simulink block library and the Simulink Coder block library. The latter is included with Simulink to support sharing models that contain Simulink Coder blocks. Many additional MathWorks products and associated block libraries are available. You cannot change a built-in block library in any way.

Benefits of Block Libraries

Block libraries are a useful componentization technique for:

- Providing frequently-used, and seldom changed, modeling utilities
- Reusing components repeatedly in a model or in multiple models

For additional information about how libraries compare to other Simulink componentization techniques, see “Componentization Guidelines” on page 12-17.

Library Browser

Simulink provides a Library Browser that you can use to display block libraries, search for blocks by name, and copy library blocks into models. All installed libraries appear in the Library Browser when you open it. See “Populate a Model” on page 4-4 for information about how to use the Simulink Library Browser.

Linked Blocks

When you copy a block from a library into a model, Simulink creates a *linked block* in the model, and connects it to the library block using a *library link*. The library block is the *prototype block*, and the linked block in the model is an *instance* of the library block. The linked block appearance and behavior

are the same as the library block. For most purposes, you can ignore the underlying link and just think of a linked block as a clone of the library block.

Copying a block from a library to another library or model does not always create a linked block. Library blocks that support linking include subsystems, masked blocks, and charts. However, the block author can choose not to make the copy a linked block by modifying the `CopyFcn`, as done by the Simulink Subsystem block.

Create and Work with Linked Blocks

In this section...

“About Linked Blocks” on page 28-4

“Create a Linked Block” on page 28-4

“Update a Linked Block” on page 28-5

“Modify Linked Blocks” on page 28-6

“Find a Linked Block’s Prototype” on page 28-7

“Find Linked Blocks in a Model” on page 28-7

About Linked Blocks

A *linked block* is an instance of a library block and contains a link to that library block that serves as the block type’s prototype. The link consists of the path of the library block that serves as the instance’s prototype. The link allows the linked block to update whenever the corresponding prototype in the library changes (“Update a Linked Block” on page 28-5). This ensures that your model always uses the latest version of the block.

Note The data tip for a linked block shows the name of the library block it references (see “Block Tool Tips” on page 23-2).

You can change the values of a linked block’s parameters (including in an existing mask). You cannot add a new mask for linked blocks or edit the mask setup, that is, add or remove mask parameters or change mask behavior.

Also, you cannot set callback parameters for a linked block. If the linked block’s prototype is a subsystem, you can make nonstructural changes to the contents of the linked subsystem (see “Modify Linked Blocks” on page 28-6).

Create a Linked Block

To create a linked block in a model or another library:

- 1 Open your model.
- 2 Open the Simulink Library Browser (see “Copy Blocks to Your Model” on page 4-4), or another library.
- 3 Use the Library Browser to find the library block that serves as a prototype of the block you want to create (see “Browse Block Libraries” on page 4-4 and “Search Block Libraries” on page 4-5).
- 4 Drag the library block from the Blocks pane and drop it into your model.

Update a Linked Block

Simulink updates out-of-date linked blocks in a model or library when you:

- Load the model or library.
- Run the simulation.
- Use the `find_system` command.
- Query the `LinkStatus` parameter of a block, using the `get_param` command (see “Check and Set Link Status Programmatically” on page 28-16).

Note Querying the `StaticLinkStatus` parameter of a block does not update any out-of-date linked blocks.

- Save changes to a library block, then Simulink automatically refreshes all links to the block in open Model Editor windows.

When you edit a library block (in the Model Editor or at the command line), then Simulink indicates stale links which are open in the Model Editor by displaying the linked blocks grayed out. Simulink refreshes any stale links to edited blocks when you activate the Model Editor window, even if you have not saved the library yet.

To manually refresh links:

- Select **Simulation > Update Diagram** (or press **Ctrl+D**).
- Select **Diagram > Refresh Blocks** (or press **Ctrl+K**) to refresh links.

- Select **Go To Library Link**.

Modify Linked Blocks

You cannot make structural changes to linked blocks, such as adding or deleting lines or blocks to the block diagram of a masked subsystem. If you want to make such changes, you must disable the linked block's link to its library prototype (see “Disable Links to Library Blocks” on page 28-11).

Parameterized Links

If you change parameter values inside a linked block, you create a *parameterized link*. You can change the values of any masked subsystem linked block parameter that does not alter the block's structure, e.g., by adding or deleting lines, blocks, or ports. An example of a nonstructural change is a change to the value of a mathematical block parameter, such as the Gain parameter of the Gain block. A linked subsystem block whose parameter values of inner blocks differ from their corresponding library blocks is called a *parameterized link*. Changing the top-level mask values does not create a parameterized link.

When saving a model containing a parameterized link, Simulink saves the changes to the local copy of the subsystem together with the path to the library copy in the model's file. When you reopen the system, Simulink copies the library block into the loaded model and applies the saved changes.

Tip To determine whether a linked block's parameter values differ from those of its library prototype, open the linked block's block diagram in an editor window. The linked block's library link indicator (if displayed) changes to a red arrow and the title bar of the editor window displaying the subsystem displays “Parameterized Link” if the linked block's parameter values differ from the library block's parameter values.

See “Display Library Links” on page 28-8.

Self-Modifying Linked Subsystems

Simulink allows linked subsystems to change their own structural contents without disabling the link. This allows you to create masked subsystems that modify their structural contents based on mask parameter dialog box values.

Find a Linked Block's Prototype

To find the source library and the prototype of a linked block, right-click the linked block and select **Library Link > Go To Library Link**.

Alternatively, select the linked block and select **Diagram > Library Link > Go To Library Link**.

If the library is open, Simulink selects and highlights the library block and makes the source library the active window. If the library is not open, Simulink first opens it and then selects the library block.

Find Linked Blocks in a Model

Use the `libinfo` command to get information about the linked blocks in the model and which library blocks they link to. The `ReferenceBlock` property gives the path of the library block to which a block links.

Work with Library Links

In this section...

- “Display Library Links” on page 28-8
- “Lock Links to Blocks in a Library” on page 28-9
- “Disable Links to Library Blocks” on page 28-11
- “Restore Disabled or Parameterized Links” on page 28-12
- “Check and Set Link Status Programmatically” on page 28-16
- “Break a Link to a Library Block” on page 28-18
- “Fix Unresolved Library Links” on page 28-19

Display Library Links





A model can have a block linked to a library block, or it can have a local instance of a block that is not linked. To enable the display of library links:

- 1 In the Model Editor window, select **Display > Library Links** and from the submenu, select one of these options:
 - a **None** — displays no links
 - b **Disabled** — displays only disabled links (the default for new models)
 - c **User Defined** — displays only links to user libraries
 - d **All** — displays all links
- 2 Observe the library link indicators.

The library link indicator is a badge in the bottom left corner of each 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. If you open a linked block, the Model Editor displays the same link badge at bottom left. You can right-click the link badge in the corner of the canvas to access link options such as **Go To Library Block**.

Link Badge	Status
Black links 	Active link
Grey separated links 	Inactive link
Black links with a red star icon 	Active and modified (parameterized link)
White links, black background 	Locked link

Note If you have a variant subsystem block inside a link block, modifying the parameters on the variant subsystem creates a link data on the topmost link block. If the link badge is visible, presence of link data is indicated by a red star on the link badge.

Lock Links to Blocks in a Library

You can lock links to a library. Lockable library links enable control of end user editing, to prevent unintentional disabling of these links. This ensures robust usage of mature stable libraries.

To lock links to a library, either:

- In your library window, select **Diagram > Lock Links To Library**.
- At the command line, use the `LockLinksToLibrary` property:

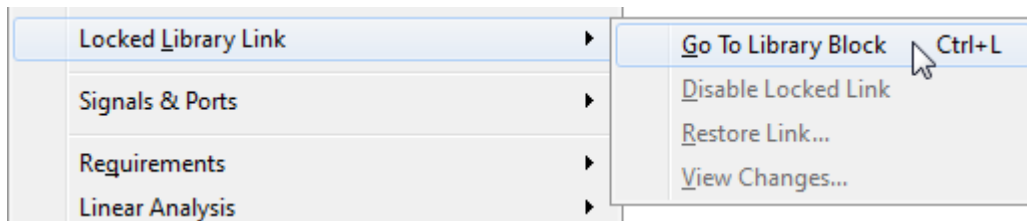
```
set_param('MyModelName', 'LockLinksToLibrary', 'on')
```

where `MyModelName` is the library file.

When you copy a block to a model from a library with locked links:

- The link is locked.
- You cannot disable locked links from the Model Editor.

If you select **Diagram** or right-click the linked block, you see the **Library Link** menu has changed to **Locked Library Link**, and the only enabled option is now **Go To Library Block**.



- If you display library links, the locked link icon has a black background.



- If you open a locked link, the window title is **Locked Link: *blockname***. The bottom left corner shows a lock icon and a link badge.



- You cannot edit locked link contents. If you try to make a structural change to a locked link (such as editing the diagram), you see a message stating 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.
- You can disable locked links only from the command line as follows:

```
set_param(gcb, 'LinkStatus', 'inactive')
```

To unlock links to a library:

- In your library window, select **Diagram > Unlock Links To Library**
- At the command line:

```
set_param('MyModeName', 'LockLinksToLibrary', 'off')
```

The status of a link (locked or not) is determined by the library state when you copy the block. 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 links.

If you use sublibraries as an organizational tool, when you lock links to a library, you might want also to lock links to any sublibraries.

Disable Links to Library Blocks

To make a structural change to a linked block, you need to disable the link between the block and the library block that serves as its prototype.

You cannot disable *locked* links from the Model Editor. See “Lock Links to Blocks in a Library” on page 28-9.

Note When you use the Model Editor to make a structural change (such as editing the diagram) to a block with an active library link, Simulink offers to disable the library link for you (unless the link is locked). If you accept, Simulink disables the link and allows you to make changes to the subsystem block.

Do not use `set_param` to make a structural change to an active link; the result of this type of change is undefined.

To disable a link:

- 1** In the Model Editor window, right-click a linked block and select **Library Link > Disable Link**.
- 2** Alternatively, select a linked block and select the menu item **Diagram > Library Link > Disable Link**.

The library link is disabled and the library link indicator changes to gray. When a library block is disabled and it is within another library block (a child of a parent library block), the model also disables the parent block containing the child block.

To disable a link from the command-line, set the `LinkStatus` property to `inactive` as follows:

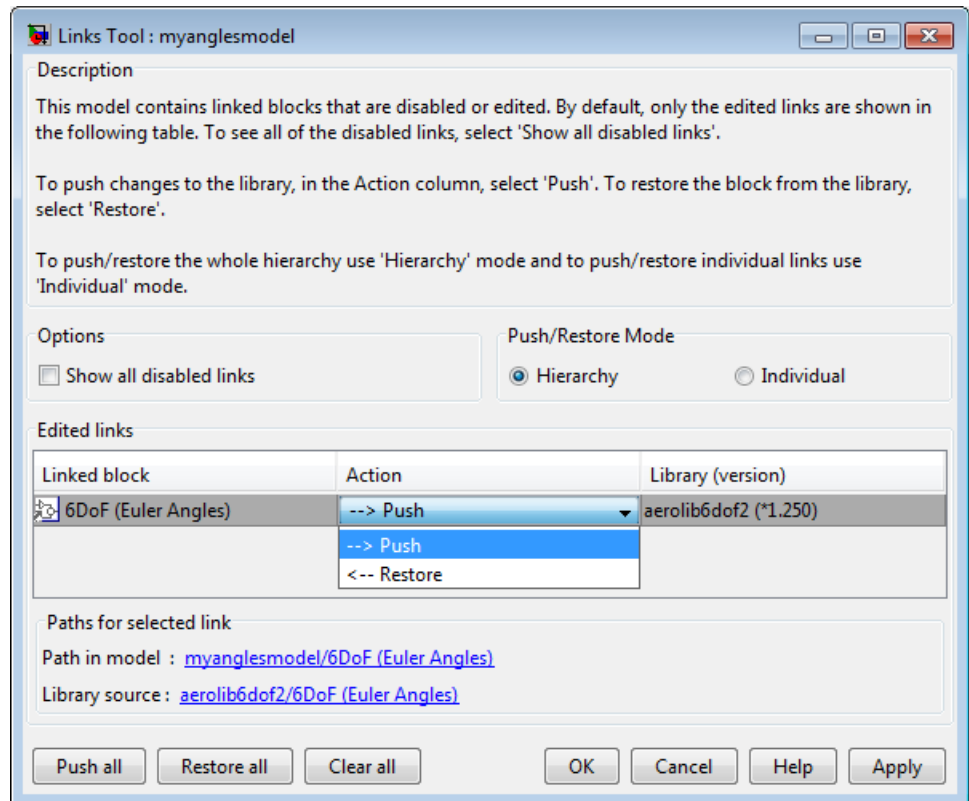
```
set_param(gcb, 'LinkStatus', 'inactive')
```

Restore Disabled or Parameterized Links

After you make changes to a disabled linked block, you may want to restore its link to the library block and resolve any differences between the two blocks. The Links Tool helps you with this task.

- 1** In the Model Editor window, select a linked block with a disabled library link.
- 2** From the **Diagram** menu (or right-click context menu), select **Library Link > Resolve Link**.

The Links Tool window opens.



The **Edited links** table has the following columns:

- **Linked block** — List of linked blocks. The list of links includes library links with structural changes (disabled links), parameterized library links (edited links), and library links that were actively chosen to be resolved.
 - **Action** — Select an action to perform on the linked block or library.
 - **Library** — List of library names and version numbers.
- 3** Select the check box **Show all disabled links** if you want to view disabled links as well as parameterized links.

4 Under **Push/Restore Mode**, choose a mode of action:

- If you want to act on individual links, select **Individual**.
- If you want to act on the whole link hierarchy, leave the default setting on **Hierarchy**. See “Pushing or Restoring Link Hierarchies” on page 28-15.

5 From the **Linked block** list, select a block name.

The Links Tool updates the **Paths for selected link** panel with links to the linked block in the model and in the library.

6 From the **Action** list, select **Push** or **Restore** for the currently selected block.


Action Choice	Links Tool Action
Push	The Links Tool looks for all changes in the link hierarchy and pushes all links with changes to their libraries. Push replaces the version of the block in the library with the version in the model.
Restore	The Links Tool looks for all disabled or edited links in the link hierarchy and restores them all with their corresponding library blocks. Restore replaces the version of the block in the model with the version in the library.
Push Individual	In Individual mode, the disabled or edited block is pushed to the library, preserving the changes inside it without acting on the hierarchy. All other links are unaffected.
Restore Individual	In Individual mode, the disabled or edited block is restored from the library, and all other links are unaffected.

To select the same action for all linked blocks, click **Push all**, **Restore all**, or **Clear all**.

- 7 When you click **OK** or **Apply**, the Links Tool performs the push or restore actions you selected in the edited links table.

After resolving a link, the versions in the library and the linked block now match.

Note Changes you push to the library are not saved until you actively save the library.

If a linked block name has a cautionary icon  before it, the model has other instances of this block linked from the same library block, and they have different changes. Choose one of the instances to push changes to the library block and restore links to the other blocks, or choose to restore all of them with the library version.

Pushing or Restoring Link Hierarchies

Caution Be cautious using Push or Restore in hierarchy mode if you have a large hierarchy of edited and disabled links. Ensure that you want to push or restore the whole hierarchy of links.

Pushing a hierarchy of disabled links affects the disabled links inside and outside in 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 without changes. This operation does not affect outside (parent) links with changes unless you also explicitly selected them for push. The Links Tool starts from the lowest links (the deepest inside) and then moves upward in the hierarchy.

Some simple 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 Links Tool can manipulate only the top level.

Check and Set Link Status Programmatically

All blocks have a `LinkStatus` parameter and a `StaticLinkStatus` parameter that indicate whether the block is a linked block.

Use `get_param(gcb, 'StaticLinkStatus')` to query the link status without updating out-of-date linked blocks.

Use `get_param` and `set_param` to query and set the `LinkStatus`, which can have the following values.

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. For example, suppose that A is a link to a subsystem in a library that contains a Gain block. Further, suppose that you open A and select the Gain block. Then, <code>get_param(gcb, 'LinkStatus')</code> returns <code>implicit</code> .
inactive	Disabled link.

Set LinkStatus Value	Description
none	Breaks link. Use none to break a link, e.g., <code>set_param(gcb, 'LinkStatus', 'none')</code>
inactive	Disables link. Use Inactive to disable a link, e.g., <code>set_param(gcb, 'LinkStatus', 'inactive')</code>
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, <code>set_param(gcb, 'LinkStatus', 'restore')</code> replaces the selected block with a link to a library block of the same type, discarding any changes in the local copy of the library block. This 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. This is equivalent to Push Individual in the Links Tool.
restoreHierarchy	Restores all disabled links in the hierarchy with their corresponding library blocks. This is equivalent to Restore in hierarchy mode in the Links Tool.
propagateHierarchy	Pushes all links with changes in the hierarchy to their libraries. This is equivalent to Push in hierarchy mode in the Links Tool. See “Restore Disabled or Parameterized Links” on page 28-12.

Note Using `get_param` to query a block’s `LinkStatus` also resolves any out-of-date block links. Use `get_param` to update library links in a model programmatically. Querying the `StaticLinkStatus` property does not resolve any out-of-date links. Query the `StaticLinkStatus` property when the call to `get_param` is in the callback of a child block querying the link status of its parent.

If you call `get_param` on a block inside a library link, Simulink resolves the link if necessary. This operation may involve loading part of the library and executing callbacks.

Break a Link to a Library Block

You can break the link between a linked block and its library block to cause the linked block to become a simple copy of the library block, unlinked to the library block. Changes to the library block no longer affect the block. Breaking links to library blocks may enable you to transport a masked subsystem model as a standalone model, without the libraries (see “Masking”).

To break the link between a linked block and its library block, you can use any of the following actions.

- Disable the link, then right-click the block and choose **Library Link > Break Link**.
- At the command line, change the value of the `LinkStatus` parameter to 'none' using this command:

```
set_param(gcb, 'LinkStatus', 'none')
```

- Right-click and drag to copy a block, and you see an offer to break links, unless the parent library has `LockLinksToLibrary` set to on. If your copied block will be a locked link, then you do not see the option to break links.

To copy and break links to multiple blocks simultaneously, select multiple blocks and then drag. Any locked links are ignored and not broken.

- When saving the model, you can break links by supplying arguments to the `save_system` command. See `save_system`.

Note Breaking library links in a model does not guarantee that you can run the model standalone, especially if the model includes blocks from third-party libraries or optional Simulink blocksets. 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. Further, breaking a link can cause a model to fail when you install a new version of the library on a system.

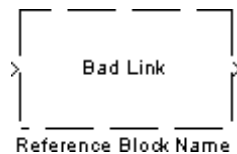
For example, suppose a block invokes a function that is supplied with the library. Now suppose that a new version of the library eliminates the function. Running a model with an unlinked copy of the block results in invocation of a now nonexistent function, causing the simulation to fail. To avoid such problems, you should generally avoid breaking links to libraries.

Fix Unresolved Library Links

If Simulink is unable to find either the library block or the source library on your MATLAB path when it attempts to update the linked block, the link becomes *unresolved*. Simulink issues an error message and displays these blocks using red dashed lines. The error message is

```
Failed to find block "source-block-name"
in library "source-library-name"
referenced by block
"linked-block-path".
```

The unresolved linked block appears like this (colored red).



To fix a bad link, you must do one of the following:

- Delete the unresolved block and copy the library block back into your model.
- Add the folder that contains the required library to the MATLAB path and select either **Simulation > Update Diagram** or **Diagram > Refresh Blocks**.
- Double-click the unresolved block to open its dialog box (see the Bad Link block reference page). On the dialog box that appears, correct the path name in the **Source block** field and click **OK**.

Create Block Libraries

In this section...

“Create a Library” on page 28-20

“Create a Sublibrary” on page 28-21

“Modify and Lock Libraries” on page 28-21

“Make Backward-Compatible Changes to Libraries” on page 28-22

Create a Library

You can create your own block library and add it to the Simulink Library Browser (see “Add Libraries to the Library Browser” on page 28-32). To create a library:

- 1 Select **Library** from the **New** submenu of the **File** menu.

Simulink creates a model file for storing the new library and displays the file in a new model editor window.

Tip You can also use the `new_system` command to create the library and the `open_system` command to open the new library.

- 2 Drag blocks from models or other libraries into the new library.

Note If you want to be able to create links in models to a block in the library, you may need to provide a mask (see “Masking”) for the block. You can also provide a mask for a subsystem in a library, but you do not need to have a mask to create links to it in models.

- 3 Save the library’s file under a new name.

Create a Sublibrary

If your library contains many blocks, consider grouping the blocks into a hierarchy of sublibraries. Creating a sublibrary entails inserting a reference in the Simulink model file of one library to the model file of another library. The referenced file is called a *sublibrary* of the parent (i.e., referencing) library. The sublibrary is said to be included by reference in the parent library.

To include a library in another library as a sublibrary:

- 1 Open the parent library.
- 2 Add a Subsystem block to the parent library.
- 3 Delete the subsystem's default input and output ports.
- 4 Create a mask for the subsystem that displays text or an image that conveys the sublibrary's purpose.
- 5 Set the subsystem's OpenFcn parameter to the name of the sublibrary's model file.
- 6 Save the parent library.

Modify and Lock Libraries

When you open a library, it is automatically locked and you cannot modify its contents. To unlock the library, select **Diagram > Unlock Library**.

When you close the library window, Simulink locks the library.

Locking a library prevents a user from inadvertently modifying a library, for example, by moving a block in the library or adding or deleting a block from the library. If you attempt to modify a locked library, Simulink displays a dialog box that allows you to unlock the library and make the change.

To unlock a block library from the MATLAB command line, use the following command:

```
set_param('library_name', 'Lock', 'off');
```

You must then relock the library from the MATLAB command line to prevent further changes. Use the following command to relock a block library:

```
set_param('library_name', 'Lock', 'on');
```

If you want to control end user editing of linked blocks and prevent unintentional disabling of links, you can lock links to a library. See “Lock Links to Blocks in a Library” on page 28-9.

When you save a library, Simulink checks file permissions and offers to try to make the library writable if necessary.

Make Backward-Compatible Changes to Libraries

Simulink provides the following features to facilitate making changes to library blocks without invalidating models that use the library blocks.

Forwarding Tables

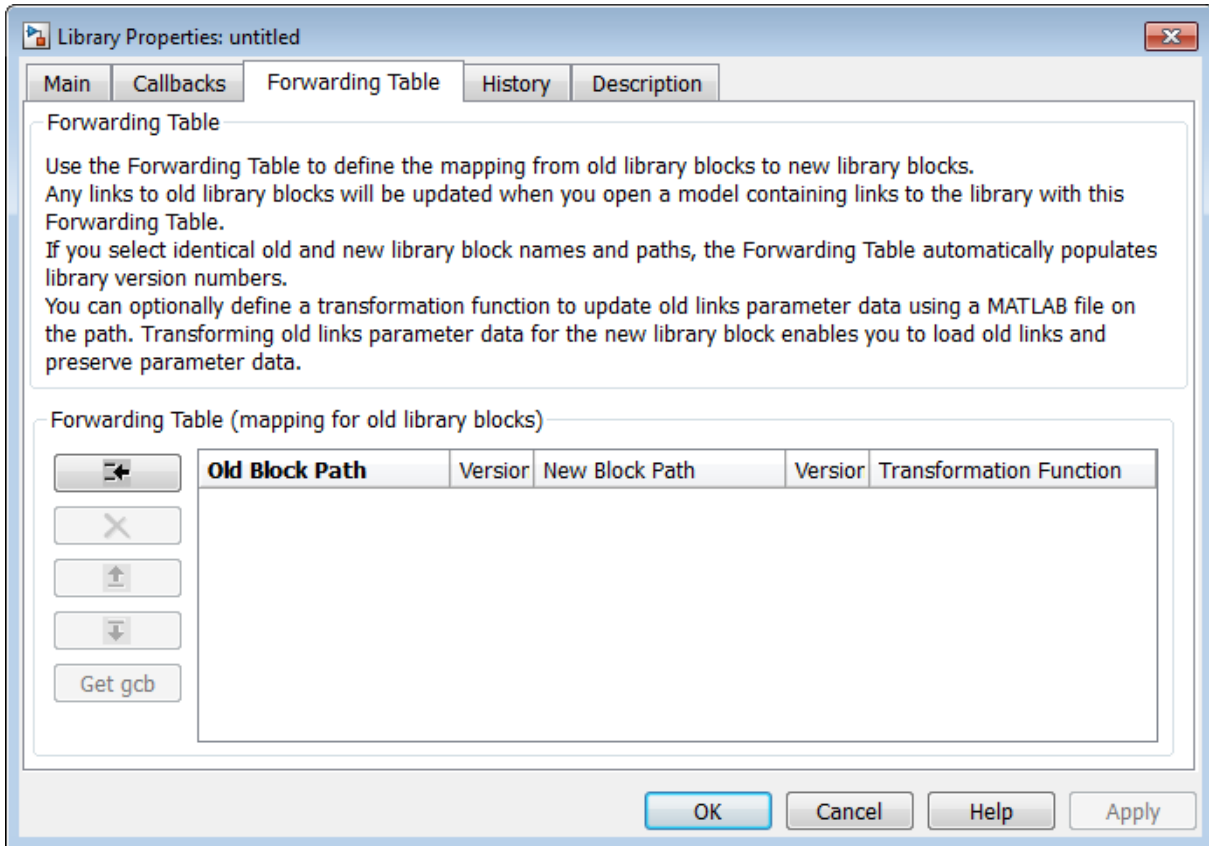
You can create forwarding tables for libraries to specify how to update links in models to reflect changes in the parameters. Use the Forwarding Table to map old library blocks to new library blocks. For example, if you rename or move a block in a library, you can use a forwarding table to enable Simulink to update models that link to the block.

After you specify the forwarding table entry in a library, any links to old library blocks will be updated when you load a model containing links to old blocks. Library authors can use the forwarding tables to automatically transform old links into updated links without any loss of functionality and data. Use the forwarding table to solve compatibility issues with models containing old links that cannot load in the current version of Simulink. Library authors do not need to run `supdate` to upgrade old links, and can reduce maintenance of legacy blocks.

To set up a forwarding table for a library,

- 1 Select **Diagram > Unlock Library**.**
- 2 Select **File > Library Properties > Library Properties**.** The Library Properties dialog box opens.

3 Select the Forwarding Table tab.



4 Define the mapping from old library blocks to new library blocks.

- a Click the Add New Entry button. A new row appears in the table.
- b Enter values in **Old Block Path** and **New Block Path** columns.

Click **GCB** to get the path of the currently selected block.

If you select identical old and new library block names and paths, the Forwarding Table automatically populates library version numbers in the **Version** columns. The `LibraryVersion` property is the model version of the library at the time the link was created.

- c** (Optional) You can define a transformation function to update old link parameter data using a MATLAB file on the path. Specify the function in the **Transformation Function** column. Transforming old link parameter data for the new library block enables you to load old links and preserve parameter data.

If you do not want to specify a transformation function, do not edit the field. When you save the library, the column will display No Transformation.

- 5** Click **OK** to apply changes and close the dialog box.

After specifying the forwarding table mapping, when you open a model containing links to the library, links to old library blocks will be updated.

To view an example of a forwarding table:

- 1** Enter

```
open_system('simulink')
```

- 2** Select **File > Library Properties > Library Properties**

- 3** Click the Forwarding Table tab.

- 4** View how the forwarding table specifies the mapping of old blocks to new blocks. You cannot make changes because the library is locked. Do not edit the forwarding table for your Simulink library.

Forwarding Table (mapping for old library blocks)

	Old Block Path	Versio	New Block Path	Versio	Transformation Funct
	simulink/Discrete/Weight Moving Average	n/a	simulink_need_slupdate/ Moving Average	n/a	No Transformation
	simulink/Lookup Tables/Interpolation (n-D) using PreLookup	n/a	simulink_need_slupdate/ using PreLookup	n/a	No Transformation
	simulink/Lookup Tables/PreLookup Index Search	n/a	simulink_need_slupdate/ Index Search	n/a	No Transformation
	simulink/Linear/Dot Prod	n/a	simulink/Math Operations/Dot Product	n/a	No Transformation
	simulink/Nonlinear/Look- Table (2-D)	n/a	simulink_need_slupdate/ Table (2-D)	n/a	No Transformation
	simulink/Nonlinear/Alget	n/a	simulink/Math Operations/Algebraic Co	n/a	No Transformation
	simulink/Linear/Slider Gain	n/a	simulink/Math Operations/Slider Gain	n/a	No Transformation
	simulink/Nonlinear/Coulc Viscous Friction	n/a	simulink/Discontinuities/t Viscous Friction	n/a	No Transformation
	simulink/Nonlinear/Manu	n/a	simulink/Signal Routing/Manual Switch	n/a	No Transformation
	simulink/Connections/Mc	n/a	simulink/Model-Wide Utilities/Model Info	n/a	No Transformation
	simulink/Linear/Matrix	n/a	simulink3/Math/Matrix	n/a	No Transformation





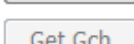
5 Click **OK** to close the dialog

The following example shows a forwarding table that defines:

- A block with the same name moving to a different library (Constant A)
- A block changing name in the same library (Block X to Block Y)
- A block changing name, moving library, and applying a transformation function (Gain A to Gain B)
- A block with three version updates (Block A) using a transformation function. When you select identical old and new library block names and paths, the Forwarding Table automatically populates version numbers

in the **Version** columns. If this is the first entry with identical names, version starts at 0, and the new version number is set to the `ModelVersion` of the library. For subsequent entries, the first version is set to the previous entry's new version, and the new version is set to the current `ModelVersion` of the library.

Forwarding Table (mapping for old library blocks)

	Old Block Path	Version	New Block Path	Version	Transformation Function
	LibA/Constant A	n/a	LibB/Constant A	n/a	No Transformation
	LibA/Block X	n/a	LibA/Block Y	n/a	No Transformation
	LibA/Gain A	n/a	LibB/Gain B	n/a	TransformationFcn1
	LibA/Block A	0.000	LibA/Block A	0.900	TransformationFcn2
	LibA/Block A	0.900	LibA/Block A	1.100	TransformationFcn2
Get Gcb	LibA/Block A	1.100	LibA/Block A	1.150	TransformationFcn2

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 following command creates a forwarding table and assigns it to a library named `Lib1`.

```
set_param('Lib1', 'ForwardingTable', {{'Lib1/A', 'Lib2/A'}
{'Lib1/B', 'Lib1/C'}});
```

The forwarding table specifies that block A has moved from `Lib1` to `Lib2`, and that block B is now named C. Suppose that you open a model that contains links to `Lib1/A` and `Lib1/B`. Simulink updates the link to `Lib1/A` to refer to `Lib2/A` and the link to `Lib1/B` to refer to `Lib1/C`. The changes become permanent when you subsequently save the model.

Writing Transformation Functions. You can use transformation functions to add or remove parameters and define parameter values. Transforming old link parameter data for the new library block enables you to load old links and preserve parameter data that differs from library values. Define your transformation function using a MATLAB file on the path, then specify the function in the Forwarding Table **Transformation Function** column.

The transformation function in your MATLAB file must be like the following:

```
function outData = TransformationFcn(inData)
```

where `inData` is a structure with fields `ForwardingTableEntry` and `InstanceData`, and `ForwardingTableEntry` is a structure with fields `__s1OldName__`, `__s1NewName__`, `__s1OldVersion__`, `__s1NewVersion__`.

This 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. Use this to combine many local functions into a single transformation function, to avoid having many transformation functions on the MATLAB path.

`InstanceData` and `NewInstanceData` are structures with fields `Name` and `Value`. Instance data means the names and values of parameters that are different from the library values.

`outData` is a structure with fields `NewInstanceData` and `NewBlockPath`.

The following example code shows how to define a transformation function that adds a parameter with value `uint8` to update a Compare To Constant block:

```
function [outData] = TransformationCompConstBlk(inData)
% Example transformation Function for old 'Compare To Const' block.
%
% If instanceData of old 'Compare To Const' block does not have
% the 'OutDataTypestr' parameter,
% add the parameter with value 'uint8'.
%
% In the Simulink library the 'ForwardingTableEntry' would be of type:
% ||__s1OldName__||A||__s1NewName__||B||__s1TransformationFcn__||TX
%
```

```
% Exact entry is:
% |__s1OldName__|fixpt_lib_4/Logic & Comparison/Compare\nTo Constant|__s1NewName__|
% simulink/Logic and Bit\nOperations/Compare\To Constant|__s1TransformationFcn__|
% TransformationCompConstBlk
%
%

outData.NewBlockPath = '';
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;
```

Creating Aliases for Mask Parameters

Simulink lets you create aliases, i.e., alternate names, for a mask's parameters. A model can then refer to the mask parameter by either its name or its alias. This 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 (see “Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes” on page 28-29).

To create aliases for a masked block's mask parameters, use the `set_param` command to set the block's `MaskVarAliases` parameter to a cell array that specifies the names of the aliases in the same order as the mask names appear in the block's `MaskVariables` parameter.

Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes. The following example illustrates the use of mask parameter aliases to create backward-compatible parameter name changes.

- 1 Create a new library. **File > New > Library**
- 2 Open the model `masking_example` described in “How Mask Parameters Work” on page 26-5. Drag the masked block named `mx+b` into your new library and rename it to `Line`.
- 3 Right-click the block and select **Properties**. In the Block Properties dialog, select the **Block Annotation** tab. In the **Enter text and tokens for annotation** box, enter

```
m = %<m>
b = %<b>
```

The block displays the value of its `m` and `b` parameters,

- 4 Right-click the block, and select **Mask > Mask Parameters**. In the Block Parameter dialog, enter 0.5 for the **Slope** and 0 for the **Intercept**.
- 5 Save the new library with the filename `mylibrary`.
- 6 Create a new Simulink model. **File > New > Model**.
- 7 From the `mylibrary` window, drag an instance of the `Line` block to your new model. Rename the instance `LineA`.
- 8 Right-click the block and select **Mask > Mask Parameters**. In the Block Parameters dialog, change the value **Slope** to -0.5. and change the value **Intercept**. to 30. Select **Display > Library Links > User Defined**.
- 9 Add a Scope and Clock block to your model and connect them to your block. Save the new model with the filename `mymodel1`.
- 10 From the **Simulation** menu, select **Model Configuration Parameters**. From the **Type** list, select `Fixed-step`. From the **Solver** list, select `discrete` (no continuous states. In the **Fixed-step size** box, enter 0.1. Simulate model.

Note that the model simulates without error.

11 Save and close `mymodel`.

12 Open `mylibrary`.

13 Edit mask. Right-click block, select **Edit mask**. In the Mask Editor dialog,

- Select the **Parameters** tab. In the **Dialog parameters** section and **Variable** column, change the variable `m` to `slope` and `b` to `intercept`.
- Select the Icon & Ports tab. In the Icon Drawing comments box, change the variable `m` to `slope` and `b` to `intercept`.

14 Right-click the Line block, select **Properties**. In the Block Properties dialog, .

- Select the **Block Annotation** tab . In the **Enter text and tokens of annotation** box, rename the `m` parameter to `slope` and the `b` parameter to `intercept`.

15 Click **OK** and save `mylibrary`.

16 Reopen `mymodel`.

Note that `LineA` icon has reverted to the appearance of its library master (i.e., `mylib/Line`) and that its annotation displays question marks for the values of `m` and `b`. These changes reflect the parameter name changes in the library block. In particular, Simulink cannot find any parameters named `m` and `b` in the library block and hence does not know what to do with the instance values for those parameters. As a result, `LineA` reverts to the default values for the slope and intercept parameters, thereby inadvertently changing the behavior of the model. The following steps show how to use parameter aliases to avoid this inadvertent change of behavior.

17 Close `mymodel`.

18 In the Library: `mylibrary` window, select the Line block.

19 Execute the following command at the MATLAB command line.

```
set_param(gcf, 'MaskVarAliases', {'m', 'b'})
```

This specifies that `m` and `b` are aliases for the Line block `slope` and `intercept` parameters.

20 Reopen `mymodel`.

Note that LineA appearance, not the annotation, now reflects the value of the slope parameter under its original name, i.e., `m`. This is because when Simulink opened the model, it found that `m` is an alias for `slope` and assigned the value of `m` stored in the model file to the LineA `slope` parameter.

21 Change LineA block annotation property to reflect LineA parameter name changes, replace

```
m = %<m>
b = %<b>
```

with

```
m = %<slope>
b = %<intercept>
```

LineA now appears with `m = -0.5` and `b = 30`.

Note that LineA annotation shows that, thanks to parameter aliasing, Simulink has correctly applied the parameter values stored for LineA in the `mymodels` file to the block renamed parameters.

Add Libraries to the Library Browser

In this section...

“Display a Library in the Library Browser” on page 28-32

“Example of a Minimal `slblocks.m` File” on page 28-32

“Add More Descriptive Information in `slblocks.m`” on page 28-33

Display a Library in the Library Browser

- 1 Create a folder in the MATLAB path for the top-level library and its sublibraries.

You must store each top-level library that you want to appear in the Library Browser in its own folder on the MATLAB path. Two top-level libraries cannot exist in the same folder.

- 2 Create or copy the top-level library and its sublibraries into the folder you created in the MATLAB path.
- 3 In the folder for the top-level library, include a `slblocks.m` file.

The approach you use to create the `slblocks.m` file depends on your requirements for describing the library:

- If a minimal `slblocks.m` file meets your needs, then create a new `slblocks.m` file, based on the example below
- If you want to describe the library more fully, consider copying an existing `slblocks.m` file to use as a template, editing the copy to describe your library (see below).

Example of a Minimal `slblocks.m` File

To display a library in the Library Browser, at a minimum you must include these lines (adjusted to describe your library; comments are not required) in the `slblocks.m` file.

```
function blkStruct = slblocks
    % Specify that the product should appear in the library browser
```



```
% and be cached in its repository  
Browser.Library = 'mylib';  
Browser.Name    = 'My Library';  
blkStruct.Browser = Browser;
```

Add More Descriptive Information in `sblocks.m`

You can review other descriptive information you may wish to include in your `sblocks.m` file by examining the comments in the Simulink library `sblocks.m` file: `matlabroot/toolbox/simulink/blocks/sblocks.m`.

Using the MATLAB Function Block

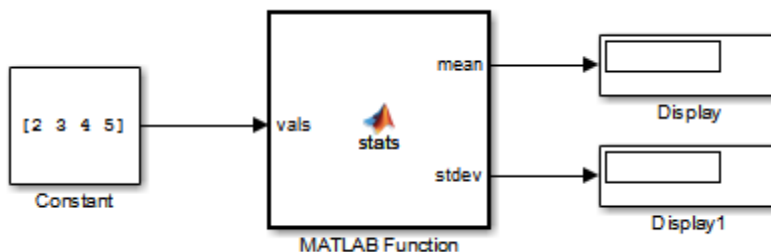
- “Integrate MATLAB Algorithm in Model” on page 29-4
- “What Is a MATLAB Function Block?” on page 29-6
- “Why Use MATLAB Function Blocks?” on page 29-8
- “Create Model That Uses MATLAB Function Block” on page 29-9
- “Code Generation Readiness Tool” on page 29-16
- “Check Code Using the Code Generation Readiness Tool” on page 29-23
- “Debugging a MATLAB Function Block” on page 29-24
- “MATLAB Function Block Editor” on page 29-34
- “MATLAB Function Reports” on page 29-53
- “Type Function Arguments” on page 29-67
- “Size Function Arguments” on page 29-75
- “Add Parameter Arguments” on page 29-77
- “Resolve Signal Objects for Output Data” on page 29-78
- “Types of Structures in MATLAB Function Blocks” on page 29-80
- “Attach Bus Signals to MATLAB Function Blocks” on page 29-81
- “How Structure Inputs and Outputs Interface with Bus Signals” on page 29-83
- “Rules for Defining Structures in MATLAB Function Blocks” on page 29-84
- “Index Substructures and Fields” on page 29-85

- “Create Structures in MATLAB Function Blocks” on page 29-86
- “Assign Values to Structures and Fields” on page 29-88
- “Initialize a Matrix Using a Non-Tunable Structure Parameter” on page 29-90
- “Define and Use Structure Parameters” on page 29-93
- “Limitations of Structures and Buses in MATLAB Function Blocks” on page 29-95
- “What Is Variable-Size Data?” on page 29-96
- “How MATLAB Function Blocks Implement Variable-Size Data” on page 29-97
- “Enable Support for Variable-Size Data” on page 29-98
- “Declare Variable-Size Inputs and Outputs” on page 29-99
- “Filter a Variable-Size Signal” on page 29-100
- “Enumerated Types Supported in MATLAB Function Blocks” on page 29-107
- “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109
- “Add Inputs, Outputs, and Parameters as Enumerated Data” on page 29-110
- “Basic Approach for Adding Enumerated Data to MATLAB Function Blocks” on page 29-112
- “Instantiate Enumerated Data in MATLAB Function Blocks” on page 29-113
- “Control an LED Display” on page 29-114
- “Operations on Enumerated Data” on page 29-118
- “Using Enumerated Data in MATLAB Function Blocks” on page 29-119
- “Share Data Globally” on page 29-120
- “Add Frame-Based Signals” on page 29-129
- “Create Custom Block Libraries” on page 29-135

-
- “Use Traceability in MATLAB Function Blocks” on page 29-156
 - “Include MATLAB Code as Comments in Generated Code” on page 29-161
 - “Enhance Code Readability for MATLAB Function Blocks” on page 29-167
 - “Control Run-Time Checks” on page 29-176
 - “Track Object Using MATLAB Code” on page 29-178
 - “Filter Audio Signal Using MATLAB Code” on page 29-206
 - “Encapsulating the Interface to External Code” on page 29-239
 - “Encapsulate Interface to an External C Library” on page 29-240
 - “Best Practices for Using `coder.ExternalDependency`” on page 29-243
 - “Update Build Information from MATLAB code” on page 29-245

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)
% #codegen

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

You will build this model in “Create Model That Uses MATLAB Function Block” on page 29-9.

Defining Local Variables for Code Generation

If you intend 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 37-2. In the example

function `stats`, the local variable `len` is defined before being used to calculate mean and standard deviation:

```
len = length(vals);
```

Generally, once you assign properties to a variable, you cannot redefine its class, size, or complexity elsewhere in the function body, but there are exceptions (see “Reassignment of Variable Properties” on page 36-9).

What Is a MATLAB Function Block?

The MATLAB Function block allows you to add MATLAB functions to Simulink models for deployment to desktop and embedded processors. This capability is useful for coding algorithms that are better stated in the textual language of the MATLAB software than in the graphical language of the Simulink product. From the MATLAB Function block, you can generate readable, efficient, and compact C/C++ code for deployment to desktop and embedded applications. .

Calling Functions in MATLAB Function Blocks

MATLAB Function blocks can call any of the following types of functions:

- **Local functions**

Local functions are defined in the body of the MATLAB Function block. In the preceding example, `avg` is a local function. See “Call Local Functions” on page 44-9.

- **MATLAB toolbox functions that support code generation**

From MATLAB Function blocks, you can call toolbox functions that support code generation. When you build your model with Simulink Coder, these functions generate C code that is optimized to meet the memory and performance requirements of desktop and embedded environments. In the preceding example, `length`, `sqrt`, and `sum` are examples of toolbox functions that support code generation. See “Call Supported Toolbox Functions” on page 44-10. For a complete list of supported functions, see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2.

- **MATLAB functions that do not support code generation**

From MATLAB Function blocks, you can also call *extrinsic* functions. These are functions on the MATLAB path that the compiler dispatches to MATLAB software for execution because the target language does not support them. These functions do not generate code; they execute only in the MATLAB workspace during simulation of the model. The Simulink Coder software attempts to compile all MATLAB functions unless you explicitly declare them to be extrinsic by using `coder.extrinsic`. See “Declaring MATLAB Functions as Extrinsic Functions” on page 44-12.

The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, it automatically calls out to MATLAB for these functions. For standalone code generation, it does not generate code for these visualization functions. This capability removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

See “Resolution of Function Calls in MATLAB Generated Code” on page 44-2.

Why Use MATLAB Function Blocks?

MATLAB Function blocks provide the following capabilities:

- **Allow you to build MATLAB functions into embeddable applications** — MATLAB Function blocks support a subset of MATLAB toolbox functions that generate efficient C/C++ code (see “Functions Supported for Code Generation”). With this support, you can use Simulink Coder to generate embeddable C code from MATLAB Function blocks that implement a variety of sophisticated mathematical applications. In this way, you can build executables that harness MATLAB functionality, but run outside the MATLAB environment.
- **Inherit 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 Simulink signals. You can also choose to specify the size and type of inputs and outputs explicitly in the Ports and Data Manager (see “Ports and Data Manager” on page 29-38) or in the Model Explorer (see “Model Explorer Overview” on page 9-2).

Create Model That Uses MATLAB Function Block

In this section...

“Adding a MATLAB Function Block to a Model” on page 29-9

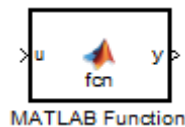
“Programming the MATLAB Function Block” on page 29-10

“Building the Function and Checking for Errors” on page 29-12

“Defining Inputs and Outputs” on page 29-14

Adding a MATLAB Function Block to a Model

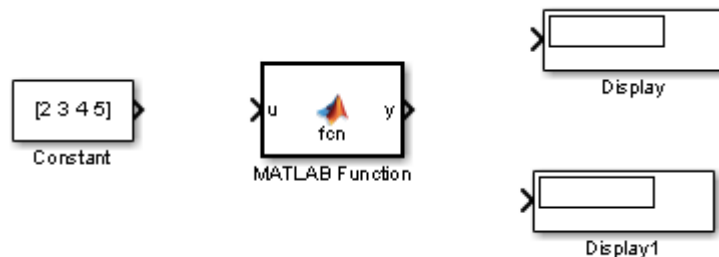
- 1 Create a new model with the Simulink product and add a MATLAB Function block to it from the User-Defined Function library:



- 2 Add the following Source and Sink blocks to the model:

- From the Sources library, add a Constant block to the left of the MATLAB Function block and set its value to the vector [2 3 4 5].
- From the Sinks library, add two Display blocks to the right of the MATLAB Function block.

The model should now have the following appearance:



- 3 In the Simulink Editor, select **File > Save As** and save the model as `call_stats_block1`.

Programming the MATLAB Function Block

The following exercise shows you how to program the block to calculate the mean and standard deviation for a vector of values:

- 1 Open the `call_stats_block1` model that you saved at the end of “Adding a MATLAB Function Block to a Model” on page 29-9. Double-click the MATLAB Function block `fcn` to open it for editing.

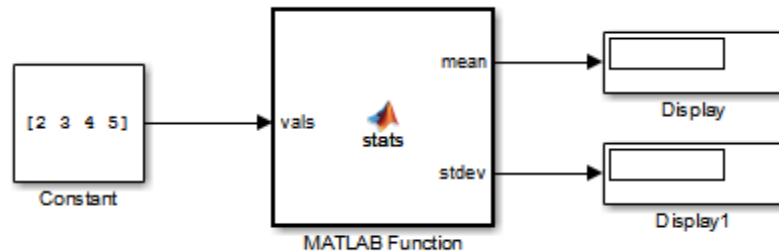
A default function signature appears, along with the `%%codegen` directive. The directive indicates that you intend to generate code from this algorithm and turns on appropriate error checking (see “Compilation Directive `%%codegen`” on page 44-8).

- 2 Edit the function header line as follows:

```
function [mean,stdev] = stats(vals)
%%codegen
```

The function `stats` 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 Save the model as `call_stats_block2`.
- 4 Complete the connections to the MATLAB Function block as shown.



- 5 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;
```

More about length

The function `length` is an example of a toolbox function that supports code generation. When you simulate this model, C code is generated for this function in the simulation application.

More about len

The class, size, and complexity of local variable `len` matches the output of the toolbox function `length`, which returns a real scalar of type `double`.

By default, implicitly declared local variables like `len` are temporary. They come into existence only when the function is called and cease to exist when the function is exited. To make implicitly declared variables persist between function calls, see “Define and Initialize Persistent Variables” on page 36-10.

More about plot

The function `plot` is not supported for code generation. The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, it automatically calls out to MATLAB for these functions. For standalone code generation, it does not generate code for these visualization functions.

- 6 Save the model as `call_stats_block2`.

Building the Function and Checking for Errors

After programming a MATLAB Function block in a Simulink model, you can build the function and test for errors. This section describes the steps:

- 1 Set up your compiler.
- 2 Build the function.
- 3 Locate and fix errors.

Setting Up Your Compiler

Before building your MATLAB Function block, you must set up your C compiler by running the `mex -setup` command, as described in the documentation for `mex`. You must run this command even if you use the default C compiler that comes with the MATLAB product for Microsoft Windows platforms. You can also use `mex` to choose and configure a different C compiler, as described in “Build MEX-Files”.

Supported Compilers for Simulation Builds. To view a list of compilers for building models containing MATLAB Function blocks for simulation:

- 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 for MATLAB Function blocks.

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.

How to Generate Code for the MATLAB Function Block

- 1 Open the `call_stats_block2` model that you saved at the end of “Programming the MATLAB Function Block” on page 29-10.
- 2 Double-click its MATLAB Function block `stats` to open it for editing.
- 3 In the MATLAB Function Block Editor, select **Build Model > Build** to compile and build the example model.

If no errors occur, the **Simulation Diagnostics** window displays a message indicating success. Otherwise, this window helps you locate errors, as described in “How to Locate and Fix Errors” on page 29-13.

How to Locate and Fix Errors

If errors occur during the build process, the **Simulation Diagnostics** window lists the errors with links to the offending code.

The following exercise shows how 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 **Simulation Diagnostics** window displays each detected error with a red button.

- 2 Click the first error line to display its diagnostic message in the bottom error window.

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 29-53.

- 3 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:

- 4 Correct the error by changing `aug` back to `avg` and recompile.

Defining Inputs and Outputs

In the `stats` function header for the MATLAB Function block you defined in “Programming the MATLAB Function Block” on page 29-10, the function argument `vals` is an input, and `mean` and `stdev` are outputs. By default, function inputs and outputs inherit their data type and size from the signals attached to their ports. In this topic, you examine input and output data for the MATLAB Function block to verify that it inherits the correct type and size.

- 1 Open the `call_stats_block2` model that you saved at the end of “Programming the MATLAB Function Block” on page 29-10. Double-click the MATLAB Function block `stats` to open it for editing.
- 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. Notice 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, 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 compilation of the model, and are reported in the **Compiled Type** and **Compiled Size** columns of the left pane.

You can specify the type of an input or output argument by selecting a type 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 29-67 and “Size Function Arguments” on page 29-75 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.

For more information, see “Ports and Data Manager” on page 29-38.

Code Generation Readiness Tool

In this section...

“What Information Does the Code Generation Readiness Tool Provide?” on page 29-16

“Summary Tab” on page 29-17

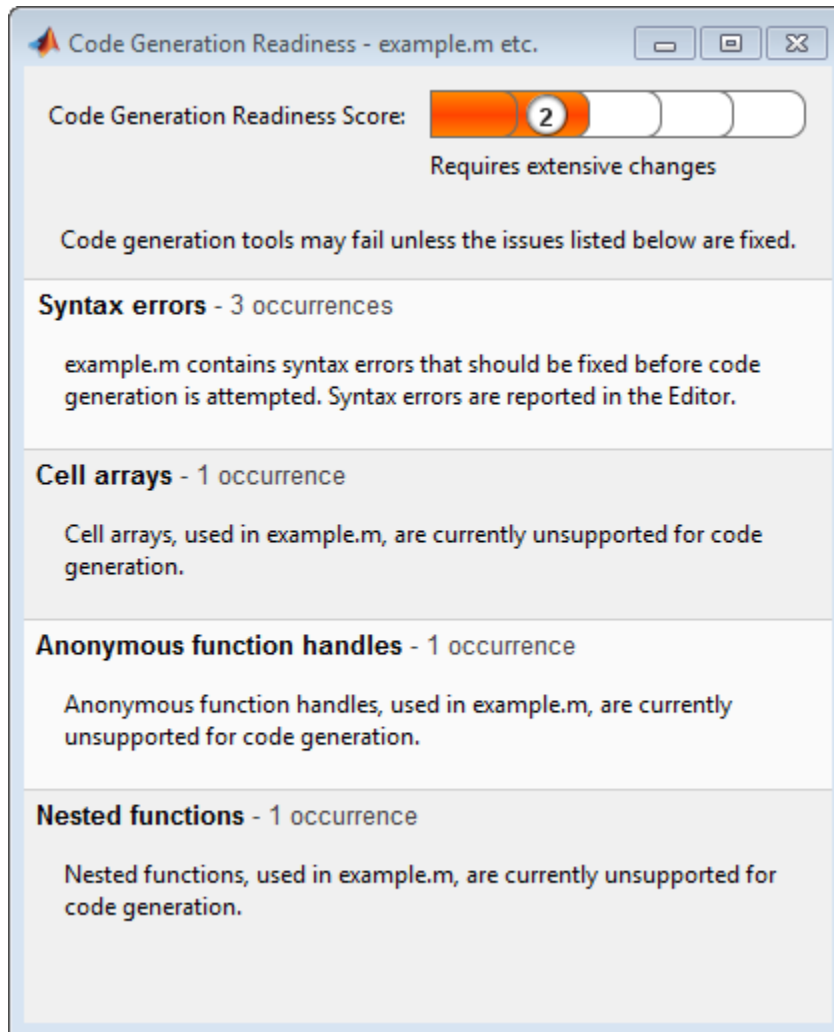
“Code Structure Tab” on page 29-19

“See Also” on page 29-22

What Information Does the Code Generation Readiness Tool Provide?

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code suitable for code generation. The tool might not detect all code generation issues. Under certain circumstances, it might report false errors. Because the tool might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.

Summary Tab

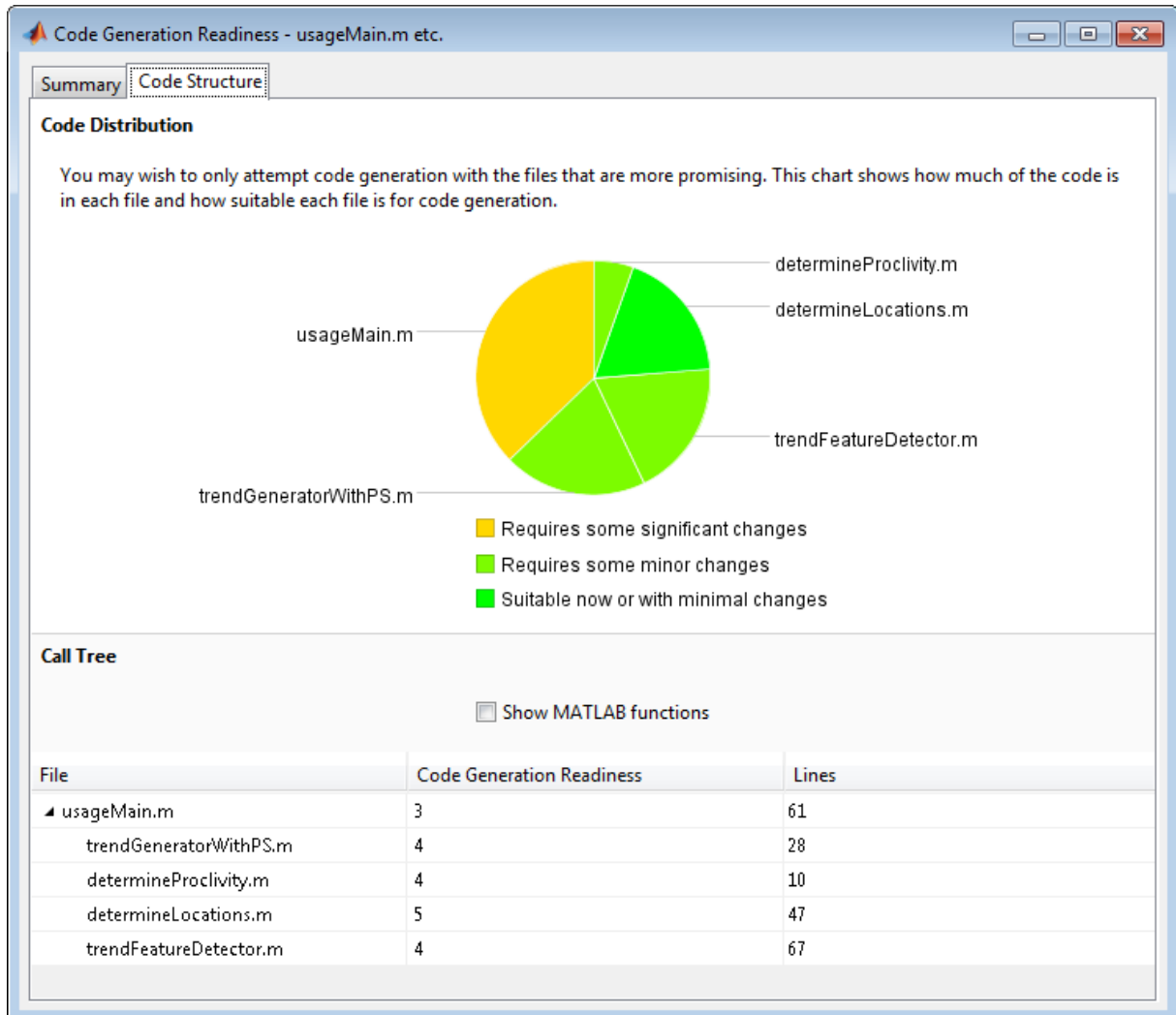


The **Summary** tab provides a **Code Generation Readiness Score** which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes.

On this tab, the tool also provides information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. Use the code analyzer to learn more about the issues and how to fix them.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as recursion, cell arrays, nested functions, and function handles.
- 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 provides information about the relative size of each file and how suitable each file is for code generation.

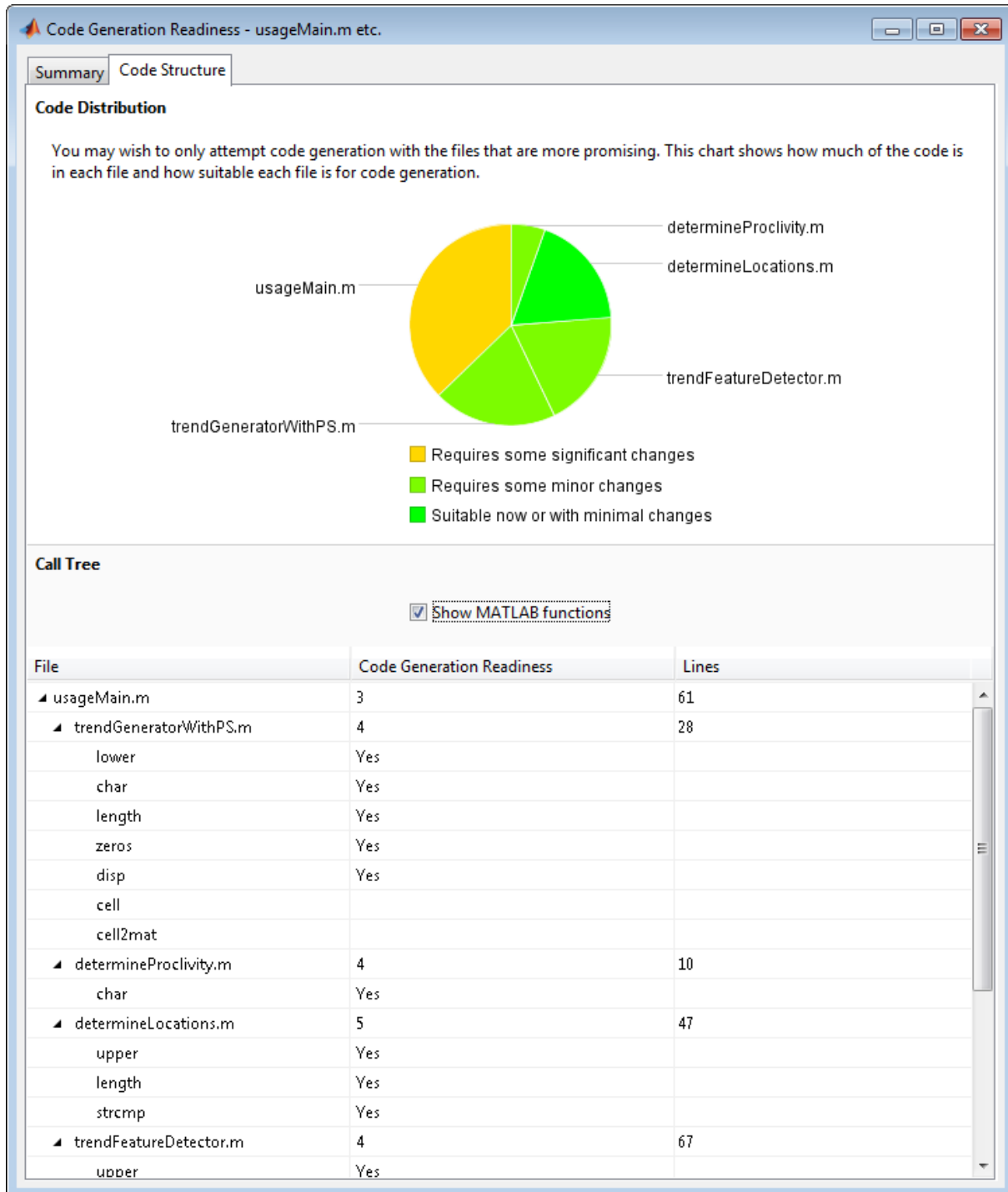
Code Distribution

The **Code Distribution** pane provides a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. This information is useful during the planning phase of a project for estimation and scheduling purposes. If the report indicates that there are multiple files not yet suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane provides information on 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 has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal 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 called by your function code. For each of these MATLAB functions, if the function is supported for code generation, the report sets **Code Generation Readiness** to Yes.



See Also

- “Check Code Using the Code Generation Readiness Tool” on page 29-23

Check Code Using the Code Generation Readiness Tool

In this section...

“Run Code Generation Readiness Tool at the Command Line” on page 29-23

“Run the Code Generation Readiness Tool From the Current Folder Browser” on page 29-23

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.

Debugging a MATLAB Function Block

In this section...

“How Debugging Affects Simulation Speed” on page 29-24

“Enabling and Disabling Debugging” on page 29-24

“Debugging the Function in Simulation” on page 29-24

“Watching Function Variables During Simulation” on page 29-28

“Checking for Data Range Violations” on page 29-30

“Debugging Tools” on page 29-30

How Debugging Affects Simulation Speed

Debugging a MATLAB Function block slows simulation. For maximum simulation speed, disable debugging as described in “Enabling and Disabling Debugging” on page 29-24.

Enabling and Disabling Debugging

There are two levels of debugging available when using MATLAB Function blocks, model level debugging and block level debugging.

Disable debugging for an entire model by clearing the **Enable debugging/animation** check box in the **Simulation Target** pane in the Configuration Parameters dialog. Disable debugging for an individual MATLAB Function block by clicking **Breakpoints > Enable Debugging** in the MATLAB Function Block Editor. If **Enable Debugging** is unavailable, then the **Simulation Target** pane in the Configuration Parameters dialog is controlling debugging.

Debugging the Function in Simulation

In “Create Model That Uses MATLAB Function Block” on page 29-9, you created an example model with a MATLAB Function block that calculates the mean and standard deviation for a set of input values.

To debug the MATLAB Function in this model:

- 1 Open the `call_stats_block2` model and double-click its MATLAB Function block `stats` to open it for editing.
- 2 In the MATLAB Function Block Editor, click the dash (-) character in the left margin of the line:

```
len = length(vals);
```

A small red ball appears in the margin of this line, indicating that you have set a breakpoint.

```
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;
```

3 Begin simulating the model:

If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. This is indicated by a small green arrow in the left 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 Select **Step** to advance execution.

The execution arrow advances to the next line of `stats`, which calls the local function `avg`.

5 Select **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 the **Step In** icon to enter it. If you want to execute the remaining lines of the local function, use **Step Out**.

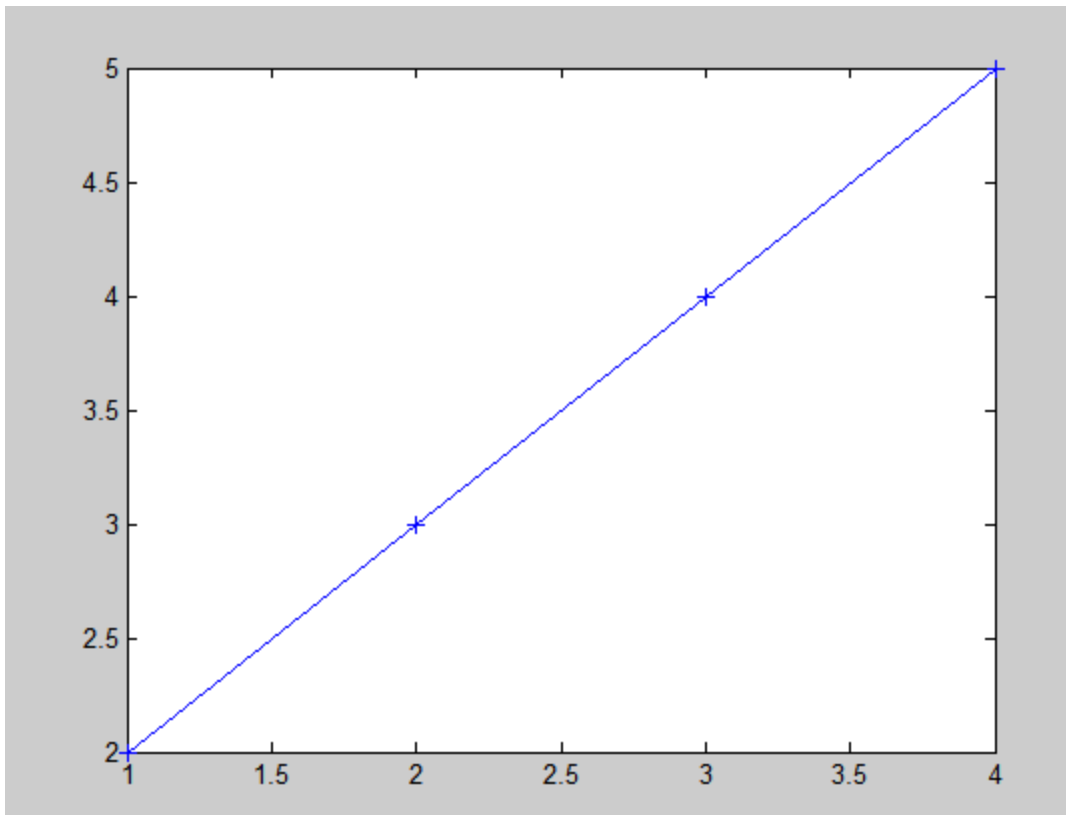
6 Select **Step** to execute the only line in the local function `avg`. When the local function `avg` finishes executing, a green arrow pointing down under its last line appears.

7 Select **Step** again to return to the function `stats`.

Execution advances to the line after the call to the local function `avg`.

8 Step again 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 Select `Continue` to continue execution of the model.

The computed values of `mean` and `stdev` now appear in the Display blocks.

- 10 In the MATLAB Function Block Editor, select **Quit Debugging** to stop simulation.

Watching Function Variables During Simulation

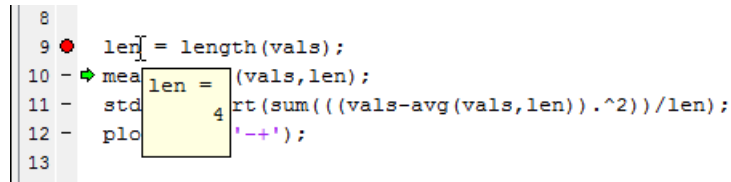
While you simulate a MATLAB Function block, you can use several tools to keep track of variable values in the function.

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:



The screenshot shows a MATLAB Function Block Editor window with a code editor. The code is as follows:

```

8
9 len = length(vals);
10 -> mean(len, len);
11 - std(len, len);
12 - plot(len, len, 'r');
13

```

A mouse cursor is positioned over the variable `len` on line 10. A yellow pop-up display box appears, showing the variable name `len` and its current value, `4`.

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
<code>ctrl-c</code>	Quit debugging and terminate simulation.
<code>dbcont</code>	Continue execution to next breakpoint.
<code>dbquit</code>	Quit debugging and terminate simulation.
<code>dbstep</code> [in out]	Advance to next program step after a breakpoint is encountered. Step over or step into/out of a MATLAB local function.
<code>help</code>	Display help for command line debugging.
<code>print <var></code>	Display the value of the variable <code>var</code> in the current scope. If <code>var</code> is a vector or matrix, you can also index into <code>var</code> . For example, <code>var(1,2)</code> .
<code>save</code>	Saves all variables in the current scope to the specified file. Follows the syntax of the MATLAB <code>save</code> command. To retrieve variables to the MATLAB base workspace, use <code>load</code> command after simulation has been ended.
<code><var></code>	Equivalent to " <code>print <var></code> " if variable is in the current scope.
<code>who</code>	Display the variables in the current scope.
<code>whos</code>	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 string, 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

When you enable debugging, MATLAB Function blocks automatically check input and output data for data range violations when the values enter or leave the blocks.

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 29-45.

Debugging Tools

Use the following tools during a MATLAB Function block debugging session:

Tool Button	Description	Shortcut Key
 Build	<p>Access this tool from the Editor tab by selecting Build Model > Build.</p> <p>Check for errors and build a simulation application (if no errors are found) for the model containing this MATLAB Function block.</p>	Ctrl+B
 Update Diagram	<p>Access this tool from the Editor tab by selecting Build Model > Update Diagram.</p> <p>Check for errors based on the latest changes you make to the MATLAB Function block.</p>	Ctrl+D
 Update Ports	<p>Access this tool from the Editor tab by selecting Build Model > Update Ports.</p> <p>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.</p>	Ctrl+Shift+A
 Run Model	<p>Start simulation of the model containing the MATLAB Function block. If execution is paused at a breakpoint, continues debugging.</p>	F5
 Stop Model	<p>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.</p>	Shift+F5

Tool Button	Description	Shortcut Key
 Set/Clear	<p>Access this tool by selecting Breakpoints > Set/Clear.</p> <p>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 in the on the selected line.</p> <p>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.</p>	F12
 Clear All	<p>Access this tool by selecting Breakpoints > Clear All.</p> <p>Clear all existing breakpoints in the MATLAB Function block code.</p>	None
 Step	<p>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.</p>	F10
 Step In	<p>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.</p>	F11

Tool Button	Description	Shortcut Key
 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

MATLAB Function Block Editor

In this section...

“Customizing the MATLAB Function Block Editor” on page 29-34

“MATLAB Function Block Editor Tools” on page 29-34

“Editing and Debugging MATLAB Function Block Code” on page 29-35



“Ports and Data Manager” on page 29-38



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
 <p>Edit Data</p>	<p>Opens the Ports and Data Manager dialog to add or modify arguments for the current MATLAB Function block (see “Ports and Data Manager” on page 29-38).</p>
 <p>View Report</p>	<p>Opens the MATLAB Function report for the MATLAB Function block. For more information, see “MATLAB Function Reports” on page 29-53.</p>

Tool Button	Description
 Simulation Target	Opens the Simulation Target pane in the Configuration Parameters dialog to enable debugging or include custom code. See “Enabling and Disabling Debugging” on page 29-24 for more information on debugging
 Go To Diagram	Displays the MATLAB function in its native diagram without closing the editor.

See “Defining Inputs and Outputs” on page 29-14 for an example of defining an input argument for a MATLAB Function block.

Editing and Debugging MATLAB Function Block Code

Commenting Out a Block of Code

To comment out a block of code in the MATLAB Function Block Editor:




- 1 Highlight the text that you would like to comment out.
- 2 Hold the mouse over the highlighted text and then right-click and select **Comment** from the context menu. (Alternatively, select a Comment tool from the Editor tab.

Note To uncomment a block of code, follow the same steps, but select **Uncomment** instead of **Comment**.

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.
	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 “Opening Files and Functions in the Command Window”.

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 29-38.

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 29-45.

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 that is open and has focus.

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 Overview” on page 9-2).

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 (see “MATLAB Function Block Properties” on page 29-39). Use this button to return to the settings used by the block after viewing data associated with the block arguments.

MATLAB Function Block Properties

This section describes each property of a MATLAB Function block.

Name. Name of the MATLAB Function block, following the same naming conventions as for Simulink blocks (see “Manipulate Block Names” on page 23-27).

Update method. Method for activating the MATLAB Function block. You can choose from the following update methods:

Update Method	Description
Inherited (default)	<p>Input from the Simulink model activates the MATLAB Function block.</p> <p>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.</p> <p>If you define data inputs, the MATLAB Function block samples at the rate of the fastest data input. If you do not</p>

Update Method	Description
	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 code generation 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.

Even when you disable this option, the code for a simulation target checks for integer overflow and underflow. If either condition occurs, simulation stops and an error is generated. If you enable debugging for the MATLAB Function block, the debugger displays the error and lets you examine the data.

If you did not enable debugging, the block generates a run-time error:

```
Overflow detected. Enable debugging for more information.
```

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.

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”).

- 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 <code>fimath</code> properties as the current default <code>fimath</code> . The edit box appears dimmed and displays the current global <code>fimath</code> in read-only form.

Setting	Description
Specify other	<p>When you select this option, you can specify your own <code>fimath</code> object in the edit box. You can do so in one of two ways:</p> <ul style="list-style-type: none"> • Constructing the <code>fimath</code> object inside the edit box. • Constructing the <code>fimath</code> 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”. <p>For more information on <code>fimath</code> objects, see “<code>fimath</code> Object Construction”.</p>

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.

Adding Data to a MATLAB Function Block

You can define data arguments for MATLAB Function blocks using the following methods:

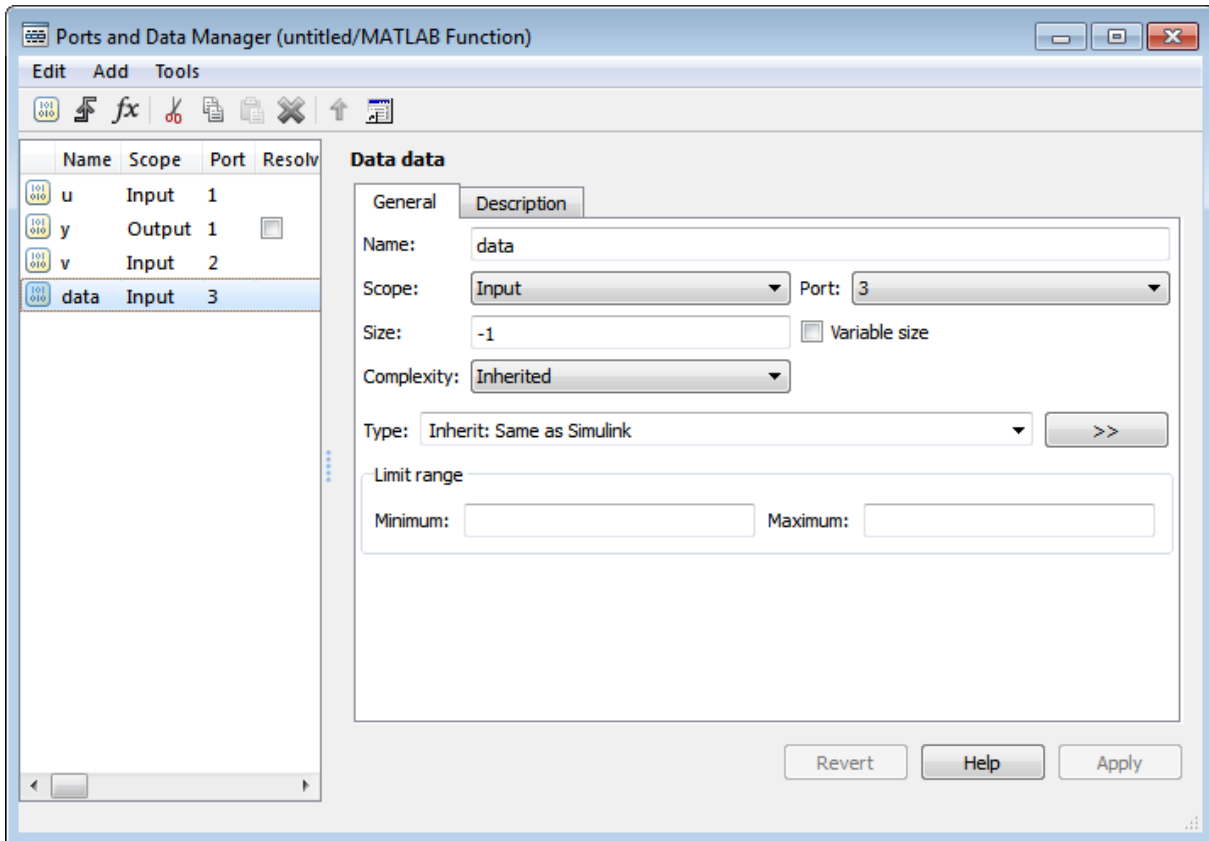
Method	For Defining	Reference
Define data directly in the MATLAB Function block code	Input and output data	See “Defining Inputs and Outputs” on page 29-14.
Use the Ports and Data Manager	Input, output, and parameter data in the MATLAB Function	See “Defining Data in the Ports and Data

Method	For Defining	Reference
	block that is open and has focus	Manager” on page 29-44.
Use the Model Explorer	Input, output, and parameter data in MATLAB Function blocks at all levels of the model hierarchy	See “Model Explorer Overview” on page 9-2

Defining Data in the Ports and Data Manager. To add a data argument and modify its properties, follow these steps:

- 1 In the Ports and Data Manager, select **Add > Data**

The Ports and Data Manager adds a default definition of the data to the MATLAB Function block.



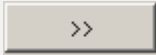
2 Modify data properties.

3 Return to the MATLAB Function block properties at any time by selecting **Tools > Block Dialog**.

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	<p>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:</p> <ul style="list-style-type: none"> • 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 4-74). • 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. <p>For more information, see “Defining Inputs and Outputs” on page 29-14 and “Add Parameter Arguments” on page 29-77.</p>
Port	Index of the port associated with the data argument. This property applies only to input and output data.
Tunable	Indicates whether the parameter used as the source of this data item is tunable (see “Tunable Parameters” on page 3-9). 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 Supported for C/C++ Code Generation — Alphabetical List” on page 34-2).

Property	Description
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. See “Symbol Resolution” on page 4-83 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 29-75. This property does not apply to Data Store Memory data. For more details, see “Size Function Arguments” on page 29-75.
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: <ul style="list-style-type: none"> • 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.
Sampling mode	Specifies how an output signal propagates through a model. This property applies only to data with scope equal to Output. You can set sampling mode to one of the following values: <ul style="list-style-type: none"> • Sample based: Propagate the signal sample by sample (default) • Frame based: Propagate the signal in batches of samples

Property	Description
<p>Type</p>	<p>Type of data object. You can specify the data type by:</p> <ul style="list-style-type: none"> • 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 “Data Types” on page 46-2). • Using the Data Type Assistant to specify a data Mode, then specifying the data type based on that mode. <hr/> <p>Note To display the Data Type Assistant, click the Show data type assistant button:</p> <div style="text-align: center;">  </div> <hr/> <p>For more information, see “Specifying Argument Types” on page 29-67.</p>
<p>Limit range</p>	<p>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.</p> <ul style="list-style-type: none"> • Minimum — The smallest value allowed for the data item during simulation. The default value is <code>-inf</code>. • Maximum — The largest value allowed for the data item during simulation. The default value is <code>inf</code>.

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.

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 “Create a Triggered Subsystem” on page 7-33.

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 29-50.

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.
Trigger	Type of event that triggers execution of the MATLAB Function block. You can select one of the following types of triggers: <ul style="list-style-type: none"> • 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.

Property	Description
	<ul style="list-style-type: none"> 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.

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 in the Simulink model to execute. A function-call subsystem is a subsystem that another block can invoke directly during a simulation. See “Create a Function-Call Subsystem” on page 7-48.

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**.

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.

MATLAB Function Reports

In this section...

- “About MATLAB Function Reports” on page 29-53
- “Location of MATLAB Function Reports” on page 29-53
- “Opening MATLAB Function Reports” on page 29-54
- “Description of MATLAB Function Reports” on page 29-54
- “Viewing Your MATLAB Function Code” on page 29-54
- “Viewing Call Stack Information” on page 29-56
- “Viewing the Compilation Summary Information” on page 29-56
- “Viewing Error and Warning Messages” on page 29-56
- “Viewing Variables in Your MATLAB Code” on page 29-58
- “Keyboard Shortcuts for the MATLAB Function Report” on page 29-64
- “Report Limitations” on page 29-65

About MATLAB Function Reports

Whenever you build a Simulink model that contains MATLAB Function blocks, Simulink automatically generates a report in HTML format for each MATLAB Function block in your model. The report helps you debug your MATLAB functions and verify that they are suitable for code generation. The report provides links to your MATLAB functions and compile-time type information for the variables and expressions in these functions. If your model fails to build, this information simplifies finding sources of error messages and aids understanding of type propagation rules.

Note If you have a Stateflow license, there is one report for each Stateflow chart, regardless of the number of MATLAB functions it contains.

Location of MATLAB Function Reports

The code generation software provides a report for each MATLAB Function block in the model `model_name` at the following location:

```
slprj/_sfprj/  
model_name/_self/  
sfun/html/
```

Opening MATLAB Function Reports

Use one of the following methods:

- In the MATLAB Function Block Editor, select **View Report**.
- In the **Simulation Diagnostics** window, select the **report** link if compilation errors occur.

Description of MATLAB Function Reports


When you build the MATLAB function, the code generation software generates an HTML report. The report provides the following information, as applicable:



- MATLAB code information, including a list of all functions and their compilation status
- Call stack information, providing information on the nesting of function calls
- Summary of compilation results, including type of target and number of warnings or errors
- List of all error and warning messages
- List of all variables in your MATLAB function

Viewing Your MATLAB Function Code

To view your MATLAB function code, click the **MATLAB code** tab. The report displays the MATLAB code for the function highlighted in the list on this tab.

The **MATLAB code** tab provides:

- A list of the MATLAB functions that have been compiled. The report displays icons next to each function name to indicate whether compilation was successful:
 -  Errors in function.

-  Warnings in function.
-  Successful compilation, no errors or warnings.
- A filter control that you can use to sort your functions by:
 - Size
 - Complexity
 - Class

Viewing Local Functions

The report annotates the local function with the name of the parent function in the list of functions on the **MATLAB code** tab.

For example, if the MATLAB function `fcn1` contains the local function `subfcn` and `fcn2` contains the local function `subfcn2`, the report displays:

```
fcn1 > subfcn1
fcn2 > subfcn2
```

Viewing Specializations

If your MATLAB function calls the same function with different types of inputs, the report numbers each of these **specializations** in the list of functions on the **MATLAB code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

```
function y = fcn(u) %#codegen
% Specializations
y = y + subfcn(single(u));
y = y + subfcn(double(u));
```

The report numbers the specializations in the list of functions.

```
fcn > subfcn > 1
fcn > subfcn > 2
```

Viewing Call Stack Information

The report provides call stack information:

- On the **Call stack** tab.
- In the list of **Callers**.

If a function is called from more than one function, this list provides details of each call site. Otherwise, the list is disabled.

Viewing Call Stack Information on the Call Stack Tab

To view call stack information, click the **Call stack** tab. The call stack lists the functions in the order that the top-level function calls them. It also lists the local functions that each function calls.

Viewing Function Call Sites in the Callers List

If a function is called from more than one function, this list provides details of each call site. To navigate between call sites, select a call site from the **Callers** list. If the function is not called more than once, this list is disabled.

Viewing the Compilation Summary Information

To view a summary of the compilation results, including type of target and number of errors or warnings, click the **Summary** tab.

Viewing Error and Warning Messages

The report provides information about errors and warnings. If errors occur during simulation of a Simulink model, simulation stops. If warnings occur, but no errors, simulation of the model continues.

The report provides information about warnings and errors by listing all errors and warnings in chronological order in the **All Messages** tab.

Viewing Errors and Warnings in the All Messages Tab

If errors or warnings occurred during compilation, click the **All Messages** tab to view a complete list of these messages. The report lists the messages in the order that the compiler detects them. It is best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message.

To locate the offending line of code for an error or warning in the list, click the message in the list. The report highlights errors in the list and MATLAB code in red and warnings in orange. Click the blue line number next to the offending line of code in the MATLAB code pane to go to the error in the source file.



Note You can fix errors only in the source file.

Function: **stats** Callers: Select a function call site:

```

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

```

Summary		All Messages (2)	Variables	
Order	Type	Function	Line	Description
1		stats	10	Undefined function or variable 'avg'.
2		stats	11	Undefined function or variable 'avg'.

Viewing Error and Warning Information in Your MATLAB Code

If errors or warnings occurred during compilation, the report underlines them in your MATLAB code. The report underlines errors in red and warnings in orange. To learn more about a particular error or warning, place your pointer over the underlined text.

Viewing Variables in Your MATLAB Code

The report provides compile-time type information for the variables and expressions in your MATLAB code, including name, type, size, complexity, and class. It also provides type information for fixed-point data types, including word length and fraction length. You can use this type information to find sources of error messages and to understand type propagation rules.

You can view information about the variables in your MATLAB code:

- On the **Variables** tab, view the list.
- In your MATLAB code, place your cursor over the variable name.

In the MATLAB code, an orange variable name indicates a compile-time constant argument to an entry-point or a specialized function. The information for these variables includes the value. You can use this information to understand the function signature. You can also use this information to see when code generation created specializations of a function with different constant argument values.

Viewing Variables in the Variables Tab

To view a list of the variables in your MATLAB function, click the **Variables** tab. The report displays a complete list of variables in the order that they appear in the function that you selected on the **MATLAB code** tab. Clicking a variable in the list highlights instances of that variable, and scrolls the MATLAB code pane so that you can view the first instance.

As applicable, the report provides the following information about each variable:

- Order
- Name

- Type
- Size
- Complexity
- Class
- DataTypeMode (DT mode) — for fixed-point data types only. For more information, see “DataTypeMode”.
- Signed — sign information for built-in data types, signedness information for fixed-point data types.
- Word length (WL) — for fixed-point data types only.
- Fraction length (FL) — for fixed-point data types only.

Note For more information on viewing fixed-point data types, see “Use Fixed-Point Code Generation Reports”.

It only displays a column if at least one variable in the code has information in that column. For example, if the code does not contain fixed-point data types, the report does not display the DT mode, WL or FL columns.

Sorting Variables in the Variables Tab. By default, the report lists the variables in the order that they appear in the selected function.

You can sort the variables by clicking the column headings on the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key when clicking the column headings.

To restore the list to the original order, click the **Order** column heading.

Viewing Structures on the Variables Tab. You can expand structures listed on the **Variables** tab to display the field properties.

Summary		All Messages (0)	Variables			
Order	Variable	Type	Size	Complex	Class	
1	s	Output	1 x 1	-	struct	
1.1	s.a	Field	1 x 1	No	double	
1.2	s.b	Field	1 x 1	No	double	
2	a	Input	1 x 1	No	double	
3	b	Input	1 x 1	No	double	

If you sort the variables by type, size, complexity or class, a structure and its fields might not appear sequentially in the list. To restore the list to the original order, click the **Order** column heading.


Viewing Information About Variable-Size Arrays in the Variables

Tab. For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon **:**.

In the following report, variable *A* is variable-size. Its maximum computed size is 1×100.

Summary		All Messages (0)	Variables			
Order	Variable	Type	Size	Complex	Class	
1	B	Output	1 x :100	No	double	
2	A	Input	1 x :100	No	double	
3	tol	Input	1 x 1	No	double	
4	k	Local	1 x 1	No	double	
5	i	Local	1 x 1	No	double	

If the code generation software cannot compute the maximum size of a variable-size array, the report displays the size as **:?**.

Summary		All Messages (1)	Variables	
Order	Type	Function	Line	Description
1		emldemo_uniquetol	10	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [1 x :?]. This error may be reported due to a limitation of the underlying analysis.

If you declare a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends * to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions do not change during execution.

Summary		All Messages (0)	Variables	Target Build Log	
Order	Variable	Type	Size	Complex	Class
1	y	Output	1 x 10 *	No	double

For more information on how to use the size information for variable-sized arrays, see “Variable-Size Data Definition for Code Generation”.

Viewing Renamed Variables in the Variables Tab. If your MATLAB function reuses a variable with different size, type, or complexity, the code generation software attempts to create separate, uniquely named variables in the generated code. For more information, see “Reuse the Same Variable with Different Properties”. The report numbers the renamed variables in the list on the **Variables** tab. When you place your pointer over a renamed variable, the report highlights only the instances of this variable that share the same data type, size, and complexity.

For example, suppose your code uses the variable `t` in a for-loop to hold a scalar double, and reuses it outside the for-loop to hold a 5x5 matrix. The report displays two variables, `t>1` and `t>2` in the list on the **Variables** tab.

```

6  if all(all(u))
7      % First time t is used to hold a scalar double value
8      t = mean(mean(u)) / numel(u);
9      u = u - t;
10 end

```

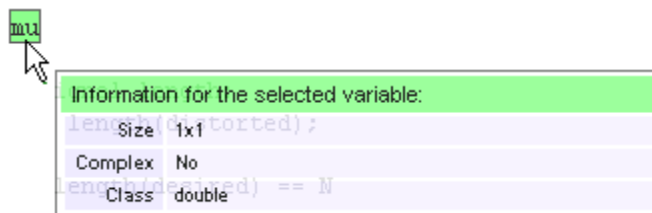
Summary	All Messages (0)	Variables				
Order	Variable	Type	Size	Complex	Class	
1	u	Input	5 x 5	No	double	
2	t > 1	Local	5 x 5	No	double	
3	t > 2	Local	1 x 1	No	double	

Viewing Simulation Minimum and Maximum Values in the Variables Tab. If you have a Fixed-Point Designer license, and you simulate your model using the Fixed-Point Tool set up to log the minimum and maximum values, you can view these values in the MATLAB Function Report. For more information, see “Log Simulation Ranges for MATLAB Function Block”.

Viewing Information About Variables and Expressions in Your MATLAB Function Code

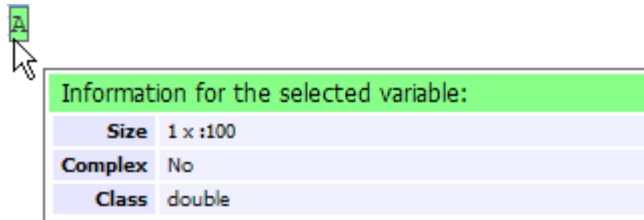
To view information about a particular variable or expression in your MATLAB function code, on the MATLAB code pane, place your pointer over the variable name or expression. The report highlights variables and expressions in different colors:

Green, when the variable has data type information at this location in the code.



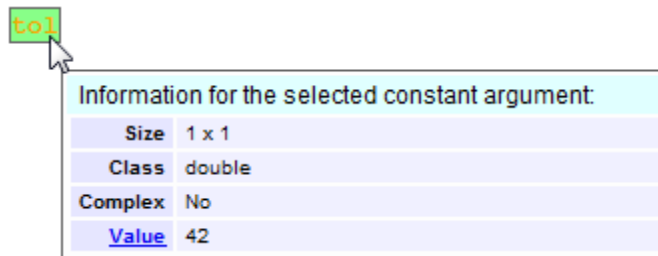
For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is

prefixed with a colon `:`. Here the array `A` is variable-sized with a maximum computed size of `1 x 100`.



Green with orange text, when a constant argument has data type and value information. When the variable is a compile-time constant argument to an entry-point or a specialized function:

- The variable name is orange.
- The information for the variable includes the value.

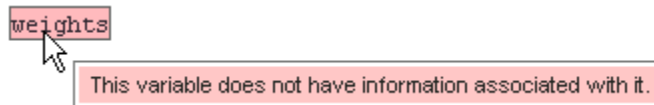


If you export the value as a variable to the base workspace, you can use the Workspace browser to view detailed information about the variable.

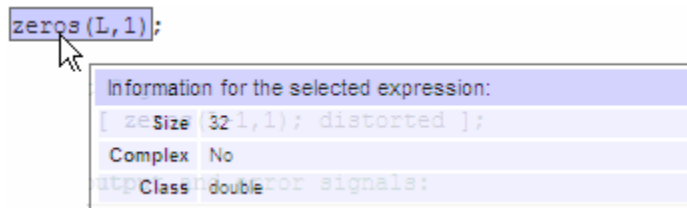
To export the value to the base workspace:

- 1 Click the **Value** link.
- 2 In the Export Constant Value dialog box, specify the **Variable name**.
- 3 Click **OK**.

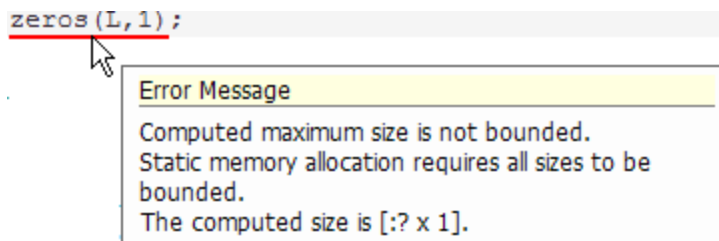
The variable and its value appear in the Workspace browser.

Pink, when the variable has no data type information.

Purple, information about expressions. You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your pointer over an expression. The report highlights expressions in purple and provides more detailed information.



Red, when there is error information for an expression. If the code generation software cannot compute the maximum size of a variable-size array, the report underlines the variable name and provides error information.

**Keyboard Shortcuts for the MATLAB Function Report**

You can use the following keyboard shortcuts to navigate between the different panes in the MATLAB Function report. Once you have selected a pane, use the Tab key to advance through data in that pane.

To select:	Use:
MATLAB Code Tab	Ctrl+m
Call Stack Tab	Ctrl+k
MATLAB Code Pane	Ctrl+w
Summary Tab	Ctrl+s
All Messages Tab	Ctrl+a
Variables Tab	Ctrl+v

Report Limitations

The report displays information about the variables and expressions in your MATLAB code with the following limitations:

varargin and varargout

The report does not support `varargin` and `varargout` arrays.

Loop Unrolling

The report does not display full information for unrolled loops. It displays data types of one arbitrary iteration.

Dead Code

The report does not display information about dead code.

Structures

The report does not provide complete information about structures.

- On the **MATLAB code** pane, the report does not provide information about all structure fields in the `struct()` constructor.
- On the **MATLAB code** pane, if a structure has a nonscalar field, and an expression accesses an element of this field, the report does not provide information for the field.

Column Headings on Variables Tab

If you scroll through the list of variables, the report does not display the column headings on the **Variables** tab.

Multiline Matrices

On the **MATLAB code** pane, the report does not support selection of multiline matrices. It supports only selection of individual lines at a time. For example, if you place your pointer over the following matrix, you cannot select the entire matrix.

```
out1 = [1 2 3;  
        4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

Type Function Arguments

In this section...

“About Function Arguments” on page 29-67

“Specifying Argument Types” on page 29-67

“Inheriting Argument Data Types” on page 29-69

“Built-In Data Types for Arguments” on page 29-70

“Specifying Argument Types with Expressions” on page 29-71

“Specifying Fixed-Point Designer Data Properties” on page 29-71

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*.

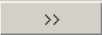
By default, the data type mode for MATLAB Function block function arguments is **Inherited**. This means that the function argument inherits its data type from the incoming or outgoing signal. To override the default type, you first choose a data type mode and then select a data type based on the mode.

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)	<p>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:</p> <ul style="list-style-type: none"> • 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. <p>See “Inheriting Argument Data Types” on page 29-69.</p>
Built in	Select from the drop-down list of supported data types, as described in “Built-In Data Types for Arguments” on page 29-70.
Fixed point	Specify the fixed-point data properties as described in “Specifying Fixed-Point Designer Data Properties” on page 29-71.
Expression	Enter an expression that evaluates to a data type, as described in “Specifying Argument Types with Expressions” on page 29-71.

Mode	What to Specify
Bus Object	<p>In the Bus object field, enter the name of a <code>Simulink.Bus</code> 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 29-83.</p> <hr/> <p>Note You can click the Edit button to create or modify <code>Simulink.Bus</code> objects using the Simulink Bus Editor (see “Attach Bus Signals to MATLAB Function Blocks” on page 29-81).</p> <hr/>
Enumerated	<p>In the Enumerated field, enter the name of a <code>Simulink.IntEnumType</code> object that you define in the base workspace. See “Enumerated Types Supported in MATLAB Function Blocks” on page 29-107 and “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.</p>

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. , and set data type mode using one of these methods:

- 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 29-70 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
<code>double</code>	64-bit double-precision floating point
<code>single</code>	32-bit single-precision floating point
<code>int32</code>	32-bit signed integer
<code>int16</code>	16-bit signed integer
<code>int8</code>	8-bit signed integer
<code>uint32</code>	32-bit unsigned integer
<code>uint16</code>	16-bit unsigned integer
<code>uint8</code>	8-bit unsigned integer
<code>boolean</code>	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 “Creating a Data Type Alias”.
 - `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

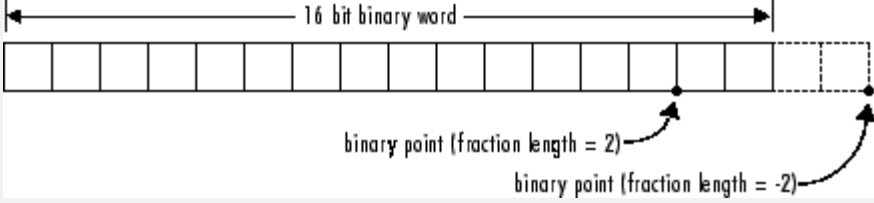
MATLAB Function blocks can represent signals and parameter values as fixed-point numbers. To simulate models that use fixed-point data in MATLAB Function blocks, you must install the Fixed-Point Designer product on your system.

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)	<p>If you select this mode, the Data Type Assistant displays the Fraction Length field, specifying the binary point location.</p> <p>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:</p>  <p>The default is 0.</p>
Slope and bias	<p>If you select this mode, the Data Type Assistant displays fields for entering the Slope and Bias.</p> <ul style="list-style-type: none"> • Slope can be any <i>positive</i> real number. The default is 1.0. • Bias can be any real number. The default value is 0.0. <p>You can enter slope and bias as expressions that contain parameters defined in the MATLAB workspace.</p>

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 on the **Value Attributes** tab.

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 on the Value Attributes tab.
Minimum	The minimum value specified on the Value Attributes tab.
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” 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 29-38) 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** From the Simulink **Analysis** menu, select **Fixed-Point Tool**.
- 2** Set the value of the **Data type override** parameter to `Double` or `Single`.

Size Function Arguments

In this section...

“Specifying Argument Size” on page 29-75

“Inheriting Argument Sizes from Simulink” on page 29-75

“Specifying Argument Sizes with Expressions” on page 29-76

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 29-75.

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 29-77.

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.

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. See “Masking” for more information.

- 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.

For information on declaring parameters for masked blocks, see “How Mask Parameters Work” on page 26-5.

Resolve Signal Objects for Output Data

In this section...

“Implicit Signal Resolution” on page 29-78

“Eliminating Warnings for Implicit Signal Resolution in the Model” on page 29-78

“Disabling Implicit Signal Resolution for a MATLAB Function Block” on page 29-79

“Forcing Explicit Signal Resolution for an Output Data Signal” on page 29-79

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*, as described in Simulink.Signal.

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 4-83 and “Explicit and Implicit Symbol Resolution” on page 4-87 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, select **Simulation > Model Configuration Parameters**.

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 Right-click the MATLAB Function block and select **Block Parameters (Subsystem)** in the context menu.

The Block Parameters dialog opens.

- 2 In the Permit hierarchical resolution field, select **ExplicitOnly** or **None**, and click **OK**.

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**.

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 (for more information, see “Combine Buses into an Array of Buses” on page 51-81). You can also define structures inside MATLAB functions that are not part of MATLAB Function blocks (see “Structure Definition for Code Generation” on page 39-2).

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 29-86.
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 39-4.
Persistent	Declare a variable to be persistent in a MATLAB function.	See “Make Structures Persistent” on page 39-9.
Parameter	Create structure data with scope of Parameter	See “Define and Use Structure Parameters” on page 29-93.

Structures in MATLAB Function blocks can contain fields of any type and size, including muxed signals, buses, and arrays of structures.

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 `e1e1`, `e1e2`, and `e1e3`. The signal `e1e3` 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 `e1e3` 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	Data Type	Size
	<code>inbus</code>	Input	1		Inherit: Same as Simulink	-1
	<code>outbus</code>	Output	1	<input type="checkbox"/>	Bus: MainBus	1
	<code>outbus1</code>	Output	2	<input type="checkbox"/>	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 29-86). 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 screenshot shows the MATLAB workspace with a tree view on the left and a table of properties on the right. The tree view shows a hierarchy: Base Workspace > MainBus (expanded) > ele1, ele2, ele3(SubBus) (expanded) > SubBus (expanded) > a1, a2. The table on the right lists the properties for these elements.

Name	Data Type	Complexity	Dimensions	Dimension
ele1	double	real	1	Fixed
ele2	single	real	1	Fixed
ele3(Su...	SubBus	real	1	Fixed

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 29-85). 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	—

To learn how to define structures in MATLAB Function blocks, see “Create Structures in MATLAB Function Blocks” on page 29-86.

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

Working with Virtual and Nonvirtual Buses

MATLAB Function blocks supports nonvirtual buses only (see “Virtual and Nonvirtual Buses” on page 51-13). When models that contain MATLAB Function block inputs and outputs are built, hidden converter blocks are used to convert bus signals for code generation from MATLAB, as follows:

- Converts incoming virtual bus signals to nonvirtual buses for inputs to structures in MATLAB Function blocks
- Converts outgoing nonvirtual bus signals from MATLAB Function blocks to virtual bus signals

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. For more information, see `Simulink.Bus`.
- MATLAB Function blocks support nonvirtual buses only (see “Working with Virtual and Nonvirtual Buses” on page 29-83).

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 39-2).

For example, in the `emldemo_bus_struct` model described in “Attach Bus Signals to MATLAB Function Blocks” on page 29-81, 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
<code>substruct.a1</code>	Field <code>a1</code> of local structure <code>substruct</code>
<code>inbus.ele3.a1</code>	Value of field <code>a1</code> of field <code>ele3</code> , a substructure of structure <code>inputinbus</code>
<code>inbus.ele3.a2(1,1)</code>	Value in row 1, column 1 of field <code>a2</code> of field <code>ele3</code> , a substructure of structure <code>input inbus</code>

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 29-80).
- 2 Based on the scope, follow these guidelines for creating the structure:

For Structure Scope:	Follow These Steps:
Input	<ol style="list-style-type: none"> 1 Create a <code>Simulink.Bus</code> object in the base workspace to define the structure input. 2 Add data to the MATLAB Function block, as described in “Adding Data to a MATLAB Function Block” on page 29-43. The data should have the following properties <ul style="list-style-type: none"> • Scope = Input • Type = Bus: <object name> <p>For <object name>, enter the name of the <code>Simulink.Bus</code> object that defines the structure input</p> <p>See “Rules for Defining Structures in MATLAB Function Blocks” on page 29-84.</p>
Output	<ol style="list-style-type: none"> 1 Create a <code>Simulink.Bus</code> object in the base workspace to define the structure output. 2 Add data to the MATLAB Function block with the following properties: <ul style="list-style-type: none"> • Scope = Output • Type = Bus: <object name> <p>For <object name>, enter the name of the <code>Simulink.Bus</code> object that defines the structure output</p> 3 Define and initialize the output structure implicitly as a variable in the MATLAB function, as described in “Structure Definition for Code Generation” on page 39-2. 4 Make sure the number, type, and size of fields in the output structure variable definition match the properties of the <code>Simulink.Bus</code> object.

For Structure Scope:	Follow These Steps:
Local	Define the structure implicitly as a local variable in the MATLAB function, as described in “Structure Definition for Code Generation” on page 39-2. By default, local variables in MATLAB Function blocks are temporary.
Persistent	Define the structure implicitly as a persistent variable in the MATLAB function, as described in “Make Structures Persistent” on page 39-9.
Parameter	<ol style="list-style-type: none">1 Create a structure variable in the base workspace.2 Add data to the MATLAB Function block with the following properties:<ul style="list-style-type: none">• Name = same name as the structure variable you created in step 1.• Scope = Parameter See “Define and Use Structure Parameters” on page 29-93.

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 <code>Simulink.Bus</code> objects in the base workspace or locally as implicit structure declarations (see “Create Structures in MATLAB Function Blocks” on page 29-86).
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 <code>Simulink.Bus</code> 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.

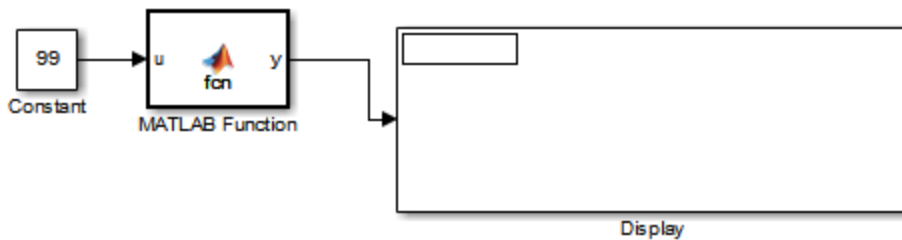
For example, the following table presents valid and invalid structure assignments based on the specifications for the model described in “Attach Bus Signals to MATLAB Function Blocks” on page 29-81:

Assignment	Valid or Invalid?	Rationale
<code>outbus = mystruct;</code>	Valid	Both <code>outbus</code> and <code>mystruct</code> have the same number, type, and size of fields. The structure <code>outbus</code> is defined by the <code>Simulink.Bus</code> object <code>MainBus</code> and <code>mystruct</code> is defined locally to match the field properties of <code>MainBus</code> .
<code>outbus = inbus;</code>	Valid	Both <code>outbus</code> and <code>inbus</code> are defined by the same <code>Simulink.Bus</code> object, <code>MainBus</code> .

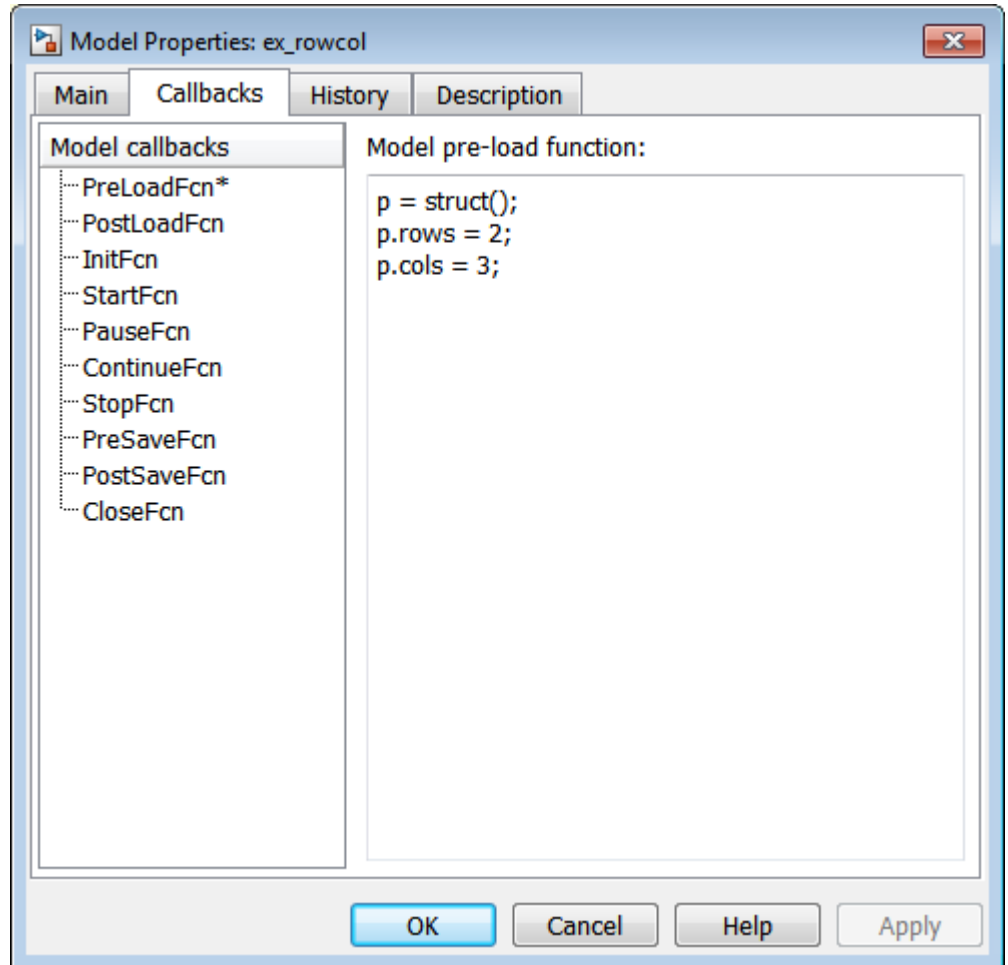
Assignment	Valid or Invalid?	Rationale
<code>outbus1 = inbus.ele3;</code>	Valid	Both <code>outbus1</code> and <code>inbus.ele3</code> have the same type and size because each is defined by the <code>Simulink.Bus</code> object <code>SubBus</code> .
<code>outbus1 = inbus;</code>	Invalid	The structure <code>outbus1</code> is defined by a different <code>Simulink.Bus</code> object than the structure <code>inbus</code> .

Initialize a Matrix Using a Non-Tunable Structure Parameter

The following simple example uses a non-tunable structure parameter input to initialize a matrix output. The model looks like this:



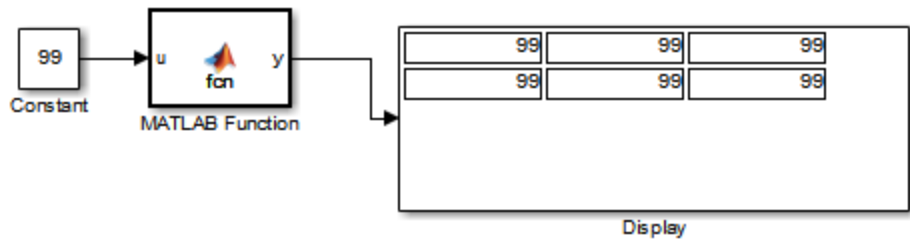
This model defines a structure variable `p` in its pre-load callback function, as follows:



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`:



Define and Use Structure Parameters

In this section...

“Defining Structure Parameters” on page 29-93

“FIMATH Properties of Non-Tunable Structure Parameters” on page 29-93

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. For other methods, see “Structure Parameters” on page 24-17.

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 non-tunable and preserve the initial value during simulation
Type	Select Inherit: Same as Simulink

3 Click **Apply**.

FIMATH Properties of Non-Tunable Structure Parameters

FIMATH properties for non-tunable 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.)

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 variable-size data with arrays of buses (see “Array of Buses Limitations” on page 51-83).

What Is Variable-Size Data?

Variable-size data is data whose size may change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time, and therefore cannot change at run time.

How MATLAB Function Blocks Implement Variable-Size Data

You can define variable-size arrays and matrices as inputs, outputs, and local data in MATLAB Function blocks. However, the block must be able to determine the upper bounds of variable-size data at compile time.

For more information about using variable-size data in Simulink, see “Variable-Size Signal Basics” on page 52-2.

Enable Support for Variable-Size Data

Support for variable-size data is enabled by default for MATLAB Function blocks. To modify this property for individual blocks:

- 1** In the MATLAB Function Block Editor, select **Edit Data**.
- 2** Select or clear the check box **Support variable-size arrays**.

Declare Variable-Size Inputs and Outputs

- 1 In the MATLAB Function Block Editor, select **Edit Data**.
- 2 Select **Add > Data**
- 3 Select the **Variable size** check box.
- 4 Set **Scope** as either Input or Output.
- 5 Enter size:

For:	What to Specify
Input	Enter -1 to inherit size from Simulink or specify the explicit size and upper bound. For example, enter [2 4] to specify a 2-D matrix where the upper bounds are 2 for the first dimension and 4 for the second.
Output	Specify the explicit size and upper bound.

Filter a Variable-Size Signal

In this section...

“About the Example” on page 29-100

“Simulink Model” on page 29-100

“Source Signal” on page 29-101

“MATLAB Function Block: uniquify” on page 29-101

“MATLAB Function Block: avg” on page 29-103

“Variable-Size Results” on page 29-104

About the Example

The following example appears throughout this section to illustrate how MATLAB Function blocks exchange variable-size data with other Simulink blocks. The model uses a variable-size vector to store the values of a white noise signal. The size of the vector may vary at run time as 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 <code>uniquify</code>	Filters out signal values that are not unique to within a specified tolerance of each other.

Simulink Block	Description
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 <code>uniquify</code> function.
Average values	Scope that displays the average signal values output from the <code>avg</code> function.

Source Signal

The band-limited white noise signal has these properties:

Parameters

Noise power:

Sample time:

Seed:

Interpret vector parameters as 1-D

The size of the noise power value defines the size of the matrix that holds the signal values — in this case, 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 = uniquetol(u,0.2);
```

The `uniquify` function calls an external MATLAB function `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 `uniquetol`:

```
function B = 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
```

`uniquetol` returns the filtered values of `A` in an output vector `B` so that $\text{abs}(B(i) - B(j)) > \text{tol}$ 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, `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, `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 sized so it can pass the values returned from `uniquetol` to the **Unique values** scope. Here are the properties of `y`:

Data y

General Description

Name: y

Scope: Output Port: 1

Data must resolve to Simulink signal object

Size: [1 9] Variable size

Complexity: Inherited

Sampling mode: Sample based

Type: Inherit: Same as Simulink

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:	The MATLAB Function block:
> 1 and divisible by 2	Averages every consecutive pair of values
> 1 but <i>not</i> 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

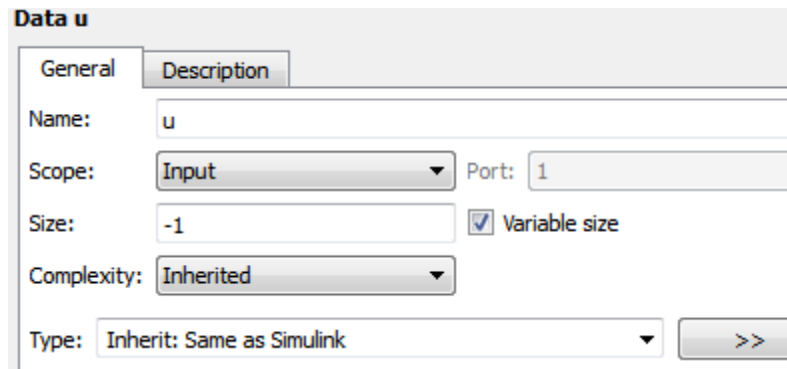
if numel(u) == 1
    y = u;
else
```

```

    k = numel(u)/2;
    if k ~= floor(k)
        u = u(2:numel(u));
    end
    y = nway(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 `nway` to calculate the average of every two consecutive signal values. Here is the code for `nway`:

```

function B = nway(A,n) %#codegen

assert(n>=1 && n<=numel(A));

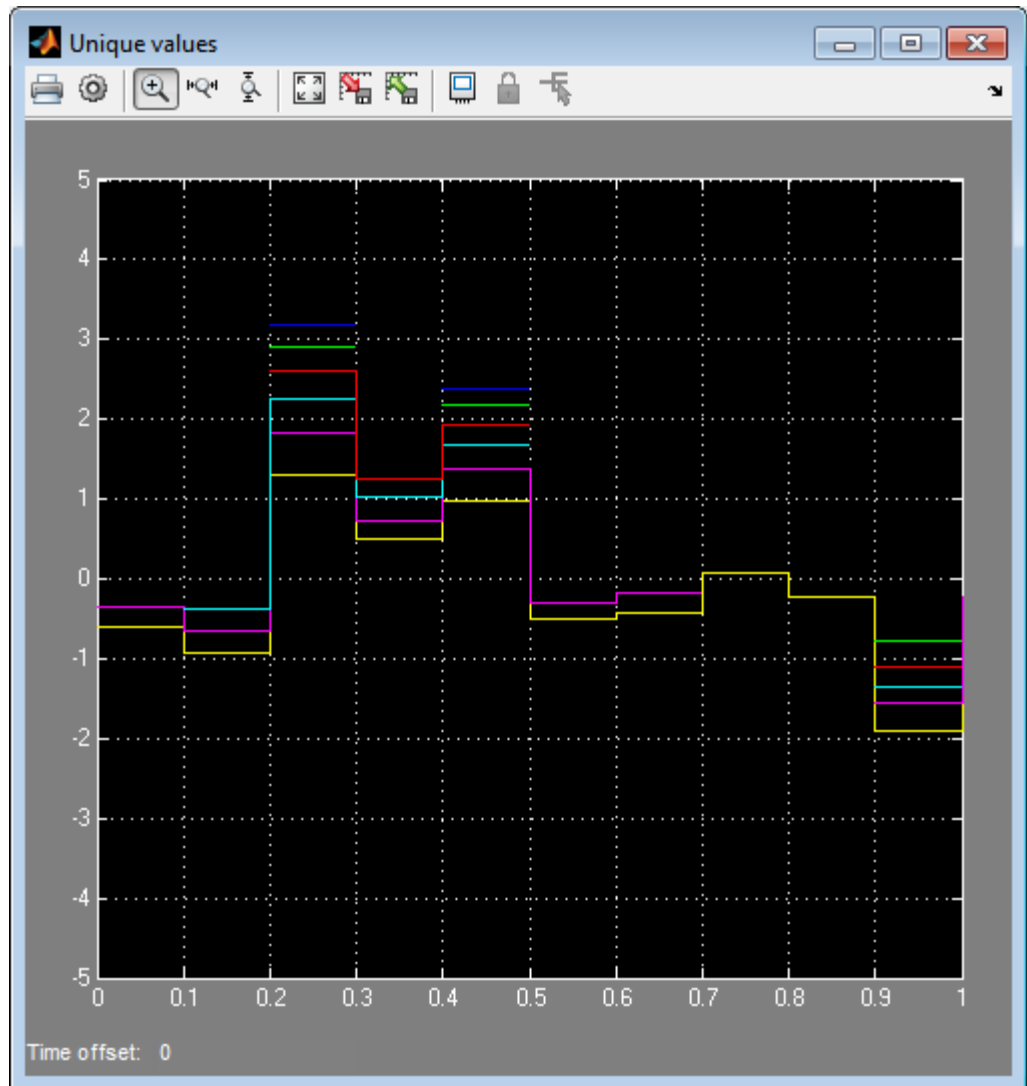
B = zeros(1,numel(A)/n);
k = 1;
for i = 1 : numel(A)/n
    B(i) = mean(A(k + (0:n-1)));
    k = k + n;
end

```

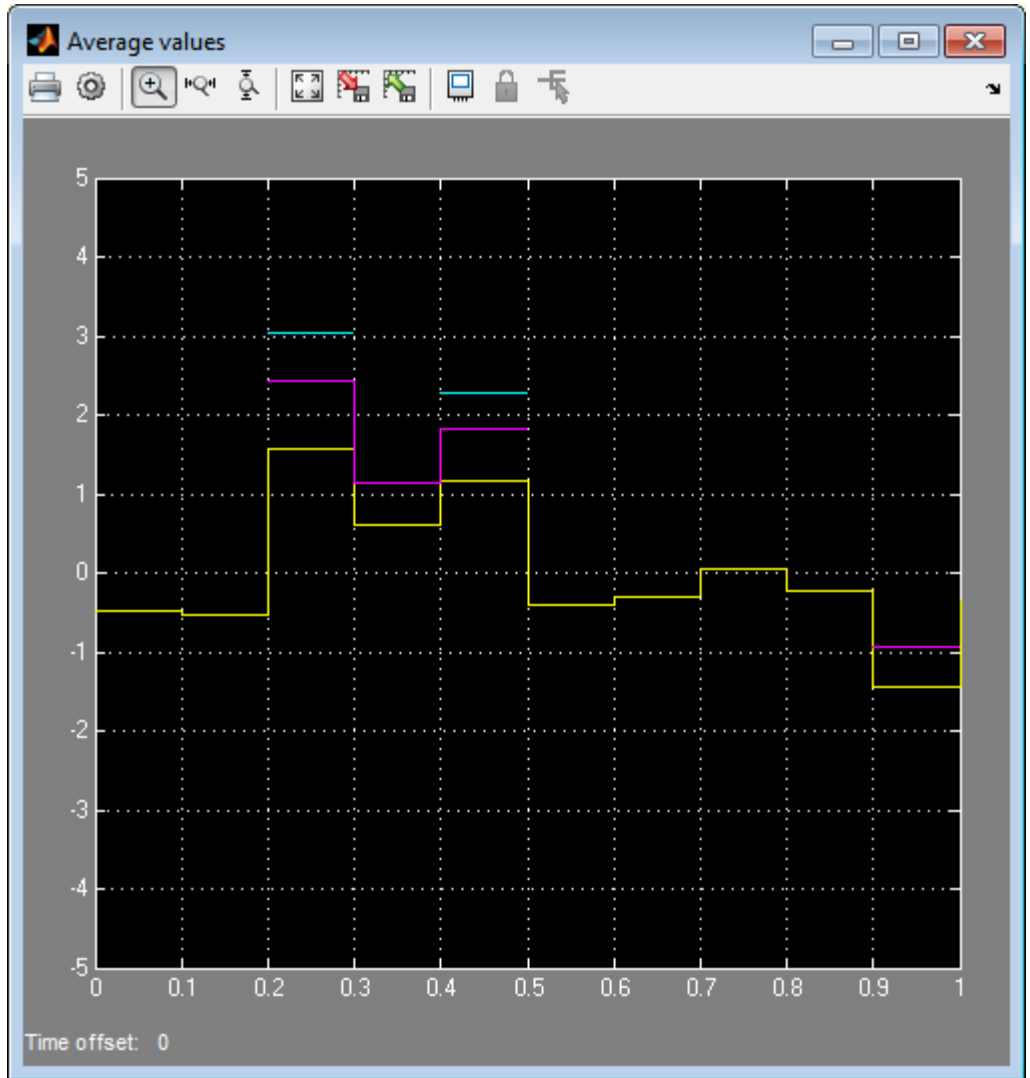
Variable-Size Results

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:



Enumerated Types Supported in MATLAB Function Blocks

Enumerated data is data that has a finite set of values. An enumerated data type is a user-defined type whose values belong to a predefined set of symbols, also called *enumerated values*. Each enumerated value consists of a name and an underlying numeric value.

Like other Simulink blocks, MATLAB Function blocks support an integer-based enumerated type derived from the class `Simulink.IntEnumType`. The instances of the class represent the values that comprise the enumerated type. The allowable values for an enumerated type must be 32-bit integers, but do not need to be consecutive.

For example, the following MATLAB script defines an integer-based enumerated data type named `PrimaryColors`:

```
classdef(Enumeration) PrimaryColors < Simulink.IntEnumType
    enumeration
        Red(1),
        Blue(4),
        Yellow(8)
    end
end
```

`PrimaryColors` is restricted to three enumerated values:

Enumerated Value	Enumerated Name	Underlying Numeric Value
Red(1)	Red	1
Blue(4)	Blue	4
Yellow(8)	Yellow	8

You can exchange enumerated data between MATLAB Function blocks and other Simulink blocks in a model as long as the enumerated type definition is based on the `Simulink.IntEnumType` class.

For comprehensive information about enumerated data support in Simulink, see “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.

Define Enumerated Data Types for MATLAB Function Blocks

You define enumerated data types for MATLAB Function blocks in the same way as for other Simulink blocks:

- 1** Create a class definition file.
- 2** Define enumerated values in an enumeration section.
- 3** Optionally override default methods in a methods section.
- 4** Save the MATLAB file on the MATLAB path.

Add Inputs, Outputs, and Parameters as Enumerated Data

You can add inputs, outputs, and parameters as enumerated data, according to these guidelines:

For:	Do This:
Inputs	Inherit from the enumerated type of the connected Simulink signal or specify the enumerated type explicitly.
Outputs	Always specify the enumerated type explicitly.
Parameters	For tunable parameters, specify the enumerated type explicitly. For non-tunable 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.

To specify an explicit enumerated type:

- a** Select Enum:<class name> from the drop-down menu in the **Type** field.
- b** Replace <class name> with the name of an enumerated data type that you defined in a MATLAB file on the MATLAB path.

For example, you can enter Enum:led in the **Type** field. (See “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.)

Note The **Complexity** field disappears when you select Enum:<class name> because enumerated data types do not support complex values.

For example, the following output ledval has an explicit enumerated type, led:

The screenshot shows the MATLAB/Simulink interface. On the left, the 'Column View' pane displays a table with the following data:

Name	Scope	Port	Resolve Signal	DataType
state	Input	1	<input type="checkbox"/>	Inherit: Same
ledval	Out...	1	<input type="checkbox"/>	Enum: led

On the right, the 'Properties' window is open, showing the 'Description' tab for the 'ledval' signal. The fields are as follows:

- Name: ledval
- Scope: Output
- Port: 1
- Data must resolve to Simulink signal object
- Size: -1
- Sampling mode: Sample based
- Type: Enum: led

To inherit the enumerated type from a connected Simulink signal (for inputs only):

- Select **Inherit:Same** as Simulink from the drop-down menu in the **Type** field.

For example, the following input state inherits its enumerated type switchmode from a Simulink signal:

Name	Scope	Port	Resolve Signal	DataType	Compiled Type
ledval	Output	1	<input type="checkbox"/>	Enum: led	led
state	Input	1	<input type="checkbox"/>	Inherit: Same as Simulink	switchmode

5 Click **Apply**.

Basic Approach for Adding Enumerated Data to MATLAB Function Blocks

Here is the basic workflow for using enumerated data in MATLAB Function blocks:

Step	Action	How?
1	Define an enumerated data type that inherits from <code>Simulink.IntEnumType</code> .	See “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.
2	Add the enumerated data to your MATLAB Function block.	See “Add Inputs, Outputs, and Parameters as Enumerated Data” on page 29-110.
3	Instantiate the enumerated type in your MATLAB Function block.	See “Instantiate Enumerated Data in MATLAB Function Blocks” on page 29-113.
4	Simulate and/or generate code.	See “Enumerations”.

Instantiate Enumerated Data in MATLAB Function Blocks

To instantiate an enumerated type in a MATLAB Function block, use dot notation to specify *ClassName.EnumName*. For example, the following MATLAB function `checkState` instantiates the enumerated types `myMode` and `myLED` from “Control an LED Display” on page 29-114. The dot notation appears highlighted in the code.

```
function led = checkState(state)
    %#codegen

    if state == myMode.ON
        led = myLED.GREEN;
    else
        led = myLED.RED;
    end
```

Control an LED Display

In this section...

“About the Example” on page 29-114

“Class Definition: switchmode” on page 29-114

“Class Definition: led” on page 29-115

“Simulink Model” on page 29-115

“MATLAB Function Block: checkState” on page 29-116

“How the Model Displays Enumerated Data” on page 29-117

About the Example

The following example illustrates how MATLAB Function blocks exchange enumerated data with other Simulink blocks. This simple model uses enumerated data to represent the modes of a device that controls the colors of an LED display. The MATLAB Function block receives an enumerated data input representing the mode and, in turn, outputs enumerated data representing the color to be displayed by the LED.

This example uses two enumerated types: `switchmode` to represent the set of allowable modes and `led` to represent the set of allowable colors. Both type definitions inherit from the built-in type `Simulink.IntEnumType` and must reside on the MATLAB path.

Class Definition: switchmode

Here is the class definition of the `switchmode` enumerated data type:

```
classdef(Enumeration) switchmode < Simulink.IntEnumType
    enumeration
        OFF(0)
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in a MATLAB file with the same name as the class, `switchmode.m`.

Class Definition: led

Here is the class definition of the led enumerated data type:

```
classdef(Enumeration) led < Simulink.IntEnumType
    enumeration
        GREEN(1),
        RED(8),
    end
end
```

This definition must reside on the MATLAB path in a file called `led.m`. The set of allowable values do not need to be consecutive integers.

Simulink Model

Open the example model by typing `emldemo_led_switch` at the MATLAB command prompt. The model contains the following blocks:

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. In the Data Type Conversion block, you specify the enumerated data type using the prefix Enum: followed by the type name. You cannot set a minimum or maximum value for a signal of an enumerated type; leave these fields at the default value []. For this example, the Data Type Conversion block parameters have these settings: <ul style="list-style-type: none"> • Output minimum: []

Simulink Block	Description
	<ul style="list-style-type: none"> • Output maximum: [] • Output data type: Enum:switchmode <p>For more information about specifying enumerated types in Simulink models, see “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.</p>
MATLAB Function checkState	Evaluates enumerated data input state to determine the color to output as enumerated data ledval. See “MATLAB Function Block: checkState” on page 29-116.
Display	Displays the enumerated value of output led.

MATLAB Function Block: checkState

The function checkState in the MATLAB Function block uses enumerated data to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state and 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
```

The input `state` inherits its enumerated type `switchmode` from the Simulink step signal; the enumerated type of output `ledval` is explicitly declared as `Enum:led`:

Name	Scope	Port	Resolve Signal	DataType	Compiled Type
ledval	Output	1	<input type="checkbox"/>	Enum: led	led
state	Input	1		Inherit: Same as Simulink	switchmode

Explicit enumerated type declarations must include the prefix `Enum:`. For more information, see “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.

How the Model Displays Enumerated Data

Wherever possible, Simulink displays the name of an enumerated value, not its underlying integer. For instance, Display blocks display the name of enumerated values. In this example, when the model simulates for less than 10 seconds, the step signal is 0, resulting in a red LED display to signify the off state.

Similarly, if the model simulates for 10 seconds or more, the step signal is 1, resulting in a green LED display to signify the on state.

Simulink scope blocks work differently. For more information, see “Enumerations and Scopes” on page 47-25.

Operations on Enumerated Data

Simulink software prevents enumerated values from being used as numeric values in mathematical computation (see “Operations on Enumerated Data” on page 29-118).

The code generation software supports the following enumerated data operations:

- Assignment (=)
- Relational operations (==, ~=, <, >, <=, >=,)
- Cast
- Indexing

For more information, see “Enumerated Data”.

Using Enumerated Data in MATLAB Function Blocks

In this section...
“When to Use Enumerated Data” on page 29-119
“Limitations of Enumerated Types” on page 29-119

When to Use Enumerated Data

You can use enumerated types to represent program states and to control program logic, especially when you need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow more robust error checking than integers or strings

For example, if you mistype the name of an element in the enumerated type, the code generation software alerts you that the element does not belong to the set of allowable values.

- Produce more efficient code than strings

For example, comparisons of enumerated values execute faster than comparisons of strings.

Limitations of Enumerated Types

Enumerated types in MATLAB Function blocks are subject to the limitations imposed by the code generation software. See “Enumerated Data Definition for Code Generation” on page 40-2.

Share Data Globally

In this section...

“When Do You Need to Use Global Data?” on page 29-120

“Using Global Data with the MATLAB Function Block” on page 29-120

“Choosing How to Store Global Data” on page 29-121

“How to Use Data Store Memory Blocks” on page 29-123

“How to Use Simulink.Signal Objects” on page 29-125

“Using Data Store Diagnostics to Detect Memory Access Issues” on page 29-127

“Limitations of Using Shared Data in MATLAB Function Blocks” on page 29-128

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. (For more information, see “About Data Stores” on page 49-2.) How you store global data depends on how many global variables you are using and the scope of these variables. For more information, see “Choosing How to Store Global Data” on page 29-121.

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 “How to Use Data Store Memory Blocks” on page 29-123 and “How to Use Simulink.Signal Objects” on page 29-125.

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:
<p>Use a small number of global variables in a single model that does not use model reference.</p>	<p>Data Store Memory blocks.</p> <hr/> <p>Note Using Data Store Memory blocks scopes the data to the model.</p>	<p>“How to Use Data Store Memory Blocks” on page 29-123</p>
<p>Use a large number of global variables in a single model that does not use model reference.</p>	<p>Simulink.Signal objects defined in the model workspace. Simulink.Signal objects offer these advantages:</p> <ul style="list-style-type: none"> • 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. 	<p>“How to Use Simulink.Signal Objects” on page 29-125</p>
<p>Share data between multiple models (including referenced models).</p>	<p>Simulink.Signal objects defined in the base workspace</p> <hr/> <p>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.</p>	<p>“How to Use Simulink.Signal Objects” on page 29-125</p>

How to Use Data Store Memory Blocks

- 1 Add a MATLAB Function block to your model.
- 2 Double-click the MATLAB Function block to open its editor.
- 3 Declare a global variable in the MATLAB Function block code, or in any MATLAB file that the MATLAB Function block code calls. For example:

```
global A;
```

- 4 Add a Data Store Memory block to your model and set the following:
 - 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.

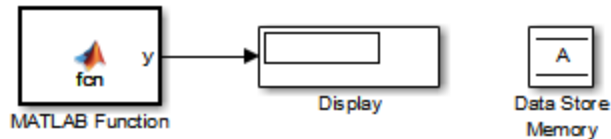
The data type cannot be auto.
 - c Set **Signal type** to real.

The signal type cannot be complex or auto.
 - d Specify an **Initial value**.

The initial value of the Data Store Memory block cannot be unspecified.
- 5 Register the variable to the MATLAB Function block.
 - a In the Ports and Data Manager, add data with the same name as the global variable.
 - b Set the **Scope** of the data to Data Store Memory.

For more information on using the Ports and Data Manager, see “Ports and Data Manager” on page 29-38.

Example: Using Data Store Memory with the MATLAB Function Block



This simple model demonstrates how a MATLAB Function block uses the global data stored in Data Store Memory block A.

1

- 2** Double-click the MATLAB Function block to open the MATLAB Function Block 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;
```

- 3** In the MATLAB Function Block Editor, select **Edit Data**.

- 4** In the Ports and Data Manager, select the data A in the left pane.

The Ports and Data Manager displays the data attributes in the right pane. Note that A has a scope of Data Store Memory.

- 5** In the model, double-click the Data Store Memory block A.

The Block Parameters dialog box opens. Note that A has an initial value of 25.

- 6** 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.

How to Use Simulink.Signal Objects

- 1 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.

- a In the Model Explorer, navigate to **`model_name`** > **Model Workspace** in the **Model Hierarchy** pane.
- b Select **Add > Simulink Signal**.
- c Ensure that these settings apply to the `Simulink.Signal` object:
 - i Set **Data type** to an explicit data type.

The data type cannot be auto.
 - ii Set **Dimensions** to be fully specified.

The signal dimensions cannot be -1 or inherited.
 - iii Set **Complexity** to real.
 - iv Set **Sample mode** to `Sample based`.
 - v Specify an **Initial value**.

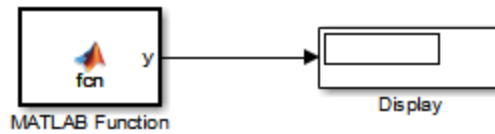
The initial value of the signal cannot be unspecified.

- 2 Register the `Simulink.Signal` object to the MATLAB Function block.
 - a In the Ports and Data Manager, add data with the same name as the `Simulink.Signal` object you created in the model (or base) workspace.
 - b Set the **Scope** of the data to `Data Store Memory`.
- 3 Declare a global variable with the same name in the code for your MATLAB Function block.

```
global Sig;
```

For more information on using the Ports and Data Manager, see “Ports and Data Manager” on page 29-38.

Example: Using a Simulink.Signal Object with a MATLAB Function Block



- 1 Open the `simulink_signal_local` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'simulink', 'examples'))
simulink_signal_local
```

- 2 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;
```

- 3 From the MATLAB Function Block Editor menu, select **Edit Data**.
- 4 In the Ports and Data Manager, select the data A in the left pane.

The Ports and Data Manager displays the data attributes in the right pane. Note that A has a scope of Data Store Memory.

- 5 From the model menu, select **View > Model Explorer**.

- 6** 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.

- 7** Click the `Simulink.Signal` object A.

The right pane displays attributes for A, including.

Attribute	Value
Data type	double
Complexity	real
Dimensions	1
Sample mode	Sample based
Initial value	5

- 8** 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 49-39.

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

Add Frame-Based Signals

In this section...

“About Frame-Based Signals” on page 29-129

“Supported Types for Frame-Based Data” on page 29-129

“Adding Frame-Based Data in MATLAB Function Blocks” on page 29-130

“Examples of Frame-Based Signals in MATLAB Function Blocks” on page 29-131

About Frame-Based Signals

MATLAB Function blocks can input and output frame-based signals in Simulink models. A frame of data is a collection of sequential samples from a single channel or multiple channels. To generate frame-based signals, you must have an available DSP System Toolbox license. For more information about using frame-based signals, see “What Is Frame-Based Processing?”.

MATLAB Function blocks automatically convert incoming frame-based signals as follows:

- Converts single-channel frame-based signals to MATLAB column vectors
- Converts multichannel frame-based signals to two-dimensional MATLAB matrices

An M-by-N frame-based signal represents M consecutive samples from each of N independent channels. N-Dimensional signals are not supported for frames.

To convert matrix or vector data to a frame-based output, use the data property called **Sampling mode** to specify that your output is a frame-based signal for downstream processing.

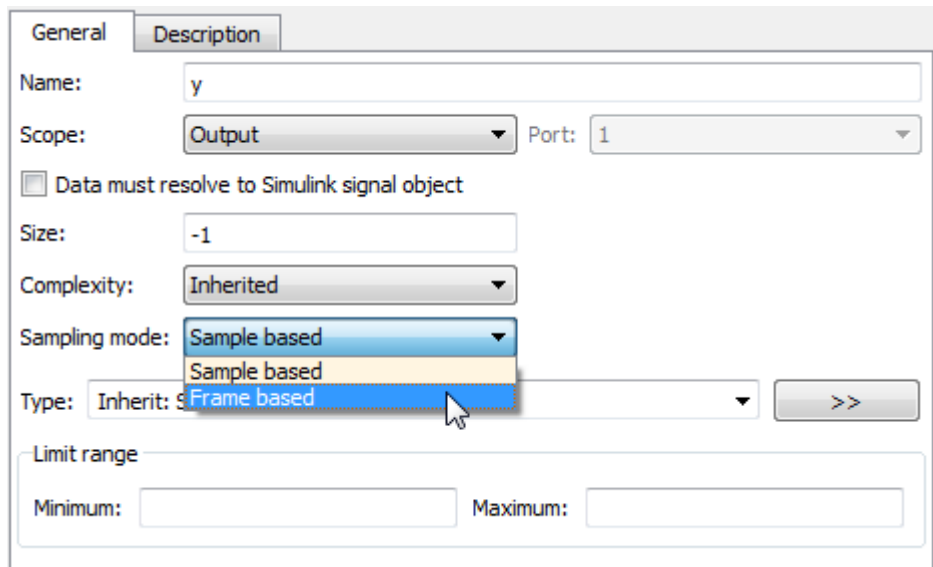
Supported Types for Frame-Based Data

MATLAB Function blocks accept frame-based signals of any data type *except* bus objects. For a list of supported types, see “Supported Variable Types” on page 36-18.

Adding Frame-Based Data in MATLAB Function Blocks

To add frame-based data to a MATLAB Function block, follow these steps:

- 1 Add an input or output, as described in “Adding Data to a MATLAB Function Block” on page 29-43.
- 2 If your data is an output, set **Sampling mode** to Frame based.



The image shows the configuration dialog for a MATLAB Function Block, specifically the 'Description' tab. The 'Name' field is set to 'y'. The 'Scope' is set to 'Output' and the 'Port' is set to '1'. There is a checkbox for 'Data must resolve to Simulink signal object' which is currently unchecked. The 'Size' field is set to '-1'. The 'Complexity' is set to 'Inherited'. The 'Sampling mode' dropdown menu is open, showing two options: 'Sample based' and 'Frame based'. A mouse cursor is pointing at the 'Frame based' option. The 'Type' field is set to 'Inherit: S' and there is a '>>' button next to it. At the bottom, there is a 'Limit range' section with 'Minimum' and 'Maximum' input fields.

Note If your data is an input, **Sampling mode** is not an option.

Note For more information on how to set data properties, see “Defining Data in the Ports and Data Manager” on page 29-44.

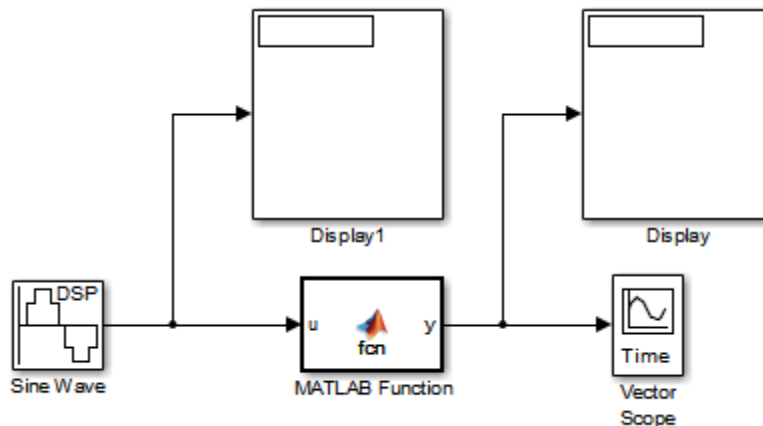
Examples of Frame-Based Signals in MATLAB Function Blocks

This topic presents examples of how to work with frame-based signals in MATLAB Function blocks.

- “Multiplying a Frame-Based Signal by a Constant Value” on page 29-131
- “Adding a Channel to a Frame-Based Signal” on page 29-133

Multiplying a Frame-Based Signal by a Constant Value

In the following example, a MATLAB Function block multiplies all the signal values in a frame-based single-channel input by a constant value and outputs the result as a frame. The input signal is a sine wave that contains 5 samples per frame.

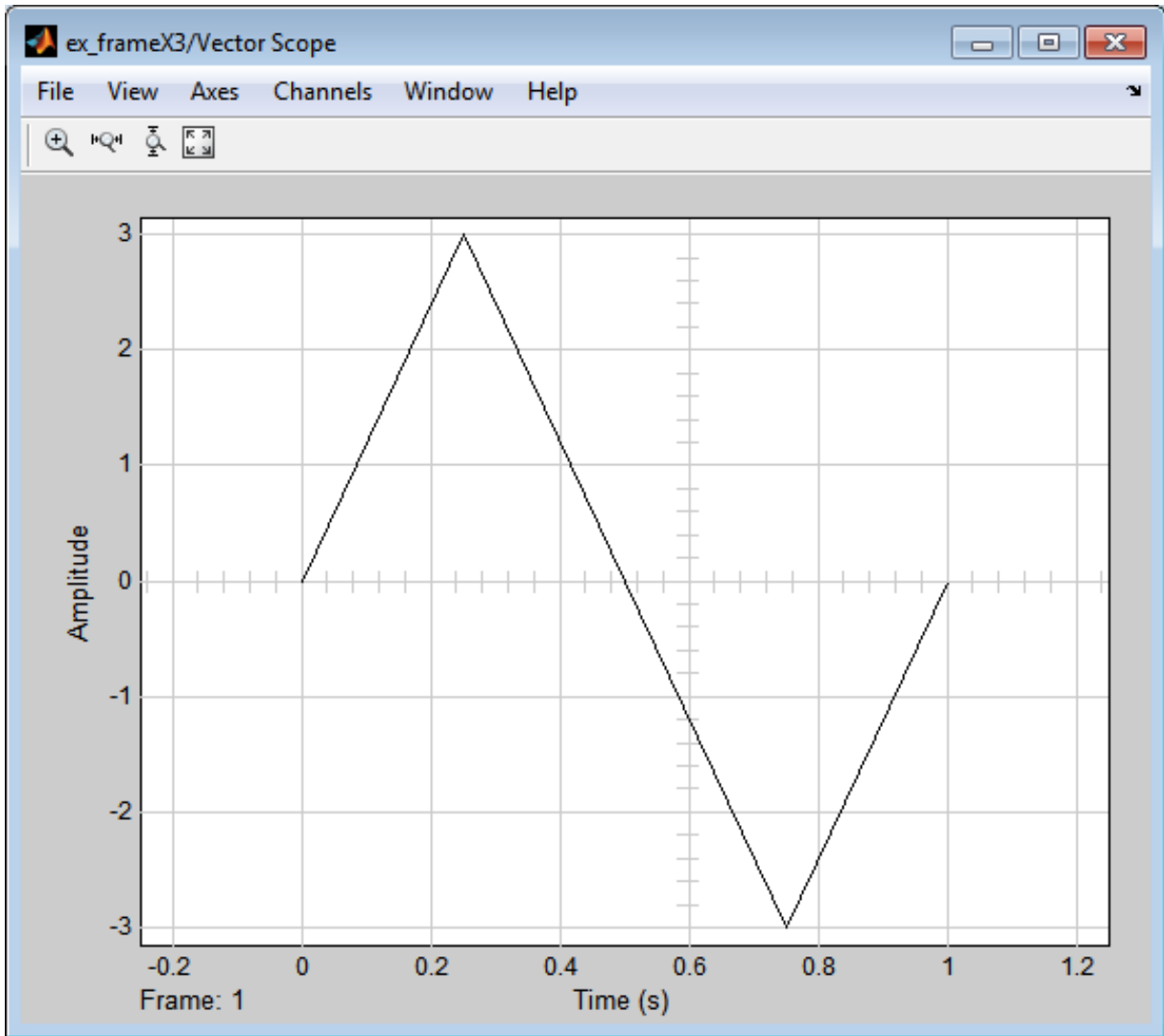


The MATLAB Function block contains the following code:

```
function y = fcn(u)
y = u*3;
```

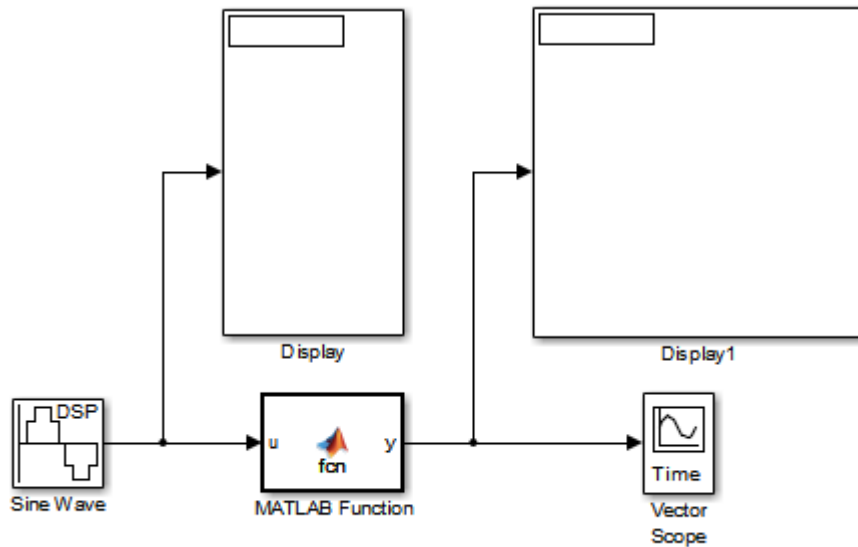
Input u and output y inherit size, complexity, and data type from the input sine wave signal, a 5-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set **Sampling mode** to **Frame based** (see “Adding Frame-Based Data in MATLAB Function Blocks”)

on page 29-130). When you simulate this model, the MATLAB Function block multiplies each input signal by 3 and outputs the result as a frame.



Adding a Channel to a Frame-Based Signal

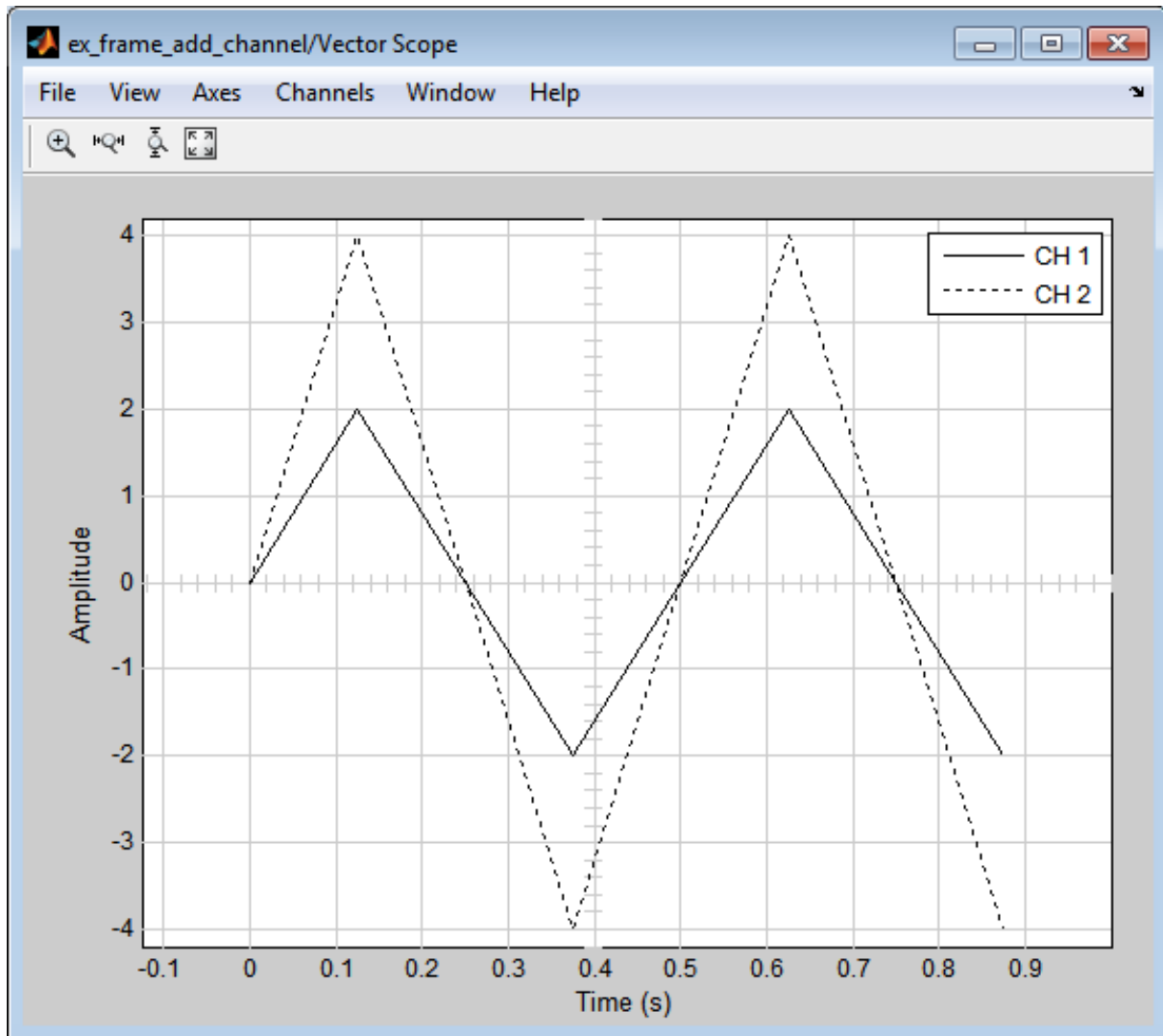
In the following example, a MATLAB Function block adds a channel to a frame-based single-channel input and outputs the multichannel result. The input signal is a sine wave that contains 8 samples per frame.



The MATLAB Function block contains the following code:

```
function y = fcn(u)
a = [0;4;0;-4;0;4;0;-4];
y = [u a];
```

Input u and output y inherit size, complexity, and data type from the input sine wave signal, an 8-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set **Sampling mode** to **Frame based** (see “Adding Frame-Based Data in MATLAB Function Blocks” on page 29-130). Local variable a defines a second column on the matrix which will be output as a frame and interpreted as a second channel by downstream blocks. When you simulate this model, the MATLAB Function block outputs the new multichannel signal.



Create Custom Block Libraries

In this section...

“When to Use MATLAB Function Block Libraries” on page 29-135

“How to Create Custom MATLAB Function Block Libraries” on page 29-135

“Example: Creating a Custom Signal Processing Filter Block Library” on page 29-136

“Code Reuse with Library Blocks” on page 29-149

“Debugging MATLAB Function Library Blocks” on page 29-154

“Properties You Can Specialize Across Instances of Library Blocks” on page 29-154

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 (see “Code Reuse with Library Blocks” on page 29-149).

For more information about Simulink block libraries, see “About Block Libraries and Linked Blocks” on page 28-2.

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 29-136.

- 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 29-154.

- 3** Optionally, customize your library code using masking.

- 4** Add instances of MATLAB Function library blocks to a Simulink model.

Example: Creating a Custom Signal Processing Filter Block Library

- “What You Will Learn” on page 29-136
- “About the Filter Algorithms” on page 29-137
- “Step 1: Add the Filter Algorithms to MATLAB Function Library Blocks” on page 29-137
- “Step 2: Configure Blocks to Inherit Properties You Want to Specialize” on page 29-138
- “Step 3: Customize Your Library Using Masking” on page 29-139
- “Step 4: Add Instances of MATLAB Library Blocks to a Simulink Model” on page 29-144

What You Will Learn

This simple example takes you through the workflow described in “How to Create Custom MATLAB Function Block Libraries” on page 29-135 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 by selecting **File > 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**.

See Also.

- “Ports and Data Manager” on page 29-38
- “Inheriting Argument Data Types” on page 29-69

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 pane open.

- d** In the Icons & Ports pane, add the following icon drawing commands:

```
disp('my_conv');
port_label('output', 1, 'c');
port_label('input', 1, 'a');
port_label('input', 2, 'b');
```

- e** Select the Parameters tab and add a dialog parameter by selecting the Add Parameter icon:



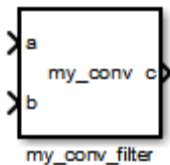
- f** Assign the following properties to the new dialog parameter:

Property	What to Specify
Prompt	Type Shape
Variable	Type shape

Property	What to Specify
Type	<ul style="list-style-type: none"> • Select popup • Enter type-specific options, each on a separate line: <ul style="list-style-type: none"> - full - same - valid • Check Enable parameter • Check Show parameter
Evaluate	Check the box
Tunable	Clear the box <hr/> <p>Note In this example, the shape parameter is not tunable. If in your own application you want to change the value of a mask parameter during simulation, check this box.</p> <hr/>
Tab name	Leave blank

- a 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.

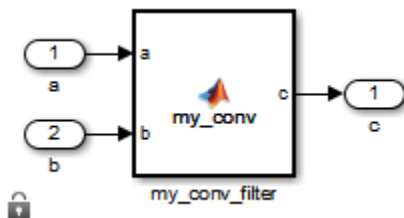
Note This is an optional step, required for this example. If you leave the default setting of **Auto**, the code generation software uses an internal rule to determine whether to inline the function or not.

- 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 29-138.
 - 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.

See Also.

- “Masking”

Step 4: Add Instances of MATLAB Library Blocks to a Simulink Model

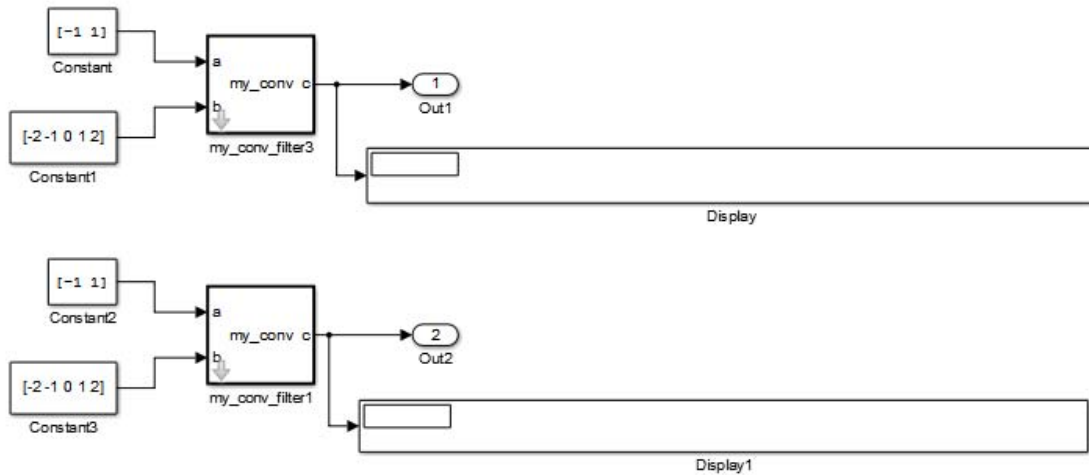
In this exercise, you will add specialized instances of the `my_conv_filter` library block to a simple test model.

- 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 options	<ul style="list-style-type: none"> • 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:



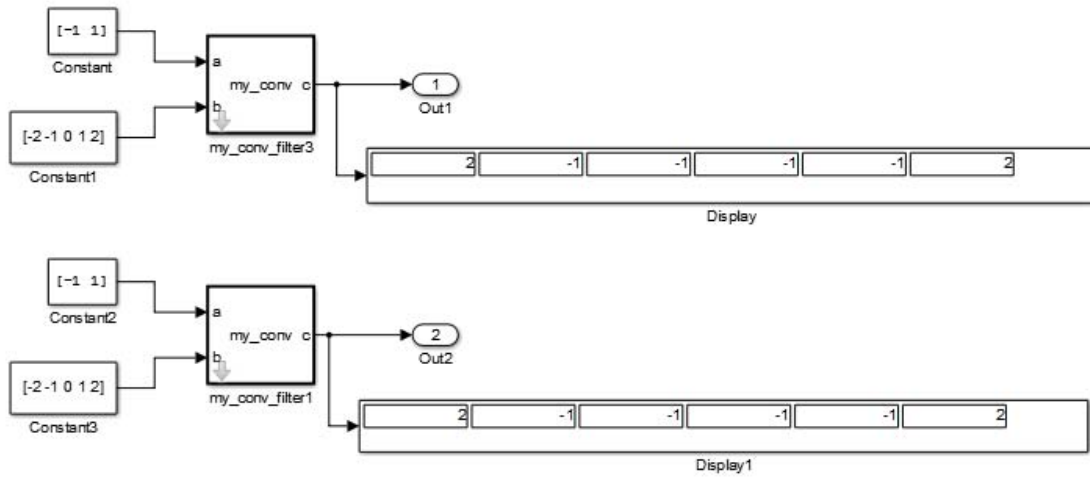
Both library instances share the same size, type, and complexity for inputs a and b respectively.

4 Double-click each library instance.

The shape parameter defaults to **full** for both instances.

5 Simulate the model.

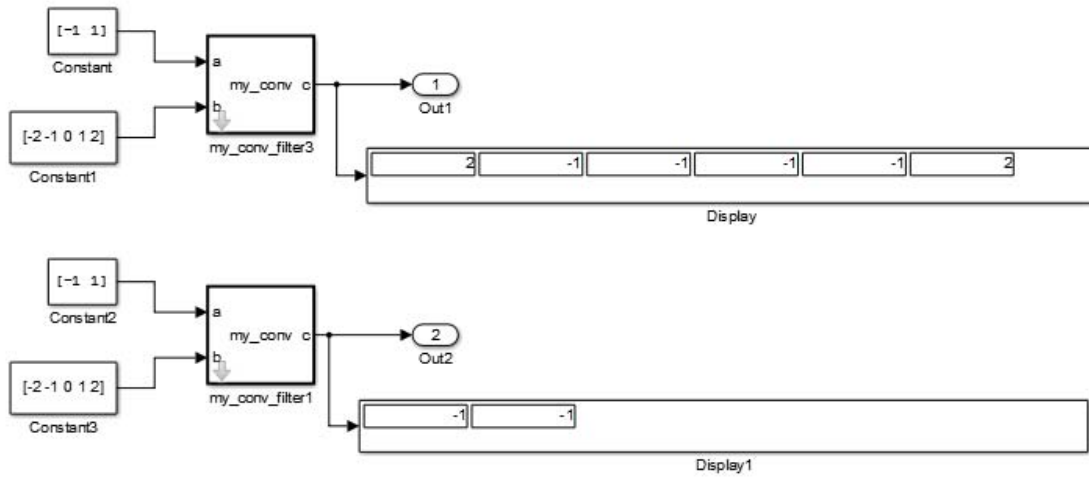
Each library instance outputs the same result, the full 2D convolution:



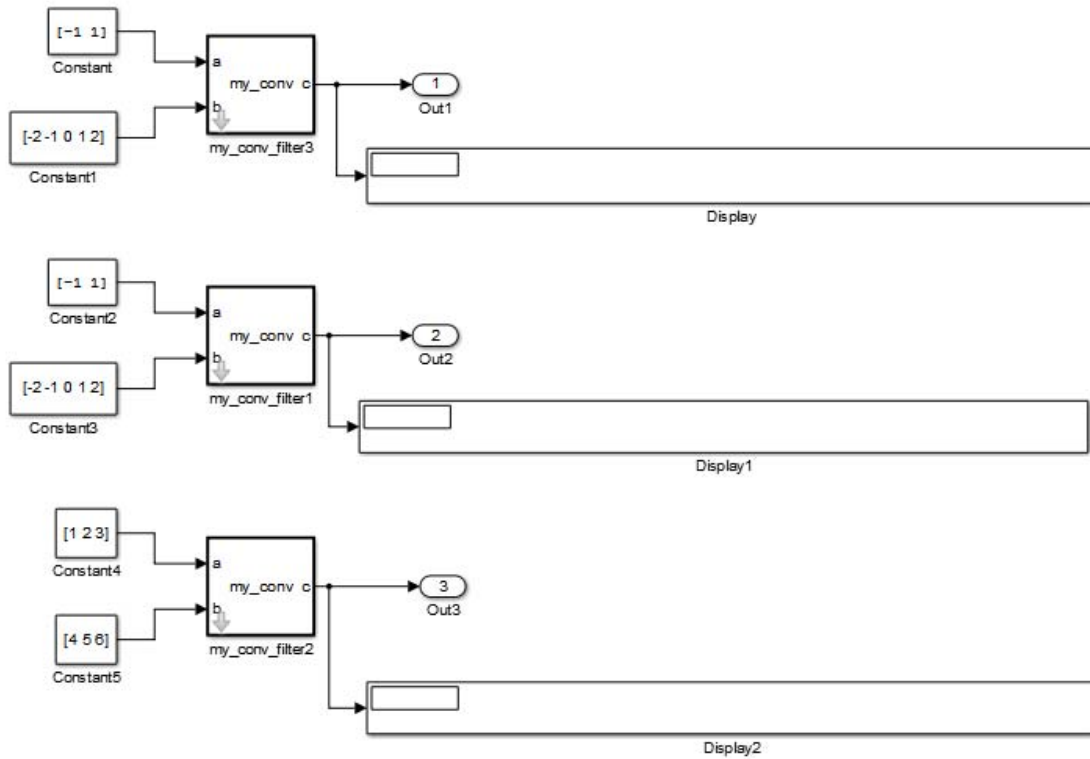
6 Specialize the second instance, `my_conv_filter1` by setting the value of its `shape` parameter to **same**.

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`:

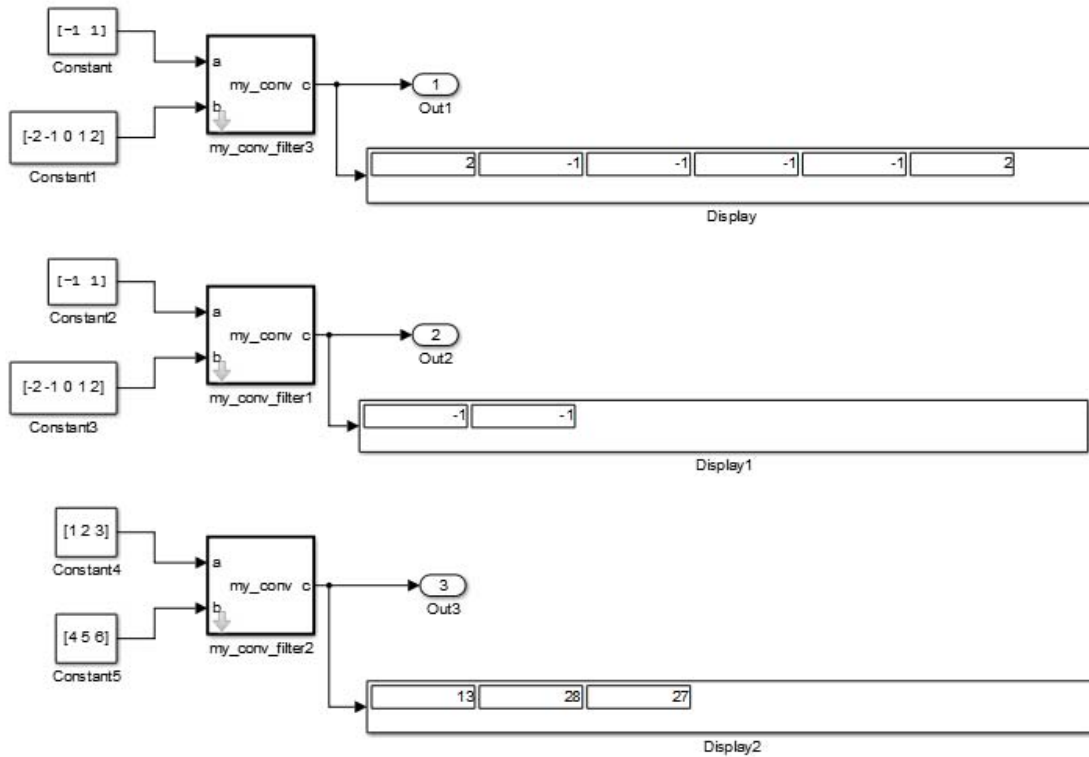


- 8** Now, add a third instance by copying my_conv_filter1. Specialize the new instance, my_conv_filter2, so that it does not inherit the same size inputs as the first two instances:



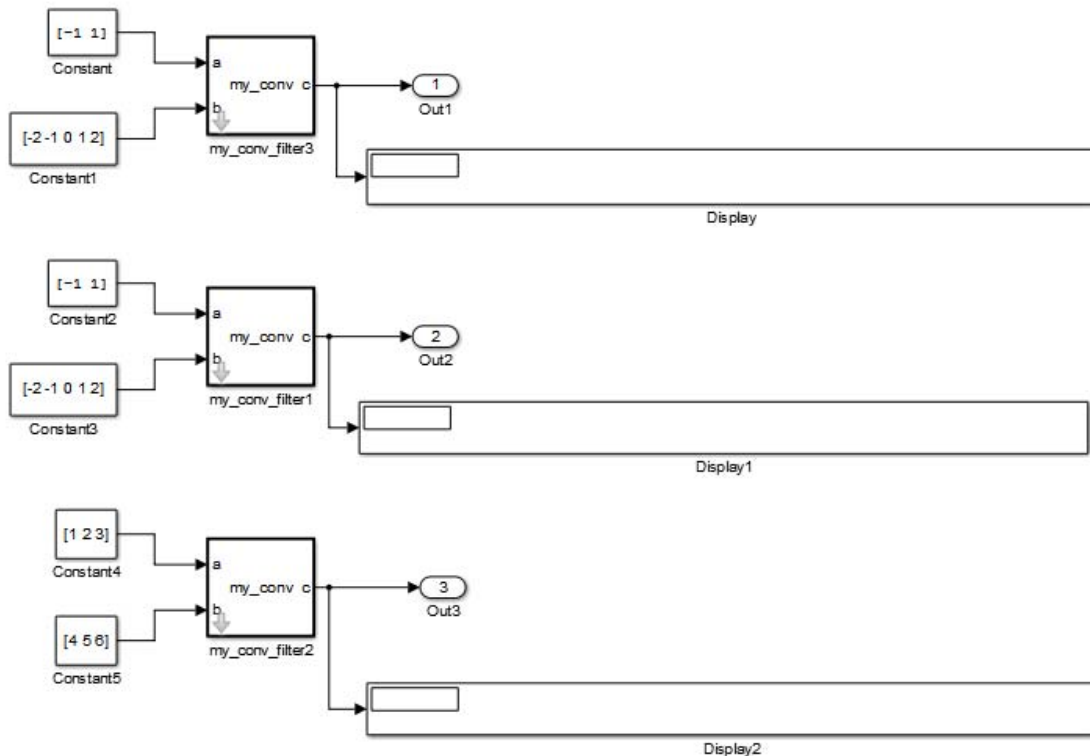
9 Simulate the model again.

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 29-144:



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 <code>ert.tlc</code> 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.

For more information, see “Debugging a MATLAB Function Block” on page 29-24

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 .

Property	Inherits by Default?	How to Specify Inheritance
Limit range	No	Specify minimum and maximum values as Simulink parameters. For example, if minimum value = <code>aParam</code> and maximum value = <code>aParam + 3</code> , different instances of a MATLAB Function library block can resolve to different <code>aParam</code> 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 .

Use Traceability in MATLAB Function Blocks

In this section...

“Extent of Traceability in MATLAB Function Blocks” on page 29-156

“Traceability Requirements” on page 29-156

“Basic Workflow for Using Traceability” on page 29-157

“Tutorial: Using Traceability in a MATLAB Function Block” on page 29-158

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 this option, the MATLAB source code appears immediately after the associated traceability tag. For more information, see “Include MATLAB Code as Comments in Generated Code” on page 29-161.

For information about how traceability works in Simulink blocks, see “About Code Tracing”.

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.

Basic Workflow for Using Traceability

The workflow for using traceability is described in “Trace Model Objects to Generated Code”. Here are the basic steps:

- 1 Open the MATLAB Function block in your Simulink model.
- 2 Define your system target file to be an Embedded Real-Time (ERT) target.
 - a In the model, select **Simulation > Model Configuration Parameters**.
 - b In the **Code Generation** pane, enter `ert.tlc` for the system target file.
- 3 Enable traceability options.
 - a In the **Code Generation > Report** pane, select **Create code generation report**.

This action automatically selects the **Open report automatically** and **Code-to-model** options.
 - b Select **Model-to-code**.

This action automatically selects all options in the **Traceability Report Contents** section.
- 4 Generate the source code and header files for your model.
- 5 Trace a line of code:

To Trace:	Do This:
Line of source code to line of generated code	Right-click in a line in your source code and select Code Generation > Navigate to Code from the context menu
Line of generated code to line of source code	Click a hyperlink in the traceability comment in your generated code

To learn how to complete each step in this workflow, see “Tutorial: Using Traceability in a MATLAB Function Block” on page 29-158

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, double-click the `flame` block to open the MATLAB Function Block Editor.
- 3 In the Simulink model window, select **Simulation > Model Configuration Parameters**.
- 4 In the **Code Generation** pane, go to the **Target selection** section and enter `ert.tlc` for the system target file. Then click **Apply**.

Note Traceability comments appear hyperlinked in generated code only for embedded real-time (`ert`) targets.

- 5 In the **Code Generation > Report** pane, select the **Create code generation report** option.

This action automatically selects the **Open report automatically** and **Code-to-model** options.

- 6 Select the **Model-to-code** option in the **Navigation** section. Then click **Apply**.

This action automatically selects all options in the **Traceability Report Contents** section.

Note For large models that contain over 1000 blocks, disable the **Model-to-code** option to speed up code generation.

- 7** Go to the **Code Generation > Interface** pane. In the **Software environment** section, select the **continuous time** option. Then click **Apply**.

Note Because this example model contains a block with a continuous sample time, you must perform this step before generating code.

- 8** In the **Code Generation** pane, click **Build** in the lower right corner.

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.

```
for (b_x = 0; b_x < 256; b_x++) {
    /* '<S2>:1:19' */
    /* '<S2>:1:21' */
    yb = loopVar_i + 2;

    /* '<S2>:1:22' */
    xb = b_x - 1;
```

Note The line numbers shown above may differ from the numbers that appear in your code generation report.

- 11** Click the `<S2>:1:19` hyperlink in this traceability comment:

```
/* '<S2>:1:19' */
```

Line 19 of the function appears highlighted in the MATLAB Function Block Editor.

- 12** You can also trace a line in a MATLAB function to a line of generated code. For example, right-click in 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`.

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 this option, 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 29-156.

For examples and information on the location of the comments in the generated code, see “Location of Comments in Generated Code” on page 29-162.

- 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 Function Help Text in the Function Banner” on page 29-165.

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, select **Simulation > Model Configuration Parameters**.
- 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.

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);

/* Output: '<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 > MLCfn_U.v) {
    /* Output: '<Root>/y' */
    /* '<S1>:1:4' y = v + 10; */
    MLCfn_Y.y = MLCfn_U.v + 10.0;
} else if (MLFcn_U.u == MLCfn_U.v) {
    /* Output: '<Root>/y' */
    /* '<S1>:1:5' elseif u == v */
    /* '<S1>:1:6' y = u * 2; */
    MLCfn_Y.y = MLCfn_U.u * 2.0;
} else {
    /* Output: '<Root>/y' */
    /* '<S1>:1:7' else */
    /* '<S1>:1:8' y = v - 10; */
    MLCfn_Y.y = MLCfn_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 <= MLCfn_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 Function Help Text in the Function Banner

You can include the function help text in the function banner of the code generated for a MATLAB Function block. 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.

- 1** In the model, select **Simulation > Model Configuration Parameters**.
- 2** In the **Code Generation > Comments** pane, select **MATLAB function help text** and click **Apply**.

Note If the function is inlined, the function help text is also inlined. Therefore, the help text for inlined functions appears in the function body in the generated code even when this option is selected.

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.

Enhance Code Readability for MATLAB Function Blocks

In this section...

“Requirements for Using Readability Optimizations” on page 29-167

“Converting If-Elseif-Else Code to Switch-Case Statements” on page 29-167

“Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements” on page 29-170

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 “Target” and “Control Code Style” in the Embedded Coder documentation.

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 29-172
2	Enable the conversion.	“Enabling the Conversion” on page 29-173
3	Generate code for your model.	“Generating Code for Your Model” on page 29-174

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	<p>Must have two or more <i>unique</i> conditions, in addition to a default.</p> <p>For more information, see “How the Conversion Handles Duplicate Conditions” on page 29-169.</p>
Each condition	<p>Must test equality only.</p> <p>Must use the same variable or expression for the LHS.</p> <hr/> <p>Note You can reverse the LHS and RHS.</p> <hr/>

Construct	Rules to Follow
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.

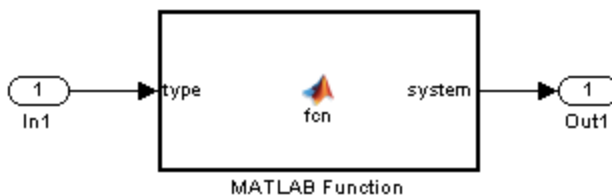
The following examples show how the conversion handles duplicate conditions.

Example of Generated Code	Code After Conversion
<pre> 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 { </pre>	<pre> switch (x) { case 1: block1; break; case 2: block2; break; case 3: block4; break; default: block6; break; </pre>

Example of Generated Code	Code After Conversion
<pre> block6 } </pre>	<pre> } </pre>
<pre> if (x == 1) { block1 } else if (x == 1) { // duplicate block2 } else { block3 } </pre>	<p>No change, because only one unique condition exists</p>

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 “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109.)



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(Enumeration) 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 29-156.

Verifying the Contents of the Block

Check that the block follows all the rules in “Rules for Conversion” on page 29-168.

Construct	How the Construct Follows the Rules
MATLAB Function block	Five unique conditions exist, in addition to the default: <ul style="list-style-type: none"> • (type == Controller.P) • (type == Controller.I) • (type == Controller.PD) • (type == Controller.PI) • (type == Controller.PID)
Each condition	Each condition: <ul style="list-style-type: none"> • Tests equality • Uses the same input for the LHS
Each LHS	Each LHS: <ul style="list-style-type: none"> • Contains a single variable • Is the input to the block and therefore not a constant • Is of enumerated type <code>Controller</code>, which you define in <code>Controller.m</code> on the MATLAB path • Has no side effects on simulation
Each RHS	Each RHS: <ul style="list-style-type: none"> • Is an enumerated value and therefore a constant • Is of enumerated type <code>Controller</code>

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” in the Stateflow documentation.

Generating Code for Your Model

In the **Code Generation** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

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.

Control Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 29-176
“When to Disable Run-Time Checks” on page 29-176
“How to Disable Run-Time Checks” on page 29-177

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**.

The Configuration Parameters dialog box opens with **Simulation Target** selected.

- 3 Clear the **Ensure memory integrity** or **Ensure responsiveness** check boxes, as applicable, and click **Apply**.

Track Object Using MATLAB Code

In this section...

“Learning Objectives” on page 29-178

“Tutorial Prerequisites” on page 29-179

“Example: The Kalman Filter” on page 29-179

“Files for the Tutorial” on page 29-182

“Tutorial Steps” on page 29-184

“Best Practices Used in This Tutorial” on page 29-204

“Key Points to Remember” on page 29-204

“Where to Learn More” on page 29-204

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 29-179
- “Required Products” on page 29-179

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 .

You must set up the C compiler before generating C code. See “Setting Up Your C Compiler” on page 29-185.

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” on page 29-180
- “Algorithm” on page 29-180
- “Filtering Process” on page 29-181

- “Reference” on page 29-182

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 (V_x and V_y), and acceleration (A_x and A_y) 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” on page 29-182
- “Location of Files” on page 29-183
- “Names and Descriptions of Files” on page 29-183

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 29-184.

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.
	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” on page 29-184
- “Setting Up Your C Compiler” on page 29-185
- “About the ex_kalman00 Model” on page 29-185
- “Adding a MATLAB Function Block to Your Model” on page 29-187
- “Checking the ex_kalman11 Model” on page 29-189
- “Simulating the ex_kalman11 Model” on page 29-190
- “Modifying the Filter to Accept a Fixed-Size Input” on page 29-192
- “Using the Filter to Accept a Variable-Size Input” on page 29-197
- “Debugging the MATLAB Function Block” on page 29-199
- “Generating C Code” on page 29-201

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

Before generating code for your Simulink model, you must set up your C compiler. For many platforms, MathWorks supplies a default compiler with MATLAB. If your installation does not include a default compiler, for a list of supported compilers for the current release of MATLAB, see and install a compiler that is suitable for your platform.

To set up a compiler:

- 1 At the MATLAB command line, enter:

```
mex -setup
```
- 2 Enter `y` to see the list of installed compilers.
- 3 Select a supported compiler.
- 4 Enter `y` to verify your choice.

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. See “Using the Current Folder Browser” for more information.

- 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** Select **File > Model Properties > 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 2×310 position matrix, and the index data increments from 1 to 310, the Selector block simply outputs one 2×1 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 `simulink` 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.

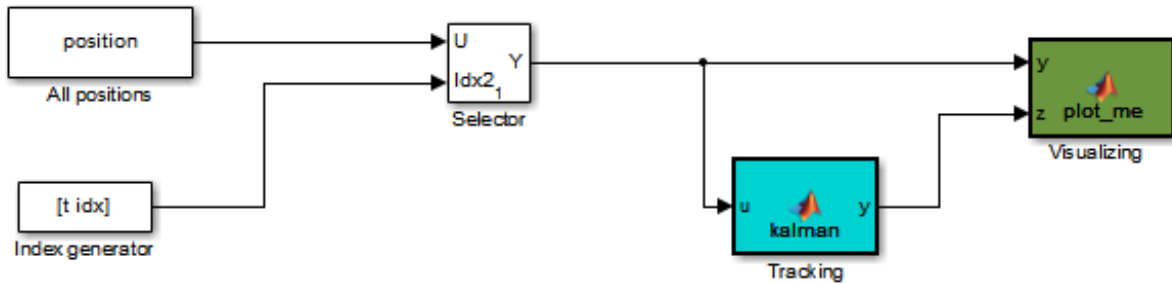
```
function y = kalman(u)
%#codegen

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.



See “Connect Blocks” on page 4-12 for more information.

- 2 Save the model.

You are now ready to check your model for errors, as described in “Checking the ex_kalman11 Model” on page 29-189.

Checking the ex_kalman11 Model

To check the model:

- 1 In the Simulink model window, select **Simulation > Update Diagram**.

Simulink checks the model and generates a warning telling you to add the `%#codegen` compilation directive to the `ex_kalman01` file. Adding this directive indicates that the file is intended for code generation and turns on code generation error checking.

- 2 Open `ex_kalman01` file and add the `%#codegen` compilation directive after the function declaration.

```
function y = ex_kalman01(z) %#codegen
```

- 3 Modify the function name to `ex_kalman02` and save the file as `ex_kalman02.m`.

- 4 Modify the model to call `ex_kalman02` by updating the code in the MATLAB Function block.

```
function y = kalman(u)
%#codegen

y = ex_kalman02(u);
```

- 5 Save the model and update diagram again.

This time the model updates successfully.

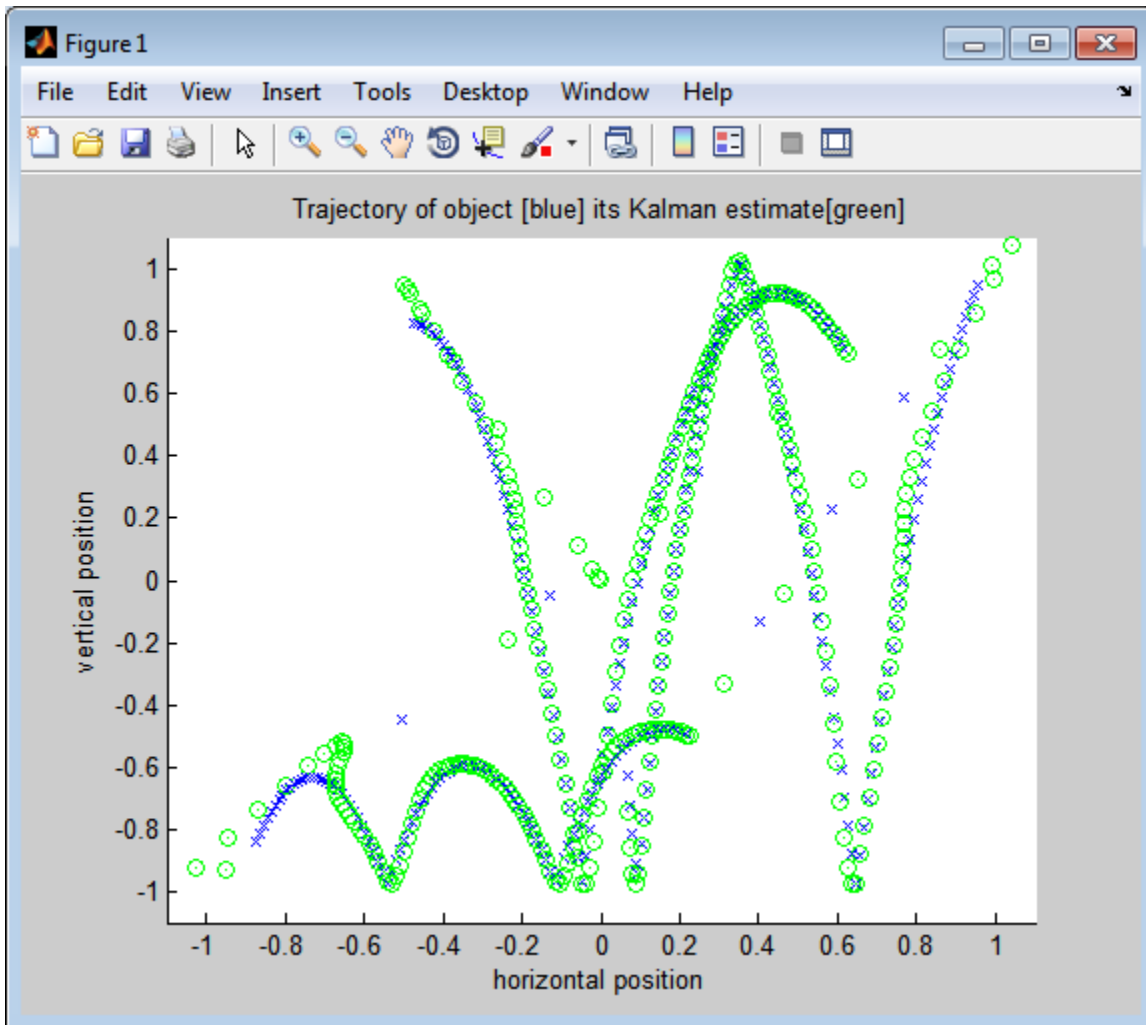
You are now ready to simulate your model, as described in “Simulating the `ex_kalman11` Model” on page 29-190.

Simulating the `ex_kalman11` Model

To simulate the model:

- 1 In the Simulink model window, select **Simulation > 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 29-192.

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 subscripting 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 here because the code generation does not support increasing the size of an array over time. Repeatedly expanding the size of an array over time can adversely affect the performance of your program. See “Preallocating Memory”.

- 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 29-194.

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 Select **File > Model Properties > 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, select **Simulation > Model Configuration Parameters**.
 - 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, select **Simulation > 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 29-197.

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 52-2.

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 Select **File > Model Properties > 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, select **Simulation > 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 29-199.

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, select **Simulation > 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 29-201.

Generating C Code

You have proved that your algorithm works in Simulink. Next you generate code for your model.

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, select **Simulation > Model Configuration Parameters**.

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**.

The **Open report automatically** option is also selected.

- d Click **Apply** and close the Configuration Parameters dialog box.
- e Save your model.

- 3 To generate code for the Tracking block:

- a In your model, select the Tracking block.

- b** In the Simulink model window, select **Code > C/C++ Code > Build Selected Subsystem**.
- 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, select **Simulation > Model Configuration Parameters**.

The Configuration Parameters dialog box opens.
 - In the left pane of the Configuration Parameters dialog box, select **Data Import/Export**.
 - In the right pane, under **Save to workspace options**, set **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”.

- 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 is no code for the `ex_kalman03` function because 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 code for the updated model.

a Select the Tracking block.

b In the model window, select **Code > C/C++Code > 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 block and launches the code generation report.

d In the left pane of the code generation report, click the `Tracking.c` link to view the generated C code.

This time the `ex_kalman04` function has code because you disabled inlining.

```
/* 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.


Where to Learn More

- “Next Steps” on page 29-205
- “Product Help” on page 29-205

Next Steps

To:	See:
Learn how to generate C code from your MATLAB code using codegen	“C Code Generation at the Command Line”
Learn more about code generation from MATLAB	“Getting Started with Simulink Coder”
Use variable-size data	“Variable-Size Data Definition for Code Generation” on page 38-3
Speed up fixed-point MATLAB code	See “Fixed-Point Code Acceleration and Generation Workflow”.
Integrate custom C code into generated code	“Specify External File Locations”
Integrate custom C code into a MATLAB function	<code>coder.ceval</code>
Generate HDL from MATLAB code	www.mathworks.com/products/slhdlcoder

Product Help

MathWorks product documentation is available online from the Open Help Browser button  on the MATLAB toolstrip.

For:	See:
Code generation from MATLAB code	“When to Generate Code from MATLAB Algorithms” on page 33-2
A list of functions that are suitable for code generation	“Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2

Filter Audio Signal Using MATLAB Code

In this section...

- “Learning Objectives” on page 29-206
- “Tutorial Prerequisites” on page 29-206
- “Example: The LMS Filter” on page 29-207
- “Files for the Tutorial” on page 29-210
- “Tutorial Steps” on page 29-211

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. For more information, see “What Is a MATLAB Function Block?” on page 29-6 and “Create Model That Uses MATLAB Function Block” on page 29-9.
- 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 29-207

- “Required Products” on page 29-207

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 “Start the Simulink Software” and “Simulink User Interface”.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- Simulink
- Simulink Coder
- DSP System Toolbox

Note If you do not have a DSP System Toolbox license, see “Track Object Using MATLAB Code” on page 29-178.

- C compiler

For a list of supported compilers, see .

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/C++ Compiler”.

Example: The LMS Filter

- “Description” on page 29-208
- “Algorithm” on page 29-208

- “Filtering Process” on page 29-209
- “Reference” on page 29-210

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 29-210
- “Location of Files” on page 29-210
- “Names and Descriptions of Files” on page 29-211

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 29-212.

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 <code>coder.nullcopy</code> .
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 <code>lms_02.m</code> .
	noise_cancel_03	Simulink model for use with <code>lms_03.m</code> .
	noise_cancel_04	Simulink model for use with <code>lms_04.m</code> .
	noise_cancel_05	Simulink model for use with <code>lms_05.m</code> .
	noise_cancel_06	Simulink model for use with <code>lms_06.m</code> .
design_templates	Simulink model containing Adapt and Reset controls.	

Tutorial Steps

- “Copying Files Locally” on page 29-212

- “Setting Up Your C Compiler” on page 29-213
- “Running the acoustic_environment Model” on page 29-213
- “Adding a MATLAB Function Block to Your Model” on page 29-214
- “Calling Your MATLAB Code As an Extrinsic Function for Rapid Prototyping” on page 29-215
- “Simulating the noise_cancel_01 Model” on page 29-219
- “Modifying the Filter to Use Streaming” on page 29-221
- “Adding Adapt and Reset Controls” on page 29-227
- “Generating Code” on page 29-232
- “Optimizing the LMS Filter Algorithm” on page 29-236

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

Before generating code for your Simulink model, you must set up your C compiler. For most platforms, MathWorks supplies a default compiler with MATLAB. If your installation does not include a default compiler, for a list of supported compilers for the current release of MATLAB, see and install a compiler that is suitable for your platform.

To install a compiler:

- 1 At the MATLAB command line, enter:

```
mex -setup
```

- 2 Enter `y` to see the list of installed compilers.
- 3 Select a supported compiler.
- 4 Enter `y` to verify your choice.

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 “Using the Current Folder Browser” 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, select **Simulation > 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. To learn how, see “Adding a MATLAB Function Block to Your Model” on page 29-214.

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 `simulink` 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 interpreter 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 “Ways to Debug MATLAB Files”.

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.

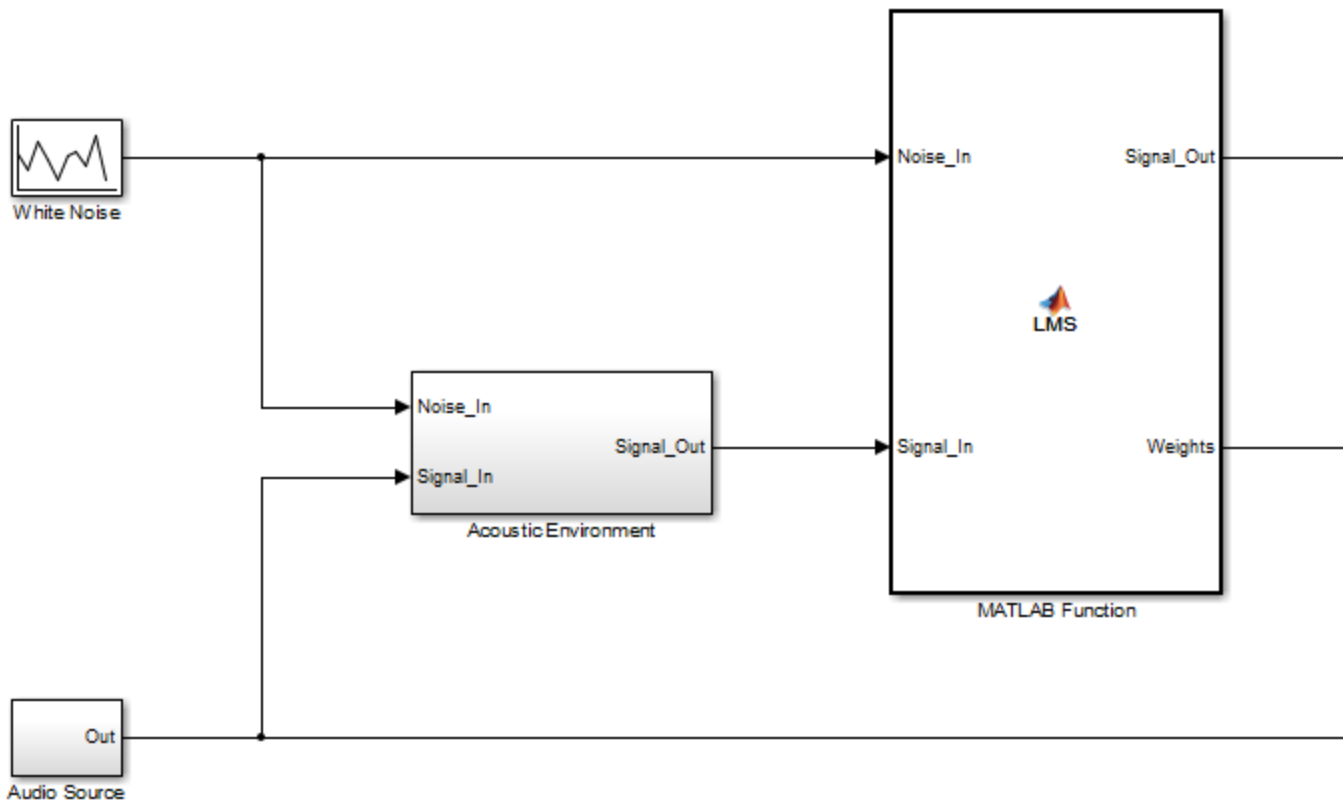
- 4 Save the model.

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.

Noise Cancellation using the LMS Filter



See “Connect Blocks” on page 4-12 for more information.

- 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 “Expected Differences in Behavior After Compiling MATLAB Code” on page 33-9.

- 3** Save the model.

You are now ready to check your model for errors.

Checking the noise_cancel_01 Model.

- 1** In the Simulink model window, select **Simulation > Update Diagram**.

Simulink fails to update diagram.

Because the Analysis and Visualization block expects to receive a frame-based signal from the LMS filter, you must configure the MATLAB Function block `Signal_Out` output parameter to be frame-based rather than sample-based.

- a** Double-click the MATLAB Function block to open the MATLAB Function Block Editor.
- b** Select **Edit Data** to open the Ports and Data Manager.
- c** Select the signal called `Signal_Out` from the list in the left pane.
- d** On the **General** tab, change the **Sampling mode** from `Sample based` to `Frame based`.

- e** Click the **Apply** button and close the Ports and Data Manager and the MATLAB Function Block Editor.
 - f** Save the model.
- 2** Check the model again; select **Simulation > Update Diagram** in the Simulink model window.

Simulink updates diagram successfully. You are ready for the next task, “Simulating the noise_cancel_01 Model” on page 29-219

Simulating the noise_cancel_01 Model

To simulate the model:

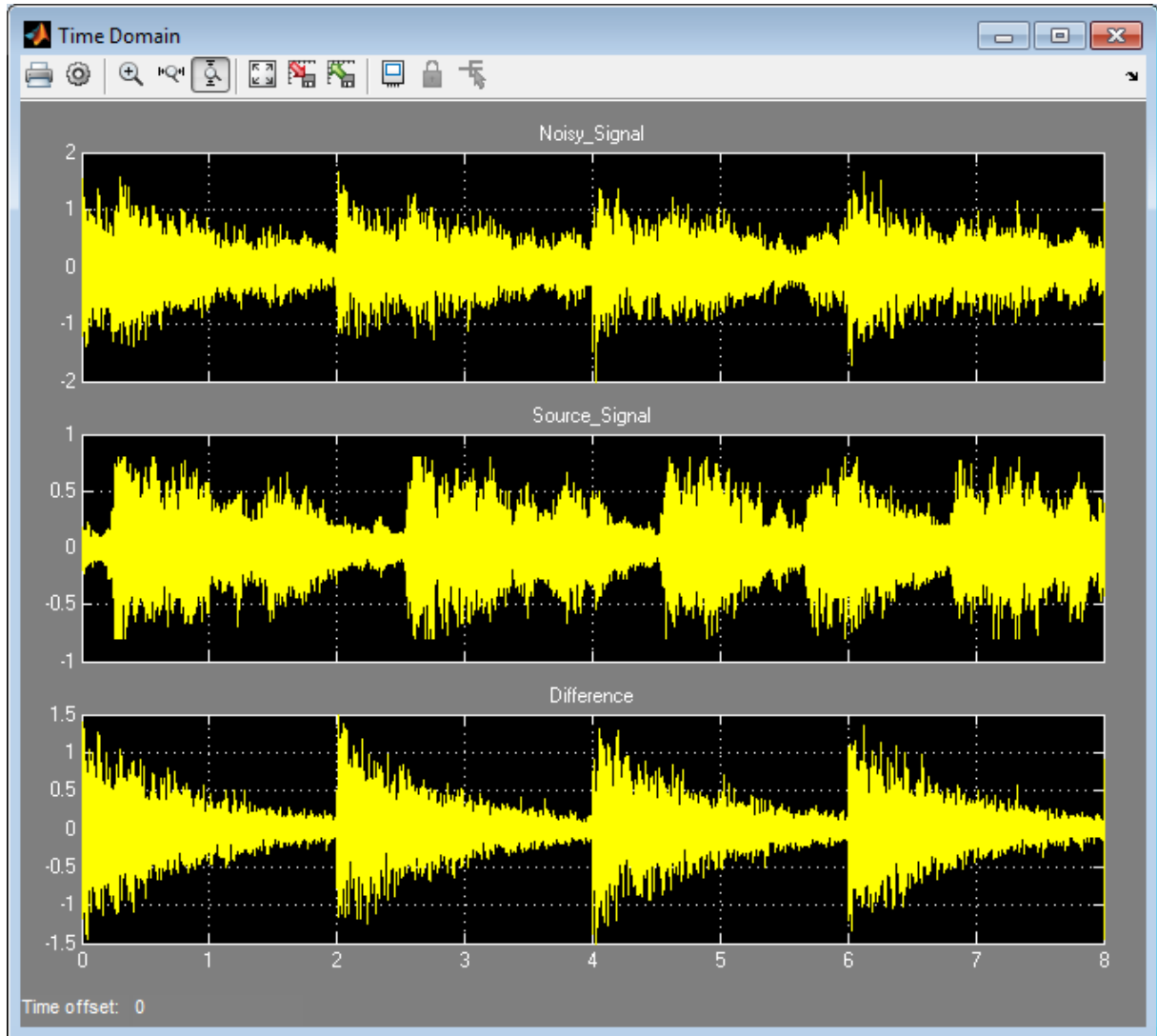
- 1** Ensure that you can see the **Time Domain** plots.

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, select **Simulation > 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 $1/F_s$ 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** from the context menu.
 - b** Select the **Callbacks** tab.
 - c** Select `InitFcn` in the **Model callbacks** pane.

Note that `FrameSize = 16*1024`, which is 16384.

- 3** `F_s` is set in the model `PostLoadFcn` callback. To view this callback, select `PostLoadFcn` in the **Model callbacks** pane:

The following MATLAB commands set up `F_s`:

```
data = load('handel.mat');  
music = data.y;  
F_s = data.Fs;
```

Modifying the Filter to Use Streaming

- “What Is Streaming?” on page 29-222
- “Why Use Streaming?” on page 29-222
- “Viewing the Modified MATLAB Code” on page 29-222
- “Summary of Changes to the Filter Algorithm” on page 29-223
- “Modifying Your Model to Call the Updated Algorithm” on page 29-224

- “Simulating the Streaming Algorithm” on page 29-225

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:

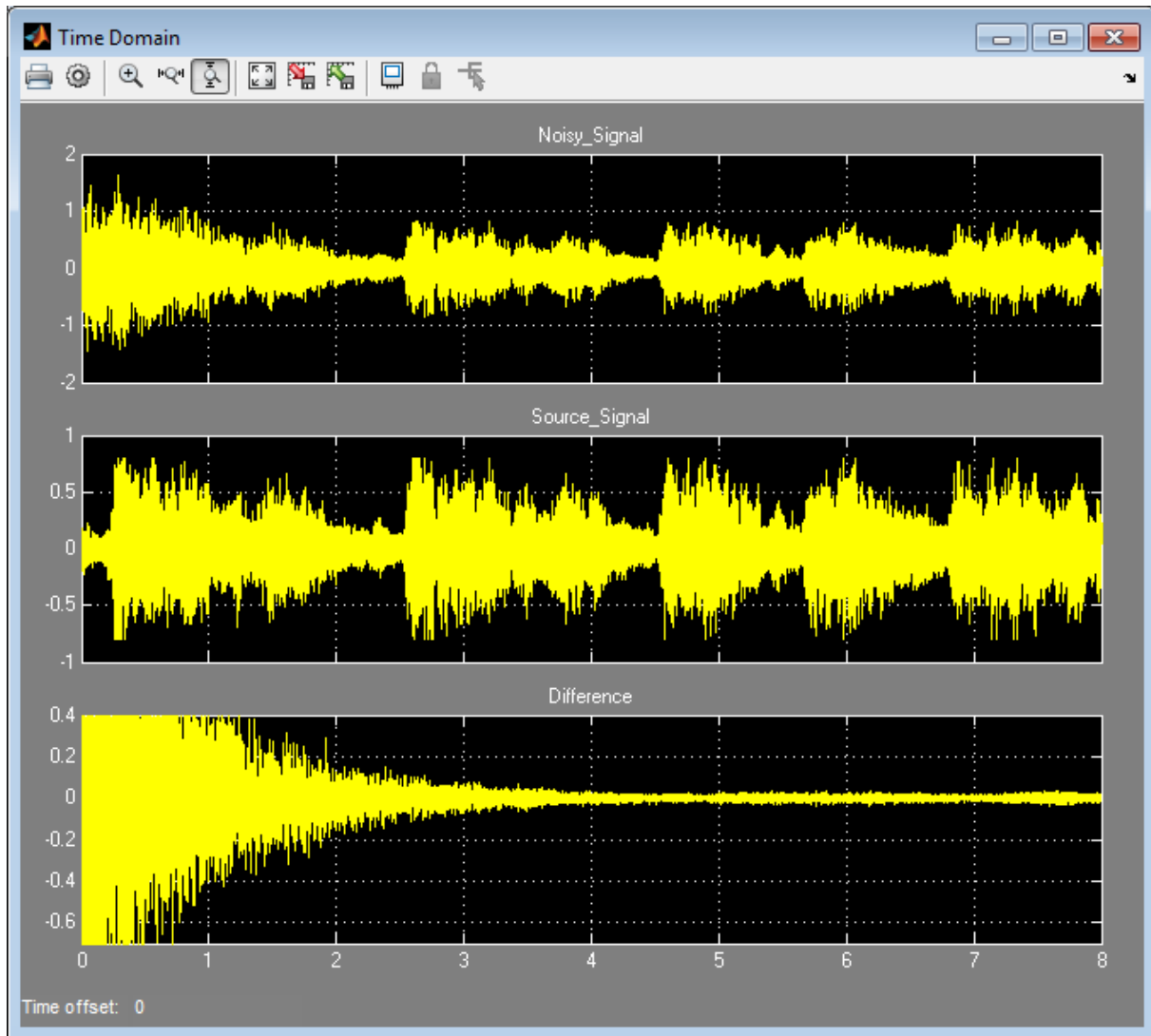
```
function [ Signal_Out, Weights ] = LMS(Noise_In, Signal_In)
% Extrinsic:
coder.extrinsic('lms_02');
% Outputs:
Signal_Out = zeros(size(Signal_In));
Weights = zeros(32,1);
% Compute LMS:
[ ~, Signal_Out, Weights ] = lms_02(Noise_In, Signal_In);
end
```

- 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** from the context menu.
 - 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?” on page 29-227
- “Modifying Your MATLAB Code” on page 29-227
- “Modifying Your Model to Use Reset and Adapt Controls” on page 29-228
- “Simulating the Model with Adapt and Reset Controls” on page 29-230

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');
```

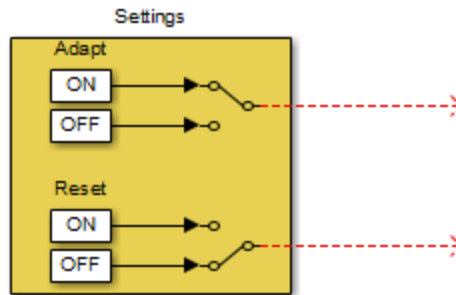
- c** Update the call to the LMS algorithm.

```
% 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.

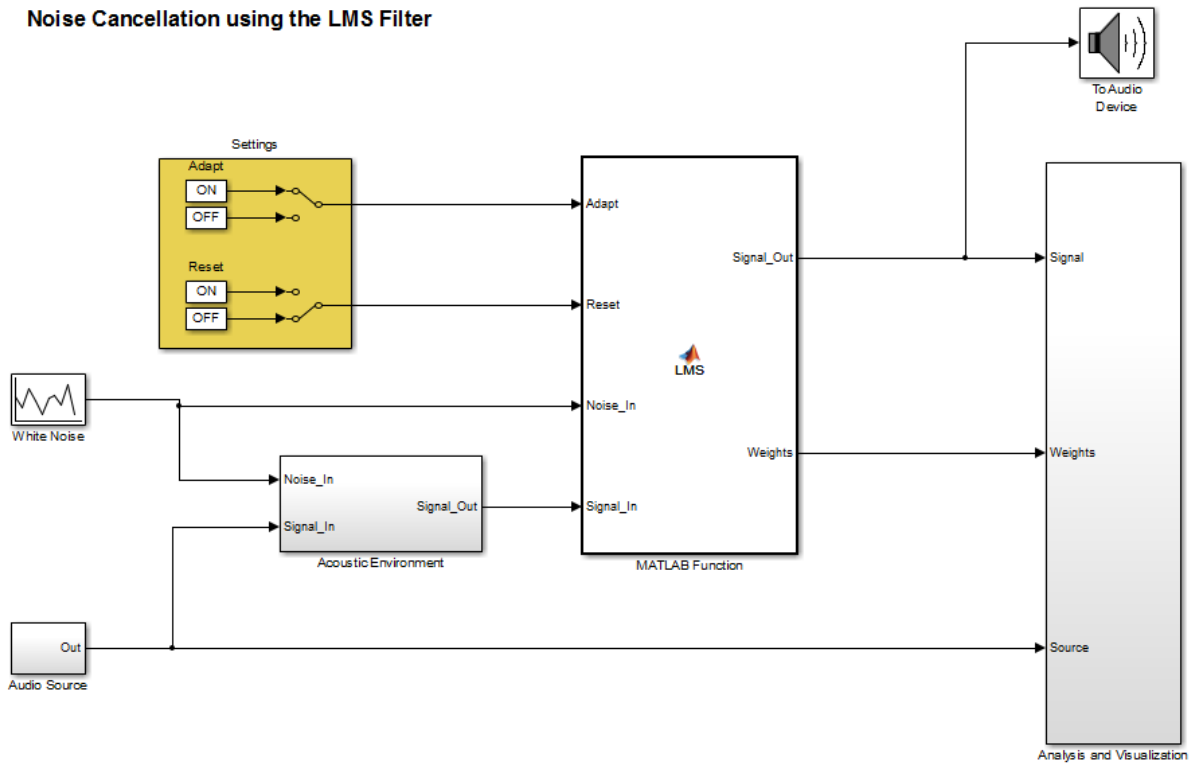


- 5** Copy the Settings block from this model to your `noise_cancel_02` model:

- 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.

Noise Cancellation using the LMS Filter



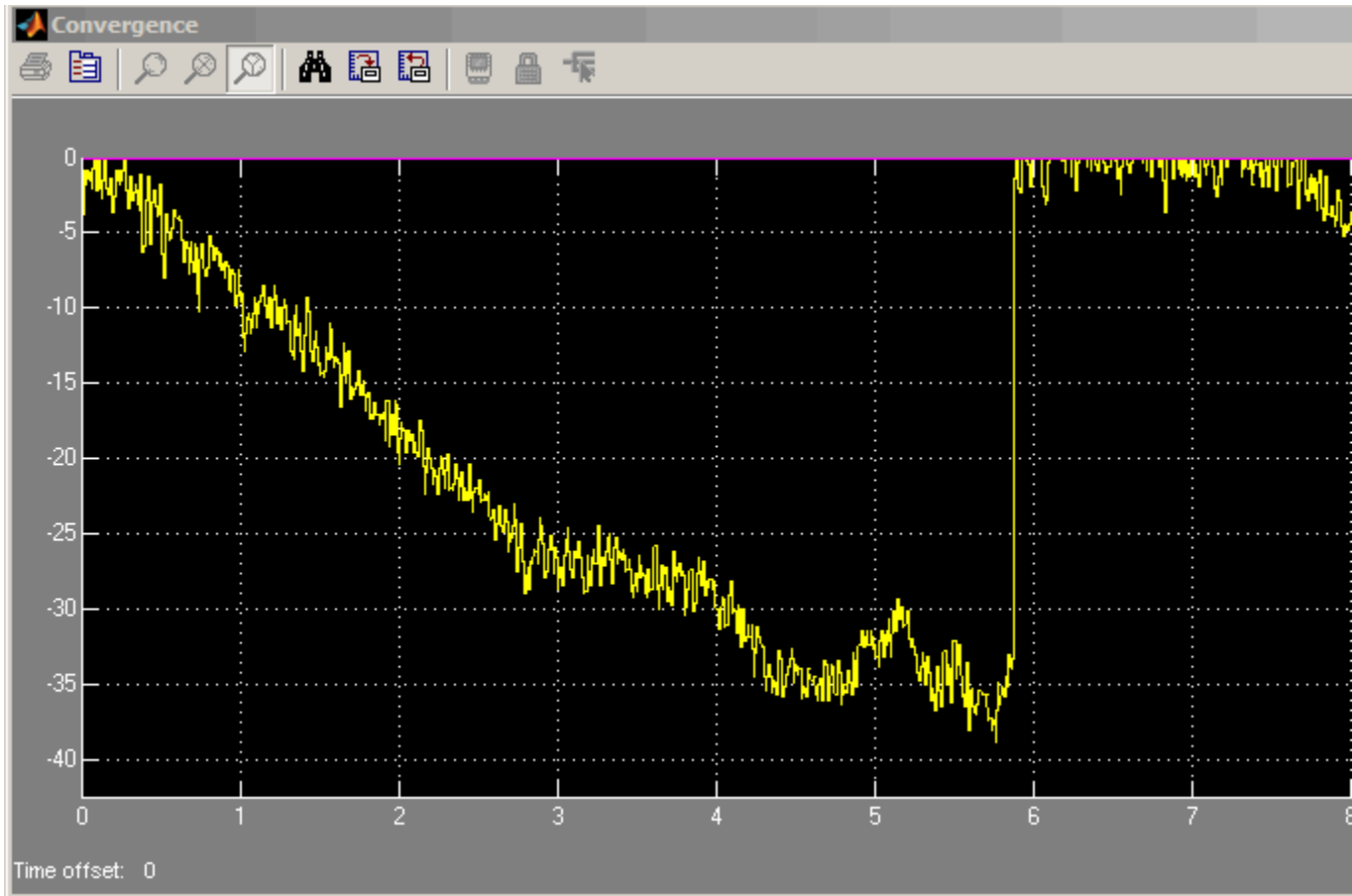
7 Save the model as `noise_cancel_03`.

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, select **Simulation > 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 toggleReset. 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 here because code generation does not support increasing the size of an array over time. Repeatedly expanding the size of an array over time can adversely affect the performance of your program. See “Preallocating Memory”.

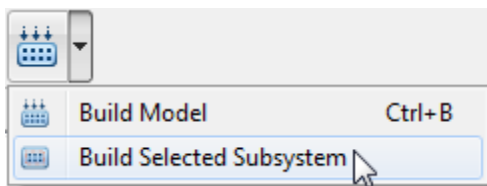
The red error markers for the two lines of code disappear. The code analyzer message indicator in the top right edge of the code turns green,

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, select **Simulation > Model Configuration Parameters**.
 - 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**.
The **Launch report automatically** option is also selected.
 - 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 launches the code generation report.

For more information on using the code generation report, see “Generate a Code Generation Report” in the Simulink Coder documentation.

- 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 launches 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> % Pre-allocate output and error signals: signal_out = zeros(FrameSize,ChannelCount); err = zeros(FrameSize,ChannelCount); </pre>	<pre> /* 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 } </pre>

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 generates C code for the subsystem and launches 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`.

Encapsulating the Interface to External Code

Use the `coder.ExternalDependency` class to encapsulate the interface between external code and MATLAB code intended for code generation. With the encapsulation, you can separate the details of the interface from your MATLAB code. The methods of `coder.ExternalDependency`:

- specify the location of external files
- update build information
- define the programming interface for external functions

In your MATLAB code, you can call the external code without providing build information.

The workflow is:

- 1** Write a class definition file for a class that derives from `coder.ExternalDependency`.
- 2** Store the class definition file in a folder on the MATLAB path.
- 3** In your MATLAB code, use a method of the class to call an external function.
- 4** Generate code from your MATLAB code.

See Also `coder.ExternalDependency` |

Related Examples

- “Encapsulate Interface to an External C Library” on page 29-240

Concepts

- “Best Practices for Using `coder.ExternalDependency`” on page 29-243

Encapsulate Interface to an External C Library

This example shows how to encapsulate the interface to an external C dynamic linked library using `coder.ExternalDependency`.

Write a function `adder` that returns the sum of its inputs.

```
function c = adder(a,b)
    %#codegen
    c = a + b;
end
```

Generate a library that contains `adder`.

```
codegen('adder', '-args', {-2,5}, '-config:dll', '-report');
```

Write the class definition file `AdderAPI.m` to encapsulate the library interface.

```
%=====
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=====

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end
    end
end
```

```
function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

    % Header files
    hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
    buildInfo.addIncludePaths(hdrFilePath);

    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    % Non-build files
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles, '', '');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
    end
end
```

```

        c = coder.ceval('adder', a, b);

        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
end
end
end

```

Write a function `adder_main` that calls the external library function `adder`.

```

function y = adder_main(x1, x2)
%#codegen
    y = AdderAPI.adder(x1, x2);
end

```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```

codegen('adder_main', '-args', {7,9}, '-report')

```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```

copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));

```

For Linux, use:

```

copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));

```

Run the MEX function and verify the result.

```

adder_main_mex(2,3)

```

See Also

`coder.ExternalDependency` | `coder.BuildConfig` | error

Concepts

- “Encapsulating the Interface to External Code” on page 29-239

Best Practices for Using `coder.ExternalDependency`

In this section...

“Terminate Code Generation for Unsupported External Dependency” on page 29-243

“Parameterize Methods for MATLAB and Generated Code” on page 29-243

“Parameterize `updateBuildInfo` for Multiple Platforms” on page 29-244

Terminate Code Generation for Unsupported External Dependency

The `isSupportedContext` method returns true if the external code interface is supported in the build context. If the external code interface is not supported, do not return false. Instead, use `error` to terminate code generation with an error message. For example:

```
function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('MyLibrary is not available for this target');
    end
end
```

Parameterize Methods for MATLAB and Generated Code

Parameterize methods that call external functions so that the methods run in MATLAB. For example:

```
...
if coder.target('MATLAB')
    % running in MATLAB, use built-in addition
    c = a + b;
else
    % running in generated code, call library function
    coder.ceval('adder_initialize');
```

```
end  
...
```

Parameterize updateBuildInfo for Multiple Platforms

Parameterize the `updateBuildInfo` method to support multiple platforms. For example, use `coder.BuildConfig.getStdLibInfo` to get the platform-specific library file extensions.

```
...  
    [~, linkLibExt, execlibExt, ~] = ctx.getStdLibInfo()  
% Link files  
linkFiles = strcat('adder', linkLibExt);  
buildInfo.addLinkObjects(linkFiles, linkPath, linkPriority, ...  
    linkPrecompiled, linkLinkOnly, group);  
...
```

See Also

`coder.ExternalDependency` | `coder.BuildConfig` | `error`

Related Examples

- “Encapsulate Interface to an External C Library” on page 29-240

Update Build Information from MATLAB code

You can choose to control aspects of the build process that occur after code generation but before compilation. For example, you can specify compiler or linker options.

To customize the build from your MATLAB code:

- 1 In your MATLAB code, call `coder.updateBuildInfo` to update the build information object. You specify a build information object method and the input arguments for the method.
- 2 Generate code from your MATLAB code.

See Also `coder.updateBuildInfo`

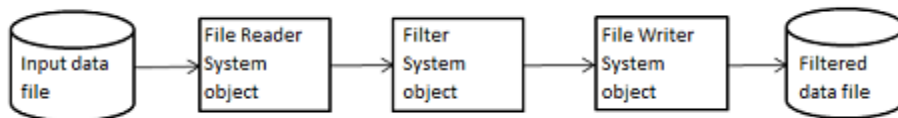
System Objects

- “What Are System Objects?” on page 30-2
- “System Design and Simulation in Simulink” on page 30-4
- “System Objects in MATLAB Code Generation” on page 30-5
- “System Objects in Simulink” on page 30-11
- “System Object Methods” on page 30-12
- “System Design in Simulink Using System Objects” on page 30-15

What Are System Objects?

A System object™ is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer license)
- C code generation (requires a MATLAB Coder or Simulink Coder license)
- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

System Design and Simulation in Simulink

You can use System objects in your model to simulate in Simulink.

- 1** Define new System object to be used in your model. See “Define New Kinds of System Objects for Use in Simulink” on page 30-15 for information.
- 2** Test your new System object in MATLAB. See “Test New System Objects in MATLAB” on page 30-21
- 3** Add the System object to your model using the MATLAB System block. See “Add System Objects to Your Simulink Model” on page 30-21 for information.
- 4** Add other Simulink blocks as needed and connect the blocks to construct your system.
- 5** Run the system

System Objects in MATLAB Code Generation

In this section...

“System Objects in Generated Code” on page 30-5

“System Objects in codegen” on page 30-9

“System Objects in the MATLAB Function Block” on page 30-9

“System Objects in the MATLAB System Block” on page 30-10

“System Objects and MATLAB® Compiler™ Software” on page 30-10

System Objects in Generated Code

You can generate C/C++ code in MATLAB from your system that contains System objects by using the MATLAB Coder product. Using this product, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. You do not need the MATLAB Coder product to generate code in Simulink.

Note Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects Code with Persistent Objects for Code Generation

This example shows how to use System objects to make MATLAB code suitable for code generation. The example highlights key factors to consider, such as passing property values and using extrinsic functions. It also shows that by using persistent objects, the object states are maintained between calls.

```
function w = lmssystem(x, d)
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.
    persistent hlms;
```

```
% Initialize persistent System objects only once
% Do this with 'if isempty(persistent variable).'
% This condition will be false after the first time.

if isempty(hlms)
    % Create LMS adaptive filter used for system
    % identification. Pass property value arguments
    % as constructor arguments. Property values must
    % be constants during compile time.

    hlms = dsp.LMSFilter(11,'StepSize',0.01);
end

[~,~,w] = step(hlms,x,d);    % Filter weights
end
```

This example shows how to compile the `lmssystem` function and produce a MEX file with the same name in the current directory.

```
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter

coefs = fir1(10,.25);
hfilt = dsp.FIRFilter('Numerator', coefs);

x = randn(1000,1);          % Input signal
hSrc = dsp.SignalSource(x,100); % Use x as input-signal with
                                % 100 samples per frame

% Generate code for lmssystem
codegen lmssystem -args {ones(100,1),ones(100,1)}

while ~isDone(hSrc)
    in = step(hSrc);
    d = step(hfilt,in) + 0.01*randn(100,1); % Desired signal
    w = lmssystem_mex(in,d);                % Call generated mex file
    stem([coefs.',w]);
end
```

For another detailed code generation example, see “Generate Code for MATLAB Handle Classes and System Objects” in the MATLAB Coder product documentation.

System Objects Code Without Persistent Objects for Code Generation

Usage Rules and Limitations for System Objects in Generated MATLAB 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.
- 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

- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note Variable-size properties in MATLAB Function block in Simulink are not supported. 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 do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
 - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
 - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation.
- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- Do not use cell arrays.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

```
f = coder.MEXConfig;
```



```
f.GlobalSyncMethod='NoSync'
```

Then, include '-config f' in your `codegen` command.

Methods Supported for Code Generation

- Code generation support is available only for these System object methods:
 - `get`
 - `getNumInputs`
 - `getNumOutputs`
 - `isDone` (for sources only)
 - `release`
 - `reset`
 - `set` (for tunable properties)
 - `step`

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 “Getting Started with MATLAB Coder” and “MATLAB Classes” 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 “What Is a MATLAB Function Block?” on page 29-6 in the Simulink documentation.

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 “What Is the MATLAB System Block?” on page 32-2 in the Simulink documentation.

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.

System Objects in Simulink

In this section...
“System Objects in the MATLAB Function Block” on page 30-11
“System Objects in the MATLAB System Block” on page 30-11

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using 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 using the MATLAB System block. This provides one way to add your own algorithm blocks into your Simulink models. For information, see “System Object Integration” in the Simulink documentation.

System Object Methods

In this section...

“What Are System Object Methods?” on page 30-12

“The Step Method” on page 30-12

“Common Methods” on page 30-13

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The step method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object.

For more information about the `step` method and other available methods, see the descriptions in “Common Methods” on page 30-13.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
<code>step</code>	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the <code>step</code> method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The <code>step</code> method returns regular MATLAB variables. Example: <code>Y = step(H,X)</code>
<code>release</code>	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the <code>release</code> method instead of a destructor.
<code>reset</code>	Resets the internal states of the object to the initial values for that object
<code>getNumInputs</code>	Returns the number of inputs (excluding the object itself) expected by the <code>step</code> method. This number varies for an object depending on whether any properties enable additional inputs.
<code>getNumOutputs</code>	Returns the number of outputs expected from the <code>step</code> method. This number varies for an object depending on whether any properties enable additional outputs.

Method	Description
<code>getDiscreteState</code>	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the <code>step</code> method on it or after you have released the object), the states are empty. If the object has no discrete states, <code>getDiscreteState</code> returns an empty structure.
<code>clone</code>	Creates another object of the same type with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object is locked.
<code>isDone</code>	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns <code>false</code> .
<code>info</code>	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design in Simulink Using System Objects

In this section...

“Define New Kinds of System Objects for Use in Simulink” on page 30-15

“Test New System Objects in MATLAB” on page 30-21

“Add System Objects to Your Simulink Model” on page 30-21

Define New Kinds of System Objects for Use in Simulink

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 “System Object Integration” in the Simulink documentation. The LMS Adaptive Filter and Integer Delay blocks in this example model are each from a System object class definition file. These files are described below.

Define System Object with Block Customizations

- 1 Create a class definition text file to define your System object. This example creates a least mean squares (LMS) filter and includes customizations to the block icon and dialog appearance.
- 2 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.
- 3 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` to reset the state properties or `DiscreteState` properties.
 - Use the `getNumInputsImpl` and `getNumOutputsImpl` methods to specify the number of inputs and outputs, respectively.
- 4** 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.
 - Use the `getPropertyGroupsImpl` method to specify groups of properties to display on the block dialog.
 - 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

        function num = getNumInputsImpl(~)
            num = 2;
        end
    function num = getNumOutputsImpl(~)
        num = 2;
    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

```

```

end

function groups = getPropertyGroupsImpl
    upperGroup = matlab.system.display.SectionGroup(...
        'Title', 'General', ...
        'PropertyList', {'Mu'}); %#ok<*EMCA>

    lowerGroup = matlab.system.display.SectionGroup(...
        'Title', 'Coefficients', ...
        'PropertyList', {'Weights', 'N'}); %#ok<*EMCA>

    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
end

```

Define System Object with Nondirect Feedthrough

- 1 Create a class definition text file to define your System object. This example creates an integer delay and includes customizations to the block icon. It implements a System object that you can use for nondirect feedthrough. See “Use System Objects in Feedback Loops” on page 32-13 for more information.
- 2 On the first line of the class definition file, subclass from `matlab.System`, `matlab.system.mixin.CustomIcon`, and `matlab.system.mixin.Nondirect`.

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.

- 3** 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` to reset the property states.
 - Use the `validatePropertiesImpl` to check that the property values are valid.
- 4** 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` to specify that the block is not direct feedthrough. Its inputs do not directly affect its outputs.
- 5** 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
    end
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 `step` method on the object multiple times with different inputs. This tests for syntax errors and other possible issues before you add it to Simulink. For example,

```
desired = 0;
actual = 0.2;
step(s,desired,actual);
```

Add System Objects to Your Simulink Model

- 1 Add your System objects to your Simulink model by using the MATLAB System block as described in “Mapping System Objects to Block Dialog Box” on page 32-17.
- 2 Add other Simulink blocks, connect them, and configure any needed parameters to complete your model as described in the Simulink documentation. See the System Identification for an FIR System Using MATLAB System Blocks Simulink example.

3 Run your model in the same way you run any Simulink model.

.

Define New System Objects

- “Define Basic System Objects” on page 31-3
- “Change Number of Step Inputs or Outputs” on page 31-5
- “Specify System Block Input and Output Names” on page 31-8
- “Validate Property and Input Values” on page 31-10
- “Initialize Properties and Setup One-Time Calculations” on page 31-13
- “Set Property Values at Construction Time” on page 31-16
- “Reset Algorithm State” on page 31-19
- “Define Property Attributes” on page 31-21
- “Hide Inactive Properties” on page 31-25
- “Limit Property Values to Finite String Set” on page 31-27
- “Process Tuned Properties” on page 31-30
- “Release System Object Resources” on page 31-32
- “Define Composite System Objects” on page 31-34
- “Define Finite Source Objects” on page 31-38
- “Save System Object” on page 31-40
- “Load System Object” on page 31-43
- “Clone System Object” on page 31-46
- “Define System Block Icon” on page 31-47
- “Add Header to System Block Dialog” on page 31-49
- “Add Property Groups to System Object and Block Dialog” on page 31-51
- “Set Output Size” on page 31-55

- “Set Output Data Type” on page 31-58
- “Set Output Complexity” on page 31-61
- “Specify Whether Output Is Fixed- or Variable-Size” on page 31-64
- “Specify Discrete State Output Specification” on page 31-67
- “Use Update and Output for Nondirect Feedthrough” on page 31-70
- “Methods Timing” on page 31-73
- “System Object Input Arguments and ~ in Code Examples” on page 31-76
- “What Are Mixin Classes?” on page 31-77

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

- 1 Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

```
edit AddOne.m
```

- 2 Subclass your object from `matlab.System`. Insert this line as the first line of your file.

```
classdef AddOne < matlab.System
```

- 3 Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle.

In this example, instead of passing in the object handle, `~` is used to indicate that the object handle is not used in the function. Using `~` instead of an object handle prevents warnings about unused variables from occurring.

By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

```
methods (Access=protected)
    function y = stepImpl(~, x)
        y = x + 1;
    end
end
```

Note Instead of manually creating your class definition file, you can use **File > New > System Object** to open a sample System object file in the editor. You then can edit that file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic System Object

```
classdef AddOne < matlab.System
%ADDONE Compute an output value one greater than the input value

    % All methods occur inside a methods declaration.
    % The stepImpl method has protected access
    methods (Access=protected)

        function y = stepImpl(~,x)
            y = x + 1;
        end
    end
end
```

See Also

[stepImpl](#) | [getNumInputsImpl](#) | [getNumOutputsImpl](#) | [matlab.System](#) |

Related Examples

- “Change Number of Step Inputs or Outputs” on page 31-5

Change Number of Step Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you do not specify the `getNumInputsImpl` and `getNumOutputsImpl` methods, the object uses the default values of 1 input and 1 output. In this case, the user must provide an input to the `step` method.

Note You should only use `getNumInputsImpl` or `getNumOutputsImpl` methods to change the number of System object inputs or outputs. Do not use any other handle objects within a System object to change the number of inputs or outputs.

To specify no inputs, you must explicitly set the number of inputs to 0 using the `getNumInputsImpl` method. To specify no outputs, you must explicitly return 0 in the `getNumOutputsImpl` method.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to accept a second input and provide a second output.

```
methods (Access=protected)
    function [y1,y2] = stepImpl(~,x1,x2)
        y1 = x1 + 1;
        y2 = x2 + 1;
    end
end
```

Update the Associated Methods

Use `getNumInputsImpl` and `getNumOutputsImpl` to specify two inputs and two outputs, respectively.

```
methods (Access=protected)
    function numIn = getNumInputsImpl(~)
        numIn = 2;
    end

    function numOut = getNumOutputsImpl(~)
        numOut = 2;
    end
end
```

Complete Class Definition File with Multiple Inputs and Outputs

```
classdef AddOne < matlab.System
%ADDONE Compute output values two greater than the input values

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods(Access=protected)

    function [y1,y2] = stepImpl(~,x1,x2)
        y1 = x1 + 1;
        y2 = x2 + 1;
    end

    % getNumInputsImpl method calculates number of inputs
    function num = getNumInputsImpl(~)
        num = 2;
    end

    % getNumOutputsImpl method calculates number of outputs
    function num = getNumOutputsImpl(~)
        num = 2;
    end
end
end
```

See Also

[getNumInputsImpl](#) | [getNumOutputsImpl](#) |

Related Examples

- “Validate Property and Input Values” on page 31-10
- “Define Basic System Objects” on page 31-3

Concepts

- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Specify System Block Input and Output Names

This example shows how to specify the names of the input and output ports of a System object–based block implemented using a MATLAB System block.

Define Input and Output Names

This example shows how to 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 strings.

```
methods (Access=protected)
    function inputName = getInputNamesImpl(~)
        inputName = 'source data';
    end

    function outputName = getOutputNamesImpl(~)
        outputName = 'count';
    end
end
```

Complete Class Definition File with Named Inputs and Outputs

```
classdef MyCounter < matlab.System

    %MyCounter Count values above a threshold

    properties
        Threshold = 1
    end
    properties (DiscreteState)
        Count
    end
end
```

```

methods
    function obj = MyCounter(varargin)
        setProperties(obj,nargin,varargin{:});
    end
end

methods (Access=protected)
    function setupImpl(obj, u)
        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

```

See Also

getNumInputsImpl | getNumOutputsImpl | getInputNamesImpl |
getOutputNamesImpl |

**Related
Examples**

- “Change Number of Step Inputs or Outputs” on page 31-5

Concepts

- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using `set.PropertyName` syntax. In this case, the *PropertyName* is `Increment`.

```
methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
            error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
end
```

This example shows how to validate the value of two interdependent properties using the `validatePropertiesImpl` method. In this case, the `UseIncrement` property value must be true and the `WrapValue` property value must be less than the `Increment` property value.

```
methods (Access=protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue < obj.Increment
            error('Wrap value must be less than increment value');
        end
    end
end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access=protected)
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
```



```
        error('Input must be numeric');
    end
end
end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
%ADDONE Compute an output value by incrementing the input value

% All properties occur inside a properties declaration.
% These properties have public access (the default)
properties (Logical)
    UseIncrement = true
end

properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
end

methods
    % Validate the properties of the object
    function set.Increment(obj, val)
        if val >= 10
            error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
end

methods (Access=protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue < obj.Increment
            error('Wrap value must be less than increment value');
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
```

```
        if ~isnumeric(x)
            error('Input must be numeric');
        end
    end

    function out = stepImpl(obj,in)
        if obj.UseIncrement
            out = in + obj.Increment;
        else
            out = in + 1;
        end
    end
end
end
```

Note All inputs default to variable-size inputs. See “Change Input Complexity or Dimensions” for more information.

See Also

[validateInputsImpl](#) | [validatePropertiesImpl](#) |

Related Examples

- “Define Basic System Objects” on page 31-3

Concepts

- “Methods Timing” on page 31-73
- “Property Set Methods”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you call the step method.

Define Public Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable string, `default.bin`. Users cannot change *nontunable* properties after the `setup` method has been called. Refer to the Methods Timing section for more information.

```
properties (Nontunable)
    Filename = 'default.bin'
end
```

Define Private Properties to Initialize

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as *hidden* to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access=private)
    pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once during the first call to the `step` method. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
    function setupImpl(obj,data)
        obj.pFileID = fopen(obj.Filename,'wb');
```

```
        if obj.pFileID < 0
            error('Opening the file failed');
        end
    end
end
```

Although not part of setup, you should close files when your code is done using them. You use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```
classdef MyFile < matlab.System
%MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access=private)
        pFileID; % The identifier of the file to open
    end

    methods (Access=protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj,data)
            obj.pFileID = fopen(obj.Filename,'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end
    end

    % This System object writes the input to the file.
    function stepImpl(obj,data)
        fwrite(obj.pFileID,data);
    end
end
```

```
% Use release to close the file to prevent the
% file handle from being left open.
function releaseImpl(obj)
    fclose(obj.pFileID);
end

% You indicate that no outputs are provided by returning
% zero from getNumOutputsImpl
function numOutputs = getNumOutputsImpl(-)
    numOutputs = 0;
end
end
end
```

See Also

setupImpl | releaseImpl | stepImpl |

Related Examples

- “Release System Object Resources” on page 31-32
- “Define Property Attributes” on page 31-21

Concepts

- “Methods Timing” on page 31-73

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (MyFile in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
    function obj = MyFile(varargin)
        setProperties(obj,nargin,varargin{:});
    end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
%MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
        Access = 'wb' % The file access string (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access=private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
```

```

        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj, ~)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end
    end

    % This System object writes the input to the file.
    function stepImpl(obj, data)
        fwrite(obj.pFileID,data);
    end

    % Use release to close the file to prevent the
    % file handle from being left open.
    function releaseImpl(obj)
        fclose(obj.pFileID);
    end

    % You indicate that no outputs are provided by returning
    % zero from getNumOutputsImpl
    function numOutputs = getNumOutputsImpl(~)
        numOutputs = 0;
    end
end
end

```

See Also

narginsetProperties |

Related Examples

- “Define Property Attributes” on page 31-21
- “Release System Object Resources” on page 31-32

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

pCount is an internal counter property of the System object obj. The user calls the reset method, which calls the resetImpl method. In this example, pCount resets to 0.

Note When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access=protected)
    function resetImpl(obj)
        obj.pCount = 0;
    end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
%Counter System object that increments a counter

    properties(Access=private)
        pCount
    end

    methods (Access=protected)
        % In step, increment the counter and return
        % its value as an output
        function c = stepImpl(obj)
            obj.pCount = obj.pCount + 1;
            c = obj.pCount;
        end

        % Reset the counter to zero.
        function resetImpl(obj)
```

```
        obj.pCount = 0;
    end

    % The step method takes no inputs
    function numIn = getNumInputsImpl(~)
        numIn = 0;
    end
end
end
end
```

See “Methods Timing” on page 31-73 for more information.

See Also `resetImpl` |

Concepts

- “Methods Timing” on page 31-73

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

System object users cannot change nontunable properties after the `setup` or `step` method has been called. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
    InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter 1 or 0 or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use 1 or 0 as a default value.

```
properties (Logical)
    Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `pCount` is constrained to accept only real, positive integers. You cannot use sparse values. The following restriction applies to a property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
    Count
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute. Properties with this attribute display their state values when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or `fi` value, but not a scaled double `fi` value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, `Constant`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess=Public` by default
- Value set only using the `setupImpl` method or when the System object is locked during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
    Count;
```

```
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
%Counter Increment a counter starting at an initial value

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        % The initial value of the counter
        InitialValue = 0
    end

    properties (Logical)
        % Whether to increment the counter
        Increment = true
    end

    % Count state variable
    properties (DiscreteState, PositiveInteger)
        Count
    end

    methods (Access=protected)
        % In step, increment the counter and return its value
        % as an output
        function c = stepImpl(obj)
            if obj.Increment
                obj.Count = obj.Count + 1;
            end
            c = obj.Count;
        end
        % Setup the Count state variable
        function setupImpl(obj)
            obj.Count = 0;
        end
        % Reset the counter to zero.
        function resetImpl(obj)
            obj.Count = obj.InitialValue;
        end
    end
end
```

```
        end
        % The step method takes no inputs
        function numIn = getNumInputsImpl(~)
            numIn = 0;
        end
    end
end
```

Concepts

- “Class Attributes”
- “Property Attributes”
- “Methods Timing” on page 31-73

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the `isInactivePropertyImpl` method to hide a property from displaying. If the `isInactiveProperty` method returns true to the property you pass in, then that property does not display.

```
methods (Access=protected)
    function flag = isInactivePropertyImpl(obj,propertyName)
        if strcmp(propertyName,'InitialValue')
            flag = obj.UseRandomInitialValue;
        else
            flag = false;
        end
    end
end
```

Complete Class Definition File with Hidden Inactive Property

```
classdef Counter < matlab.System
    %Counter Increment a counter

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        % Allow the user to set the initial value
        UseRandomInitialValue = true
        InitialValue = 0
    end

    % The private count variable, which is tunable by default
    properties (Access=private)
        pCount
    end

    methods (Access=protected)
```

```
% In step, increment the counter and return its value
% as an output
function c = stepImpl(obj)
    obj.pCount = obj.pCount + 1;
    c = obj.pCount;
end

%Reset the counter to either a random value or the initial
% value.
function resetImpl(obj)
    if obj.UseRandomInitialValue
        obj.pCount = rand();
    else
        obj.pCount = obj.InitialValue;
    end
end

% The step method takes no inputs
function numIn = getNumInputsImpl(~)
    numIn = 0;
end

% This method controls visibility of the object's properties
function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
    else
        flag = false;
    end
end
end
end
```

See Also [isInactivePropertyImpl](#) |

Limit Property Values to Finite String Set

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

String sets use two related properties. You first specify the user-visible property name and default string value. Then, you specify the associated hidden property by appending “Set” to the property name. You must use a capital “S” in “Set.”

In the “Set” property, you specify the valid string values as a cell array of the `matlab.system.Stringset` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
    Color = 'blue'
end
```

```
properties (Hidden,Transient)
    ColorSet = matlab.system.StringSet({'red','blue','green'});
end
```

Complete Class Definition File with String Set

```
classdef Whiteboard < matlab.System
    %Whiteboard Draw lines on a figure window
    %
    % This System object illustrates the use of StringSets

    properties
        Color = 'blue'
    end

    properties (Hidden,Transient)
        % Let them choose a color
        ColorSet = matlab.system.StringSet({'red','blue','green'});
    end
```

```
methods(Access = protected)
function stepImpl(obj)
    h = Whiteboard.getWhiteboard();
    plot(h, ...
        randn([2,1]),randn([2,1]), ...
        'Color',obj.Color(1));
end
function releaseImpl(obj)
    cla(Whiteboard.getWhiteboard());
    hold('on');
end
function n = getNumInputsImpl(~)
    n = 0;
end
function n = getNumOutputsImpl(~)
    n = 0;
end
end

methods (Static)
function a = getWhiteboard()
    h = findobj('tag','whiteboard');
    if isempty(h)
        h = figure('tag','whiteboard');
        hold('on');
    end
    a = gca;
end
end
end
```

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk = Whiteboard;
```

```
% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color  = 'blue';

% Take a few steps
for i=1:3
    hGreenInk.step();
    hBlueInk.step();
end

%% Clear the whiteboard
hBlueInk.release();

%% Display System object used in this example
type('Whiteboard.m');
```

See Also `matlab.system.StringSet` |

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        obj.pLookupTable = obj.MiddleC * ...
            (1+log(1:obj.NumNotes)/log(12));
    end
end
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
    %TuningFork Illustrate the processing of tuned parameters
    %

    properties
        MiddleC = 440
        NumNotes = 12
    end

    properties (Access=private)
        pLookupTable
    end

    methods(Access=protected)
        function resetImpl(obj)
            obj.MiddleC = 440;
        end
    end
end
```

```
        obj.pLookupTable = obj.MiddleC * ...
            (1+log(1:obj.NumNotes)/log(12));
    end

    function hz = stepImpl(obj,noteShift)
        % A noteShift value of 1 corresponds to obj.MiddleC
        hz = obj.pLookupTable(noteShift);
    end

    function processTunedPropertiesImpl(obj)
        % Generate a lookup table of note frequencies
        obj.pLookupTable = obj.MiddleC * ...
            (1+log(1:obj.NumNotes)/log(12));
    end
end
end
end
```

See Also `processTunedPropertiesImpl` |

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
    function releaseImpl(obj)
        cla(Whiteboard.getWhiteboard());
        hold('on');
    end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
%Whiteboard Draw lines on a figure window
%
% This System object shows the use of StringSets
%
    properties
        Color = 'blue'
    end

    properties (Hidden)
        % Let user choose a color
        ColorSet = matlab.system.StringSet({'red','blue','green'});
    end

    methods(Access=protected)
        function stepImpl(obj)
            h = Whiteboard.getWhiteboard();
            plot(h, ...
                randn([2,1]), randn([2,1]), ...
                'Color',obj.Color(1));
        end
    end
end
```

```
end

function releaseImpl(obj)
    cla(Whiteboard.getWhiteboard());
    hold('on');
end

function n = getNumInputsImpl(~)
    n = 0;
end
function n = getNumOutputsImpl(~)
    n = 0;
end
end

methods (Static)
function a = getWhiteboard()
    h = findobj('tag','whiteboard');
    if isempty(h)
        h = figure('tag','whiteboard');
        hold('on');
    end
    a = gca;
end
end
end
```

See Also `releaseImpl` |

Related Examples

- “Initialize Properties and Setup One-Time Calculations” on page 31-13

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the pFir and pIir property values.

```
properties (Nontunable, Access = private)
    pFir % store the FIR filter
    pIir % store the IIR filter
end

methods
    function obj = Filter(varargin)
        setProperties(obj, nargin, varargin{:});
        obj.pFir = FIR(obj.zero);
        obj.pIir = IIR(obj.pole);
    end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
% Filter System object with a single pole and a single zero
%
% This System object illustrates composition by
% composing an instance of itself.
%

    properties (Nontunable)
        zero = 0.01
        pole = 0.5
    end
end
```



```
end

properties (Nontunable,Access=private)
    pZero % store the FIR filter
    pPole % store the IIR filter
end

methods
    function obj = Filter(varargin)
        setProperties(obj,nargin, varargin{:});
        % Create instances of FIR and IIR as
        % private properties
        obj.pZero = Zero(obj.zero);
        obj.pPole = Pole(obj.pole);
    end
end

methods (Access=protected)
    function setupImpl(obj,x)
        setup(obj.pZero,x);
        setup(obj.pPole,x);
    end

    function resetImpl(obj)
        reset(obj.pZero);
        reset(obj.pPole);
    end

    function y = stepImpl(obj,x)
        y = step(obj.pZero,x) + step(obj.pPole,x);
    end

    function releaseImpl(obj)
        release(obj.pZero);
        release(obj.pPole);
    end
end
end
```

Class Definition File for IIR Component of Filter

```
classdef Pole < matlab.System

    properties
        Den = 1
    end

    properties (Access=private)
        tap = 0
    end

    methods
        function obj = Pole(varargin)
            setProperties(obj,nargin,varargin{:},'Den');
        end
    end

    methods (Access=protected)
        function y = stepImpl(obj,x)
            y = x + obj.tap * obj.Den;
            obj.tap = y;
        end
    end

end
```

Class Definition File for FIR Component of Filter

```
classdef Zero < matlab.System

    properties
        Num = 1
    end

    properties (Access=private)
        tap = 0
    end

    methods
        function obj = Zero(varargin)
            setProperties(obj, nargin,varargin{:},'Num');
        end
    end

end
```

```
        end
    end

    methods (Access=protected)
        function y = stepImpl(obj,x)
            y = x + obj.tap * obj.Num;
            obj.tap = x;
        end
    end

end
```

See Also `nargin`

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

- 1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...  
    matlab.system.mixin.FiniteSource
```

- 2 Specify the end of the source with the isDoneImpl method. In this example, the source has two iterations.

```
methods (Access = protected)  
    function bDone = isDoneImpl(obj)  
        bDone = obj.NumSteps==2  
    end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...  
    matlab.system.mixin.FiniteSource  
    %RunTwice System object that runs exactly two times  
    %  
    properties (Access=private)  
        NumSteps  
    end  
  
    methods (Access=protected)  
        function resetImpl(obj)  
            obj.NumSteps = 0;  
        end  
  
        function y = stepImpl(obj)  
            if ~obj.isDone()  
                obj.NumSteps = obj.NumSteps + 1;  
                y = obj.NumSteps;  
            else
```

```
        y = 0;
    end
end

function bDone = isDoneImpl(obj)
    bDone = obj.NumSteps==2;
end
end

methods (Access=protected)
    function n = getNumInputsImpl(~)
        n = 0;
    end
    function n = getNumOutputsImpl(~)
        n = 1;
    end
end
end

end
```

See Also `matlab.system.mixin.FiniteSource` |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods(Access=protected)
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.child = matlab.System.saveObject(obj.child);
        s.protected = obj.protected;
        s.pdependentprop = obj.pdependentprop;
        if isLocked(obj)
            s.state = obj.state;
        end
    end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

    properties (Access=private)
        child
        pdependentprop
    end

    properties (Access=protected)
        protected = rand;
    end

    properties (DiscreteState=true)
        state
    end
end
```

```
end

properties (Dependent)
    dependentprop
end

methods
    function obj = MySaveLoader(varargin)
        obj@matlab.System();
        setProperties(obj, nargin, varargin{:});
    end
end

methods(Access = protected)
    function setupImpl(obj, varargin)
        obj.state = 42;
    end

    function out = stepImpl(obj, in)
        obj.state = in;
        out = obj.state;
    end
end

% Serialization
methods(Access=protected)
    function s = saveObjectImpl(obj)
        % Call the base class method
        s = saveObjectImpl@matlab.System(obj);

        % Save the child System objects
        s.child = matlab.System.saveObject(obj.child);

        % Save the protected & private properties
        s.protected = obj.protected;
        s.pdependentprop = obj.pdependentprop;

        % Save the state only if object locked
        if isLocked(obj)
```

```
        s.state = obj.state;
    end
end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protected = s.protected;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
end
end
```

See Also

[saveObjectImpl](#) | [loadObjectImpl](#) |

Related Examples

- “Load System Object” on page 31-43

Load System Object

This example shows how to load a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to assign the child object struct data to the associated object property. Assign protected and dependent property data to the associated object properties. If the object was locked when it was saved, assign the object's state to the associated property. Load the saved public properties with the `loadObjectImpl` method.

```
methods(Access=protected)
    function loadObjectImpl(obj,s,wasLocked)
        obj.child = matlab.System.loadObject(s.child);
        obj.protected = s.protected;
        obj.pdependentprop = s.pdependentprop;
        if wasLocked
            obj.state = s.state;
        end
        loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

    properties (Access=private)
        child
        pdependentprop
    end

    properties (Access=protected)
        protected = rand;
    end

    properties (DiscreteState=true)
```

```
        state
    end

    properties (Dependent)
        dependentprop
    end

    methods
        function obj = MySaveLoader(varargin)
            obj@matlab.System();
            setProperties(obj, nargin, varargin{:});
        end
    end

    methods(Access = protected)
        function setupImpl(obj,varargin)
            obj.state = 42;
        end

        function out = stepImpl(obj,in)
            obj.state = in;
            out = obj.state;
        end
    end

    % Serialization
    methods(Access=protected)
        function s = saveObjectImpl(obj)
            % Call the base class method
            s = saveObjectImpl@matlab.System(obj);

            % Save the child System objects
            s.child = matlab.System.saveObject(obj.child);

            % Save the protected & private properties
            s.protected = obj.protected;
            s.pdependentprop = obj.pdependentprop;

            % Save the state only if object locked
```

```
        if isLocked(obj)
            s.state = obj.state;
        end
    end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protected = s.protected;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
end
end
```

See Also

[saveObjectImpl](#) | [loadObjectImpl](#) |

Related Examples

- “Save System Object” on page 31-40

Clone System Object

This example shows how to clone a System object.

Clone System Object

You can define your own clone method, which is useful for copying objects without saving their state. The default `cloneImpl` method copies both a System object™ and its current state. If an object is locked, the default `cloneImpl` creates a cloned object that is also locked. An example of when you may want to write your own clone method is for cloning objects that handle resources. These objects cannot allocate resources twice and you would not want to save their states. To write your clone method, use the `saveObject` and `loadObject` methods to perform the clone within the `cloneImpl` method.

```
methods(Access=protected)
    function obj2 = cloneImpl(obj1)
        s = saveObject (obj1);
        obj2 = loadObject(s);
    end
end
```

Complete Class Definition File with Clone

```
classdef PassThrough < matlab.System
    methods (Access=protected)
        function y = stepImpl(~,u)
            y = u;
        end
        function obj2 = cloneImpl(obj1)
            s = matlab.System.saveObject(obj1);
            obj2 = matlab.System.loadObject(s);
        end
    end
end
```

See Also

`cloneImpl` | `saveObjectImpl` | `loadObjectImpl` |

Define System Block Icon

This example shows how to define the block icon of a System object–based block implemented using a MATLAB System block.

Use the CustomIcon Class and Define the Icon

- 1 Subclass from custom icon class.

```
classdef MyCounter < matlab.System & ...
    matlab.system.mixin.CustomIcon
```

- 2 Use `setIconImpl` to specify the block icon as `New Counter` with a line break (`\n`) between the two words.

```
methods (Access=protected)
    function icon = getIconImpl(~)
        icon = sprintf('New\nCounter');
    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
end
```

```
methods (Access=protected)
    function setupImpl(obj, u)
        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 = sprintf('New\nCounter');
    end
end
end
```

See Also

[getIconImpl](#) | [matlab.system.mixin.CustomIcon](#) |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Add Header to System Block Dialog

This example shows how to add a header panel to a System object–based block implemented using a MATLAB System block.

Define Header Title and Text

This example shows how to 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.

You always set the `getHeaderImpl` method access to `protected` because it is an internal method that end users do not directly call or run.

```
methods(Static,Access=protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header('MyCounter',...
            'Title','My Enhanced Counter');
    end
end
```

Complete Class Definition File with Defined Header

```
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
```

```
        function N = getNumInputsImpl(obj)
```

```
            N = 1;
```

```
        end
```

```
    function N = getNumOutputsImpl(obj)
```

```
        N = 1;
```

```
    end
```

```
end
```

```
end
```

See Also

[getHeaderImpl](#) | [matlab.system.display.Header](#) |

Add Property Groups to System Object and Block Dialog

This example shows how to define property sections and section groups for System object display. The sections and section groups display as panels and tabs, respectively, in the MATLAB System block dialog.

Define Section of Properties

This example shows how to use `matlab.system.display.Section` and `getPropertyGroupsImpl` to define two property group sections by specifying their titles and property lists.

If you do not specify a property in `getPropertyGroupsImpl`, the block does not display that property.

```
methods(Static,Access=protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});
        groups = [valueGroup,thresholdGroup];
    end
end
```

Define Group of Sections

This example shows how to use `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` to define two tabs, each containing specific properties.

```
methods(Static,Access=protected)
    function groups = getPropertyGroupsImpl
        upperGroup = matlab.system.display.Section(...
            'Title','Upper threshold',...
            'PropertyList',{'UpperThreshold'});
        lowerGroup = matlab.system.display.Section(...
```

```
        'Title','Lower threshold', ...
        'PropertyList',{'UseLowerThreshold','LowerThreshold'});

    thresholdGroup = matlab.system.display.SectionGroup(...
        'Title', 'Parameters', ...
        'Sections', [upperGroup,lowerGroup]);

    valuesGroup = matlab.system.display.SectionGroup(...
        'Title', 'Initial conditions', ...
        'PropertyList', {'StartValue'});

    groups = [thresholdGroup, valuesGroup];
end
end
```

Complete Class Definition File with Property Group and Separate Tab

```
classdef EnhancedCounter < matlab.System
    % EnhancedCounter Count values considering thresholds

    properties
        UpperThreshold = 1;
        LowerThreshold = 0;
    end

    properties(Nontunable)
        StartValue = 0;
    end

    properties(Logical,Nontunable)
        % Count values less than lower threshold
        UseLowerThreshold = true;
    end

    properties (DiscreteState)
        Count;
    end

    methods(Static,Access=protected)
        function groups = getPropertyGroupsImpl
```

```

upperGroup = matlab.system.display.Section(...
    'Title', 'Upper threshold', ...
    'PropertyList', {'UpperThreshold'});
lowerGroup = matlab.system.display.Section(...
    'Title', 'Lower threshold', ...
    'PropertyList', {'UseLowerThreshold', 'LowerThreshold'});

thresholdGroup = matlab.system.display.SectionGroup(...
    'Title', 'Parameters', ...
    'Sections', [upperGroup, lowerGroup]);

valuesGroup = matlab.system.display.SectionGroup(...
    'Title', 'Initial conditions', ...
    'PropertyList', {'StartValue'});

    groups = [thresholdGroup, valuesGroup];
end
end

methods(Access=protected)
function setupImpl(obj, ~, ~)
    obj.Count = obj.StartValue;
end

function y = stepImpl(obj, u)
    if obj.UseLowerThreshold
        if (u > obj.UpperThreshold) || ...
            (u < obj.LowerThreshold)
            obj.Count = obj.Count + 1;
        end
    else
        if (u > obj.UpperThreshold)
            obj.Count = obj.Count + 1;
        end
    end
    y = obj.Count;
end

function resetImpl(obj)
    obj.Count = obj.StartValue;
end
end

```

```
function flag = isInactivePropertyImpl(obj, prop)
    flag = false;
    switch prop
        case 'LowerThreshold'
            flag = ~obj.UseLowerThreshold;
        end
    end
end
end
end
```

See Also

`getPropertyGroupsImpl` | `matlab.system.display.Section` |
`matlab.system.display.SectionGroup` |

Concepts

- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Set Output Size

This example shows how to specify the size of a System object output using the `getOutputSizeImpl` method. This method indicates the output size when the size cannot be inferred from the inputs during Simulink model compilation.

Subclass from the Propagates Mixin Class

To use the `getOutputSizeImpl` method, you must subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Specify Output Size

Use the `getOutputSizeImpl` method to specify the output size.

```
methods (Access=protected)
    function sizeout = getOutputSizeImpl(~)
        sizeout = [1 1];
    end
end
```

Complete Class Definition File with Specified Output Size

```
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
    end
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 (u1 > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function resetImpl(obj)
    obj.Count = 0;
end

function N = getNumInputsImpl(~)
    N = 2;
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
```

```
        end
    end
end
```

See Also

`matlab.system.mixin.Propagates` | `getOutputSizeImpl` |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Set Output Data Type

This example shows how to specify the data type of a System object output using the `getOutputDataTypeImpl` method. This method indicates the data type of the output when the data type cannot be inferred from the inputs during Simulink model compilation.

Subclass from the Propagates Mixin Class

To use the `getOutputDataTypeImpl` method, you must subclass from the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...  
    matlab.system.mixin.Propagates
```

Specify Output Data Type

Use the `getOutputDataTypeImpl` method to specify the output data type as a double.

```
methods (Access=protected)  
    function dataout = getOutputDataTypeImpl(~)  
        dataout = 'double';  
    end  
end
```

Complete Class Definition File with Specified Output Data Type

```
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 resetImpl(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 (u1 > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function N = getNumInputsImpl(~)
    N = 2;
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
    end
end
```

See Also

`matlab.system.mixin.Propagates` | `getOutputDataTypeImpl` |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Set Output Complexity

This example shows how to specify whether a System object output is a complex or real value. You use the `isOutputComplexImpl` method when the output complexity cannot be inferred from the inputs during Simulink model compilation.

Subclass from the Propagates Mixin Class

To use the `isOutputComplexImpl` method, you must subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Specify Output Complexity

Use the `isOutputComplexImpl` method to specify that the output is real.

```
methods (Access=protected)
    function cplxout = isOutputComplexImpl(~)
        cplxout = false;
    end
end
```

Complete Class Definition File with Specified Complexity

```
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
    end
end
```

```
end

function resetImpl(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 (u1 > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function N = getNumInputsImpl(~)
    N = 2;
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
end
end
```

See Also

`matlab.system.mixin.Propagates` | `isOutputComplexImpl` |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Specify Whether Output Is Fixed- or Variable-Size

This example shows how to specify whether a System object output has fixed- or variable-size output. . You use the `isOutputFixedSizeImpl` method when the output type cannot be inferred from the inputs during Simulink model compilation.

Subclass from the Propagates Mixin Class

To use the `isOutputFixedSizeImpl` method, you must subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...  
    matlab.system.mixin.Propagates
```

Specify Output as Fixed Size

Use the `isOutputFixedSizeImpl` method to specify that the output is fixed size.

```
methods (Access=protected)  
    function fixedout = isOutputFixedSizeImpl(~)  
        fixedout = true;  
    end  
end
```

Complete Class Definition File with Specified Output Data Type

```
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 resetImpl(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 (u1 > obj.Threshold)
        obj.Count = obj.Count + 1;
    end
    y = obj.Count;
end

function N = getNumInputsImpl(~)
    N = 2;
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
    end
end
```

See Also

`matlab.system.mixin.Propagates` | `isOutputFixedSizeImpl` |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Specify Discrete State Output Specification

This example shows how to specify the size, data type, and complexity of a discrete state property. You must use the `getDiscreteStateSpecificationImpl` method when your System object has a property that is defined with the `DiscreteState` attribute. This method indicates the output specifications when those specifications cannot be inferred during Simulink model compilation.

Subclass from the Propagates Mixin Class

To use the `getDiscreteStateSpecificationImpl` method, you must subclass from both the `matlab.System` base class and from the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Specify Discrete State Output Specification

Use the `getDiscreteStateSpecificationImpl` method to specify the size and data type. Also specify the complexity of a discrete state property, which is used in the counter reset example.

```
methods (Access=protected)
    function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
        sz = [1 1];
        dt = 'double';
        cp = false;
    end
end
```

Complete Class Definition File with Discrete State Output Specification

```
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 resetImpl(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 (u1 > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function N = getNumInputsImpl(~)
            N = 2;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            sz = [1 1];
            dt = 'double';
            cp = false;
        end

        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end

        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end

        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
    end
end
```

```
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
    end
end
```

See Also

`matlab.system.mixin.Propagates | getDiscreteStateSpecificationImpl`
|

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Use Update and Output for Nondirect Feedthrough

This example shows how to implement nondirect feedthrough for a System object using the `updateImpl`, `outputImpl` and `isInputDirectFeedthroughImpl` methods. In nondirect feedthrough, the object's 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. This enables you to use that 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 &...
    matlab.system.mixin.CustomIcon
    %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 positive non-zero scalar value.
            end
            if (numel(obj.InitialOutput)>1)
                error('Initial Output must be scalar value.');
            end
        end

        function setupImpl(obj, ~)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function resetImpl(obj)
```

```
        obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
    end

    function [y] = outputImpl(obj, ~)
        y = obj.PreviousInput(end);
    end
    function updateImpl(obj, u)
        obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
    end
    function flag = isInputDirectFeedthroughImpl(~,~)
        flag = false;
    end
end
end
```

See Also

[matlab.system.mixin.Nondirect](#) | [outputImpl](#) | [updateImpl](#) | [isInputDirectFeedthroughImpl](#) |

Concepts

- “What Are Mixin Classes?” on page 31-77
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 31-76

Methods Timing

In this section...

“Setup Method Call Sequence” on page 31-73

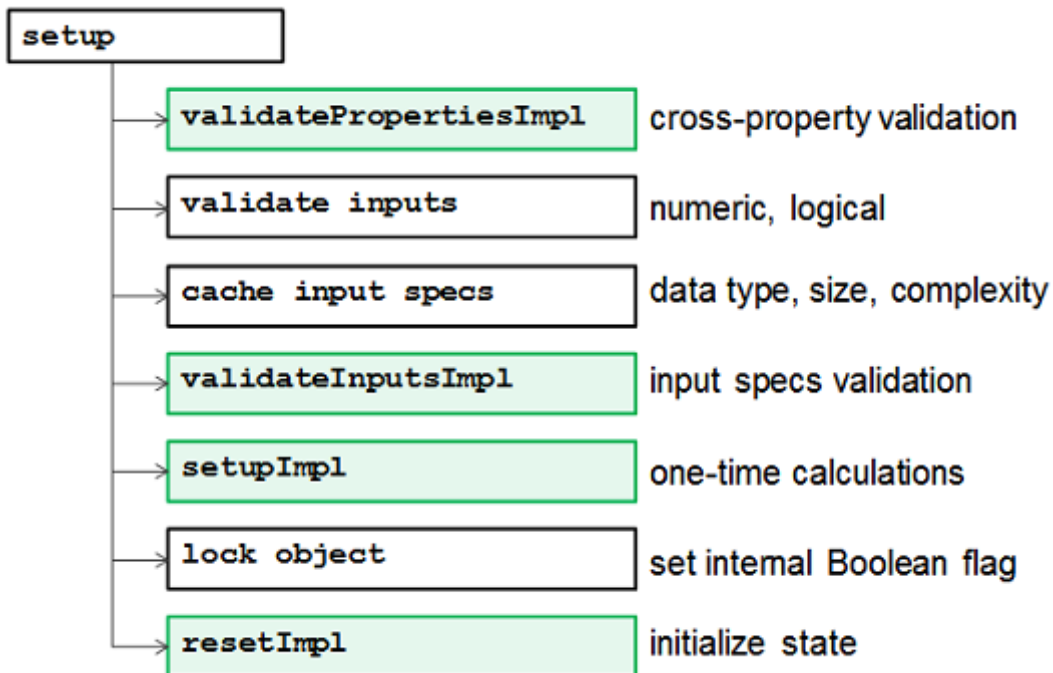
“Step Method Call Sequence” on page 31-74

“Reset Method Call Sequence” on page 31-74

“Release Method Call Sequence” on page 31-75

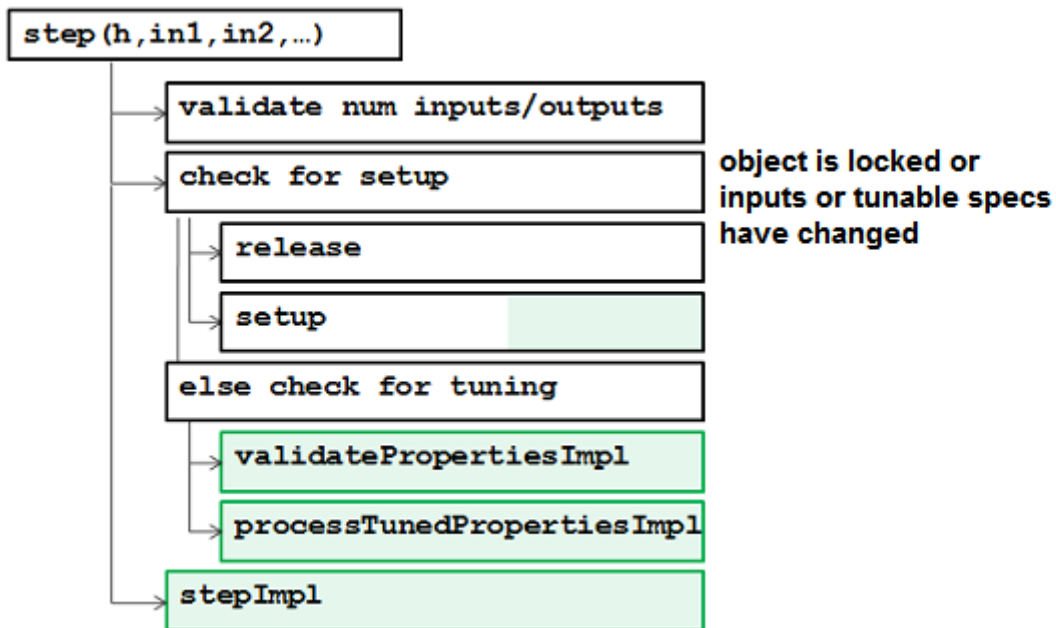
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.



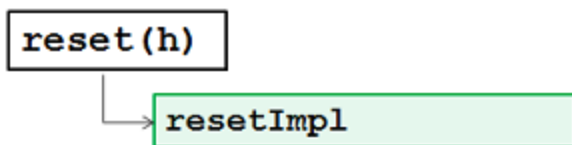
Step Method Call Sequence

This hierarchy shows the actions performed when you call the step method.



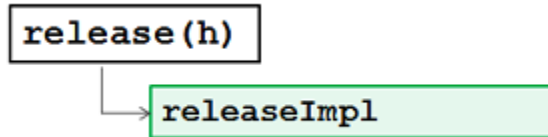
Reset Method Call Sequence

This hierarchy shows the actions performed when you call the reset method.



Release Method Call Sequence

This hierarchy shows the actions performed when you call the release method.



See Also

setupImpl | stepImpl | releaseImpl | resetImpl |

Related Examples

- “Release System Object Resources” on page 31-32
- “Reset Algorithm State” on page 31-19
- “Set Property Values at Construction Time” on page 31-16
- “Define Basic System Objects” on page 31-3

Concepts

- “What Are System Object Methods?” on page 30-12
- “The Step Method” on page 30-12
- “Common Methods” on page 30-13

System Object Input Arguments and ~ in Code Examples

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In many examples, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables.

What Are Mixin Classes?

Mixin classes are partial classes that you can combine in various combinations to form desired behaviors using multiple inheritance. System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

The following mixin classes are available for use with System objects.

- `matlab.system.mixin.CustomIcon` — Defines a block icon for System objects in the MATLAB System block
- `matlab.system.mixin.FiniteSource` — Adds the `isDone` method to System objects that are sources
- `matlab.system.mixin.Nondirect` — Allows the System object, when used in the MATLAB System block, to support nondirect feedthrough by making the runtime callback functions, `output` and `update` available
- `matlab.system.mixin.Propagates` — Enables System objects to operate in the MATLAB System block using the interpreted execution

System Objects in Simulink

- “What Is the MATLAB System Block?” on page 32-2
- “Implement a MATLAB System Block” on page 32-7
- “Change Blocks Implemented with System Objects” on page 32-10
- “Change Block Icon and Port Labels” on page 32-11
- “Use System Objects in Feedback Loops” on page 32-13
- “Simulation Modes” on page 32-15
- “Mapping System Objects to Block Dialog Box” on page 32-17
- “Considerations for Using System Objects in Simulink” on page 32-22
- “Simulink Engine Interaction with System Object Methods” on page 32-24
- “Add and Implement Propagation Methods” on page 32-27
- “Troubleshoot System Objects in Simulink” on page 32-30

What Is the MATLAB System Block?

In this section...

“Why Use the MATLAB System Block?” on page 32-2

“Choosing the Right Block Type” on page 32-2

“System Objects” on page 32-3

“Interpreted Execution or Code Generation” on page 32-3

“MATLAB System Block Limitations” on page 32-4

“MATLAB System and System Objects Examples” on page 32-5

Why Use the MATLAB System Block?

System objects let you implement algorithms using the MATLAB language. The MATLAB System block enables you to implement 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
- 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
- Fcn block

For more information, see “Types of Custom Blocks” on page 27-3 and “Comparison of Custom Block Functionality” on page 27-7.

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 “Define System Objects”.

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 (also known as the MATLAB interpreter).
- 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 Supported for C/C++ Code Generation — Alphabetical List” on page 34-2.

MATLAB System Block Limitations

These capabilities are currently not supported.

Category	Limitation Description	Workaround
System Objects	Tunable logical and string properties of the System object are nontunable parameters in the MATLAB System block.	—
Masking	You cannot create a mask on the MATLAB System block.	Consider one of the following: <ul style="list-style-type: none"> • Use System object authoring API to define parameters, icons, and port labels that masking gives you. • Put the MATLAB System in a Subsystem block and then mask the Subsystem block.
For Each Subsystem	For Each Subsystem blocks cannot contain a MATLAB System block.	—
Data Types	<ul style="list-style-type: none"> • System objects cannot use buses or the enumerated data type. • Fixed-point signals with nonbinary point scaling or nonzero bias. 	—
Sample Time	Cannot use MATLAB System blocks to model continuous time or multirate systems.	MATLAB System blocks can exist in continuous time or multirate models.
Linearizations	Jacobian based linearization.	—

Category	Limitation Description	Workaround
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.	—
Debugging	MATLAB debugging for code generation-based simulation.	Set the MATLAB System Simulate using parameter to Interpreted execution , 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 (Simulink Verification and Validation Software)	Simulink Verification and Validation model coverage report does not cover MATLAB files used by the MATLAB System block. The software ignores the setting of the model Analysis > Coverage > Settings > Coverage for MATLAB files option.	—
Check Model Compatibility (Simulink Design Verifier™ Software)	Simulink Design Verifier cannot perform compatibility check for MATLAB System block if it is contained in selected Subsystem or Model block.	—

MATLAB System and System Objects Examples

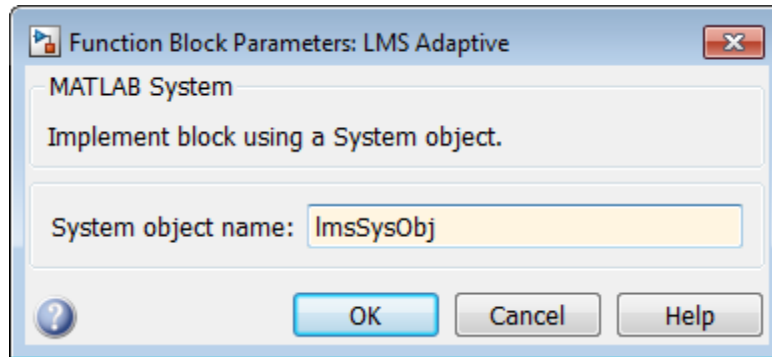
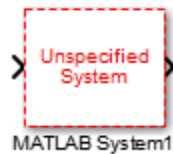
For examples of MATLAB System and System objects, see:

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 <code>Source code</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.

Implement a MATLAB System Block

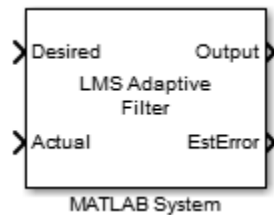
In the procedure, you implement a block and assign a System object to it. This example uses a System object named `lmsSysObj`. The System Identification for an FIR System Using MATLAB System Blocks example uses a similar System object. You can then explore the block to see the effect.

- 1 Create a new model and add the MATLAB System block from the User-Defined Functions library. Double-click the block and enter the full path name for a System object in the **System object name** parameter.



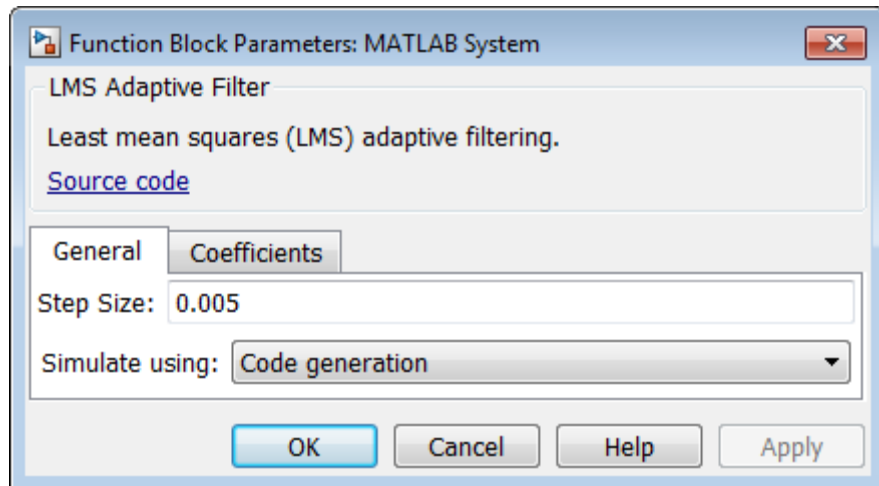
- 2 Click **OK**.

The MATLAB System block icon and port labels are updated to those of the corresponding System object.



Understanding the MATLAB System Block

- 1 Double-click the block. The MATLAB System dialog box has been updated to reflect the System object parameters. This dialog box has 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 with the `getHeaderImpl` method.
- 2 Click the `Source code` link and observe that the public and active properties in the System object appear in the MATLAB System block dialog box as block parameters.

- 3 Choose how you want the model to simulate the block in 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.) This example uses code generation.

Related Examples

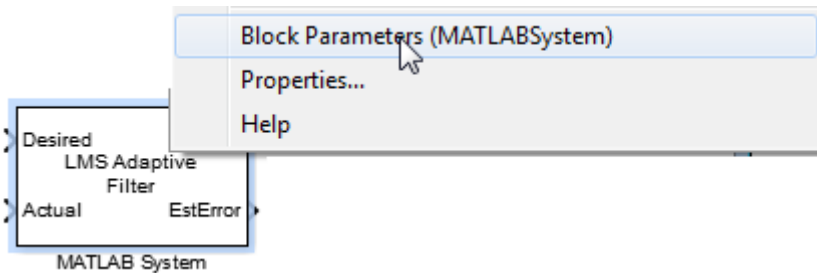
- “System Identification for an FIR System Using MATLAB System Blocks”

Concepts

- “Mapping System Objects to Block Dialog Box” on page 32-17
- “Simulation Modes” on page 32-15

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, enter a new class name in **System object name**.



Change Block Icon and Port Labels

To change the icon appearance of your block, you must use the `matlab.system.mixin.CustomIcon` class. You can define port labels using System object methods.

- 1 Add the `matlab.system.mixin.CustomIcon` class name to the System object, after the `matlab.System` class. For example:

```
classdef lmsSysObj < matlab.System & matlab.system.mixin.CustomIcon
```

This code subclasses from the `matlab.system.mixin.CustomIcon` class in addition to the `matlab.System` base class.

- 2 To define the icon, implement the `getIconImpl` method.
- 3 To define the port labels, implement the following optional methods to change the input and output port labels. You do not need the `matlab.system.mixin.CustomIcon` class to use these methods.

```
getInputNamesImpl  
getOutputNamesImpl
```

If you do not implement these methods, the System object uses the input and output port names from the `stepImpl` method. If you are using the `matlab.system.mixin.Nondirect` class and do not implement these methods, 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 the methods for the following classes:

Description	matlab.system.display Methods
Define header text for property group.	<code>matlab.system.display.Header</code>
Group properties together.	<code>matlab.system.display.Section</code>
Group properties into a separate tab.	<code>matlab.system.display.SectionGroup</code>

Related Examples

- “System Identification for an FIR System Using MATLAB System Blocks”

Concepts

- “System Block Icon and Dialog”

Use System Objects in Feedback Loops

If your algorithm needs to process nondirect feedthrough data through the System object, use the `matlab.system.mixin.Nondirect` class. This class uses the `output` and `update` methods to 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 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. A delay object is an example of a nondirect feedthrough object.

- 1 Add the `matlab.system.mixin.Nondirect` class to the top of the parent class file for the System object, after the `matlab.System` class. For example:

```
IntegerDelaySysObj < matlab.System & matlab.system.mixin.Nondirect
```

This step subclasses from the `matlab.system.mixin.Nondirect` class in addition to the `matlab.System` base class.

- 2 Implement the following methods:

```
outputImpl  
updateImpl
```

When implementing the `outputImpl` method, do not access the System object inputs for which the direct feedthrough flag is false.

- 3 To process the data as nondirect feedthrough, implement the `isInputDirectFeedthroughImpl` method.
- 4 Set the `isInputDirectFeedthroughImpl` flag to false to indicate the input port does not have direct feedthrough. Retain the default values for all other input and output port properties that are set.

The processing of the nondirect feedthrough changes the way that the software calls the System object methods within the context of the Simulink engine.

Related Examples

- “System Identification for an FIR System Using MATLAB System Blocks”

Concepts

- “Simulink Engine Interaction with System Object Methods” on page 32-24
- “System Block Nondirect Feedthrough”

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 with a valid System object class name enables 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. 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.
- 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 32-27.

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 block, where 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 do 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.

Mapping System Objects to Block Dialog Box

The System object source code controls the appearance of the block dialog box. This section describes System object to block dialog box mapping 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.

In this section...
“System Object to Block Dialog Box Default Mapping” on page 32-17
“System Object to Block Dialog Box Custom Mapping” on page 32-19

System Object to Block Dialog Box Default Mapping

The following figure shows how the source code corresponds to the dialog box elements if 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.)

```

classdef WidgetTypes < matlab.System
%WidgetTypes Show available dialog widget types
% This class contains properties to illustrate how System block dialog
% widgets are rendered for different System object property types.

properties(Nontunable, Logical)
%LogicalProperty A logical property
% This property has the logical attribute.
LogicalProperty = true;
end

|
properties(Nontunable)
StringSetProperty = 'default'; % A string set property
EditFieldProperty = 10; % An edit field property
end
properties(SetAccess = protected, Nontunable, Logical)
ReadOnlyLogicalProperty = true;
end
properties(SetAccess = protected, Nontunable)
ReadOnlyStringSetProperty = 'default';
ReadOnlyEditFieldProperty = 10;
end

properties(Constant, Hidden)
StringSetPropertySet = ...
    matlab.system.StringSet({'default', 'nondefault'});
end

methods
function obj = WidgetTypes(varargin)
    setProperties(obj, nargin, varargin{:});
end
end
methods(Access = protected)
function y = stepImpl(~, u)
    y = u;
end
end
end
end
    
```

Header description from class help summary

Header title from class name

Labels from property help summary

Property type and default value from property attributes

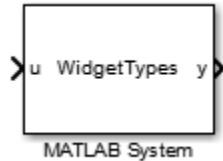
Simulate mode widget included in all dialogs

Link to open source MATLAB code

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 32-13.



For an example of a custom block dialog box, see “System Object to Block Dialog Box Custom Mapping” on page 32-19.

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` method from `matlab.system.display.Header` and `getPropertyGroupsImpl` method from `matlab.system.display.SectionGroup` 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.

```

classdef lmsSysObj < matlab.System & matlab.system.mixin.CustomIcon
%lmsSysObj Least mean squares (LMS) adaptive filtering.

methods(Static, Access=protected)

function header = getHeaderImpl
header = matlab.system.display.Header(...
'Title', 'LMS Adaptive Filter');
end

function groups = getPropertyGroupsImpl
firstGroup = matlab.system.display.SectionGroup(...
'Title', 'General', ...
'PropertyList', {'Mu'});

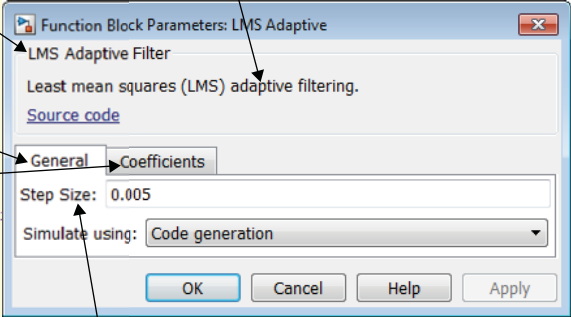
secondGroup = matlab.system.display.SectionGroup(...
'Title', 'Coefficients', ...
'PropertyList', {'TrueCoefficients', 'NumCoeffs'});

groups = [firstGroup, secondGroup];
end
end
    
```

Header description from class help summary

Specified header title overrides class name

Tabs from SectionGroups

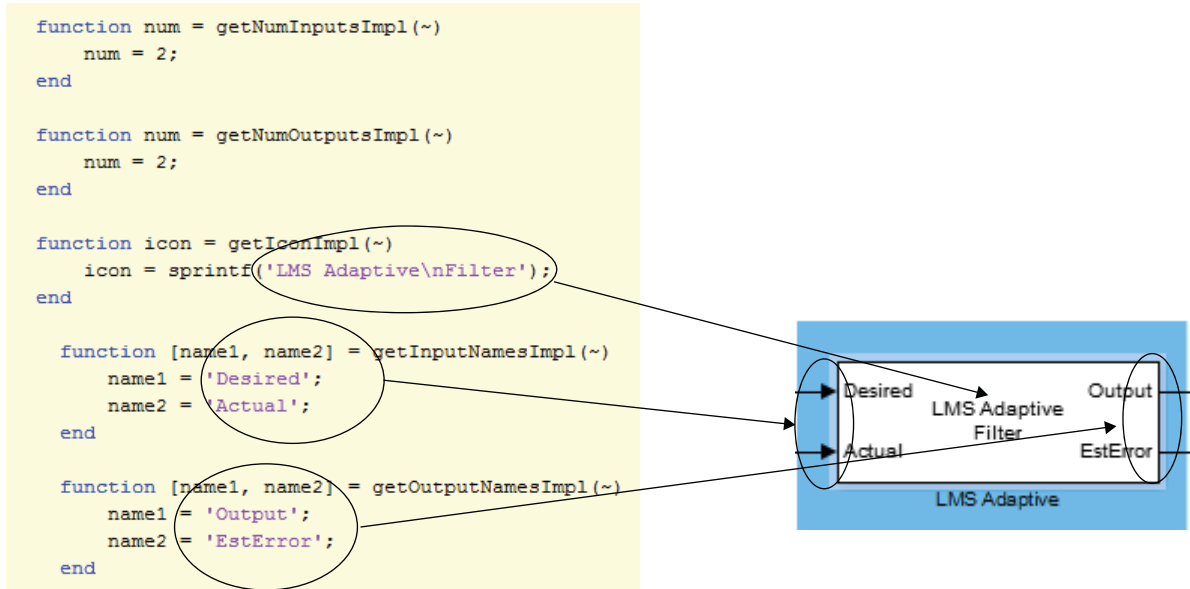


```

properties
% Mu Step Size
Mu = 0.005;
end
    
```

Label from product help summary

The source code for this System object also defines two input and output ports for the block.



Concepts

- “Change Block Icon and Port Labels” on page 32-11
- “Modify MATLAB System Block Dialog” on page 32-11

Considerations for Using System Objects in Simulink

System Objects in Simulink

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 `matlab.system.mixin.Propagates` methods. This is true 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`.

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.

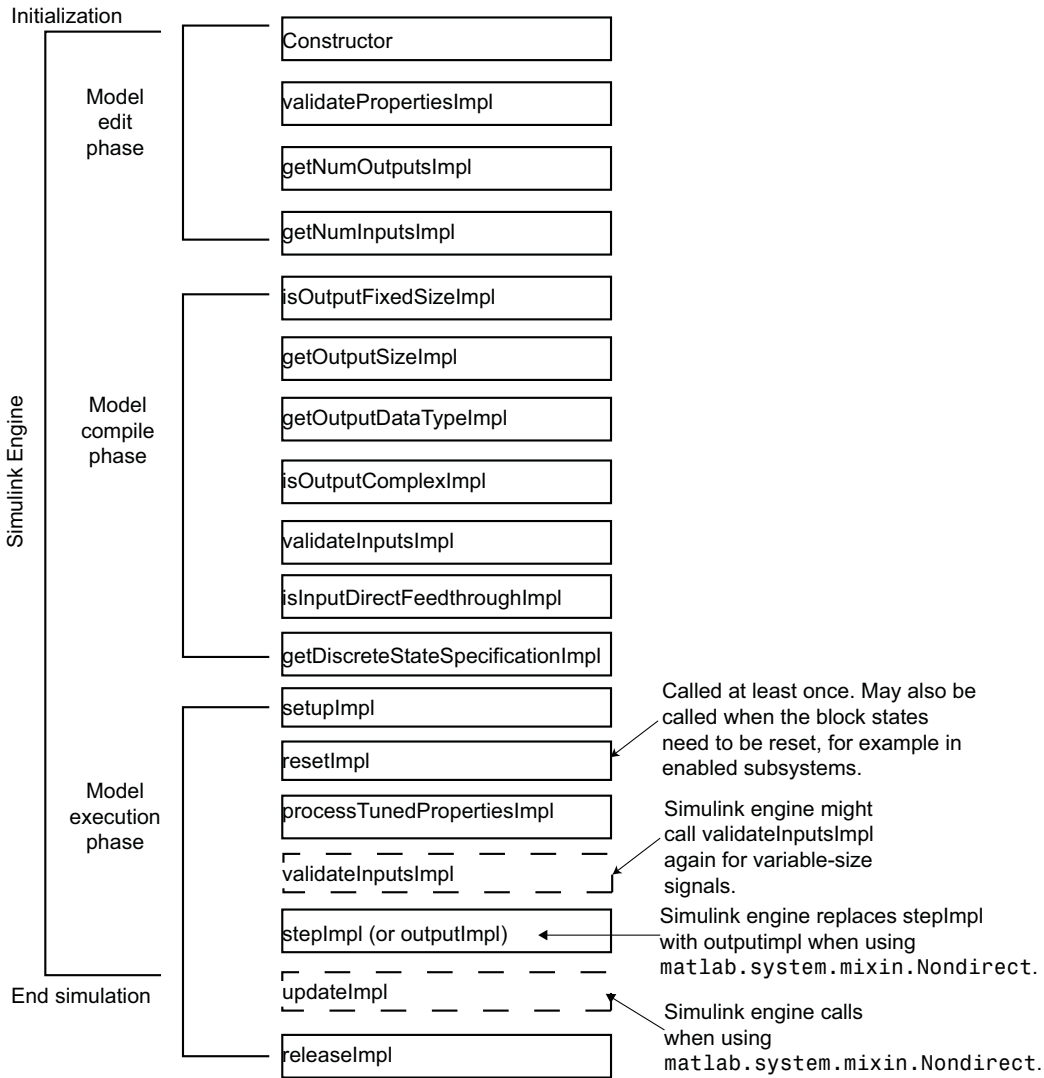
Default Property Values

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.

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.



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 `matlab.system.mixin.Propagates`.
- Simulink calls `saveObjectImpl` and `loadObjectImpl` for saving and restoring `SimState` and the `Simulation Stepper`.
- Default implementations save and restore all properties with public access, including `DiscreteState`.

Add and Implement Propagation Methods

In this section...

“When to Use Propagation Methods” on page 32-27

“Add Propagation Methods to System Objects” on page 32-27

“Implement Propagation Methods” on page 32-28

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.

Add Propagation Methods to System Objects

Propagation methods are in the class `matlab.system.mixin.Propagates`. To add these methods to the System object, add the `matlab.system.mixin.Propagates` class to the top of the parent class file for the System object, after the `matlab.System` class. For example:

```
classdef Counter < matlab.System & matlab.system.mixin.Propagates
```

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 <code>getOutputSize</code> .	<code>getOutputSizeImpl</code>	<ul style="list-style-type: none"> • Only one input • Only one output • An input size that is the same as the output size 	<ul style="list-style-type: none"> • <code>FindIfFixedInput</code> or <code>FindIfVarSizeInput</code> block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis block in Illustration of Law of Large Numbers
Gets data types of output ports. The associated method is <code>getOutputDataType</code> .	<code>getOutputDataTypeImpl</code>	<ul style="list-style-type: none"> • Only one input • Only one output • Output data type always the same as the input data type 	<ul style="list-style-type: none"> • <code>FindIfFixedInput</code> or <code>FindIfVarSizeInput</code> block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis block in Illustration of Law of Large Numbers

Description	Propagation Method	Default Implementation Should Suffice if	Example
Indicates whether output ports are complex or not. The associated method is <code>isOutputComplex</code> .	<code>isOutputComplexImpl</code>	<ul style="list-style-type: none"> • Only one input • Only one output • Output complexity always the same as the input complexity 	<ul style="list-style-type: none"> • <code>FindIfFixedInput</code> or <code>FindIfVarSizeInput</code> block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis block in Illustration of Law of Large Numbers
Whether output ports are fixed size. The associated method is <code>isOutputFixedSize</code> .	<code>isOutputFixedSizeImpl</code>	<ul style="list-style-type: none"> • Only one input • Only one output • Output and input are fixed-size 	<ul style="list-style-type: none"> • <code>FindIfFixedInput</code> or <code>FindIfVarSizeInput</code> block in MATLAB System Block with Variable-Size Input and Output Signals • Analysis block in Illustration of Law of Large Numbers
Gets the size, data type, and complexity of a discrete state property. The associated method is <code>getDiscreteStateSpecification</code> .	<code>getDiscreteStateSpecificationImpl</code>	No <code>DiscreteState</code> properties	N/A

Concepts

- “System Block Output Specifications”

Troubleshoot System Objects in Simulink

In this section...

“Class Not Found” on page 32-30

“Error Invoking Object Method” on page 32-30

“Performance” on page 32-31

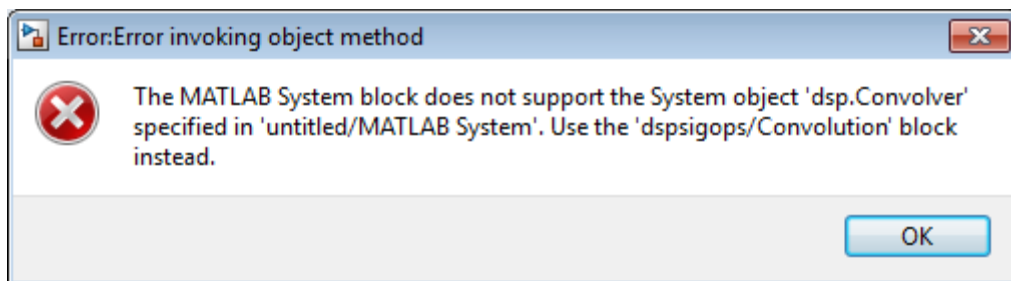
Class Not Found

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.

Concepts

- “Add and Implement Propagation Methods” on page 32-27

Design Considerations for C/C++ Code Generation

- “When to Generate Code from MATLAB Algorithms” on page 33-2
- “Which Code Generation Feature to Use” on page 33-4
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 33-6
- “MATLAB Code Design Considerations for Code Generation” on page 33-7
- “Expected Differences in Behavior After Compiling MATLAB Code” on page 33-9
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 33-13

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 Builder™ JA
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

To:	Use:
Deploy web-based or Windows applications	<ul style="list-style-type: none">• MATLAB Builder NE• MATLAB Builder JA
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 “MEX Function Generation at the Command Line”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder user interface	MATLAB Coder	Try this in “C Code Generation Using the Project Interface”.
	codegen function	MATLAB Coder	Try this in “C Code Generation at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder user interface	MATLAB Coder	See “Accelerate MATLAB Algorithms”.
	codegen function	MATLAB Coder	
Integrate MATLAB code into Simulink	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code” on page 29-178.
Speed up fixed-point MATLAB code	fiaccel function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB”.
Integrate custom C code into MATLAB and generate efficient, readable code	codegen function	MATLAB Coder	Learn more in “Specify External File Locations”.

To...	Use...	Required Product	To Explore Further...
Integrate custom C code into code generated from MATLAB	<code>coder.ceval</code> function	MATLAB Coder	Learn more in <code>coder.ceval</code> .
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 32-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”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” on page 38-4

Expected Differences in Behavior After Compiling MATLAB Code

In this section...

“Why Are There Differences?” on page 33-9

“Character Size” on page 33-9

“Order of Evaluation in Expressions” on page 33-9

“Termination Behavior” on page 33-10

“Size of Variable-Size N-D Arrays” on page 33-10

“Size of Empty Arrays” on page 33-11

“Floating-Point Numerical Results” on page 33-11

“NaN and Infinity Patterns” on page 33-12

“Code Generation Target” on page 33-12

“MATLAB Class Initial Values” on page 33-12

“Variable-Size Support for Code Generation” on page 33-12

Why Are There Differences?

To convert MATLAB code to C/C++ code that works efficiently, the code generation process introduces optimizations that intentionally cause the generated code to behave differently — and sometimes produce different results — from the original source code. This section describes these differences.

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 “Code Generation for Characters” on page 37-6.

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. For example, rewrite:

```
A = f1() + f2();
```

as

```
A = f1();  
A = A + f2();
```

so that the generated code calls `f1` before `f2`.

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they do not have side effects. As a result, the generated code may 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 38-24.

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 38-25.

Floating-Point Numerical Results

The generated code might not produce the same floating-point numerical results as MATLAB in the following situations:

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, which may produce different results from platform-specific BLAS implementations in MATLAB.

NaN and Infinity Patterns

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.

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 Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation” on page 41-4.

Variable-Size Support for Code Generation

For incompatibilities with MATLAB in variable-size support for code generation, see:

- “Incompatibility with MATLAB for Scalar Expansion”
- “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays”
- “Incompatibility with MATLAB in Determining Size of Empty Arrays”
- “Incompatibility with MATLAB in Vector-Vector Indexing”
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation”

MATLAB Language Features Supported for C/C++ Code Generation

MATLAB supports the following language features in generated code:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Variable-Size Data Definition for Code Generation” on page 38-3)
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 38-28)
- Complex numbers (see “Code Generation for Complex Data” on page 37-4)
- Numeric classes (see “Supported Variable Types” on page 36-18)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB”)
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see “Define and Initialize Persistent Variables” on page 36-10)
- Global variables.
- Structures
- Characters (see “Code Generation for Characters” on page 37-6)
- Function handles
- Frames (see “Add Frame-Based Signals” on page 29-129)
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions
- MATLAB classes

- Ability to call functions (see “Resolution of Function Calls in MATLAB Generated Code” on page 44-2)

MATLAB Language Features Not Supported for C/C++ Code Generation

MATLAB does not support the following features in generated code:

- Anonymous functions
- Cell arrays
- Java
- Nested functions
- Recursion
- Sparse matrices
- try/catch statements

Functions Supported for Code Generation

- “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2
- “Functions Supported for C/C++ Code Generation — Categorical List” on page 34-80

Functions Supported for C/C++ Code Generation – Alphabetical List

You can generate efficient C/C++ code for a subset of MATLAB and toolbox functions that you call from MATLAB code. In generated code, each supported function has the same name, arguments, and functionality as its MATLAB or toolbox counterparts. However, to generate code for these functions, you must adhere to certain limitations when calling them from your MATLAB source code. These limitations appear in the list below.

To find supported functions by MATLAB category or toolbox, see “Functions Supported for C/C++ Code Generation — Categorical List” on page 34-80.

Note For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB”.

Function	Product	Remarks and Limitations
abs	MATLAB	—
abs	Fixed-Point Designer	—
accumneg	Fixed-Point Designer	—
accumpos	Fixed-Point Designer	—
acos	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
acosd	MATLAB	—

Function	Product	Remarks and Limitations
acosh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Designer	—
aicstest	Phased Array System Toolbox™	Does not support variable-size inputs.
albersheim	Phased Array System Toolbox	Does not support variable-size inputs.
all	MATLAB	—
all	Fixed-Point Designer	—
ambgfun	Phased Array System Toolbox	Does not support variable-size inputs.
and	MATLAB	—
any	MATLAB	—
any	Fixed-Point Designer	—
aperture2gain	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
asec	MATLAB	—
asecd	MATLAB	—
asech	MATLAB	—
asin	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
asind	MATLAB	—
asinh	MATLAB	—
assert	MATLAB	<ul style="list-style-type: none"> Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time. For standalone code generation, excluded from the generated code.
atan	MATLAB	—
atan2	MATLAB	—
atan2	Fixed-Point Designer	—
atan2d	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.

Function	Product	Remarks and Limitations
az2broadside	Phased Array System Toolbox	Does not support variable-size inputs.
azel2phitheta	Phased Array System Toolbox	Does not support variable-size inputs.
azel2phithetapat	Phased Array System Toolbox	Does not support variable-size inputs.
azel2uv	Phased Array System Toolbox	Does not support variable-size inputs.
azel2uvpat	Phased Array System Toolbox	Does not support variable-size inputs.
azelaxes	Phased Array System Toolbox	Does not support variable-size inputs.
barthannwin	Signal Processing Toolbox™	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change.
bartlett	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change.
beat2range	Phased Array System Toolbox	Does not support variable-size inputs.
besselap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter order must be a constant. Expressions or variables are allowed if their values do not change.
beta	MATLAB	—
betacdf	Statistics Toolbox™	—
betainc	MATLAB	—

Function	Product	Remarks and Limitations
betaincinv	MATLAB	—
betainv	Statistics Toolbox	—
betaIn	MATLAB	—
betapdf	Statistics Toolbox	—
betarnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
betastat	Statistics Toolbox	—
bi2de	Communications System Toolbox	—
billingsleyicm	Phased Array System Toolbox	Does not support variable-size inputs.
bin2dec	MATLAB	<ul style="list-style-type: none"> • Does not match MATLAB when the input is empty.
binaryFeatures	Computer Vision System Toolbox	—
binocdf	Statistics Toolbox	—
binoinv	Statistics Toolbox	—
binopdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
binornd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
binostat	Statistics Toolbox	—
bitand	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—
bitget	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitget	Fixed-Point Designer	—
bitmax	MATLAB	—
bitor	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point inputs. The arguments must belong to an unsigned integer class.

Function	Product	Remarks and Limitations
<code>bitor</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for slope-bias scaled <code>fi</code> objects.
<code>bitorreduce</code>	Fixed-Point Designer	—
<code>bitreplicate</code>	Fixed-Point Designer	—
<code>bitrevorder</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time.
<code>bitrol</code>	Fixed-Point Designer	—
<code>bitror</code>	Fixed-Point Designer	—
<code>bitset</code>	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
<code>bitset</code>	Fixed-Point Designer	—
<code>bitshift</code>	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
<code>bitshift</code>	Fixed-Point Designer	—
<code>bitsliceget</code>	Fixed-Point Designer	—
<code>bitsll</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • Generated code may not handle out of range shifting.
<code>bitsra</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • Generated code may not handle out of range shifting.
<code>bitsrl</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • Generated code may not handle out of range shifting.

Function	Product	Remarks and Limitations
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitxor	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled fi objects.
bitxorreduce	Fixed-Point Designer	—
blackman	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change.
blackmanharris	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change.
blanks	MATLAB	—
blkdiag	MATLAB	—
bohmanwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change.
broadside2az	Phased Array System Toolbox	Does not support variable-size inputs.
bsxfun	MATLAB	—
buttap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter order must be a constant. Expressions or variables are allowed if their values do not change.

Function	Product	Remarks and Limitations
butter	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
buttord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
bwlookup	Image Processing Toolbox™	<ul style="list-style-type: none"> • For best results, specify an input image of class <code>logical</code>.
bwmorph	Image Processing Toolbox	<ul style="list-style-type: none"> • The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cart2sphvec	Phased Array System Toolbox	Does not support variable-size inputs.
cast	MATLAB	—
cat	MATLAB	—
cbfweights	Phased Array System Toolbox	Does not support variable-size inputs.
cdf	Statistics Toolbox	—
ceil	MATLAB	—
ceil	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
cfirpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
char	MATLAB	—
cheb1ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Product	Remarks and Limitations
cheby2	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change.
chi2cdf	Statistics Toolbox	—
chi2inv	Statistics Toolbox	—
chi2pdf	Statistics Toolbox	—
chi2rnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> The output is nonscalar. An input parameter is invalid for the distribution.
chi2stat	Statistics Toolbox	—
chol	MATLAB	<ul style="list-style-type: none"> When there are two output arguments, either make the input matrix variable-size in both dimensions, or, if the input matrix must be fixed size, copy the input matrix to a variable-size matrix before calling chol. <pre> coder.varsize('B'); B = A; [B,p] = chol(B); </pre>
circpol2pol	Phased Array System Toolbox	Does not support variable-size inputs.
circshift	MATLAB	—
class	MATLAB	—

Function	Product	Remarks and Limitations
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Designer	—
computer	MATLAB	<ul style="list-style-type: none"> Information about the computer on which the code generation software is running. Use only when the code generation target is S-function (Simulation) or MEX-function.
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Designer	—
conndef	Image Processing Toolbox	All input arguments must be compile-time constants.
conv	MATLAB	—
conv	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar,

Function	Product	Remarks and Limitations
		MATLAB computes the output using the ProductMode of the governing fimath.
conv2	MATLAB	—
convergent	Fixed-Point Designer	—
convn	MATLAB	—
cordicabs	Fixed-Point Designer	• Variable-size signals are not supported.
cordicangle	Fixed-Point Designer	• Variable-size signals are not supported.
cordicatan2	Fixed-Point Designer	• Variable-size signals are not supported.
cordiccart2pol	Fixed-Point Designer	• Variable-size signals are not supported.
cordiccxp	Fixed-Point Designer	• Variable-size signals are not supported.
cordiccos	Fixed-Point Designer	• Variable-size signals are not supported.
cordicpol2cart	Fixed-Point Designer	• Variable-size signals are not supported.
cordicrotate	Fixed-Point Designer	• Variable-size signals are not supported.
cordicsin	Fixed-Point Designer	• Variable-size signals are not supported.
cordicsincos	Fixed-Point Designer	• Variable-size signals are not supported.
cornerPoints	Computer Vision System Toolbox	—
corrcoef	MATLAB	• Row-vector input is only supported when the first two inputs are vectors and nonscalar.
cos	MATLAB	—

Function	Product	Remarks and Limitations
cos	Fixed-Point Designer	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant.
csc	MATLAB	—
cscd	MATLAB	—
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Designer	—
cumprod	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to <code>double</code> first.
cumsum	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to <code>double</code> first.
cumtrapz	MATLAB	—
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Length of transform dimension must be a power of two. If specified, the <code>pad</code> or <code>truncation</code> value must be constant. Expressions or variables are allowed if their values do not change.
de2bi	Communications System Toolbox	—
deal	MATLAB	—

Function	Product	Remarks and Limitations
deblank	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from the char class. • Input values must be in the range 0-127.
dec2bin	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant and <i>n</i> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <i>n</i> must be at least 52 for double, 23 for single, 16 for char, 32 for int32, 16 for int16, and so on.
dec2hex	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant and <i>n</i> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <i>n</i> must be at least 13 for double, 6 for single, 4 for char, 8 for int32, 4 for int16, and so on.
dechirp	Phased Array System Toolbox	Does not support variable-size inputs.
deconv	MATLAB	—
del2	MATLAB	—
delayseq	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
depressionang	Phased Array System Toolbox	Does not support variable-size inputs.
det	MATLAB	—
detectFASTFeatures	Computer Vision System Toolbox	—
detectMSERFeatures	Computer Vision System Toolbox	—
detectSURFFeatures	Computer Vision System Toolbox	—
detrend	MATLAB	<ul style="list-style-type: none"> • If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> ▪ Be real. ▪ Be sorted in ascending order. ▪ Restrict elements to integers in the interval $[1, n-2]$. n is the number of elements in a column of input argument X, or the number of elements in X when X is a row vector. ▪ Contain all unique values.
diag	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value. • For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), <code>diag</code>: <ul style="list-style-type: none"> ▪ Treats the input as a vector input. ▪ Returns a matrix with the given vector along the specified diagonal. • For variable-size inputs that are not variable-length vectors, <code>diag</code>:

Function	Product	Remarks and Limitations
		<ul style="list-style-type: none"> ▪ Treats the input as a matrix. ▪ Does not support inputs that are vectors at run time. ▪ Returns a variable-length vector. <p>If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, the output is 0-by-1 not 0-by-0. However, if the input is a constant size 0-by-0, the output is [].</p> <ul style="list-style-type: none"> • For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), <code>diag</code> treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force <code>diag</code> to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use: <ul style="list-style-type: none"> ▪ <code>diag(x(:))</code> instead of <code>diag(x)</code> ▪ <code>diag(x(:),k)</code> instead of <code>diag(x,k)</code>
<code>diag</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • If supplied, the index, <i>k</i>, must be a real and scalar integer value that is not a <code>fi</code> object.
<code>diff</code>	MATLAB	<ul style="list-style-type: none"> • If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.
<code>disparity</code>	Computer Vision System Toolbox	—
<code>divide</code>	Fixed-Point Designer	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant. Its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Complex and imaginary divisors are not supported. • The syntax <code>T.divide(a,b)</code> is not supported.

Function	Product	Remarks and Limitations
dop2speed	Phased Array System Toolbox	Does not support variable-size inputs.
dopsteeringvec	Phased Array System Toolbox	Does not support variable-size inputs.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Designer	—
downsample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs.
dpss	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change.
effearthradius	Phased Array System Toolbox	Does not support variable-size inputs.
eig	MATLAB	<ul style="list-style-type: none"> QZ algorithm used in all cases, whereas MATLAB might use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors, and the eigenvalues in D might not be in the same order as in MATLAB. With one input, $[V,D] = \text{eig}(A)$, the results will be similar to those obtained using $[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ in MATLAB, except that for code generation, the columns of V are normalized. Options 'balance', 'nobalance' are not supported for the standard eigenvalue problem, and 'chol' is not supported for the symmetric generalized eigenvalue problem.

Function	Product	Remarks and Limitations
		<ul style="list-style-type: none"> • Outputs are of complex type.
ellip	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipke	MATLAB	—
ellipord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
end	Fixed-Point Designer	—
epipolarLine	Computer Vision System Toolbox	—
eps	MATLAB	—
eps	Fixed-Point Designer	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	MATLAB	—
eq	Fixed-Point Designer	Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—

Function	Product	Remarks and Limitations
erfinv	MATLAB	—
error	MATLAB	For standalone code generation, excluded from the generated code.
espritdoa	Phased Array System Toolbox	Does not support variable-size inputs.
estimate Fundamental Matrix	Computer Vision System Toolbox	—
estimate Uncalibrated Rectification	Computer Vision System Toolbox	—
extractFeatures	Computer Vision System Toolbox	—
evcdf	Statistics Toolbox	—
evinv	Statistics Toolbox	—
evpdf	Statistics Toolbox	—
evrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
evstat	Statistics Toolbox	—
exp	MATLAB	—
expcdf	Statistics Toolbox	—
expint	MATLAB	—

Function	Product	Remarks and Limitations
<code>expinv</code>	Statistics Toolbox	—
<code>expm</code>	MATLAB	—
<code>expm1</code>	MATLAB	—
<code>exppdf</code>	Statistics Toolbox	—
<code>exprnd</code>	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
<code>expstat</code>	Statistics Toolbox	—
<code>extractFeatures</code>	Computer Vision System Toolbox	—
<code>eye</code>	MATLAB	<ul style="list-style-type: none"> • <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>eye</code> method for other classes. For example, <code>eye(m, n, 'myclass')</code> does not invoke <code>myclass.eye(m,n)</code>.
<code>factor</code>	MATLAB	<ul style="list-style-type: none"> • The maximum double precision input is 2^{33}. • The maximum single precision input is 2^{25}. • The input <code>n</code> cannot have type <code>int64</code> or <code>uint64</code>.
<code>factorial</code>	MATLAB	—
<code>false</code>	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
<code>fcdf</code>	Statistics Toolbox	—

Function	Product	Remarks and Limitations
fclose	MATLAB	—
fft	MATLAB	<ul style="list-style-type: none"> Length of input vector must be a power of 2.
fft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftshift	MATLAB	—
fi	Fixed-Point Designer	<ul style="list-style-type: none"> Use to create a fixed-point constant or variable. The default constructor syntax without input arguments is not supported. The syntax <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>. If the input value is not known at compile time, you must provide complete <code>numericType</code> information. <code>numericType</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter	Fixed-Point Designer	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
filter2	MATLAB	—

Function	Product	Remarks and Limitations
filtfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fimath	Fixed-Point Designer	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the fimath object defined in the MATLAB Function dialog in the Model Explorer. • Use to create fimath objects in generated code.
find	MATLAB	<ul style="list-style-type: none"> • Issues an error if a variable-sized input becomes a row vector at run time. <hr/> <p>Note This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> • For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.
finv	Statistics Toolbox	—
fir1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Product	Remarks and Limitations
fir2	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
firls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpmord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
fix	MATLAB	—
fix	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
<code>fixed.Quantizer</code>	Fixed-Point Designer	—
<code>flattopwin</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>flintmax</code>	MATLAB	—
<code>flipdim</code>	MATLAB	—
<code>fliplr</code>	MATLAB	—
<code>flipud</code>	MATLAB	—
<code>floor</code>	MATLAB	—
<code>floor</code>	Fixed-Point Designer	—
<code>fopen</code>	MATLAB	<ul style="list-style-type: none"> • Does not support: <ul style="list-style-type: none"> ▪ <code>machineformat</code>, encoding, or <code>fileID</code> inputs ▪ message output ▪ <code>fopen('all')</code> • If you disable extrinsic calls, you cannot return <code>fileIDs</code> created with <code>fopen</code> to MATLAB or extrinsic functions. You can only use such <code>fileIDs</code> internally. • You can open up to 20 files when generating C/C++ executables, static libraries, or dynamic libraries.
<code>fpdf</code>	Statistics Toolbox	—

Function	Product	Remarks and Limitations
fprintf	MATLAB	<ul style="list-style-type: none"> • Does not support: <ul style="list-style-type: none"> ▪ b and t subtypes on %u, %o %x, and %X formats ▪ \$ flag for reusing input arguments ▪ printing arrays • There is no automatic casting. Input arguments must match their format types for predictable results. • Escaped characters are limited to the decimal range of 0–127. • A call to fprintf with fileID equal to 1 or 2 becomes printf in the generated C/C++ code in the following cases: <ul style="list-style-type: none"> ▪ The fprintf call is inside a parfor loop. ▪ Extrinsic calls are disabled. • When the MATLAB behavior differs from the C compiler behavior, fprintf matches the C compiler behavior in the following cases: <ul style="list-style-type: none"> ▪ The format specifier has a corresponding C format specifier, for example, %e or %E. ▪ The fprintf call is inside a parfor loop. ▪ Extrinsic calls are disabled. • When you call fprintf with the format specifier %s, do not put a null character in the middle of the input string. Use fprintf(fid, '%c', char(0)) to write a null character. • When you call fprintf with an integer format specifier, the type of the integer argument must be a type that the target

Function	Product	Remarks and Limitations
		hardware can represent as a native C type. For example, if you call <code>fprintf('%d', int64(n))</code> , the target hardware must have a native C type that supports a 64-bit integer.
<code>freqspace</code>	MATLAB	—
<code>freqz</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • <code>freqz</code> with no output arguments produces a plot only when the function call terminates in a semicolon. See “<code>freqz</code> With No Output Arguments”.
<code>frnd</code>	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
<code>fspecial</code>	Image Processing Toolbox	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
<code>fsp1</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>fstat</code>	Statistics Toolbox	—
<code>full</code>	MATLAB	—

Function	Product	Remarks and Limitations
fzero	MATLAB	<ul style="list-style-type: none"> • The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument. • Supports up to three output arguments. Does not support the fourth output argument (the output structure). • Only supports the TolX and FunValCheck fields of an options input structure. Ignores other options in an options input structure. You cannot use the optimset function to create the options structure. Create this structure directly, for example, <pre style="margin-left: 20px;">opt.TolX = tol; opt.FunValCheck = 'on';</pre> The input structure field names must match exactly.
gain2aperture	Phased Array System Toolbox	Does not support variable-size inputs.
gamcdf	Statistics Toolbox	—
gaminv	Statistics Toolbox	—
gamma	MATLAB	—
gammainc	MATLAB	—
gammaincinv	MATLAB	—
gammaln	MATLAB	—
gampdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
gamrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
gamstat	Statistics Toolbox	—
gaussfir	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
gausswin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
geocdf	Statistics Toolbox	—
geoinv	Statistics Toolbox	—
geomean	Statistics Toolbox	—
geopdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
geornd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
geostat	Statistics Toolbox	—
get	Fixed-Point Designer	<ul style="list-style-type: none"> • The syntax <code>structure = get(o)</code> is not supported.
getlsb	Fixed-Point Designer	—
getmsb	Fixed-Point Designer	—
gevpdf	Statistics Toolbox	—
gevinv	Statistics Toolbox	—
gevpdf	Statistics Toolbox	—
gevrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
gevstat	Statistics Toolbox	—
global2localcoord	Phased Array System Toolbox	Does not support variable-size inputs.
gpcdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
gpinv	Statistics Toolbox	—
gppdf	Statistics Toolbox	—
gprnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
gpstat	Statistics Toolbox	—
gradient	MATLAB	—
grazingang	Phased Array System Toolbox	Does not support variable-size inputs.
gt	MATLAB	—
gt	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
hadamard	MATLAB	—
hamming	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
hankel	MATLAB	—
hann	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
harmmean	Statistics Toolbox	—

Function	Product	Remarks and Limitations
hdlram	Fixed-Point Designer	—
hex2dec	MATLAB	—
hex2num	MATLAB	<ul style="list-style-type: none"> For $n = \text{hex2num}(S)$, $\text{size}(S,2) \leq \text{length}(\text{num2hex}(0))$
hilb	MATLAB	—
hist	MATLAB	<ul style="list-style-type: none"> Histogram bar plotting not supported; call with at least one output argument. If supplied, the second argument x must be a scalar constant. Inputs must be real.
histc	MATLAB	<ul style="list-style-type: none"> The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.
horizonrange	Phased Array System Toolbox	Does not support variable-size inputs.
horzcat	Fixed-Point Designer	—
hygecdf	Statistics Toolbox	—
hygeinv	Statistics Toolbox	—
hygepdf	Statistics Toolbox	—
hygernd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> The output is nonscalar. An input parameter is invalid for the distribution.

Function	Product	Remarks and Limitations
hygestat	Statistics Toolbox	—
hypot	MATLAB	—
icdf	Statistics Toolbox	—
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
idivide	MATLAB	<ul style="list-style-type: none"> • For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> • Length of input vector must be a power of 2. • Output of ifft block is complex. • Does not support the 'symmetric' option.
ifft2	MATLAB	<ul style="list-style-type: none"> • Length of input matrix dimensions must each be a power of 2. • Does not support the 'symmetric' option.
ifftn	MATLAB	<ul style="list-style-type: none"> • Length of input matrix dimensions must each be a power of 2. • Does not support the 'symmetric' option.
ifftshift	MATLAB	—
imag	MATLAB	—
imag	Fixed-Point Designer	—
imcomplement	Image Processing Toolbox	Does not support int64 and uint64 data types.

Function	Product	Remarks and Limitations
imfill	Image Processing Toolbox	<p>The optional input connectivity, <code>conn</code>, and the string 'holes' must be compile-time constants.</p> <p>Supports only up to 3-D inputs. (No N-D support.)</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code>, is not supported in code generation.</p> <p><code>locations</code> can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile time, you cannot change it at run time. However, the number of points in <code>locations</code> can be varied at run time.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmax	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmin	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imreconstruct	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Function	Product	Remarks and Limitations
imregionalmax	Image Processing Toolbox	The optional second input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imregionalmin	Image Processing Toolbox	The optional second input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
ind2sub	MATLAB	<ul style="list-style-type: none"> The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.
inf	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
insertMarker	Computer Vision System Toolbox	<ul style="list-style-type: none"> 'Color' input cannot be a cell array. position input cannot be a cornerPoints object. marker input must be constant. marker input cannot be 's'.
insertShape	Computer Vision System Toolbox	<ul style="list-style-type: none"> position input cannot be a cell array. 'Color' input cannot be a cell array. shape input must be constant.
int8, int16, int32, int64	MATLAB	<p>You cannot use int64 in a :</p> <ul style="list-style-type: none"> MATLAB Function block in a Simulink model MATLAB function in a Stateflow chart

Function	Product	Remarks and Limitations
int8, int16, int32, int64	Fixed-Point Designer	<p>You cannot use int64 in a :</p> <ul style="list-style-type: none"> • MATLAB Function block in a Simulink model • MATLAB function in a Stateflow chart
integralImage	Computer Vision System Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> • Supports only linear and nearest interpolation methods. • Does not handle evenly spaced X indices separately. • X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
interp2	MATLAB	<ul style="list-style-type: none"> • Supports only $5 \leq \text{margin} \leq 7$. • XI and YI must be the same size. • Supports only 'linear' and 'nearest' methods. • For best results, supply X and Y as vectors. • When the X or Y inputs are not vectors, interp2 references only the first row of X and first column of Y. Supports "plaid" input for X and Y but does not verify that the input data is "plaid". • X and Y must contain monotonically increasing values. If your application provides monotonically decreasing values, first use flip1r and flipud to change X, Y, and Z to monotonically increasing form before calling interp2.

Function	Product	Remarks and Limitations
intersect	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs <code>ia</code> and <code>ib</code> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0. • When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> ▪ Set <code>setOrder</code> to 'stable'. ▪ Sort the real input in complex ascending order (by absolute value). Suppose the real input is <code>x</code>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.

Function	Product	Remarks and Limitations
intfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change.
intmax	MATLAB	—
intmin	MATLAB	—
inv	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	MATLAB	—
ipermute	MATLAB	—
iptcheckconn	Image Processing Toolbox	All input arguments must be compile-time constants.
iqr	Statistics Toolbox	—
isa	MATLAB	—
iscell	MATLAB	—
ischar	MATLAB	—
iscolumn	MATLAB	—
iscolumn	Fixed-Point Designer	—
isdeployed	MATLAB Compiler	<ul style="list-style-type: none"> Returns true and false as appropriate for MEX and SIM targets Returns false for other targets
isempty	MATLAB	—
isempty	Fixed-Point Designer	—
isEpipoleInImage	Computer Vision System Toolbox	—

Function	Product	Remarks and Limitations
isequal	MATLAB	—
isequal	Fixed-Point Designer	—
isequaln	MATLAB	—
isfi	Fixed-Point Designer	—
isfield	MATLAB	<ul style="list-style-type: none"> • Does not support cell input for second argument
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isfinite	Fixed-Point Designer	—
isfloat	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Designer	—
isinteger	MATLAB	—
isletter	MATLAB	<ul style="list-style-type: none"> • Input values from the char class must be in the range 0-127
islogical	MATLAB	—
ismac	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
ismatrix	MATLAB	—

Function	Product	Remarks and Limitations
ismcc	MATLAB Compiler	<ul style="list-style-type: none"> • Returns true and false as appropriate for MEX and SIM targets. • Returns false for other targets.
ismember	MATLAB	<ul style="list-style-type: none"> • The second input, B, must be sorted in ascending order. • Complex inputs must be single or double.
isnan	MATLAB	—
isnan	Fixed-Point Designer	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Designer	—
isnumerictype	Fixed-Point Designer	—
ispc	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version you use for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
isprime	MATLAB	<ul style="list-style-type: none"> • The maximum double precision input is 2^{33}. • The maximum single precision input is 2^{25}. • The input X cannot have type int64 or uint64.
isreal	MATLAB	—
isreal	Fixed-Point Designer	—
isrow	MATLAB	—
isrow	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
isscalar	MATLAB	—
isscalar	Fixed-Point Designer	—
assigned	Fixed-Point Designer	—
issorted	MATLAB	—
isspace	MATLAB	<ul style="list-style-type: none"> • Input values from the char class must be in the range 0–127.
issparse	MATLAB	—
isstrprop	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from char and integer classes. • Input values must be in the range 0-127.
isstruct	MATLAB	—
istrellis	Communications System Toolbox	—
isunix	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
isvector	MATLAB	—
isvector	Fixed-Point Designer	—
kaiser	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Product	Remarks and Limitations
kaiserord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Computation performed at run time.
kron	MATLAB	—
kurtosis	Statistics Toolbox	—
label2rgb	Image Processing Toolbox	<p>Referring to the standard syntax:</p> <p>RGB = label2rgb(L,map,zerocolor,order)</p> <ul style="list-style-type: none"> Submit at least two input arguments: the label matrix, L, and the colormap matrix, map. map must be an n-by-3, double, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. If you set the boundary color zerocolor to the same color as one of the regions, label2rgb will not issue a warning. If you supply a value for order, it must be 'noshuffle'.
lcm	MATLAB	—
lcmvweights	Phased Array System Toolbox	Does not support variable-size inputs.
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
levinson	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
lineToBorderPoints	Computer Vision System Toolbox	—
linsolve	MATLAB	<ul style="list-style-type: none"> • The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: <ul style="list-style-type: none"> ▪ UT ▪ LT ▪ UHESS = true (the TRANSA can be either true or false) ▪ SYM = true and POSDEF = true <p>Other options are equivalent to using <code>mldivide</code>.</p>
linspace	MATLAB	—
load	MATLAB	<ul style="list-style-type: none"> • Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>. • Does not support use of the function without assignment to a structure or array. For example, use <code>S = load(filename)</code>, not <code>load(filename)</code>. • The output <code>S</code> must be the name of a structure or array without any subscripting. For example, <code>S[i] = load('myFile.mat')</code> is not allowed.

Function	Product	Remarks and Limitations
		<ul style="list-style-type: none"> Arguments to <code>load</code> must be compile-time constant strings. Does not support loading objects. If the MAT-file contains unsupported constructs, use <code>load(filename, variables)</code> to load only the supported constructs. You cannot use <code>save</code> in a function intended for code generation. The code generation software does not support the <code>save</code> function. Furthermore, you cannot use <code>coder.extrinsic</code> with <code>save</code>. Prior to generating code, you can use <code>save</code> to save the workspace data to a MAT-file. <p>You must use <code>coder.varsize</code> to explicitly declare variable-size data loaded using the <code>load</code> function.</p>
<code>local2globalcoord</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>log</code>	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value <code>x</code> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
<code>log2</code>	MATLAB	—
<code>log10</code>	MATLAB	—
<code>log1p</code>	MATLAB	—
<code>logical</code>	MATLAB	—
<code>logical</code>	Fixed-Point Designer	—
<code>logncdf</code>	Statistics Toolbox	—

Function	Product	Remarks and Limitations
logninv	Statistics Toolbox	—
lognpdf	Statistics Toolbox	—
lognrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
lognstat	Statistics Toolbox	—
logspace	MATLAB	—
lower	MATLAB	<ul style="list-style-type: none"> • Supports only char inputs. • Input values must be in the range 0-127.
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	MATLAB	—
lt	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
lu	MATLAB	—
mad	Statistics Toolbox	Input <code>dim</code> cannot be empty.
magic	MATLAB	—

Function	Product	Remarks and Limitations
matchFeatures	Computer Vision System Toolbox	—
max	MATLAB	—
max	Fixed-Point Designer	—
maxflat	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Inputs must be constant. Expressions or variables are allowed if their values do not change.
mdltest	Phased Array System Toolbox	Does not support variable-size inputs.
mean	MATLAB	—
mean	Fixed-Point Designer	—
median	MATLAB	—
median	Fixed-Point Designer	—
meshgrid	MATLAB	—
mfilename	MATLAB	—
min	MATLAB	—
min	Fixed-Point Designer	—
minus	MATLAB	—
minus	Fixed-Point Designer	<ul style="list-style-type: none"> • Any non-fi input must be constant. Its value must be known at compile time so that it can be cast to a fi object.
mldivide	MATLAB	—
mnpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
mod	MATLAB	<ul style="list-style-type: none"> Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. <p>If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.</p>
mode	MATLAB	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array).
moment	Statistics Toolbox	If order is nonintegral and <code>X</code> is real, use <code>moment(complex(X), order)</code> .
mpower	MATLAB	—
mpower	Fixed-Point Designer	<ul style="list-style-type: none"> The exponent input, <code>k</code>, must be constant; that is, its value must be known at compile time. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.

Function	Product	Remarks and Limitations
mpy	Fixed-Point Designer	<ul style="list-style-type: none"> When you provide complex inputs to the mpy function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
mrdivide	MATLAB	—
mrdivide	Fixed-Point Designer	—
MSERRegions	Computer Vision System Toolbox	—
mtimes	MATLAB	—
mtimes	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar,

Function	Product	Remarks and Limitations
		MATLAB computes the output using the ProductMode of the governing fimath.
mvdweights	Phased Array System Toolbox	Does not support variable-size inputs.
NaN or nan	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
nancov	Statistics Toolbox	If the input is variable-size and is [] at run time, returns [] not NaN.
nanmax	Statistics Toolbox	—
nanmean	Statistics Toolbox	—
nanmedian	Statistics Toolbox	—
nanmin	Statistics Toolbox	—
nanstd	Statistics Toolbox	—
nansum	Statistics Toolbox	—
nanvar	Statistics Toolbox	—
nargchk	MATLAB	<ul style="list-style-type: none"> Output structure does not include stack information. <hr/> <p>Note nargchk will be removed in a future release.</p> <hr/>
nargin	MATLAB	—
narginchk	MATLAB	—

Function	Product	Remarks and Limitations
nargout	MATLAB	<ul style="list-style-type: none"> For a function with no output arguments, returns 1 if called without a terminating semicolon. <hr/> <p>Note This behavior also affects extrinsic calls with no terminating semicolon. <code>nargout</code> is 1 for the called function in MATLAB.</p> <hr/>
nargoutchk	MATLAB	—
nbincdf	Statistics Toolbox	—
nbininv	Statistics Toolbox	—
nbinpdf	Statistics Toolbox	—
nbinrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> The output is nonscalar. An input parameter is invalid for the distribution.
nbinstat	Statistics Toolbox	—
ncfcdf	Statistics Toolbox	—
ncfinv	Statistics Toolbox	—
ncfpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
nctrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nctstat	Statistics Toolbox	—
nchoosek	MATLAB	<ul style="list-style-type: none"> • When the first input, x, is a scalar, nchoosek returns a binomial coefficient. In this case, x must be a nonnegative integer. It cannot have type int64 or uint64. • When the first input, x, is a vector, nchoosek treats it as a set. In this case, x can have type int64 or uint64. • The second input, k, cannot have type int64 or uint64.
nctcdf	Statistics Toolbox	—
nctinv	Statistics Toolbox	—
nctpdf	Statistics Toolbox	—
nctrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nctstat	Statistics Toolbox	—

Function	Product	Remarks and Limitations
ncx2cdf	Statistics Toolbox	—
ncx2rnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
ncx2stat	Statistics Toolbox	—
ndgrid	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Designer	—
ne	MATLAB	—
ne	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
nearest	Fixed-Point Designer	—
nextpow2	MATLAB	—
nnz	MATLAB	—
noisepow	Phased Array System Toolbox	Does not support variable-size inputs.
nonzeros	MATLAB	—
norm	MATLAB	—
normcdf	Statistics Toolbox	—
normest	MATLAB	—
norminv	Statistics Toolbox	—

Function	Product	Remarks and Limitations
normpdf	Statistics Toolbox	—
normrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
normstat	Statistics Toolbox	—
not	MATLAB	—
npwgntthresh	Phased Array System Toolbox	Does not support variable-size inputs.
nthroot	MATLAB	—
null	MATLAB	<ul style="list-style-type: none"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
num2hex	MATLAB	—
numberofelements	Fixed-Point Designer	numberofelements will be removed in a future release. Use numel instead.
numel	MATLAB	—
numel	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
numerictype	Fixed-Point Designer	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a numerictype object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a non-fixed-point signal. • Use to create numerictype objects in the generated code.
nuttallwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
ones	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> • Might return a different basis than MATLAB
padarray	Image Processing Toolbox	<p>Supports only up to 3-D inputs.</p> <p>Input arguments, padval and direction, are expected to be compile-time constants.</p>
parzenwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
pascal	MATLAB	—
pdf	Statistics Toolbox	—
permute	MATLAB	—
permute	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
phitheta2azel	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2azelpat	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uv	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uvpat	Phased Array System Toolbox	Does not support variable-size inputs.
physconst	Phased Array System Toolbox	Does not support variable-size inputs.
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—
plus	Fixed-Point Designer	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
poisscdf	Statistics Toolbox	—
poissinv	Statistics Toolbox	—
poisspdf	Statistics Toolbox	—
poissrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> The output is nonscalar. An input parameter is invalid for the distribution.
poisstat	Statistics Toolbox	—

Function	Product	Remarks and Limitations
pol2cart	MATLAB	—
pol2circpol	Phased Array System Toolbox	Does not support variable-size inputs.
polellip	Phased Array System Toolbox	Does not support variable-size inputs.
polloss	Phased Array System Toolbox	Does not support variable-size inputs.
polratio	Phased Array System Toolbox	Does not support variable-size inputs.
polsignature	Phased Array System Toolbox	Does not support variable-size inputs.
poly	MATLAB	<ul style="list-style-type: none"> Does not discard nonfinite input values Complex input produces complex output
poly2trellis	Communications System Toolbox	—
polyfit	MATLAB	—
polyval	MATLAB	—
pow2	Fixed-Point Designer	—
power	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation. Returns NaN in generated code when both X and Y are real, but $\text{power}(X, Y)$ is complex. To get the complex result, make the input value X complex by passing in $\text{complex}(X)$. For example, $\text{power}(\text{complex}(X), Y)$. Generates an error during simulation. Returns NaN in generated code when both X and Y are real, but $X.^Y$ is complex. To get the complex result, make the input value X complex by using $\text{complex}(X)$. For example, $\text{complex}(X).^Y$.

Function	Product	Remarks and Limitations
power	Fixed-Point Designer	<ul style="list-style-type: none"> The exponent input, <code>k</code>, must be constant. Its value must be known at compile time.
prctile	Statistics Toolbox	<ul style="list-style-type: none"> “Automatic dimension restriction” If the output <code>Y</code> is a vector, the orientation of <code>Y</code> differs from MATLAB when all of the following are true: <ul style="list-style-type: none"> You do not supply the <code>dim</code> input. <code>X</code> is a variable-size array. <code>X</code> is not a variable-length vector. <code>X</code> is a vector at run time. The orientation of the vector <code>X</code> does not match the orientation of the vector <code>p</code>. In this case, the output <code>Y</code> matches the orientation of <code>X</code> not the orientation of <code>p</code>.
primes	MATLAB	<ul style="list-style-type: none"> The maximum double precision input is 2^{32}. The maximum single precision input is 2^{24}. The input <code>n</code> cannot have type <code>int64</code> or <code>uint64</code>.
prod	MATLAB	—
psi	MATLAB	—
pulsint	Phased Array System Toolbox	Does not support variable-size inputs.
qr	MATLAB	—

Function	Product	Remarks and Limitations
quad2d	MATLAB	<ul style="list-style-type: none"> Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.
quadgk	MATLAB	—
quantile	Statistics Toolbox	—
quantize	Fixed-Point Designer	—
quatconj	Aerospace Toolbox	—
quatdivide	Aerospace Toolbox	—
quatinv	Aerospace Toolbox	—
quatmod	Aerospace Toolbox	—
quatmultiply	Aerospace Toolbox	—
quatnorm	Aerospace Toolbox	—
quatnormalize	Aerospace Toolbox	—
radareqpow	Phased Array System Toolbox	Does not support variable-size inputs.
radareqrng	Phased Array System Toolbox	Does not support variable-size inputs.
radareqsnr	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
radarvcd	Phased Array System Toolbox	Does not support variable-size inputs.
radialspeed	Phased Array System Toolbox	Does not support variable-size inputs.
rand	MATLAB	—
randg	Statistics Toolbox	—
randi	MATLAB	—
randn	MATLAB	—
random	Statistics Toolbox	—
randperm	MATLAB	—
range	Fixed-Point Designer	—
range2beat	Phased Array System Toolbox	Does not support variable-size inputs.
range2bw	Phased Array System Toolbox	Does not support variable-size inputs.
range2time	Phased Array System Toolbox	Does not support variable-size inputs.
rangeangle	Phased Array System Toolbox	Does not support variable-size inputs.
rank	MATLAB	—
raylcdf	Statistics Toolbox	—
raylinv	Statistics Toolbox	—
raylpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
raylrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
raylstat	Statistics Toolbox	—
rcond	MATLAB	—
rdcoupling	Phased Array System Toolbox	Does not support variable-size inputs.
rdivide	MATLAB	—
rdivide	Fixed-Point Designer	—
real	MATLAB	—
real	Fixed-Point Designer	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Designer	—
realmin	MATLAB	—
realmin	Fixed-Point Designer	—
realpow	MATLAB	—
realsqrt	MATLAB	—

Function	Product	Remarks and Limitations
rectwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
reinterprecast	Fixed-Point Designer	—
rem	MATLAB	<ul style="list-style-type: none"> • Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. • If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.
removefimath	Fixed-Point Designer	—
repmat	MATLAB	—
repmat	Fixed-Point Designer	—
resample	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
rescale	Fixed-Point Designer	—
reshape	MATLAB	—
reshape	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
rng	MATLAB	<ul style="list-style-type: none"> For library and executable code generation targets, and for MEX targets when extrinsic calls are disabled, supports only the 'default' input and these generator inputs: <ul style="list-style-type: none"> 'twister' 'v4' 'v5normal' <p>For these targets, the output of <code>s=rng</code> in the generated code differs from the MATLAB output. You cannot return the output of <code>s=rng</code> from the generated code and pass it to <code>rng</code> in MATLAB.</p> <ul style="list-style-type: none"> For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.
rocdfa	Phased Array System Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Does not support <code>NonfluctuatingNoncoherent</code> signal type.
rocsnr	Phased Array System Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Does not support <code>NonfluctuatingNoncoherent</code> signal type.
rootmusicdoa	Phased Array System Toolbox	Does not support variable-size inputs.
roots	MATLAB	<ul style="list-style-type: none"> Output is variable size Output is complex Roots may not be in the same order as MATLAB Roots of poorly conditioned polynomials may not match MATLAB
rosser	MATLAB	—

Function	Product	Remarks and Limitations
rot90	MATLAB	—
rotx	Phased Array System Toolbox	Does not support variable-size inputs.
roty	Phased Array System Toolbox	Does not support variable-size inputs.
rotz	Phased Array System Toolbox	Does not support variable-size inputs.
round	MATLAB	—
round	Fixed-Point Designer	—
rsf2csf	MATLAB	—
schur	MATLAB	Might sometimes return a different Schur decomposition in generated code than in MATLAB.
sec	MATLAB	—
secd	MATLAB	—
sech	MATLAB	—
sensorcov	Phased Array System Toolbox	Does not support variable-size inputs.
sensorsig	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
setdiff	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ Do not use [] to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' and 'rows' options, the output <code>ia</code> is a column vector. If <code>ia</code> is empty, it is 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0. • When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> ▪ Set <code>setOrder</code> to 'stable'. ▪ Sort the real input in complex ascending order (by absolute value). Suppose the real input is <code>x</code>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.
setfimath	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
setxor	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors with the same orientation. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' flag, the inputs must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> ▪ Set setOrder to 'stable'. ▪ Sort the real input in complex ascending order (by absolute value). Suppose the real input is x. Use sort(complex(x)) or sortrows(complex(x)).

Function	Product	Remarks and Limitations
<code>sfi</code>	Fixed-Point Designer	—
<code>sgolay</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>shiftdim</code>	MATLAB	Second argument must be a constant.
<code>shnidman</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>sign</code>	MATLAB	—
<code>sign</code>	Fixed-Point Designer	—
<code>sin</code>	MATLAB	—
<code>sin</code>	Fixed-Point Designer	—
<code>sind</code>	MATLAB	—
<code>single</code>	MATLAB	—
<code>single</code>	Fixed-Point Designer	—
<code>sinh</code>	MATLAB	—
<code>size</code>	MATLAB	—
<code>size</code>	Fixed-Point Designer	—
<code>skewness</code>	Statistics Toolbox	—
<code>sort</code>	MATLAB	If the input is a complex type, <code>sort</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sort(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data” on page 37-4.

Function	Product	Remarks and Limitations
sort	Fixed-Point Designer	—
sortrows	MATLAB	If the input is a complex type, <code>sortrows</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sortrows(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data” on page 37-4.
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time.
speed2dop	Phased Array System Toolbox	Does not support variable-size inputs.
sph2cart	MATLAB	—
sph2cartvec	Phased Array System Toolbox	Does not support variable-size inputs.
spsmooth	Phased Array System Toolbox	Does not support variable-size inputs.
squeeze	MATLAB	—
sqrt	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN in generated code when the input value <code>x</code> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
sqrt	Fixed-Point Designer	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sqrtm	MATLAB	—
std	MATLAB	—
steervec	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
stokes	Phased Array System Toolbox	Does not support variable-size inputs.
storedInteger	Fixed-Point Designer	—
storedIntegerToDouble	Fixed-Point Designer	—
str2func	MATLAB	<ul style="list-style-type: none"> String must be constant/known at compile time
strcmp	MATLAB	—
strcmpi	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.
stretchfreq2rng	Phased Array System Toolbox	Does not support variable-size inputs.
strfind	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. If <code>pattern</code> does not exist in <code>str</code>, returns <code>zeros(1,0)</code> not <code>[]</code>. To check for an empty return, use <code>isempty</code>. Inputs must be character row vectors.
strjust	MATLAB	—
strncmp	MATLAB	—
strncmpi	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.
strep	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. If <code>oldSubstr</code> does not exist in <code>origStr</code>, returns <code>blanks(0)</code>. To check for an empty return, use <code>isempty</code>. Inputs must be character row vectors.
strtok	MATLAB	—

Function	Product	Remarks and Limitations
strtrim	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from the char class. • Input values must be in the range 0-127.
struct	MATLAB	—
structfun	MATLAB	<ul style="list-style-type: none"> • Does not support the ErrorHandler option. • The number of outputs must be less than or equal to three.
sub	Fixed-Point Designer	—
sub2ind	MATLAB	<ul style="list-style-type: none"> • The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.
subsasgn	Fixed-Point Designer	—
subspace	MATLAB	—
subsref	Fixed-Point Designer	—
sum	MATLAB	—
sum	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
surfacegamma	Phased Array System Toolbox	Does not support variable-size inputs.
surfclutterrcs	Phased Array System Toolbox	Does not support variable-size inputs.
SURFPoints	Computer Vision System Toolbox	—
svd	MATLAB	Uses a different SVD implementation than MATLAB. As the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.

Function	Product	Remarks and Limitations
swapbytes	MATLAB	Inheritance of the class of the input to swapbytes in a MATLAB Function block is supported only when the class of the input is double. For non-double inputs, the input port data types must be specified, not inherited.
systemp	Phased Array System Toolbox	Does not support variable-size inputs.
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
taylorwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Inputs must be constant
tcdf	Statistics Toolbox	—
time2range	Phased Array System Toolbox	Does not support variable-size inputs.
times	MATLAB	—
times	Fixed-Point Designer	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • When you provide complex inputs to the <code>times</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
tinvs	Statistics Toolbox	—
toeplitz	MATLAB	—

Function	Product	Remarks and Limitations
tpdf	Statistics Toolbox	—
trace	MATLAB	—
trapz	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Designer	—
triang	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change.
tril	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
tril	Fixed-Point Designer	<ul style="list-style-type: none"> • If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
triu	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
triu	Fixed-Point Designer	<ul style="list-style-type: none"> • If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
trnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
true	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
tstat	Statistics Toolbox	—

Function	Product	Remarks and Limitations
tukeywin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change.
typecast	MATLAB	<ul style="list-style-type: none"> Value of string input argument <code>type</code> must be lower case You might receive a size error when you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks. To avoid this error, specify the block's input port data types explicitly.
ufi	Fixed-Point Designer	—
uint8, uint16, uint32, uint64	MATLAB	You cannot use <code>uint64</code> in a : <ul style="list-style-type: none"> MATLAB Function block in a Simulink model MATLAB function in a Stateflow chart
uint8, uint16, uint32, uint64	Fixed-Point Designer	You cannot use <code>uint64</code> in a : <ul style="list-style-type: none"> MATLAB Function block in a Simulink model MATLAB function in a Stateflow chart
uminus	MATLAB	—
uminus	Fixed-Point Designer	—
unidcdf	Statistics Toolbox	—
unidinv	Statistics Toolbox	—
unidpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
unidrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
unidstat	Statistics Toolbox	—
unifcdf	Statistics Toolbox	—
unifinv	Statistics Toolbox	—
unifpdf	Statistics Toolbox	—
unifrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
unifstat	Statistics Toolbox	—
unifgrid	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
union	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ Inputs A and B must be vectors with the same orientation. If you specify the 'legacy' option, inputs A and B must be row vectors. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> ▪ Set setOrder to 'stable'. ▪ Sort the real input in complex ascending order (by absolute value). Suppose the real input is x. Use sort(complex(x)) or sortrows(complex(x)).

Function	Product	Remarks and Limitations
unique	MATLAB	<ul style="list-style-type: none"> • When you do not specify the 'rows' option: <ul style="list-style-type: none"> ▪ The input A must be a vector. If you specify the 'legacy' option, the input A must be a row vector. ▪ The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. ▪ The input [] is not supported. Use a 1-by-0 or 0-by-1 input, for example , zeros(1,0), to represent the empty set. ▪ If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'rows' option and the 'legacy' option, outputs ia and ic are column vectors. If these outputs are empty, they are 0-by-1, even if the output C is 0-by-0. • When the setOrder is 'sorted' or when you specify the 'legacy' option, the input A must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be single or double.
unwrap	MATLAB	<ul style="list-style-type: none"> • Row vector input is only supported when the first two inputs are vectors and nonscalar • Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors

Function	Product	Remarks and Limitations
upfirdn	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change.
uplus	MATLAB	—
uplus	Fixed-Point Designer	—
upper	MATLAB	<ul style="list-style-type: none"> Supports only char inputs. Input values must be in the range 0-127.
upperbound	Fixed-Point Designer	—
upsample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Either declare input n as constant, or use the <code>assert</code> function in the calling function to set upper bounds for n. For example, <code>assert(n<10)</code>
uv2azel	Phased Array System Toolbox	Does not support variable-size inputs.
uv2azelpat	Phased Array System Toolbox	Does not support variable-size inputs.
uv2phitheta	Phased Array System Toolbox	Does not support variable-size inputs.
uv2phithetapat	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
val2ind	Phased Array System Toolbox	Does not support variable-size inputs.
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Designer	—
wblcdf	Statistics Toolbox	—
wblinv	Statistics Toolbox	—
wblpdf	Statistics Toolbox	—
wblrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
wblstat	Statistics Toolbox	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Does not support the case where A is a matrix. • Does not support partial (abbreviated) strings of <code>biased</code>, <code>unbiased</code>, <code>coeff</code>, or <code>none</code>. • Computation performed at run time.
xor	MATLAB	—

Function	Product	Remarks and Limitations
yulewalk	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
zeros	MATLAB	<ul style="list-style-type: none">• Dimensions must be real, nonnegative, integers.
zp2tf	MATLAB	—
zscore	Statistics Toolbox	—

Functions Supported for C/C++ Code Generation – Categorical List

In this section...

- “Aerospace Toolbox Functions” on page 34-81
- “Arithmetic Operator Functions” on page 34-82
- “Bit-Wise Operation Functions” on page 34-82
- “Casting Functions” on page 34-83
- “Communications System Toolbox Functions” on page 34-83
- “Complex Number Functions” on page 34-83
- “Computer Vision System Toolbox Functions” on page 34-84
- “Data and File Management Functions” on page 34-86
- “Data Type Functions” on page 34-86
- “Derivative and Integral Functions” on page 34-87
- “Discrete Math Functions” on page 34-87
- “Error Handling Functions” on page 34-88
- “Exponential Functions” on page 34-88
- “Filtering and Convolution Functions” on page 34-89
- “Fixed-Point Designer Functions” on page 34-89
- “Histogram Functions” on page 34-98
- “Image Processing Toolbox Functions” on page 34-98
- “Input and Output Functions” on page 34-100
- “Interpolation and Computational Geometry Functions” on page 34-101
- “Linear Algebra” on page 34-101
- “Logical Operator Functions” on page 34-101
- “MATLAB® Compiler™ Functions” on page 34-102
- “MATLAB Desktop Environment Functions” on page 34-102
- “Matrix and Array Functions” on page 34-102

In this section...
“Nonlinear Numerical Methods” on page 34-106
“Phased Array System Toolbox Functions” on page 34-107
“Polynomial Functions” on page 34-110
“Programming Utilities” on page 34-110
“Relational Operator Functions” on page 34-110
“Rounding and Remainder Functions” on page 34-111
“Set Functions” on page 34-111
“Signal Processing Functions in MATLAB” on page 34-112
“Signal Processing Toolbox Functions” on page 34-112
“Special Values” on page 34-117
“Specialized Math” on page 34-117
“Statistical Functions” on page 34-118
“Statistics Toolbox Functions” on page 34-118
“String Functions” on page 34-124
“Structure Functions” on page 34-125
“Trigonometric Functions” on page 34-125

Aerospace Toolbox Functions

Function	Description
quatconj	Calculate conjugate of quaternion
quatdivide	Divide quaternion by another quaternion
quatinv	Calculate inverse of quaternion
quatmod	Calculate modulus of quaternion
quatmultiply	Calculate product of two quaternions
quatnorm	Calculate norm of quaternion
quatnormalize	Normalize quaternion

Arithmetic Operator Functions

See “Array vs. Matrix Operations” for detailed descriptions of the following operator equivalent functions.

Function	Description
<code>ctranspose</code>	Complex conjugate transpose (')
<code>idivide</code>	Integer division with rounding option
<code>isa</code>	Determine if input is object of given class
<code>ldivide</code>	Left array divide
<code>minus</code>	Minus (-)
<code>mldivide</code>	Left matrix divide (\)
<code>mpower</code>	Equivalent of array power operator (.^)
<code>mrdivide</code>	Right matrix divide
<code>mtimes</code>	Matrix multiply (*)
<code>plus</code>	Plus (+)
<code>power</code>	Array power
<code>rdivide</code>	Right array divide
<code>times</code>	Array multiply
<code>transpose</code>	Matrix transpose (')
<code>uminus</code>	Unary minus (-)
<code>uplus</code>	Unary plus (+)

Bit-Wise Operation Functions

Function	Description
<code>flintmax</code>	Largest consecutive integer in floating-point format
<code>swapbytes</code>	Swap byte ordering

Casting Functions

Data Type	Description
cast	Cast variable to different data type
char	Create character array (string)
class	Query class of object argument
double	Convert to double-precision floating point
int8, int16, int32, int64	Convert to signed integer data type
logical	Convert to Boolean true or false data type
single	Convert to single-precision floating point
typecast	Convert data types without changing underlying data
uint8, uint16, uint32, uint64	Convert to unsigned integer data type

Communications System Toolbox Functions

Function	Remarks/Limitations
bi2de	—
de2bi	—
istrellis	—
poly2trellis	—

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components
conj	Return the conjugate of a complex number
imag	Return the imaginary part of a complex number
isnumeric	Return true for numeric arrays

Function	Description
isreal	Return false (0) for a complex number
isscalar	Return true if array is a scalar
real	Return the real part of a complex number
unwrap	Correct phase angles to produce smoother phase plots

Computer Vision System Toolbox Functions

Function	Description
binaryFeatures	Object for storing binary feature vectors
cornerPoints	Object for storing corner points
detectFASTFeatures	Find corners using FAST algorithm
detectMSERFeatures	Detect MSER features
detectSURFFeatures	Detect SURF features
disparity	Disparity map between stereo images

Function	Description
<code>epipolarLine</code>	Compute epipolar lines for stereo images
<code>estimateFundamentalMatrix</code>	Estimate fundamental matrix from corresponding points in stereo image
<code>estimateUncalibratedRectification</code>	Uncalibrated stereo rectification
<code>extractFeatures</code>	Extract interest point descriptors
<code>insertMarker</code>	Insert markers in image or video
<code>insertShape</code>	Insert shapes in image or video
<code>integralImage</code>	Compute integral image
<code>isEpipoleInImage</code>	Determine whether image contains epipole
<code>lineToBorderPoints</code>	Intersection points of lines in image and image border

Function	Description
matchFeatures	Find matching image features
MSERRegions	Object for storing MSER regions
SURFPoints	Object for storing SURF interest points
vision.KalmanFilter	Kalman filter for object tracking

Data and File Management Functions

Function	Description
computer	Information about computer on which MATLAB software is running
fclose	Close one or all open files
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
load	Load data from MAT-file into workspace

Data Type Functions

Function	Description
deal	Distribute inputs to outputs
iscell	Determine whether input is cell array

Function	Description
nargchk	Validate number of input arguments Note nargchk will be removed in a future release.
narginchk	Validate number of input arguments
nargoutchk	Validate number of output arguments
str2func	Construct function handle from function name string
structfun	Apply function to each field of scalar structure

Derivative and Integral Functions

Function	Description
cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives
gradient	Numerical gradient
trapz	Trapezoidal numerical integration

Discrete Math Functions

Function	Description
factor	Return a row vector containing the prime factors of n
gcd	Return an array containing the greatest common divisors of the corresponding elements of integer arrays
isprime	Array elements that are prime numbers
lcm	Least common multiple of corresponding elements in arrays
nchoosek	Binomial coefficient or all combinations
primes	Generate list of prime numbers

Error Handling Functions

Function	Description
assert	Generate error when condition is violated
error	Display message and abort function

Exponential Functions

Function	Description
exp	Exponential
expm	Matrix exponential
expm1	Compute $\exp(x) - 1$ accurately for small values of x
factorial	Factorial function
log	Natural logarithm
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
log10	Common (base 10) logarithm
log1p	Compute $\log(1+x)$ accurately for small values of x
nextpow2	Next higher power of 2
nthroot	Real n th root of real numbers
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	N/A
all	N/A
any	N/A
atan2	N/A
bitand	Not supported for slope-bias scaled fi objects.
bitandreduce	N/A
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled fi objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	Generated code may not handle out of range shifting.
bitsra	Generated code may not handle out of range shifting.
bitsrl	Generated code may not handle out of range shifting.
bitxor	Not supported for slope-bias scaled fi objects.
bitxorreduce	N/A
ceil	N/A

Function	Remarks/Limitations
complex	N/A
conj	N/A
conv	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordicexp	Variable-size signals are not supported.
cordiccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
cos	N/A
ctranspose	N/A
diag	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.

Function	Remarks/Limitations
divide	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
double	N/A
end	N/A
eps	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> The default constructor syntax without any input arguments is not supported. If the <code>numerictype</code> is not fully specified, the input to <code>fi</code> must be a constant, a <code>fi</code>, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows <code>fi</code> to autoscale its fraction length based on the known data type of the input. <code>numerictype</code> object information must be available for nonfixed-point Simulink inputs.
filter	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
fimath	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer. Use to create <code>fimath</code> objects in the generated code.
fix	N/A
fixed.Quantizer	N/A
floor	N/A

Function	Remarks/Limitations
ge	Not supported for fixed-point signals with different biases.
get	The syntax <code>structure = get(o)</code> is not supported.
getlsb	N/A
getmsb	N/A
gt	Not supported for fixed-point signals with different biases.
hdlram	N/A
horzcat	N/A
imag	
int8, int16, int32, int64	You cannot use int64 in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow chart.
iscolumn	N/A
isempty	N/A
isequal	N/A
isfi	N/A
isfimath	N/A
isfimathlocal	N/A
isfinite	N/A
isinf	N/A
isnan	N/A
isnumeric	N/A
isnumericitype	N/A
isreal	N/A
isrow	N/A
isscalar	N/A
assigned	N/A
isvector	N/A
le	Not supported for fixed-point signals with different biases.

Function	Remarks/Limitations
length	N/A
logical	N/A
lowerbound	N/A
lsb	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	Not supported for fixed-point signals with different biases.
max	N/A
mean	N/A
median	N/A
min	N/A
minus	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mpower	<ul style="list-style-type: none"> • When the exponent <code>k</code> is a variable and the input is a scalar, the <code>ProductMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code>. • When the exponent <code>k</code> is a variable and the input is not scalar, the <code>SumMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code>. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>SpecifyPrecision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when the first input, <code>a</code>, is nonscalar. However, when <code>a</code> is a scalar,

Function	Remarks/Limitations
	MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code> .
mpy	When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code> .
mrdivide	N/A
mtimes	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>SpecifyPrecision</code> or <code>KeepLSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	<code>numberofelements</code> will be removed in a future release. Use <code>numel</code> instead.
numel	N/A

Function	Remarks/Limitations
<code>numericType</code>	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numericType</code> object that is populated with the signal's data type and scaling information. Returns the data type when the input is a nonfixed-point signal. Use to create <code>numericType</code> objects in generated code.
<code>permute</code>	N/A
<code>plus</code>	Any non- <code>fi</code> inputs must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
<code>pow2</code>	N/A
<code>power</code>	When the exponent <code>k</code> is a variable, the <code>ProductMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code> .
<code>quantize</code>	N/A
<code>range</code>	N/A
<code>rdivide</code>	N/A
<code>real</code>	N/A
<code>realmax</code>	N/A
<code>realmin</code>	N/A
<code>reinterpretcast</code>	N/A
<code>removefimath</code>	N/A
<code>repmat</code>	N/A
<code>rescale</code>	N/A
<code>reshape</code>	N/A
<code>round</code>	N/A
<code>setfimath</code>	N/A
<code>sfi</code>	N/A
<code>sign</code>	N/A
<code>sin</code>	N/A

Function	Remarks/Limitations
single	N/A
size	N/A
sort	N/A
sqrt	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
times	<ul style="list-style-type: none"> • Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32, uint64	You cannot use uint64 in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow chart.

Function	Remarks/Limitations
uminus	N/A
uplus	N/A
upperbound	N/A
vertcat	N/A

Histogram Functions

Function	Description
hist	Non-graphical histogram
histc	Histogram count

Image Processing Toolbox Functions

You must have the MATLAB Coder software installed to generate C/C++ code from MATLAB for these functions.

Function	Remarks/Limitations
bwlookup	For best results, specify an input image of class <code>logical</code> .
bwmorph	The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code> .
conndef	All input arguments must be compile-time constants.
fspecial	All inputs must be compile-time constants. Expressions or variables are allowed if their values do not change.
imcomplement	Does not support <code>int64</code> and <code>uint64</code> data types.

Function	Remarks/Limitations
imfill	<p>The optional input connectivity, <code>conn</code> and the string 'holes' must be a compile time constants.</p> <p>Supports only up to 3-D inputs. (No N-D support.)</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code> is not supported in code generation.</p> <p><code>locations</code> can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run-time. However, the number of points in <code>locations</code> can be varied at run-time.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmax	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmin	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imreconstruct	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imregionalmax	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Function	Remarks/Limitations
<code>imregionalmin</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
<code>iptcheckconn</code>	All input arguments must be compile-time constants.
<code>label2rgb</code>	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. • <code>map</code> must be an <code>n-by-3, double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. • If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning. • If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
<code>padarray</code>	<p>Support only up to 3-D inputs.</p> <p>Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.</p>

Input and Output Functions

Function	Description
<code>nargin</code>	Return the number of input arguments a user has supplied
<code>nargout</code>	Return the number of output return values a user has requested

Interpolation and Computational Geometry Functions

Function	Description
cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
meshgrid	Generate X and Y arrays for 3-D plots
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Linear Algebra

Function	Description
linsolve	Solve linear system of equations
null	Null space
orth	Range space of matrix
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root

Logical Operator Functions

Function	Description
and	Logical AND (&&)
bitand	Bitwise AND
bitcmp	Bitwise complement

Function	Description
bitget	Bit at specified position
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
not	Logical NOT (~)
or	Logical OR ()
xor	Logical exclusive-OR

MATLAB Compiler Functions

Function	Description
isdeployed	Determine whether code is running in deployed or MATLAB mode
ismcc	Test if code is running during compilation process (using mcc)

MATLAB Desktop Environment Functions

Function	Description
ismac	Determine if version is for Mac OS X platform
ispc	Determine if version is for Windows (PC) platform
isunix	Determine if version is for UNIX platform

Matrix and Array Functions

Function	Description
abs	Return absolute value and complex magnitude of an array
all	Test if all elements are nonzero

Function	Description
angle	Phase angle
any	Test for any nonzero elements
blkdiag	Construct block diagonal matrix from input arguments
bsxfun	Applies element-by-element binary operation to two arrays with singleton expansion enabled
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly
compan	Companion matrix
cond	Condition number of a matrix with respect to inversion
cov	Covariance matrix
cross	Vector cross product
cumprod	Cumulative product of array elements
cumsum	Cumulative sum of array elements
det	Matrix determinant
diag	Return a matrix formed around the specified diagonal vector and the specified diagonal (0, 1, 2,...) it occupies
diff	Differences and approximate derivatives
dot	Vector dot product
eig	Eigenvalues and eigenvectors
eye	Identity matrix
false	Return an array of 0s for the specified dimensions
find	Find indices and values of nonzero elements
flipdim	Flip array along specified dimension
flipplr	Flip matrix left to right
flipud	Flip matrix up to down
full	Convert sparse matrix to full matrix
hadamard	Hadamard matrix

Function	Description
hankel	Hankel matrix
hilb	Hilbert matrix
ind2sub	Subscripts from linear index
inv	Inverse of a square matrix
invhilb	Inverse of Hilbert matrix
ipermute	Inverse permute dimensions of array
iscolumn	True if input is a column vector
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequaln	Test arrays for equality, treating NaNs as equal
isfinite	Detect finite elements of an array
isfloat	Determine if input is floating-point array
isinf	Detect infinite elements of an array
isinteger	Determine if input is integer array
islogical	Determine if input is logical array
ismatrix	True if input is a matrix
isnan	Detect NaN elements of an array
isrow	True if input is a row vector
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
kron	Kronecker tensor product
length	Return the length of a matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square

Function	Description
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndgrid	Generate arrays for N-D functions and interpolation
ndims	Number of dimensions
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
norm	Vector and matrix norms
normest	2-norm estimate
numel	Number of elements in array or subscripted array
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed random numbers
randperm	Random permutation
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array
reshape	Reshape one array into the dimensions of another
rng	Control random number generation
rosser	Classic symmetric eigenvalue test problem

Function	Description
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Nonlinear Numerical Methods

Function	Description
fzero	Find root of continuous function of one variable
quad2d	Numerically evaluate double integral over planar region
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Phased Array System Toolbox Functions

Function	Description
aictest	Dimension of signal subspace
albersheim	Required SNR using Albersheim's equation
ambgfun	Ambiguity function
aperture2gain	Convert effective aperture to gain
az2broadside	Convert azimuth angle to broadside angle
azel2phitheta	Convert angles from azimuth/elevation form to phi/theta form
azel2phithetapat	Convert radiation pattern from azimuth/elevation to phi/theta form
azel2uv	Convert azimuth/elevation angles to u/v coordinates
azel2uvpat	Convert radiation pattern from azimuth/elevation form to u/v form
azelaxes	Spherical basis vectors in 3-by-3 matrix form
beat2range	Convert beat frequency to range
billingsleyicm	Billingsley's intrinsic clutter motion (ICM) model
broadside2az	Convert broadside angle to azimuth angle
cart2sphvec	Convert vector from Cartesian components to spherical representation
cbfweights	Conventional beamformer weights
circpol2pol	Convert circular component representation of field to linear component representation
dechirp	Perform dechirp operation on FMCW signal
delayseq	Delay or advance sequence
depressionang	Depression angle of surface target
dop2speed	Convert Doppler shift to speed
dopsteeringvec	Doppler steering vector
effearthradius	Effective earth radius
espritdoa	Direction of arrival using TLS ESPRIT
fsp1	Free space path loss
gain2aperture	Convert gain to effective aperture

Function	Description
global2localcoord	Convert global to local coordinates
grazingang	Grazing angle of surface target
horizonrange	Horizon range
lcmvweights	Narrowband linearly constrained minimum variance (LCMV) beamformer weights
local2globalcoord	Convert local to global coordinates
mdltest	Dimension of signal subspace
mvdrweights	Minimum variance distortionless response (MVDR) beamformer weights
noisepow	Receiver noise power
npwgnthresh	Detection SNR threshold for signal in white Gaussian noise
phitheta2azel	Convert angles from phi/theta form to azimuth/elevation form
phitheta2azelpat	Convert radiation pattern from phi/theta form to azimuth/elevation form
phitheta2uv	Convert phi/theta angles to u/v coordinates
phitheta2uvpat	Convert radiation pattern from phi/theta form to u/v form
physconst	Physical constants
pol2circpol	Convert linear component representation of field to circular component representation
polellip	Parameters of ellipse traced out by tip of a polarized field vector
polloss	Polarization loss
polratio	Ratio of vertical to horizontal linear polarization components of a field
polsignature	Copolarization and cross-polarization signatures
pulsint	Pulse integration
radareqpow	Peak power estimate from radar equation
radareqrng	Maximum theoretical range estimate
radareqsnr	SNR estimate from radar equation

Function	Description
radarvcd	Vertical coverage diagram
radialspeed	Relative radial speed
range2beat	Convert range to beat frequency
range2bw	Convert range resolution to required bandwidth
range2time	Convert propagation distance to propagation time
rangeangle	Range and angle calculation
rdcoupling	Range Doppler coupling
rocpfa	Receiver operating characteristic curves by false-alarm probability
rocsnr	Receiver operating characteristic curves by SNR
rootmusicdoa	Direction of arrival using Root MUSIC
rotx	Rotation matrix for rotations around x-axis
roty	Rotation matrix for rotations around y-axis
rotz	Rotation matrix for rotations around z-axis
sensorcov	Sensor spatial covariance matrix
sensorsig	Simulate received signal at sensor array
shnidman	Required SNR using Shnidman's equation
speed2dop	Convert speed to Doppler shift
sph2cartvec	Convert vector from spherical basis components to Cartesian components
spsmooth	Spatial smoothing
steervec	Steering vector
stokes	Stokes parameters of polarized field
stretchfreq2rng	Convert frequency offset to range
surfacegamma	Gamma value for different terrains
surfclutterrcs	Surface clutter radar cross section (RCS)
systemp	Receiver system-noise temperature
time2range	Convert propagation time to propagation distance

Function	Description
unigrid	Uniform grid
uv2azel	Convert u/v coordinates to azimuth/elevation angles
uv2azelpat	Convert radiation pattern from u/v form to azimuth/elevation form
uv2phitheta	Convert u/v coordinates to phi/theta angles
uv2phithetapat	Convert radiation pattern from u/v form to phi/theta form
val2ind	Uniform grid index

Polynomial Functions

Function	Description
poly	Polynomial with specified roots
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation
roots	Polynomial roots

Programming Utilities

Function	Description
mfilename	File name of currently running function

Relational Operator Functions

Function	Description
eq	Equal (==)
ge	Greater than or equal to (>=)
gt	Greater than (>)
le	Less than or equal to (<=)

Function	Description
lt	Less than (<)
ne	Not equal (~=)

Rounding and Remainder Functions

Function	Description
ceil	Round toward plus infinity
ceil	Round toward positive infinity
convergent	Round toward nearest integer with ties rounding to nearest even integer
fix	Round toward zero
fix	Round toward zero
floor	Round toward minus infinity
floor	Round toward negative infinity
mod	Modulus (signed remainder after division)
nearest	Round toward nearest integer with ties rounding toward positive infinity
rem	Remainder after division
round	Round toward nearest integer
round	Round <code>fi</code> object toward nearest integer or round input data using quantizer object

Set Functions

Function	Description
intersect	Set intersection of two arrays
ismember	Array elements that are members of set array
issorted	Determine whether set elements are in sorted order
setdiff	Set difference of two arrays

Function	Description
setxor	Set exclusive OR of two arrays
union	Set union of two arrays
unique	Unique values in array

Signal Processing Functions in MATLAB

Function	Description
chol	Cholesky factorization
conv	Convolution and polynomial multiplication
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
filter	Filter a data sequence using a digital filter that works for both real and complex inputs
freqspace	Frequency spacing for frequency response
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse discrete Fourier transform shift
svd	Singular value decomposition
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

Signal Processing Toolbox Functions

These functions do not support variable-size inputs, you must define the size and type of the function inputs. For more information, see “Specifying Inputs in Code Generation from MATLAB”.

Note Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for codegen, use `coder.Constant`.

Function	Remarks/Limitations
barthannwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bartlett	Window length must be a constant. Expressions or variables are allowed if their values do not change.
besselap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
bitrevorder	—
blackman	Window length must be a constant. Expressions or variables are allowed if their values do not change.
blackmanharris	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bohmanwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
buttap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
butter	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
buttord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cfirpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
dct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
<code>firls</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>firpm</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>firpmord</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>firrcos</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>flattopwin</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>freqz</code>	<code>freqz</code> with no output arguments produces a plot only when the function call terminates in a semicolon. See “ <code>freqz</code> With No Output Arguments”.
<code>gaussfir</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>gausswin</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>hamming</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>hann</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>idct</code>	Length of transform dimension must be a power of two. If specified, the <code>pad</code> or <code>truncation</code> value must be constant. Expressions or variables are allowed if their values do not change.
<code>intfilt</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>kaiser</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>kaiserord</code>	—
<code>levinson</code>	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
maxflat	All inputs must be constant. Expressions or variables are allowed if their values do not change.
nutallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	<ul style="list-style-type: none"> • Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. • Variable-size inputs are not supported.
upsample	Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code> . For example, <code>assert(n<10)</code>
xcorr	—
yulewalk	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Special Values

Symbol	Description
eps	Floating-point relative accuracy
inf	IEEE® arithmetic representation for positive infinity
intmax	Largest possible value of specified integer type
intmin	Smallest possible value of specified integer type
NaN or nan	Not a number
pi	Ratio of the circumference to the diameter for a circle
realmax	Largest positive floating-point number
realmin	Smallest positive floating-point number

Specialized Math

Symbol	Description
beta	Beta function
betainc	Incomplete beta function
betaincinv	Beta inverse cumulative distribution function
betaln	Logarithm of beta function
ellipke	Complete elliptic integrals of first and second kind
erf	Error function
erfc	Complementary error function
erfcinv	Inverse of complementary error function
erfcx	Scaled complementary error function
erfinv	Inverse error function
expint	Exponential integral
gamma	Gamma function
gammainc	Incomplete gamma function

Symbol	Description
gammaincinv	Inverse incomplete gamma function
gammaaln	Logarithm of the gamma function
psi	Psi (polygamma) function

Statistical Functions

Function	Description
corrcoef	Correlation coefficients
mean	Average or mean value of array
median	Median value of array
mode	Most frequent values in array
std	Standard deviation
var	Variance

Statistics Toolbox Functions

Function	Description
betacdf	Beta cumulative distributive function
betainv	Beta inverse cumulative distribution function
betapdf	Beta probability density function
betarnd	Beta random numbers
betastat	Beta mean and variance
binocdf	Binomial cumulative distribution function
binoinv	Binomial inverse cumulative distribution function
binopdf	Binomial probability density function
binornd	Binomial random numbers
binostat	Binomial mean and variance

Function	Description
<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>chi2cdf</code>	Chi-square cumulative distribution function
<code>chi2inv</code>	Chi-square inverse cumulative distribution function
<code>chi2pdf</code>	Chi-square probability density function
<code>chi2rnd</code>	Chi-square random numbers
<code>chi2stat</code>	Chi-square mean and variance
<code>evcdf</code>	Extreme value cumulative distribution function
<code>evinv</code>	Extreme value inverse cumulative distribution
<code>evpdf</code>	Extreme value probability density function
<code>evrnd</code>	Extreme value random numbers
<code>evstat</code>	Extreme value mean and variance
<code>expcdf</code>	Exponential cumulative distribution function
<code>expinv</code>	Exponential inverse cumulative distribution function
<code>exppdf</code>	Exponential probability density function
<code>exprnd</code>	Exponential random numbers
<code>expstat</code>	Exponential mean and variance
<code>fcdf</code>	F cumulative distribution function
<code>finv</code>	F inverse cumulative distribution function
<code>fpdf</code>	F probability density function
<code>frnd</code>	F random numbers
<code>fstat</code>	F mean and variance
<code>gamcdf</code>	Gamma cumulative distribution function
<code>gaminv</code>	Gamma inverse cumulative distribution function
<code>gampdf</code>	Gamma probability density function
<code>gamrnd</code>	Gamma random numbers
<code>gamstat</code>	Gamma mean and variance

Function	Description
geocdf	Geometric cumulative distribution function
geoinv	Geometric inverse cumulative distribution function
geomean	Geometric mean
geopdf	Geometric probability density function
geornd	Geometric random numbers
geostat	Geometric mean and variance
gevcdf	Generalized extreme value cumulative distribution function
gevinv	Generalized extreme value inverse cumulative distribution function
gevpdf	Generalized extreme value probability density function
gevrnd	Generalized extreme value random numbers
gevstat	Generalized extreme value mean and variance
gpcdf	Generalized Pareto cumulative distribution function
gpinv	Generalized Pareto inverse cumulative distribution function
gppdf	Generalized Pareto probability density function
gprnd	Generalized Pareto random numbers
gpstat	Generalized Pareto mean and variance
harmmean	Harmonic mean
hygecdf	Hypergeometric cumulative distribution function
hygeinv	Hypergeometric inverse cumulative distribution function
hygepdf	Hypergeometric probability density function
hygernd	Hypergeometric random numbers
hygestat	Hypergeometric mean and variance
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object

Function	Description
kurtosis	Kurtosis
logncdf	Lognormal cumulative distribution function
logninv	Lognormal inverse cumulative distribution function
lognpdf	Lognormal probability density function
lognrnd	Lognormal random numbers
lognstat	Lognormal mean and variance
mad	Mean or median absolute deviation
mnpdf	Multinomial probability density function
moment	Central moments
nancov	Covariance ignoring NaN values
nanmax	Maximum ignoring NaN values
nanmean	Mean ignoring NaN values
nanmedian	Median ignoring NaN values
nanmin	Minimum ignoring NaN values
nanstd	Standard deviation ignoring NaN values
nansum	Sum ignoring NaN values
nanvar	Variance, ignoring NaN values
nbincdf	Negative binomial cumulative distribution function
nbinv	Negative binomial inverse cumulative distribution function
nbnpdf	Negative binomial probability density function
nbnrnd	Negative binomial random numbers
nbinstat	Negative binomial mean and variance
ncfcdf	Noncentral F cumulative distribution function
ncfinv	Noncentral F inverse cumulative distribution function
ncfpdf	Noncentral F probability density function
ncfrnd	Noncentral F random numbers

Function	Description
ncfstat	Noncentral F mean and variance
nctcdf	Noncentral t cumulative distribution function
nctinv	Noncentral t inverse cumulative distribution function
nctpdf	Noncentral t probability density function
nctrnd	Noncentral t random numbers
nctstat	Noncentral t mean and variance
ncx2cdf	Noncentral chi-square cumulative distribution function
ncx2rnd	Noncentral chi-square random numbers
ncx2stat	Noncentral chi-square mean and variance
normcdf	Normal cumulative distribution function
norminv	Normal inverse cumulative distribution function
normpdf	Normal probability density function
normrnd	Normal random numbers
normstat	Normal mean and variance
pdf	Probability density function of probability distribution object
poisscdf	Poisson cumulative distribution function
poissinv	Poisson inverse cumulative distribution function
poisspdf	Poisson probability density function
poissrnd	Poisson random numbers
poisstat	Poisson mean and variance
prctile	Percentiles of a data set
quantile	Quantiles of a data set
randg	Gamma random numbers with unit scale
random	Random numbers
raylcdf	Rayleigh cumulative distribution function
raylinv	Rayleigh inverse cumulative distribution function

Function	Description
raylpdf	Rayleigh probability density function
raylrnd	Rayleigh random numbers
raylstat	Rayleigh mean and variance
skewness	Skewness
tcdf	Student's <i>t</i> cumulative distribution function
tinvs	Student's <i>t</i> inverse cumulative distribution function
tpdf	Student's <i>t</i> probability density function
trnd	Student's <i>t</i> random numbers
tstat	Student's <i>t</i> mean and variance
unidcdf	Discrete uniform cumulative distribution function
unidinv	Discrete uniform inverse cumulative distribution function
unidpdf	Discrete uniform probability density function
unidrnd	Discrete uniform random numbers
unidstat	Discrete uniform mean and variance
unifcdf	Continuous uniform cumulative distribution function
unifinv	Continuous uniform inverse cumulative distribution function
unifpdf	Continuous uniform probability density function
unifrnd	Continuous uniform random numbers
unifstat	Continuous uniform mean and variance
wblcdf	Weibull cumulative distribution function
wblinv	Weibull inverse cumulative distribution function
wblpdf	Weibull probability density function
wblrnd	Weibull random numbers
wblstat	Weibull mean and variance
zscore	Standardized <i>z</i> -scores

String Functions

Function	Description
bin2dec	Convert binary number string to decimal number
bitmax	Maximum double-precision floating-point integer
blanks	Create string of blank characters
char	Create character array (string)
deblank	Strip trailing blanks from end of string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
ischar	True for character array (string)
isletter	Array elements that are alphabetic letters
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
lower	Convert string to lowercase
num2hex	Convert singles and doubles to IEEE hexadecimal strings
strcmp	Compare strings (case sensitive)
strcmpi	Compare strings (case insensitive)
strfind	Find one string within another
strjust	Justify character array
strncmp	Compare first n characters of strings (case sensitive)
strncmpi	Compare first n characters of strings (case insensitive)
strep	Find and replace substring
strtok	Selected parts of string
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

Structure Functions

Function	Description
isfield	Determine whether input is structure array field
struct	Create structure
isstruct	Determine whether input is a structure

Trigonometric Functions

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atan2d	Four-quadrant inverse tangent, result in degrees
atand	Inverse tangent; result in degrees

Function	Description
atanh	Inverse hyperbolic tangent
cos	Cosine
cosd	Cosine; result in degrees
cosh	Hyperbolic cosine
cot	Cotangent; result in radians
cotd	Cotangent; result in degrees
coth	Hyperbolic cotangent
csc	Cosecant; result in radians
cscd	Cosecant; result in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant; result in radians
secd	Secant; result in degrees
sech	Hyperbolic secant
sin	Sine
sind	Sine; result in degrees
sinh	Hyperbolic sine
tan	Tangent
tand	Tangent; result in degrees
tanh	Hyperbolic tangent

System Objects Supported for Code Generation

System Objects Supported for C/C++ Code Generation

In this section...

“Code Generation for System Objects” on page 35-2

“Computer Vision System Toolbox System Objects” on page 35-2

“Communications System Toolbox System Objects” on page 35-7

“DSP System Toolbox System Objects” on page 35-13

“Phased Array System Toolbox System Objects” on page 35-19

“Image Acquisition Toolbox System Objects” on page 35-23

Code Generation for System Objects

You can generate C/C++ code for a subset of System objects provided by Communications System Toolbox, DSP System Toolbox, Computer Vision System Toolbox, and Phased Array System Toolbox. To use these System objects, you need to install the requisite toolbox.

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 on MATLAB objects, see “Begin Using Object-Oriented Programming”.

Computer Vision System Toolbox System Objects

If you install Computer Vision System Toolbox software, you can generate C/C++ code for the following Computer Vision System Toolbox System objects. For more information on how to use these System objects, see “System Objects in MATLAB Code Generation”.

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
vision.BoundaryTracer	Trace object boundaries in binary images.
vision.ContrastAdjuster	Adjust image contrast by linear scaling.
vision.Deinterlacer	Remove motion artifacts by deinterlacing input video signal.
vision.EdgeDetector	Find edges of objects in images.
vision.ForegroundDetector	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
vision.HistogramBasedTracker	Track object in video based on histogram. This object supports tunable properties in code generation.
vision.HistogramEqualizer	Enhance contrast of images using histogram equalization.
vision.TemplateMatcher	Perform template matching by shifting template over image.
Conversions	
vision.Autothresher	Convert intensity image to binary image.
vision.ChromaResampler	Downsample or upsample chrominance components of images.
vision.ColorSpaceConverter	Convert color information between color spaces.
vision.DemosaicInterpolator	Demosaic Bayer's format images.
vision.GammaCorrector	Apply or remove gamma correction from images or video streams.
vision.ImageComplementer	Compute complement of pixel values in binary, intensity, or RGB images.
vision.ImageDataTypeConverter	Convert and scale input image to specified output data type.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
Feature Detection, Extraction, and Matching	
<code>vision.CornerDetector</code>	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
<code>vision.Convolver</code>	Compute 2-D discrete convolution of two input matrices.
<code>vision.ImageFilter</code>	Perform 2-D FIR filtering of input matrix.
<code>vision.MedianFilter</code>	2D median filtering.
Geometric Transformations	
<code>vision.GeometricRotator</code>	Rotate image by specified angle.
<code>vision.GeometricScaler</code>	Enlarge or shrink image size.
<code>vision.GeometricShearer</code>	Shift rows or columns of image by linearly varying offset.
<code>vision.GeometricTransformer</code>	Apply projective or affine transformation to an image.
<code>vision.GeometricTransformEstimator</code>	Estimate geometric transformation from matching point pairs.
<code>vision.GeometricTranslator</code>	Translate image in two-dimensional plane using displacement vector.
Morphological Operations	
<code>vision.ConnectedComponentLabeler</code>	Label and count the connected regions in a binary image.
<code>vision.MorphologicalClose</code>	Perform morphological closing on image.
<code>vision.MorphologicalDilate</code>	Perform morphological dilation on an image.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.MorphologicalErode</code>	Perform morphological erosion on an image.
<code>vision.MorphologicalOpen</code>	Perform morphological opening on an image.
Object Detection	
<code>vision.CascadeObjectDetector</code>	Detect objects using the Viola-Jones algorithm.
<code>vision.HistogramBasedTracker</code>	Track object in video based on histogram. This object supports tunable properties in code generation.
<code>vision.PointTracker</code>	Track points in video using Kanade-Lucas-Tomasi (KLT) algorithm.
<code>vision.PeopleDetector</code>	Detect upright people using HOG features.
Sinks	
<code>vision.VideoPlayer</code>	Send video data to computer screen. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<code>vision.DeployableVideoPlayer</code>	Send video data to computer screen.
<code>vision.VideoFileWriter</code>	Write video frames and audio samples to multimedia file.
Sources	
<code>vision.VideoFileReader</code>	Read video frames and audio samples from compressed multimedia file.
Statistics	
<code>vision.Autocorrelator</code>	Compute 2-D autocorrelation of input matrix.
<code>vision.BlobAnalysis</code>	Compute statistics for connected regions in a binary image.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.Crosscorrelator</code>	Compute 2-D cross-correlation of two input matrices.
<code>vision.Histogram</code>	Generate histogram of each input matrix. This object has no tunable properties.
<code>vision.LocalMaximaFinder</code>	Find local maxima in matrices.
<code>vision.Maximum</code>	Find maximum values in input or sequence of inputs.
<code>vision.Mean</code>	Find mean value of input or sequence of inputs.
<code>vision.Median</code>	Find median values in an input.
<code>vision.Minimum</code>	Find minimum values in input or sequence of inputs.
<code>vision.PSNR</code>	Compute peak signal-to-noise ratio (PSNR) between images.
<code>vision.StandardDeviation</code>	Find standard deviation of input or sequence of inputs.
<code>vision.Variance</code>	Find variance values in an input or sequence of inputs.
Text & Graphics	
<code>vision.AlphaBlender</code>	Combine images, overlay images, or highlight selected pixels.
<code>vision.MarkerInserter</code>	Draw markers on output image.
<code>vision.ShapeInserter</code>	Draw rectangles, lines, polygons, or circles on images.
<code>vision.TextInserter</code>	Draw text on image or video stream.
Transforms	
<code>vision.DCT</code>	Compute 2-D discrete cosine transform.
<code>vision.FFT</code>	Two-dimensional discrete Fourier transform.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.HoughLines</code>	Find Cartesian coordinates of lines that are described by rho and theta pairs.
<code>vision.HoughTransform</code>	Find lines in images via Hough transform.
<code>vision.IDCT</code>	Compute 2-D inverse discrete cosine transform.
<code>vision.IFFT</code>	Two-dimensional inverse discrete Fourier transform.
<code>vision.Pyramid</code>	Perform Gaussian pyramid decomposition.
Utilities	
<code>vision.ImagePadder</code>	Pad or crop input image along its rows, columns, or both.

Communications System Toolbox System Objects

If you install Communications System Toolbox software, you can generate C/C++ code for the following Communications System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects”.

Supported Communications System Toolbox System Objects

Object	Description
Source Coding	
<code>comm.DifferentialDecoder</code>	Decode binary signal using differential decoding
<code>comm.DifferentialEncoder</code>	Encode binary signal using differential coding
Channels	
<code>comm.AWGNChannel</code>	Add white Gaussian noise to input signal
<code>comm.LTEMIMOChannel</code>	Filter input signal through LTE MIMO multipath fading channel

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.MIMOChannel	Filter input signal through MIMO multipath fading channel
comm.BinarySymmetricChannel	Introduce binary errors
Equalizers	
comm.MLSEEqualizer	Equalize using maximum likelihood sequence estimation
Filters	
comm.IntegrateAndDumpFilter	Integrate discrete-time signal with periodic resets
Measurements	
comm.ACPR	Measure adjacent channel power ratio
comm.CCDF	Measure complementary cumulative distribution function
comm.EVM	Measure error vector magnitude
comm.MER	Measure modulation error ratio
Sources	
comm.BarkerCode	Generate Barker code
comm.HadamardCode	Generate Hadamard code
comm.KasamiSequence	Generate a Kasami sequence
comm.OVSFCode	Generate OVSF code
comm.PNSequence	Generate a pseudo-noise (PN) sequence
comm.WalshCode	Generate Walsh code from orthogonal set of codes
Error Detection and Correction – Block Coding	
comm.BCHDecoder	Decode data using BCH decoder
comm.BCHEncoder	Encode data using BCH encoder
comm.LDPCDecoder	Decode binary low-density parity-check code
comm.LDPCEncoder	Encode binary low-density parity-check code

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.RSDecoder	Decode data using Reed-Solomon decoder
comm.RSEncoder	Encode data using Reed-Solomon encoder
Error Detection and Correction – Convolutional Coding	
comm.ConvolutionalEncoder	Convolutionally encode binary data
comm.ViterbiDecoder	Decode convolutionally encoded data using Viterbi algorithm
Error Detection and Correction – Cyclic Redundancy Check Coding	
comm.CRCDetector	Detect errors in input data using cyclic redundancy code
comm.CRCGenerator	Generate cyclic redundancy code bits and append to input data
comm.HDLCRCGenerator	Generate CRC code bits and append to input data, optimized for HDL code generation
comm.TurboDecoder	Decode input signal using parallel concatenated decoding scheme
comm.TurboEncoder	Encode input signal using parallel concatenated encoding scheme
Interleavers – Block	
comm.AlgebraicDeinterleaver	Deinterleave input symbols using algebraically derived permutation vector
comm.AlgebraicInterleaver	Permute input symbols using an algebraically derived permutation vector
comm.BlockDeinterleaver	Deinterleave input symbols using permutation vector
comm.BlockInterleaver	Permute input symbols using a permutation vector
comm.MatrixDeinterleaver	Deinterleave input symbols using permutation matrix
comm.MatrixInterleaver	Permute input symbols using permutation matrix

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.MatrixHelicalScanDeinterleaver	Deinterleave input symbols by filling a matrix along diagonals
comm.MatrixHelicalScanInterleaver	Permute input symbols by selecting matrix elements along diagonals
Interleavers – Convolutional	
comm.ConvolutionalDeinterleaver	Restore ordering of symbols using shift registers
comm.ConvolutionalInterleaver	Permute input symbols using shift registers
comm.HelicalDeinterleaver	Restore ordering of symbols using a helical array
comm.HelicalInterleaver	Permute input symbols using a helical array
comm.MultiplexedDeinterleaver	Restore ordering of symbols using a set of shift registers with specified delays
comm.MultiplexedInterleaver	Permute input symbols using a set of shift registers with specified delays
MIMO	
comm.OSTBCCombiner	Combine inputs using orthogonal space-time block code
comm.OSTBCEncoder	Encode input message using orthogonal space-time block code
Digital Baseband Modulation – Phase	
comm.BPSKDemodulator	Demodulate using binary PSK method
comm.BPSKModulator	Modulate using binary PSK method
comm.DBPSKModulator	Modulate using differential binary PSK method
comm.DPSKDemodulator	Demodulate using M-ary DPSK method
comm.DPSKModulator	Modulate using M-ary DPSK method
comm.DQPSKDemodulator	Demodulate using differential quadrature PSK method
comm.DQPSKModulator	Modulate using differential quadrature PSK method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.DBPSKDemodulator	Demodulate using M-ary DPSK method
comm.QPSKDemodulator	Demodulate using quadrature PSK method
comm.QPSKModulator	Modulate using quadrature PSK method
comm.PSKDemodulator	Demodulate using M-ary PSK method
comm.PSKModulator	Modulate using M-ary PSK method
comm.OQPSKDemodulator	Demodulate offset quadrature PSK modulated data
comm.OQPSKModulator	Modulate using offset quadrature PSK method
Digital Baseband Modulation – Amplitude	
comm.GeneralQAMDemodulator	Demodulate using arbitrary QAM constellation. This object has no tunable properties in code generation.
comm.GeneralQAMModulator	Modulate using arbitrary QAM constellation
comm.PAMDemodulator	Demodulate using M-ary PAM method
comm.PAMModulator	Modulate using M-ary PAM method
comm.RectangularQAMDemodulator	Demodulate using rectangular QAM method
comm.RectangularQAMModulator	Modulate using rectangular QAM method
Digital Baseband Modulation – Frequency	
comm.FSKDemodulator	Demodulate using M-ary FSK method
comm.FSKModulator	Modulate using M-ary FSK method
Digital Baseband Modulation – Trellis Coded	
comm.GeneralQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation
comm.GeneralQAMTCMModulator	Convolutionally encode binary data and map using arbitrary QAM constellation
comm.PSKTCMDemodulator	Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.PSKTCMModulator	Convolutionally encode binary data and map using M-ary PSK constellation
comm.RectangularQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to rectangular QAM constellation
comm.RectangularQAMTCMModulator	Convolutionally encode binary data and map using rectangular QAM constellation
Digital Baseband Modulation – Continuous Phase	
comm.CPFSKDemodulator	Demodulate using CPFSK method and Viterbi algorithm
comm.CPFSKModulator	Modulate using CPFSK method
comm.CPMDemodulator	Demodulate using CPM method and Viterbi algorithm
comm.CPModulator	Modulate using CPM method
comm.GMSKDemodulator	Demodulate using GMSK method and the Viterbi algorithm
comm.GMSKModulator	Modulate using GMSK method
comm.MSKDemodulator	Demodulate using MSK method and the Viterbi algorithm
comm.MSKModulator	Modulate using MSK method
RF Impairments	
comm.MemorylessNonlinearity	Apply memoryless nonlinearity to input signal
comm.PhaseFrequencyOffset	Apply phase and frequency offsets to input signal. The PhaseOffset property of this object is not tunable in code generation.
comm.PhaseNoise	Apply phase noise to complex baseband signal
comm.ThermalNoise	Add receiver thermal noise
Synchronization – Timing Phase	

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.EarlyLateGateTimingSynchronizer</code>	Recover symbol timing phase using early-late gate method
<code>comm.GardnerTimingSynchronizer</code>	Recover symbol timing phase using Gardner's method
<code>comm.GMSKTimingSynchronizer</code>	Recover symbol timing phase using fourth-order nonlinearity method
<code>comm.MSKTimingSynchronizer</code>	Recover symbol timing phase using fourth-order nonlinearity method
<code>comm.MuellerMullerTimingSynchronizer</code>	Recover symbol timing phase using Mueller-Muller method
Synchronization Utilities	
<code>comm.CPMCarrierPhaseSynchronizer</code>	Recover carrier phase of baseband CPM signal
<code>comm.DiscreteTimeVCO</code>	Generate variable frequency sinusoid
Converters	
<code>comm.BitToInteger</code>	Convert vector of bits to vector of integers
<code>comm.IntegerToBit</code>	Convert vector of integers to vector of bits
Sequence Operators	
<code>comm.Descrambler</code>	Descramble input signal
<code>comm.GoldSequence</code>	Generate Gold sequence
<code>comm.Scrambler</code>	Scramble input signal

DSP System Toolbox System Objects

If you install DSP System Toolbox software, you can generate C/C++ code for the following DSP System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects”.

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
dsp.BurgAREstimator	Compute estimate of autoregressive model parameters using Burg method
dsp.BurgSpectrumEstimator	Compute parametric spectral estimate using Burg method
dsp.CepstralToLPC	Convert cepstral coefficients to linear prediction coefficients
dsp.LevinsonSolver	Solve linear system of equations using Levinson-Durbin recursion
dsp.LPCToAutocorrelation	Convert linear prediction coefficients to autocorrelation coefficients
dsp.LPCToCepstral	Convert linear prediction coefficients to cepstral coefficients
dsp.LPCToLSF	Convert linear prediction coefficients to line spectral frequencies
dsp.LPCToLSP	Convert linear prediction coefficients to line spectral pairs
dsp.LPCToRC	Convert linear prediction coefficients to reflection coefficients
dsp.LSFToLPC	Convert line spectral frequencies to linear prediction coefficients
dsp.LSPToLPC	Convert line spectral pairs to linear prediction coefficients
dsp.RCToAutocorrelation	Convert reflection coefficients to autocorrelation coefficients
dsp.RCToLPC	Convert reflection coefficients to linear prediction coefficients
Filters	
dsp.AffineProjectionFilter	Adaptive filter using the Affine Projection algorithm

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.AllpoleFilter</code>	IIR Filter with no zeros. Only the Denominator property is tunable for code generation.
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters
<code>dsp.CICDecimator</code>	Decimate input using Cascaded Integrator-Comb filter
<code>dsp.CICInterpolator</code>	Interpolate signal using Cascaded Integrator-Comb filter
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations. The <code>SOSMatrix</code> and <code>ScaleValues</code> properties are not supported for code generation.
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRFilter</code>	Static or time-varying FIR filter. Only the Numerator property is tunable for code generation.
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.IIRFilter</code>	Infinite Impulse Response (IIR) filter. Only the Numerator and Denominator properties are tunable for code generation.
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
<code>dsp.RLSFilter</code>	Adaptive filter using the Recursive Least Squares (RLS) algorithm
Math Operations	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements
<code>dsp.LDLFactor</code>	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
<code>dsp.LowerTriangularSolver</code>	Solve $LX = B$ for X when L is lower triangular matrix
<code>dsp.LUFactor</code>	Factor square matrix into lower and upper triangular matrices
<code>dsp.Normalizer</code>	Normalize input
<code>dsp.UpperTriangularSolver</code>	Solve $UX = B$ for X when U is upper triangular matrix
Quantizers	
<code>dsp.ScalarQuantizerDecoder</code>	Convert each index value into quantized output value
<code>dsp.ScalarQuantizerEncoder</code>	Perform scalar quantization encoding
<code>dsp.VectorQuantizerDecoder</code>	Find vector quantizer codeword for given index value
<code>dsp.VectorQuantizerEncoder</code>	Perform vector quantization encoding
Scopes	
<code>dsp.SpectrumAnalyzer</code>	Display frequency spectrum of time-domain signals. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<code>dsp.TimeScope</code>	Display time-domain signals. This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
Signal Management	

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.Counter	Count up or down through specified range of numbers
dsp.DelayLine	Rebuffer sequence of inputs with one-sample shift
Signal Operations	
dsp.Convolver	Compute convolution of two inputs
dsp.Delay	Delay input by specified number of samples or frames
dsp.Interpolator	Interpolate values of real input samples
dsp.NCO	Generate real or complex sinusoidal signals
dsp.PeakFinder	Determine extrema (maxima or minima) in input signal
dsp.PhaseUnwrapper	Unwrap signal phase
dsp.VariableFractionalDelay	Delay input by time-varying fractional number of sample periods
dsp.VariableIntegerDelay	Delay input by time-varying integer number of sample periods
dsp.Window	Generate or apply window function. This object has no tunable properties for code generation.
dsp.ZeroCrossingDetector	Calculate number of zero crossings of a signal
Sinks	
dsp.AudioPlayer	Write audio data to computer's audio device
dsp.AudioFileWriter	Write audio file
dsp.UDPsender	Send UDP packets to the network
Sources	
dsp.AudioFileReader	Read audio samples from an audio file
dsp.AudioRecorder	Read audio data from computer's audio device
dsp.SignalSource	Import variable from workspace

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.SineWave	Generate discrete sine wave. This object has no tunable properties for code generation.
dsp.UDPReceiver	Receive UDP packets from the network
Statistics	
dsp.Autocorrelator	Compute autocorrelation of vector inputs
dsp.Crosscorrelator	Compute cross-correlation of two inputs
dsp.Histogram	Output histogram of an input or sequence of inputs. This object has no tunable properties for code generation.
dsp.Maximum	Compute maximum value in input
dsp.Mean	Compute average or mean value in input
dsp.Median	Compute median value in input
dsp.Minimum	Compute minimum value in input
dsp.RMS	Compute root-mean-square of vector elements
dsp.StandardDeviation	Compute standard deviation of vector elements
dsp.Variance	Compute variance of input or sequence of inputs
Transforms	
dsp.AnalyticSignal	Compute analytic signals of discrete-time inputs
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input
dsp.IDCT	Compute inverse discrete cosine transform (IDCT) of input
dsp.IFFT	Compute inverse fast Fourier transform (IFFT) of input

Phased Array System Toolbox System Objects

If you install Phased Array System Toolbox software, you can generate C/C++ code for the following Phased Array System Toolbox System objects. For more information on how to use these System objects, see “About Code Generation”.

Supported Phased Array System Toolbox System Objects

Object	Description
Antenna and Microphone Elements	
phased.CosineAntennaElement	Cosine antenna element
phased.CrossedDipoleAntennaElement	Crossed-dipole antenna element
phased.CustomAntennaElement	Custom antenna element
phased.CustomMicrophoneElement	Custom microphone
phased.IsotropicAntennaElement	Isotropic antenna element
phased.OmnidirectionalMicrophoneElement	Omnidirectional microphone
phased.ShortDipoleAntennaElement	Short-dipole antenna element
Array Geometries and Analysis	
phased.ULA	Uniform linear array
phased.URA	Uniform rectangular array
phased.ConformalArray	Conformal array
phased.PartitionedArray	Phased array partitioned into subarrays
phased.ReplicatedSubarray	Phased array formed by replicated subarrays
phased.SteeringVector	Sensor array steering vector
phased.ArrayGain	Sensor array gain
phased.ArrayResponse	Sensor array response
phased.ElementDelay	Sensor array element delay estimator
Signal Radiation and Collection	
phased.Collector	Narrowband signal collector
phased.Radiator	Narrowband signal radiator

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
phased.WidebandCollector	Wideband signal collector Note Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.
Waveforms, Transmitter, and Receiver	
phased.LinearFMWaveform	Linear FM pulse waveform
phased.PhaseCodedWaveform	Phase-coded pulse waveform
phased.RectangularWaveform	Rectangular pulse waveform
phased.SteppedFMWaveform	Stepped FM pulse waveform
phased.FMCWWaveform	FMCW Waveform
phased.MatchedFilter	Matched filter
phased.Transmitter	Transmitter
phased.ReceiverPreamp	Receiver preamp
Beamforming	
phased.PhaseShiftBeamformer	Narrowband phase shift beamformer
phased.LCMVBeamformer	Narrowband LCMV beamformer
phased.MVDRBeamformer	Narrowband MVDR (Capon) beamformer
phased.SubbandPhaseShiftBeamformer	Subband phase shift beamformer
phased.FrostBeamformer	Frost beamformer Note Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
<code>phased.TimeDelayBeamformer</code>	Time delay beamformer Note Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.
<code>phased.TimeDelayLCMVBeamformer</code>	Time delay LCMV beamformer Note Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.
<code>phased.SteeringVector</code>	Sensor array steering vector
Direction of Arrival (DOA) Estimation	
<code>phased.SumDifferenceMonopulseTracker</code>	Sum and difference monopulse for ULA
<code>phased.SumDifferenceMonopulseTracker2D</code>	Sum and difference monopulse for URA
<code>phased.BeamspaceEstimator</code>	Beamspace spatial spectrum estimator for ULA
<code>phased.BeamspaceEstimator2D</code>	2-D beamspace spatial spectrum estimator
<code>phased.MVDREstimator</code>	MVDR (Capon) spatial spectrum estimator for ULA
<code>phased.MVDREstimator2D</code>	2-D MVDR (Capon) spatial spectrum estimator
<code>phased.RootMUSICEstimator</code>	Root MUSIC direction of arrival (DOA) estimator
<code>phased.RootWSFEstimator</code>	Root WSF direction of arrival (DOA) estimator
<code>phased.ESPRITEstimator</code>	ESPRIT direction of arrival (DOA) estimator
<code>phased.BeamspaceESPRITEstimator</code>	Beamspace ESPRIT direction of arrival (DOA) estimator

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
Space-Time Adaptive Processing (STAP)	
phased.STAPSMIBeamformer	Sample matrix inversion (SMI) beamformer
phased.DPCACanceller	Displaced phase center array (DPCA) pulse canceller
phased.ADPCACanceller	Adaptive DPCA (ADPCA) pulse canceller
phased.AngleDopplerResponse	Angle-Doppler response
Detection	
phased.CFARDetector	Constant false alarm rate (CFAR) detector
phased.MatchedFilter	Matched filter
phased.RangeDopplerResponse	Range-Doppler response
phased.StretchProcessor	Stretch processor for linear FM waveform
phased.TimeVaryingGain	Time varying gain control
Environment and Target Models	
phased.FreeSpace	Free space environment Note Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.
phased.RadarTarget	Radar target
phased.ConstantGammaClutter	Constant gamma clutter simulation
phased.BarrageJammer	Barrage jammer
Coordinate Systems and Motion Modeling	
phased.Platform	Motion platform

Image Acquisition Toolbox System Objects

If you install Image Acquisition Toolbox™ software, you can generate C/C++ code for the VideoDevice System object. See `imaq.VideoDevice` and “Code Generation with VideoDevice System Object”.

Defining MATLAB Variables for C/C++ Code Generation

- “Variables Definition for Code Generation” on page 36-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 36-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 36-7
- “Reassignment of Variable Properties” on page 36-9
- “Define and Initialize Persistent Variables” on page 36-10
- “Reuse the Same Variable with Different Properties” on page 36-11
- “Avoid Overflows in for-Loops” on page 36-16
- “Supported Variable Types” on page 36-18

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 36-3.

Best Practices for Defining Variables for C/C++ Code Generation

In this section...

“Define Variables By Assignment Before Using Them” on page 36-3

“Use Caution When Reassigning Variables” on page 36-6

“Use Type Cast Operators in Variable Definitions” on page 36-6

“Define Matrices Before Assigning Indexed Variables” on page 36-6

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:
<code>a = 14.7;</code>	a as a real double scalar.
<code>b = a;</code>	b with properties of a (real double scalar).
<code>c = zeros(5,2);</code>	c as a real 5-by-2 array of doubles.
<code>d = [1 2 3 4 5; 6 7 8 9 0];</code>	d as a real 5-by-2 array of doubles.
<code>y = int16(3);</code>	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 (see Defining a Variable for Multiple Execution Paths on page 36-4).

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 (see Defining Fields in a Structure on page 36-5).

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 36-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see “Define and Initialize Persistent Variables” on page 36-10.

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 $c \leq 0$),.

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
...
```

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 39-2.

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 36-9.

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 38-28.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

“When Redundant Copies Occur” on page 36-7

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 36-7

“Defining Uninitialized Variables” on page 36-8

When Redundant Copies Occur

During C/C++ code generation, MATLAB 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 36-7.

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 = fcn %#codegen

N = 5;
X = zeros(1,N);
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        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 = fcn2 %#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
```


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 36-11.

Define and Initialize Persistent Variables

Persistent variables are local to the function in which they are defined, but they retain their values in memory between calls to the function. To define persistent variables for C/C++ code generation, use the `persistent` statement, as in this example:

```
persistent PROD_X;
```

The definition should appear at the top of the function body, after the header and comments, but before the first use of the variable. During code generation, the value of the persistent variable is initialized to an empty matrix by default. You can assign your own value after the definition by using the `isempty` statement, as in this example:

```
function findProduct(inputvalue) %#codegen
persistent PROD_X

if isempty(PROD_X)
    PROD_X = 1;
end
PROD_X = PROD_X * inputvalue;
end
```

Reuse the Same Variable with Different Properties

In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 36-11

“When You Cannot Reuse Variables” on page 36-12

“Limitations of Variable Reuse” on page 36-14

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 MATLAB 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 (see “Viewing Variables in Your MATLAB Code” on page 29-58).

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)));
```

To compile this example and see how MATLAB renames the reused variable `t`, see Variable Reuse in an `if` Statement on page 36-12.

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
```

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 Compile example1.

For example, to generate a MEX function, enter:

```
codegen -o example1x -report example1.m -args {ones(5,5)}
```

Note `codegen` requires a MATLAB Coder license.

When the compilation is complete, `codegen` generates a MEX function, `example1x` in the current folder, and provides a link to the code generation report.

3 Open the code generation report.

4 In the MATLAB code pane of the code generation report, place your pointer over the variable *t* inside the `if` statement.

The code generation report highlights both instances of *t* in the `if` statement because they share the same class, size, and complexity. It displays the data type information for *t* at this point in the code. Here, *t* is a scalar double.

```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

Information for the selected variable:	
Size	1 x 1
Complex	No
Class	double

5 In the MATLAB code pane of the report, place your pointer over the variable *t* outside the for-loop.

This time, the report highlights both instances of t outside the `if` statement. The report indicates that t might hold up to 25 doubles. The size of t is `:25`, that is, a column vector containing a maximum of 25 doubles.

```
t = find(u);
y = sum(u(t(2:end-1)));
```

Information for the selected variable:	
Size	:25
Complex	No
Class	double

6 Click the **Variables** tab to view the list of variables used in `example1`.

The report displays a list of the variables in `example1`. There are two uniquely named local variables $t>1$ and $t>2$.

7 In the list of variables, place your pointer over $t>1$.

The code generation report highlights both instances of t in the `if` statement.

8 In the list of variables, place your pointer over $t>2$

The code generation report highlights both instances of t outside the `if` statement.

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`.
- Variables whose names are controlled using `coder.cstructname`.
- 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.

Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generation software detects that a loop variable might overflow on the last iteration of the for-loop, it reports an error.

To avoid this error, use the workarounds provided in the following table.

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> • The loop counter increments by 1 • The end value equals the maximum value of the integer type • The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:</p> <pre>N=intmax('int16') for k=N-10:N</pre> <p>with:</p> <pre>for k=1:10</pre>
<ul style="list-style-type: none"> • The loop counter decrements by 1 • The end value equals the minimum value of the integer type • The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:</p> <pre>N=intmin('int32') for k=N+10:-1:N</pre> <p>with:</p> <pre>for k=10:-1:1</pre>

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> • The loop counter 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 <p>The loop covers the full range of the integer type.</p>	<p>Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double. For example, rewrite:</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N % Loop body end to M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N) % Loop body end</pre>
<ul style="list-style-type: none"> • The loop counter increments or decrements by a value not equal to 1 • On last loop iteration, the loop variable value is not equal to the end value <hr/> <p>Note The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.</p> <hr/>	<p>Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.</p>

Supported Variable Types

You can use the following data types for C/C++ code generation from MATLAB:

Type	Description
char	Character array (string)
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	See “Fixed-Point Data Types”.

Defining Data for Code Generation

- “Data Definition for Code Generation” on page 37-2
- “Code Generation for Complex Data” on page 37-4
- “Code Generation for Characters” on page 37-6

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 the MATLAB environment:

Data	What's Different	More Information
Complex numbers	<ul style="list-style-type: none"> • 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 <hr/> <p>Note Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</p> <hr/>	“Code Generation for Complex Data” on page 37-4
Characters	Restricted to 8 bits of precision	“Code Generation for Characters” on page 37-6

Data	What's Different	More Information
Enumerated data	<ul style="list-style-type: none">• Supports integer-based enumerated types only• Restricted use in switch statements and for-loops	“Enumerated Data”
Function handles	<ul style="list-style-type: none">• Function handles must be scalar values• Same bound variable cannot reference different function handles• 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” on page 37-4

“Expressions With Complex Operands Yield Complex Results” on page 37-5

Restrictions When Defining Complex Variables

For code generation, you must set the complexity of variables at the time of assignment, either by assigning a complex constant or using the `complex` function, as in these examples:

```
x = 5 + 6i; % x is a complex number by assignment.
y = 7 + 8j; % y is a complex number by assignment.
x = complex(5,6); % x is the complex number 5 + 6i.
```

Once you set the type and size of a variable, you cannot cast it to another type or size. In the following example, the variable `x` is defined as complex and stays complex:

```
x = 1 + 2i; % Defines x as a complex variable.
y = int16(x); % Real and imaginary parts of y are int16.
x = 3; % x now has the value 3 + 0i.
```

Mismatches can also occur when you assign a real operand the complex result of an operation:

```
z = 3; % Sets type of z to double (real)
z = 3 + 2i; % ERROR: cannot recast z to complex
```

As a workaround, set the complexity of the operand to match the result of the operation:

```
m = complex(3); % Sets m to complex variable of value 3 + 0i
m = 5 + 6.7i; % Assigns a complex result to a complex number
```

Expressions With Complex Operands Yield Complex Results

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 example:

```
x = 2 + 3i;  
y = 2 - 3i;  
z = x + y; % z is 4 + 0i.
```

In MATLAB, this code generates the real result $z = 4$. During code generation, the types for x and y are known, but their values are not. Because either or both operands in this expression are complex, z is defined as a complex variable requiring storage for both 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.
```

Code Generation for Characters

The complete set of Unicode characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to generate code from your MATLAB algorithm.

Code Generation for Variable-Size Data

- “What Is Variable-Size Data?” on page 38-2
- “Variable-Size Data Definition for Code Generation” on page 38-3
- “Bounded Versus Unbounded Variable-Size Data” on page 38-4
- “Control Memory Allocation of Variable-Size Data” on page 38-5
- “Specify Variable-Size Data Without Dynamic Memory Allocation” on page 38-6
- “Variable-Size Data in Code Generation Reports” on page 38-9
- “Define Variable-Size Data for Code Generation” on page 38-11
- “C Code Interface for Arrays” on page 38-17
- “Diagnose and Fix Variable-Size Data Errors” on page 38-18
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 38-22
- “Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation” on page 38-30

What Is Variable-Size Data?

Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.

For example, in the following MATLAB function `nway`, `B` is a variable-size array; its length is not known at compile time.

```
function B = nway(A,n)
% Compute average of every N elements of A and put them in B.
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    error('n <= 0 or does not divide number of elements evenly');
end
```

Variable-Size Data Definition for Code Generation

In the MATLAB language, data can vary in size. By contrast, the semantics of generated code constrains the class, complexity, and shape of every expression, variable, and structure field. Therefore, for code generation, you must use each variable consistently. Each variable must:

- Be either complex or real (determined at first assignment)
- Have a consistent shape

For fixed-size data, the shape is the same as the size returned in MATLAB.

For example, if `size(A) == [4 5]`, the shape of variable A is 4 x 5.

For variable-size data, the shape can be abstract. That is, one or more dimensions can be unknown (such as `4x?` or `?x?`).

By default, the compiler detects code logic that attempts to change these fixed attributes after initial assignments, and flags these occurrences as errors during code generation. However, you can override this behavior by defining variables or structure fields as variable-size data.

For more information, see “Bounded Versus Unbounded Variable-Size Data” on page 38-4

Bounded Versus Unbounded Variable-Size Data

Bounded variable-size data has fixed upper bounds. *Unbounded variable-size data* does not have fixed upper bounds; this data *must* be allocated on the heap and requires dynamic memory allocation. You cannot use dynamic memory allocation for variable-size data in MATLAB Function blocks. Use bounded instead of unbounded variable-size data.

Control Memory Allocation of Variable-Size Data

Data whose size (in bytes) is greater than or equal to the dynamic memory allocation threshold is allocated on the heap. The default dynamic memory allocation threshold is 64 kilobytes. Data whose size is less than this threshold is allocated on the stack.

Dynamic memory allocation is an expensive operation; the performance cost might be too high for small data sets. If you use small variable-size data sets or data that does not change size at run time, disable dynamic memory allocation. See .

. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 38-6.

Specify Variable-Size Data Without Dynamic Memory Allocation

In this section...

“Fixing Upper Bounds Errors” on page 38-6

“Specifying Upper Bounds for Variable-Size Data” on page 38-6

Fixing Upper Bounds Errors

If MATLAB cannot determine or compute the upper bound, you must specify an upper bound. See “Specifying Upper Bounds for Variable-Size Data” on page 38-6 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 38-20

Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 38-6
- “Specifying Upper Bounds for Local Variable-Size Data” on page 38-6
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 38-8

When to Specify Upper Bounds for Variable-Size Data

When using static allocation on the stack during code generation, MATLAB must be able to determine upper bounds for variable-size data. Specify the upper bounds explicitly for variable-size data from external sources, such as inputs and outputs.

Specifying Upper Bounds for Local Variable-Size Data

When using static allocation, MATLAB uses a sophisticated analysis to calculate the upper bounds of local data at compile time. However, when the analysis fails to detect an upper bound or calculates an upper bound that is not precise enough for your application, you need to specify upper bounds explicitly for local variables.

Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data. Use the `assert` function with relational operators to constrain the value of variables that specify the dimensions of variable-size data. 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, defining `L` and `M` as variable-sized matrices with upper bounds of 5 for each dimension.

Specifying the Upper Bounds for All Instances of a Local Variable.

Use the `coder.varsize` function to specify the upper bounds for all instances of a local variable in a 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`:

- First dimension is fixed at size 1
- Second dimension can grow to an upper bound of 10

By default, `coder.varsize` assumes dimensions of 1 are fixed size. For more information, see the `coder.varsize` reference page.

Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For code in a MATLAB Function block, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function y = var_by_assign(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```


Variable-Size Data in Code Generation Reports

In this section...
“What Reports Tell You About Size” on page 38-9
“How Size Appears in Code Generation Reports” on page 38-10
“How to Generate a Code Generation Report” on page 38-10

What Reports Tell You About Size

Code generation reports:

- Differentiate fixed-size from variable-size data
- Identify variable-size data with unknown upper bounds
- Identify variable-size data with fixed dimensions

If you define a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends * to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions does not change during execution.

- Provide guidance on how to fix size mismatch and upper bounds errors.

How Size Appears in Code Generation Reports

Variable	Type	Size
B	Output	1 x :?
A	Input	1 x :100
n	Input	1 x 1

:? means variable size, unknown upper bound

No colon prefix (:) means fixed size

:100 means variable size, upper bound = 100

Variable	Type	Size
y	Output	1 x 10*

* means that you declared y as variable size, but subsequently fixed its dimensions

How to Generate a Code Generation Report

When you build a Simulink® model that contains MATLAB Function blocks, Simulink automatically generates a report in HTML format for each MATLAB Function block in your model. See “MATLAB Function Reports” on page 29-53

Define Variable-Size Data for Code Generation

In this section...
“When to Define Variable-Size Data Explicitly” on page 38-11
“Using a Matrix Constructor with Nonconstant Dimensions” on page 38-12
“Inferring Variable Size from Multiple Assignments” on page 38-12
“Defining Variable-Size Data Explicitly Using <code>coder.varsize</code> ” on page 38-13

When to Define Variable-Size Data Explicitly

For code generation, you must assign variables to have a specific class, size, and complexity before using them in operations or returning them as outputs. Generally, you cannot reassign variable properties after the initial assignment. Therefore, attempts to grow a variable or structure field after assigning it a fixed size might cause a compilation error. In these cases, you must explicitly define the data as variable sized using one of these methods:

Method	See
Assign the data from a variable-size matrix constructor such as <ul style="list-style-type: none"> • <code>ones</code> • <code>zeros</code> • <code>repmat</code> 	“Using a Matrix Constructor with Nonconstant Dimensions” on page 38-12
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Inferring Variable Size from Multiple Assignments” on page 38-12
Define all instances of a variable to be variable sized	“Defining Variable-Size Data Explicitly Using <code>coder.varsize</code> ” on page 38-13

Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For code in a MATLAB Function block, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function y = var_by_assign(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

Inferring Variable Size from Multiple Assignments

You can define variable-size data by assigning multiple, constant sizes to the same variable before you use (read) the variable in your code. When MATLAB uses static allocation on the stack for code generation, 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, MATLAB assumes that the dimension is fixed at that size. The assignments can specify different shapes as well as sizes.

Inferring Upper Bounds from Multiple Definitions with Different Shapes

```
function y = var_by_multiassign(u) %#codegen
if (u > 0)
    y = ones(3,4,5);
else
    y = zeros(3,1);
end
```

When static allocation is used, this function infers that `y` is a matrix with three dimensions, where:

- First dimension is fixed at size 3
- Second dimension is variable with an upper bound of 4

- Third dimension is variable with an upper bound of 5

The code generation report represents the size of matrix `y` like this:

Variable	Type	Size
<code>y</code>	Output	3 x :4 x :5

Defining Variable-Size Data Explicitly Using `coder.versize`

Use the function `coder.versize` to define one or more variables or structure fields as variable-size data. Optionally, you can also specify which dimensions vary along with their upper bounds (see “Specifying Which Dimensions Vary” on page 38-13). For example:

- Define `B` as a variable-size 2-by-2 matrix, where each dimension has an upper bound of 64:

```
coder.versize('B', [64 64]);
```

- Define `B` as a variable-size matrix:

```
coder.versize('B');
```

When you supply only the first argument, `coder.versize` assumes all dimensions of `B` can vary and that the upper bound is `size(B)`.

For more information, see the `coder.versize` reference page.

Specifying Which Dimensions Vary

You can use the function `coder.versize` to specify which dimensions vary. For example, the following statement defines `B` as a row vector whose first dimension is fixed at 2, but whose second dimension can grow to an upper bound of 16:

```
coder.versize('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 “Defining Variable-Size Matrices with Singleton Dimensions” on page 38-14).

For more information about the syntax, see the `coder.varsize` reference page.

Allowing a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix `Y` with fixed 2-by-2 dimensions before 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`, MATLAB infers `Y` to be a fixed-size, 2-by-2 matrix and generates a size mismatch error during code generation.

Defining 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 using matrix constructor expressions or matrix functions.

For example, in this function, both X and Y behave like vectors where only their second dimensions are variable sized:

```
function [X,Y] = dim_singleton_vectors(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. For more information, see the `coder.varsize` reference page.

Defining Variable-Size Structure Fields

To define structure fields as variable-size arrays, use colon (:) as the index expression. The colon (:) indicates that all elements of the array are variable sized. 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 sized.

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 sized.

C Code Interface for Arrays

C Code Interface for Statically Allocated Arrays

In generated code, MATLAB contains two pieces of information about statically allocated arrays: the maximum size of the array and its actual size.

For example, consider the MATLAB function `uniquetol`:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B');
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
```

Generate code for `uniquetol` specifying that input `A` is a variable-size real double vector whose first dimension is fixed at 1 and second dimension can vary up to 100 elements.

In the generated code, the function declaration is:

There are two pieces of information about `A`:

- `double A_data[100]`: the maximum size of input `A` (where 100 is the maximum size specified using `coder.typeof`).
- `int A_size[2]`: the actual size of the input.

Diagnose and Fix Variable-Size Data Errors

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 38-18

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 38-20

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.

Performing Binary Operations on Fixed and Variable-Size Operands

You cannot perform binary operations on operands of different sizes. Operands have different sizes if one has fixed dimensions and the other has variable dimensions. For example:

```
function z = mismatch_operands(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x + y;
```

When you compile this function, you get an error because `y` has fixed dimensions (3 x 3), but `x` has variable dimensions. Fix this problem by using explicit indexing to make `x` the same size as `y`:

```
function z = mismatch_operands_fix(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x(1:3,1:3) + y;
```

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” on page 38-22

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 38-24

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 38-25

“Incompatibility with MATLAB in Determining Class of Empty Arrays” on page 38-26

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 38-27

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 38-28

“Incompatibility with MATLAB in Concatenating Variable-Size Matrices” on page 38-29

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 38-29

Incompatibility with MATLAB for Scalar Expansion

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

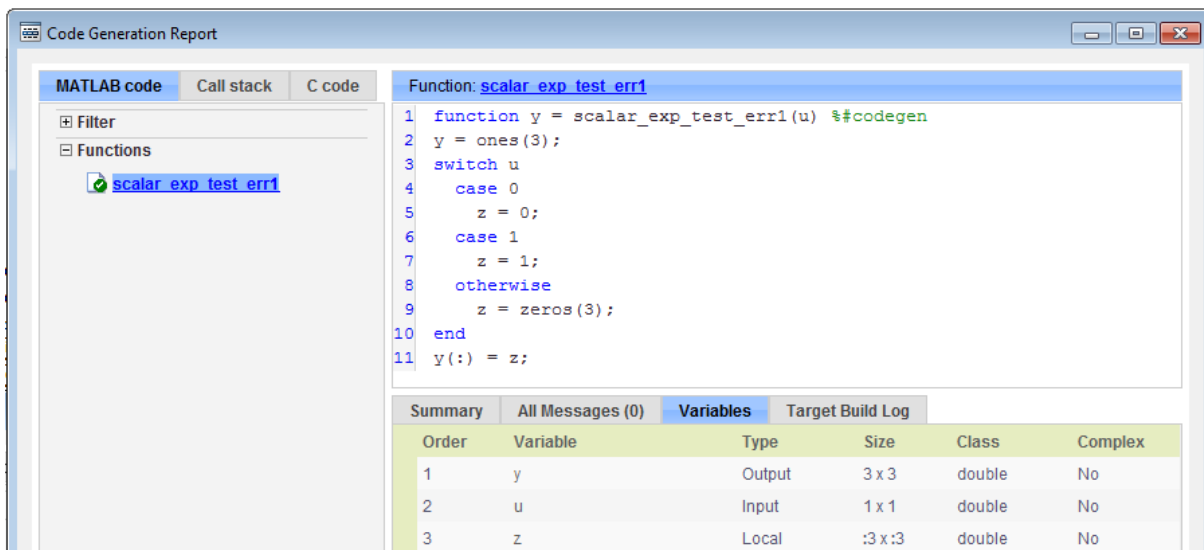
During code generation, the standard MATLAB scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Instead, it generates a size mismatch error at run time for MEX

functions. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior.

For example, in the following function, `z` is scalar for the `switch` statement case 0 and case 1. MATLAB applies scalar expansion when evaluating `y(:) = z;` for these two cases.

```
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 generation software determines that `z` is variable size with an upper bound of 3.



The screenshot shows the Code Generation Report window for the function `scalar_exp_test_err1`. The window is divided into several sections:

- Navigation:** MATLAB code, Call stack, C code.
- Filter:** A search box for filtering functions.
- Functions:** A list of functions, with `scalar_exp_test_err1` selected.
- Code View:** The MATLAB code for the function, showing a `switch` statement with three cases. Line 11 shows `y(:) = z;`.
- Summary Table:** A table with columns for Order, Variable, Type, Size, Class, and Complex. It lists three variables: `y` (Output, 3 x 3, double, No), `u` (Input, 1 x 1, double, No), and `z` (Local, :3 x :3, double, No).

Order	Variable	Type	Size	Class	Complex
1	y	Output	3 x 3	double	No
2	u	Input	1 x 1	double	No
3	z	Local	:3 x :3	double	No

If you run the MEX function with `u` equal to zero or one, even though `z` is scalar at run time, the generated code does not perform scalar expansion and a run-time error occurs.

```
scalar_exp_test_err1_mex(0)
Sizes mismatch: 9 ~= 1.
```

```
Error in scalar_exp_test_err1 (line 11)
y(:) = z;
```

Workaround

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 generation software 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.

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 generation software 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 Vector-Vector Indexing

In vector-vector indexing, you use one vector as an index into another vector. When either vector is variable sized, you might get a run-time error during code generation. Consider the index expression `A(B)`. The general rule for indexing is that `size(A(B)) == size(B)`. However, when both `A` and `B` are vectors, MATLAB applies a special rule: use the orientation of `A` as the orientation of the output. For example, if `size(A) == [1 5]` and `size(B) == [3 1]`, then `size(A(B)) == [1 3]`.

In this situation, if the code generation software detects that both `A` and `B` are vectors at compile time, it applies the special rule and gives the same result

as MATLAB. However, if either **A** or **B** is a variable-size matrix (has shape $? \times ?$) at compile time, the code generation software applies only the general indexing rule. Then, if both **A** and **B** become vectors at run time, the code generation software reports a run-time error when you run the MEX function. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior. It is best practice to generate and test a MEX function before generating C code.

Workaround

Force your data to be a vector by using the colon operator for indexing: `A(B(:))`. For example, suppose 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. As a result, the code generation software applies the standard vector-vector indexing rule.

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

The following limitations apply 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` as highlighted in the following code.

```
M=zeros(1,10);  
for i = 1:10  
    M(i) = 5;  
end
```

- `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 in the following example:

```
...  
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, as shown in the highlighted code.

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

For code generation, when you concatenate variable-sized arrays, the dimensions that are not being concatenated must match exactly.

Dynamic Memory Allocation Not Supported for MATLAB Function Blocks

You cannot use dynamic memory allocation for variable-size data in MATLAB Function blocks. Use bounded instead of unbounded variable-size data.

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation

In this section...

“Common Restrictions” on page 38-30

“Toolbox Functions with Variable Sizing Restrictions” on page 38-31

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 Variable Sizing Restrictions” on page 38-31.

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 $1 \times n$ or $n \times 1$ (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

When the function selects the working dimension automatically, it bases the selection on the upper bounds for the dimension sizes. In the case of the `sum` function, `sum(X)` selects its working dimension automatically, while `sum(X, dim)` uses `dim` as the explicit working dimension.

For example, if `X` is a variable-size matrix with dimensions $1 \times 3 \times 5$, `sum(x)` behaves like `sum(X,2)` in generated code. In MATLAB, it behaves like `sum(X,2)` provided `size(X,2)` is not 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`. Consequently, you get a run-time error if an automatically selected working dimension assumes a length of 1 at run time.

To avoid the issue, specify the intended working dimension explicitly as a constant value.

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 Variable Sizing Restrictions

The following restrictions apply to specific toolbox functions, but only for code generation.

Function	Restrictions with Variable-Size Data
all	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
any	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
bsxfun	<ul style="list-style-type: none"> • Dimensions expand only where one input array or the other has a fixed length of 1.
cat	<ul style="list-style-type: none"> • Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.

Function	Restrictions with Variable-Size Data
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 38-30. • Input vectors must have the same orientation, either both row vectors or both column vectors.
cov	<ul style="list-style-type: none"> • For cov(X), see “Array-to-vector restriction” on page 38-31.
cross	<ul style="list-style-type: none"> • Variable-size array inputs that become vectors at run time must have the same orientation.
deconv	<ul style="list-style-type: none"> • For both arguments, see “Variable-length vector restriction” on page 38-30.
detrend	<ul style="list-style-type: none"> • For first argument for row vectors only, see “Array-to-vector restriction” on page 38-31 .
diag	<ul style="list-style-type: none"> • See “Array-to-vector restriction” on page 38-31 .
diff	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • 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	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30.

Function	Restrictions with Variable-Size Data
filter	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 38-30. • See “Automatic dimension restriction” on page 38-30.
hist	<ul style="list-style-type: none"> • For second argument, see “Variable-length vector restriction” on page 38-30. • For second input argument, see “Array-to-scalar restriction” on page 38-31.
histc	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30.
ifft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30.
ind2sub	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
interp1	<ul style="list-style-type: none"> • For the Y input and xi input, see “Array-to-vector restriction” on page 38-31. • Y input can become a column vector dynamically. • A run-time error occurs if Y input is not a variable-length vector and becomes a row vector at run time.
ipermute	<ul style="list-style-type: none"> • Order input must be fixed size.
issorted	<ul style="list-style-type: none"> • For optional rows input, see “Variable-length vector restriction” on page 38-30.

Function	Restrictions with Variable-Size Data
magic	<ul style="list-style-type: none">• Argument must be a constant.• Output can be fixed-size matrices only.
max	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 38-30.
mean	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 38-30.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
median	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 38-30.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
min	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 38-30.
mode	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 38-30.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
mtimes	<ul style="list-style-type: none"> When an input is variable-size, MATLAB determines whether to generate code for a general matrix*matrix multiplication or a scalar*matrix multiplication, based on whether one of the arguments is a fixed-size scalar. If neither argument is a fixed-size scalar, the inner dimensions of the two arguments must agree even if a variable-size matrix input is a scalar at run time.
nchoosek	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Order input must be fixed-size.
planerot	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> See “Variable-length vector restriction” on page 38-30.
polyfit	<ul style="list-style-type: none"> For first and second arguments, see “Variable-length vector restriction” on page 38-30.
prod	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 38-30. An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
rand	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>rand(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>rand([1,N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
randn	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randn(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randn([1,N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
reshape	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 38-30.
shiftdim	<ul style="list-style-type: none"> • 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. Consequently, 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).

Function	Restrictions with Variable-Size Data
	<ul style="list-style-type: none"> • First input argument must have the same number of dimensions when you supply a positive number of shifts.
std	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
sub2ind	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
sum	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
trapz	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
typecast	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 38-30 on first argument.
var	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 38-30. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Code Generation for MATLAB Structures

- “Structure Definition for Code Generation” on page 39-2
- “Structure Operations Allowed for Code Generation” on page 39-3
- “Define Scalar Structures for Code Generation” on page 39-4
- “Define Arrays of Structures for Code Generation” on page 39-7
- “Make Structures Persistent” on page 39-9
- “Index Substructures and Fields” on page 39-10
- “Assign Values to Structures and Fields” on page 39-12
- “Pass Large Structures as Input Parameters” on page 39-14

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 39-3
Observe restrictions on properties and values of scalar structures.	“Define Scalar Structures for Code Generation” on page 39-4
Make structures uniform in arrays.	“Define Arrays of Structures for Code Generation” on page 39-7
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 29-85
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 29-88

Structure Operations Allowed for Code Generation

To generate efficient standalone code for MATLAB structures, you are restricted to the following operations:

- Define structures as local and persistent variables by assignment and using the `struct` function
- 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...

“Restriction When Using struct” on page 39-4

“Restrictions When Defining Scalar Structures by Assignment” on page 39-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 39-4

“Restriction on Adding New Fields After First Use” on page 39-5

Restriction When Using struct

When you use the `struct` function to create scalar structures for code generation, you cannot create structures of cell arrays.

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” on page 39-7

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 39-7

“Defining an Array of Structures Using Concatenation” on page 39-8

Ensuring Consistency of Fields

When you create an array of MATLAB structures with the intent of generating code, you must be sure that each structure field in the array has the same size, type, and complexity.

Once you have created the array of structures, you can make the structure fields variable-size using `coder.varsize`. For more information, see “Declare a variable-size structure field.”

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 39-4.
- 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 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 “Concatenating 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;  
    ...
```

Make Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the `isempty` statement, as described in “Define and Initialize Persistent Variables” on page 36-10.

For example, the following function defines structure `X` to be persistent and initializes its fields `a` and `b`:

```
function f(u) %#codegen
persistent X

if isempty(X)
    X.a = 1;
    X.b = 2;
end
```

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 39-7 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

Use these guidelines when assigning values to a structure, substructure, or field for code generation:

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.

Do not use field values as constants

The values stored in the fields of a structure are not treated as constant values in generated code. Therefore, you cannot use field values to set the size or class of other data. For example, the following code generates a compiler error if variable-sizing is disabled:

```
...
Y.a = 3;
Y.b = 5;
X = zeros(Y.a,Y.b); % Generates an error
```

In this example, even though you set fields `a` and `b` of structure `Y` to the values 3 and 5 respectively, `Y.a` and `Y.b` are not constants in generated code. Therefore, they are not valid arguments to pass to the function `zeros`.

Note An exception to this behavior occurs if the structure is declared completely using the `struct` function

```
...  
Y = struct('a',3,'b',5);  
X = zeros(Y.a,Y.b); % Generates a fixed-size 3 X 5 matrix
```

Do not assign mxArray's to structures

You cannot assign mxArray's to structure elements; convert mxArray's to known types before code generation (see “Working with mxArray's” on page 44-17).

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 generation software 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 Enumerated Data

- “Enumerated Data Definition for Code Generation” on page 40-2
- “Enumerated Types Supported for Code Generation” on page 40-3
- “When to Use Enumerated Data for Code Generation” on page 40-4
- “Generate Code for Enumerated Data from MATLAB Function Blocks” on page 40-5
- “Define Enumerated Data for Code Generation” on page 40-6
- “Instantiate Enumerated Types for Code Generation” on page 40-8
- “Operations on Enumerated Data Allowed for Code Generation” on page 40-9
- “Include Enumerated Data in Control Flow Statements” on page 40-12
- “Customize Enumerated Types Based on Simulink.IntEnumType” on page 40-18
- “Control Names of Enumerated Type Values in Generated Code” on page 40-19
- “Change and Reload Enumerated Data Types” on page 40-21
- “Restrictions on Use of Enumerated Data in for-Loops” on page 40-22
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 40-23

Enumerated Data Definition for Code Generation

To generate efficient standalone code for enumerated data, you must define and use enumerated types differently than you normally would when running your code in the MATLAB environment:

What's Different	More Information
Supports integer-based enumerated types only	"Enumerated Types Supported in MATLAB Function Blocks" on page 29-107
Each enumerated data type must be defined in a separate file on the MATLAB path	"Define Enumerated Data Types for MATLAB Function Blocks" on page 29-109
Restricted set of operations	"Operations on Enumerated Data" on page 29-118
Restricted use in for-loops	"Restrictions on Use of Enumerated Data in for-Loops" on page 40-22

Enumerated Types Supported for Code Generation

Enumerated Type Based on Simulink.IntEnumType

This enumerated data type is based on the built-in type `Simulink.IntEnumType`, which is available with a Simulink license. Use this enumerated type when exchanging enumerated data with Simulink blocks and Stateflow charts.

Syntax

```
classdef(Enumeration) type_name < Simulink.IntEnumType
```

Example

```
classdef(Enumeration) myMode < Simulink.IntEnumType
    enumeration
        OFF(0)
        ON(1)
    end
end
```

How to Use

Here are the basic guidelines for using enumerated data based on `Simulink.IntEnumType`:

Application	What to Do
When exchanging enumerated data with Simulink blocks	Define enumerated data in MATLAB Function blocks in Simulink models. Requires Simulink software.
When exchanging enumerated data with Stateflow charts	Define enumerated data in MATLAB functions in Stateflow charts. Requires Simulink and Stateflow software.

See “About Simulink Enumerations” on page 47-2 for more information about enumerated types based on `Simulink.IntEnumType`

When to Use Enumerated Data for Code Generation

You can use enumerated types to represent program states and to control program logic, especially when you need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow more robust error checking than integers or strings

For example, if you mistype the name of an element in the enumerated type, you get a compile-time error that the element does not belong to the set of allowable values.

- Produce more efficient code than strings

For example, comparisons of enumerated values execute faster than comparisons of strings.

Generate Code for Enumerated Data from MATLAB Function Blocks

Step	Action	How?
1	Define an enumerated data type that inherits from <code>Simulink.IntEnumType</code> .	See “Define Enumerated Data Types for MATLAB Function Blocks” on page 29-109
2	Add the enumerated data to your MATLAB Function block.	See “Add Inputs, Outputs, and Parameters as Enumerated Data” on page 29-110
3	Instantiate the enumerated type in your MATLAB Function block.	See “Instantiate Enumerated Data in MATLAB Function Blocks” on page 29-113
4	Simulate and/or generate code.	See “Enumerations”

This workflow requires the following licenses:

- Simulink (for simulation)
- MATLAB Coder and Simulink Coder (for code generation)

Define Enumerated Data for Code Generation

Follow these steps to define enumerated data for code generation from MATLAB algorithms:

- 1 Create a class definition file.

In the MATLAB Command Window, select **File > New > Class**.

- 2 Enter the class definition as follows:

```
classdef(Enumeration) EnumTypeName < int32
```

For example, the following code defines an enumerated type called `sysMode`:

```
classdef(Enumeration) sysMode < int32
    ...
end
```

EnumTypeName is a case-sensitive string that must be unique among data type names and workspace variable names. It must inherit from the built-in type `int32`.

- 3 Define enumerated values in an enumeration section as follows:

```
classdef(Enumeration) EnumTypeName < int32
    enumeration
        EnumName(N)
        ...
    end
end
```

For example, the following code defines a set of two values for enumerated type `sysMode`:

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

Each enumerated value consists of a string *EnumName* and an underlying integer *N*. Each *EnumName* must be unique within its type, but can also appear in other enumerated types. The underlying integers need not be either consecutive or ordered, nor must they be unique within the type or across types.

4 Save the file on the MATLAB path.

The name of the file must match the name of the enumerated data type. The match is case sensitive.

To add a 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`.

For examples of enumerated data type definitions, see “Define Enumerated Data for Code Generation” on page 40-6.

Naming Enumerated Types for Code Generation

You must use a unique name for each enumerated data type. The name of an enumerated data type cannot match the name of a toolbox function supported for code generation, or another data type or a variable in the MATLAB base workspace. Otherwise, a name conflict occurs.

For example, you cannot name an enumerated data type `mode` because MATLAB for code generation provides a toolbox function of the same name.

For a list of toolbox functions supported for code generation, see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2.

Instantiate Enumerated Types for Code Generation

To instantiate an enumerated type for code generation from MATLAB algorithms, use dot notation to specify *ClassName.EnumName*. For an example, see “Include Enumerated Data in Control Flow Statements” on page 40-12.

Operations on Enumerated Data Allowed for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples are based on the definitions of the enumeration type `LEDcolor` described in .

Assignment Operator, =

Example	Result
<pre>xon = LEDcolor.GREEN xoff = LEDcolor.RED</pre>	<pre>xon = GREEN xoff = RED</pre>

Relational Operators, < > <= >= == ~=

Example	Result
<pre>xon == xoff</pre>	<pre>ans = 0</pre>
<pre>xon <= xoff</pre>	<pre>ans = 1</pre>
<pre>xon > xoff</pre>	<pre>ans = 0</pre>

Cast Operation

Example	Result
<code>double(LEDcolor.RED)</code>	ans = 2
<code>z = 2</code> <code>y = LEDcolor(z)</code>	z = 2 y = RED

Indexing Operation

Example	Result
<code>m = [1 2]</code> <code>n = LEDcolor(m)</code> <code>p = n(LEDcolor.GREEN)</code>	m = 1 2 n = GREEN RED p = GREEN

Control Flow Statements: if, switch, while

Statement	Example	Executable Example
if	<pre> if state == sysMode.ON led = LEDcolor.GREEN; else led = LEDcolor.RED; end </pre>	<p>“if Statement with Enumerated Data Types” on page 40-12</p>
switch	<pre> switch button case VCRButton.Stop state = VCRState.Stop; case VCRButton.PlayOrPause state = VCRState.Play; case VCRButton.Next state = VCRState.Forward; case VCRButton.Previous state = VCRState.Rewind; otherwise state = VCRState.Stop; end </pre>	<p>“switch Statement with Enumerated Data Types” on page 40-13</p>
while	<pre> while state ~= State.Ready switch state case State.Standby initialize(); state = State.Boot; case State.Boot boot(); state = State.Ready; end end end </pre>	<p>“while Statement with Enumerated Data Types” on page 40-16</p>

Include Enumerated Data in Control Flow Statements

The following control statements work with enumerated operands in generated code. However, there are restrictions (see “Restrictions on Use of Enumerated Data in for-Loops” on page 40-22).

if Statement with Enumerated Data Types

This example is based on the definition of the enumeration types `LEDcolor` and `sysMode`. The function `displayState` uses these enumerated data types to activate an LED display.

Class Definition: `sysMode`

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `sysMode.m`.

Class Definition: `LEDcolor`

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
end
```

This definition must reside on the MATLAB path in a file called `LEDcolor.m`.

MATLAB Function: `displayState`

This function uses enumerated data to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state and lights a red LED display to indicate the OFF state.


```
function led = displayState(state)
%#codegen

if state == sysMode.ON
    led = LEDcolor.GREEN;
else
    led = LEDcolor.RED;
end
```

Build and Test a MEX Function for displayState

- 1 Generate a MEX function for displayState. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen displayState -args {sysMode.ON}
```

- 2 Test the function. For example,

```
displayState(sysMode.OFF)
```

You should get the following result:

```
ans =

    RED
```

switch Statement with Enumerated Data Types

This example is based on the definition of the enumeration types `VCRState` and `VCRButton`. The function `VCR` uses these enumerated data types to set the state of the VCR.

Class Definition: VCRState

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
        Play(2),
        Forward(3),
```

```
        Rewind(4)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRState.m`.

Class Definition: VCRButton

```
classdef(Enumeration) VCRButton < int32
    enumeration
        Stop(1),
        PlayOrPause(2),
        Next(3),
        Previous(4)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRButton.m`.

MATLAB Function: VCR

This function uses enumerated data to set the state of a VCR, based on the initial state of the VCR and the state of the VCR button.

```
function s = VCR(button)
    %#codegen

    persistent state

    if isempty(state)
        state = VCRState.Stop;
    end

    switch state
        case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}
            state = handleDefault(button);
        case VCRState.Play
            switch button
```

```

        case VCRButton.PlayOrPause, state = VCRState.Pause;
        otherwise, state = handleDefault(button);
    end
case VCRState.Pause
    switch button
        case VCRButton.PlayOrPause, state = VCRState.Play;
        otherwise, state = handleDefault(button);
    end
end
s = state;

function state = handleDefault(button)
switch button
    case VCRButton.Stop, state = VCRState.Stop;
    case VCRButton.PlayOrPause, state = VCRState.Play;
    case VCRButton.Next, state = VCRState.Forward;
    case VCRButton.Previous, state = VCRState.Rewind;
    otherwise, state = VCRState.Stop;
end

```

Build and Test a MEX Function for VCR

- 1 Generate a MEX function for VCR. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen -args {VCRButton.Stop} VCR
```

- 2 Test the function. For example,

```
s = VCR(VCRButton.Stop)
```

You should get the following result:

```
s =

    Stop
```

while Statement with Enumerated Data Types

This example is based on the definition of the enumeration type `State`. The function `Setup` uses this enumerated data type to set the state of a device.

Class Definition: `State`

```
classdef(Enumeration) State < int32
    enumeration
        Standby(0),
        Boot(1),
        Ready(2)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `State.m`.

MATLAB Function: `Setup`

The following function `Setup` uses enumerated data to set the state of a device.

```
function s = Setup(initState)
%#codegen

state = initState;

if isempty(state)
    state = State.Standby;
end

while state ~= State.Ready
    switch state
        case State.Standby
            initialize();
            state = State.Boot;
        case State.Boot
            boot();
            state = State.Ready;
    end
end
```

```
s = state;

function initialize()
% Perform initialization.

function boot()
% Boot the device.
```

Build and Test a MEX Executable for Setup

- 1 Generate a MEX executable for Setup. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen Setup -args {State.Standby}
```

- 2 Test the function. For example,

```
s = Setup(State.Standby)
```

You should get the following result:

```
s =

    Ready
```

Customize Enumerated Types Based on Simulink.IntEnumType

You can customize a Simulink enumerated type by using the same techniques that work with MATLAB classes, as described in [Modifying Superclass Methods and Properties](#). For more information, see “[Customize Simulink Enumeration](#)” on page 47-4.

Control Names of Enumerated Type Values in Generated Code

This example shows how to control the name of enumerated type values in code generated by MATLAB Coder. (Requires a MATLAB Coder license.) The example uses the enumerated data type definitions and function `displayState` described in “Include Enumerated Data in Control Flow Statements” on page 40-12.

- 1 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

`codegen` generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 2 Click the **View Report** link.

- 3 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The report displays the header file containing the enumerated data type definition.

```
typedef enum LEDcolor
{
    LEDcolor_GREEN = 1,
    LEDcolor_RED
} LEDcolor;
```

The enumerated value names include the class name prefix `LEDcolor_`.

- 4 Modify the definition of `LEDcolor` to override the `addClassNameToEnumNames` method. Set the return value to `false` instead of `true` so that the enumerated value names in the generated code do not contain the class prefix.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
```

```
        RED(2),
    end

    methods(Static)
        function y=addClassNameToEnumNames()
            y=false;
        end
    end
end
```

5 Clear existing class instances:

```
clear classes
```

6 Generate code again.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

7 Open the code generation report and look at the enumerated type definition in `displayState_types.h`.

```
typedef enum LEDcolor
{
    GREEN = 1,
    RED
} LEDcolor;
```

This time the enumerated value names do not include the class name prefix.

For more information, see:

- `codegen`
- “Include Enumerated Data in Control Flow Statements” on page 40-12 for a description of the example function `displayState` and its enumerated type definitions

Change and Reload Enumerated Data Types

You can change the definition of an enumerated data type 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 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 enumerated data type from the base workspace and cache.

If In Base Workspace...	If In Cache...
Do one of the following: <ul style="list-style-type: none">• Locate and delete specific obsolete instances.• Delete the classes from the workspace by using the <code>clear classes</code> command. For more information, see <code>clear</code>.	<ul style="list-style-type: none">• Clear MEX functions that are caching instances of the class.

Restrictions on Use of Enumerated Data in for-Loops

Do not use enumerated data as the loop counter variable in for-loops

To iterate over a range of enumerated data with consecutive values, you can cast the enumerated data to `int32` in the loop counter.

For example, suppose you define an enumerated type `ColorCodes` as follows:

```
classdef(Enumeration) ColorCodes < int32
    enumeration
        Red(1),
        Blue(2),
        Green(3)
        Yellow(4)
        Purple(5)
    end
end
```

Because the enumerated values are consecutive, you can use `ColorCodes` data in a for-loop like this:

```
...
for i = int32(ColorCodes.Red):int32(ColorCodes.Purple)
    c = ColorCodes(i);
    ...
end
```

Toolbox Functions That Support Enumerated Types for Code Generation

The following MATLAB toolbox functions support enumerated types for code generation:

- `cast`
- `cat`
- `circshift`
- `flipdim`
- `fliplr`
- `flipud`
- `histc`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `shiftdim`
- `sort`
- `sortrows`

- squeeze

Code Generation for MATLAB Classes

- “MATLAB Classes Definition for Code Generation” on page 41-2
- “Classes That Support Code Generation” on page 41-8
- “Generate Code for MATLAB Value Classes” on page 41-9
- “Generate Code for MATLAB Handle Classes and System Objects” on page 41-15
- “MATLAB Classes in Code Generation Reports” on page 41-18
- “Troubleshooting Issues with MATLAB Classes” on page 41-21

MATLAB Classes Definition for Code Generation

To generate efficient standalone code for MATLAB classes, you must use classes differently than you normally would when running your code in the MATLAB environment.

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an @-folder.	“Creating a Single, Self-Contained Class Definition File”
Restricted set of language features.	“Language Limitations” on page 41-2
Restricted set of code generation features.	“Code Generation Features Not Compatible with Classes” on page 41-3
Definition of class properties.	“Defining Class Properties for Code Generation” on page 41-4
Use of handle classes.	“Generate Code for MATLAB Handle Classes and System Objects” on page 41-15
Calls to base class constructor.	“Calls to Base Class Constructor” on page 41-6
Global variables containing MATLAB objects are not supported for code generation.	N/A

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
- 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`
 - `delete`
 - `eq`
 - `findobj`
 - `findpro`

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
```
- If an entry-point MATLAB function has an input or output that is a MATLAB class, you cannot generate code for this function.

For example, if function `f00` takes one input, `a`, that is a MATLAB object, you cannot generate code for `f00` by executing:

```
codegen foo -args {a}
```

- You cannot generate code for a value class that has a `set.prop` method. For example, you cannot generate code for the following `Square` class because of the `set.side` method.

```
classdef Square < Shape %#codegen
    properties
        side;
    end
    methods
        function obj = Square(side)
            obj = obj@Shape(side^2);
            obj.side = side;
        end
        function set.side(obj,value)
            obj.side = value;
            obj.area = value^2;
        end
    end
end
```

To generate code for this class, modify the class definition to remove the `set.side` method.

- 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 to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- The `coder.nullcopy` function does not support MATLAB classes as inputs.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you normally would when running your code in the MATLAB environment:

- 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 generation software 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 generation software assumes that it is undefined at the beginning of the constructor. The code generation software does not assign an empty matrix as the default.
 - If the property does not have an initial value and the code generation software 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;
```

- If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.
- `coder. varsize` is not supported for class properties.

- 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).

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
```

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 41-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 41-15

For more information, see:

- “Classes in the MATLAB Language”
- “MATLAB Classes Definition for Code Generation” on page 41-2

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 In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the complete list of properties, expand the list as shown for `obj`.

The screenshot shows the 'Code Generation Report' window. On the left is a tree view of the code structure, including classes like Rhombus, Shape, and Square, and methods like Rhombus, getarea, and use_shape. The main area displays the MATLAB code for the Rhombus class and its constructor. Below the code is a table with columns for Order, Variable, Type, Size, Class, and Complex.

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
18

```

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1	centerX	Property	1 x 1	double	No
1.2	centerY	Property	1 x 1	double	No
1.3	diag1	Property	1 x 1	double	No
1.4	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

8 At the top right side of the report, expand the **Calls** list.

The **Calls** list shows that there is a call to the Rhombus constructor from use_shape and that this constructor calls the Shape constructor.

The screenshot shows the 'Code Generation Report' window with the following content:

MATLAB code | Call stack | C code

Method: Rhombus | Calls: Select a function call: [dropdown menu]

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
18

```

Summary | All Messages (0) | Variables | Target Build Log

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1	centerX	Property	1 x 1	double	No
1.2	centerY	Property	1 x 1	double	No
1.3	diag1	Property	1 x 1	double	No
1.4	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

- 9 The constructor for the Rhombus class calls the Shape method of the base Shape class: `obj@Shape`. In the report, click the Shape link in this call.

The screenshot shows the 'Code Generation Report' window. On the left is a tree view of the code structure, including classes like Rhombus, Shape, and Square, and methods like Rhombus, getarea, and Shape. The main area displays the MATLAB code for the Rhombus class. A blue circle highlights the `obj@Shape` link in the constructor function. Below the code is a table with columns for Order, Variable, Type, Size, Class, and Complex.

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape:centerX,centerY;
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
18

```

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1	centerX	Property	1 x 1	double	No
1.2	centerY	Property	1 x 1	double	No
1.3	diag1	Property	1 x 1	double	No
1.4	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

The link takes you to the Shape method in the Shape class definition.

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.

```
>> codegen -report testAddOne -args {0}
Code generation successful: View report
```

- 4 Click the **View report** link.

- 5 In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.

The screenshot shows the Code Generation Report window with the MATLAB code for the `testAddOne` function. The code is as follows:

```

1 function y = testAddOne(x)
2 %%codegen
3 y = AddOne();
4
5 end

```

The **Variables** tab is selected, showing the following table:

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	x	Input	1 x 1	double	No
3	p	Local	1 x 1	AddOne	-
3.1	<i>isInitialized</i>	Property	1 x 1	logical	-
3.2	<i>isReleased</i>	Property	1 x 1	logical	-
3.3	<i>TunablePropsChanged</i>	Property	1 x 1	logical	-
3.4	<i>inputVarSize1</i>	Property	1 x 8	uint32	No

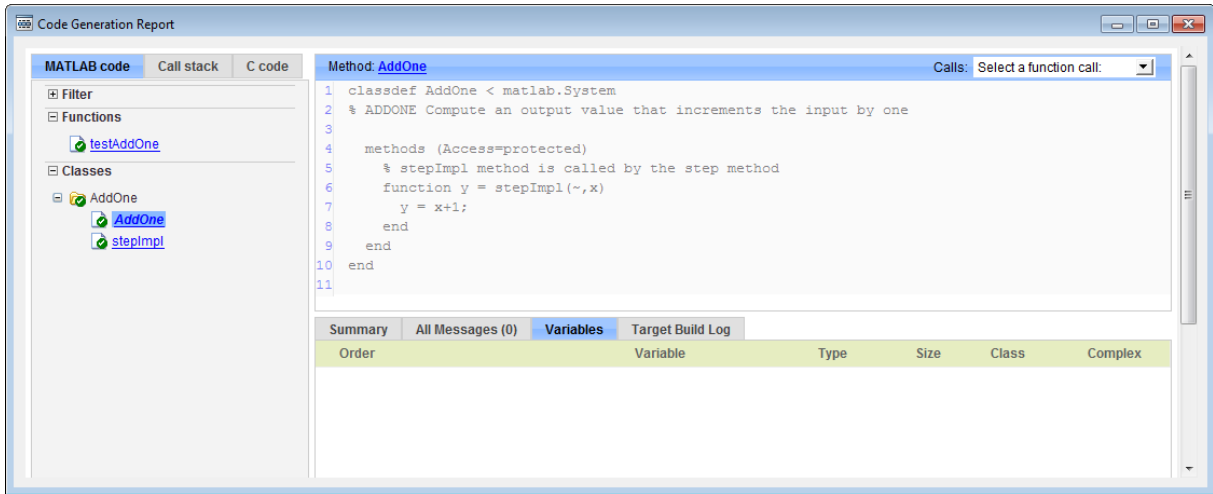
An information tooltip for the selected variable `p` is visible, showing:

```

Information for the selected variable:
Size 1 x 1
Class AddOne

```

- 6 To view the class definition, on the **Classes** panel, click `AddOne`.



The screenshot shows the Code Generation Report window with the following content:

MATLAB code | Call stack | C code

Method: **AddOne** | Calls: Select a function call:

```
1 classdef AddOne < matlab.System
2 % ADDONE Compute an output value that increments the input by one
3
4 methods (Access=protected)
5 % stepImpl method is called by the step method
6 function y = stepImpl(~,x)
7     y = x+1;
8 end
9 end
10 end
11
```

Summary | All Messages (0) | **Variables** | Target Build Log

Order	Variable	Type	Size	Class	Complex
-------	----------	------	------	-------	---------

MATLAB Classes in Code Generation Reports

What Reports Tell You About Classes

Code generation reports:

- Provide a hierarchical tree of the classes used in your MATLAB code.
- Display a list of methods for each class in the MATLAB code tab.
- Display the objects used in your MATLAB code together with their properties on the **Variables** tab.
- Provide a filter so that you can sort methods by class, size, and complexity.
- List the set of calls from and to the selected method in the **Calls** list.

How Classes Appear in Code Generation Reports

In the MATLAB Code Tab

The report displays an alphabetical hierarchical list of the classes used in the your MATLAB code. For each class, you can:

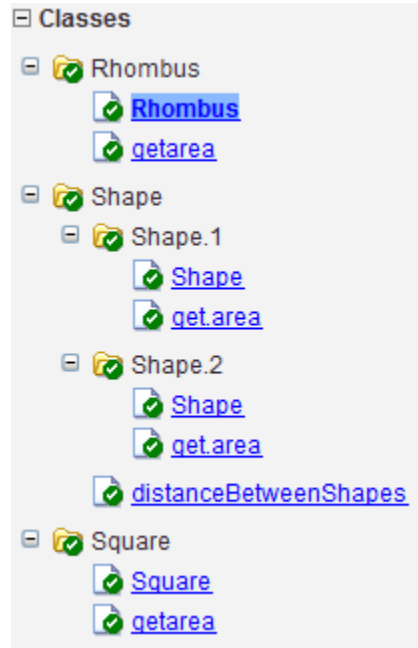
- Expand the class information to view the class methods.
- View a class method by clicking its name. The report displays the methods in the context of the full class definition.
- Filter the methods by size, complexity, and class by using the **Filter functions and methods** option.

Default Constructors. If a class has a default constructor, the report displays the constructor in italics.

Specializations. If the same class is specialized into multiple different classes, the report differentiates the specializations by grouping each one under a single node in the tree. The report associates the class definition functions and static methods with the primary node. It associates the instance-specific methods with the corresponding specialized node.

For example, consider a base class, `Shape` that has two specialized subclasses, `Rhombus` and `Square`. The `Shape` class has an abstract method, `getarea`,

and a static method, `distanceBetweenShapes`. The code generation report, displays a node for the specialized `Rhombus` and `Square` classes with their constructors and `getarea` method. It displays a node for the `Shape` class and its associated static method, `distanceBetweenShapes`, and two instances of the `Shape` class, `Shape1` and `Shape2`.



Packages. If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see “Packages Create Namespaces”.

In the Variables Tab

The report displays the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. Click the + symbol next to the object name to open the list.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a

MATLAB object with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

How to Generate a Code Generation Report

Add the `-report` option to your `codegen` command (requires a MATLAB Coder license)

Troubleshooting Issues with MATLAB Classes

Class *class* does not have a property with name *name*

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.

Workaround

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;
```

Code Generation for Function Handles

- “Function Handle Definition for Code Generation” on page 42-2
- “Define and Pass Function Handles for Code Generation” on page 42-3
- “Function Handle Limitations for Code Generation” on page 42-5

Function Handle Definition for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently. You can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported for code generation (see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2)

Note You cannot define handles that reference extrinsic MATLAB functions.

- Define function handles as scalar values
- Pass function handles as arguments to other functions (excluding extrinsic functions)

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in “Function Handle Limitations for Code Generation” on page 42-5

Define and Pass Function Handles for Code Generation

The following code example shows how to define and call function handles for code generation. You can copy the example to a MATLAB Function block in Simulink or MATLAB function in Stateflow. To convert this function to a MEX function using `codegen`, uncomment the two calls to the `assert` function, highlighted below:

```
function addval(m)
%#codegen

% Define class and size of primary input m
% Uncomment next two lines to build MEX function with codegen
% assert(isa(m,'double'));
% assert(all (size(m) == [3 3]));

% Pass function handle to addone
% to add one to each element of m
m = map(@addone, m);
disp(m);

% Pass function handle to addtwo
% to add two to each element of m
m = map(@addtwo, m);
disp(m);

function y = map(f,m)
    y = m;
    for i = 1:numel(y)
        y(i) = f(y(i));
    end

function y = addone(u)
y = u + 1;

function y = addtwo(u)
y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed

by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

If you have MATLAB Coder, you can use the function `codegen` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

- 1 At the MATLAB command prompt, issue this command:

```
codegen addval
```

- 2 Define and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = zeros(3)
```

- 3 Execute the function by typing this command:

```
addval(m)
```

You should see the following result:

```
0    0    0
0    0    0
0    0    0

1    1    1
1    1    1
1    1    1

3    3    3
3    3    3
3    3    3
```

For more information, see “MEX Function Generation at the Command Line”.

Function Handle Limitations for Code Generation

Function handles must be scalar values.

You cannot store function handles in matrices or structures.

You cannot use the same bound variable to reference different function handles.

After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example

```
%Incorrect code
...
x = @plus;
x = @minus;
...
```

This code produces a compilation error.

You cannot pass function handles to or from extrinsic functions.

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions. For more information, see “Declaring MATLAB Functions as Extrinsic Functions” on page 44-12

You cannot pass function handles to or from primary functions.

You cannot pass function handles as inputs to or outputs from primary 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 primary 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, indicating that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of primary inputs.

You cannot view function handles from the debugger

You cannot display or watch function handles from the debugger. They appear as empty matrices.

Defining Functions for Code Generation

- “Specify Variable Numbers of Arguments” on page 43-2
- “Supported Index Expressions” on page 43-3
- “Apply Operations to a Variable Number of Arguments” on page 43-4
- “Implement Wrapper Functions” on page 43-7
- “Pass Property/Value Pairs” on page 43-8
- “Variable Length Argument Lists for Code Generation” on page 43-10

Specify Variable Numbers of Arguments

You can use `varargin` in a function definition to specify that the function accepts a variable number of input arguments for a given input argument. You can use `varargout` in a function definition to specify that the function returns a variable number of arguments for a given output argument.

When you use `varargin` and `varargout` for code generation, there are the following limitations:

- You cannot use `varargout` in the function definition for a top-level function.
- You cannot use `varargin` in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use `varargin` to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

Common applications of `varargin` and `varargout` for code generation are to:

- “Apply Operations to a Variable Number of Arguments” on page 43-4
- “Implement Wrapper Functions” on page 43-7
- “Pass Property/Value Pairs” on page 43-8

Code generation relies on loop unrolling to produce simple and efficient code for `varargin` and `varargout`. This technique permits most common uses of `varargin` and `varargout`, but some uses are not allowed (see “Variable Length Argument Lists for Code Generation” on page 43-10).

For more information about using `varargin` and `varargout` in MATLAB functions, see [Passing Variable Numbers of Arguments](#).

Supported Index Expressions

In MATLAB, `varargin` and `varargout` are cell arrays. Generated code does not support cell arrays, but does allow you to use the most common syntax — curly braces `{}` — for indexing into `varargin` and `varargout` arrays, as in this example:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i};
end

```

You can use the following index expressions. The *exp* arguments must be constant expressions or depend on a loop index variable.

Expression		Description
<code>varargin</code> (<i>read only</i>)	<code>varargin{exp}</code>	Read the value of element <i>exp</i>
	<code>varargin{exp1: exp2}</code>	Read the values of elements <i>exp1</i> through <i>exp2</i>
	<code>varargin{:}</code>	Read the values of all elements
<code>varargout</code> (<i>read and write</i>)	<code>varargout{exp}</code>	Read or write the value of element <i>exp</i>

Note The use of `()` is not supported for indexing into `varargin` and `varargout` arrays.

Apply Operations to a Variable Number of Arguments

You can use `varargin` and `varargout` in `for`-loops to apply operations to a variable number of arguments. To index into `varargin` and `varargout` arrays in generated code, the value of the loop index variable must be known at compile time. Therefore, during code generation, the compiler attempts to automatically unroll these `for`-loops. Unrolling eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. For example, the following function automatically unrolls its `for`-loop in the generated code:

```

%#codegen
function [cmLen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmLen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

When to Force Loop Unrolling

To automatically unroll `for`-loops containing `varargin` and `varargout` expressions, the relationship between the loop index expression and the index variable must be determined at compile time.

In the following example, the function `fcn` cannot detect a logical relationship between the index expression `j` and the index variable `i`:

```

%#codegen
function [x,y,z] = fcn(a,b,c)

[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;
for i = 1:length(varargin)
    j = j+1;
    varargout{j} = varargin{j};
```

```
end
```

As a result, the function does not unroll the loop and generates a compilation error:

```
Nonconstant expression or empty matrix.
This expression must be constant because
its value determines the size or class of some expression.
```

To fix the problem, you can force loop unrolling by wrapping the loop header in the function `coder.unroll`, as follows:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    j = 0;
    for i = coder.unroll(1:length(varargin))
        j = j + 1;
        varargout{j} = varargin{j};
    end;

```

Using Variable Numbers of Arguments in a for-Loop

The following example multiplies a variable number of input dimensions in inches by 2.54 to convert them to centimeters:

```

%#codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end

```

Key Points About the Example

- `varargin` and `varargout` appear in the local function `inch_2_cm`, not in the top-level function `conv_2_metric`.
- The index into `varargin` and `varargout` is a for-loop variable

For more information, see “Variable Length Argument Lists for Code Generation” on page 43-10.

Implement Wrapper Functions

You can use `varargin` and `varargout` to write wrapper functions that accept up to 64 inputs and pass them directly to another function.

Passing Variable Numbers of Arguments from One Function to Another

The following example passes a variable number of inputs to different optimization functions, based on a specified input method:

```
%#codegen
function answer = fcn(method,a,b,c)
answer = optimize(method,a,b,c);

function answer = optimize(method,varargin)
    if strcmp(method,'simple')
        answer = simple_optimization(varargin{:});
    else
        answer = complex_optimization(varargin{:});
    end
...

```

Key Points About the Example

- You can use `{:}` to read all elements of `varargin` and pass them to another function.
- You can mix variable and fixed numbers of arguments.

For more information, see “Variable Length Argument Lists for Code Generation” on page 43-10.

Pass Property/Value Pairs

You can use `varargin` to pass property/value pairs in functions. However, for code generation, you must take precautions to avoid type mismatch errors when evaluating `varargin` array elements in a `for`-loop:

If	Do This:
You assign <code>varargin</code> array elements to local variables in the <code>for</code> -loop	Verify that for all pairs, the size, type, and complexity are the same for each property and the same for each value
Properties or values have different sizes, types, or complexity	Do not assign <code>varargin</code> array elements to local variables in a <code>for</code> -loop; reference the elements directly

For example, in the following function `test1`, the sizes of the property strings and numeric values are not the same in each pair:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        name = varargin{i};
        value = varargin{i+1};
        switch name
            case 'size'
                v = set_size(v, value);
            case 'rgb'
                v = set_color(v, value);
            otherwise
                % do nothing
        end
    end
end
end

```


...

Generated code determines the size, type, and complexity of a local variable based on its first assignment. In this example, the first assignments occur in the first iteration of the for-loop:

- Defines local variable `name` with size equal to 4
- Defines local variable `value` with a size of scalar

However, in the second iteration, the size of the property string changes to 3 and the size of the numeric value changes to a vector, resulting in a type mismatch error. To avoid such errors, reference `varargin` array values directly, not through local variables, as highlighted in this code:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        switch varargin{i}
            case 'size'
                v = set_size(v, varargin{i+1});
            case 'rgb'
                v = set_color(v, varargin{i+1});
            otherwise
        end
    end
end
...

```

Variable Length Argument Lists for Code Generation

Use variable length argument lists in top-level functions according to guidelines

When you use `varargin` and `varargout` for code generation, there are the following limitations:

- You cannot use `varargout` in the function definition for a top-level function.
- You cannot use `varargin` in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use `varargin` to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

A *top-level function* is:

- The function called by Simulink in a MATLAB Function block or by Stateflow in a MATLAB function.
- The function that you provide on the command line to `codegen` or `fiaccel`.

For example, the following code generates compilation errors:

```
##codegen
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

To fix the problem, write a top-level function that specifies a fixed number of inputs and outputs. Then call `inch_2_cm` as an external function or local function, as in this example:

```
##codegen
function [cmL, cmW, cmH] = conv_2_metric(inL, inW, inH)
[cmL, cmW, cmH] = inch_2_cm(inL, inW, inH);
```

```
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

Use curly braces {} to index into the argument list

For code generation, you can use curly braces {}, but not parentheses (), to index into `varargin` and `varargout` arrays. For more information, see “Supported Index Expressions” on page 43-3.

Verify that indices can be computed at compile time

If you use an expression to index into `varargin` or `varargout`, make sure that the value of the expression can be computed at compile time. For examples, see “Apply Operations to a Variable Number of Arguments” on page 43-4.

Do not write to varargin

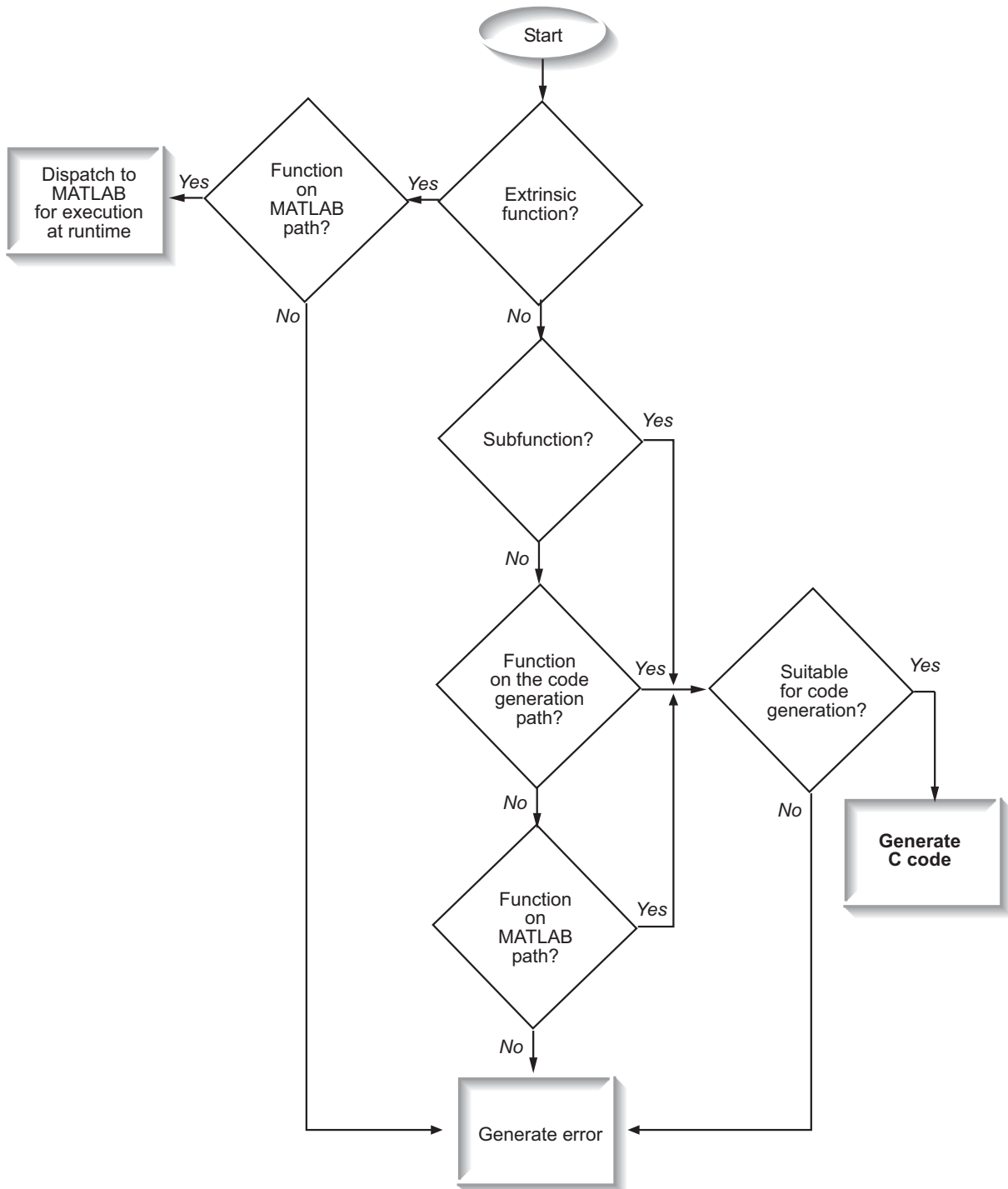
Generated code treats `varargin` as a read-only variable. If you want to write to input arguments, copy the values into a local variable.

Calling Functions for Code Generation

- “Resolution of Function Calls in MATLAB Generated Code” on page 44-2
- “Resolution of File Types on Code Generation Path” on page 44-6
- “Compilation Directive `%#codegen`” on page 44-8
- “Call Local Functions” on page 44-9
- “Call Supported Toolbox Functions” on page 44-10
- “Call MATLAB Functions” on page 44-11

Resolution of Function Calls in MATLAB Generated Code

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:



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 44-4.
- Attempts to compile functions unless the code generation software 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 44-12. During simulation, the code generation software generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 44-6

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.

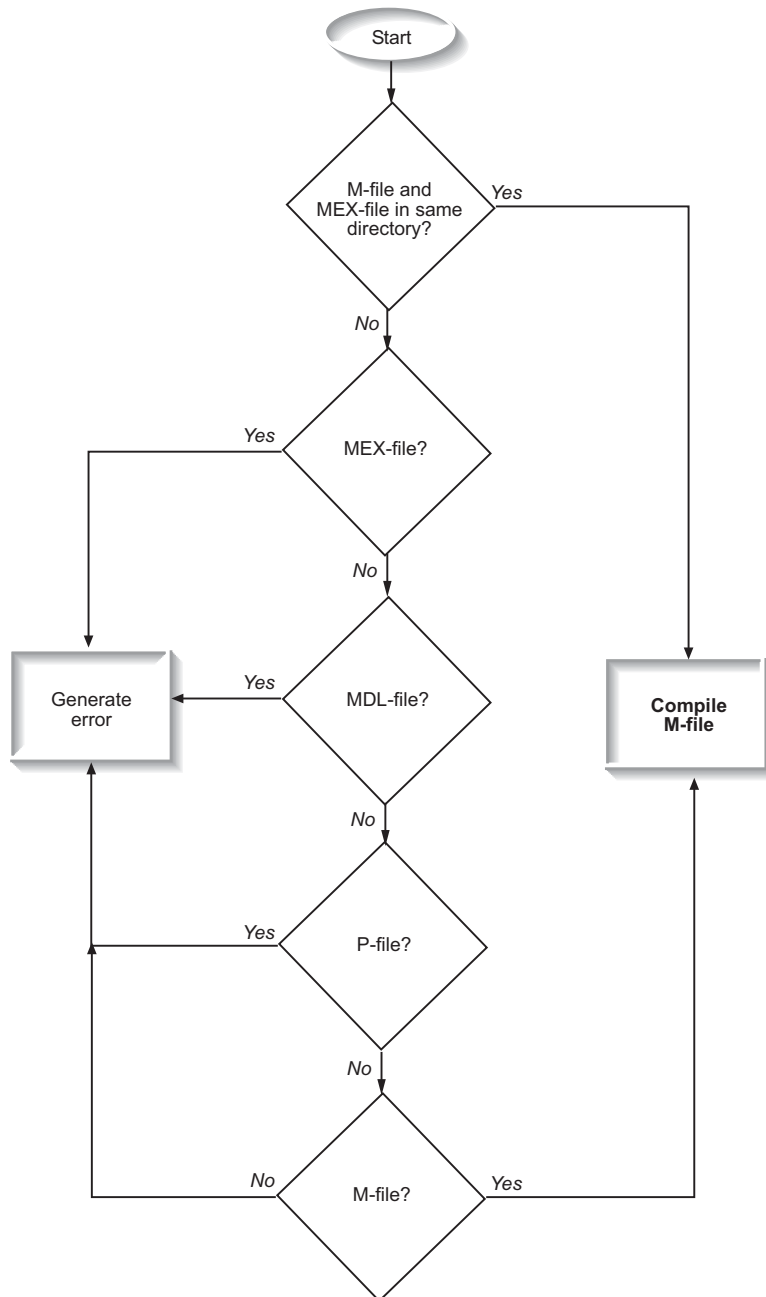
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

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.

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:



Compilation Directive `%#codegen`

Add the `%#codegen` directive (or pragma) to your function 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.

Call Local Functions

Local functions are functions defined in the body of MATLAB function. They work the same way for code generation as they do when executing your algorithm in the MATLAB environment.

The following example illustrates how to define and call a local function mean:

```
function [mean, stdev] = stats(vals)
%#codegen

% Calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals, '-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

Call Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 34-2.

Call MATLAB Functions

The code generation software attempts to generate code for functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using `coder.extrinsic`. During simulation, the code generation software generates code for these functions, but does not generate their internal code. During standalone code generation, MATLAB attempts to determine whether the visualization function affects the output of the function in which it is called. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the visualization function from the generated code. Otherwise, compilation errors occur.

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 generation software dispatches calls to the `plot` function to MATLAB. 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.

The screenshot shows the MATLAB IDE interface. On the left, there is a 'Functions' pane with a filter and a list of functions: 'stats' and 'stats > avg'. The main editor displays the code for the 'stats' function. The code is as follows:

```

1 function [mean, stdev] = stats(vals)
2 %#codegen
3
4 % Calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 len = length(vals);
8 mean = avg(vals, len);
9 stdev = sqrt(sum((vals-avg(vals,len)).^2)/len);
10 plot(vals, '-+');
11

```

A tooltip is visible over the `plot` function call on line 10, stating: "Only supported within the MATLAB environment."

For unsupported functions other than common visualization functions, you must declare the functions (like `pause`) to be extrinsic (see “Resolution of Function Calls in MATLAB Generated Code” on page 44-2). Extrinsic functions are not compiled, but instead executed in MATLAB during

simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 44-16).

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 44-12).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 44-16).

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`:

```
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 and axis as extrinsic

coder.extrinsic('patch');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```


The code generation software detects that `axis` is not supported for code generation and automatically treats it as an extrinsic function. The compiler does not generate code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

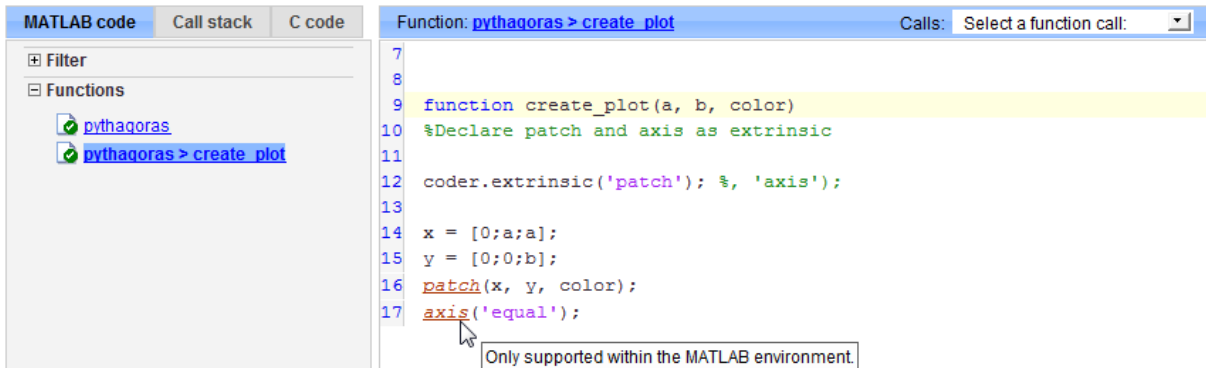
To test the function, follow these steps:

- 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`.

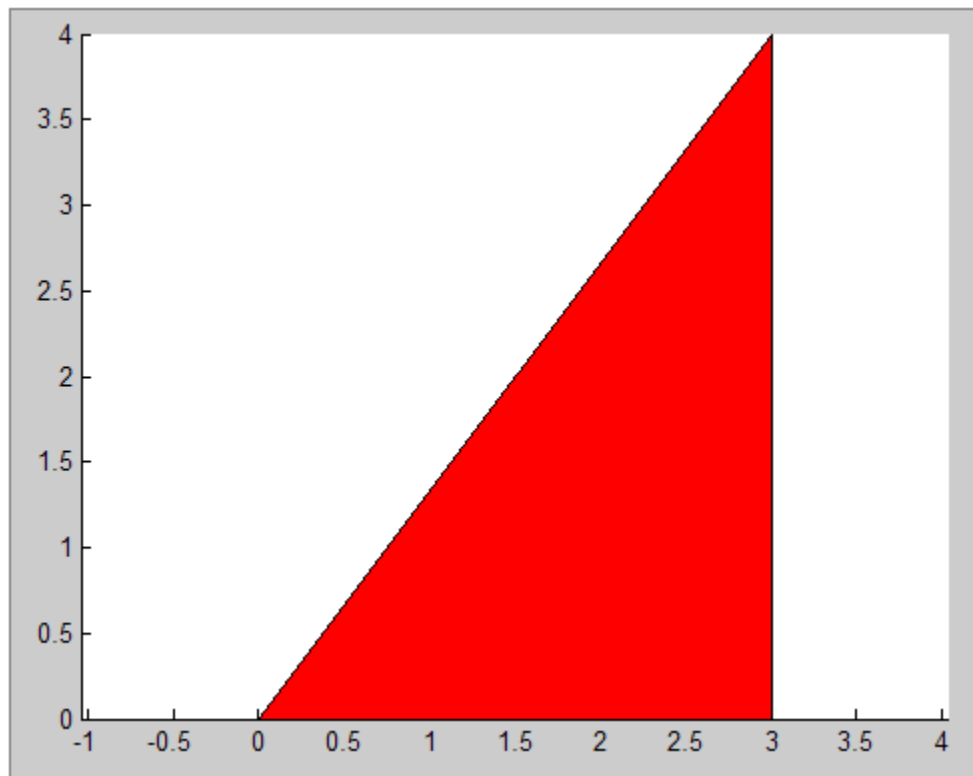
The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.



- 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 — such as `pause` — during simulation, without generating unnecessary code (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 44-16).
- 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 44-17).
- 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 44-15).

- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 44-15). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 44-16).

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` are 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 44-16.

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.

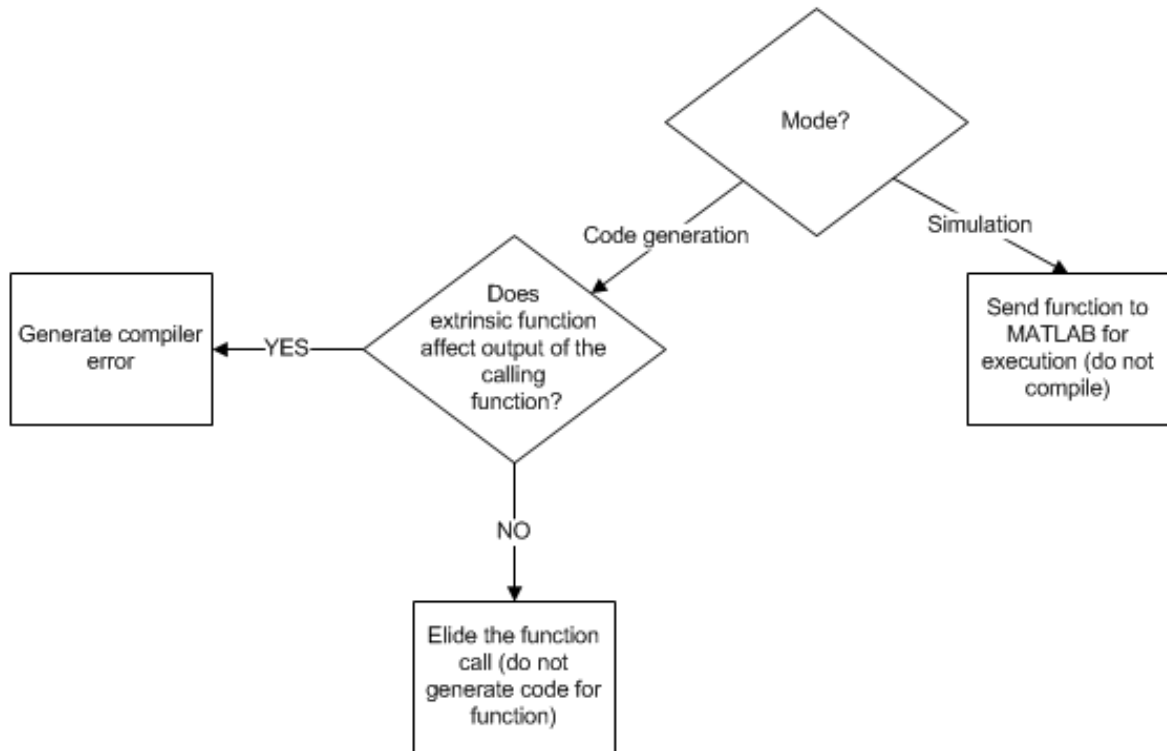
Consider the following example:

```
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`.

How MATLAB Resolves Extrinsic Functions During Simulation

MATLAB resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, MATLAB generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with `mxArrays`” on page 44-17). Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, MATLAB 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 44-18.

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 MATLAB 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's 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 may 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

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Generate Efficient and Reusable Code

- “Optimization Strategies” on page 45-3
- “Modularize MATLAB Code” on page 45-6
- “Eliminate Redundant Copies of Function Inputs” on page 45-7
- “Inline Code” on page 45-9
- “Control Inlining Using Configuration Object” on page 45-11
- “Fold Function Calls into Constants” on page 45-14
- “Control Stack Space Usage” on page 45-16
- “Stack Allocation and Performance” on page 45-17
- “Rewrite Logical Array Indexing as a Loop” on page 45-18
- “Dynamic Memory Allocation and Performance” on page 45-19
- “Minimize Dynamic Memory Allocation” on page 45-21
- “Provide Maximum Size for Variable-size Arrays” on page 45-22
- “Disable Dynamic Memory Allocation During Code Generation” on page 45-29
- “Set Dynamic Memory Allocation Threshold” on page 45-30
- “Excluding Unused Paths from Generated Code” on page 45-33
- “Prevent Code Generation for Unused Execution Paths” on page 45-34
- “Generate Code with Parallel for-loops (parfor)” on page 45-37
- “Minimize Redundant Operations in Loops” on page 45-39

- “Unroll for-Loops” on page 45-41
- “Support for Integer Overflow and Non-Finites” on page 45-44
- “Integrate Custom Code” on page 45-46
- “MATLAB® Coder™ Optimizations in Generated Code” on page 45-52
- “Generate Reusable Code” on page 45-56

Optimization Strategies

MATLAB Coder introduces certain optimizations when generating C/C++ code or MEX functions from your MATLAB code. For more information, see “MATLAB Coder Optimizations in Generated Code”.

To optimize your generated code further, you can:

- Adapt your MATLAB code.
- Control code generation using the configuration object from the command-line or the Project Settings dialog box.

To optimize the execution speed of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have for-loops whose iterations are independent of each other.	“Generate Code with Parallel for-loops (parfor)”
You have variable-size arrays in your MATLAB code.	“Minimize Dynamic Memory Allocation”
You have multiple variable-size arrays in your MATLAB code. You want dynamical memory allocation for larger arrays and static allocation for smaller ones.	“Set Dynamic Memory Allocation Threshold”
You want your generated function to be called by reference.	“Eliminate Redundant Copies of Function Inputs”
You are calling small functions in your MATLAB code.	“Inline Code”
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	“Control Inlining Using Configuration Object”
You do not want to generate code for expressions that contain constants only.	“Fold Function Calls into Constants”

Condition	Action
You have loop operations in your MATLAB code that do not depend on the loop index.	“Minimize Redundant Operations in Loops”
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	“Disable Support for Integer Overflow”
You know beforehand that Inf-s and NaN-s will not occur during execution of your generated code.	“Disable Support for Non-Finites”
You have for-loops with few iterations.	“Unroll for-Loops”
You already have legacy C/C++ code optimized for your target environment.	“Integrate Custom Code”

To optimize the memory usage of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have if/else/elseif statements or switch/case/otherwise statements in your MATLAB code. You do not require some branches of the statements in your generated code.	“Prevent Code Generation for Unused Execution Paths”
You have logical array indexing in your MATLAB code. For more information, see “Using Logicals in Array Indexing”.	“Rewrite Logical Array Indexing as a Loop”
You want your generated function to be called by reference.	“Eliminate Redundant Copies of Function Inputs”
You have limited stack space for your generated code.	“Control Stack Space Usage”
You are calling small functions in your MATLAB code.	“Inline Code”

Condition	Action
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	“Control Inlining Using Configuration Object”
You do not want to generate code for expressions that contain constants only.	“Fold Function Calls into Constants”
You have loop operations in your MATLAB code that do not depend on the loop index.	“Minimize Redundant Operations in Loops”
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	“Disable Support for Integer Overflow”
You know beforehand that Inf-s and NaN-s will not occur during execution of your generated code.	“Disable Support for Non-Finites”

Modularize MATLAB Code

For large MATLAB code, streamline code generation by modularizing the code:

- 1** Break up your MATLAB code into smaller, self-contained sections.
- 2** Save each section in a MATLAB function.
- 3** Generate C/C++ code for each function.
- 4** Call the generated C/C++ functions in sequence from a wrapper MATLAB function using `coder.ceval`.
- 5** Generate C/C++ code for the wrapper function.

Besides streamlining code generation for the original MATLAB code, this approach also supplies you with C/C++ codes for the individual sections. You can reuse these codes later by integrating them with other generated C/C++ code using `coder.ceval`.

Eliminate Redundant Copies of Function Inputs

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, MATLAB passes the variable by reference in the generated code 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 you rewrite function foo without the optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

MATLAB generates code that 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 this 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
```


Inline Code

MATLAB uses internal heuristics to determine whether or not 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`.

In this section...

“Prevent Function Inlining” on page 45-9

“Use Inlining in Control Flow Statements” on page 45-9

Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
    coder.inline('never');
    y = x;
end
```

Use Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
% Vector division produces a for-loop.
```

```
% Prohibit inlining to reduce code size.  
    coder.inline('never');  
end  
  
if any(divisor == 0)  
    error('Can not divide by 0');  
end  
  
y = dividend / divisor;
```

Related Examples

- “Control Inlining Using Configuration Object”

Control Inlining Using Configuration Object

This example shows how to control inlining behavior using the codegen configuration object. Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at a large number of different sites in your code. The generated code can be large due to the function being inlined every time it is called.

The call sites must be different. For instance, inlining does not lead to large code if the function to be inlined is called several times inside a loop.

- You have limited RAM or stack space.

In this section...

“Control Size of Functions Inlined” on page 45-11

“Control Size of Functions After Inlining” on page 45-12

“Control Stack Size Limit on Inlined Functions” on page 45-12

Control Size of Functions Inlined

You can control the maximum size of functions that can be inlined from the Project Settings dialog box or the command line. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold**, to the maximum size that you want.
- At the command line, create a codegen configuration object. Set the value of the property, `InlineThreshold`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineThreshold = 100;
```

Generate code using this configuration object.

Control Size of Functions After Inlining

You can control the maximum size of functions after inlining from the Project Settings dialog box or the command line. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold max**, to the maximum size that you want.
- At the command line, create a codegen configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineThresholdMax = 100;
```

Generate code using this configuration object.

Control Stack Size Limit on Inlined Functions

Specifying a limit on the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even after the function is executed. The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that can be accommodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can control the stack size limit on inlined functions from the Project Settings dialog box or the command line.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline stack limit**, to the maximum size that you want.
- At the command line, create a codegen configuration object. Set the value of the property, `InlineStackLimit`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineStackLimit = 2000;
```

Generate code using this configuration object.

Related Examples

- “Inline Code”

Fold Function Calls into Constants

This example shows how to specify constants in generated code using `coder.const`. The code generation software folds an expression or a function call in a `coder.const` statement into a constant in generated code. Because the generated code does not have to evaluate the expression or call the function every time, this optimization reduces the execution time of the generated code.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software generates code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int i0;
    static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                         49, 64, 81, 100 };

    for (i0 = 0; i0 < 10; i0++) {
        y[i0] = (double)iv0[i0] + Shift;
    }
}
```

See Also `coder.const`

Control Stack Space Usage

This example shows how to set the maximum stack space used by the generated code. Set the maximum stack usage when:

- You have limited stack space, for instance, in case of embedded targets.
- Your C compiler reports a run-time stack overflow.

The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that can be accommodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Control Stack Space Usage Using Project Interface

- 1 On the **Build** tab **Settings** pane, set the **Output type** to `C/C++ Static Library`, `C/C++ Dynamic Library`, or `C/C++ Executable` (depending on your requirements).
- 2 Click the **More settings** link to open the **Project Settings** dialog box.
- 3 On the **Memory** tab, set the field, **Stack usage max**, to the value that you want.

Control Stack Space Usage from Command Line

- 1 Create a configuration object for code generation.

Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```

- 2 Set the property, `StackUsageMax`, to the value that you want.

```
cfg.StackUsageMax=400000;
```

Concepts

- “Stack Allocation and Performance”

Stack Allocation and Performance

By default, local variables are allocated on the stack. Large variables that do not fit on the stack are statically allocated in memory.

Stack allocation typically uses memory more efficiently than static allocation. However, stack space is sometimes limited, typically in embedded processors. MATLAB Coder allows you to manually set a limit on the stack space usage to make your generated code suitable for your target hardware. You can choose this limit based on the target hardware configurations. For more information, see “Control Stack Space Usage”.

Rewrite Logical Array Indexing as a Loop

Rewriting logical array indexing as a loop can optimize the generated code for both speed and readability. For more information on logical array indexing, see “Using Logicals in Array Indexing”.

For example, the MATLAB function, `foo`, uses logical array indexing.

```
function x = foo(x,N) %#codegen
assert(all(size(x) == [1 100]))
x(x>N) = N;
```

The generated C code for this function is not very efficient. Rewrite the MATLAB code to use a loop instead of logical indexing:

```
function x = foo_rewrite(x,N) %#codegen
assert(all(size(x) == [1 100]))
for ii=1:numel(x)
    if x(ii) > N
        x(ii) = N;
    end
end
```

Dynamic Memory Allocation and Performance

To achieve faster execution of generated code, minimize dynamic (or run-time) memory allocation of arrays.

MATLAB Coder does not provide a size for unbounded arrays in generated code. Instead, such arrays are referenced indirectly through pointers. For such arrays, memory cannot be allocated during compilation of generated code. Based on storage requirements for the arrays, memory is allocated and freed at run time as required. This run-time allocation and freeing of memory leads to slower execution of the generated code. For more information on dynamic memory allocation, see “Bounded Versus Unbounded Variable-Size Data”.

When Dynamic Memory Allocation Occurs

Dynamic memory allocation occurs when the code generation software cannot find upper bounds for variable-size arrays. The software cannot find upper bounds when you specify the size of an array using a variable that is not a compile-time constant. An example of such a variable is an input variable (or a variable computed from an input variable).

Instances in the MATLAB code that might lead to dynamic memory allocation are:

- Array initialization: You specify array size using a variable whose value is known only at run time.
- After initialization of an array:
 - You declare the array as variable-size using `coder.varsize` without explicit upper bounds. After this declaration, you expand the array by concatenation inside a loop. The number of loop runs is known only at run time.
 - You use a `reshape` function on the array. At least one of the size arguments to the `reshape` function is known only at run time.

If you know the maximum possible size of the array, you can avoid dynamic memory allocation. You can then provide an upper bound for the array and

prevent dynamic memory allocation in generated code. For more information, see “Minimize Dynamic Memory Allocation” on page 45-21.

Minimize Dynamic Memory Allocation

When possible, you should minimize dynamic memory allocation since it leads to slower execution of generated code. Dynamic memory allocation occurs when the code generation software cannot find upper bounds for variable-size arrays.

You can avoid dynamic memory allocation of a variable-size array if you know its maximum possible size. To do so, follow these steps:

- 1 “Provide Maximum Size for Variable-size Arrays” on page 45-22.
- 2 Depending on your requirements, do one of the following:
 - “Disable Dynamic Memory Allocation During Code Generation” on page 45-29.
 - “Set Dynamic Memory Allocation Threshold”

Caution If a variable-size array in the MATLAB code does not have a maximum size, disabling dynamic memory allocation leads to a code generation error. Before disabling dynamic memory allocation, you must provide a maximum size for variable-size arrays in your MATLAB code.

Concepts

- “Dynamic Memory Allocation and Performance”

Provide Maximum Size for Variable-size Arrays

To constrain array size for variable-size arrays, do one of the following:

-

Constrain Array Size Using `assert` Statements

If the variable specifying array size is not a compile-time constant, use an `assert` statement with relational operators to constrain the variable. Doing so helps the code generation software to determine a maximum size for the array. An array of size equal to this maximum size is then defined in the generated code (static memory allocation).

The following examples constrain array size using `assert` statements:

-

When Array Size Is Specified by Input Variables

Define a function `array_init` which initializes an array `y` with input variable `N`:

```
function y = array_init (N)
    assert(N < 25); % Generates exception if N >= 25
    y = zeros(1,N);
```

The `assert` statement ensures that `y` is assigned an array of size 25 in the generated code. In the absence of the `assert` statement, `y` is assigned a pointer to an array in the generated code, thus allowing dynamic memory allocation.

-

When Array Size Is Obtained from Computation Using Input Variables

Define a function, `array_init_from_prod`, which takes two input variables, `M` and `N`, and uses their product to specify the size of an array, `y`.

The following code restricts the product of M and N to 25. It then uses this product to specify size of an array, y..

```
function y = array_init_from_prod (M,N)
    size=M*N;
    assert(size < 25); % Generates exception if size >= 25
    y=zeros(1,size);
```

The assert statement ensures that y is assigned an array of size 25 in the generated code.

Alternatively, if you restrict M and N individually, it leads to dynamic memory allocation:

```
function y = array_init_from_prod (M,N)
    assert(M < 5);
    assert(N < 5);
    size=M*N;
    y=zeros(1,size);
```

This code causes dynamic memory allocation because M and N can both have unbounded negative values. Therefore, their product can be unbounded and positive even though, individually, their positive values are bounded.

Tip Place the assert statement on a variable immediately before it is used to specify array size.

Tip You can use assert statements to restrict array sizes in most cases. When expanding an array inside a loop, this strategy does not work if the number of loop runs is known only at run time.

•

Restrict Concatenations in a Loop Using `coder.varsize` with Upper Bounds

You can expand arrays beyond their initial size by concatenation. When you concatenate additional elements inside a loop, there are two syntax rules for expanding arrays.

1

Array size during initialization is not a compile-time constant

If the size of an array during initialization is not a compile-time constant, you can expand it by concatenating additional elements:

```
function out=ExpandArray(in) % Expand an array by five elements
    out = zeros(1,in);
    for i=1:5
        out = [out 0];
    end
```

2

Array size during initialization is a compile-time constant

Before concatenating elements, you have to declare the array as variable-size using `coder.varsize`:

```
function out=ExpandArray() % Expand an array by five elements
    out = zeros(1,5);
    coder.varsize('out');
    for i=1:5
        out = [out 0];
    end
```

Either case leads to dynamic memory allocation. To prevent dynamic memory allocation in such cases, use `coder.varsize` with explicit upper bounds. This example shows how to use `coder.varsize` with explicit upper bounds:

Restrict Concatenations Using `coder.varsize` with Upper Bounds

- 1 Define a function, `RunningAverage`, that calculates the running average of an N-element subset of an array:

```
function avg=RunningAverage(N)

% Array whose elements are to be averaged
NumArray=[1 6 8 2 5 3];

% Initialize average:
% These will also be the first two elements of the function output
avg=[0 0];

% Place a bound on the argument
coder.varsize('avg',[1 8]);

% Loop to calculate running average
for i=1:N
    s=0;
    s=s+sum(NumArray(1:i));
    avg=[avg s/i];
    % Increase the size of avg as required by concatenation
end
```

The output, `avg`, is an array that you can expand as required to accommodate the running averages. As a new running average is calculated, it is added to the array `avg` through concatenation, thereby expanding the array.

Because the maximum possible number of running averages is equal to the number of elements in `NumArray`, you can supply an explicit upper bound for `avg` in the `coder.varsize` statement. In this example, the upper bound is 8 (the two initial elements plus the six elements of `NumArray`).

- 2 Generate code for `RunningAverage` with input argument of type double:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned an array of size 8 (static memory allocation). The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void RunningAverage (double N, double avg_data[8], int avg_size[2])
```

- 3** By contrast, if you remove the explicit upper bound, the generated code dynamically allocates `avg`.

Replace the statement

```
coder.varsized('avg',[1 8]);
```

with:

```
coder.varsized('avg');
```

- 4** Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned a pointer to an array, thereby allowing dynamic memory allocation. The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void Test(double N, mxArray_real_T *avg)
```

Note Dynamic memory allocation also occurs if you precede `coder.varsized('avg')` with the following `assert` statement:

```
assert(N < 6);
```

The `assert` statement does not restrict the number of concatenations within the loop.

•

Constrain Array Size When Rearranging a Matrix

The statement `out = reshape(in,m,n,...)` takes an array, `in`, as an argument and returns array, `out`, having the same elements as `in`, but reshaped as an `m-by-n-by-...` matrix. If one of the size variables `m,n,...` is not a compile-time constant, then dynamic memory allocation of `out` takes place.

To avoid dynamic memory allocation, use an `assert` statement before the `reshape` statement to restrict the size variables `m,n,...` to `numel(in)`. This example shows how to use an `assert` statement before a `reshape` statement:

Rearrange a Matrix into Given Number of Rows

- 1 Define a function, `ReshapeMatrix`, which takes an input variable, `N`, and reshapes a matrix, `mat`, to have `N` rows:

```
function [out1,out2] = ReshapeMatrix(N)

    mat = [1 2 3 4 5; 4 5 6 7 8]
    % Since mat has 10 elements, N must be a factor of 10
    % to pass as argument to reshape

    out1 = reshape(mat,N,[]);
    % N is not restricted

    assert(N < numel(mat));
    % N is restricted to number of elements in mat
    out2 = reshape(mat,N,[]);
```

- 2 Generate code for `ReshapeArray` using the `codegen` command (the input argument does not have to be a factor of 10):

```
codegen -config:lib -report ReshapeArray -args 3
```

While `out1` is dynamically allocated, `out2` is assigned an array with size 100 (=10 X 10) in the generated code.

Tip If your system has limited memory, do not use the `assert` statement in this way. For an n -element matrix, the `assert` statement creates an n -by- n matrix, which might be large.

Related Examples

- “Minimize Dynamic Memory Allocation”
- “Disable Dynamic Memory Allocation During Code Generation”
- “Set Dynamic Memory Allocation Threshold”

Concepts

- “Dynamic Memory Allocation and Performance”

Disable Dynamic Memory Allocation During Code Generation

Disabling dynamic memory allocation during code generation leads to faster execution of generated code. You can disable dynamic memory allocation explicitly from the project settings dialog box or the command line.

To disable dynamic memory allocation in the Project Settings box :

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.
- 2 In the **Project Settings** dialog box **Memory** tab, under **Enable variable-sizing**, set **Dynamic memory allocation** to **Never**.

To disable dynamic memory allocation from the command line:

- 1 In the MATLAB workspace, define the configuration object:

```
cfg=coder.config('lib');
```

- 2 Set the `DynamicMemoryAllocation` property of the configuration object to `Off`:

```
cfg.DynamicMemoryAllocation = 'Off';
```

Disabling dynamic memory allocation leads to a code generation error if a variable-size array in the MATLAB code does not have a maximum upper bound. Therefore, you can also use this feature to identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the ones that are dynamically allocated in the generated code.

Related Examples

- “Minimize Dynamic Memory Allocation”
- “Provide Maximum Size for Variable-size Arrays”
- “Set Dynamic Memory Allocation Threshold”

Concepts

- “Dynamic Memory Allocation and Performance”

Set Dynamic Memory Allocation Threshold

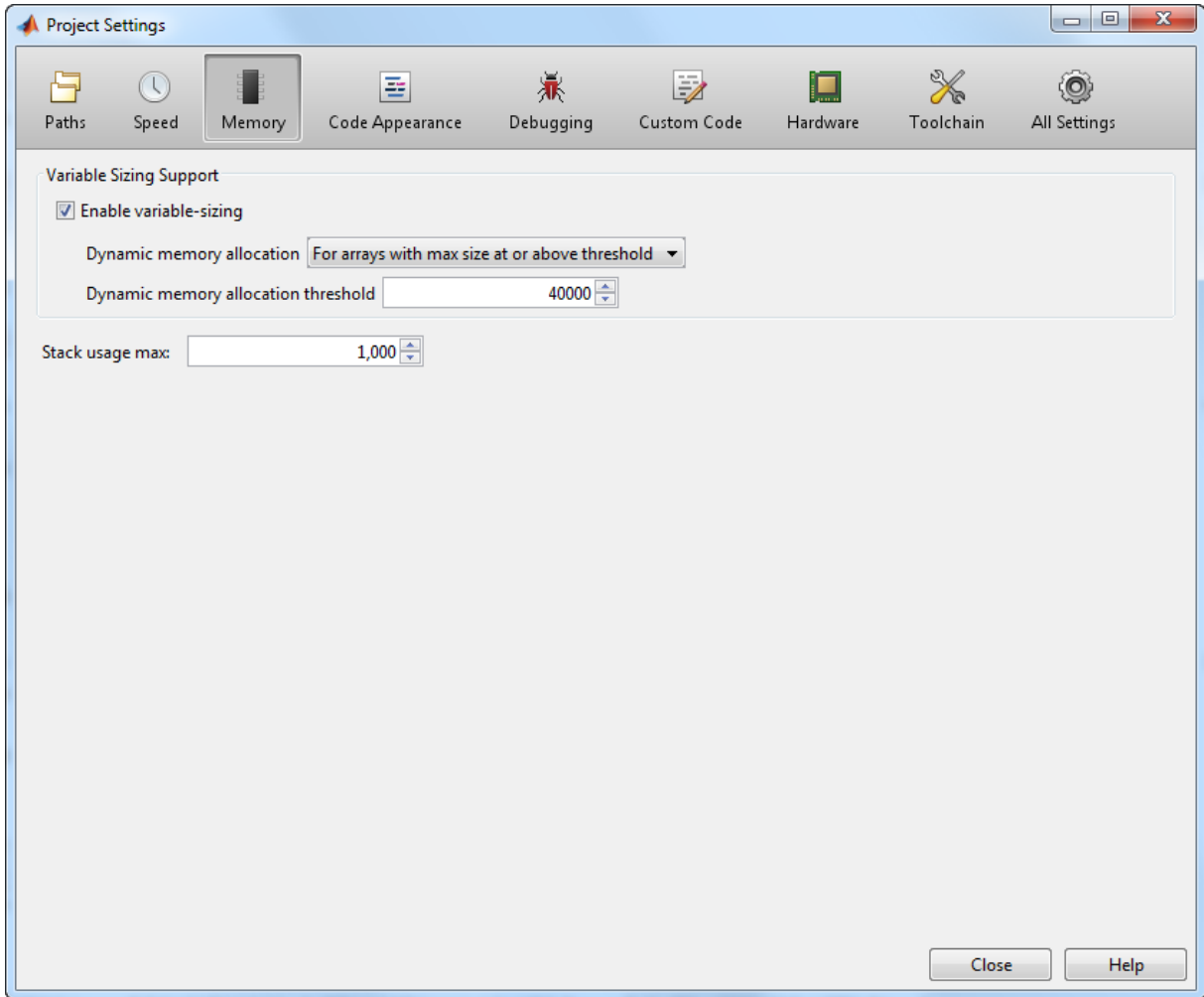
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-size arrays, you can disable it only for arrays below a certain size. Set a dynamic memory allocation threshold to disable dynamic memory allocation for array size below the threshold and enable it for array size at or above the threshold.

Use this strategy when you want to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, it can be more efficient to speed up generated code by allocating memory statically. Though static memory allocation can lead to unused storage space, it might not be a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, you can reduce storage requirements significantly using dynamic memory allocation.

Set Dynamic Memory Allocation Threshold Using Project Interface

- 1 On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
- 2 On the **Memory** tab, select **Enable variable-sizing**.
- 3 On the same tab, select the **For arrays with max size at or above threshold** option in the **Dynamic memory allocation** list.
- 4 Set the **Dynamic memory allocation threshold** to the value that you want.



The **Dynamic memory allocation threshold** value is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that can be accommodated by a certain value of `DynamicMemoryAllocationThreshold`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Set Dynamic Memory Allocation Threshold from Command Line

- 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```

- 2 Set the property, `DynamicMemoryAllocation`, to `'Threshold'`.

```
cfg.DynamicMemoryAllocation='Threshold';
```

- 3 Set the property, `DynamicMemoryAllocationThreshold`, to the value that you want.

```
cfg.DynamicMemoryAllocationThreshold = 40000;
```

The value stored in `DynamicMemoryAllocationThreshold` is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that can be accommodated by a certain value of `DynamicMemoryAllocationThreshold`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Related Examples

- “Minimize Dynamic Memory Allocation”
- “Provide Maximum Size for Variable-size Arrays”
- “Disable Dynamic Memory Allocation During Code Generation”

Concepts

- “Dynamic Memory Allocation and Performance”

Excluding Unused Paths from Generated Code

In certain situations, you do not need some branches of an `if/elseif/if` statement or a `switch/case/otherwise` statement in your generated code. For instance :

- You have a MATLAB function that performs multiple tasks determined by a control-flow variable. You might not need some of the tasks in the code generated from this function.
- You have an `if/elseif/if` statement in a MATLAB function performing different tasks based on the nature (type/value) of the input. In some cases, you know the nature of the input beforehand. If so, you do not need some branches of the `if` statement.

You can prevent code generation for the unused branches of an `if/elseif/else` statement or a `switch/case/otherwise` statement. Declare the control-flow variable as a constant. The code generation software generates code only for the branch that is chosen by the control-flow variable.

Related Examples

- “Prevent Code Generation for Unused Execution Paths”

Prevent Code Generation for Unused Execution Paths

In this section...

“Prevent Code Generation When Local Variable Controls Flow” on page 45-34

“Prevent Code Generation When Input Variable Controls Flow” on page 45-35

If a variable controls the flow of an `if/elseif/if` statement or a `switch/case/otherwise` statement, declare it as constant so that code generation takes place for one branch of the statement only.

Depending on the nature of the control-flow variable, you can declare it as constant in two ways:

- If the variable is local to the MATLAB function, assign it to a constant value in the MATLAB code. For an example, see “Prevent Code Generation When Local Variable Controls Flow” on page 45-34.
- If the variable is an input to the MATLAB function, you can declare it as constant using `coder.Constant`. For an example, see “Prevent Code Generation When Input Variable Controls Flow” on page 45-35.

Prevent Code Generation When Local Variable Controls Flow

- 1 Define a function `SquareOrCube` which takes an input variable, `in`, and squares or cubes its elements based on whether the choice variable, `ch`, is set to `s` or `c`:

```
function out = SquareOrCube(ch,in) %#codegen
    if ch=='s'
        out = in.^2;
    elseif ch=='c'
        out = in.^3;
    else
        out = 0;
    end
```

- 2** Generate code for `SquareOrCube` using the `codegen` command:

```
codegen -config:lib SquareOrCube -args {'s',zeros(2,2)}
```

The generated C code squares or cubes the elements of a 2-by-2 matrix based on the input for `ch`.

- 3** Add the following line to the definition of `SquareOrCube`:

```
ch = 's';
```

The generated C code squares the elements of a 2-by-2 matrix. The choice variable, `ch`, and the other branches of the `if/elseif/if` statement do not appear in the generated code.

Prevent Code Generation When Input Variable Controls Flow

- 1** Define a function `MathFunc`, which performs different mathematical operations on an input, `in`, depending on the value of the input, `flag`:

```
function out = MathFunc(flag,in) %#codegen
    %# codegen
    switch flag
        case 1
            out=sin(in);
        case 2
            out=cos(in);
        otherwise
            out=sqrt(in);
    end
```

- 2** Generate code for `MathFunc` using the `codegen` command:

```
codegen -config:lib MathFunc -args {1,zeros(2,2)}
```

The generated C code performs different math operations on the elements of a 2-by-2 matrix based on the input for `ch`.

- 3** Generate code for `MathFunc`, declaring the argument, `flag`, as a constant using `coder.Constant`:

```
codegen -config:lib MathFunc -args {coder.Constant(1),zeros(2,2)}
```

The generated C code finds the sine of the elements of a 2-by-2 matrix. The variable, `flag`, and the `switch/case/otherwise` statement do not appear in the generated code.

Concepts

- “Excluding Unused Paths from Generated Code”

Generate Code with Parallel for-loops (parfor)

This example shows how to generate C code for a MATLAB algorithm that contains a parfor-loop.

- 1 Write a MATLAB function that contains a parfor-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 Generate C code for test_parfor. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor
```

Because you did not specify the maximum number of threads to use, the generated C code executes the loop iterations in parallel on the available number of cores.

- 3 To specify a maximum number of threads, rewrite the function test_parfor as follows:

```
function a = test_parfor(u) %#codegen
a=ones(10,256);
r=rand(10,256);
parfor (i=1:10,u)
    a(i,:)=real(fft(r(i,:)));
end
```

- 4 Generate C code for test_parfor. Use -args 0 to specify that the input, u, is a scalar double. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor -args 0
```

In the generated code, the iterations of the parfor-loop run on at most the number of cores specified by the input, u. If less than u cores are available, the iterations run on the cores available at the time of the call.

Concepts

- “Algorithm Acceleration Using Parallel for-loops (parfor)”
- “Classification of Variables in parfor-loops”
- “Reduction Assignments in parfor-loops”

Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix B is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
    C=C + inv(B)*A^i*B;
end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

- 1 Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of n terms in the following power series expansion:

$$C = 1 + B^{-1}AB + B^{-1}A^2B + \dots$$

```
function C=SeriesFunc(A,B,n)
```

```
% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));
```

```
% Perform the series sum
for i=1:n
    C=C+inv(B)*A^i*B;
end
```

- 2 Generate code for `SeriesFunc` with 4-by-4 matrices passed as input arguments for A and B:

```
X = coder.typeof(zeros(4));
codegen -config:lib -launchreport SeriesFunc -args {X,X,10}
```

In the generated code, the inversion of **B** is performed n times inside the loop. It is more economical to perform the inversion operation once outside the loop because it does not depend on the loop index.

3 Modify `SeriesFunc` as follows:

```
function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));

% Perform the inversion outside the loop
inv_B=inv(B);

% Perform the series sum
for i=1:n
    C=C+inv_B*A^i*B;
end
```

This procedure performs the inversion of **B** only once, leading to faster execution of the generated code.

Unroll for-Loops

Unrolling for-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant.

You can also force loop unrolling for individual functions by wrapping the loop header in a `coder.unroll` function. For more information, see `coder.unroll`.

Limit Copying the for-loop Body in Generated Code

To limit the number of times that you copy the body of a for-loop in generated code:

- 1 Write a MATLAB function `getrand(n)` that uses a for-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the for-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Calling getrand 8 times triggers unroll
y1 = getrand(8);
% Calling getrand 50 times does not trigger unroll
y2 = getrand(50);
```

```
function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
coder.inline('never');

% Set flag variable dounroll to repeat loop body
% only for fewer than 10 iterations
dounroll = n < 10;
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
```

```
for i = coder.unroll(1:2:n, dounroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
% Loop body ends
```

- 2** In the default output folder, `codegen/lib/test_unroll`, generate C static library code for `test_unroll` :

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the `for`-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void getrand(double y[8])
{
    /* Turn off inlining to make */
    /* generated code easier to read */
    /* Set flag variable dounroll to repeat loop body */
    /* only for fewer than 10 iterations */
    /* Declare size, class, and complexity */
    /* of variable y by assignment */
    memset(&y[0], 0, sizeof(double) << 3);

    /* Loop body begins */
    y[2] = b_rand();
    y[4] = b_rand();

    /* Loop body ends */
}
```

The generated C code for `getrand(50)` does not unroll the `for`-loop because the number of iterations is greater than 10:

```
static void b_getrand(double y[50])
{
    int i;
    int b_i;
```

```
/* Turn off inlining to make */
/* generated code easier to read */
/* Set flag variable dounroll to repeat loop body */
/* only for fewer than 10 iterations */
/* Declare size, class, and complexity */
/* of variable y by assignment */
memset(&y[0], 0, 50U * sizeof(double));

/* Loop body begins */
for (i = 0; i < 25; i++) {
    b_i = (i << 1) + 1;
    if ((b_i > 2) && (b_i < 48)) {
        y[b_i - 1] = b_rand();
    }
}
```

Support for Integer Overflow and Non-Finites

In addition to code generated for your MATLAB function, the code-generation software generates supporting code for the following situations:

- The result of an integer operation falls outside the range that a data type can represent. This situation is known as integer overflow.
- Non-finite values (`inf` and `NaN`) are generated from an operation. The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c` and `rtGetNaN.c` (with corresponding header files).

You can suppress generation of the supporting code if you know beforehand that such situations will not arise. This action reduces the size and increases the speed of generated code at the cost of potentially producing results that do not match simulation in case the situations arise.

Disable Support for Integer Overflow

You can disable support for integer overflow in the project settings dialog box or at the command line. On disabling this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.

- In the project settings dialog box:
 - 1 On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
 - 2 To disable support for integer overflow, on the **Speed** tab, clear **Saturate on integer overflow**.
- At the command line:
 - 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```
 - 2 To disable support for integer overflow, set the `SaturateOnIntegerOverflow` property to `false`.

```
cfg.SaturateOnIntegerOverflow = false;
```

Disable Support for Non-Finites

You can disable support for non-finites (inf and NaN) in the project settings dialog box or at the command line.

- In the project settings dialog box:
 - 1** On the **Build** tab **Settings** pane, set the **Output type** to C/C++ Static Library, C/C++ Dynamic Library, or C/C++ Executable (depending on your requirements).
 - 2** Click the **More settings** link to open the **Project Settings** dialog box.
 - 3** To disable support for integer overflow, on the **Speed** tab, clear **Support non-finite numbers**.

- At the command line:

- 1** Create a configuration object for code generation. Use `coder.config` with arguments 'lib', 'dll' or 'exe' (depending on your requirements). For example:

```
cfg = coder.config('lib');
```

- 2** To disable support for integer overflow, set the `SupportNonFinite` property to `false`.

```
cfg.SupportNonFinite = false;
```

Integrate Custom Code

This example shows how to integrate custom code to enhance performance of generated code. Although MATLAB Coder generates optimized code for most applications, you might have legacy code optimized for your specific requirements. For example:

- You have custom libraries optimized for your target environment.
- You have custom libraries for functions not supported by MATLAB Coder.
- You have custom libraries that meet standards set by your company.

In such cases, you can integrate your custom code with the code generated by MATLAB Coder.

This example illustrates how to integrate the function `cublasSgemm` from the NVIDIA® CUDA® Basic Linear Algebra Subroutines (CUBLAS) library in generated code. This function performs matrix multiplication on a Graphics Processing Unit (GPU).

1 Define a class `ExternalLib_API` that derives from the class `coder.ExternalDependency`. `ExternalLib_API` defines an interface to the CUBLAS library through the following methods:

- `getDescriptiveName`: Returns a descriptive name for `ExternalLib_API` to be used for error messages.
- `isSupportedContext`: Determines if the build context supports the CUBLAS library.
- `updateBuildInfo`: Adds header file paths and link files to the build information.
- `GPU_MatrixMultiply`: Defines the interface to the CUBLAS library function `cublasSgemm`.

ExternalLib_API.m

```
classdef ExternalLib_API < coder.ExternalDependency
    %#codegen

    methods (Static)
```

```
function bName = getDescriptiveName(~)
    bName = 'ExternalLib_API';
end

function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('CUBLAS library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, ~, ~] = ctx.getStdLibInfo();

    % Include header file path
    % Include header files later using coder.cinclude
    hdrFilePath = 'C:\My_Includes';
    buildInfo.addIncludePaths(hdrFilePath);

    % Include link files
    linkFiles = strcat('libcublas', linkLibExt);
    linkPath = 'C:\My_Libs';
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    linkFiles = strcat('libcudart', linkLibExt);
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

end

%API for library function 'cuda_MatrixMultiply'
function C = GPU_MatrixMultiply(A, B)
    assert(isa(A,'single'), 'A must be single.');
```

```
assert(isa(B,'single'), 'B must be single.');
```

```
if(coder.target('MATLAB'))  
    C=A*B;  
else  
  
    % Include header files  
    %     for external functions and typedefs  
    % Header path included earlier using updateBuildInfo  
    coder.cinclude('"cuda_runtime.h"');  
    coder.cinclude('"cublas_v2.h"');
```

```
    % Compute dimensions of input matrices  
    m = int32(size(A, 1));  
    k = int32(size(A, 2));  
    n = int32(size(B, 2));  
  
    % Declare pointers to matrices on destination GPU  
    d_A = coder.opaque('float*');  
    d_B = coder.opaque('float*');  
    d_C = coder.opaque('float*');
```

```
    % Compute memory to be allocated for matrices  
    % Single = 4 bytes  
    size_A = m*k*4;  
    size_B = k*n*4;  
    size_C = m*n*4;
```

```
    % Define error variables  
    error = coder.opaque('cudaError_t');  
    cudaSuccessV = coder.opaque('cudaError_t', ...  
        'cudaSuccess');
```

```
    % Assign memory on destination GPU  
    error = coder.ceval('cudaMalloc', ...  
        coder.wref(d_A), size_A);  
    assert(error == cudaSuccessV, ...  
        'cudaMalloc(A) failed');  
    error = coder.ceval('cudaMalloc', ...  
        coder.wref(d_B), size_B);
```



```
assert(error == cudaSuccessV, ...
        'cudaMalloc(B) failed');
error = coder.ceval('cudaMalloc', ...
                   coder.wref(d_C), size_C);
assert(error == cudaSuccessV, ...
        'cudaMalloc(C) failed');

% Define direction of copying
hostToDevice = coder.opaque('cudaMemcpyKind', ...
                             'cudaMemcpyHostToDevice');

% Copy matrices to destination GPU
error = coder.ceval('cudaMemcpy', ...
                   d_A, coder.rref(A), size_A, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(A) failed');

error = coder.ceval('cudaMemcpy', ...
                   d_B, coder.rref(B), size_B, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(B) failed');

% Define type and size for result
C = zeros(m, n, 'single');

error = coder.ceval('cudaMemcpy', ...
                   d_C, coder.rref(C), size_C, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

% Define handle variables for external library
handle = coder.opaque('cublasHandle_t');
blasSuccess = coder.opaque('cublasStatus_t', ...
                            'CUBLAS_STATUS_SUCCESS');

% Initialize external library
ret = coder.opaque('cublasStatus_t');
ret = coder.ceval('cublasCreate', coder.wref(handle));
assert(ret == blasSuccess, 'cublasCreate failed');

TRANSA = coder.opaque('cublasOperation_t', ...
                      'CUBLAS_OP_N');
```

```

        alpha = single(1);
        beta = single(0);

        % Multiply matrices on GPU
        ret = coder.ceval('cublasSgemm', handle, ...
            TRANSA,TRANSA,m,n,k, ...
            coder.rref(alpha),d_A,m, ...
            d_B,k, ...
            coder.rref(beta),d_C,k);

        assert(ret == blasSuccess, 'cublasSgemm failed');

        % Copy result back to local host
        deviceToHost = coder.opaque('cudaMemcpyKind', ...
            'cudaMemcpyDeviceToHost');
        error = coder.ceval('cudaMemcpy', coder.wref(C), ...
            d_C, size_C, deviceToHost);
        assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

    end
end
end
end
end

```

- 2** To perform the matrix multiplication using the interface defined in method `GPU_MatrixMultiply` and the build information in `ExternalLib_API`, include the following line in your MATLAB code:

```
C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

For instance, you can define a MATLAB function `Matrix_Multiply` that solely performs this matrix multiplication.

```
function C = Matrix_Multiply(A, B) %#codegen
    C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

- 3** Define a MEX configuration object using `coder.config`. For using the CUBLAS libraries, set the target language for code generation to C++.

```
cfg=coder.config('mex');
cfg.TargetLang='C++';
```

- 4 Generate code for `Matrix_Multiply` using `cfg` as the configuration object and two 2 X 2 matrices of type `single` as arguments. Since `cublasSgemv` supports matrix multiplication for data type `float`, the corresponding MATLAB matrices must have type `single`.

```
codegen -config cfg Matrix_Multiply ...  
        -args {ones(2,'single'),ones(2,'single')}
```

- 5 Test the generated MEX function `Matrix_Multiply_mex` using two 2 X 2 identity matrices of type `single`.

```
Matrix_Multiply_mex(eye(2,'single'),eye(2,'single'))
```

The output is also a 2 X 2 identity matrix.

See Also

[coder.ceval](#) | [coder.opaque](#) | [coder.rref](#) | [coder.wref](#) |
[assertcoder.ExternalDependency](#) | [coder.BuildConfig](#) |

Related Examples

- “Encapsulate Interface to an External C Library”

Concepts

- “Encapsulating the Interface to External Code”

MATLAB Coder Optimizations in Generated Code

In this section...

“Constant Folding” on page 45-52

“Loop Fusion” on page 45-53

“Successive Matrix Operations Combined” on page 45-54

“Unreachable Code Elimination” on page 45-54

In order to improve the execution speed and memory usage of generated code, MATLAB Coder introduces the following optimizations:

Constant Folding

When possible, the code generation software evaluates expressions in your MATLAB code that involve compile-time constants only. In the generated code, it replaces these expressions with the result of the evaluations. This behavior is known as constant folding. Because of constant folding, the generated code does not have to evaluate the constants during execution.

The following example shows MATLAB code that is constant-folded during code generation. The function `MultiplyConstant` multiplies every element in a matrix by a scalar constant. The function evaluates this constant using the product of three compile-time constants, `a`, `b` and `c`.

```
function out=MultiplyConstant(in) %#codegen
    a=pi^4;
    b=1/factorial(4);
    c=exp(-1);
    out=in.*(a*b*c);
end
```

The code generation software evaluates the expressions involving compile-time constants, `a`, `b`, and `c`. It replaces these expressions with the result of the evaluation in generated code.

Constant folding can occur when the expressions involve scalars only. To explicitly enforce constant folding of expressions in other cases, use the

`coder.const` function. For more information, see “Fold Function Calls into Constants”.

Control Constant Folding

You can control the maximum number of instructions that can be constant-folded from the command line or the Project Settings dialog box.

- At the command line, create a configuration object for code generation. Set the property `ConstantFoldingTimeout` to the value that you want.

```
cfg=coder.config('lib');
cfg.ConstantFoldingTimeout = 200;
```

- In the Project Settings dialog box, on the **All Settings** tab, set the field **Constant folding timeout** to the value that you want.

Loop Fusion

When possible, the code generation software fuses successive loops with the same number of runs into a single loop in the generated code. This optimization reduces loop overhead.

The following code contains successive loops, which are fused during code generation. The function `SumAndProduct` evaluates the sum and product of the elements in an array `Arr`. The function uses two separate loops to evaluate the sum `y_f_sum` and product `y_f_prod`.

```
function [y_f_sum,y_f_prod] = SumAndProduct(Arr) %#codegen
    y_f_sum = 0;
    y_f_prod = 1;
    for i = 1:length(Arr)
        y_f_sum = y_f_sum+Arr(i);
    end
    for i = 1:length(Arr)
        y_f_prod = y_f_prod*Arr(i);
    end
```

The code generated from this MATLAB code evaluates the sum and product in a single loop.

Successive Matrix Operations Combined

When possible, the code generation software converts successive matrix operations in your MATLAB code into a single loop operation in generated code. This optimization reduces excess loop overhead involved in performing the matrix operations in separate loops.

The following example contains code where successive matrix operations take place. The function `ManipulateMatrix` multiplies every element of a matrix `Mat` with a `factor`. To every element in the result, the function then adds a `shift` :

```
function Res=ManipulateMatrix(Mat, factor, shift)
    Res=Mat*factor;
    Res=Res+shift;
end
```

The generated code combines the multiplication and addition into a single loop operation.

Unreachable Code Elimination

When possible, the code generation software suppresses code generation from unreachable procedures in your MATLAB code. For instance, if a branch of an `if/elseif/else` statement is unreachable, then code is not generated for that branch.

The following example contains unreachable code, which is eliminated during code generation. The function `SaturateValue` returns a value based on the range of its input `x`.

```
function y_b = SaturateValue(x) %#codegen
    if x>0
        y_b = x;
    elseif x>10 %This is redundant
        y_b = 10;
    else
        y_b = -x;
    end
```

The second branch of the `if/elseif/else` statement is unreachable. If the variable `x` is greater than 10, it is also greater than 0. Therefore, the first branch is executed in preference to the second branch.

MATLAB Coder does not generate code for the unreachable second branch.

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 in MATLAB Generated Code”.

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.

Managing Data

- Chapter 46, “Working with Data”
- Chapter 47, “Enumerations and Modeling”
- Chapter 48, “Importing and Exporting Simulation Data”
- Chapter 49, “Working with Data Stores”

Working with Data

- “Data Types” on page 46-2
- “Data Objects” on page 46-35
- “Define Data Classes” on page 46-51
- “Supported Property Types” on page 46-56
- “Upgrade Level-1 Data Classes” on page 46-57
- “Associating User Data with Blocks” on page 46-59
- “Design Minimum and Maximum” on page 46-60

Data Types

In this section...

- “About Data Types” on page 46-2
- “Data Types Supported by Simulink” on page 46-3
- “Fixed-Point Data” on page 46-4
- “Enumerations” on page 46-6
- “Bus Objects” on page 46-6
- “Block Support for Data and Signal Types” on page 46-7
- “Create Signals of a Specific Data Type” on page 46-7
- “Specify Block Output Data Types” on page 46-8
- “Specify Data Types Using Data Type Assistant” on page 46-16
- “Display Port Data Types” on page 46-28
- “Data Type Propagation” on page 46-29
- “Data Typing Rules” on page 46-29
- “Typecast Signals” on page 46-30
- “Validate a Floating-Point Embedded Model” on page 46-30
- “Validate a Single-Precision Model” on page 46-31

About Data Types

The term *data type* refers to the way in which a computer represents numbers in memory. A data type determines the amount of storage allocated to a number, 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. The Simulink software 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 Rules” on page 46-29). In that way, you can avoid introducing data type errors that prevent your model from running to completion or simulating at all.

Data Types Supported by Simulink

Simulink supports all built-in numeric MATLAB data types except `int64` and `uint64`. 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
<code>double</code>	Double-precision floating point
<code>single</code>	Single-precision floating point
<code>int8</code>	Signed 8-bit integer
<code>uint8</code>	Unsigned 8-bit integer
<code>int16</code>	Signed 16-bit integer

Name	Description
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer

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`).

Several blocks support bus objects (`Simulink.Bus`) as data types. See “Bus Objects” on page 46-6.

Many Simulink blocks also support fixed-point data types. 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
```

Fixed-Point Data

The Simulink software 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 46-5 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 46-20.

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Fixed-Point Designer software. However, even if you do not have Fixed-Point Designer software, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, the Simulink software temporarily overrides fixed-point data types with floating-point data types when simulating the model.

Note If you use `fi` objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set `fioref` to prevent the checkout of a Fixed-Point Designer license.

To simulate a model without using Fixed-Point Designer:

- 1 In the **Model Hierarchy** pane, select the root model.

- 2 In the Simulink Editor, select **Analysis > Fixed-Point Tool**.

The Fixed-Point Tool opens.

- 3 In the section **Settings for selected system**:

- Set **Fixed-point instrumentation mode** to `Force off`.
- Set **Data type override** to `Double` or `Single`.
- Set **Data type override applies to** `All numeric types`.

- 4 If you use `fi` objects or embedded numeric data types in your model, set the `fipref` `DataTypeOverride` property to `TrueDoubles` or `TrueSingles` (to be consistent with the model-wide data type override setting) and the `DataTypeOverrideAppliesTo` property to `All numeric types`.

For example, at the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ...  
         'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

Enumerations

An *enumeration* is a user-defined data type with a fixed set of names that represent a single type of value. When the data type of an entity is an enumeration, the value of the entity must be one of the values defined that enumeration. For information about enumerations, see “Data Types”.

Bus Objects

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 46-27 for information about how to specify a bus object as a data type for blocks and classes.

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, 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`.

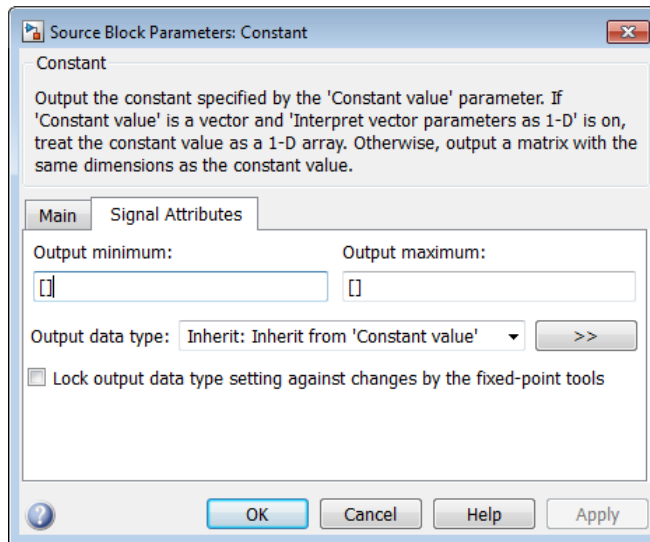
Create Signals of a Specific Data Type

You can introduce a 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.

Specify Block Output Data Types

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:

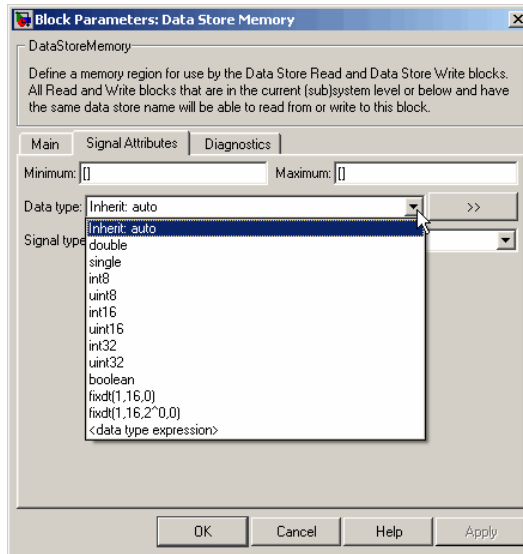
For Information About...	See...
Valid data type values that you can specify	“Entering Valid Data Type Values” on page 46-9
An assistant that helps you specify valid data type values	“Specify Data Types Using Data Type Assistant” on page 46-16
Specifying valid data type values for multiple blocks simultaneously	“Using the Model Explorer for Batch Editing” on page 46-12

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 46-10)
- The name of a built-in data type (see “Built-In Data Types” on page 46-11)
- An expression that evaluates to a data type (see “Data Type Expressions” on page 46-11)

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 46-29). 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.

Inheritance Rule	Description
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.
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.

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 46-3 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**

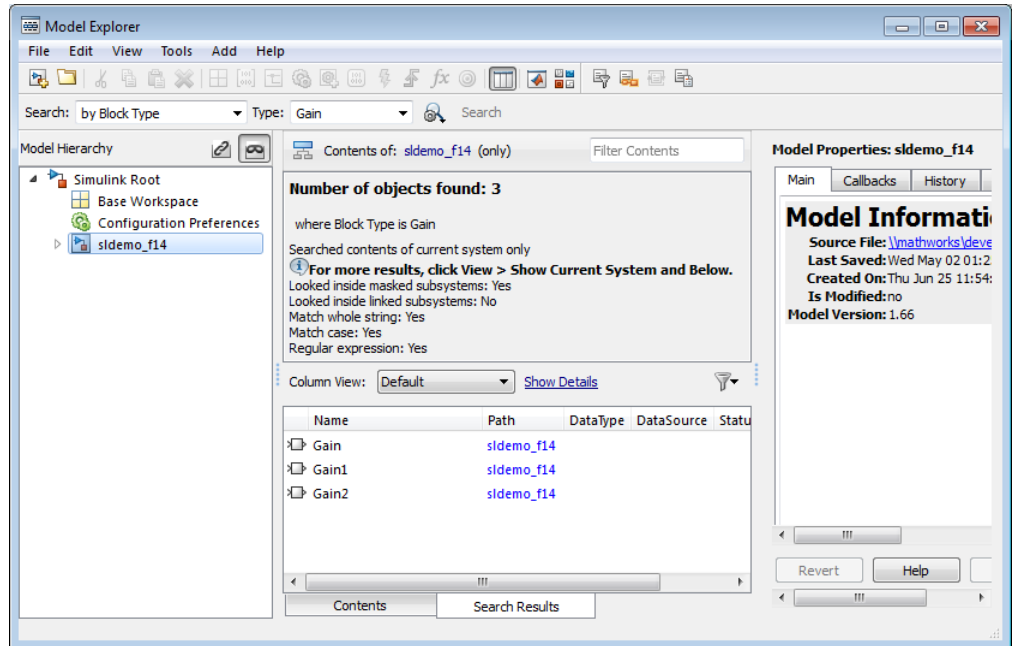
Specify the value of a data type parameter as the name of a data object that represents a data type. Simulink data objects that you instantiate from classes, such as `Simulink.NumericType` and `Simulink.AliasType`, simplify the task of making model-wide changes to output data types and allow you to use custom aliases for data types. See “Data Objects” on page 46-35 for more information about Simulink data objects.

Using the Model Explorer for Batch Editing

Using the Model Explorer (see “Model Explorer Overview” on page 9-2), you can assign the same output data type to multiple blocks simultaneously. For example, the `slexAircraftExample` model that comes with the Simulink product contains numerous Gain blocks. Suppose you want to specify the **Output data type** parameter of all the Gain blocks in the model as `single`. You can achieve this task as follows:

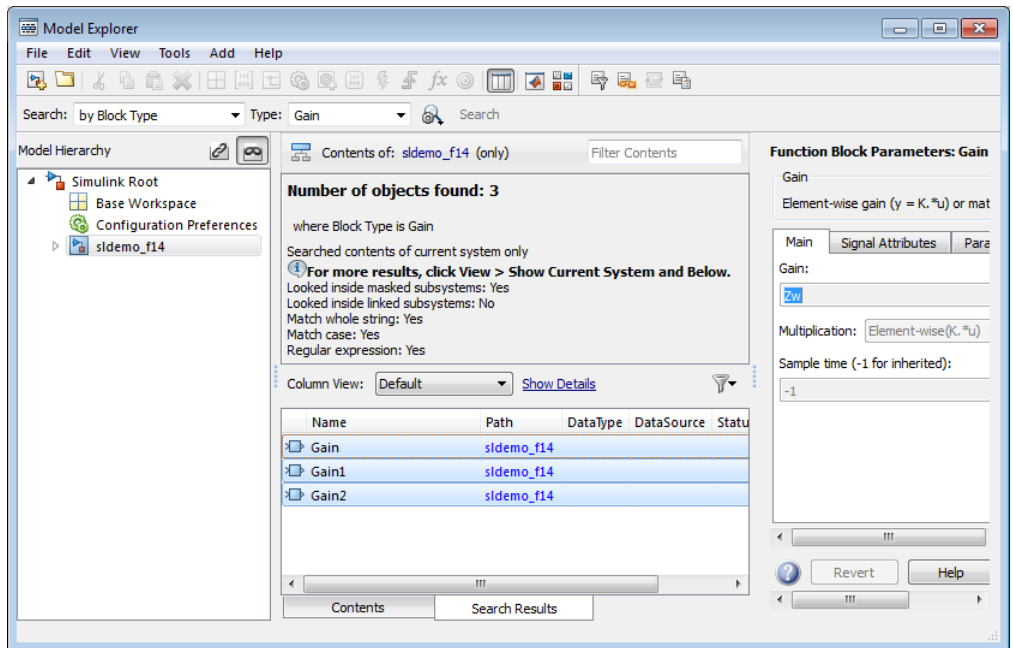
- 1 Use the Model Explorer search bar (see “Search Using Model Explorer” on page 9-63) to identify all blocks in the `slexAircraftExample` model of type `Gain`.

The Model Explorer **Contents** pane lists all Gain blocks in the model.



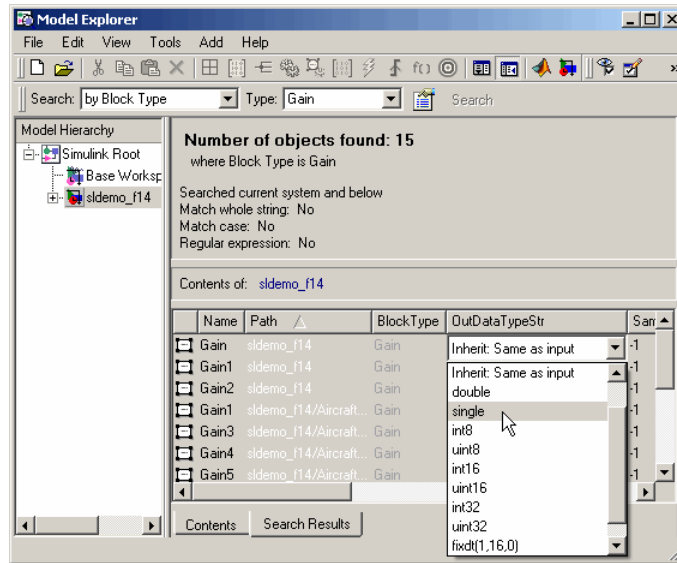
- In the Model Explorer **Contents** pane, select all the Gain blocks whose **Output data type** parameter you want to specify.

Model Explorer highlights the rows corresponding to your selections.



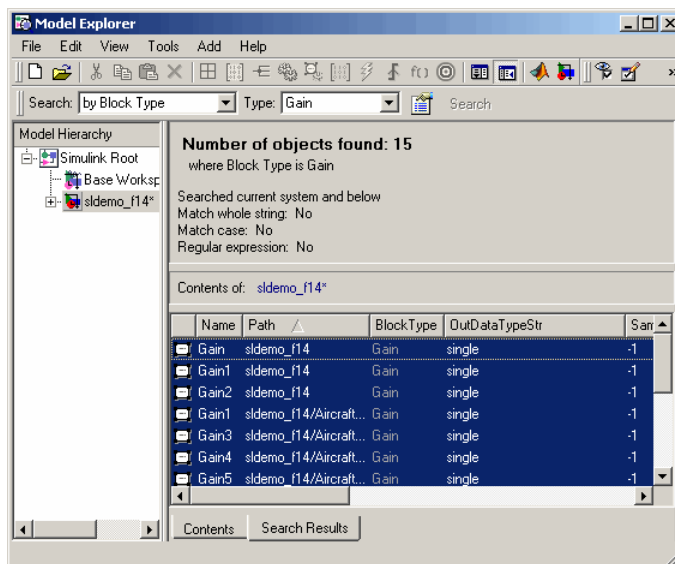
- 3 In the Model Explorer **Contents** pane, click the data type associated with one of the selected Gain blocks.

Model Explorer displays a pull-down menu with valid data type options.



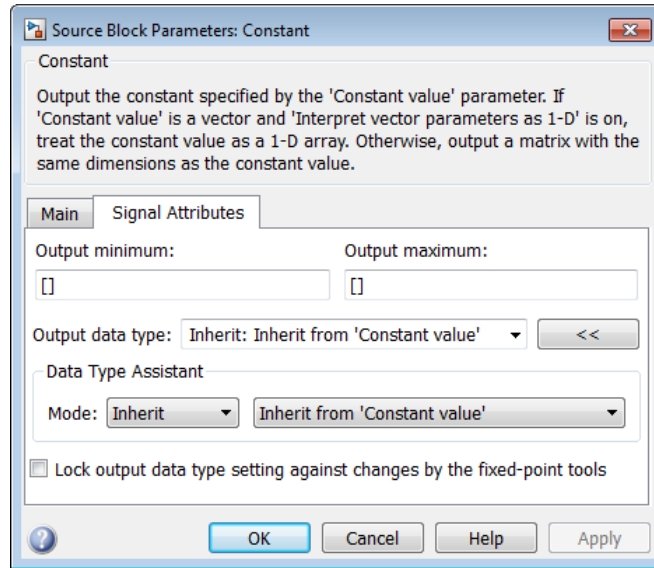
- 4 In the pull-down menu, enter or select the desired data type, for example, single.

Model Explorer specifies the data type of all selected items as single.

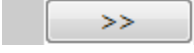



Specify Data Types Using Data Type Assistant

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.

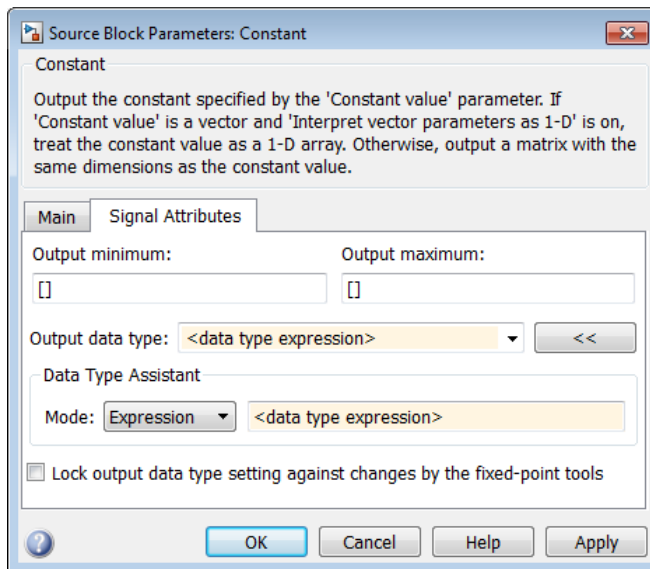
Use the **Data Type Assistant** to specify a data type as follows:

- 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

Mode	Description
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.

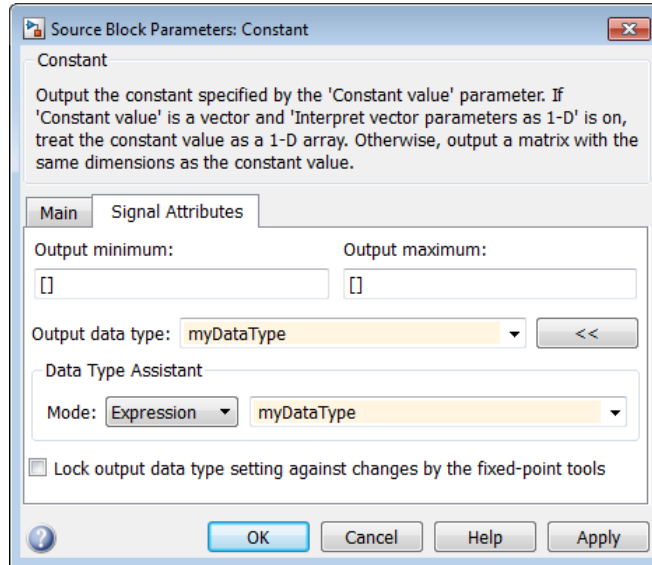


- 2 In the field that is to the right of the **Mode** field, select or enter a data type.

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:

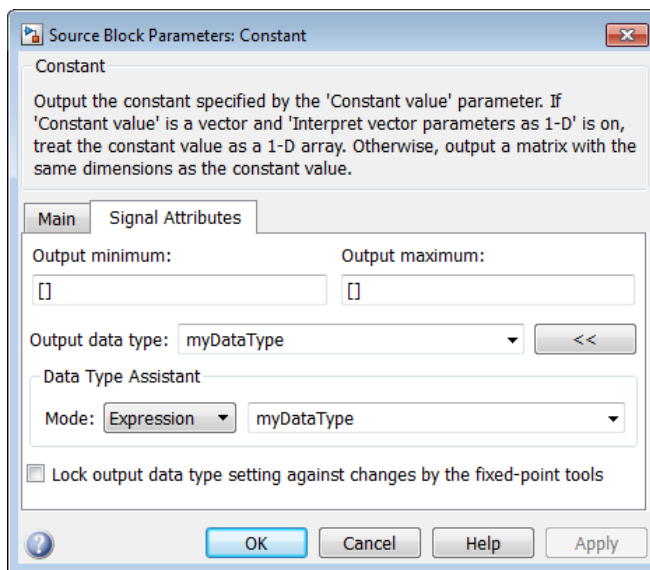
```
myDataType = Simulink.AliasType
myDataType.BaseType = 'single'
```

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.

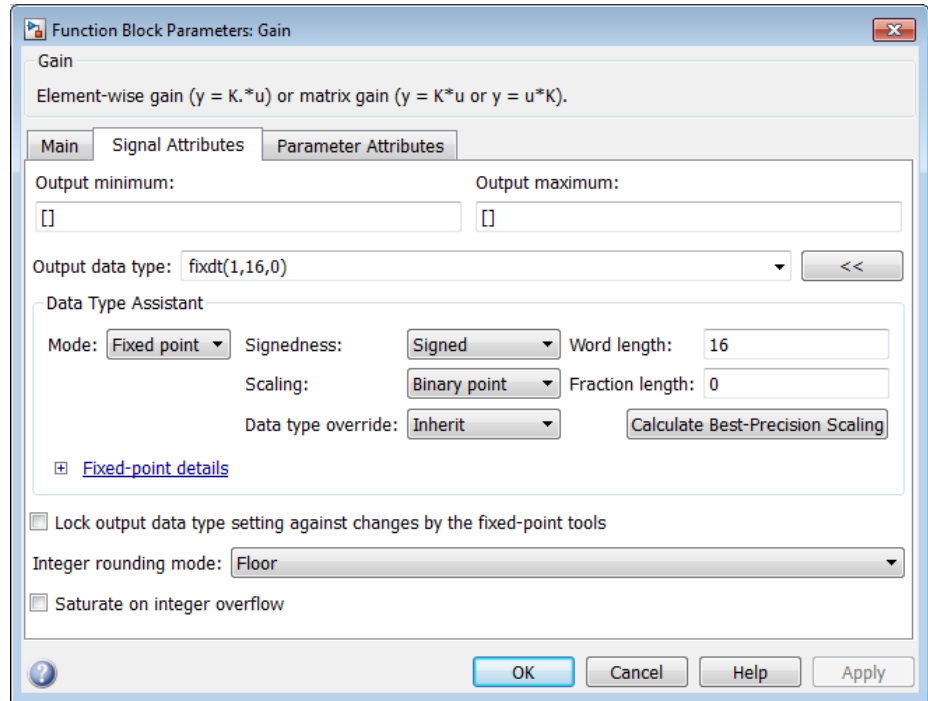
The assistant uses the data type that you specified to populate the associated data type parameter in the block or object dialog box. In the following example, the **Output data type** parameter of the Constant block specifies the same expression that you entered using the assistant.



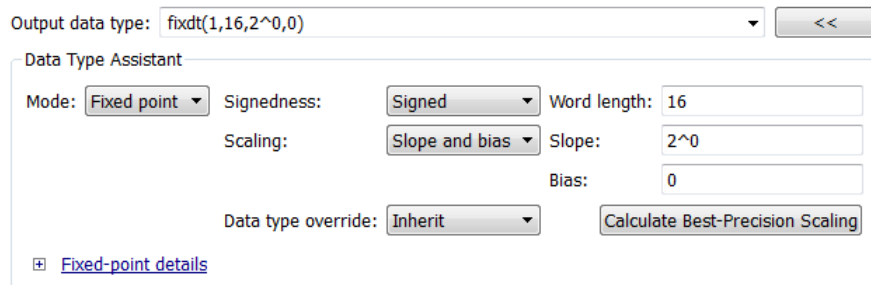
For more information about the data types that you can specify using the **Data Type Assistant**, see “Entering Valid Data Type Values” on page 46-9. For details about specifying fixed-point data types, see “Specify Fixed-Point Data Types with the Data Type Assistant”.

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 a detailed discussion about fixed-point data, see “Fixed-Point Basics”. 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:

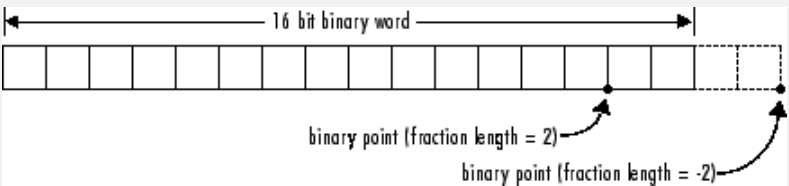


You can use the Data Type Assistant to set these fixed-point properties:

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 32. 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	<p>If you select this mode, the Data Type Assistant displays the Fraction Length field, which specifies the binary point location.</p> <p>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:</p>  <p>The diagram shows a horizontal row of 16 square boxes representing bits. Above the boxes, a double-headed arrow spans the entire width and is labeled "16 bit binary word". Below the boxes, two curved arrows point to specific positions. The first arrow points to the second box from the right and is labeled "binary point (fraction length = 2)". The second arrow points to the right of the last box and is labeled "binary point (fraction length = -2)".</p> <p>The default binary point is 0.</p>
Slope and bias	<p>If you select this mode, the Data Type Assistant displays fields for entering the Slope and Bias.</p> <p>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.</p>

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”.

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, <code>Simulink.Signal</code> object or Stateflow chart in Simulink that is using the signal.
Off	Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

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

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. The Simulink software displays the scaling values in the **Fraction Length** field or the **Slope** and **Bias** fields. For more information, see “Constant Scaling for Best Precision”.

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:

Output minimum: Output maximum:

Output data type:

Data Type Assistant

Mode: Signedness: Word length:

Scaling: Slope:

Bias:

Data type override:

[Fixed-point details](#)

Representable maximum:	32767
Output maximum:	<input type="text" value="[]"/>
Constant value:	90
Output minimum:	<input type="text" value="[]"/>
Representable minimum:	-32768

Precision:

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 “Signal Ranges” on page 50-45 and “Check Parameter Values” on page 24-10 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. For information about these three quantities, see “Fixed-Point Basics”.

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:

Output minimum: Output maximum:

Output data type: <<

Data Type Assistant

Mode: Signedness: Word length:

Scaling: Fraction length:

Data type override:

[Fixed-point details](#)

Representable maximum:	32767	
Output maximum:	50000	Outside representable range by 17233 (17233 x precision)
Output minimum:	MySymbol	Cannot evaluate
Representable minimum:	-32768	

Precision:

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's **Signal Attributes** tab, the subpane could look like this:

☐ Fixed-point details	
Representable maximum:	32767
Output maximum:	☐
Upper saturation limit:	inf
Initial condition:	1 .. 4
Lower saturation limit:	-inf
Output minimum:	☐
Representable minimum:	-32768
Precision:	1

Note that 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.

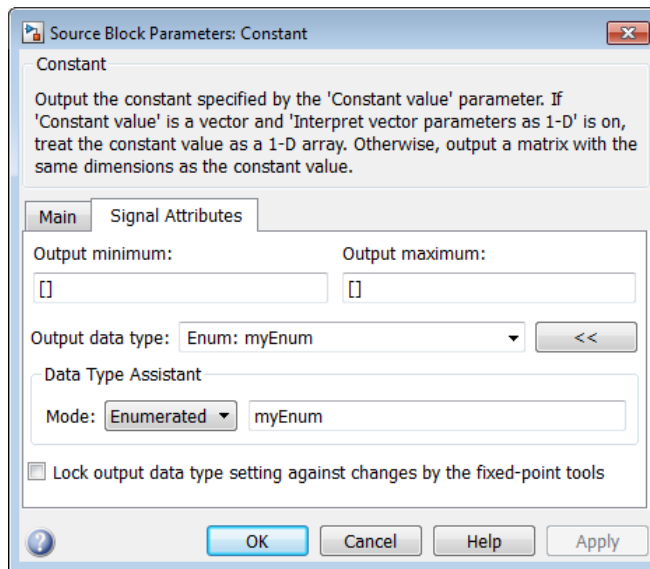
Note also that `Initial condition` displays the value `1..4`. The actual value is a vector or matrix whose smallest element is 1 and largest element is 4. To conserve space, the **Fixed-point details** subpane shows only the smallest and largest element of a vector or matrix. An ellipsis (`..`) replaces the omitted 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”.

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.



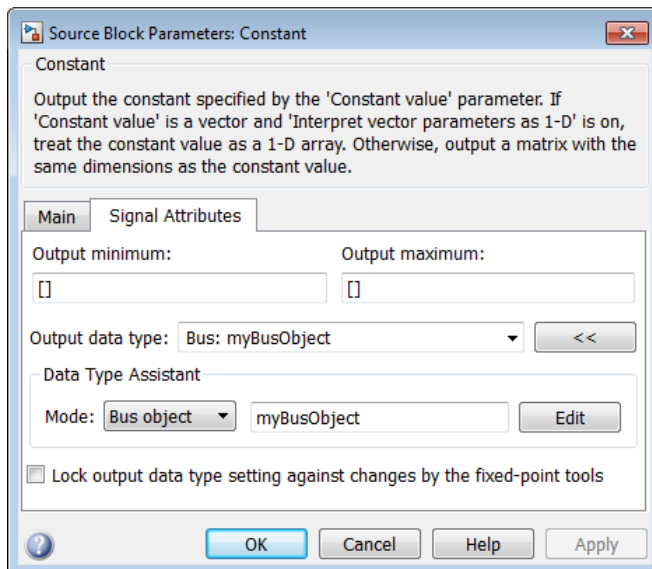
For details about enumerated data types, see “Data Types”.

Specify a Bus Object Data Type

The blocks listed in the section called “Bus Objects” on page 46-6 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 “Associating Bus Objects with Simulink Blocks” on page 51-25

Display Port Data Types

In the Simulink Editor, select **Display > Signals & Ports > Port Data Types**. The port data type display is not automatically updated when you change the data type of a diagram element. To refresh the display, press **Ctrl+D**.

Data Type Propagation

Whenever you start a simulation, enable display of port data types, or refresh the port data type display, the Simulink software 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. See “Typecast Signals” on page 46-30 for more information.

Data Typing Rules

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, the Simulink software 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.

Typecast Signals

An error is displayed whenever it detects that a signal is connected to a block that does not accept the signal's data type. If you want to create such a connection, you must explicitly typecast (convert) the signal to a type that the block does accept. You can use the Data Type Conversion block to perform such conversions.

Validate a Floating-Point Embedded Model

You can use data type override mode to switch the data types in your model. This capability allows you to maintain one model but simulate and generate code for multiple data types, and also 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 affecting any other data types in your model.

Apply a Data Type Override to Floating-Point Data Types

To apply a 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 a data type override using one of the following methods. In these examples, both methods change all floating-point data types to single.

From the Command Line. For example:

```
set_param(gcs, 'DataTypeOverride', 'Single');  
set_param(gcs, 'DataTypeOverrideAppliesTo', 'Floating-point');
```

Using the Fixed-Point Tool. For example:

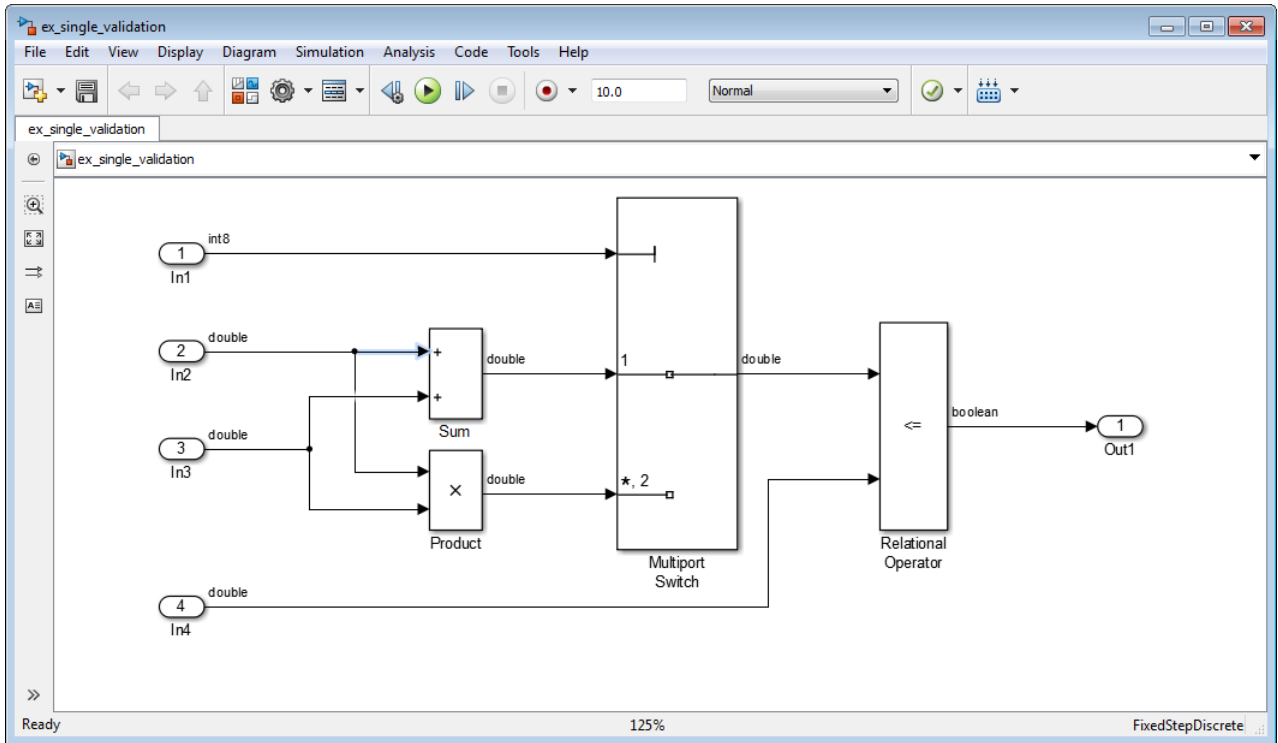
- 1** In the Simulink Editor, select **Analysis > Fixed-Point Tool**.
The Fixed-Point Tool opens.
- 2** Select **View > Show Settings for Selected System**.
- 3** In the Fixed-Point Tool right pane, under **Settings for selected system**:
 - a** Set **Data type override** to **Single**.
 - b** Set **Data type override applies** to **Floating-point**.

For more information on data type override settings, see `fxptdlg`.

Validate a Single-Precision Model

This example uses the `ex_single_validation` model to show how you can use data type overrides. 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
```

Run Model Advisor Check.

- 1 From the model menu, select **Analysis > Model Advisor > 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 generates a warning that multiple blocks use double-precision floating-point operations.

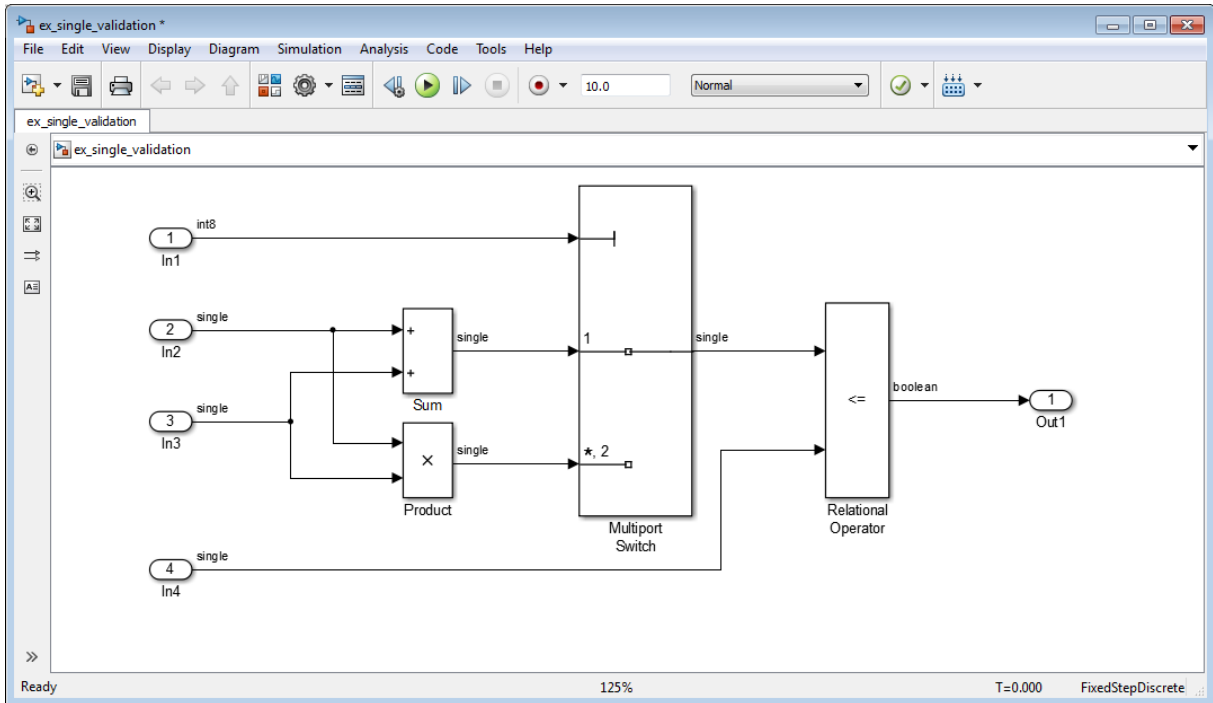
Override Floating-Point Data Types With Singles.

- 1 In the Simulink Editor, select **Analysis > Fixed-Point Tool**.

The Fixed-Point Tool opens.

- 2 In the Fixed-Point Tool right pane, under **Settings for selected system**:
 - a Set **Data type override** to Single.
 - b Set **Data type override applies to** Floating-point.
 - c Click **Apply**.
- 3 In the model menu, select **Simulation > Update Diagram**.

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.



Re-run the Model Advisor Check. In the Model Advisor, run the **Identify questionable operations for strict single-precision design** check again.

The check passes indicating that this algorithm is suitable for single-precision embedded use.

Data Objects

In this section...

“About Data Object Classes” on page 46-35
“About Data Object Methods” on page 46-36
“Create Data Objects Using Model Explorer” on page 46-38
“Using the Model Explorer to Create Data Objects” on page 46-39
“About Object Properties” on page 46-40
“Changing Object Properties” on page 46-41
“Handle Versus Value Classes” on page 46-42
“Comparing Data Objects” on page 46-44
“Saving and Loading Data Objects” on page 46-44
“Using Data Objects in Simulink Models” on page 46-45
“Creating Persistent Data Objects” on page 46-45
“Data Object Wizard” on page 46-45

About Data Object Classes

You can create entities called data objects that specify values, data types, tunability, value ranges, and other key attributes of block outputs and parameters. You can create various types of data objects and assign them to workspace variables. You can use the variables in Simulink dialog boxes to specify parameter and signal attributes. This allows you to make model-wide changes to parameter and signal specifications simply by changing the values of a few variables. With Simulink objects you can parameterize the specification of a model's data attributes.

Note This section uses the term *data* to refer generically to signals and parameters.

The Simulink software uses objects called data classes to define the properties of specific types of data objects. The classes also define functions, called

methods, for creating and manipulating instances of particular types of objects. A set of built-in classes are provided for specifying specific types of attributes. Some MathWorks products based on Simulink, such as the Simulink Coder product, also provide classes for specifying data attributes specific to their applications. See the documentation for those products for information on the classes they provide. You can also create subclasses of some of these built-in classes to specify attributes specific to your applications (see “Define Data Classes” on page 46-51).

Memory structures called *packages* are used to store the code and data that implement data classes. The classes provided by the Simulink software reside in the Simulink package. Classes provided by products based on Simulink reside in packages provided by those products. You can create your own packages for storing the classes that you define.

Class Naming Convention

Simulink uses dot notation to name classes:

PACKAGE.CLASS

where CLASS is the name of the class and PACKAGE is the name of the package to which the class belongs, for example, Simulink.Parameter. This notation allows you to create and reference identically named classes that belong to different packages. In this notation, the name of the package is said to qualify the name of the class.

Note Class and package names are case sensitive. You cannot, for example, use A.B and a.b interchangeably to refer to the same class.

About 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. The Simulink software 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.

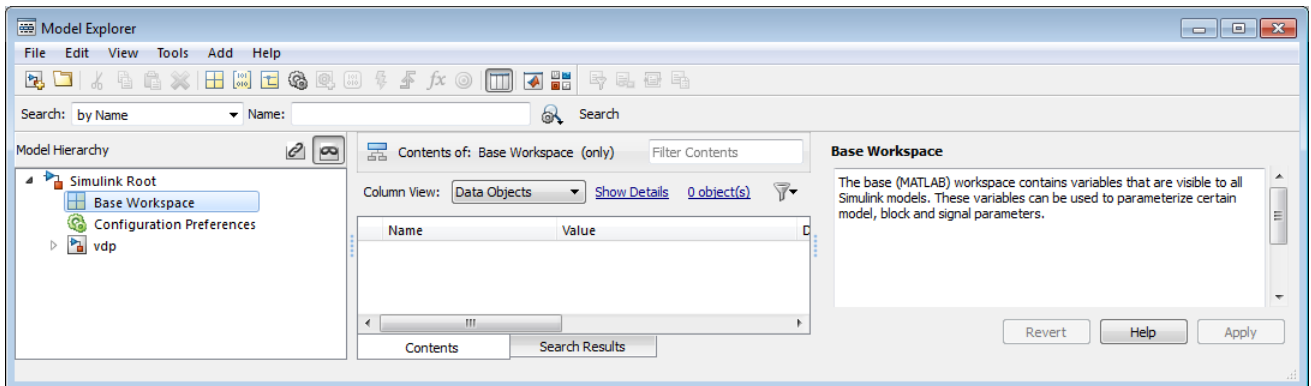
The value returned by a constructor depends on whether its class is a handle class or a value class. The constructor for a handle class returns a handle to the instance that it creates if the class of the instance is a handle class; otherwise, it returns the instance itself (see “Handle Versus Value Classes” on page 46-42).

Create Data Objects Using Model Explorer

This example shows how to create data objects using Model Explorer.

Open the Model Explorer. In the Simulink Editor, select **View > Model Explorer**.

In the **Model Hierarchy** pane, select **Base Workspace**.



Click **Add > Simulink Parameter**

A `Simulink.Parameter` object appears in the base workspace with its properties displayed in the **Model Properties** pane.

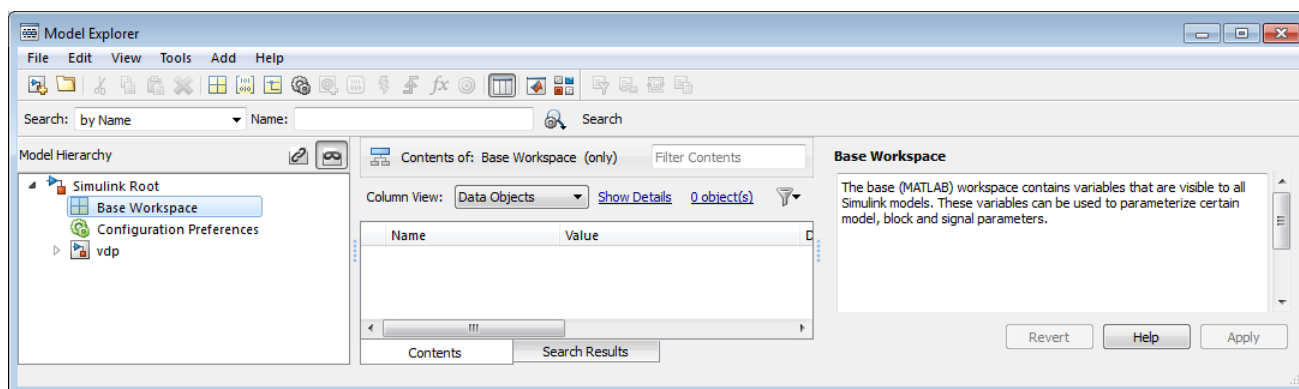
Using the Model Explorer to Create Data Objects

You can use the Model Explorer (see “Model Explorer Overview” on page 9-2) as well as MATLAB commands to create data objects. To use the Model Explorer,

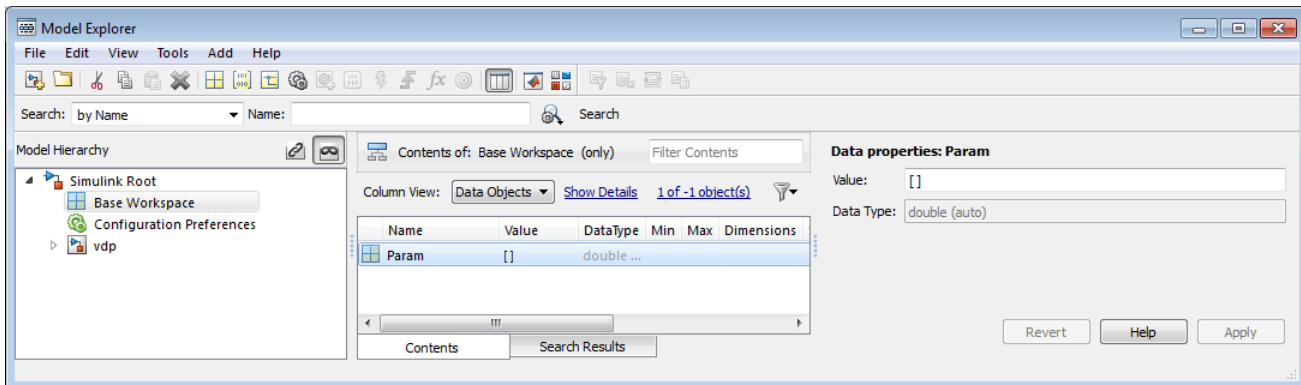
- 1 Select the workspace in which you want to create the object in the Model Explorer **Model Hierarchy** pane.

Only `Simulink.Parameter` and `Simulink.Signal` objects for which the storage class is set to `Auto` can reside in a model workspace. You must create all other Simulink data objects in the base MATLAB workspace to ensure the objects are unique within the global Simulink context and accessible to all models.

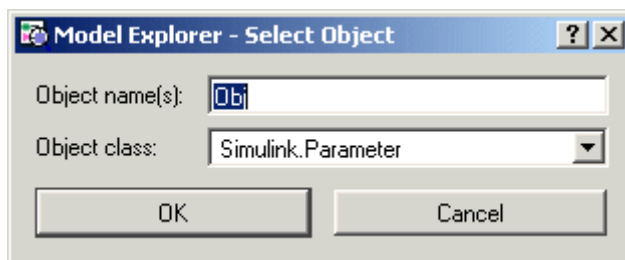
Note Subclasses of `Simulink.Parameter` and `Simulink.Signal` classes, including `mpt.Parameter` and `mpt.Signal` objects (Embedded Coder license required), can reside in a model workspace only if their storage class is set to `Auto`.



- 2 Select the type of the object that you want to create (for example, **Simulink Parameter** or **Simulink Signal**) from the Model Explorer’s **Add** menu or from its toolbar. The Simulink software creates the object, assigns it to a variable in the selected workspace, and displays its properties in the Model Explorer’s **Contents** and **Dialog** panes.



If the type of object you want to create does not appear on the **Add** menu, select **Find Custom** from the menu. The MATLAB path is searched for all data object classes derived from Simulink class on the MATLAB path, including types that you have created, and displays the result in a dialog box.



- 3 Select the type of object (or objects) that you want to create from the **Object class** list and enter the names of the workspace variables to which you want the objects to be assigned in the **Object name(s)** field. Simulink creates the specified objects and displays them in the Model Explorer's **Contents** pane.

About Object Properties

Object properties are variables associated with an object that specify properties of the entity that the object represents, for example, the size of a data type. The object's class defines the names, value types, default values, and valid value ranges of the object's properties.

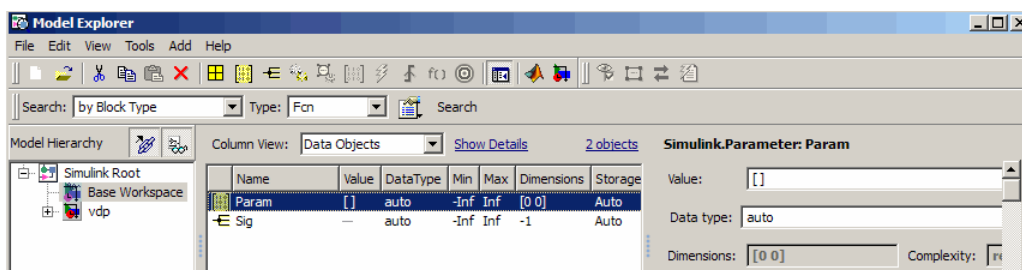
Changing Object Properties

You can use either the Model Explorer (see “Using the Model Explorer to Change an Object’s Properties” on page 46-41) or MATLAB commands to change a data object’s properties (see “Use MATLAB Commands to Change Workspace Data” on page 4-78).

Using 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: Contents Pane” on page 9-19). 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.

Using 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 46-42), `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';
```

Handle Versus Value Classes

Simulink 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 46-38) 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.

You can determine whether a variable is assigned to an instance of a class or to a handle to that class by evaluating it at the MATLAB command line. MATLAB appends the text (`handle`) to the name of the object class in the value display, e.g.,

```
>> gain = Simulink.Parameter  
  
gain =  
  
Simulink.Parameter (handle)  
    Value: []  
    CoderInfo: [1x1 Simulink.ParamCoderInfo]  
Description: ''  
    DataType: 'auto'  
        Min: []  
        Max: []  
    DocUnits: ''  
Complexity: 'real'  
Dimensions: [0 0]
```

Copying Handle Classes

Use the `copy` method of a handle class to create copies of instances of that class. For example, `Simulink.ConfigSet` is a handle class 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 = activeConfig.copy;
develConfig.Name = 'develConfig';
attachConfigSet(gcs, develConfig);
```

Comparing 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 = A.isequal(B)

result =

     1
```

Saving and Loading Data Objects

You can use the `save` command to save data objects in a MAT-file and the `load` command to restore them to the MATLAB workspace in the same or a later session. 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.

Using Data Objects in Simulink Models

You can use data objects in Simulink models as parameters and signals. Using data objects as parameters and signals allows you to specify simulation and code generation options on an object-by-object basis.

Creating Persistent Data Objects

To create parameter and signal objects that persist across Simulink sessions, first write a script that creates objects (see “Define Data Classes” on page 46-51) or at the command line and save them in a MAT-file (see “Saving and Loading Data Objects” on page 46-44). Then use either the script or a load command as the `PreLoadFcn` callback routine for the model that uses the objects. For example, suppose you save the data objects in a file named `data_objects.mat` and the model to which they apply is open and active. Then, entering the following command

```
set_param(gcs, 'PreLoadFcn', 'load data_objects');
```

at the MATLAB command line sets `load data_objects` as the model’s preload function. This in turn causes the data objects to be loaded into the model workspace whenever you open the model.

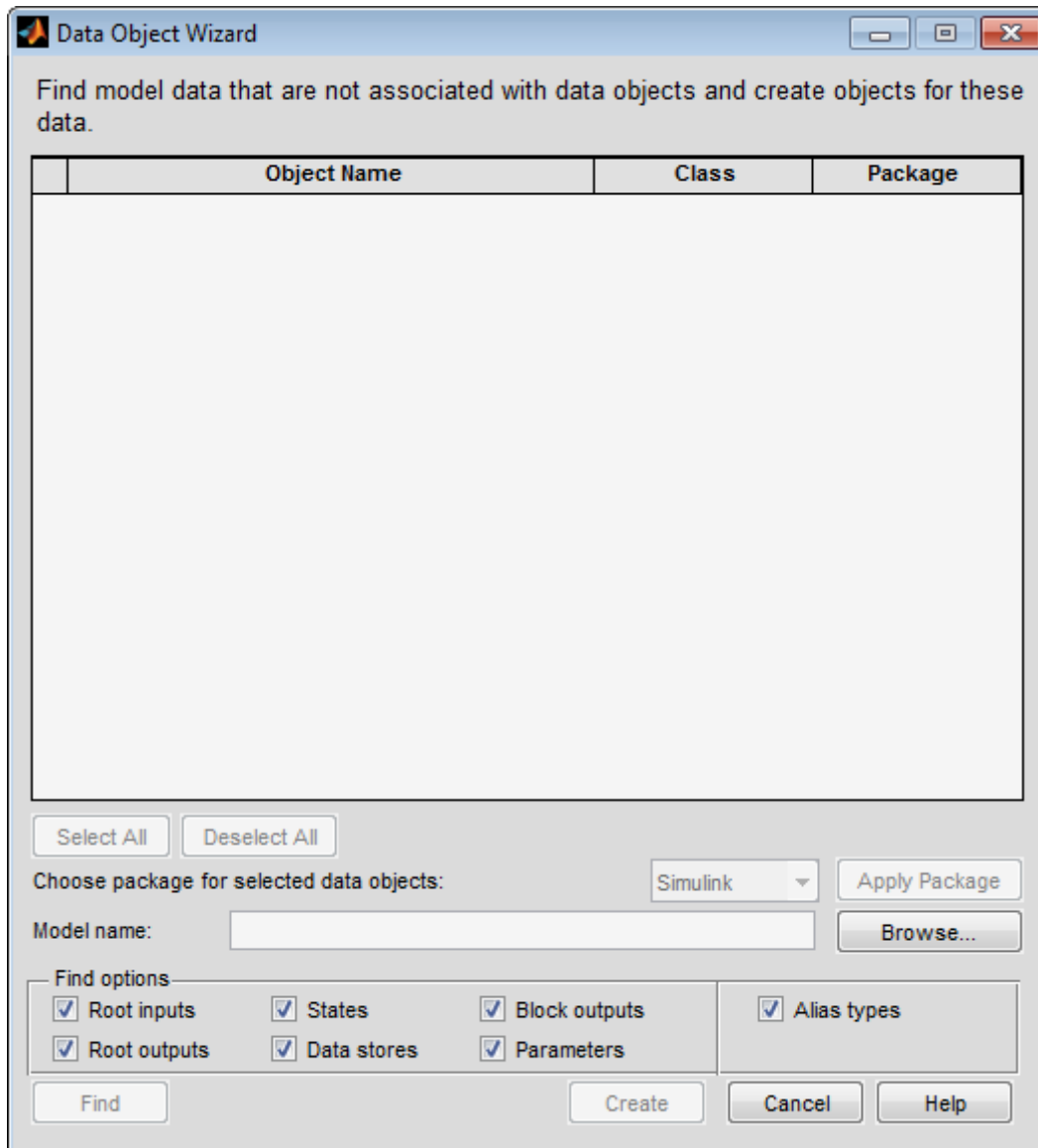
Data Object Wizard

The Data Object Wizard allows you to determine quickly which model data are not associated with data objects and to create and associate data objects with the data.

To use the wizard to create data objects:

- 1 In the Simulink Editor, select **Code > Data Objects > Data Object Wizard**.

The Data Object Wizard appears.



- 2** Enter, if necessary, the name of the model you want to search in the wizard's **Model name** field.

By default the wizard displays the name of the model from which you opened the wizard. You can enter the name of another model in this field. If the model is not open, the wizard opens the model.

- 3** In **Find options**, uncheck any of the data object types that you want the search to ignore.

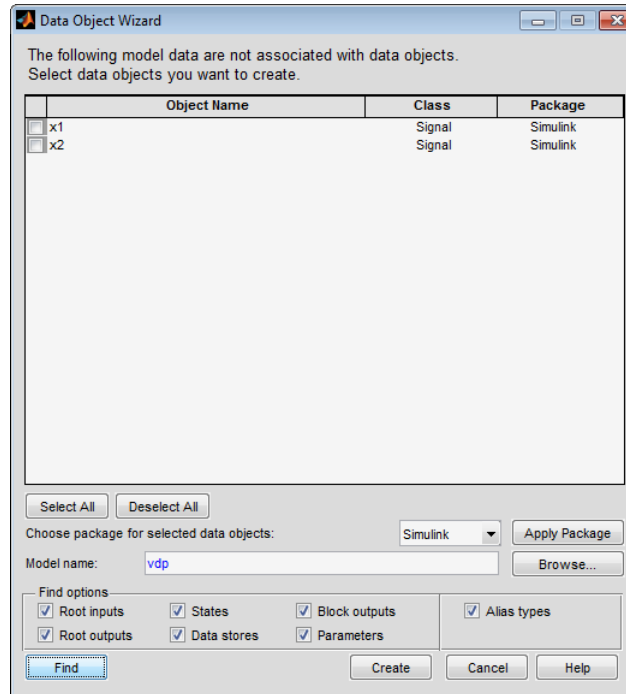
The search options include:

Option	Description
Root inputs	Named signals from root-level input ports
Root outputs	Named signals from root-level output ports
States	States associated with any instances of the following discrete block types: Discrete Filter Discrete State-Space Discrete-Time Integrator Discrete Transfer Fcn Discrete Zero-Pole Memory PID Controller PID Controller (2DOF) Unit Delay
Data stores	Data stores (see "About Data Stores" on page 49-2)
Block outputs	Named signals emitted by non-root-level blocks.

Option	Description
Parameters	<ul style="list-style-type: none"> • Parameters of any instances of the following block types: <ul style="list-style-type: none"> Constant Gain Relay • Stateflow data with a Scope of Parameter. <p>For more information, see “Share Simulink Parameters with Charts” in the Stateflow documentation.</p>
Alias types	<p>Data whose data type is a registered custom data type. This option applies only if you are generating code from the model. See “Create Data Objects with Data Object Wizard” in the Embedded Coder documentation for more information.</p>

4 Click the wizard’s **Find** button.

The wizard displays the search results in the data objects table.



- 5 Check the data for which you want the wizard to create data objects.
- 6 If you want the wizard to use data object classes from a package other than the Simulink standard class package to create the data objects, select the package from the **Choose package for selected data objects** list and then select **Apply Package** to confirm your choice.

Note User-defined packages that you add to the Data Object Wizard must contain a `Simulink.Signal` subclass named `Signal` and a `Simulink.Parameter` subclass named `Parameter`.

- 7 Click **Create**.

The wizard creates data objects of the appropriate class for the data selected in the search results table.

Note Use the Model Explorer to view and edit the created data objects.

Define Data Classes

This example shows how to subclass Simulink data classes.

Use MATLAB class syntax to create a data class in a package, and, optionally, assign properties to the data class and define custom storage classes.

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
```

Optional: Add properties to data class

The `properties` and `end` keywords enclose a property definition block.

See “Supported Property Types” on page 46-56.

```
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 % clasdef
```

Note The MATLAB editor will not recognize the attributes `PropertyType` and `AllowedValues`, because they are only defined for Simulink data classes. You can suppress `mlint` warnings about these unrecognized attributes by right-clicking the attribute in the MATLAB editor and selecting **Suppress “Unknown attribute name ‘...’” > In This File**.

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.

In this example, the constructor initializes the value of object `obj` based on an optional input argument.

```
clasdef Parameter < Simulink.Parameter
    methods
        function obj = Parameter(optionalValue)
            if (nargin == 1)
                obj.Value = optionalValue;
            end
        end
    end % methods
end % clasdef
```

Optional: Define custom storage classes

Use the `setupCoderInfo` method to configure the `CoderInfo` object of your class. Then, create a call to the `useLocalCustomStorageClasses` method and launch 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 Launch the Custom Storage Class Designer for your package.

```
cscdesigner('mypkg')
```

- 3 Define custom storage classes. “Use Custom Storage Class Designer”.

Optional: Define custom attributes for custom storage classes

- 1 Create a MATLAB file `myCustomAttribs.m` and open it for editing.
- 2 Define a subclass of `Simulink.CustomStorageClassAttributes` MATLAB class syntax.

```
classdef CSCAttributes < Simulink.CustomStorageClassAttributes
    properties(PropertyType = 'char')
        AlternateName = '';
    end
end % classdef
```

- 3 Override the default implementation of the `getInstanceSpecificProps` method.

Note This is an optional step. By default, you can set all custom attributes to be instance-specific in the Custom Storage Class designer so that they are modifiable for each data object. However, you can limit which properties are allowed to be instance-specific.

```
classdef CSCAttributes < Simulink.CustomStorageClassAttributes
    properties(PropertyType = 'char')
        AlternateName = '';
    end % properties

    properties(PropertyType = 'logical scalar')
        IsAlternateNameInstanceSpecific = true;
    end

    methods
        function props = getInstanceSpecificProps(obj)
            if obj.IsAlternateNameInstanceSpecific
                props = findprop(obj, 'AlternateName');
            end
        end
    end % methods
end % classdef
```

Alternate way of defining data classes

- 1 Locate the +SimulinkDemos data class package on the MATLAB path.

Note The +SimulinkDemos folder is at
matlabroot/toolbox/simulink/simdemos/dataclasses

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 as desired.

Supported Property Types

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	<code>properties(PropertyType = 'double scalar')</code>
int32 number	<code>properties(PropertyType = 'int32 scalar')</code>
Logical number	<code>properties(PropertyType = 'logical scalar')</code>
String (char)	<code>properties(PropertyType = 'char')</code>
String with limited set of allowed values	<code>properties(PropertyType = 'char', AllowedValues = {'a', 'b', 'c'})</code>

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.

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(gcf, '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(gcf, 'UserData')
```

The following command saves the user data associated with a block in the model file of the model containing the block.

```
set_param(gcf, '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.

Design Minimum and Maximum

In this section...

“Use of Design Minimum and Maximum” on page 46-60

“Valid Values for Design Minimum and Maximum” on page 46-60

Use of Design Minimum and Maximum

You can specify the design minimum and maximum for model data such as blocks and data objects. Simulink uses the design minimum and maximum as follows.

- To define a valid range for Simulink parameters and signals and use it in range-checking
- To calculate best-precision scaling for fixed-point data types
- To calculate derived minimum and maximum for model data for which design minimum and maximum are not specified

Valid Values for Design Minimum and Maximum

Simulink no longer allows you to specify the design minimum and maximum as $-\text{Inf}/\text{Inf}$. The default design minimum or maximum is $[\]$.

Previously, you could specify the design minimum and maximum as $-\text{Inf}/\text{Inf}$. However, this specification is ambiguous.

It may imply that the design minimum and maximum are explicitly specified; in other words, it may imply that the parameter or signal can have any value. It may also imply that the design minimum and maximum are unspecified. While this ambiguity may not have a significant effect on range-checking, it could affect the calculation of derived minimum and maximum or the checking of data type validity.

Note Simulink generates an error or warning when you specify the design minimum and maximum as $-\text{Inf}/\text{Inf}$.

Avoiding Specifying Infinite Design Minimum or Maximum

There are three sources for the warning Simulink generates if the design minimum and/or maximum are set to $-\text{Inf}/\text{Inf}$. Each source requires a different solution.

- 1** MATLAB code
 - a** Use error handling tools such as `dbstop` and `lastwarn` to locate the MATLAB code that is setting the design minimum and maximum to $-\text{Inf}/\text{Inf}$.
 - b** Either remove these lines of code from the MATLAB file or replace instances of $-\text{Inf}$ and Inf with `[]`.
- 2** MAT-file: Resave the MAT-file
- 3** SLX file: Resave the SLX file

Enumerations and Modeling

- “About Simulink Enumerations” on page 47-2
- “Define Simulink Enumerations” on page 47-3
- “Use Enumerated Data in Simulink Models” on page 47-10
- “Simulink Constructs that Support Enumerations” on page 47-22
- “Simulink Enumeration Limitations” on page 47-25

About Simulink Enumerations

Simulink enumerations are built on enumerations defined for the MATLAB language. They are subclasses of the abstract superclass `Simulink.IntEnumType`, and inherit from that superclass the capabilities necessary to be used in the Simulink environment.

Before you begin to use enumerations in a modeling context, you should understand information provided in “Enumerations”.

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”.

Define Simulink Enumerations

In this section...

“Workflow to Define a Simulink Enumeration” on page 47-3

“Create Simulink Enumeration Class” on page 47-3

“Customize Simulink Enumeration” on page 47-4

“Save Enumeration in a MATLAB File” on page 47-7

“Change and Reload Enumerations” on page 47-7

“Import Enumerations Defined Externally to MATLAB” on page 47-8

Workflow to Define a Simulink Enumeration

- 1 Create a class definition.
- 2 Optionally, customize the enumeration.
- 3 Optionally, save the enumeration in a MATLAB file.

Create Simulink Enumeration Class

To create a Simulink enumeration class, in the class definition:

- Define the class as a subclass of `Simulink.IntEnumType`
- 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 “Target” and “Hardware Implementation Pane” for more information.

For more information on superclasses, see “Converting to Superclass Value”.

Customize Simulink Enumeration

About Simulink Enumeration Customizations

You can customize a Simulink enumeration by using the same techniques that work with MATLAB classes, as described in “Modifying Superclass Methods and Properties”.

A primary source of customization are the methods associated with an enumeration.

Inherited Methods

Enumeration class definitions can include an optional `methods` section. Simulink enumerated classes inherit the following static methods from the superclass `Simulink.IntEnumType`. For more information about these methods, see `Simulink.defineIntEnumType`.

Default Method	Description	Default Value Returned or Specified	Usage Context
<code>getDescription</code>	Returns a description of the enumeration.	' '	Code generation
<code>getHeaderFile</code>	Specifies the file in which the enumeration is defined for code generation.	' '	Code generation
<code>addClassNameToEnumNames</code>	Specifies whether the class name becomes a prefix in generated code.	<code>false</code> — prefix is not used	Code generation

Overriding Inherited Methods

You can override the inherited methods to customize the behavior of an enumeration. To override a method, include a customized version of the method in the `methods` section in the enumerated class definition. If you do not want to override the inherited methods, omit the `methods` section.

Specifying a Default Enumerated Value

Simulink software 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” in the Simulink Coder documentation.

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 Returns the default enumerated value.
% This value must be an instance of the enumerated class.
% If this method is not defined, the first enumeration is used.
    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 47-15 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 Enumerations

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...
<p>Do one of the following:</p> <ul style="list-style-type: none"> • Locate and delete specific obsolete instances. • Delete everything from the workspace by using the <code>clear</code> command. 	<ul style="list-style-type: none"> • 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.

For more information about applying enumeration changes, see “Modifying and Reloading Classes”.

Import Enumerations Defined Externally to MATLAB

If you have enumerations defined externally to MATLAB—for example, in a data dictionary, that you want to import for use within the Simulink environment, you can do so programmatically with calls to the function `Simulink.defineIntEnumType`. This function 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:

- String 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);
```

Use Enumerated Data in Simulink Models

In this section...

“Simulate with Enumerations” on page 47-10

“Specify Enumerations as Data Types” on page 47-12

“Get Information About Enumerations” on page 47-13

“Enumeration Value Display” on page 47-13

“Instantiate Enumerations” on page 47-15

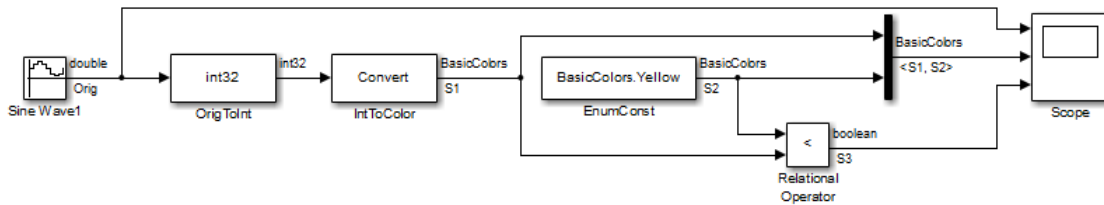
“Enumerated Values in Computation” on page 47-19

Simulate with Enumerations

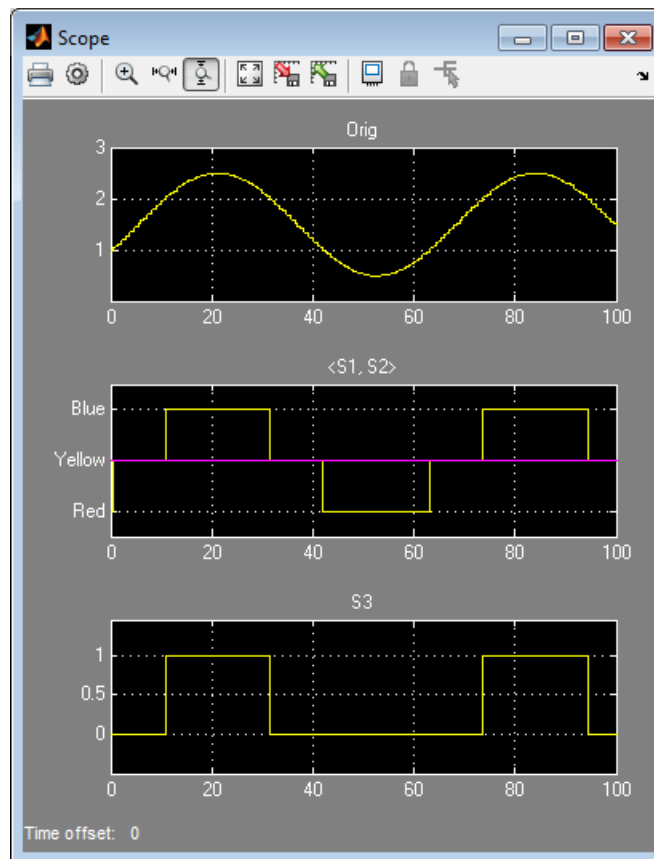
Consider the following enumeration class definition—`BasicColors` with enumerated values `Red`, `Yellow`, and `Blue`, with `Blue` as the default value:

```
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”. 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.

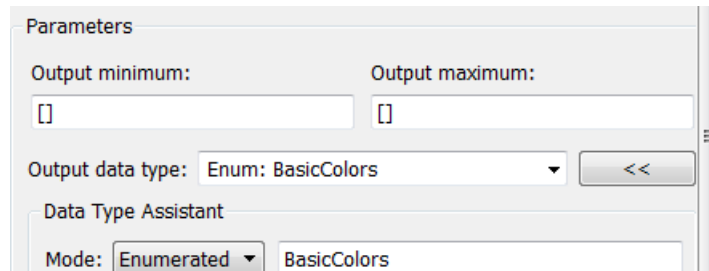
The sort order used by the comparison is the numeric order of the underlying integers of the compared values, *not* the lexical order in which the enumerated values appear in the enumerated class definition. In this example the two orders are the same, but they need not be. See “Specify Enumerations as Data Types” on page 47-12 and “Enumerated Values in Computation” on page 47-19 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 47-19 for more information.

Get Information About Enumerations

Use the enumeration function 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

See the enumeration function for details.

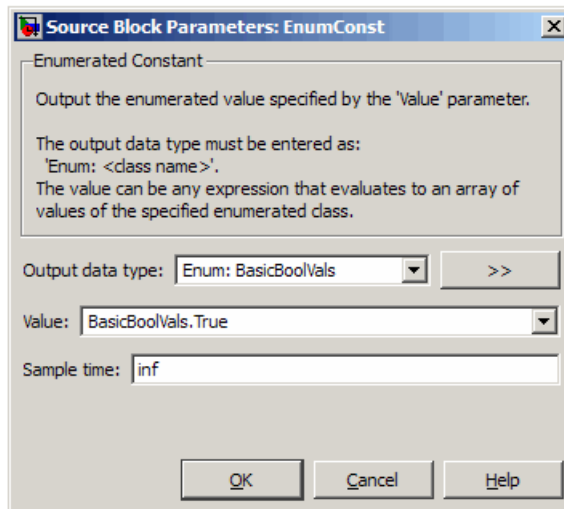
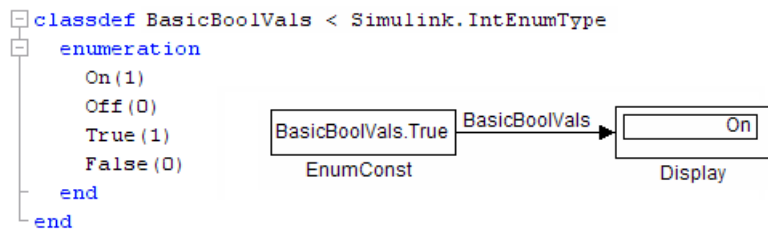
Enumeration Value Display

Wherever possible, Simulink software 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 47-12. 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:



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 47-3, you can type:

```
bcy = BasicColors.Yellow
```

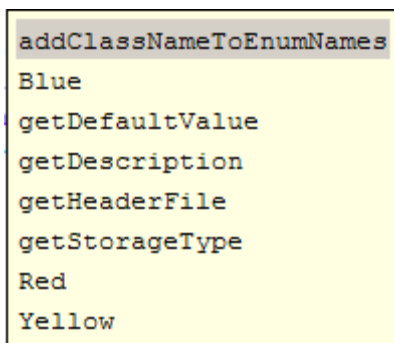
```
bcy =
```

```
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 “Tab Completion” 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 =
```

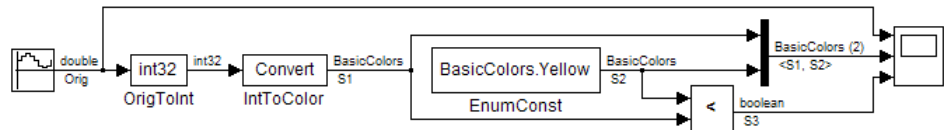
```
2
```

In either case, MATLAB returns the result of the cast in a 1x1 array of the relevant data type.

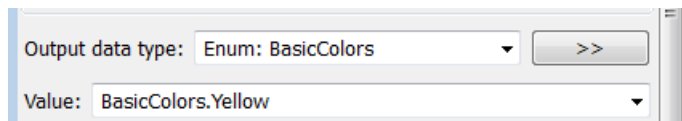
Although casting is possible, use of enumeration values is not robust in cases where enumeration values and the integer equivalents defined for an enumeration class might change.

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:

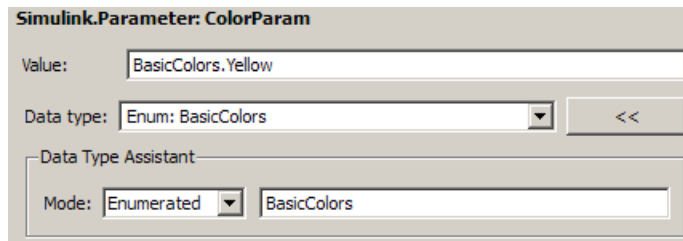


You can enter any valid MATLAB expression that evaluates to an enumerated value, including arrays and workspace variables. For example, you could enter `BasicColors(1)`, or if you had previously executed

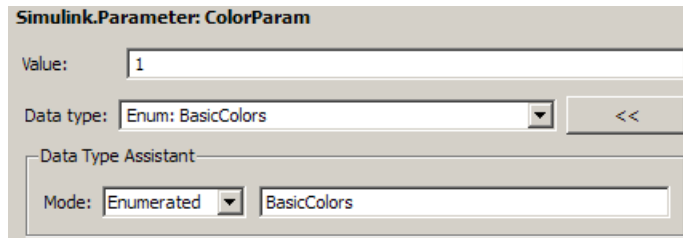
`bcy = BasicColors.Yellow` in the MATLAB Command Window, you could enter `bcy`. As another example, you could enter an array, such as `as[BasicColors.Red, BasicColors.Yellow, BasicColors.Blue]`.

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 47-12.



You *cannot* specify the integer value of an enumeration member for the **Value** parameter. See “Enumerated Values in Computation” on page 47-19 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” 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 47-19 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.

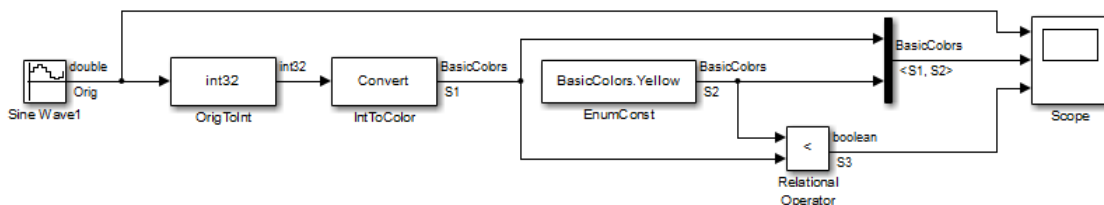
When a block like the Switch block or Multiport Switch block selects among multiple data signals, and any data signal is of an enumerated type, all the data signals must be of that same enumerated type. When a block inputs both control and data signals, as Switch and Multiport Switch do, the control 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.

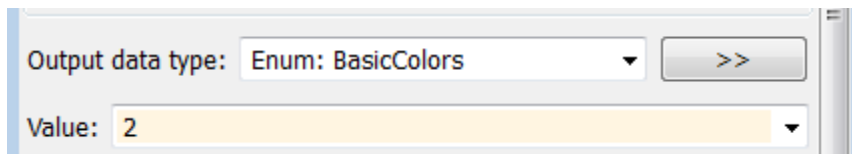
You cannot use a Data Type Conversion block to cast a numeric signal of any non-integer data type to an enumerated type. For example, the model used in “Simulate with Enumerations” on page 47-10 needed two Data Conversion blocks to convert a sine wave to enumerated values.



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.

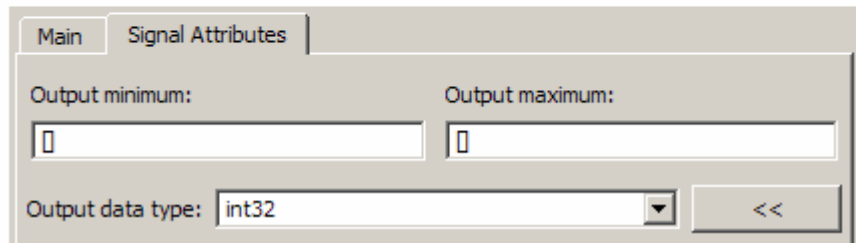
You cannot cast a block parameter of an enumeration to any other data type. For example, suppose that a Constant block specifies a **Constant value** of `BasicColors.Blue` and an **Output data type** of `int32`.



Main | Signal Attributes

Constant value:

BasicColors.Blue



Main | Signal Attributes

Output minimum: 0 Output maximum: 0

Output data type: 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.

Simulink Constructs that Support Enumerations

In this section...
“Overview” on page 47-22
“Block Support” on page 47-22
“Class Support” on page 47-24
“Logging Enumerated Data” on page 47-24
“Importing Enumerated Data” on page 47-24

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 47-19 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 47-25 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 51-22)
- 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

Root-level outports, 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 “Export Runtime Information” for more information.

Importing Enumerated Data

Root-level inports 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 “Import Data” for more information.

Simulink Enumeration Limitations

In this section...

“Enumerations and Scopes” on page 47-25

“Enumerated Types for Switch Blocks” on page 47-25

“Nonsupport of Enumerations” on page 47-25

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 software. 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 might support enumerations, but currently does not do so.
- Generated code does not support logging enumerated data.
- Custom Stateflow targets do not support enumerated types.
- HDL Coder does not support enumerations.

Importing and Exporting Simulation Data

- “Using Simulation Data” on page 48-3
- “Export Simulation Data” on page 48-4
- “Data Format for Exported Simulation Data” on page 48-9
- “Limit Amount of Exported Data” on page 48-15
- “Samples to Export for Variable-Step Solvers” on page 48-17
- “Export Signal Data Using Signal Logging” on page 48-21
- “Configure a Signal for Signal Logging” on page 48-24
- “View Signal Logging Configuration” on page 48-31
- “Enable Signal Logging for a Model” on page 48-38
- “Override Signal Logging Settings” on page 48-45
- “Access Signal Logging Data” on page 48-59
- “Techniques for Importing Signal Data” on page 48-72
- “Import Data to Model a Continuous Plant” on page 48-78
- “Import Data to Test a Discrete Algorithm” on page 48-81
- “Import Data for an Input Test Case” on page 48-82
- “Import Signal Logging Data” on page 48-86
- “Import Data to Root-Level Input Ports” on page 48-88
- “Import and Map Root-Level Inport Data” on page 48-92
- “Import MATLAB timeseries Data” on page 48-121

- “Import Structures of timeseries Objects for Buses” on page 48-123
- “Import Simulink.Timeseries and Simulink.TsArray Data” on page 48-133
- “Import Data Arrays” on page 48-134
- “Import MATLAB Time Expression Data” on page 48-136
- “Import Data Structures” on page 48-137
- “Import and Export States” on page 48-142

Using Simulation Data

Working with Simulation Data

During simulation, you can:

- Import input signal and initial state data from a workspace or file.
- Export output signal and state data to a workspace or file.

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.

Also, import data into a model for testing and analysis.

Export Simulation Data

In this section...

“Simulation Data” on page 48-4

“Approaches for Exporting Signal Data” on page 48-4

“Enable Simulation Data Export” on page 48-6

“View Logged Simulation Data With the Simulation Data Inspector” on page 48-7

“Memory Performance” on page 48-7

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 during simulation for later retrieval and postprocessing. Exporting data is also known as “data logging” or “saving simulation data.”

You can also import the exported data to use as input for simulating a model.

Approaches for Exporting Signal Data

Exporting simulation data very often involves exporting signal data. You can use a variety of approaches for exporting signal data.

Export Approach	Usage	Documentation
Connect a signal to a Scope block or viewer.	<p>If you use a Scope block for viewing results during simulation, consider also using the Scope block to export data.</p> <p>Document in the diagram the workspace variables used to store signal data.</p>	Scope

Export Approach	Usage	Documentation
	<p>Save output at a sample rate other than the base sample rate.</p> <p>Scopes store data and can be memory intensive.</p>	
<p>Connect a signal to a To File block.</p>	<p>Consider using a To File block for exporting large amounts of data.</p> <p>Document in the diagram the workspace variables used to store signal data.</p> <p>Save output at a sample rate other than the base sample rate.</p> <p>Use the MAT-file only after the simulation has completed.</p>	<p>To File</p>
<p>Connect a signal to a To Workspace block.</p>	<p>Document in the diagram the workspace variables used to store signal data.</p> <p>Save output at a sample rate other than the base sample rate.</p>	<p>To Workspace</p>
<p>Connect a signal to a root-level Outport block.</p>	<p>Consider using this approach for logging data in a top-level model, if the model already includes an Outport block.</p>	<p>Outport</p>

Export Approach	Usage	Documentation
Set the signal logging properties for a signal.	<p>Use signal logging to avoid adding blocks.</p> <p>Log signals based on individual signal rates.</p> <p>Data is available only when simulation is paused or completed.</p> <p>Use signal logging to log array of buses signals.</p>	“Export Signal Data Using Signal Logging” on page 48-21
Configure Simulink to export time, state, and output data.	<p>Consider exporting this data to capture information about the simulation as a whole.</p> <p>Outputs and states are logged at the base sample rate of the model.</p>	<p>“Data Format for Exported Simulation Data” on page 48-9</p> <p>“Limit Amount of Exported Data” on page 48-15</p> <p>“Samples to Export for Variable-Step Solvers” on page 48-17</p>
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 49-27
Use the <code>sim</code> command to log simulation data programmatically.	<p>Use <code>sim</code> to export to one data object the time, states, and signal simulation data.</p> <p>Select the Return as single object parameter when simulating the model using the <code>sim</code> command inside a function or a <code>parfor</code> loop.</p>	<code>sim</code>

Enable Simulation Data Export

To export the states and root-level outputs of a model to the MATLAB base workspace during simulation of the model, use one of these interfaces:

- **Configuration Parameters > Data Import/Export** pane (for details, see “Data Import/Export Pane”)
- `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 Exported Simulation Data” on page 48-9
 - “Limit Amount of Exported Data” on page 48-15
 - “Samples to Export for Variable-Step Solvers” on page 48-17

View Logged Simulation Data With the Simulation Data Inspector

To inspect exported simulation data interactively, consider using the Simulation Data Inspector tool.

The Simulation Data Inspector has some limitations on the kinds of logged data that it displays. See “Limitations of the Simulation Data Inspector” on page 17-56.

Memory Performance

When exporting simulation data, 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 time stamp, the length of the interval (time step), and the total number of time stamps.
- When multiple signals use identical sequences of time stamps, the signals share a single stored time stamp sequence. This may reduce memory use for logged data by as much as a factor of two. An example when this memory performance can be a critical performance factor is when logging bus signals that have thousands of bus elements.

Data Format for Exported Simulation Data

In this section...

“Data Format for Block-Based Exported Data” on page 48-9

“Data Format for Model-Based Exported Data” on page 48-9

“Signal Logging Format” on page 48-9

“Logged Data Store Format” on page 48-10

“State and Output Data Format” on page 48-10

Data Format for Block-Based Exported Data

You can use the Scope, To File, or To Workspace blocks to export simulation data. Each of these blocks has a parameter that you use to specify the data format.

Data Format for Model-Based Exported 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 48-9
- “Logged Data Store Format” on page 48-10
- “State and Output Data Format” on page 48-10

Signal Logging Format

Use the Dataset format for signal logging data in new models. Select the format using the **Configuration Parameters > Data Import/Export > Signal logging format** parameter.

For details, see “Specify the Signal Logging Data Format” on page 48-39.

For backwards compatibility, Simulink supports the `ModelDataLogs` format for signal logging. The `ModelDataLogs` format will be removed in a future release. For details, see “Migrate from ModelDataLogs to Dataset Format” on page 48-40.

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 49-30.

State and Output Data Format

For exported state, final state, and output data, use one of the following formats:

- “Array” on page 48-10
- “Structure with Time” on page 48-11
- “Structure” on page 48-14
- “Per-Port Structure” on page 48-14

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 — Right-click the signal and in the Properties dialog box, select Log signal. For details, see “Export Signal Data Using Signal Logging” on page 48-21.
- 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.

Array

If you select this Array option, Simulink saves the states and outputs of a model in a state and output array, respectively.

The state matrix has the name specified in the **Save to workspace** area (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 contains 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 **Save to workspace** area (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 outputs are:

- Either all scalars or all vectors (or all matrices for states)
- Either all real or all complex
- All of the same data type

Use the `Structure` or `Structure with time` output formats (see “Structure with Time” on page 48-11) if your model outputs and states do not meet these conditions.

Structure with Time

If you select this format, Simulink saves the model states and outputs in structures that have their names specified in the **Save to workspace** area. 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 the outputs are scalars or vectors, the **values** field is a matrix each of whose rows represents an output at the time specified by the corresponding element of the time vector.
- If the outputs are matrix (2-D) values, the **values** field is a 3-D array of dimensions M-by-N-by-T where 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 or the type of state (continuous or discrete).

- **blockName**

Specifies the name of the corresponding output port or block with states.

- **inReferencedModel**

Contains a value of 1 if the **signals** field records the final state of a block that resides in the submodel. 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 48-140 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.

Per-Port Structure

This format consists of a separate structure-with-time or structure-without-time for each output port. Each output data structure has only one `signals` field. To specify this option, enter the names of the structures in the **Output** text field as a comma-separated list, `out1, out2, ..., outN`, where `out1` is the data for your model's first port, `out2` for the second input port, and so on.

Limit Amount of Exported Data

In this section...

“Decimation” on page 48-15

“Limit Data Points to Last” on page 48-15

Decimation

To skip samples when exporting data, apply a decimation factor. For example, a decimation factor of 2 saves every other sample.

The approach you use to specify a decimation factor depends on the kind of logging data.

Kind of 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 Last

To limit the number of samples saved to be only the most recent samples, set the **Limit Data Points to Last** parameter.

The approach you use depends on the kind of logging data.

Kind of 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.
State and output	Enter a value in the field to the right of the Limit Data Points to Last label.

Samples to Export for Variable-Step Solvers

In this section...

“Output Options” on page 48-17

“Refine Output” on page 48-17

“Produce Additional Output” on page 48-18

“Produce Specified Output Only” on page 48-19

Output Options

Use the **Output options** list on the **Data Import/Export** configuration pane 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

For more information, see “Data Import/Export Pane”.

Refine Output

The **Refine output** option provides additional output points when the simulation output is too coarse. 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 as well as 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 is capable of taking 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 4 should 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 3-24).

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** 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 needs to more accurately simulate the model.

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 3-24).
- Set the **Output times** 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

In Normal and Accelerator mode, 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 `[0:10]` 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 may 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 3-24).
- Set the **Output times** parameter to a value other than the default empty matrix (`[]`).

- In Normal and Accelerator mode, 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 of the time steps in between the values that you specify, and the simulation start and stop times.
- For triggered subsystems and function-call subsystems, the calling function must inherit the sampling rate.

Export Signal Data Using Signal Logging

In this section...

“Signal Logging” on page 48-21

“Signal Logging Workflow” on page 48-21

“Signal Logging in Rapid Accelerator Mode” on page 48-22

“Signal Logging for Array of Buses Signals” on page 48-22

“Signal Logging Limitations” on page 48-23

Signal Logging

Use signal logging to capture signal data from a simulation. To enable signal logging, set the signal logging property for individual signals and enable signal logging for the model. For details, see “Configure a Signal for Signal Logging” on page 48-24 and “Enable Signal Logging for a Model” on page 48-38.

For a summary of other approaches to capture signal data, see “Export Simulation Data” on page 48-4.

Signal Logging Workflow

Signal logging involves this general workflow:

- 1** Configure individual signals for signal logging, including controlling how much output the simulation generates. See “Configure a Signal for Signal Logging” on page 48-24.
- 2** (Optional) Review the signal logging configuration for a model. See “View Signal Logging Configuration” on page 48-31.
- 3** Enable signal logging globally for a model. See “Enable Signal Logging for a Model” on page 48-38.
- 4** Specify the format for the logged signal data. See “Specify the Signal Logging Data Format” on page 48-39.
- 5** (Optional) Specify a name for the signal logging data. See “Specify Logging Name” on page 48-27.

- 6 (Optional) Specify which samples to export for models with variable-step solvers. See “Limit the Data Logged for a Signal” on page 48-29.
- 7 (Optional) Configure the model to display signal logging data in the Simulation Data Inspector. See “View Logged Signal Data with the Simulation Data Inspector” on page 48-60.
- 8 (Optional) Override the signal logging property settings in the model or referenced models. See “Override Signal Logging Settings” on page 48-45.
- 9 Simulate the model.
- 10 Access the signal logging data. See “Access Signal Logging Data” on page 48-59.

Signal Logging in Rapid Accelerator Mode

Use the Dataset format for signal logging data in Rapid Accelerator mode. Select the format using the **Configuration Parameters > Data Import/Export > Signal logging format** parameter.

For details, see “Specify the Signal Logging Data Format” on page 48-39.

Signal logging in Rapid Accelerator mode does not log:

- Signals inside referenced models
- Bus signals or array of buses signals

When you update or simulate a model that contains signals in referenced models, bus signals, or array of buses signals, Simulink displays a warning that those signals are not logged.

You cannot use signal logging in Rapid Accelerator mode if you set the **Configuration Parameters > Solver > Sample time constraint** parameter to Ensure sample time independent.

Signal Logging for Array of Buses Signals

Use the Dataset format for logging array of buses signals. Select the format using the **Configuration Parameters > Data Import/Export > Signal logging format** parameter.

For details, see “Specify the Signal Logging Data Format” on page 48-39.

Simulink does not log array of buses signals 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 48-22.
- Top-model software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes support signal logging. For a description of limitations of signal logging using SIL and PIL modes, see “Internal Signal Logging Support” in the Embedded Coder documentation.
- Array of buses signals support signal logging, with the requirements and limitations described in “Signal Logging for Array of Buses Signals” on page 48-22.
- You cannot log signals in For Each subsystems.
- You cannot log local data in Stateflow Truth Table blocks.

Configure a Signal for Signal Logging

In this section...

“Enable Logging for a Signal” on page 48-24

“Specify Logging Name” on page 48-27

“Limit the Data Logged for a Signal” on page 48-29

Enable Logging for a Signal


Enable logging for a signal with one of the following techniques:

- “Enable Signal Logging Using Simulink Editor Menu Options” on page 48-24
- “Enable Signal Logging Using the Signal Properties Dialog Box” on page 48-25
- “Enable Signal Logging Using Model Explorer” on page 48-25
- “Programmatic Interface” on page 48-26

You can use the Simulink Editor menu options or the Model Explorer to enable signal logging for several signals at once, compared to opening the Signal Properties dialog box repeatedly for each signal that you want to log.


However, the Model Explorer:

- Does not display unnamed signals
- Might need to be reconfigured to display the `DataLogging` property (which sets up logging for a signal)

A signal for which you enable signal logging is a *logged signal*. By default, Simulink displays a logged signal indicator  for each logged signal.

Enable Signal Logging Using Simulink Editor Menu Options

- 1 In the Simulink Editor, select one or more signals.

2 Click the **Record** button arrow  and click **Log/Unlog Selected Signals**.

Alternatively, you can select one or more signals and check **Simulation > Output > Log/Unlog Selected Signals**.

If you select multiple signals, the signal logging configuration that Simulink sets depends on whether any of the selected signals are already configured with signal logging enabled.

Signal Logging for Selected Signals	Result of Enabling the Log/Unlog Selected Signals Option
At least one of the selected signals does not have signal logging enabled.	Enables signal logging for all of the selected signals
All selected signals have signal logging enabled.	Disables signal logging for all of the selected signals

Enable Signal Logging Using the Signal Properties Dialog Box

- 1** In the Simulink Editor, right-click the signal.
- 2** From the context menu, select **Signal Properties**.
- 3** In the Signal Properties dialog box, in the **Logging and accessibility** tab, select **Log signal**.
- 4** Click **Log signal**.

Enable Signal Logging Using Model Explorer

Note The only signals that Model Explorer displays are named signals. See “Signal Names” on page 50-19.

- 1** In the Model Explorer **Model Hierarchy** pane, select the node that contains the signal for which you want to enable signal logging.

- 2 If the **Contents** pane does not display the `DataLogging` property, set the **Column view** to **Signals** or add the `DataLogging` property to the current view. For details about column views, see “Control Model Explorer Contents Using Views” on page 9-26.
- 3 Enable the `DataLogging` property for one or more signals.

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 to log. For example, for the Mu block output signal.

```
ph = get_param('vdp/Mu','PortHandles')
```

- 3 Enable signal logging for the desired output signal.

```
set_param(ph.Output(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 48-50.

If you use the `ModelDataLogs` signal logging format, you can also use test points to log data in referenced models (but not for a top-level model). However, when using the `ModelDataLogs` format in a referenced model with root-level function calls, the logged data is incorrect. Instead, use the `Dataset` format for signal logging. The `Dataset` format supports using test points for logging in referenced models for signals configured for signal logging.

For more information on test points, see “Designating a Signal as a Test Point” on page 50-59. For a description of the benefits of using the Dataset format, see “Signal logging format”.

Specify Logging Name

You can specify a name, called the logging name, to the object that Simulink uses to store logging data for a signal. Specifying a logging name can be useful for signals that are unnamed or that share a duplicate name with another signal in the model hierarchy. Specifying logging names, rather than using the logging names that Simulink generates, can make the logged data easier to analyze.

Note The logging name is for a specific signal. The logging name is distinct from the signal logging name, which is the name for the object containing all the logged signal data for a model. The default signal logging name is `logout`. For details about the signal logging name, see “Specify a Name for the Signal Logging Data” on page 48-44.

To specify a logging name, use *one* of the following approaches:

- “Using the Signal Properties Dialog Box to Specify a Logging Name” on page 48-28
- “Using the Model Explorer to Specify a Logging Name” on page 48-28
- “Programmatically Specifying a Logging Name” on page 48-28

If you do not specify a custom logging name, Simulink uses the signal name. If the signal does not have a name, the action Simulink takes depends on the signal logging format:

- **Dataset** — Uses a blank name
- **ModelDataLogs** — Generates a default name that is composed of the block name and port number. For example, if the block name is `MyBlock` and the signal being logged is the first output of this block, Simulink generates the name `SL_MyBlock1`.

Using the Signal Properties Dialog Box to Specify a Logging Name

- 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.

Using the Model Explorer to Specify a Logging Name

- 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 “Control Model Explorer Contents Using Views” on page 9-26.
- 3 Enter a logging name for one or more signals using the `LoggingName` column.

Programmatically Specifying a Logging Name

Enable signal logging programmatically for selected blocks with the output `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 to log. For example, for the Mu block output signal:

```
ph = get_param('vdp/Mu','PortHandles');
```

- 3 Enable signal logging for the desired output signal:

```
set_param(ph.Outputport(1), 'DataLogging', 'on');
```

The logged signal indicator appears.

- 4 Issue commands that use the `DataLoggingNameMode` and `DataLoggingName` parameters. For example:

```
set_param(ph.Outputport(1), 'DataLoggingNameMode', 'Custom');  
set_param(ph.Outputport(1), 'DataLoggingName', 'x2_log');
```

Limit the Data Logged for a Signal

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

For details, see “Limit Amount of Exported Data” on page 48-15.

To limit data logged for a signal, use either the Signal Properties dialog box or the Model Explorer. The following two sections describe how to use each of these interfaces.

Using the Signal Properties Dialog Box to Limit the Amount of Data Logged

- 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**

Using the 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 “Control Model Explorer Contents Using Views” on page 9-26.
- 3** To specify a decimation factor, edit the `Decimation` and `DecimateData` properties. To limit the number of samples logged, edit the `LimitDataPoints` property.

View Signal Logging Configuration

In this section...

“Approaches for Viewing the Signal Logging Configuration” on page 48-31

“Use Simulink Editor to View Signal Logging Configuration” on page 48-32

“Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34

“Use Model Explorer to View Signal Logging Configuration” on page 48-36

Approaches for Viewing the Signal Logging Configuration

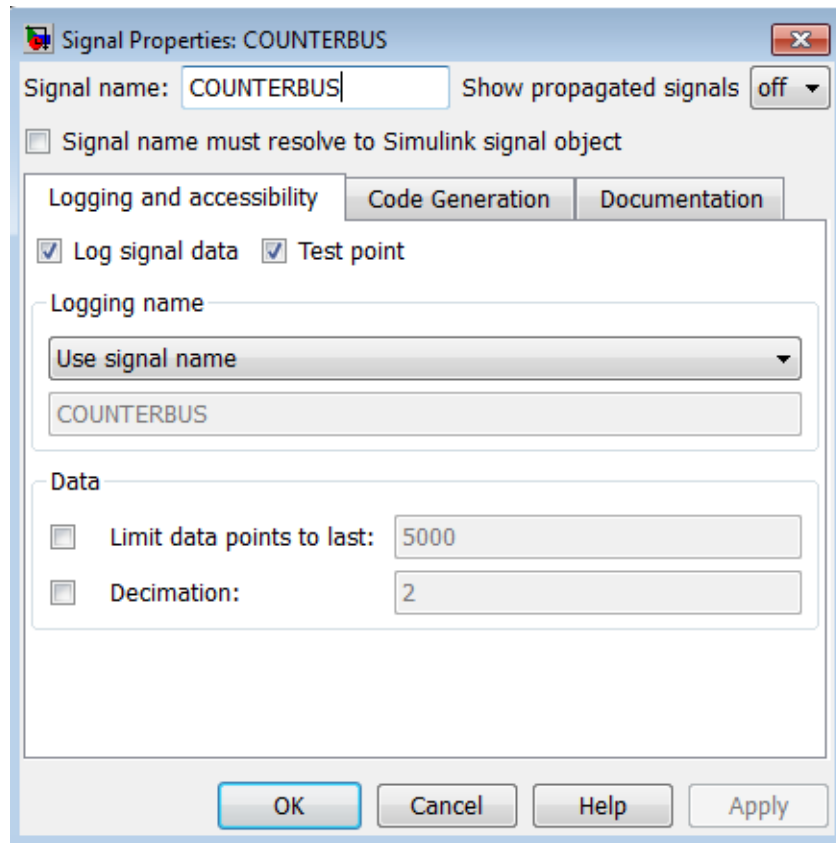
Signal Logging Configuration Viewing Approach	Usage	Documentation
In the Simulink Editor, view signal logging indicators.	<p>Consider using this approach for models with few signals configured for signal logging and the model hierarchy is shallow.</p> <p>This approach avoids leaving the Simulink Editor.</p> <p>You need to open the Signal Properties dialog box for each signal.</p>	“Use Simulink Editor to View Signal Logging Configuration” on page 48-32
Use the Signal Logging Selector.	<p>Consider using for models with deep hierarchies.</p> <p>View a model that has signal logging override settings for some signals.</p> <p>View the configuration as part of specifying a subset of</p>	“Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34

Signal Logging Configuration Viewing Approach	Usage	Documentation
	<p>signals to log from all signals configured for signal logging.</p> <p>View signal logging configuration without displaying the signal logging indicators in the model.</p> <p>View signal logging configuration information such as decimation and output options in one window.</p>	
<p>Use the Model Explorer.</p>	<p>View signal logging configuration in the context of other model component properties.</p> <p>You may need to adjust the column view to display signal logging properties.</p>	<p>“Use Model Explorer to View Signal Logging Configuration” on page 48-36</p>

Use Simulink Editor to View Signal Logging Configuration

By default, Simulink Editor displays an indicator on each signal that is set up to log signal data. To view the signal logging setting for a signal:

- 1** Right-click the signal, and from the context menu, select **Signal Properties**.
- 2** Select the **Logging and accessibility** tab.



For example, in the following model the output of the Sine Wave block is logged:



If you use the command-line interface to override logging for a signal, the Simulink Editor continues to display the signal logging indicator for that signal in the Simulink Editor. When you simulate the model, Simulink displays a red signal logging indicator for all signals configured to be logged,

reflecting any overrides. For details about configuring a signal for logging, see “Configure a Signal for Signal Logging” on page 48-24.

A logged signal can also be a test point. See “Test Points” on page 50-59 for information about test points.

To turn the display of logging indicators off, clear **Display > Signals & Ports > Testpoint & Logging Indicators**.

Use Signal Logging Selector to View Signal Logging Configuration




- 1 Open the model for which you want to view the signal logging configuration.
- 2 Open the Signal Logging Selector, using one of the following approaches:
 - In the **Configuration Parameters > Data Import/Export** pane, in the **Signals** area, select the **Configure Signals to Log** button.
If necessary, select **Signal logging** to enable the **Configure Signals to Log** button.
 - 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**.
- 3 In the **Model Hierarchy** pane, select the model node for which you want to view the signal logging configuration. For example:



To expand a node in the **Model Hierarchy** pane, right-click that the arrow to the left of the node.

If there are no logged signals for a node, the **Contents** pane is empty.

If you open the Signal Logging Selector for a model that uses model referencing, then in the **Model Hierarchy** pane, the check box to the left of a model node indicates the override configuration of the node.




Check Box	Signal Logging Configuration
	<p>For the top-level model node, logs all logged signals in the top model.</p> <p>For a Model block node, logs all logged signals in the model reference hierarchy for that block.</p>
	<p>For the top-level model node, disables logging for all logged signals in the top model.</p> <p>For a Model block node, logs disables logging for all signals in the model reference hierarchy for that block.</p>
	<p>For the top-level model node, logs all logged signals that have the DataLogging setting enabled.</p> <p>For a Model block node, logs all logged signals in the model reference hierarchy for that block that have the DataLogging setting enabled.</p>

Viewing the Signal Logging Configuration for Subsystems, Masked 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 all that include logged signals
Masked subsystem	Does not display
Linked library	Displays all that include logged signals

You can control how the **Model Hierarchy** pane displays subsystems, masked subsystems, and linked libraries. To do control how, 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 “Displaying Linked Library Subsystems” on page 9-13 and “Displaying Masked Subsystems” on page 9-14.

- To display all subsystems, including subsystems that do not include logged signals, 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

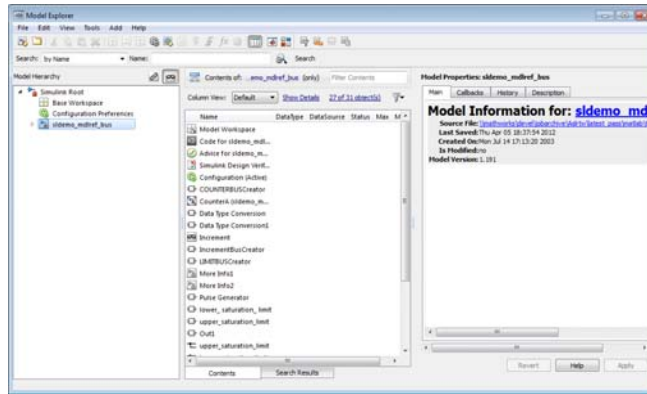
To find a specific signal or property value for a signal, use the **Filter Contents** edit box. Use the same approach as you use in the Model Explorer; for details, see “Filtering Contents” on page 9-48.

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**.

Use Model Explorer to View Signal Logging Configuration

- 1 Open the model for which you want to view the signal logging configuration. Select the top-level model in a model reference hierarchy gives you to access to logging configuration information for referenced models.
- 2 In the **Contents** pane, set **Column View** to the **Signals** view.



For further information, see “Model Explorer: Model Hierarchy Pane” on page 9-9 and “Model Explorer: Contents Pane” on page 9-19.

Enable Signal Logging for a Model

In this section...

“Enable and Disable Logging Globally for a Model” on page 48-38

“Specify the Signal Logging Data Format” on page 48-39

“Specify a Name for the Signal Logging Data” on page 48-44

Enable and Disable Logging Globally for a Model

To log a signal, you must configure it for signal logging. For details, see “Configure a Signal for Signal Logging” on page 48-24.

Enable or disable logging globally for all signals that you configure for signal logging in a model. By default, signal logging is enabled. Simulink logs signals only if **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.

To disable signal logging, use *one* of these approaches.

- Clear the **Configuration Parameters > Data Import/Export > Signal logging** parameter.
- Use the `SignalLogging` parameter. For example:

```
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 48-45.

Specify the Signal Logging Data Format

The signal logging format determines how Simulink stores the logged signal data. You can store the data using either the Dataset or ModelDataLogs format.

Set the Signal Logging Format

To set the signal logging format, use either of these approaches:

- Set the **Configuration Parameters > Data Import/Export > Signal logging format** parameter to either Dataset (default) or ModelDataLogs.
- Use the `SignalLoggingSaveFormat` parameter, with a value of Dataset or ModelDataLogs. For example:

```
set_param(bdroot, 'SignalLoggingSaveFormat', 'Dataset')
```

Use the Dataset Format for Signal Logging in New Models

Use the Dataset format for signal logging for new models. The ModelDataLogs format is supported for backwards compatibility. The ModelDataLogs format will be removed in a future 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 timeseries methods such as `filter`, `detrend`, and `resample`.
- Supports logging multiple data values for a given time step, which can be important for Iterator subsystem and Stateflow signal logging.
- Provides an easy to analyze format for logged signal data for models with deep hierarchies, bus signals, and signals with duplicate or invalid names.
- Supports signal logging in Rapid Accelerator mode.
- Avoids the limitations of the ModelDataLogs format, which Bug Report 495436 describes.

When you specify the Dataset format, Simulink stores the data using a `Simulink.SimulationData.Dataset` object.

Note You must use the `Dataset` format to log arrays of buses.

Migrate from ModelDataLogs to Dataset Format

The `ModelDataLogs` logging format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release. To enable existing models that use `ModelDataLogs` format to continue to work in future releases, migrate those models to use `Dataset` format.

Use the Upgrade Advisor to upgrade a model to use `Dataset` format, using *one* of these approaches:

- In the Simulink Editor, select **Analysis > Model Advisor > Upgrade Advisor**
- From the MATLAB command line, use the `upgradeadvisor` function.

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 dataset to use `Dataset` format. For example, to update the `older_model_dataset` from `ModelDataLogs` format to `Dataset` format:

```
new_dataset = logout.convertToDataset('older_model_dataset')
```

Depending upon your particular circumstances, converting a model from using `ModelDataLogs` format to using `Dataset` format may require that you make some modifications to your existing models and to code in callbacks, functions, scripts, or tests. The following table identifies possible issues that you may need 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 <code>ModelDataLogs</code> programmatic interface to access data may result in an error.	<p>Check for code that uses <code>ModelDataLogs</code> format access methods. Update that code to use <code>Dataset</code> format access methods.</p> <p>For example, suppose existing code includes the following line:</p> <pre>logout.('Subsystem Name').X.data</pre> <p>Replace that code with a <code>Dataset</code> access method:</p> <pre>logout.get('x').Values.Data</pre> <p>or</p> <pre>logout.getElement('x').Values.Data</pre>
Logging bus signals requires a configuration parameter change.	<p>Logging buses in <code>Dataset</code> format requires that Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals be set to error.</p> <p>To configure a model for proper bus usage, run the Upgrade Advisor with the Check bus usage check.</p>
Mux block signal names are lost.	<p>The <code>Dataset</code> format treats Mux block signals as a vector.</p> <p>If you need to identify signals by signal names, replace Mux blocks with a Bus Creator blocks. Set Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals to error.</p>

Possible Issue After Conversion to Dataset Format	Solution
Signal Viewer cannot be used for signal logging.	If you use the Dataset format for signal logging, then Simulink does not log the signals configured to be logged in the Signal Viewer. Configure the signal for signal logging.
The unpack method generates an error.	<p>The unpack method, which is supported for <code>Simulink.ModelDataLogs</code> and <code>Simulink.SubsysDataLogs</code> objects, is <i>not</i> supported for <code>Simulink.SimulationData.Dataset</code> objects.</p> <p>For example, if the logged data has three fields: x, y, and z, then:</p> <ul style="list-style-type: none"> • For <code>ModelDataLogs</code> format data, the <code>mlog.unpack</code> method creates three variables in the base workspace. • For <code>Dataset</code> format data, access methods by names. For example: <code>x = logouts.get('x').Values</code>
The <code>ModelDataLogs</code> and <code>Dataset</code> formats have different naming rules for unnamed signals.	<p>If necessary, add signal names.</p> <ul style="list-style-type: none"> • In <code>ModelDataLogs</code> format, for an unnamed signal coming from a block, Simulink assigns a name in this form: <code>SL_BlockName+<portIndex></code> For example, <code>SL_Gain1</code>. • In <code>Dataset</code> format, elements do not need a name, so Simulink leaves the signal name empty. • For both <code>ModelDataLogs</code> and <code>Dataset</code> formats, Simulink assigns the same name

Possible Issue After Conversion to Dataset Format	Solution
	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.
Running or updating a model that uses model referencing might return a signal logging format inconsistency error.	Follow the approach described in “Model Reference Signal Logging Format Consistency” on page 48-43.

Model Reference Signal Logging Format Consistency

If signal logging is enabled for a top model, then the signal logging format for the non-protected referenced models must be the same as the signal logging format for the top model.

Simulink performs signal logging format consistency checking during model update or when you run a simulation. Simulink does not report inconsistencies during code generation for model reference simulation target code.

If Simulink reports a signal logging format inconsistency, then use *one* of the following approaches:

- Use the Upgrade Advisor (with the `upgradeadvisor` function) to upgrade a model to use Dataset format.
- Use the `Simulink.SimulationData.updateDatasetFormatLogging` function to convert a model and its referenced models to use Dataset format for signal logging.
- Turn off signal logging for the model, including for all referenced models, by clearing the **Configuration Parameters > Data Import/Export > Signal logging** parameter check box.
- Disable logging for all signals in this top-level Model block.

- 1 Select the **Configuration Parameters > Data Import/Export > Configure Signals to Log** button.
- 2 In the Signal Logging Selector dialog box, in the **Model Hierarchy** pane, clear the check box for the top Model block in the model reference hierarchy that generated the error.

Specify a Name for the Signal Logging Data

You use the signal logging name to access the signal logging data. The default name for the signal logging data is `logout`. Specifying a 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_logout` to identify the data as being the signal logging data for the car model.

To specify a different 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')
```

Override Signal Logging Settings

In this section...

“Benefits of Overriding Signal Logging Settings” on page 48-45
“Two Interfaces for Overriding Signal Logging Settings” on page 48-45
“Scope of Signal Logging Setting Overrides” on page 48-46
“Signal Logging Selector” on page 48-47
“Command-Line Interface for Overriding Signal Logging Settings” on page 48-53

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. Override signal logging properties to reduce memory overhead and to facilitate the analysis of simulation logging results.

Overriding signal logging properties is useful when you want to:

- Focus on only a few logged signals by disabling logging for most of the logged signals
- 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:

- “Signal Logging Selector” on page 48-47
- “Command-Line Interface for Overriding Signal Logging Settings” on page 48-53

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.ModelDataLoggingInfo` 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`.

If you close and 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.

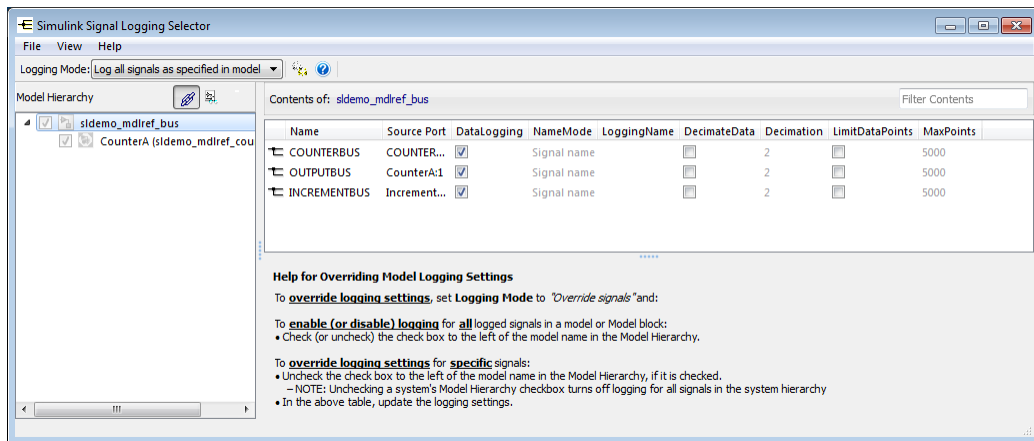
To use the signal logging settings that the model defines, set the logging mode to use the logging settings defined in the model. That setting causes all logged signals to be included in the signal logging data, ignoring any overrides. Using the Signal Logging Selector to override logging for a specific signal does not affect the signal logging indicator for that signal.

Note Simulink rebuilds a model in the following situation:

- 1** The model contains one or more signals configured for signal logging.
 - 2** You simulate the model in Rapid Accelerator mode.
 - 3** You use the Signal Logging Selector to modify the signal logging configuration.
 - 4** You simulate the model in Rapid Accelerator mode again.
-

Signal Logging Selector


- 1** Open the Signal Logging Selector, using one of the following approaches:
 - In the **Configuration Parameters > Data Import/Export** pane, in the **Signals** area, select the **Configure Signals to Log** button.
If necessary, select **Signal logging** to enable the **Configure Signals to Log** button.
 - 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.

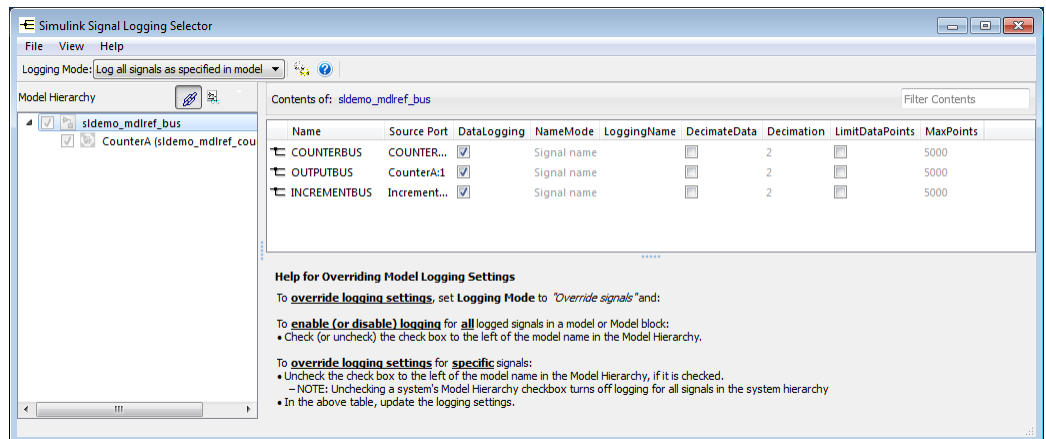
- View the node containing the logged signals that you want to override. If necessary, expand nodes or configure the Model Hierarchy pane to display masked subsystems. See "Use Signal Logging Selector to View Signal Logging Configuration" on page 48-34.
- Override signal logging settings. Use one of the following approaches, depending on whether or not your model uses model referencing:
 - "Models Without Model Referencing: Overriding Signal Logging Settings" on page 48-49
 - "Models with Model Referencing: Overriding Signal Logging Settings" on page 48-50

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, in the **Signals** area, select 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 "Use Signal Logging Selector to View Signal Logging Configuration" on page 48-34.

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 the one or more of the following procedures:

- “Enable Logging for All Logged Signals” on page 48-50
- “Disable Logging for All Logged Signals in a Model Node” on page 48-51
- “Override Signal Logging for a Subset of Signals” on page 48-52
- “Override Other Signal Logging Properties” on page 48-52

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 either solid blue () or empty () .

To enable logging of all logged signals for a node:

- 1** Open the Signal Logging Selector. In the **Configuration Parameters > Data Import/Export** pane, in the **Signals** area, select 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 “Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34.
- 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 Logged Signals in a Model 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 either checked () or solid blue ()

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, in the **Signals** area, select 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 “Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34.
- 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 reference 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, in the **Signals** area, select 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 “Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34.
- 4 In the **Model Hierarchy** pane, ensure that the check box for the top-level model or Model block is either solid blue  or . 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, in the **Signals** area, select 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 “Use Signal Logging Selector to View Signal Logging Configuration” on page 48-34.

- 4 In the **Model Hierarchy** pane, ensure that the check box for the top-level model or Model block is solid blue  or . 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.

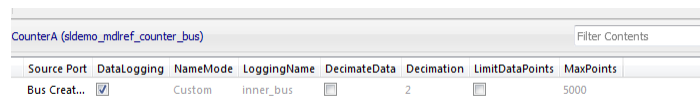
To enable logging for multiple signals, hold the **Shift** or **Ctrl** key and select a range of signals or individual signals. Select the check box in the DataLogging column of one of the highlighted signals.

- 6 In the **Contents** pane table, modify the settings for properties, such as DecimateData and Decimation.

Command-Line Interface for Overriding Signal Logging Settings

The 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:



Source Port	DataLogging	NameMode	LoggingName	DecimateData	Decimation	LimitDataPoints	MaxPoints
Bus Creat...	<input checked="" type="checkbox"/>	Custom	inner_bus	<input type="checkbox"/>	2	<input type="checkbox"/>	5000

- 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 48-54
- “Specify Which Models to Log” on page 48-55
- “Log a Subset of Signals” on page 48-56
- “Override Other Signal Logging Properties” on page 48-58

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_mdhref_counter_bus')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_bus_logging')));
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);
close_system('ex_bus_logging');
close_system('ex mdlref_counter_bus');
```

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

Use the `LoggingMode` property of a `Simulink.SimulationData.ModelLoggingInfo` object to specify whether to use the signal logging settings as specified in the model and all referenced models, or to override those settings.

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, as described in the `Simulink.SimulationData.ModelLoggingInfo` documentation.

The following 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', 'e
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
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', 'e
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
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);
close_system('ex_bus_logging');
close_system('ex_mdref_counter_bus');
```

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 data logging information object.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
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);
close_system('ex_bus_logging');
close_system('ex_mdhref_counter_bus');
```

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', 'e
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
```

```
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
    'ex_bus_logging');
pos = mi.findSignal({'ex_bus_logging/CounterA' ...
    'ex_mdhref_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);
close_system('ex_bus_logging');
close_system('ex_mdhref_counter_bus');
```

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', 'e
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'e
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel...
    ('ex_bus_logging');
pos = mi.findSignal({'ex_bus_logging/CounterA' ...
    'ex_mdhref_counter_bus/Bus Creator'}, 1);
mi.Signals(pos).LoggingInfo.DecimateData = true;
mi.Signals(pos).LoggingInfo.Decimation = 2;
```

Access Signal Logging Data

In this section...

“View Signal Logging Data” on page 48-59

“Signal Logging Object” on page 48-60

“View Logged Signal Data with the Simulation Data Inspector” on page 48-60

“Programmatically Access Logged Signal Data Saved in Dataset Format” on page 48-61

“Programmatically Access Logged Signal Data Saved in ModelDataLogs Format” on page 48-67

View Signal Logging Data

To generate signal logging data, simulate the model. You can view the signal logging data for a paused or completed simulation, using one of these interfaces:

- The Simulation Data Inspector
- Programmatically, using MATLAB commands

To access signal logging data programmatically, the approach you use depends on the signal logging data format (Dataset or ModelDataLogs). For details, see:

- “Programmatically Access Logged Signal Data Saved in Dataset Format” on page 48-61
- “Programmatically Access Logged Signal Data Saved in ModelDataLogs Format” on page 48-67

Note If you do not see logging data for a signal that you configured in the model for signal logging, check the logging configuration using the Signal Logging Selector. Use the Signal Logging Selector to enable logging for a signal whose logging is overridden. For details, see “View Signal Logging Configuration” on page 48-31 and “Override Signal Logging Settings” on page 48-45.

Signal Logging Object

Simulink saves signal logging data in a signal logging object, which you access with a MATLAB workspace variable.

The type of the signal logging object depends on the signal logging format that you choose. For details, see “Specify the Signal Logging Data Format” on page 48-39.

- Dataset format — Uses a `Simulink.SimulationData.Dataset` object
- ModelDataLogs format — Uses a `Simulink.ModelDataLogs` object

The default name of the signal logging variable is `logout`. You can change the signal logging name. For details, see “Specify a Name for the Signal Logging Data” on page 48-44.

View Logged Signal Data with the Simulation Data Inspector

You can use the Simulation Data Inspector to view logged signal data.

To view logged signal data with the Simulation Data Inspector, in the Simulink Editor, use one of the following approaches:

- To display logged signals whenever a simulation ends or when you pause a simulation, select **Simulation > Model Configuration Parameters > Inspect signals when simulation is stopped/paused**.
- To launch the Simulation Data Inspector tool to display the data immediately, select **Simulation > Output > Simulation Data Inspector**.

For additional information about using the Simulation Data Inspector, see “Inspect Signal Data with Simulation Data Inspector” on page 17-2.

Programmatically Access Logged Signal Data Saved in Dataset Format

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 consists of a structure of MATLAB timeseries objects.

The `Simulink.SimulationData.Dataset` class provides two methods for accessing signal logging data.

Name	Description
<code>get</code> You can also use the <code>getElement</code> method, which shares the same syntax and behavior as the <code>get</code> method.	Get element or collection of elements from the dataset, based on index, name, or block path.
<code>numElements</code>	Get number of elements in the dataset.

The `ex_bus_logging` model illustrates how to access signal logging data.

For additional examples 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. To access the signal logging data for an array of buses, navigate the bus

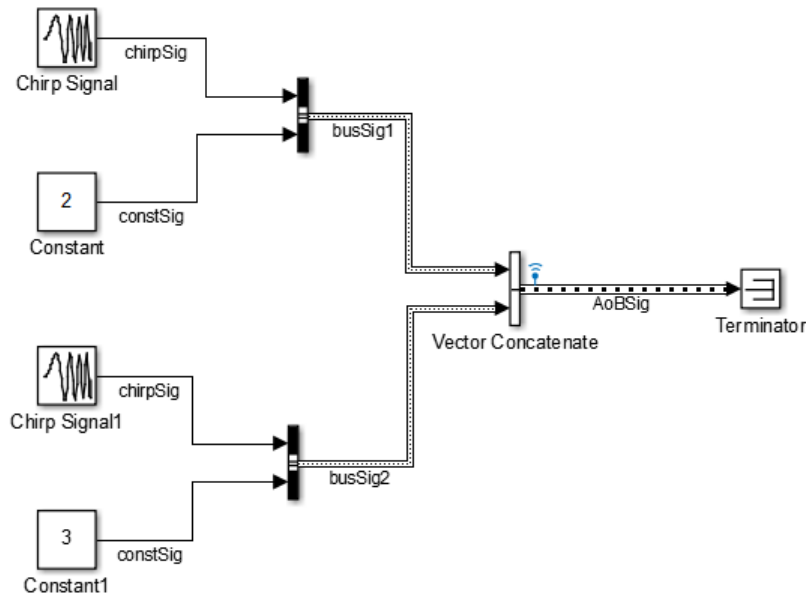
hierarchy and specify the index of the bus element whose data you want to view.

The general approach is to use a `Simulink.SimulationData.Dataset.get` (or `getElement`) method with the logged data (by default, the `logouts` variable). Use the command (using an index into the array) to navigate from the array of buses down to a bus in the array of buses. Use the command to return data for a specific signal.

For an example that shows how to log array of buses data, see `sldemo_md1ref_bus`. Below are additional examples of accessing array of buses signal logging data.

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`.

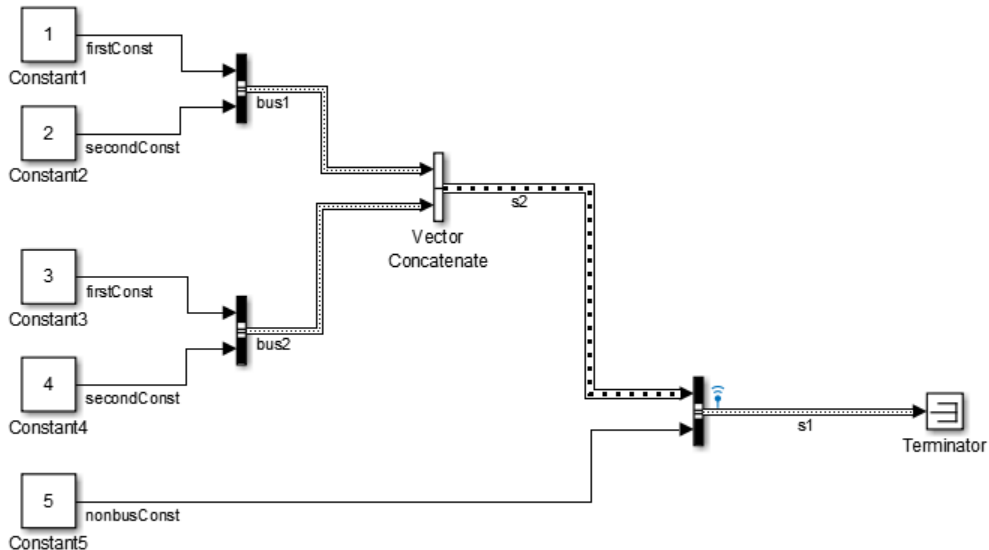
```
logout.get('AoBSig').Values(2).chirpSig.Data
```

```
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`.

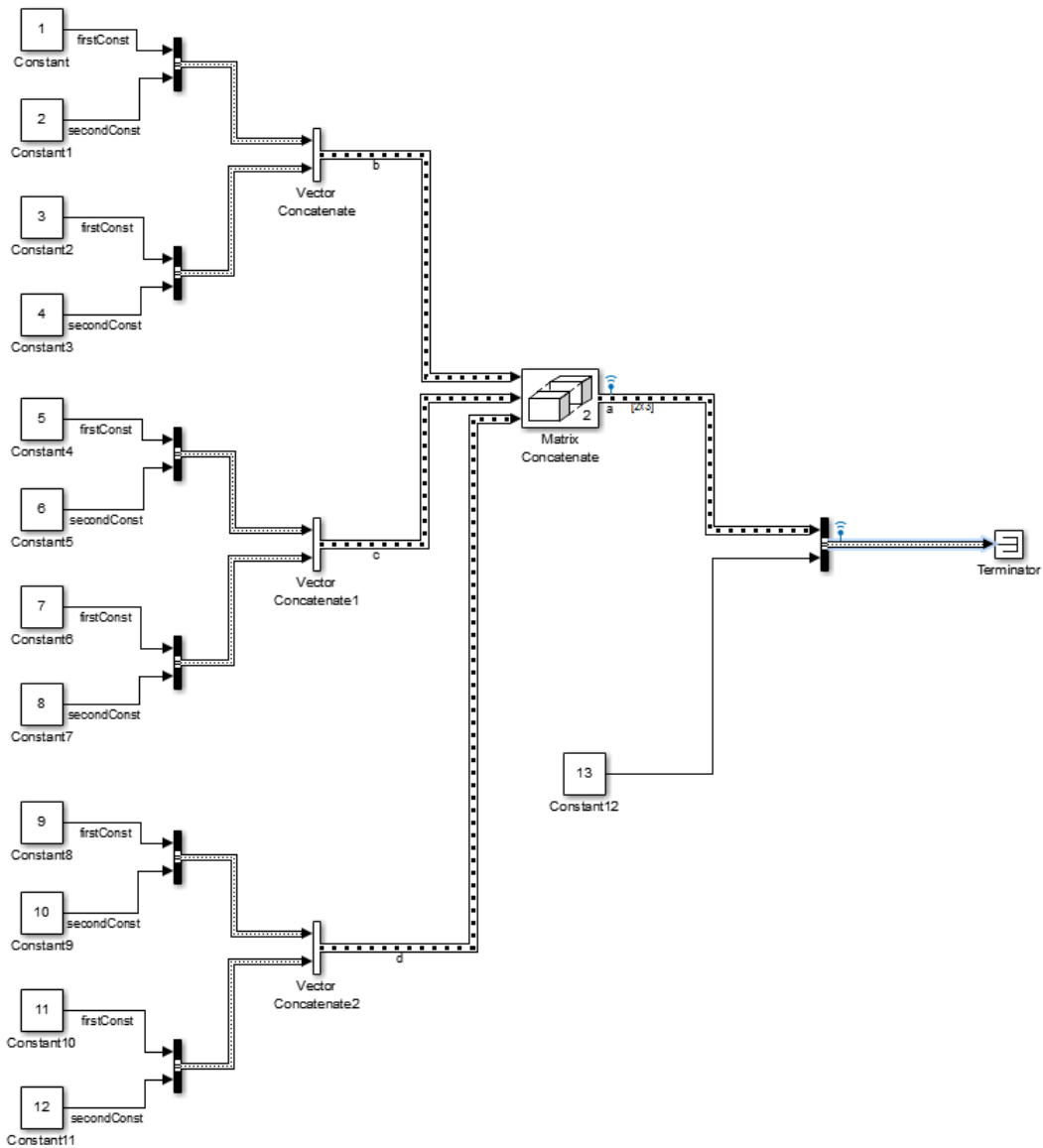
```
logout.get('s1').Values.s2(2).firstConst.Data
```

```
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.

```
logout.get('topBus').Values.a(2,2).firstConst.Data  
  
ans=  
  
7  
7  
7  
7  
7  
7  
7  
7  
7  
7  
7  
7
```

Accessing Data for Signals with a Duplicate Name

For a model with signals with multiple signals that have the same signal name, the signal logging data includes a `Simulink.SimulationData.Signal` object for each signal that has a duplicate name. Simulink sorts the data for those signals based on the model hierarchy, in top-down order.

To access a specific signal that has a duplicate name, use *one* of these approaches:

- Visually inspect the `Simulink.SimulationData.Signal` objects to find the data for the specific signal.
- Use the `Simulink.SimulationData.Dataset.getElement` method, specifying the blockpath for the source block of the signal.
- Create a script to iterate through the signals with a duplicate signal name, using the `Simulink.SimulationData.Dataset.getElement` method with an index argument.
- Use the following approach, if 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.

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'))
```

Programmatically Access Logged Signal Data Saved in ModelDataLogs Format

Note The ModelDataLogs signal logging format is supported for backward compatibility. For new models, use the Dataset format.

When you use the ModelDataLogs signal logging format, Simulink saves the logging data in a `Simulink.ModelDataLogs` object. For information on extracting signal data from that object, see `Simulink.ModelDataLogs`. The `Simulink.ModelDataLogs` object contains signal data objects to capture signal logging information for specific model elements.

Model Element	Signal Data Object
Top-level or referenced model	<code>Simulink.ModelDataLogs</code>
Subsystem in a model	<code>Simulink.SubsysDataLogs</code>
Scope block in a model	<code>Simulink.ScopeDataLogs</code>
Signal other than a bus or Mux signal	<code>Simulink.Timeseries</code>
Bus signal or Mux signal	<code>Simulink.TsArray</code>

Virtual Signals

A *virtual signal* is a signal that graphically represents other signals or parts of other signals. Virtual signals have no functional or mathematical significance. For more information, see “Virtual Signals” on page 50-11.

The nonvirtual components of a virtual signal are called *regions*. A virtual signal can contain the same region more than once. For example, if the same nonvirtual signal is connected to two input ports of a block, the block outputs a virtual signal that has two regions.

When you use the `ModelDataLogs` signal logging format, the log of a virtual signal that contains duplicate regions includes all of the regions, even though the data in each is the same. Logged virtual signal regions appear in the log in a `Simulink.TsputArray` object. The log gives the duplicate regions unique names, using the syntax: `<signal_name>_reg<#counter>`.

Bus Signals

You can log bus signals. When you use `ModelDataLogs` signal logging format, Simulink stores each logged bus signal data in a separate `Simulink.TsArray` object.

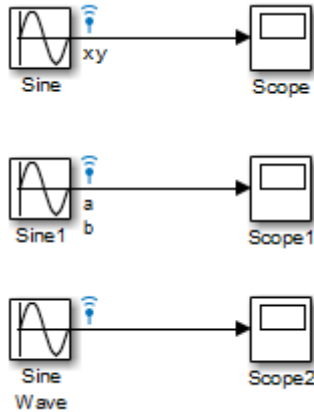
The hierarchy of a bus signal is preserved in the logged signal data. The logged name of a signal in a virtual bus derives from the name of the source signal. The logged name of a signal in a nonvirtual bus derives from the applicable bus object, and can differ from the name of the source signal. See “Composite Signals” for information about those capabilities.

Handling Spaces and Newlines in Logged Names

Signal names in data logs can have spaces or newlines in their names when:

- The signal is named and the name includes a space or newline character.
- The signal is unnamed and originates in a block whose name includes a space or newline character.
- The signal exists in a subsystem or referenced model, and the name of the subsystem, Model block, or of any superior block, includes a space or newline character.

The following three examples show a signal whose name contains a space, a signal whose name contains a newline, and an unnamed signal that originates in a block whose name contains a newline:



The following example shows how to handle spaces or new lines in logged names, if a model uses:

- ModelDataLogs for the signal logging format
- The default of logstout for the signal logging data

logstout =

```
Simulink.ModelDataLogs (model_name):
  Name                Elements  Simulink Class
  ('x y')              1        Timeseries
  ('a
b')                   1        Timeseries
  ('SL_Sine
Wave1')              1        Timeseries
```

You cannot access any of the `Simulink.Timeseries` objects in this log using TAB name completion or by typing the name to MATLAB. This syntax is

not recognized because the space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```
>> logout.x y

??? logout.x y
      |
Error: Unexpected MATLAB expression.
```

To reference a `Simulink.Timeseries` object whose name contains a space, enclose the element containing the space in single quotes:

```
>> logout('x y')

      Name: 'x y'
      BlockPath: 'model_name/Sine'
      PortIndex: 1
      SignalName: 'x y'
      ParentName: 'x y'
      TimeInfo: [1x1 Simulink.TimeInfo]
      Time: [51x1 double]
      Data: [51x1 double]
```

To reference a `Simulink.Timeseries` object whose name contains a newline, concatenate to construct the element containing the newline:

```
>> cr=sprintf('\n')
>> logout(['a' cr 'b'])
```

The same techniques work when a space or newline in a data log derives from the name of:

- An unnamed logged signal's originating block
- A subsystem or Model block that contains any logged signal
- Any block that is superior to such a block in the model hierarchy

This code can reference logged data for the signal:

```
>> logout(['SL_Sine' cr 'Wave1'])
```


For names with multiple spaces, newlines, or both, repeat and combine the two techniques as needed to specify the intended name to MATLAB.

Note You cannot use these techniques for TAB name completion.

Techniques for Importing Signal Data

In this section...
“Signal Data Import Techniques Summary” on page 48-72
“Comparison of Techniques” on page 48-73
“Time and Signal Values for Imported Data” on page 48-75

Signal Data Import Techniques Summary

Simulink provides several techniques for importing signal data into a model.

Signal Data Import Technique	Description
Root-level Inport, Trigger, Enable, or Function-Call Subsystem block	<p>Supplies external inputs from the MATLAB (base), model, or mask workspace. Specify the external inputs in the Configuration Parameters > Data Import/Export > Input parameter or as a <code>sim</code> command argument.</p> <p>To import and map signal and bus data to root-level inports at the same time, you can use the Inport Mapping tool. For details, see “Import and Map Root-Level Inport Data” on page 48-92. Use this method if you have many signals to import and map.</p> <p>For an example of a more manual way to import signal data and map it to root-level inport, see “Import Data to Root-Level Input Ports” on page 48-88. Consider this method if you have few signals to import and map.</p>
From File block	Reads data from a MAT-file and outputs the data as a signal.

Signal Data Import Technique	Description
From Workspace block	Reads data from the MATLAB (base), model, or mask workspace and outputs the data as a signal. For an example, see “Use From Workspace Block to Import an Input Test Case” on page 48-83.
Signal Builder block	Provides a graphical interface for creating and generating interchangeable groups of signals. For examples, see “Use Signal Builder Block to Import an Input Test Case” on page 48-84 and “Import Signal Data”.

Each of these techniques:

- Uses blocks to represent the signal data import sources visually
- Supports data interpolation and extrapolation, which support the use of incomplete sets of signal data
- Supports zero-crossing detection

Comparison of Techniques

Each technique is well-suited to meet one or more of the following modeling considerations.

Modeling Consideration	Suggested Technique
Purpose of importing signal data	
To perform local, temporary testing by importing a small set of signal data	<ul style="list-style-type: none"> • From File block • From Workspace block • Signal Builder block • Root-level Inport, Trigger, Enable, or Function-Call Subsystem block
To test a model to be used as a referenced model	Root-level Inport, Trigger, Enable, or Function-Call Subsystem block

Modeling Consideration	Suggested Technique
To verify a model using multiple test cases	<p>Signal Builder block, using signal groups.</p> <p>If the Simulink Verification and Validation software is installed, in signal groups you can use the Verification Manager to enable or disable individual Model Verification blocks. For details, see “Model Verification Blocks and the Verification Manager”.</p>
To perform local, temporary testing by importing a small set of signal data	<p>Depending on the modeling goal, as discussed in “Time and Signal Values for Imported Data” on page 48-75, choose from the following techniques:</p> <ul style="list-style-type: none"> • From File block • From Workspace block • Signal Builder block • Root-level Inport, Trigger, Enable, or Function-Call Subsystem block
Signal data	
Very large dataset	From File block, which incrementally loads the data
Data exported using a To File block	From File block
Data exported using a To Workspace block	From Workspace block
Excel spreadsheet	Signal Builder block, which can import Excel spreadsheet data directly into Simulink
Variable-size signals	From Workspace block
Array of buses signals	root Inport block

Modeling Consideration	Suggested Technique
Data storage	
In a block	Signal Builder block
In the base or model workspace	<ul style="list-style-type: none"> • From Workspace block • Root-level Inport, Trigger, Enable, or Function-Call Subsystem block
In a MAT-file separate from the model file	From File block

Time and Signal Values for Imported Data

Depending on your model development or testing goal, the approach for specifying time and signal values can vary. Each of these guidelines applies to one or more of the signal data import techniques (From File, From Workspace, Signal Builder, and root-level Inport or Trigger blocks).

Modeling Goal	Time and Signal Values	Additional Notes
Import data representing a continuous plant	<p>Specify a time vector and signal values extracted a continuous plan (for example, data that you acquire experimentally or from the results of a previous simulation).</p> <p>Use any of the data formats listed in “Input Data” on page 48-89. Here are recommended formats for the following imported data sources:</p> <ul style="list-style-type: none"> • Another simulation — Dataset 	<p>In the Inport block parameters dialog box, select Interpolate data.</p> <p>To ensure that the Simulink variable time solver executes at the times that you specify in the imported data, set the Configuration Parameters > Data Import/Export > Output options parameter to Produce additional output.</p>

Modeling Goal	Time and Signal Values	Additional Notes
	<ul style="list-style-type: none"> • An equation — MATLAB time expression • Experimental data — MATLAB timeseries, data array, or structure <p>For details, see “Specify Time Data” on page 48-139. For an example of using a structure to specify the time and signal values, see “Import Data to Model a Continuous Plant” on page 48-78.</p>	
Test a discrete algorithm	<p>Use a structure, with an empty time vector.</p> <p>Specify signal values, but no time vector. (One signal value is read at each time step, using the sample time of the source block.)</p> <p>For an example of using a structure to specify the signal values, see “Import Data to Test a Discrete Algorithm” on page 48-81.</p>	In the Inport block parameters dialog box, clear Interpolate data .
Create an input test case (An input test case is typically composed of steps and	<p>Specify a time vector and signal values, but specify only the time steps at points where the shape of the output jumps.</p> <p>For details about specifying a time vector, see “Specify Time Data” on page 48-139.</p>	<p>Select the Interpolate data parameter.</p> <p>For the input, consider using a From Workspace, From File, or Signal Builder block, which trigger</p>

Modeling Goal	Time and Signal Values	Additional Notes
ramps, with discontinuities.)	Use any of the input data formats described in “Input Data” on page 48-89, except for MATLAB time expressions.	a zero-crossing at the discontinuities. For examples of using a From Workspace block and a Signal Builder block, see “Import Data for an Input Test Case” on page 48-82.

Import Data to Model a Continuous Plant

In this section...

“Share Simulation Data Across Models” on page 48-78

“Example of Importing Data to Model a Continuous Plant” on page 48-78

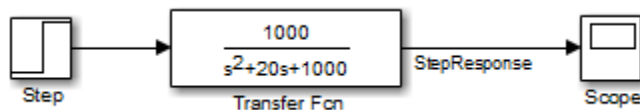
Share Simulation Data Across Models

When reusing data from a variable step size simulation for simulation in another model, the second simulation must read the data at the same time steps as the first simulation.


The following example illustrates how to reuse signal logging data from the simulation of one model in the simulation of a second model. For more information, see “Import Signal Logging Data” on page 48-86.

Example of Importing Data to Model a Continuous Plant

- 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 reuse. In the Simulink Editor, select that signal, click the **Record** button arrow  and click **Log/Unlog Selected Signals**.

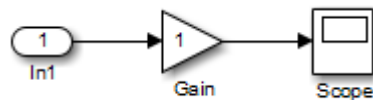
Note To enable signal logging, select the **Configuration Parameters > Data Import/Export > Signal logging** parameter. This model has **Signal logging** enabled, and has the **Signal logging format** parameter set to Dataset.

3 Simulate the model.

Simulating the model saves a variable-step signal to the workspace, using the `logout` variable. The signal logging output is a `Simulink.SimulationData.Dataset` object.

Use the `Simulink.SimulationData.Dataset.getElement` method to access the logged data. The logged data for individual signals is stored in `Simulink.SimulationData.Signal` objects. For this model, there is one logged signal: `StepResponse`.

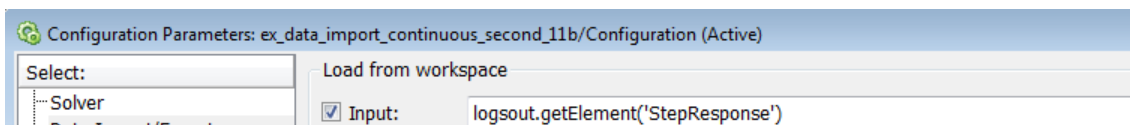
4 Open a second model, named `ex_data_import_continuous_second`.



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 import the logged data. 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, as shown below:



- 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:

```
logout.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 “Minor Time Steps” on page 3-23.

Import Data to Test a Discrete Algorithm

In this section...

“Specify a Signal-Only Structure” on page 48-81

“Example of Importing Data to Test a Discrete Algorithm” on page 48-81

Specify a Signal-Only Structure

To import data for a discrete signal, specify a signal-only structure as the input value. Do not specify a time vector.

Example of Importing Data to Test a Discrete Algorithm

- 1 Set the sample time for the Inport, Trigger, or From Workspace block.
- 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 for the second time step is read from the second element of the value array, and so on.

For details about how to specify the signal value and dimension data, see “Specify Signal Data” on page 48-138.

- 3 In the block parameters dialog box for the block that imports the data, clear the **Interpolate data** parameter.
- 4 If you are using a From Workspace block to import data, set the **Form output after final data value by** parameter to a value other than Extrapolation.

Import Data for an Input Test Case

In this section...
“Guidelines for Importing a Test Case” on page 48-82
“Example of Test Case Data” on page 48-82
“Use From Workspace Block to Import an Input Test Case” on page 48-83
“Use Signal Builder Block to Import an Input Test Case” on page 48-84

Guidelines for Importing a Test Case

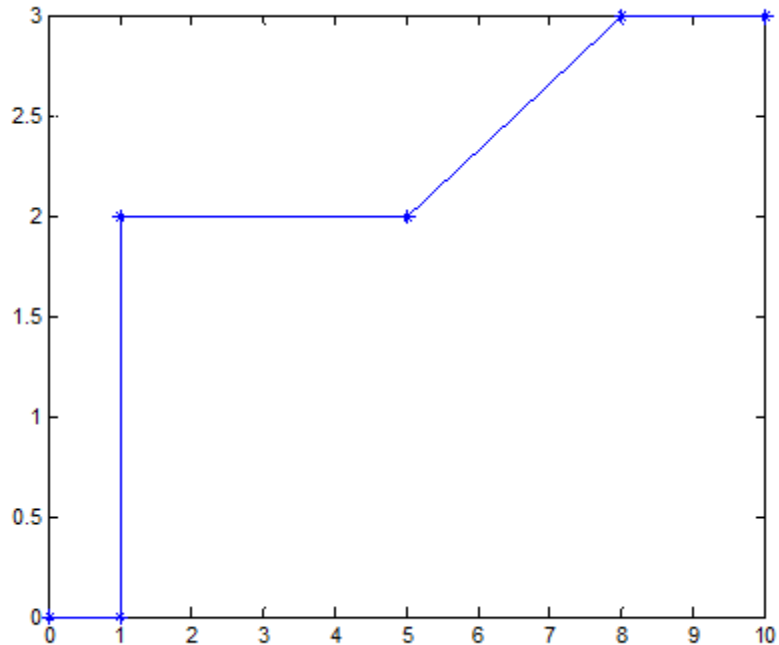
Typically when importing a test case in Simulink, you want to:

- 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 Builder block, all of which support zero-crossing detection.

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 to Import an Input Test Case

- 1 Open the model `ex_data_import_test_case_from_workspace`.



- 2 Enable zero-crossing detection. In the From Workspace block dialog, select **Enable zero-crossing detection**.

- 3 Create a signal structure for the test case. At each discontinuity, enter a duplicate entry in the time vector. As described in the From Workspace block documentation, this generates a zero-crossing and forces the variable-step solver to take a time step at this exact time.

Define the var structure representing the test case:

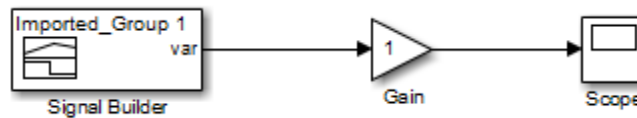
```
var.time = [0 1 1 5 5 8 8 10];
var.signals.values = [0 0 2 2 2 3 3 3]';
var.signals.dimensions = 1;
```

- 4 To import the test case structure, in the From Workspace block dialog, in the **Data** parameter, specify var.
- 5 Simulate the model. The Scope block reflects the test case data.

Use Signal Builder Block to Import an Input Test Case

As an alternative to using a From Workspace block, you can use a Signal Builder block to either create a signal interactively or to import a signal from a MAT-file.

- 1 Open the ex_data_import_signal_builder model.



- 2 Create a structure and save it in a MAT-file:

```
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;
var.signals.label = 'var';
save var.mat var
```

- 3 Double-click the Signal Builder block to open its dialog box.
- 4 Select **File > Import From File** menu item, and select the var.mat file.

5 In the **Select** parameter, select **Replace existing dataset**. In the **Data to Import** section, select the **Select All** check box. Confirm the selection and click **OK**.

The Signal Builder block display reflects the test case data.

For a detailed example that shows how to use a Signal Builder block as the input source for your model and to import your own signal data to the model, see “Import Signal Data”.

Import Signal Logging Data

You can log signal data from a simulation, and then import that data into a model. For more information about signal logging, see “Export Signal Data Using Signal Logging” on page 48-21.

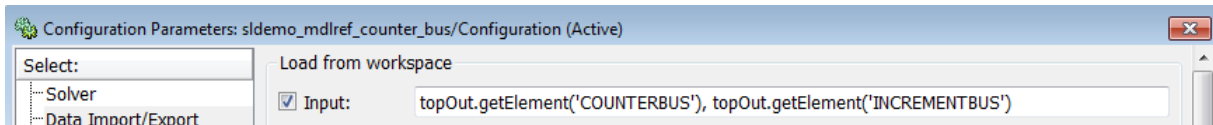
The imported signal logging data provides the input for simulating the model. You can use imported signal logging data to perform standalone simulation of a referenced model.

- 1 Set the **Configuration Parameters > Data Import/Export > Signal logging format** parameter to Dataset.
- 2 Use the default signal logging output variable, `logout`, or specify a variable using the **Configuration Parameters > Data Import/Export > Signal logging** edit box.
- 3 Simulate the parent model.

The signal logging output is a `Simulink.SimulationData.Dataset` object.

- 4 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.
- 5 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:



- 6 Simulate the referenced model.

For an example of loading logged signal data into a model:

- 1** Open the `sldemo_md1ref_bus` model.
- 2** In the right corner of the model, double-click the left question mark block.
- 3** See the “Logging Model Reference Signals” and “Loading Data” sections.

To import signal logging data for array of buses signals, see “Import Array of Buses Data” on page 48-126.

Import Data to Root-Level Input Ports

In this section...

“Root-Level Input Ports” on page 48-88

“Enable Data Import” on page 48-88

“Input Data” on page 48-89

“Import Bus Data” on page 48-91

Root-Level Input Ports

You can import data from a workspace and apply it to a root-level:

- Enable block
- Inport block
- Trigger block that has an edge-based (rising, falling, or either) trigger type

Note You cannot use input ports to import buses in Rapid Accelerator, RSim, or External modes.

You can also import data from a workspace and apply it to a From Workspace block. For details, see the From Workspace documentation and “Import Data for an Input Test Case” on page 48-82.

If you want to import many signals to root-level input ports, see “Import and Map Root-Level Inport Data” on page 48-92.

Enable Data Import

To enable data import:

- 1** Select the **Configuration Parameters > Data Import/Export > Input** parameter.
- 2** Enter an external input specification in the adjacent edit box and click **Apply**.

For details, see “Input Data” on page 48-89.

Input Data

Use the **Configuration Parameters > Data Import/Export > Input** parameter to import data from a workspace and apply it to the root-level input ports of a model during a simulation run.

Simulink linearly interpolates or extrapolates input values as necessary if you select the **Interpolate data** option for the corresponding Inport, Enable, or Trigger block.

Note The use of the **Input** box is independent of the setting of the **Format** list on the **Data Import/Export** pane.

Simulink resolves symbols used in the external input specification as described in “Symbol Resolution” on page 4-83. See the documentation of the `sim` command for some data import capabilities that are available only for programmatic simulation.

Forms of Input Data

The input data can take any of the following forms:

- MATLAB timeseries — For details, see the following sections:
 - “Import MATLAB timeseries Data” on page 48-121
 - “Import Structures of timeseries Objects for Buses” on page 48-123
- Array — See “Import Data Arrays” on page 48-134.
- `Simulink.SimulationData.Signal` — For details, see “Import Signal Logging Data” on page 48-86
- Structure — See “Import Data Structures” on page 48-137.
- A 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 — See “Import MATLAB Time Expression Data” on page 48-136.
- `Simulink.Timeseries` and `Simulink.TsArray` — See “Import Simulink.Timeseries and Simulink.TsArray Data” on page 48-133.

Note The `Simulink.Timeseries` and `Simulink.TsArray` formats are supported for compatibility with earlier releases. In new models, use one of the other supported formats.

Input Expressions

In the **Input** box, specify the signal input using either of these approaches:

- Create data at runtime 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 either:
 - A 2D array of input values versus time for all input ports, in the form $UT = [T, U1, \dots, Un]$, where $T = [t1, \dots, tm]'$
 - A comma-separated list of variables or MATLAB expressions. For details, see “Comma-Separated Lists for the Input Parameter” on page 48-90.

Comma-Separated Lists for the Input Parameter. Each variable or expression corresponds to a specific input port. Each variable or expression in the list should evaluate to the appropriate object that corresponds to one of the root-level input ports of the model, with the first item corresponding to the first root-level input port, the second to the second root-level input port, and so on.

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, and the trigger port as the last item.

To simplify the specification of external data to input, you can load data for a subset of root-level Inport ports, without 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 50-52.

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
```

Import Bus Data

To import bus data to root input ports, use a structure of MATLAB `timeseries` objects. For details, see “Import Structures of `timeseries` Objects for Buses” on page 48-123.

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
```

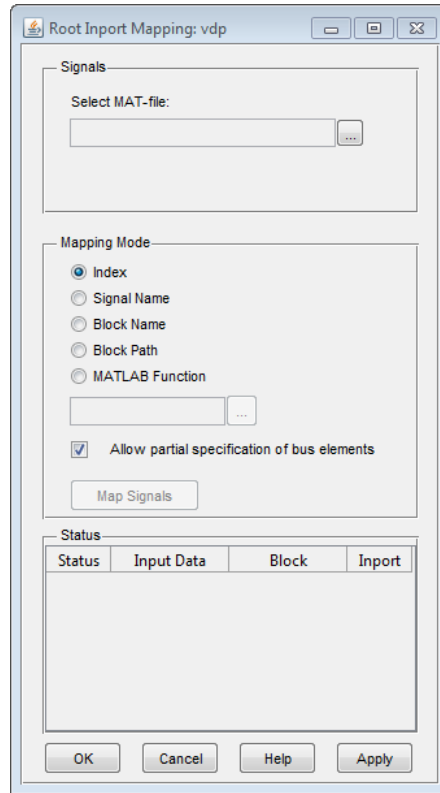
You can initialize bus signals, including using partial specification of initialization data. For details, see “Specify Initial Conditions for Bus Signals” on page 51-69.

For details about importing array of bus data to a root Inport block, see “Import Array of Buses Data” on page 48-126.

Import and Map Root-Level Inport Data

In this section...
“Importing and Mapping Workflow” on page 48-94
“MAT-File for Import and Mapping” on page 48-95
“Import and Map Signal Data” on page 48-97
“Import and Map Bus Data” on page 48-98
“Understanding Mapping Results” on page 48-102
“Convert Test Harness Model to Harness-Free Mode” on page 48-107
“Converting Test Harness Model to Harness-Free Model” on page 48-107
“Alternative Workflows to Load Data” on page 48-116
“Create Custom Mapping File Function” on page 48-117

To import and map signal and bus data to root-level inports at the same time, use the Root Inport Mapping dialog box. (You can open or use the Root Inport Mapping dialog box when the model is stopped. You cannot use it when the model is running or paused.) For alternative ways to import data, see “Techniques for Importing Signal Data” on page 48-72



Note If data already exists in the MATLAB workspace, you cannot map it to root-level inports using this dialog box. Use this dialog box to import the data, then map it, in one operation.

Map data to root-level inports using one of these methods:

- **Index** — Assigns sequential index numbers, starting at 1, to the data in the MAT-file, and maps this data to the corresponding inport.
 - If there is more data than inports, the remaining data are mapped to enable and then trigger inports, in that order.

- If the data is not in the form of a dataset, the data are processed in the order in which they are passed or parsed from the data file. This process is the order in which they appear in the file.
- **Signal Name** — Assign data to ports according to the name of the signal on the port. If the signal name of a data element matches the name of a signal at a port, the data is mapped to the corresponding port.
- **Block Name** — Assign data to ports according to the name of the root-inport block. If the block name of a data element matches the name of a root-inport block, the data is mapped to the corresponding port.
- **Block Path** — Assign data to ports according to the block path of the root-inport block. If the block path of a data element matches the block path of a root-inport block, the data is mapped to the corresponding port.
- **MATLAB Function** — Apply mappings according to the definitions in a custom file.

Importing and Mapping Workflow

- 1 Identify the signals you want to import and map to the model root-level inports.

Tip Consider using a naming convention for signals and buses such that this grouping of data (signal group) is interchangeable. This convention allows you to quickly switch the groups of input data into and out of a model. For example, you can have two MAT-files whose data has different values but the same variable names.

- 2 Create a MAT-file that contains the signal or bus data that you want to import into your model and map to the root-level inports of your model (see “MAT-File for Import and Mapping” on page 48-95).
- 3 Determine how you want to map data (for example, by signal index or signal name), and perform the mapping.

To map data using one of the standard map formats, see:

- “Import and Map Signal Data” on page 48-97

- “Import and Map Bus Data” on page 48-98

To create a custom mapping file function to map data to root-level inports, see “Create Custom Mapping File Function” on page 48-117.

MAT-File for Import and Mapping

The Root Inport Mapping dialog box requires a MAT-file that contains data with one of these data types or formats.

Data Formats	Index Mapping	Signal Name Mapping	Block Name Mapping	Block Path Mapping	Custom Mapping
Simulink.SimulationData.-Dataset	✓	✓	✓	✓	✓
timeseries (MATLAB and Simulink)	✓	✓	✓	✓ (only for Simulink timeseries)	✓
Simulink.SimulationData.-Signal	✓	✓	✓	✓	✓
Structure with Time, Structure without Time	✓				
TsArray	✓			✓	✓
Data Array	✓				
Time Expression	✓				
Array of buses	✓	✓	✓		✓
Asynchronous function-call signals	✓	✓	✓		✓

For example, a signal data MAT-file with three signals (*signal1*, *signal2*, and *signal3*) with block path specified might have workspace variables as shown

in this example. This file has signals that have signal names, block names, block paths, and index values. This means that you can map the data using Signal Name, Block Name, Block Path, or Index mapping mode. If there are corresponding root-level inports with corresponding indices, signal names, or block paths, the tool maps the signal to those ports.

```
signal1 =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
    Name: 'signalGain5'
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    PortType: 'outport'
    PortIndex: 1
    Values: [1x1 timeseries]

  Methods, Superclasses
>> signal2

signal2 =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
    Name: 'signalGain10'
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    PortType: 'outport'
    PortIndex: 1
    Values: [1x1 timeseries]

  Methods, Superclasses
>> signal3

signal3 =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData
```

Properties:

```
Name: 'signalGain15'  
BlockPath: [1x1 Simulink.SimulationData.BlockPath]  
PortType: 'output'  
PortIndex: 1  
Values: [1x1 timeseries]
```

Methods, Superclasses**Notes:**

- Only one `Simulink.SimulationData.Dataset` data set per MAT-file can load. If the MAT-file has multiple data sets of this type, the first data set listed alphabetically loads.
- The number of signals with the formats structures, with and without time, data arrays, and time expressions must be the same as the number of model inports, enable ports, and trigger ports.
- If you load a MAT-file that contains fixed-point data but you do not have a Fixed-Point Designer license, timeseries data is loaded and mapped, but model simulation will be inconsistent.
- Data arrays are matrices. If you have a data array where the *number of columns*-1 is equal to the number of root-level inports (including trigger and enable ports), the Root Inport Mapping dialog box tries to map this data for the entire model. In this case, you can choose only the **Index** mapping method. If the *number of columns*-1 does not equal the number of root-level inports (including trigger and enable ports), the Root Inport Mapping dialog box tries to map the data array to a single inport. In this case, you can choose any of the mapping methods.


Import and Map Signal Data

Import Signal Data

- 1 Before you start, create a MAT-file that contains the signal data you want to import and map.

- 2** In the Simulink model you want to import the signal data to, open the Configuration Parameters dialog box.
- 3** In the Data Import/Export pane, in the **Load from workspace** section, click **Edit Input**.

The Root Inport Mapping dialog box appears.

- 4** In the Signals section, click  and select the MAT-file that contains the signal data, and then click **Open**.

To map the data, see “Map Data” on page 48-100.

Import and Map Bus Data

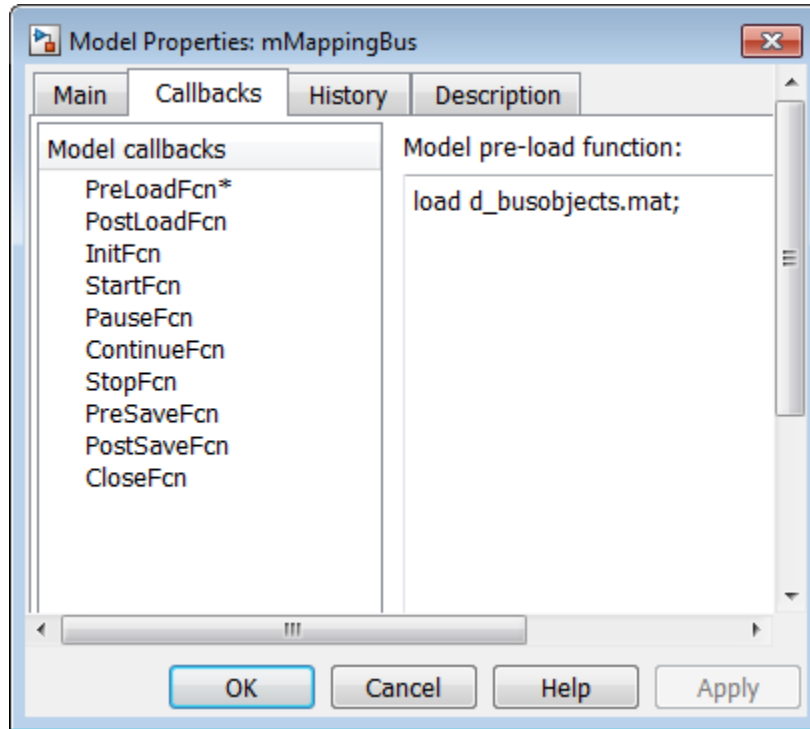
You can import and map bus data as well as signal data. This topic assumes that you are familiar with bus objects and how to create them. For more information on buses, see “Bus Objects” on page 51-24.

Set up to Import Bus Data

- 1** In the MATLAB workspace, create a bus object for the bus data you want to import and map.
- 2** Save the bus object definition to a MAT-file, for example, `d_myBusObj.mat`.
- 3** Create a different MAT-file that contains the bus data you want to import for the bus object.
- 4** Open a model you want to import the data to.
- 5** In the Simulink Editor, select **File > Model Properties > Model Properties**.

The Model Properties dialog box appears.

- 6** In the Callbacks pane, enter a load function to load the bus object definition and click **OK**. The figure shows an example of loading a bus object from `d_myBusObj.mat`.




Continue to “Import Bus Data” on page 48-99.

Import Bus Data

If you have not yet set up to import bus data, see “Set up to Import Bus Data” on page 48-98.

- 1 In the Simulink model you want to import the signal data to, open the Configuration Parameters dialog box.
- 2 In the Data Import/Export pane, in the **Load from workspace** section, click **Edit Input**.

The Root Inport Mapping dialog box appears.

- 3 In the Signals section, click  and select the MAT-file that contains the bus data, and then click **Open**.

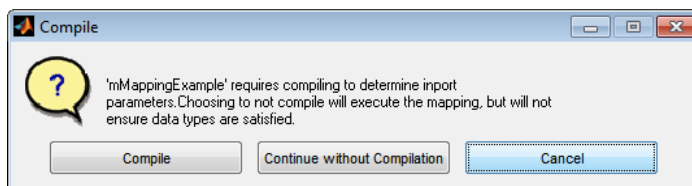
To map the data, see “Map Data” on page 48-100.

Map Data

This topic assumes that you have already imported signal or bus data. If you have not yet done so:

- For signal data, see “Import Signal Data” on page 48-97.
- For bus data, see “Import Bus Data” on page 48-99.

- 1 Select the mapping mode you want to use (for a description of the modes, see “Import and Map Root-Level Inport Data” on page 48-92), and then click **Map Signals**.
- 2 If you are importing bus data, select the **Allow partial specification of bus elements** check box to import bus data that is only partially defined in the MAT-file. If you clear this check box and import partially specified bus data, the partially specified bus data cannot map properly to root-level inports.
- 3 Respond to the prompt to complete the mapping.

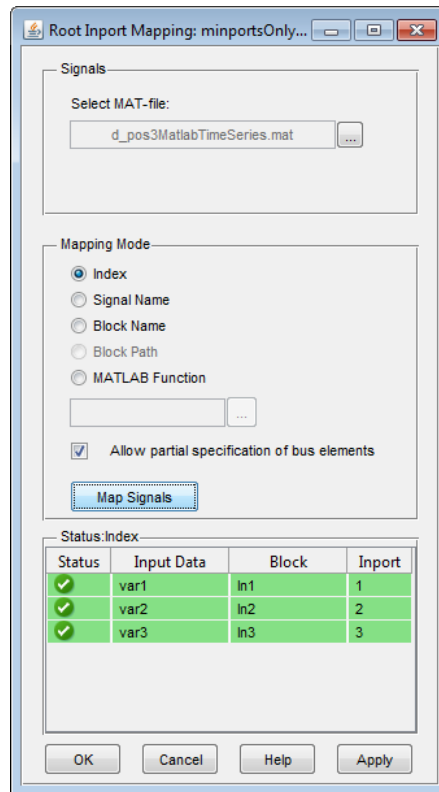


Select one of the following:

- **Compile** — Maps the imported data to the root-level inport and compiles the model to check data types of root-level inports and imported data.
- **Continue without Compilation** — Maps the imported data to the root-level inport but does not compile the model. When you select this option, the software compares the data and inport parameters to the root-level port and displays the results. See “Understanding Mapping

Results” on page 48-102. If the comparison data is inherited from the connected block, you receive a warning.

The results of a signal mapping are displayed in the Status area.



The Status area of the tool lists the input data and the status of the mapping.

Note To understand the mapping results, see “Understanding Mapping Results” on page 48-102.

4 To apply changes to the model, click **Apply**.

5 Save and close the model.



The mapping definition for the input data is saved to the model.


The next time you load input data of the same signal group into the workspace, simulating this model automatically uses the mapping definition.

After you save the mapping definition for a model, you can automate data loading. For more information, see “Alternative Workflows to Load Data” on page 48-116.

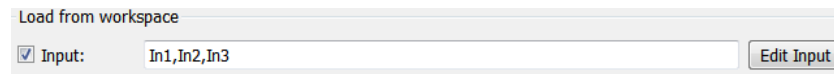
Understanding Mapping Results

When you complete the import and map process, the Root Inport Mapping dialog box displays the results in the status area:

Status	Compile	Continue Without Compile
	The properties of the mapped data and the inport are appropriate for simulation.	The data type, dimension, and signal type properties of the data and inport are compatible.
	N/A	<p>Comparison of data and root-level port data type, dimension, and signal type properties unable to determine if there is a match. If you do not compile before mapping, the tool cannot evaluate whether all the data types match unless you explicitly specify the Inport data types. In other words:</p> <ul style="list-style-type: none"> Inport block parameter Data type cannot be <code>Inherit:auto</code>.

Status	Compile	Continue Without Compile
		<ul style="list-style-type: none"> Inport block parameter Signal type cannot be -1.
	The properties of the mapped data and the inport are not appropriate for simulation.	One or more of the properties data type, dimensions, and signal type of the data and inport are not compatible.

In the Root Inport Mapping dialog box, click **OK** or **Apply** to select the **Input** check box in the **Data Import/Export** pane of the Configuration Parameters dialog box. This action 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** or **Apply**.

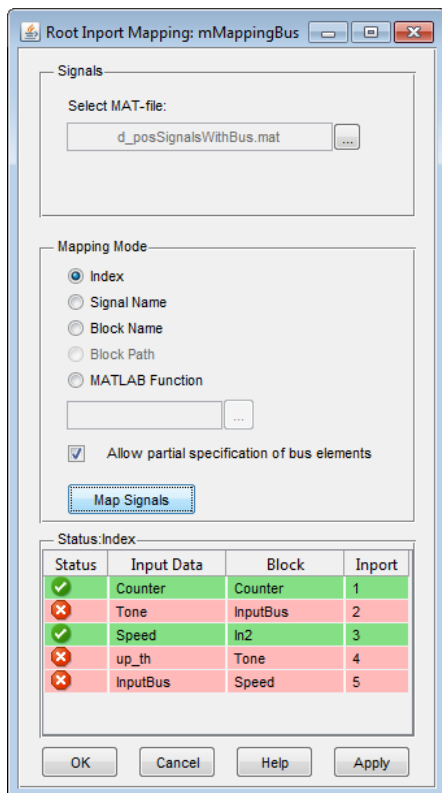


Root-level input ports that have not been mapped display as empty ([]).

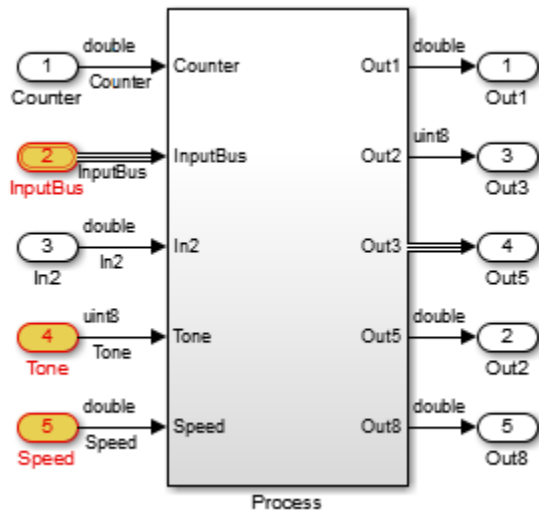
To inspect the imported data, you can:

- 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.

This graphic shows successes and failures:

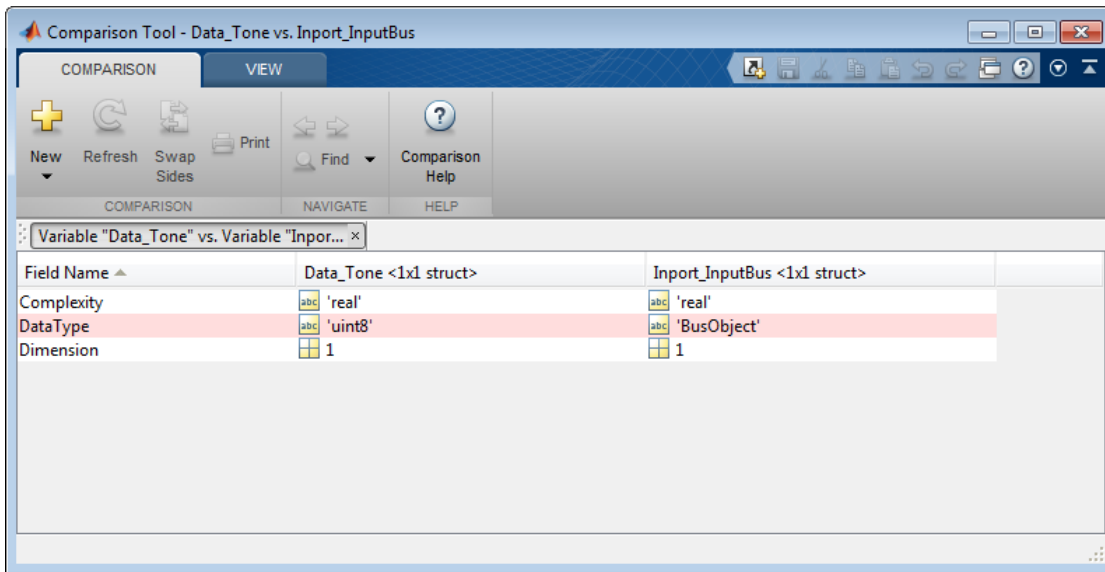


If there are warnings or failures, the software highlights the Inport block associated with the data. Warnings display as yellow Inport blocks outlined in orange, failures display as yellow Inport blocks outlined with bold red, and successes display as normal Inport blocks outlined with blue. Selecting the line item highlights the associated Inport block. For example:





To investigate warnings and failures, click the line item you want to correct in the Status section of the Root Inport Mapping dialog box. The Comparison Tool is displayed with the columns in the left-right order of field name, input data, and root-level inport.

If you click a successful line item, the line turns blue and the Inport block is outlined in blue.



Note When the input is a bus, click the levels of the bus object to drill down to the individual elements in the bus.

Use the Comparison Tool to evaluate your next action. For example, you can edit the data in the MAT-file or edit the root-level inport. When you are done, reimport the data again to map it.

Tip Each time you click a status item, a new Comparison Tool window appears. To dock all Comparison Tool windows in one window, click the Dock button . Click the Undock button  to return to the default .

For more information on the Comparison Tool, see “Comparing Files and Folders”.

Convert Test Harness Model to Harness-Free Mode

- 1 In the MATLAB Command Window, click the question mark.

The Documentation Center window opens.

- 2 Navigate to **Simulink > Examples > Modeling Features > Modeling Concurrency > Converting Test Harness Model to Harness-Free Model**.

Converting Test Harness Model to Harness-Free Model

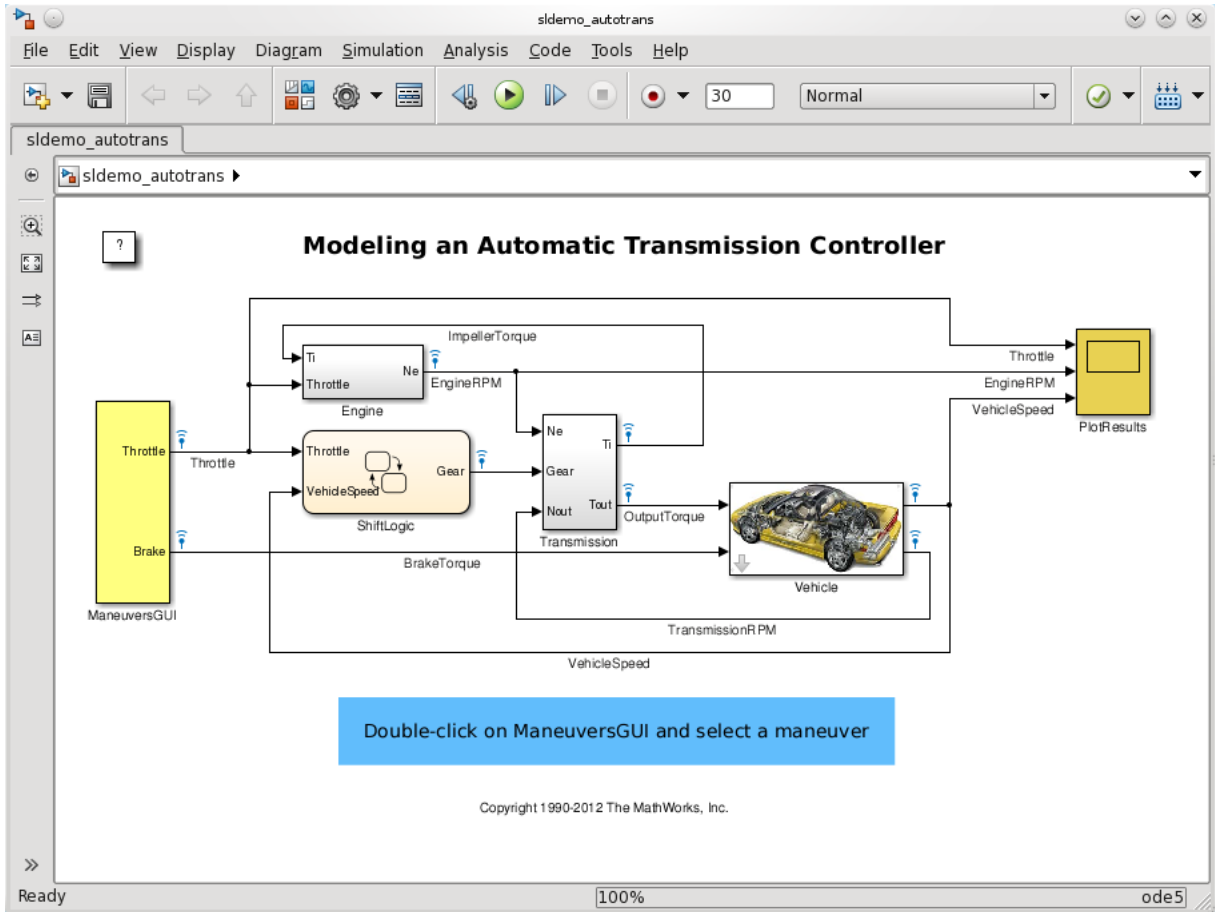
This example shows how to convert a harness model that uses a Signal Builder block as an input to a harness-free model with root inports. The example collects data from the harness model and stores it in MAT-files, for use by the harness-free model. After storing the data, the example removes the Signal 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, capture 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`.



Use MATLAB function `slexAutotransRootInportsSaveActiveGroup` to save the Signal Builder block data of the active group into a MAT-file. At the command line type `slexAutotransRootInportsSaveActiveGroup('slexAutotransRootInport','ManeuversGUI')`. This function creates a MAT-file in the current directory with the same name as the active group. The function `slexAutotransRootInportsSaveActiveGroup` has been provided with this example.

Remove the Signal Builder Block

Remove the Signal Builder block named ManeuversGUI and replace it with two inports.

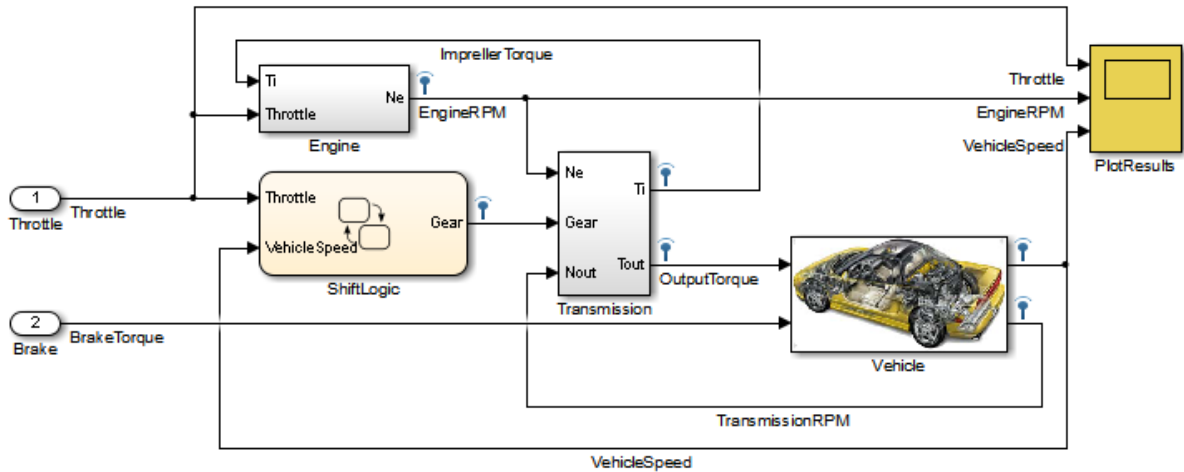
- 1** Delete the Signal Builder block named ManeuversGUI.
- 2** Open the **Simulink Library Browser** and select **Commonly Used Blocks**.
- 3** Drag and drop two **input ports** from the **Library Browser** to the model.
- 4** Connect the input ports to the lines previously connected to the Signal Builder block.
- 5** 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.

?

Modeling an Automatic Transmission Controller with Root Imports



Copyright 2012 - 2013 The MathWorks, Inc.

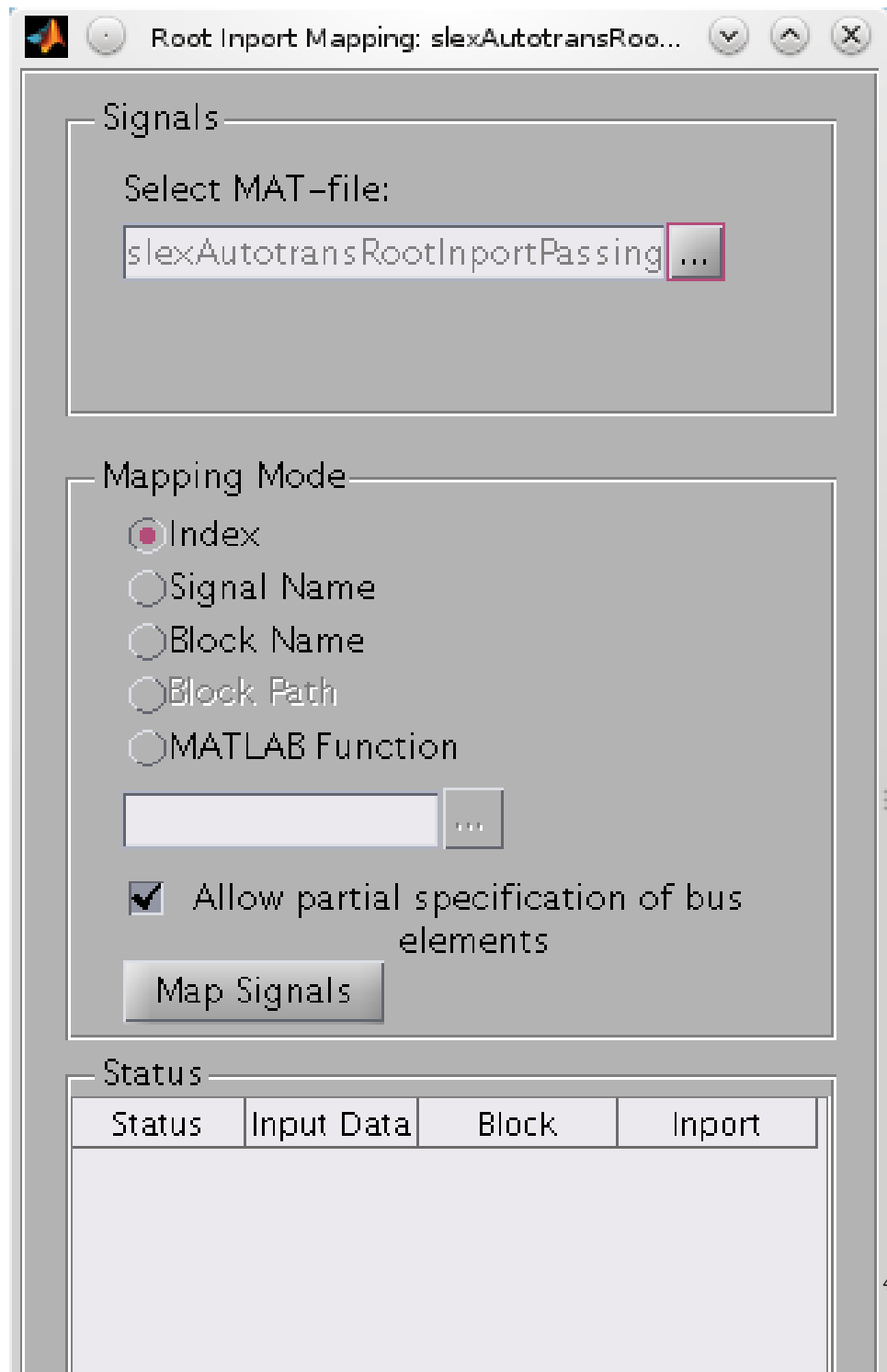
Set Up Harness-Free Inputs

Now that the model is harness-free, set up the inputs that you already saved (See "Save Harness Data to MAT-Files").

From the **Simulation->Model Configuration Parameters->Data Import/Export** pane, click the **Edit Input** button.

Map Signals to Root Inport

The Root Inport Mapping tool opens.



This 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, navigate to the Signals panel and click the **browse(...)** MAT-file button. In the browser, select the MAT-file that you saved earlier.

Select a Mapping Mode

Once 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:

- **Index** - 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.

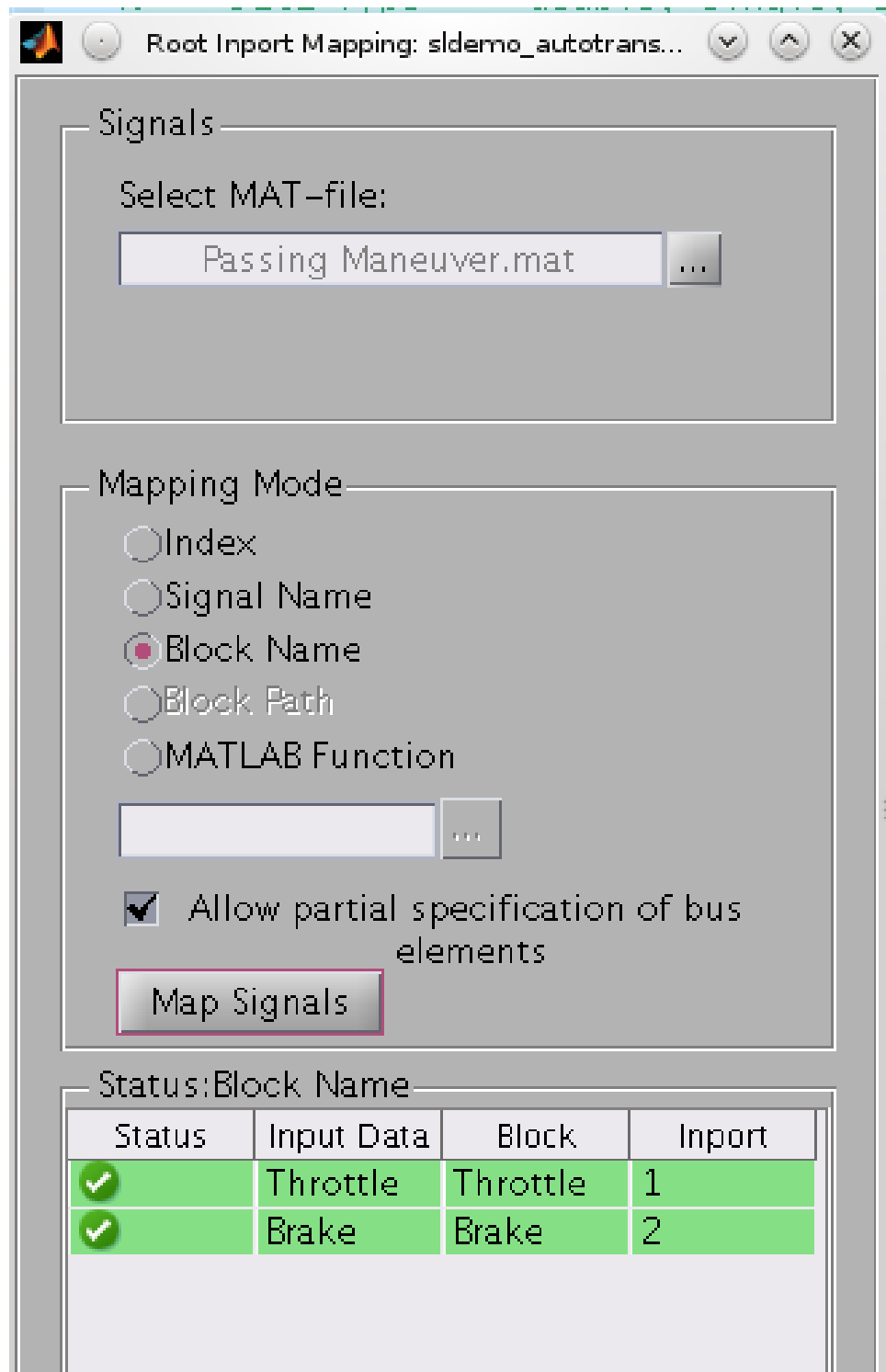
Earlier in this example, you used the `slexAutotransRootInportsSaveActiveGroup` to save input data to variables of the same name as the harness signals Throttle and Brake, and you added input ports with names matching the variables. Given the set of conditions for the input data and the model input ports, the best choice for a mapping criteria is **Block Name**. Using this criteria, Simulink tries 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 **Map Signals** button.

The software prompts you to compile, continue without compiling, or cancel the mapping. Choose to compile the model to verify the mapping.

When compiling the data, Simulink evaluates inports against the following criteria to determine whether or not there is a compatibility issue. The status of this compatibility is reflected by the table colors green, orange, or red. Clicking a cell in the table which has orange or red color will open the Comparison Tool for further inspection.

- **Data Type** - Double, single, enum,
- **Complexity** - Real or complex
- **Dimensions** - Signal dimensions vs port dimensions

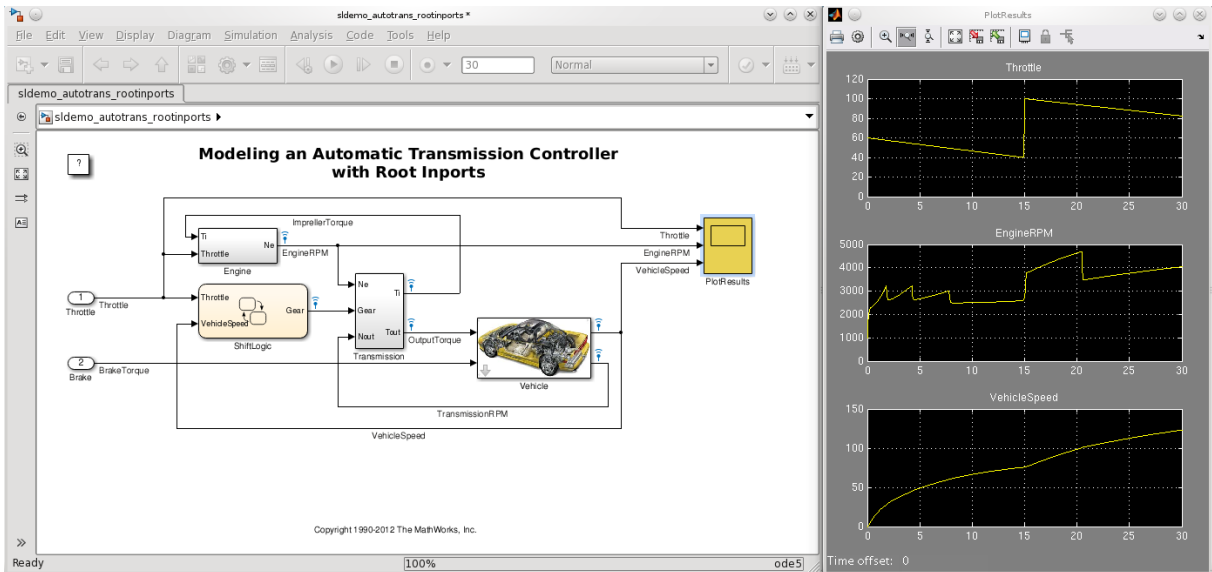


Finalize the Inputs to the Model

After reviewing the results of the mapping compatibility, finalize the inputs to the model by clicking the **Apply** or **OK** button on the Root Inport Mapping tool. These buttons load the data that has been mapped to the input ports from the MAT-file to the base workspace. Simulink sets the **Model Configuration Parameters->Data Import/Export->External Input** edit box with the proper comma separated list of inputs. To apply the changes to the model, in the Configuration Parameters dialog, click **Apply**.

Simulating the Model

With the changes applied you can now simulate the model and view the results. Click the **Play** button on the model. To view the results of the simulation, double-click the Scope Block **PlotResults**.



Alternative Workflows to Load Data

After saving the mapping definition to a model, you can automate data loading and/or 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 like:

```
load('signaldata.mat');  
simout = sim('model_name');
```

To automate testing and load different signal groups, consider using a script.

For example, the following example code creates timeseries data and simulates a model after loading each signal group. It:

- 1 Creates signal groups with variable names *In1*, *In2*, and *In3*, and saves these variables to MAT-files.
- 2 Simulates a model after loading each signal group.

Note The variable names must match the import data variables in the **Data Import/Export > Input** parameter of the Configuration Parameters dialog box.

```
% 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));  
  
    % Save the data  
    save([fileName '_' num2str(k) '.mat' ], 'var1', 'var2', 'var3');  
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 Pane

If you are satisfied with the data and mapping, you can set up your model to automatically load a MAT-file of the same signal group by calling the `load` function in the model `PreLoadFcn` model property node.

1 After saving the MAT-file, in the Simulink editor, select **File > Model Properties > Model Properties**.

2 In the Model Properties window, select the `PreLoadFcn` node.

3 Enter a load function to load the signal data MAT-file. For example,

```
load d_signal_data.mat;
```

4 Click **OK** and save the model.

Create Custom Mapping File Function

Create a custom mapping file function if you do not want to use the mapping modes in the Root Inport Mapping dialog box to map your data to root-level input ports. 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 explicitly map a signal.

The custom mapping file function is also useful if you want to map by block name, but the data contains a signal whose name does not match one of the block names.

See the following files in `matlabroot\help\toolbox\simulink\example` for examples:

- `BlockNameIgnorePrefixMap.m` — Custom mapping file function that ignores the prefix of a signal name when importing
- `BlockNameIgnorePrefixData.mat` — MAT-file of signal data to be imported
- `ex_BlockNameIgnorePrefixExample` — Model file into which you can import and map data


Follow these general steps to create a custom mapping file function:

- 1** Create a MATLAB function with the following input parameters:
 - Model name
 - Signal names specified as a cell array of strings
 - 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:

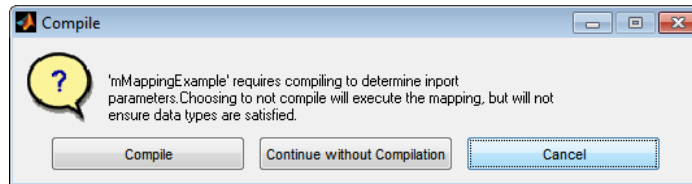
- 1** Open the model you want to import data to (for example, `ex_BlockNameIgnorePrefixExample`).
- 2** Open the Configuration Parameters dialog box for the model and select the Data Import/Export pane.
- 3** In the **Load from workspace** section, click **Edit Input**.

The Root Inport Mapping dialog box appears.

- 4 In the Signals section, click  and select the MAT-file that contains the signal data (for example, BlockNameIgnorePrefixData.mat).
- 5 From the Mapping Mode list, click **MATLAB Function**.
- 6 In the **MATLAB Function** parameter, select the MATLAB function file (for example, BlockNameIgnorePrefixMap.m) using the browser button.

Tip The Root Inport Mapping dialog box parses your custom code. Parsing reorders output alphabetically and verifies that data types are consistent.

- 7 Click **Apply**, and click **Compile** in response to the message:



The software compiles the model and updates the Root Inport Mapping dialog box.

- 8 Click **OK**.

To understand the mapping results, see “Understanding Mapping Results” on page 48-102.

- 9 Save and close the model.

The next time you load input data of the same signal group into the workspace, simulating this model automatically uses the mapping definition.

For another example that shows how to create a custom mapping file function, from the Simulink landing page, click **Examples > Modeling Features > Custom Mapping for External Inputs of a Model**.

After you save the mapping definition for a model, you can automate data loading. For more information, see “Alternative Workflows to Load Data” on page 48-116.

Custom Mapping Modes Similar to Simulink Modes

If you have a custom mapping mode that is similar to one of the Simulink mapping modes, you can use the `getS1RootInportMap` function in your custom mapping file function.

For an example of a custom mapping function that uses this function, from the Simulink Documentation Center main page, click **Examples > Modeling Features > Using Simulink Mapping Modes in Custom Mapping Functions**.

Command-Line Interface for Input Variables

Use the `getInputString` function to programmatically supply a set of input variables to the `sim` command or to a list of input variables that you can manually paste in the **Configuration Parameters > Data Import/Export > Input** parameter.

Import MATLAB timeseries Data

In this section...

“Specify Time Dimension” on page 48-121

“Models with Multiple Root Inport Blocks” on page 48-122

A root-level Inport, Enable, Trigger, and From Workspace block can import data specified by a MATLAB `timeseries` object residing 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.

Specify Time Dimension

When you create a MATLAB `timeseries` object to import data to Simulink, the time dimension 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. <pre>t = (0:10)'; ts = timeseries(sin(t), t);</pre>
2D (including row and column vectors) or greater	Last	Constructor for a matrix signal. Time is aligned with the last dimension. <pre>t = 0;</pre>

Signal Data Dimension or Type	Time Dimension Alignment	Example of timeseries Constructor
2D row vector, and there is only one time step	Last	<pre>ts = timeseries([1 2; 3 4], t);</pre> <p>'InterpretSingleRowDataAs3D', true</p> <p>For example:</p> <pre>t = 0; ts = timeseries([1 2], t, 'InterpretSingleRowDataAs3D', true);</pre>

Models with Multiple Root Inport Blocks

To use a MATLAB `timeseries` object for a root Inport block in a model that has multiple root Inport blocks, convert all of the other root Inport block data that uses `Simulink.TsArray` or `Simulink.Timeseries` objects to MATLAB `timeseries` objects or to a structure of MATLAB `timeseries` objects.

You can use the `Simulink.Timeseries.convertToMATLABTimeseries` method to convert a `Simulink.Timeseries` object to a MATLAB `timeseries` object. For example, if `sim_ts` is a `Simulink.Timeseries` object, then the following line converts `sim_ts` to a MATLAB `timeseries` object:

```
ts = sim_ts.convertToMATLABTimeseries;
```

Import Structures of timeseries Objects for Buses

In this section...

“Imported Bus Data Requirements” on page 48-123

“Convert Simulink.TsArray Objects” on page 48-123

“Import Bus Data” on page 48-124

“Import Array of Buses Data” on page 48-126

Imported Bus Data Requirements

A root-level input port defined by a bus object (see `Simulink.Bus`) can import data from a structure of MATLAB `timeseries` objects that represent the bus elements for which you want to specify values. Any bus elements for which you do not include a field in the structure use ground values.

The structure of MATLAB `timeseries` 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.

Convert Simulink.TsArray Objects

If you use logged data from a model whose **Configuration Parameters > Data Import/Export > Signal logging format** parameter is set to `ModelDataLogs` format instead of the default `Dataset` format, consider converting the data to use a structure of MATLAB `timeseries`

objects. The `ModelDataLogs` format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release.

You can create a structure of MATLAB `timeseries` objects from a `Simulink.TsArray` object. For example, if `tsa` is a `Simulink.TsArray` object:

```
input = Simulink.SimulationData.createStructOfTimeseries(tsa);
```

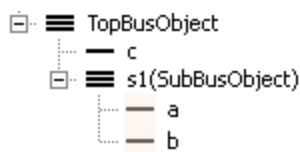
Note If you use a structure of MATLAB `timeseries` objects for a root `Inport` block in a model that has multiple root `Inport` blocks, all the root `Inport` blocks must use MATLAB `timeseries` objects. Convert any root `Inport` block data that uses `Simulink.TsArray` or `Simulink.Timeseries` objects to be MATLAB `timeseries` objects.

Import Bus Data

Use the following procedure to set up a model to import bus data to a root `Inport` block.

The examples in this procedure assume that you have a model that is set up as follows:

- The `TopBusObject` bus object has two elements:
 - `c`
 - `s1`, which is a sub-bus that has two elements: `a` and `b`.



- The model has two root `Inport` blocks: `In1` and `In2`.
 - The `In1` `Inport` block imports non-bus data.
 - The `In2` `Inport` block imports bus data of type `TopBusObject`.

- 1 Create a MATLAB `timeseries` object for each root Inport or Trigger block for which you want to import non-bus data.

For example:

```
N = 10;
Ts = 1;
t1 = ((0:N)* Ts)';
d1 = sin(t1);
in1 = timeseries(d1,t1)
```

- 2 Create a structure of MATLAB `timeseries` objects, with one `timeseries` object for each leaf bus element for which you do not want to use ground values.

For example, to specify non-ground values for all the elements in the `s2` bus:

```
in2.c = timeseries(d1,t1);
in2.s1.a = timeseries(d2,t2);
in2.s1.b = timeseries(d3,t3);
```

The MATLAB `timeseries` objects that you create must match the corresponding bus elements, as described in “Imported Bus Data Requirements” on page 48-123.

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, you can use the `Simulink.Bus.getNumLeafBusElements` and `Simulink.Bus.getLeafBusElements` methods. For example, for the bus object `MyBus`:

```
num_el = MyBus.getNumLeafBusElements;
el_list = MyBus.getLeafBusElements;
```

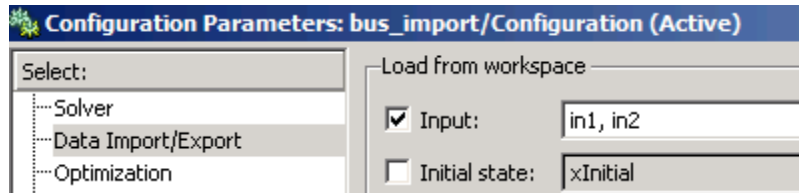
To create a structure of MATLAB `timeseries` objects from a bus object and a cell array of `timeseries` or `Simulink.Timeseries` objects, use the `Simulink.SimulationData.createStructOfTimeseries` utility. For example:

```
input = ...
Simulink.SimulationData.createStructOfTimeseries('MyBus',...
{ts1,ts2,ts3});
```

The number of `timeseries` objects in the cell array must match the number of leaf elements in the bus object (in this example, `num_el`). The data type, dimensions, and complexity of each `timeseries` object must match those attributes of the corresponding bus object leaf node (in this example, the attributes listed in `el_list`).

- 3 In the **Configuration Parameters > Import/Export > Input** parameter edit box, enter a comma-separated list of MATLAB `timeseries` objects and structures of MATLAB `timeseries` objects.

For example, the `in1` `timeseries` object and the `in2` structure of `timeseries` objects.



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 the logged data for an array of buses signal from a previous simulation as roundtrip input to a root-level `Inport` block 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 fully specify 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

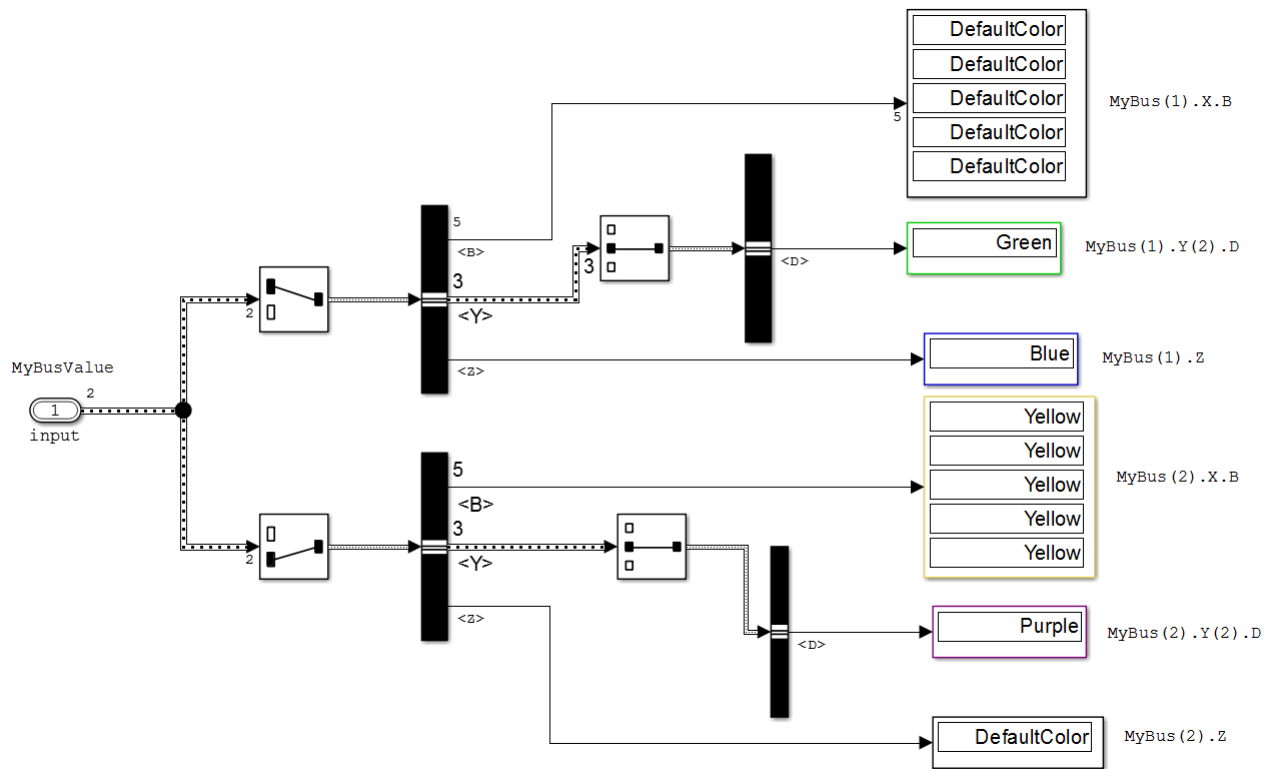
To specify partial data for array of buses, create a MATLAB array of structures with MATLAB `timeseries` objects at the leaf nodes.

The structure that you create to specify partial data must be consistent with these rules:

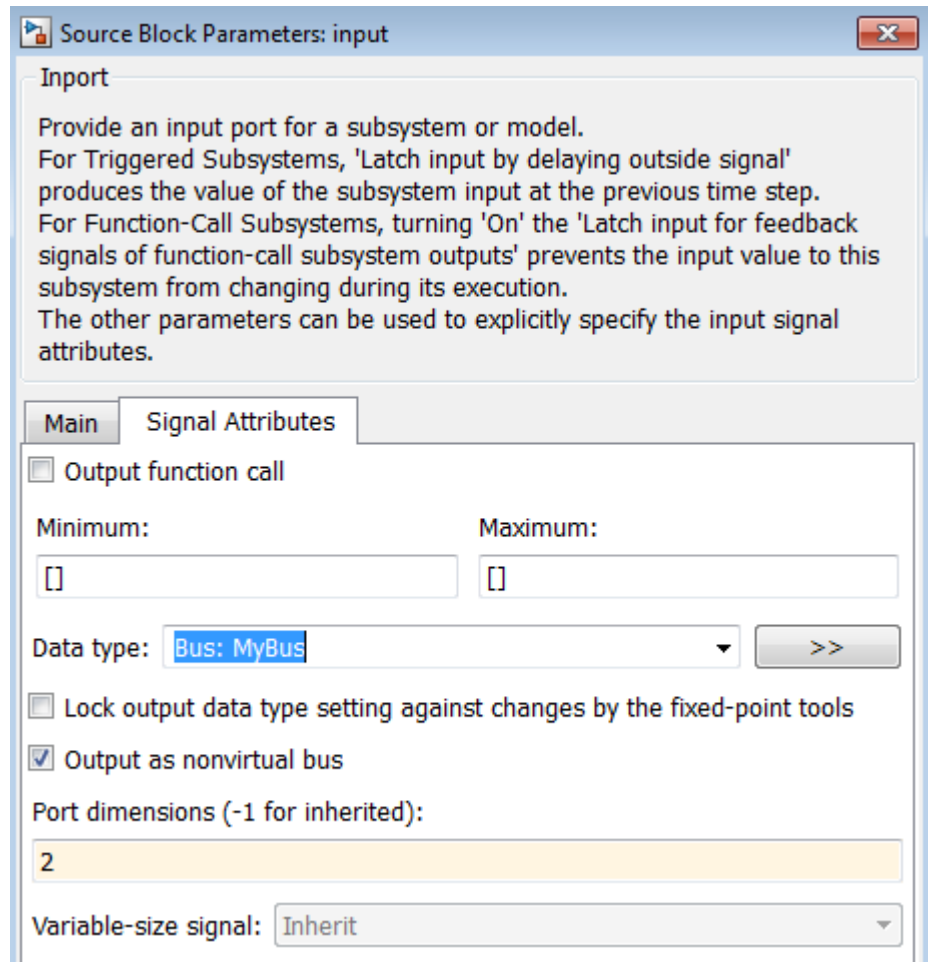
- You can omit fields, including leaf nodes and sub-branches. You can also omit dimensions. If you do not specify a field, Simulink uses the ground value for that field.
- For sub-bus nodes, make the dimension of each field equal to, or smaller than, the dimension for the corresponding node of the array 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`, it looks like this:

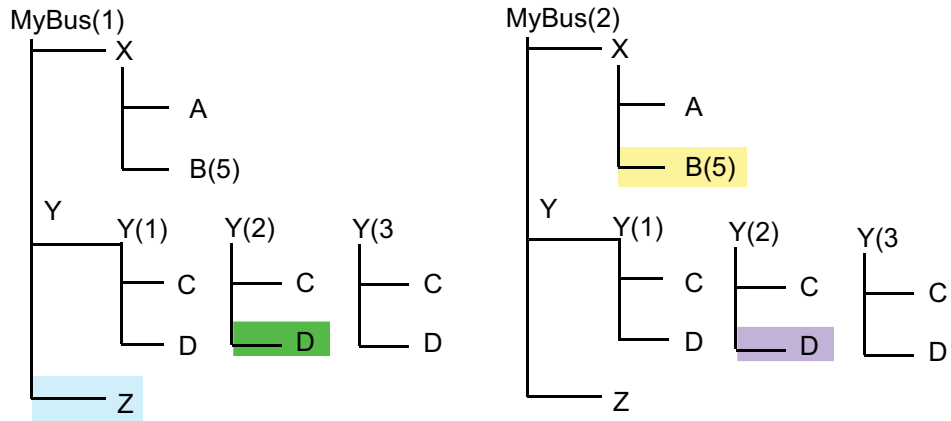


The input Inport block uses the MyBus bus object as its data type.



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`.

```

7 %% Create timeseries data for leaf signals
8 - N = 10; Ts = 1;
9 - Time = ((0:N)*Ts)';
10 - [m, n] = size(Time);
11
12 - ZData = repmat(ex_partial_loading_aob_basicColors.Blue,m,n);
13 - BData = repmat(ex_partial_loading_aob_basicColors.Yellow,m,5);
14 - D1Data = repmat(ex_partial_loading_aob_basicColors.Green,m,n);
15 - D2Data = repmat(ex_partial_loading_aob_basicColors.Purple,m,n);
16
17 % Create individual timeseries objects to represent Z, B, D1 and D2
18 - ZT = timeseries(ZData, Time, 'Name', 'Z');
19 - BT = timeseries(BData, Time, 'Name', 'B');
20 - D1T = timeseries(D1Data, Time, 'Name', 'D1');
21 - D2T = timeseries(D2Data, Time, 'Name', 'D2');
22
23 %% Construct MyBusValue(1)
24
25 % Y(2).D
26 - Y(2) = struct('D',D1T);
27
28 % Set X.B to empty to allow the second element to support specification for
29 % this field
30 - X = struct('B',[]);
31
32 % MyBusValue(1)
33 - MyBusValue(1) = struct('Z',ZT,'Y',Y,'X',X);
34 - clear X Y;
35
36 %% Construct MyBusValue(2)
37
38 % Specify timeseries for MyBusValue(2).X.B
39 - MyBusValue(2).X.B = BT;
40
41 % Specify timeseries for MyBusValue(2).Y(2).D
42 - MyBusValue(2).Y(2).D = D2T;
43

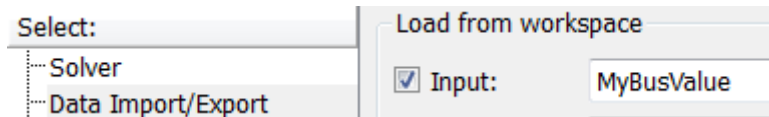
```

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 sub-buses of `Y` (that is, `Y(1)` and `Y(3)`).

This example specifies data for `Y(2)`; you can skip the first and last sub-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`.



Import Simulink.Timeseries and Simulink.TsArray Data

In this section...
“Use MATLAB Timeseries for New Models” on page 48-133
“Simulink.TsArray Data” on page 48-133

Use MATLAB Timeseries for New Models

For new models, have root-level Inport block or Trigger blocks import MATLAB timeseries data.

However, you can import existing Simulink.Timeseries object data. Importing Simulink.Timeseries objects allows you to import data logged by a previous simulation run that used the ModelDataLogs format for signal logging (see “Export Signal Data Using Signal Logging” on page 48-21).

Simulink.TsArray Data

Objects of the Simulink.TsArray class have a variable number of properties. The first property, called Name, specifies the log name of the logged signal. The remaining properties reference logs for the elements of the logged signal: Simulink.Timeseries objects for elementary signals and Simulink.TsArrayobjects for mux or bus signals. The name of each property is the log name of the corresponding signal.

Import Data Arrays

In this section...

“Data Array Format” on page 48-134

“Specify the Input Expression” on page 48-134

Data Array Format

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. In particular, each column represents the input for a different Inport or Trigger block signal (in sequential order) and each row is the input value for the corresponding time point. For a Trigger block, the signal driving 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's input ports.

Specify the Input Expression

The default input expression for a model is `[t,u]` and the default input format is `Array`. So if you define `t` and `u` in the MATLAB workspace, you need only select the **Input** option to input data from the model workspace. For example, suppose that a model has two input ports, `In1` that accepts two signals, and `In2` that accepts one signal. Also, suppose that the MATLAB workspace defines `t` and `u` as follows:

```
N = 10;  
Ts = 0.1  
t = ((0:N)* Ts)';  
u = [sin(t), cos(t), 4*cos(t)];
```

When the simulation runs, the signals `sin(t)` and `cos(t)` will be assigned to `In1` and the signal `4*cos(t)` will be assigned to `In2`.

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/or data types other than `double`.

Import MATLAB Time Expression Data

Specify the Input Expression

You can use a MATLAB time expression to import data from a workspace. To use a time expression, enter the expression as a string (i.e., 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. For example, suppose that a model has one vector Inport that accepts two signals. Furthermore, suppose that `timefcn` is a user-defined function that returns a row vector two elements long. The following 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 model's input ports. Note that the Simulink software 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)`.

Import Data Structures

In this section...

“Data Structures” on page 48-137

“One Structure for All Ports or a Structure for Each Port” on page 48-138

“Specify Signal Data” on page 48-138

“Specify Time Data” on page 48-139

“Examples of Specifying Signal and Time Data” on page 48-140

Data Structures

The Simulink software can read 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 a Structure Array” in the MATLAB documentation.

The structure always includes a signals substructure, which contains a values field and a dimensions field. For details about the signal data, see “Specify Signal Data” on page 48-138. Depending on the modeling task that you want to perform, the structure can also include a time field.

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 “One Structure for All Ports or a Structure for Each Port” on page 48-138.

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) or continuous signals (the signal is defined for all values of time). For discrete signals, use a structure that has an empty time vector. For continuous signals, 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 details, see:

- “Import Data to Test a Discrete Algorithm” on page 48-81

- “Import Data to Model a Continuous Plant” on page 48-78
- “Import Data for an Input Test Case” on page 48-82

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. For details, see “Specify Signal Data” on page 48-138.

For continuous signals, you may want to specify a `time` field, which contains a time vector. See “Time and Signal Values for Imported Data” on page 48-75.

One Structure for All Ports or a Structure 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 separated structure for each port.

The per-port structure format consists of a separate structure-with-time or structure-without-time for each port. Each port’s input data structure 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`, where `in1` is the data for your model’s first port, `in2` for the second input port, and so on.

The rest of the section about importing structure data focuses on specifying one structure for all ports.

Specify Signal Data

Each `signals` substructure must contain two fields: `values` and `dimensions`.

The Values Field

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 where `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 your model's 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

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.

Note You must 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, an error message is displayed when you try to simulate the model.

Specify Time Data

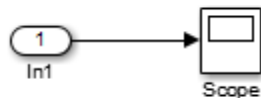
You can specify a time vector as part of the data structure to import. The “Time and Signal Values for Imported Data” on page 48-75 section indicates when you may want to add a time vector.

The following table provides recommendations for how to specify time values, based on the kind of signal data you want to import.

Signal Data	Time Data Recommendation
Inport or Trigger block with a discrete sample time	Do not specify a time vector. Simulink reads one signal value at each time step.
Evenly-spaced discrete signals	Consider using an expression in the following form: <code>TimeVector = Ts * (0:N);</code> where Ts is the time step and N is the number of time steps.
Unevenly-spaced values	Use any valid MATLAB array expression; for example, <code>[1:5 5:10]</code> or <code>(1 6 10 15)</code> . If the root-level input port is from a From Workspace, From File, or Signal Builder block, which support zero-crossing detection, 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 = (0:N)*Ts';
```

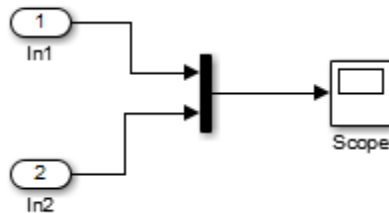
```

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 Input 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. To do this, define a structure, **a**, as follows, in the MATLAB workspace:

```

a.time = (0:0.1:1)';
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 Note that in this model you do not need to specify the dimension and data type, because the default values are 1 and **double**.

Import and Export States

In this section...

“State Information” on page 48-142

“Save State Information” on page 48-142

“Import Initial States” on page 48-146

“Import and Export State Information for Referenced Models” on page 48-149

State Information

Some blocks maintain state information that Simulink uses for calculating values during simulation. For example, the state information for a Unit Delay block is the output signal value from the previous simulation step, which the block uses for calculating the output signal value for the current simulation step.

You can use saved state information to capture a known state. Some examples of uses of saved state information 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.

Save State Information

You can save state information by either:

- Saving the final state of a simulation, using the `SimState`
- Saving partial state information at the end of a simulation or for each simulation step, using a structure, structure with time, or array

`SimState` provides complete state information. However, use of `SimState` has some limitations, such as:

- You can use only the Normal or the Accelerator mode of simulation.

- `SimState` does not support code generation, including Model Reference in accelerated modes.

To ensure that `SimState` meets your modeling requirements, see “Limitations of the `SimState`” on page 14-38.

When you save final states without the `SimState`, you save only logged states — the continuous and discrete states of blocks — which are a subset of the complete simulation state of the model. However, if you use a structure or structure with time, you may find that the state information is easier to read than if you use a `SimState`.

For more information, see:

- “Save Complete State Information with `SimState`” on page 48-143
- “Save Partial Final State Information” on page 48-143

Save Complete State Information with `SimState`

To save complete state information, save the `SimState` 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 complete `SimState` in final state** parameter.
- 3** In the edit box adjacent to the **Save complete `SimState` in final state** parameter, enter a variable name for the `SimState`.
- 4** Simulate the model.

For more information about using the `SimState`, see “Save and Restore Simulation State as `SimState`” on page 14-32.

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** Also in the **Data Import/Export** pane, set the **Format** parameter to **Structure** or **Structure with time**. For details, see “Format for Saved State Information” on page 48-144
- 4** Simulate the model.

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 **Structure** or **Structure with time**, unless you need to use array format for compatibility with a legacy model. For details, see “Format for Saved State Information” on page 48-144.
- 4** Simulate the model.

Format for Saved State Information

If you do not use the `SimState` 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:

- Array (default)
- Structure

- Structure with time

The `Structure` and `Structure with time` formats are easier to read and consistent across simulations. Also, these two 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.

The `Array` option for the **Configuration Parameters > Data Import/Export > Format** option supports compatibility with models developed in earlier releases, when Simulink supported only the array format for saving state information.

The array format reflects the order of signals. The order of saved state information can change between simulations when you change any of the following:

- The model (even without changing the signal)
- The simulation mode
- The code generation mode

Examine State Information Saved Without the SimState

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.

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, at the MATLAB command line, type

```
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, type

```
xFinal.signals(1)  
  
ans =  
  
    values: 2.0108  
  dimensions: 1  
    label: 'CSTATE'  
  blockName: 'vdp/x1'  
  stateName: ''  
inReferencedModel: 0
```

The values and blockName fields of first state structure shows that the final value for the output signal of the x1 block was 2.018.

Note If you write a script to analyze state information, use a combination of label and blockName values to uniquely identify a specific state. Do not rely on the order of the states.

Import Initial States

To import states, enable **Configuration Parameters > Data Import/Export > Initial state** and specify a variable that contains the

initial state values. For example, you could specify a variable that contains state information saved from a previous simulation.

You can import state information to initialize a simulation:

- 1** Enable **Configuration Parameters > Data Import/Export > Initial state**.
- 2** In the **Initial state** edit box, enter the name of the variable for the state information that you want to use for initialization. You can create your own state information in MATLAB or you can use state information saved from a previous simulation. For details about using saved state information, see “Import Saved State Information” on page 48-147.

The initial values that the variable specifies override the initial state values that the blocks in the model specify in initial condition parameters.

Import Saved State Information

You can import saved state information as the initial state.

- 1** Enable **Configuration Parameters > Data Import/Export > Initial state**.
- 2** In the **Initial state** edit box, enter the name of the variable in the **Final states** edit box. The state information that Simulink loads depends on the setting of **Configuration Parameters > Data Import/Export > Save complete SimState in final state**.

Setting for the “Save complete SimState in final state” parameter	State Information
Enabled	Complete state information. For details, see “Save and Restore Simulation State as SimState” on page 14-32.
Cleared	State information for logged states (the continuous and discrete states of blocks), in the format specified in Configuration Parameters >

Setting for the “Save complete SimState in final state” parameter	State Information
	Data Import/Export > Format. For details, see “Format for Saved State Information” on page 48-144.

For example, the following commands create an initial state structure that initialize 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 state.

```
% Open the vdp model
vdp

% Use getInitialState to obtain an initial state structure
states = Simulink.BlockDiagram.getInitialState('vdp');

% Set the initial value of the signals structure element
% associated with x2 to 2.
states.signals(2).values = 2;

% Remove the signals structure element associated with x1
states.signals(1) = [];
```

To use the `states` variable, for the vdp model, enable the **Configuration Parameters > Data Import/Export > Initial state** option (in the **Load from workspace** area). Enter `states` into the associated edit field. When you run the model, note that 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.

Import and Export State Information for Referenced Models

Saved State Information

When Simulink saves states from a referenced model in the structure-with-time format, Simulink adds a Boolean subfield (named `inReferencedModel`) to the `signals` field of the saved data structure. The value of this additional field is true (1) if the `signals` field records the final state of a block that resides in the submodel, and a 0 otherwise. For example:

```
>> xout.signals(1)

ans =

        values: [101x1 double]
    dimensions: 1
         label: 'DSTATE'
    blockName: [1x66 char]
inReferencedModel: 1
```

If the `signals` field records a submodel state, its `blockName` subfield contains a compound path of a top model path and a submodel path. The top model path is the path from the model root to the Model block that references the submodel. The submodel path is the path from the submodel root to the block whose state the `signals` field records. The compound path uses a `|` character to separate the top and submodel paths. For example:

```
>> xout.signals(1).blockName

ans =

sldemo_md1ref_basic/CounterA|sldemo_md1ref_counter/Previous Output
```

State Initialization

Use the structure or structure with time format to initialize the states of a top model and the models that it references.

Working with Data Stores

- “About Data Stores” on page 49-2
- “Data Stores with Data Store Memory Blocks” on page 49-8
- “Data Stores with Signal Objects” on page 49-12
- “Access Data Stores with Simulink Blocks” on page 49-15
- “Data Store Examples” on page 49-24
- “Log Data Stores” on page 49-27
- “Order Data Store Access” on page 49-32
- “Data Store Diagnostics” on page 49-39
- “Data Stores and Software Verification” on page 49-47

About Data Stores

In this section...

“Local and Global Data Stores” on page 49-2

“When to Use a Data Store” on page 49-3

“Create Data Stores” on page 49-3

“Access Data Stores” on page 49-4

“Configure Data Stores” on page 49-5

“Data Stores with Buses and Arrays of Buses” on page 49-5

Local and Global Data Stores

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. 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 49-24.

For information about using referenced models, see “Model Reference”.

Customized Data Store Access Functions in Generated Code

Embedded Coder provides a custom storage class that you can use to specify customized data store access functions in generated code. See “Apply Custom Storage Classes” and “GetSet Custom Storage Classes”.

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. See “Data Stores and Software Verification” on page 49-47 for more information.

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.

Create Data Stores

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 49-8.
- A `Simulink.Signal` object can act as a local or global data store. See “Data Stores with Signal Objects” on page 49-12.

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.

Access Data Stores

To write a signal to a data store, use a Data Store Write block, which inputs the value of a signal and writes that value to the data store.

To read a signal from a data store, use a Data Store Read block, which reads the value in the data store and outputs that value as a signal.

For Data Store Write and Data Store Read blocks, to identify the data store to be read from or written to, specify the data store name as a block parameter. See “Access Data Stores with Simulink Blocks” on page 49-15 for more information.

Data Store Logging

You can log the values of a local or global data store data variable for all the steps in a simulation. See “Log Data Stores” on page 49-27.

Configure 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 49-6.

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 49-47.
- 2** Create data stores using the techniques described in “Data Stores with Data Store Memory Blocks” on page 49-8 or “Data Stores with Signal Objects” on page 49-12. For greater reliability, consider assigning rather than inheriting data store attributes, as described in “Specifying Data Store Memory Block Attributes” on page 49-8.
- 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 49-15.
- 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 49-32.
- 5** Apply the techniques described in “Data Store Diagnostics” on page 49-39 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 the Simulink Coder documentation.

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 “Configure Data Stores” on page 49-5.

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 “Manage Bus Objects with the Bus Editor” on page 51-28.
- 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 46-27.
- 4** In the **Model Configuration Parameters > Diagnostics > Connectivity** pane, set the **Mux blocks used to create bus** diagnostic to error. For details, see “Prevent Bus and Mux Mixtures” on page 51-100.
- 5** If you use a MATLAB structure for the initial value of the data store, then in the **Model Configuration Parameters > Diagnostics > Data Validity** pane, set the **Underspecified initialization detection**

diagnostic to error. For details, see “Specify Initial Conditions for Bus Signals” on page 51-69 and “Underspecified initialization detection”.

- 6 (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 49-17.

Data Stores with Data Store Memory Blocks

In this section...

“Creating the Data Store” on page 49-8

“Specifying Data Store Memory Block Attributes” on page 49-8

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 dialog box to define the data store’s properties. The **Data store name** property specifies the name to 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 49-9 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:

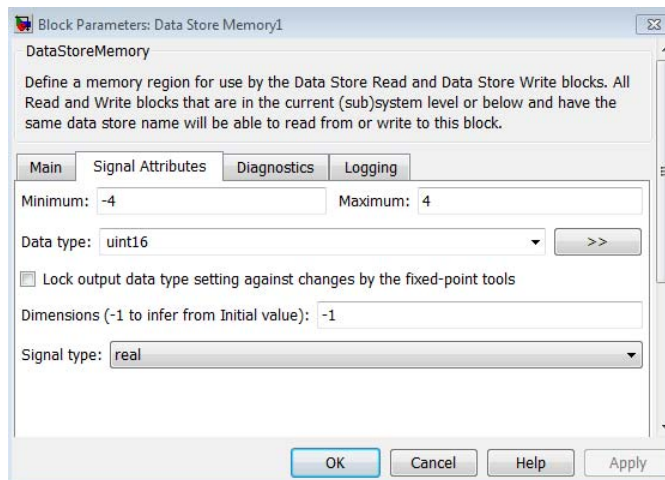
- Data type

- 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 to specify the data type and complexity of a data store. In the next figure, the dialog box sets the **Data type** to `uint16` and the **Signal type** to `real`.

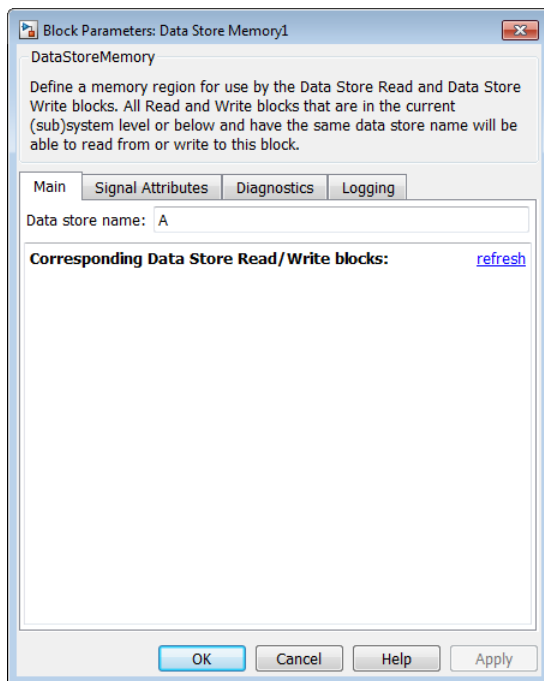


Specifying Attributes Using a Signal Object

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 49-12, 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.



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.SignalCoderInfo]
  Description: ''
  DataType: 'auto'
  Min: []
```

```
Max: []
DocUnits: ''
Dimensions: 1
DimensionsMode: 'auto'
Complexity: 'auto'
SampleTime: -1
SamplingMode: 'auto'
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 49-12.

Data Stores with Signal Objects

In this section...

“Creating the Data Store” on page 49-12

“Local and Global Data Stores” on page 49-12

“Signal Object Attributes for Data Stores” on page 49-12

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, 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 parameter 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.SamplingMode='Sample based';
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
- SamplingMode

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

Parameter	Global Data Store Value
Dimensions	Can be set or inherited
SampleTime	Can be set or inherited
SamplingMode	Must be 'Sample based'

Access Data Stores with Simulink Blocks

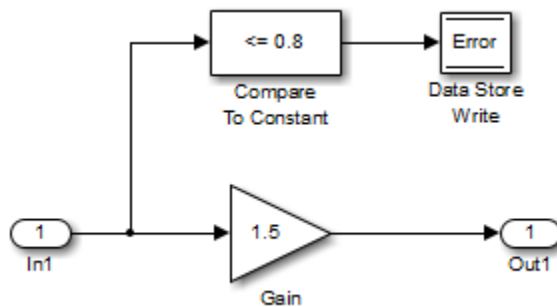
In this section...

- “Writing to a Data Store” on page 49-15
- “Reading from a Data Store” on page 49-15
- “Accessing a Global Data Store” on page 49-16
- “Accessing Specific Bus and Matrix Elements” on page 49-17

Writing to a Data Store

To set the value of a data store at each time step:

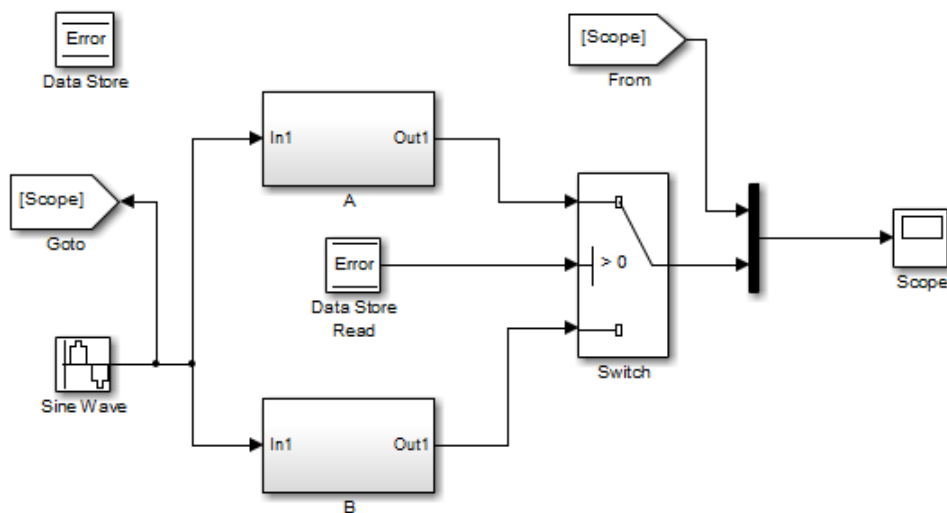
- 1 Create an instance of a Data Store Write block at the level of your model that computes the value.
- 2 Set the Data Store Write block **Data store name** parameter to the name of the data store to which you want it to write data.
- 3 Connect the output of the block that computes the value to the input of the Data Store Write block.



Reading from a Data Store

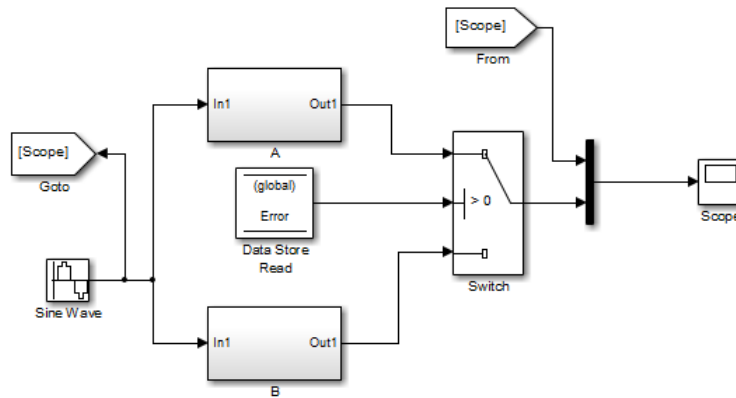
To get the value of a data store at each time step:

- 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.



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:

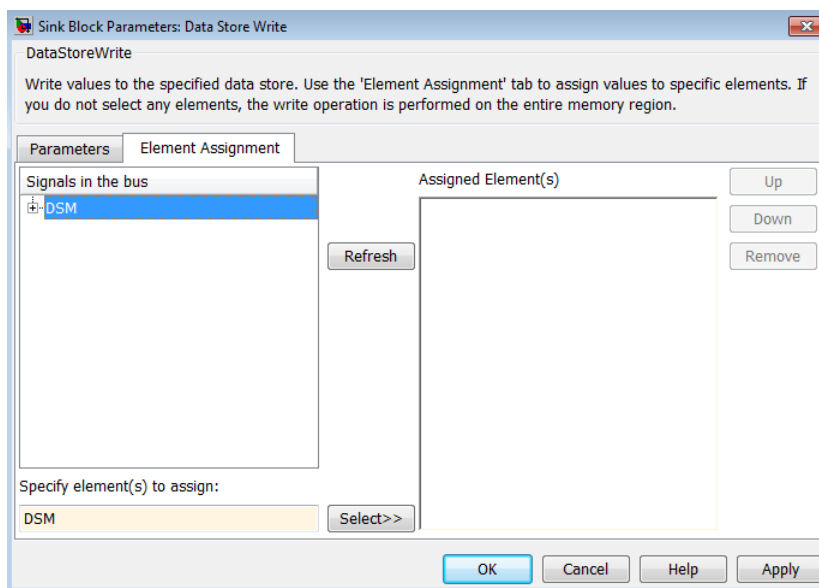
- Reducing the number of blocks in the model. For example, you can eliminate a Data Store Read and Bus Selector block pair or a Data Store Write and Bus Assignment block pair for each specific bus element that you want to access).
- Faster simulation of models with large buses and arrays of buses.

Writing Specific Elements to a Data Store

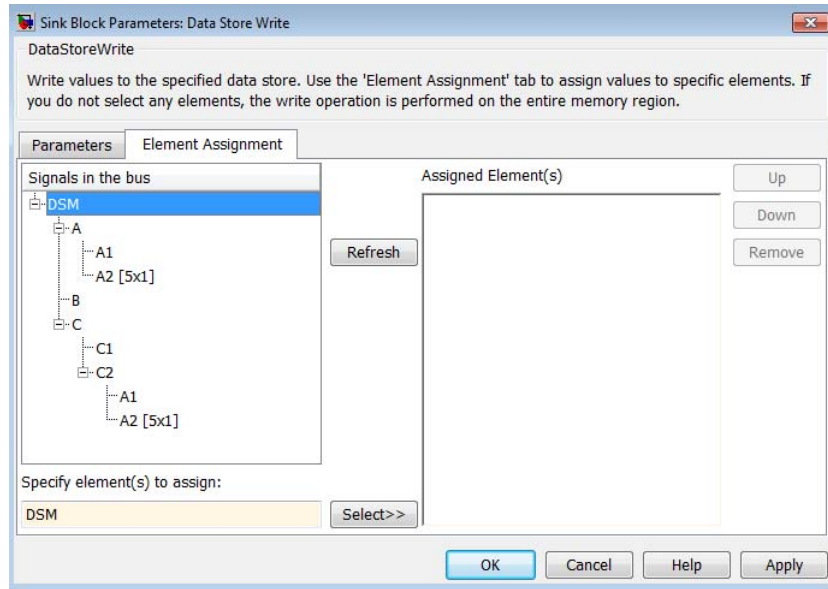
Note The following procedure describes how to use the Data Store Write block interface to write specific elements to a data store. You can also perform this task at the command line, using the `DataStoreElements` parameter to specify elements. For details, see “Specification using the command line” on page 49-23.

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:

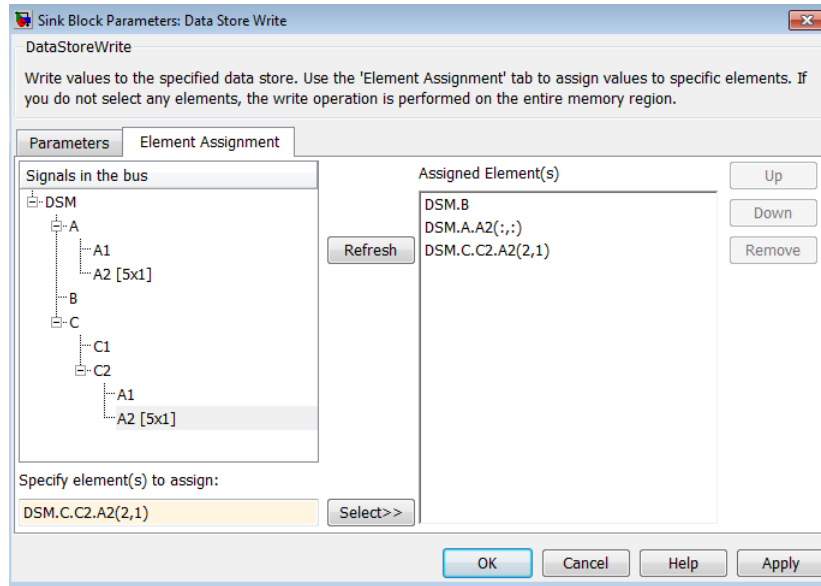


- 2 Expand all the elements in the **Signals in the bus** list.



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 subbus), select A2 [5x1]. Then click **Select>>**.
- To write the second element of A2 in the C2 subbus, 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 49-21.

- 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 49-18. The Data Store Read block differs slightly from the Data Store Write block. A Data Store Read block has:

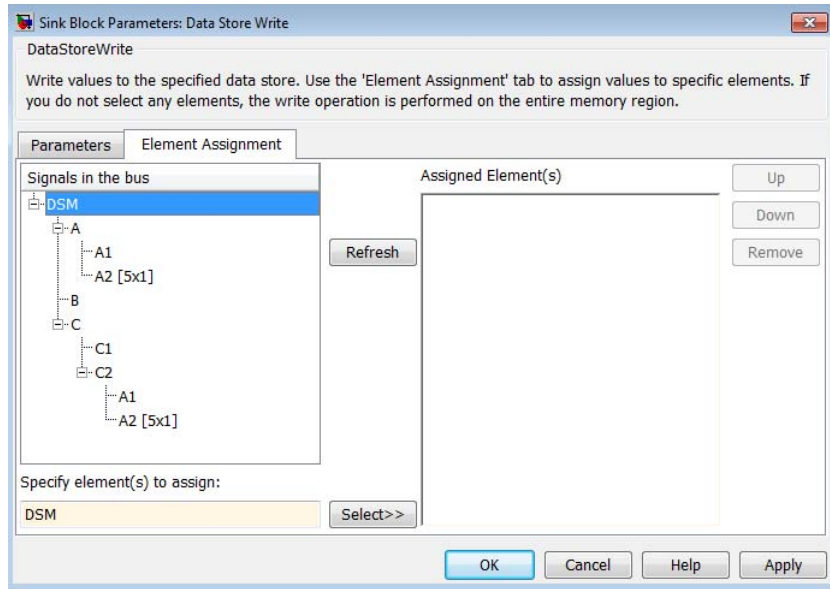
- An **Element Selection** pane instead of an **Element Assignment** pane
- A **Selected element(s)** list instead of an **Assigned element(s)** list

Specifying Elements to Assign or Select

Use MATLAB matrix element syntax to specify specific elements. For details about specifying matrices in MATLAB, see “Creating and Concatenating 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 subbus of the A bus, as shown in the bus hierarchy used in “Writing Specific Elements to a Data Store” on page 49-18.



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 subbus of the A bus, as shown in the bus hierarchy used in “Writing Specific Elements to a Data Store” on page 49-18.

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(gcf, '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.

Data Store Examples

In this section...
“Overview” on page 49-24
“Local Data Store Example” on page 49-24
“Global Data Store Example” on page 49-25

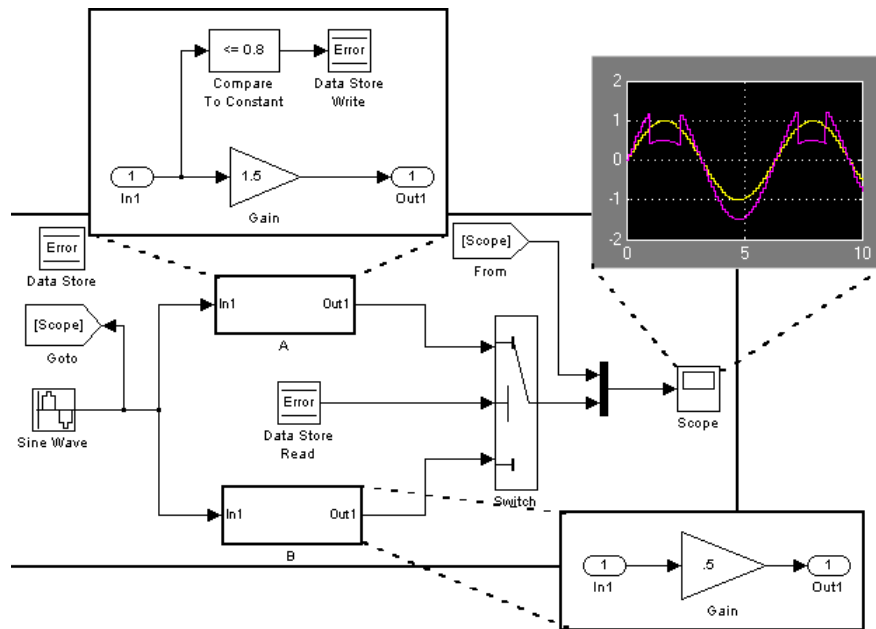
Overview

The following examples illustrate techniques for defining and accessing data stores. See “Order Data Store Access” on page 49-32 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 49-39 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_mdref_dsm` model, which shows how to use global data stores to share data among referenced models.

Local Data Store Example

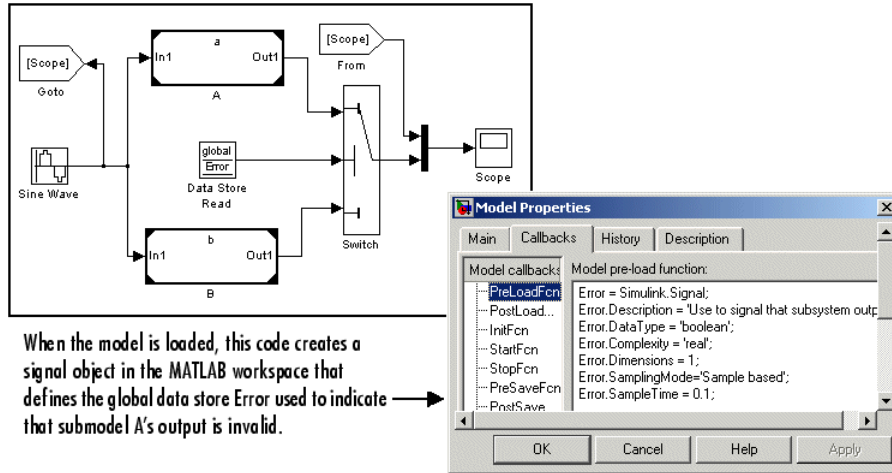
The following model illustrates creation and access of a local data store, which is visible only in a model or particular subsystem.



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 submodels 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.

Log Data Stores

In this section...

“Logging Local and Global Data Store Values” on page 49-27

“Supported Data Types, Dimensions, and Complexity for Logging Data Stores” on page 49-27

“Data Store Logging Limitations” on page 49-28

“Logging Data Stores Created with a Data Store Memory Block” on page 49-28

“Logging Icon for the Data Store Memory Block” on page 49-29

“Logging Data Stores Created with a Simulink.Signal Object” on page 49-29

“Accessing Data Store Logging Data” on page 49-30

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 for identifying the impact of a model modification

To see an example of logging a global data store, see the `sldemo_mdref_dsm` model.

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.
- If you set the **Model Configuration Parameters > Solver > Tasking mode for periodic sample times** parameter to **MultiTasking**, then you cannot log Data Store Memory blocks that use asynchronous sample times or hybrid sample times (that is, sample times resulting from when different data sources for the data store have different sample times).

For details about viewing information about sample times, see “View Sample Time Information” on page 5-9.

- You cannot log data stores that use custom data types.

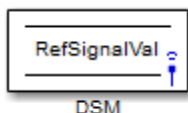
Logging Data Stores Created with a Data Store Memory Block

To log a local data store that you create with a Data Store Memory block:

- 1 In the Block Parameters dialog box for the Data Store Memory block that you want to log, select the **Logging** pane.
- 2 Select the **Log signal data** check box.
- 3 Optionally, specify limits for the amount of data logged, using the **Minimum** and **Maximum** parameters.
- 4 Enable data store logging with the **Model Configuration Parameters > Data Import/Export > Data stores** parameter.
- 5 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 49-12 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 49-12.
- 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
<code>Simulink.SimulationData.BlockPath</code>	Represents a fully specified Simulink block path; use for capturing the full model reference hierarchy
<code>Simulink.SimulationData.Dataset</code>	Stores logged data elements and provides searching capabilities; use to group <code>Simulink.SimulationData.Element</code> objects in a single object
<code>Simulink.SimulationData.DataStoreMemory</code>	Stores logging information from a data store during simulation

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_mdref_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 Simulink1.SimulationData.BlockPath]  
DSM Writers: [101x1 uint32]  
Values: 101x1 timeseries]
```

To access an element by index, use the `Simulink.SimulationData.Dataset.getElement` method.

Order Data Store Access

In this section...
“About Data Store Access Order” on page 49-32
“Ordering Access Using Function Call Subsystems” on page 49-32
“Ordering Access Using Block Priorities” on page 49-36

About Data Store Access Order

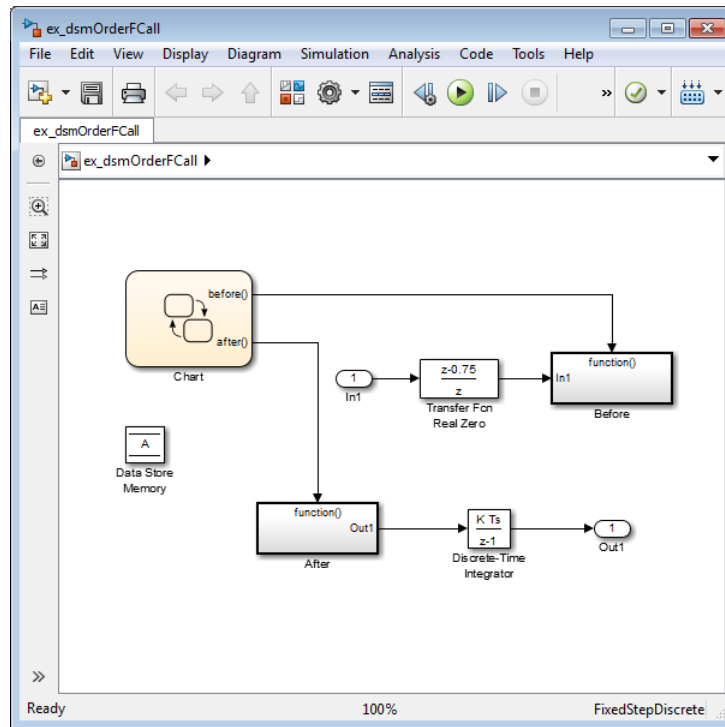
To obtain correct results from data stores, you must control the order of execution of the data store’s reads and writes. If a data store’s read occurs before its write, latency is introduced into the algorithm: the read obtains the value that was computed and stored in the previous time step, rather than the value computed and stored in the current time step.

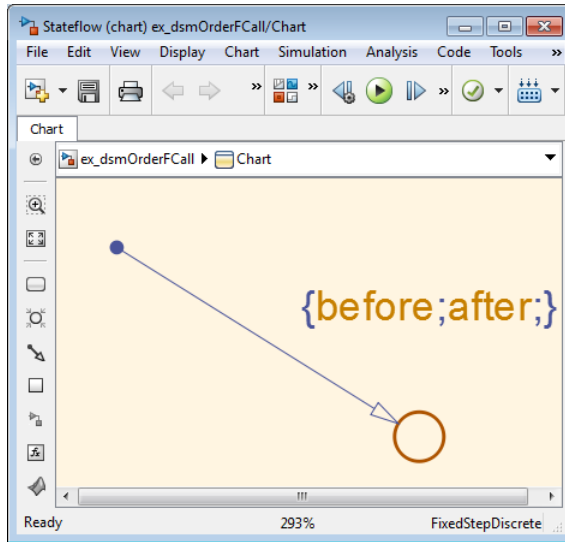
Such latency may cause the system to behave other than as designed, and in some cases may destabilize the system. Even if these problems do not occur, an uncontrolled access order could change from one release of Simulink to the next.

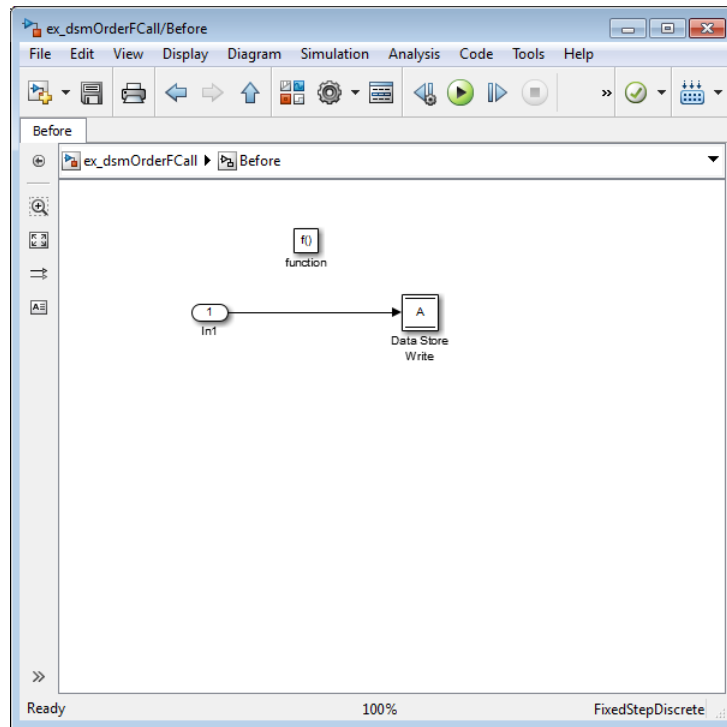
This section describes several strategies for explicitly controlling the order of execution of a data store’s reads and writes. See “Data Store Diagnostics” on page 49-39 for techniques you can use to detect and correct potential data store errors without running simulations.

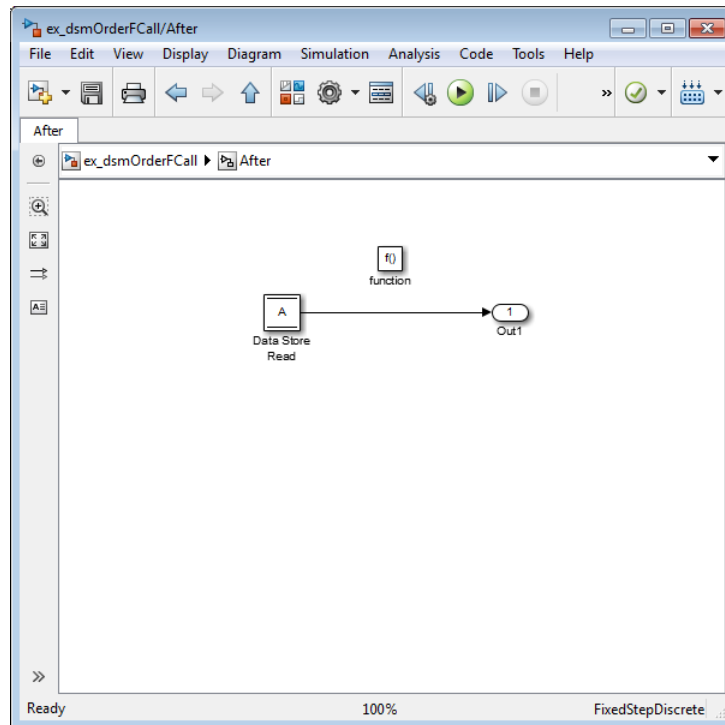
Ordering Access Using Function Call Subsystems

You can use function call subsystems to control the execution order of model components that access data stores. The next figure shows this technique:





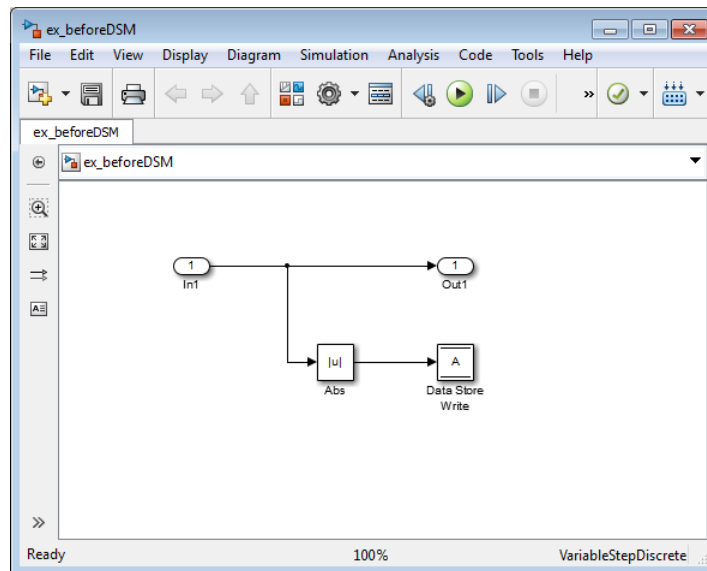
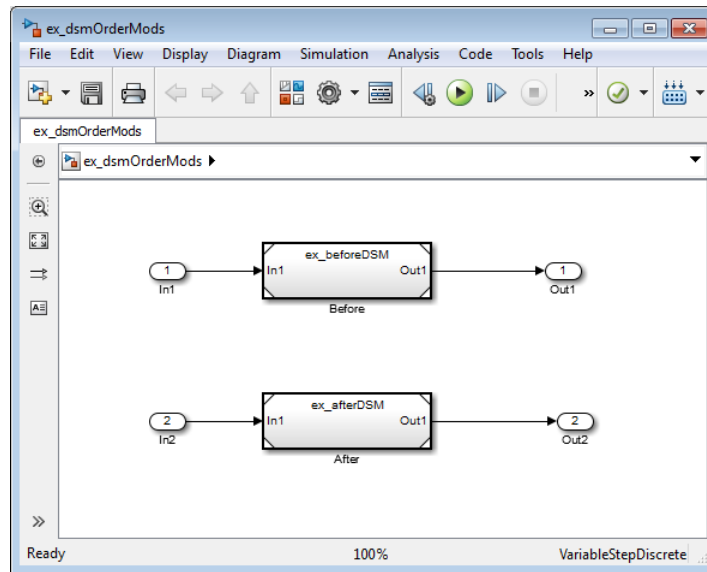


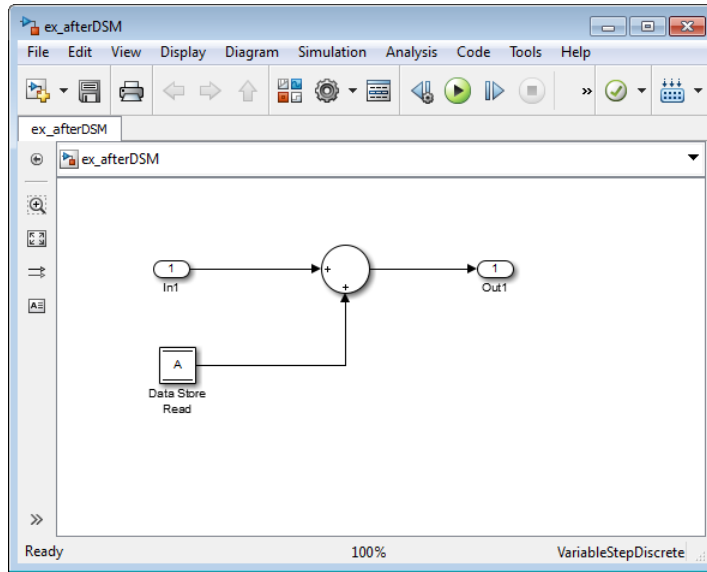


The subsystem *Before* contains the Data Store Write, and the Stateflow chart calls that subsystem before it calls the subsystem *After*, which contains the Data Store Read.

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 than `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 Store Diagnostics

In this section...

“About Data Store Diagnostics” on page 49-39

“Detecting Access Order Errors” on page 49-39

“Detecting Multitasking Access Errors” on page 49-42

“Detecting Duplicate Name Errors” on page 49-44

“Data Store Diagnostics in the Model Advisor” on page 49-46

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 when a data store is read from before written to within a given time step
- **Detect write after read:** Detect when a data store is written to after being read from within a given time step
- **Detect write after write:** Detect when a data store is written to multiple times within a given time step

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 “Diagnostics Pane: Data Validity” 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:

The image displays three overlapping windows from the MATLAB/Simulink environment:

- Top Window (dsmErrDiag):** A block diagram showing a constant block 'Constant1' with value '12' connected to a 'Data Store Write' block 'z'. A 'Chart' block is connected to a 'Display1' block and an 'Out3' block. A 'Data Store Memory' block 'z' is also present.
- Middle Window (Stateflow chart dsmErrDiag/Chart):** A stateflow chart with a single state containing the code:
$$\{y = 2*Z+A[0];$$
$$Z--; A--;\}$$
- Bottom Window (Simulation Diagnostics: dsmErrDiag):** A message window showing a block error. The message text is: "The block 'dsmErrDiag/Chart' is writing to element 1 of the data store 'dsmErrDiag/Data Store Memory' but the block 'dsmErrDiag/Data Store Write' has already written to this memory at time 0.0".

An error occurred during simulation because the data store A is read from the Stateflow chart before the Data Store Write updates it. 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 disable in the Data Store Write block. If you use this technique, be sure to set the parameter to error in all other Data Store Write blocks aside from those which are to be intentionally excluded from the diagnostic.

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

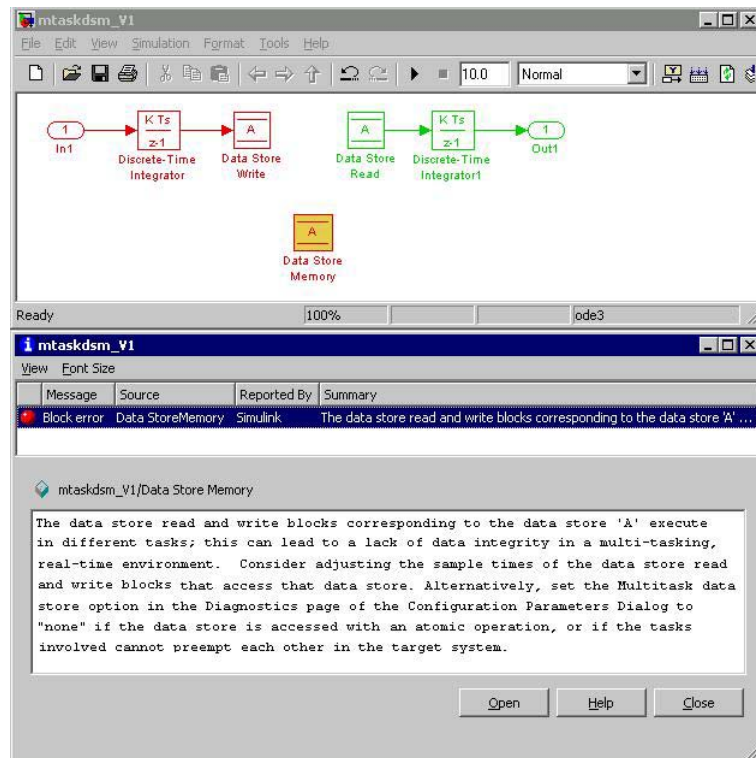
Data integrity may be compromised if a data store is read from in one task and written to in another task. For example, suppose that:

- 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.

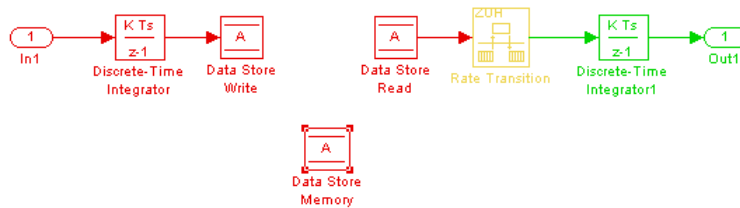
Unless you are certain that task preemption cannot cause data integrity problems, set the compile-time diagnostic **Model Configuration Parameters > Diagnostics > Data Validity > Data Store Memory Block > Multitask Data Store** to warning (the default) or error. This diagnostic flags any case of a data 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.



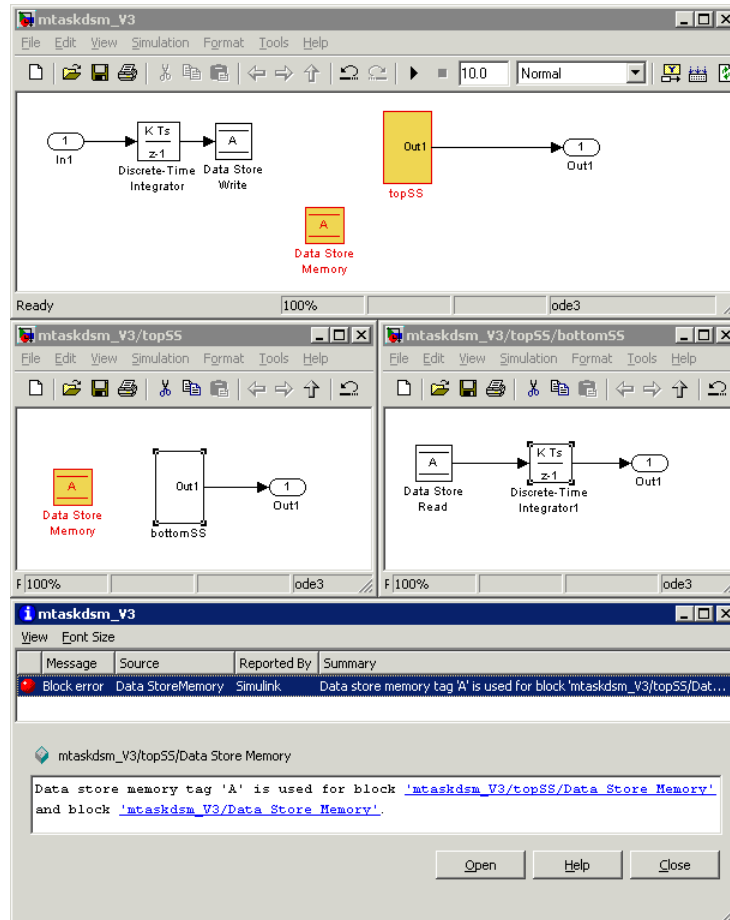
With this change, the data store write can continue to occur at the faster rate. This may be important if that data store must be read at that faster rate elsewhere in the model.

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”

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 inports and outports 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.

Managing Signals

- Chapter 50, “Working with Signals”
- Chapter 51, “Using Composite Signals”
- Chapter 52, “Working with Variable-Size Signals”

Working with Signals

- “Signal Basics” on page 50-2
- “Signal Types” on page 50-8
- “Virtual Signals” on page 50-11
- “Signal Values” on page 50-15
- “Signal Names and Labels” on page 50-19
- “Signal Label Propagation” on page 50-24
- “Signal Dimensions” on page 50-34
- “Determine Output Signal Dimensions” on page 50-36
- “Display Signal Sources and Destinations” on page 50-42
- “Signal Ranges” on page 50-45
- “Initialize Signals and Discrete States” on page 50-52
- “Test Points” on page 50-59
- “Display Signal Attributes” on page 50-62
- “Display Port Numbers When Addressing Errors” on page 50-67
- “Signal Groups” on page 50-69

Signal Basics

In this section...

- “About Signals” on page 50-2
- “Creating Signals” on page 50-3
- “Signal Line Styles” on page 50-4
- “Signal Properties” on page 50-5
- “Testing Signals” on page 50-6

About Signals

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)

Many blocks can accept or output signals of any data or numeric type and dimensionality. Other blocks impose restrictions on the attributes of the signals that they can handle.

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).

Note Simulink signals do not travel along the lines that connect blocks in the same way that electrical signals travel along a wire. This analogy is misleading because it suggests that a block diagram represents physical connections between blocks, which is not the case. Simulink signals are mathematical, not physical, entities, and the lines in a block diagram represent mathematical, not physical, relationships among blocks.

Creating Signals

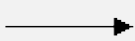

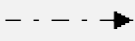
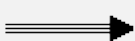
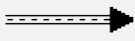


Create a signal by adding a source block to your model. For example, you can create a signal that varies sinusoidally with time, by dragging an instance of the Sine block from the Simulink Sources library into your model. See “Sources” for information about blocks that create signals in a model.

You can also use the Signal & Scope Manager to create signals in your model, without using blocks. See “Signal and Scope Manager” on page 16-23 for more information.

Signal Line Styles

A Simulink model can include many different types of signals. For details, see “Signal Types” on page 50-8.

Simulink uses a variety of line styles to display different types of signals in the model window. Assorted line styles help you to differentiate the signal types.

Signal Type	Line Style	Description
Scalar and Nonscalar		Simulink uses a thin, solid line to represent scalar and nonscalar signals.
Nonscalar (optional)		When you enable the Wide nonscalar lines option, Simulink uses a thick, solid line to represent nonscalar signals. For information about line display options, see “Display Signal Attributes” on page 50-62.
Control signal		Simulink uses a thin, dash-dot line to represent control signals.
Virtual Bus		Simulink uses a triple line with a solid core to represent virtual signal buses.
Nonvirtual Bus		Simulink uses a triple line with a dotted core to represent nonvirtual signal buses.
Array of Buses		Simulink uses a heavy triple line with a dotted core to represent array of bus signals.
Variable-Size		Simulink uses a solid wide line with a white dotted core to represent a variable-size signal.

Other than using the **Wide nonscalar lines** option to display nonscalar signals as thick, solid lines, you cannot customize or control the line style of signals. See “Wide Nonscalar Lines” on page 50-66 for more information about this option.

As you construct a block diagram, Simulink uses a thin, solid line to represent all signal types. After you update or start simulation of the block diagram, Simulink redraws the lines, using the specified line styles.

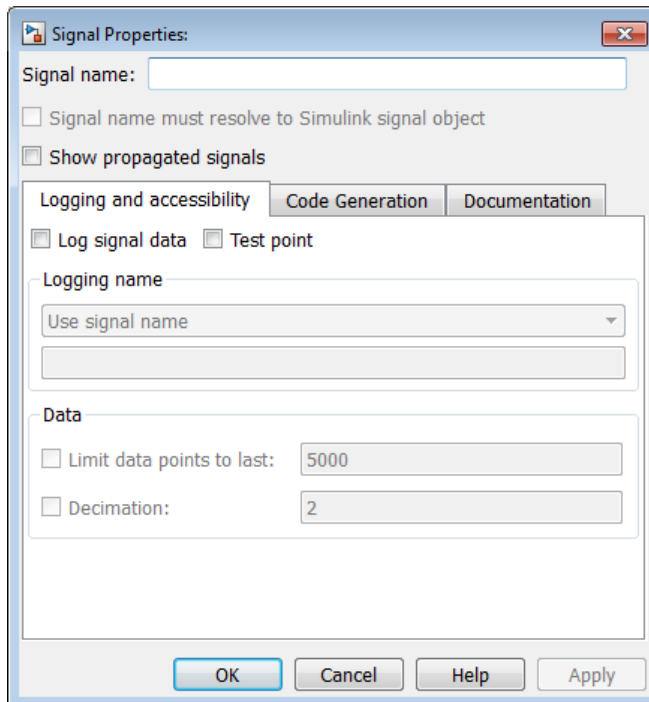
Signal Properties

Specifying Signal Properties

Use 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 open the Signal Properties dialog box, right-click a signal and choose **Properties**.



Displaying Signal Attributes in the Model

Displaying signal attributes in the model diagram can make the model easier to understand at a glance. For example, in the Simulink Editor, use the **Display > Signals & Ports** 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 50-62.

Display Signal Source and Destination

To highlight a signal and its source or destination blocks:

- Right-click a signal.
- In the context menu, select either **Highlight Signal to Source** or **Highlight Signal to Destination**.

For details, see “Display Signal Sources and Destinations” on page 50-42.

Testing Signals

You can perform the following kinds of tests on signals:

- “Minimum and Maximum Values” on page 50-6
- “Connection Validation” on page 50-7

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 “Signal Ranges” on page 50-45.

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. If any incompatibilities exist, Simulink reports an error and terminates the simulation.

To detect signal compatibility errors before running a simulation, update the diagram. In the Simulink Editor, select **Simulation > Update Diagram**.

Signal Groups

The Signal Builder block displays interchangeable groups of signal sources. Use the Signal Builder to create or edit groups of signals and to switch the groups into and out of a model.

Signal groups can greatly facilitate testing a model, especially when you use them in conjunction with Simulink Assertion blocks and the Model Coverage Tool in the Simulink Verification and Validation product.

For details, see “Signal Groups” on page 50-69.

Signal Types

In this section...
“Summary of Signal Types” on page 50-8
“Control Signals” on page 50-9
“Composite (Bus) Signals” on page 50-9

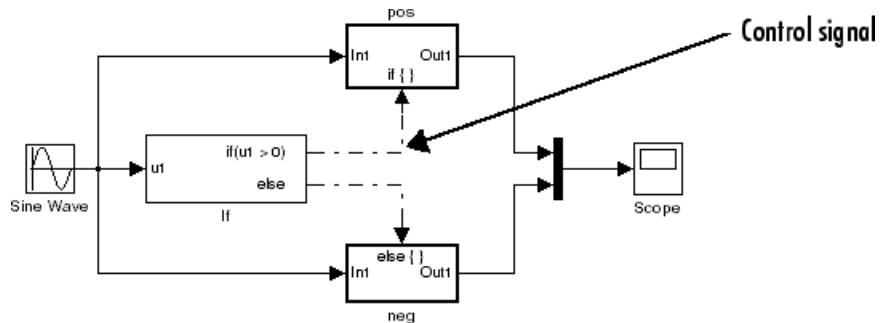
Summary of Signal Types

You can use many different kinds of signals in a model. The following table summarizes the signal types, and links to sections that describe each type in detail.

Signal Type	Description
Array of buses	An array whose elements are buses. See “Combine Buses into an Array of Buses” on page 51-81.
Bus (Composite)	A Simulink composite signal made up of other signals, optionally including other bus signals. See “Composite (Bus) Signals” on page 50-9.
Control	Signal used by one block to initiate execution of another block. For example, a signal that executes a function-call or action subsystem. For details, see “Control Signals” on page 50-9.
Nonvirtual	Signal that occupies its own storage. A nonvirtual bus reads inputs and writes outputs by accessing copies of the component signals.
Mux	A virtual vector created with a Mux block. See “Mux Signals” on page 50-11.
Variable-Size	Signal whose size (the number of elements in a dimension), in addition to its values, can change during a model simulation.
Virtual	Signal that represents another signal or set of signals. A virtual signal is used for graphical purposes and has no functional effect. See “Virtual Signals” on page 50-11.

Control Signals

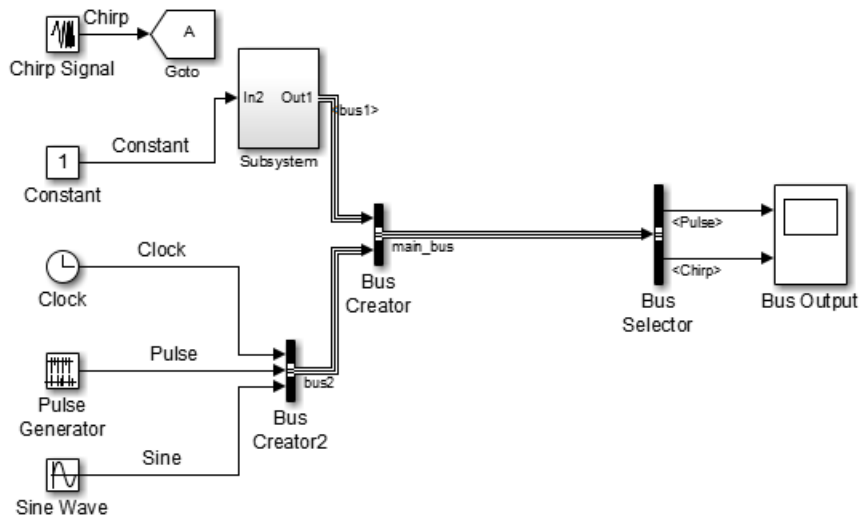
A *control signal* is a signal used by one block to initiate execution of another block. For example, a signal that executes a function-call or action subsystem is a control signal. When you update or simulate a block diagram, Simulink uses a dash-dot pattern to redraw lines representing the control signals.



Composite (Bus) Signals

You can group multiple signals into a hierarchical composite signal, called a *bus*, route the bus from block to block, and extract constituent signals from the bus where needed. When you have many parallel signals, buses can simplify the appearance of a model and help to clarify generated code. A bus can be either virtual or nonvirtual.

For example, if you open and simulate the `Bus Signal` example model, the `bus1`, `bus2`, and `main_bus` signals are bus signals. These virtual bus signals use the triple line style.



For details, see “Composite Signals” on page 51-3.

Virtual Signals

In this section...
“About Virtual Signals” on page 50-11
“Mux Signals” on page 50-11

About Virtual Signals

A *virtual signal* is a signal that graphically represents other signals or parts of other signals. Virtual signals are purely graphical entities; they have no mathematical or physical significance. 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 Bus Creator, can generate either virtual or nonvirtual signals.

The nonvirtual components of a virtual signal are called *regions*. A virtual signal can contain the same region more than once. For example, if the same nonvirtual signal is connected to two input ports of a Mux block, the block outputs a virtual signal that has two regions. The regions behave as they would if they had originated in two different nonvirtual signals, even though the resulting behavior duplicates information.

Bus signals can also be virtual or nonvirtual. For details, see “Types of Simulink Buses” on page 51-7.

Mux Signals

A Simulink *mux* is a virtual signal that graphically combines two or more scalar or vector signals into one signal line. A Simulink mux is not a hardware multiplexer, which combines multiple data streams into a single channel. A Simulink mux does not combine signals in any functional sense: it exists only virtually, and its only purpose is to simplify the visual appearance of a model. Using a mux has no effect on simulation or generated code.

You can use a mux anywhere that you could use an ordinary (contiguous) vector. For example, you can perform calculations on a mux. The computation affects each constituent value in the mux just as if the values existed in a contiguous vector, and the result is a contiguous vector, not a mux. Using a

mux to perform computations on multiple vectors avoids the overhead of copying the separate values to contiguous storage.

The Simulink documentation refers, sometimes interchangeably, to “muxes”, “vectors”, and “wide signals”, and all three terms appear in Simulink GUI labels and API names. This terminology can be confusing, because most vector signals, which are also called wide signals, are nonvirtual and hence are not muxes. To avoid confusion, reserve the term “mux” to refer specifically to a virtual vector.

A mux is a *virtual* vector signal. The constituent signals of a mux retain their separate existence in every way, except visually. You can also combine scalar and vector signals into a *nonvirtual* vector signal, by using a Vector Concatenate block. The signal output by a Vector Concatenate block is an ordinary contiguous vector, inheriting no special properties from the fact that it was created from separate signals.

To create a composite signal whose constituent signals retain their identities and can have different data types, use a Bus Creator block rather than a Mux block. For details, see “Composite Signals” on page 51-3. Although you can use a Mux block to create a composite signal in some cases, MathWorks discourages this practice. See “Prevent Bus and Mux Mixtures” on page 51-100 for more information.

Using Muxes

The Signal Routing library provides two virtual blocks for implementing muxes:

Mux

Combine several input signals into a mux (virtual vector) signal

Demux

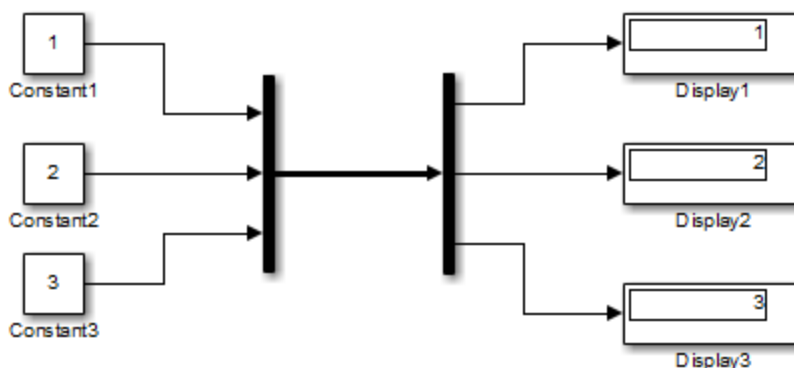
Extract and output the values in a mux (virtual vector) signal

To implement a mux signal:

- 1 Select a Mux and Demux block from the Signal Routing library.
- 2 Set the Mux block **Number of inputs** and the Demux block **Number of outputs** block parameters to the desired values.

- 3 Connect the Mux, Demux, and other blocks as needed to implement the desired signal.

The next figure shows three signals that are input to a Mux block, transmitted as a mux signal to a Demux block, and output as separate signals.



The Mux and Demux blocks are the left and right vertical bars, respectively. To reduce visual complexity, neither block displays a name. In this example, the line connecting the blocks, representing the mux signal, is wide because the model has been built with **Display > Signals & Ports > Wide Nonscalar Lines** option enabled. See “Display Signal Attributes” on page 50-62 for details.

Signals input to a Mux block can be any combination of scalars, vectors, and muxes. The signals in the output mux appear in the order in which they were input to the Mux block. You can use multiple Mux blocks to create a mux in several stages, but the result is flat, not hierarchical, just as if the constituent signals had been combined using a single mux block.

The values in all signals input to a Mux block must have the same data type.

If a Demux block attempts to output more values than exist in the input signal, an error occurs. A Demux block can output fewer values than exist in the input mux, and can group the values it outputs into different scalars and vectors than were input to the Mux block. However, the Demux block cannot rearrange the order of those values. For details, see Demux.

Note MathWorks discourages using Mux and Demux blocks to create and access buses under any circumstances. See “Prevent Bus and Mux Mixtures” on page 51-100 for details.

Signal Values

In this section...

“Signal Data Types” on page 50-15

“Signal Dimensions, Size, and Width” on page 50-15

“Complex Signals” on page 50-16

“Initializing Signal Values” on page 50-16

“Viewing Signal Values” on page 50-17

“Displaying Signal Values in Model Diagrams” on page 50-17

“Exporting Signal Data” on page 50-18

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 “Data Types” on page 46-2 for more information.

Signal Dimensions, Size, and Width

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.

The size of a signal refers to the number of elements that a signal contains. The size of a matrix (2-D) signal is generally expressed as M-by-N, where M is the number of columns and N is the number of rows making up the signal. The size of a vector signal is referred to as the width of the signal.

For more information, see “Signal Dimensions” on page 50-34.

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.

Initializing 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	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 Output, 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 50-52.

Viewing 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 “View Simulation Results” on page 16-2. For detailed information about:

- Blocks that you can use to display signals in a model, see “Sinks”
- Signal viewers, see “Signal Viewer Tasks” on page 16-6
- The Signal & Scope Manager, see “Signal and Scope Manager” on page 16-23
- Test points, which are signals that Simulink guarantees to be observable when using a Floating Scope block in a model, see “Test Points” on page 50-59.

Displaying 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” on page 50-17
- “Display Signal Value After Simulation” on page 50-18

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, select **Display > Data Display in Simulation**.
- 2** From the submenu, select either **Show Value Labels When Hovering** or **Show Value Labels When Clicked**.
- 3** To change display options, use the **Options** submenu.

For details, see “Display Port Values” on page 23-30.

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**.

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 48-4.

Signal Names and Labels

In this section...
“Signal Names” on page 50-19
“Signal Labels” on page 50-22

Signal Names

You can name a signal. The signal name appears below a signal, displayed as a signal label (for details, see “Signal Labels” on page 50-22).

Choosing a Signal Name

The syntactic requirements for a signal name depend on how the name is used. The most common cases are:

- The signal is named so that it can be resolved to a `Simulink.Signal` object. (See `Simulink.Signal`.) The signal name must then 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`.
- 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 48-21.) Such a signal name can contain space and newline characters. These can improve readability but sometimes require special handling techniques, as described in “Handling Spaces and Newlines in Logged Names” on page 48-68.
- The signal name exists only to clarify the diagram, and has no computational significance. Such a signal name can contain anything and never needs 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 may require going back and 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.

To name a signal, use one of the following approaches:

- “Assign a name from the Simulink block diagram” on page 50-20
- “Assign a Name in the Signal Properties Dialog Box” on page 50-20
- “Assign a name from the MATLAB Command Window” on page 50-21

Assign a name from the Simulink block diagram

- 1** Double-click a signal.

An edit box appears next to the signal.

Note When you create a signal label, take care to double-click the line. If you click in an unoccupied area close to the line, you will create a model annotation instead.

- 2** Enter the desired name, then click somewhere outside the edit box.

The signal now has the specified name. A label that shows the name appears at the location where you entered it.

For a named multibranch signal, you can put a duplicate label on any branch of the signal by double-clicking the branch.

Assign a Name in the Signal Properties Dialog Box

- 1** Right-click a signal and from the context menu, choose **Properties**.

A Signal Properties dialog box opens.

- 2 In the **Signal Name** field, enter a name. Click **OK** or **Apply**.

A label showing the name appears on every branch of the signal.

Assign a name from the MATLAB Command Window

You can also use the MATLAB Command Window to set the name parameter of the port or line that represents the signal:

- 1 Select the source block for the line or port.
- 2 In the MATLAB Command Window, type code similar to the following:

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

Change the Name of a Signal

To change the name of a signal:

- 1 Double-click a signal line.
An editing box opens around the label.
- 2 Change the text and then click away from the label.
All labels update to reflect the change.

Alternatively, you can edit the name in the **Signal Properties > Signal name** field.

Remove a Signal Name

To remove a signal name, delete all characters in the name, in any label on the signal or in the **Signal Properties > Signal name** field.

To delete a label without deleting the signal name, click near the edge of the label to select its surrounding box, then press **Delete**.

Signal Labels

A signal label is text that appears next to the line representing a signal. The signal label displays the signal name. Simulink creates a label for a signal when you assign it a name. For details, see “Signal Names” on page 50-19 .

Move Signal Labels

Labels can appear above or below horizontal lines or line segments, and left or right of vertical lines or line segments. Labels can appear at either end, at the center, or in any combination of these locations.

To move a signal label, drag the label to a new location on the line. When you release the mouse button, the label fixes its position near the line. You cannot drag a label away from its signal, but only to a different location adjacent to the signal.

Edit Signal Labels

To edit an existing signal label, select it:

- To replace the label, click the label, double-click or drag the cursor to select the entire label, then enter the new label.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

Change the Font of a Signal Label

To change the font of a signal label:

- 1** Select the signal.
- 2** Click **Diagram > Format > Font Style**.
- 3** Select a font from the **Select Font** dialog box.

Copy Signal Labels

To copy a signal label, hold down the **Ctrl** key while dragging the label to another location on the line. When you release the mouse button, the label appears in both the original and the new locations.

Delete Signal Labels

To delete all occurrences of a signal label, delete all the characters in the label. When you click outside the label, the labels are deleted. To delete a single occurrence of the label, hold down the **Shift** key while you select the label, then press the **Delete** or **Backspace** key.

Show Propagated Signal Labels

You can have Simulink pass a signal name to downstream connection blocks. Examples of connection blocks that support signal label propagation include the Subsystem and Signal Specification blocks.

For details, see “Signal Label Propagation” on page 50-24.

Signal Label Propagation

In this section...

“Propagated Signal Labels” on page 50-24

“Blocks That Support Signal Label Propagation” on page 50-25

“Display Propagated Signal Labels” on page 50-25

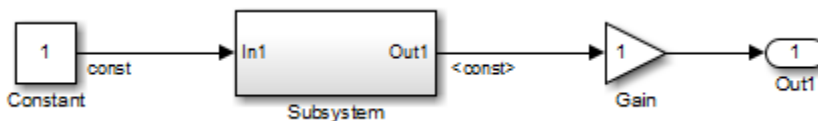
“How Simulink Propagates Signal Labels” on page 50-26

Propagated Signal Labels

When you enable the display of signal label propagation for output signals of the blocks listed in “Blocks That Support Signal Label Propagation” on page 50-25:

- 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 50-26.

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 root inports)
- Signal Specification
- Subsystem (through subsystem Inport and Outport blocks)
- Trigger
- Two-Way Connection (a Simscape block)

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 Signal Hierarchy Viewer.

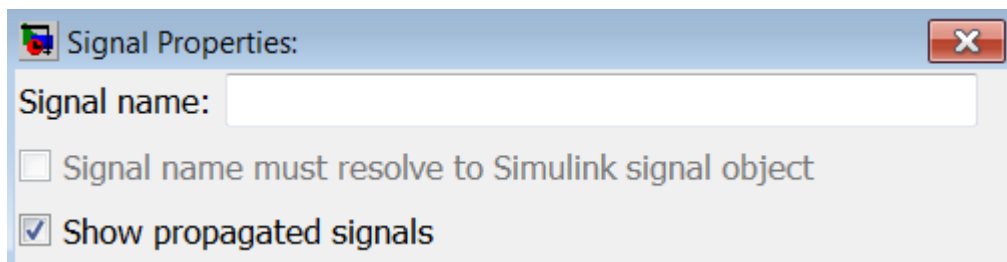
The Signal Properties dialog box for a signal indicates whether that signal supports signal label propagation. The **Show propagated signals** parameter is available only for blocks that support signal label propagation. For details, see “Display Propagated Signal Labels” on page 50-25.

Display Propagated Signal Labels

To display a propagated signal label:

- 1 Set **Model Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** to error.

- 2 Right-click the signal for which you want to display a propagated signal label and select **Properties**.
- 3 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.

If a signal already has a label, then an *alternative* approach for displaying a propagated signal label is:

- 1 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.

How Simulink Propagates Signal Labels

Understanding how Simulink propagates signal labels helps you to:

- Anticipate the scope of the signal label propagation, from source to final destination
- Configure your model to display signal labels for the signals that you want

For output signals from supported blocks, you can choose to have Simulink display propagated signal labels. For a list of supported blocks, see “Blocks That Support Signal Label Propagation” on page 50-25.

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:

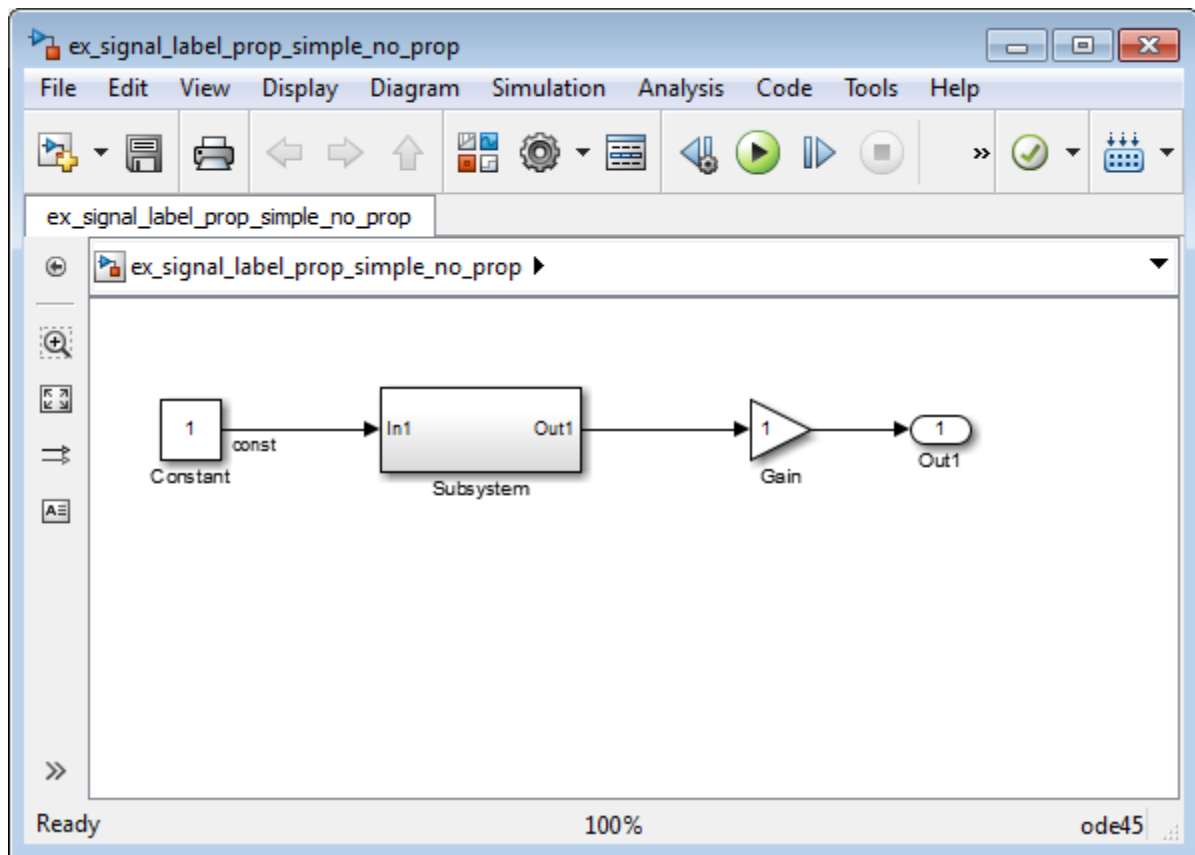
- “Processing for Referenced Models” on page 50-31
- “Processing for Variants and Configurable Subsystems” on page 50-33

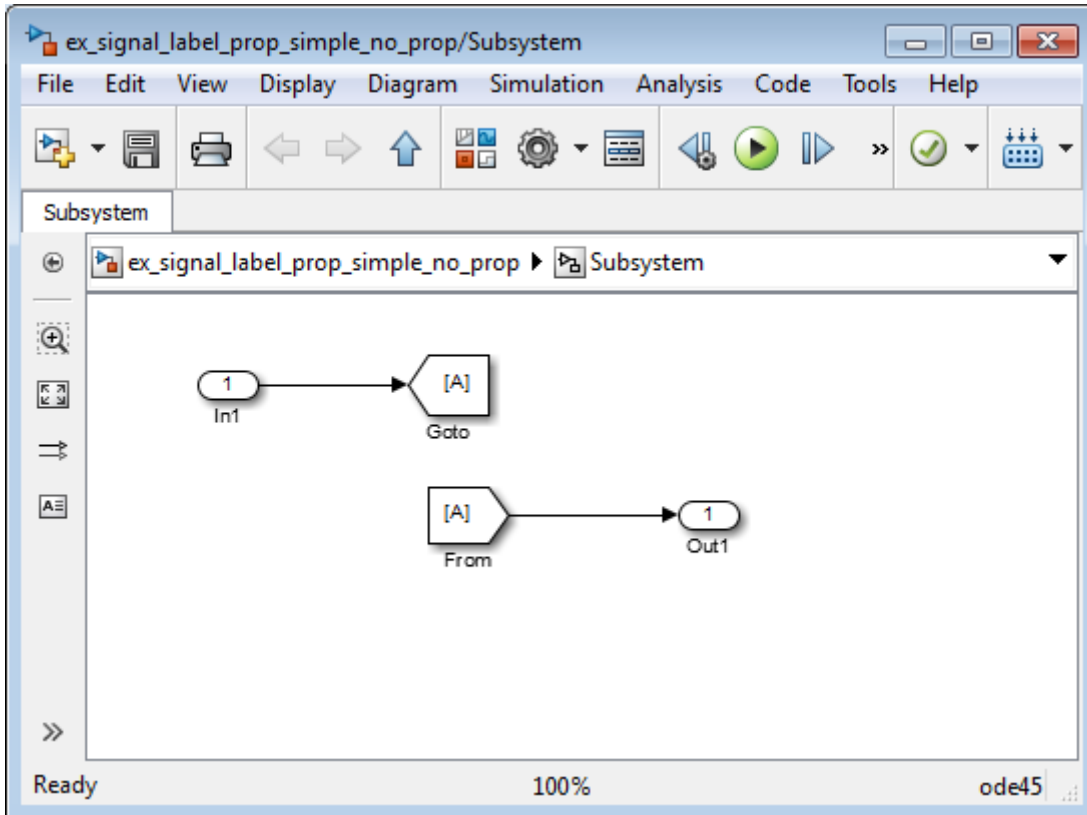
General Signal Label Propagation Processing

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.

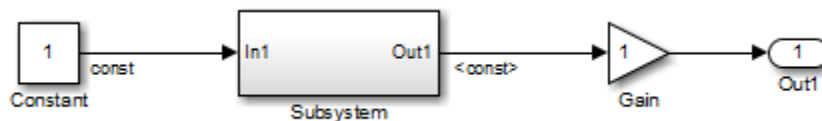
For example, in the following model, suppose that you enable signal label propagation for the output signal for the Subsystem block (that is, the signal connected to the Out1 port).



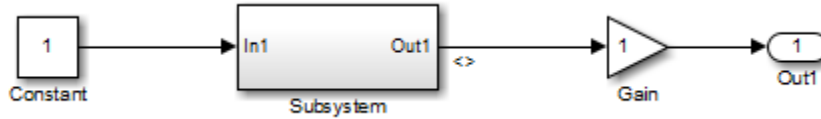


Simulink checks inside the subsystem, checks upstream from the From and GoTo blocks (which support signal label propagation and do not have a name), and then checks farther upstream, to the Constant block, which does not support signal label propagation.

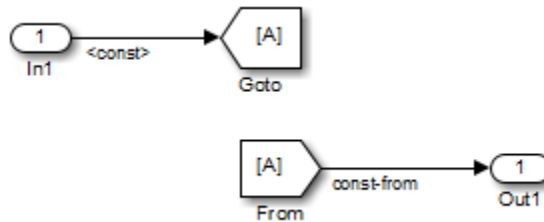
Simulink uses the signal name of the Constant block output signal, const. The propagated signal label for the Subsystem output signal is <const>.



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 (<>).



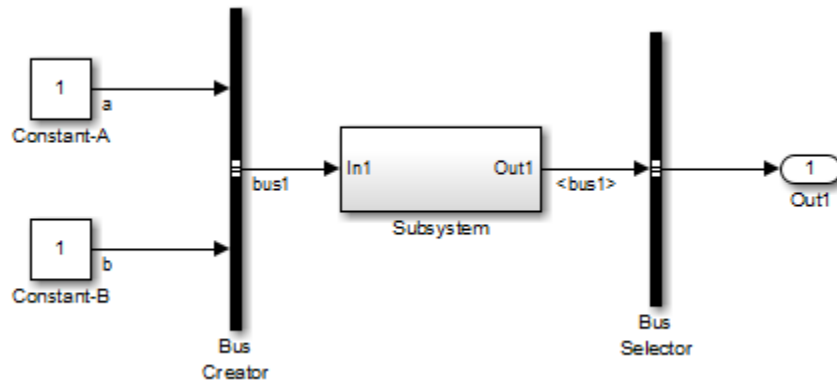
Suppose that in the Subsystem block you enable signal label propagation for the output signal from the In1 block, and you use the Signal Properties dialog box to specify the signal name `const - from` for the output signal of the From block, as shown below.



The propagated signal label for the Subsystem 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).



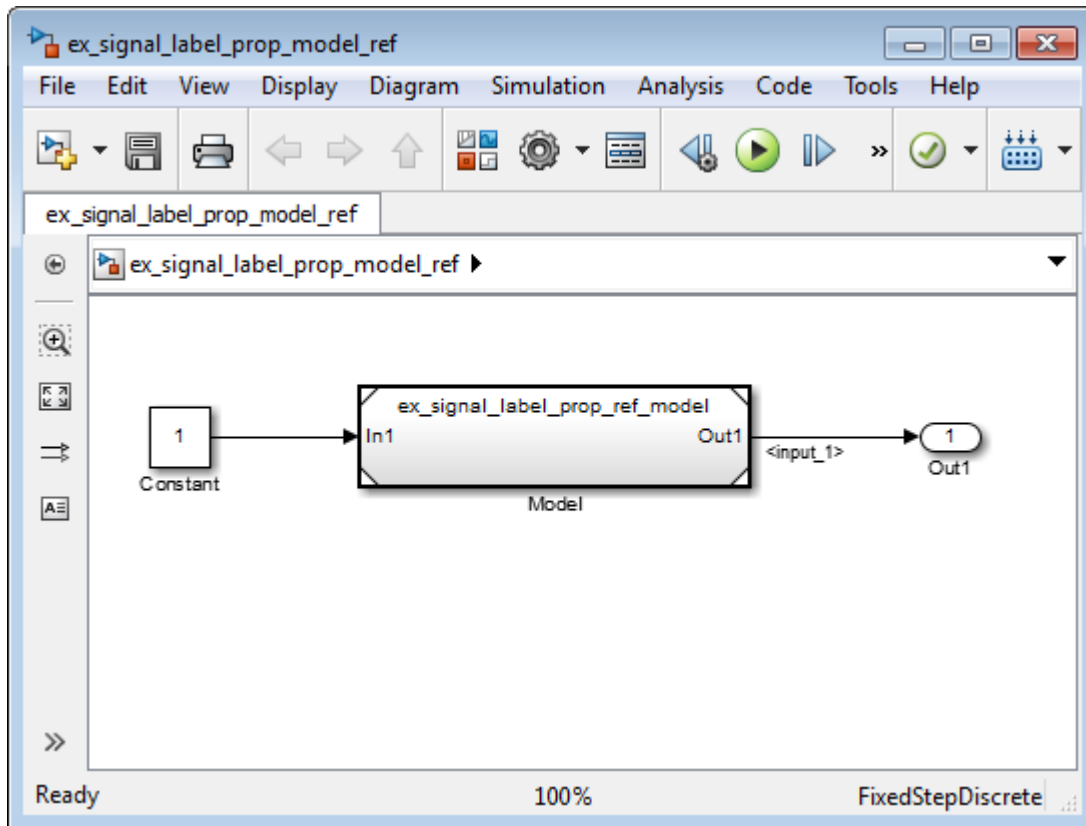
Processing for Referenced Models

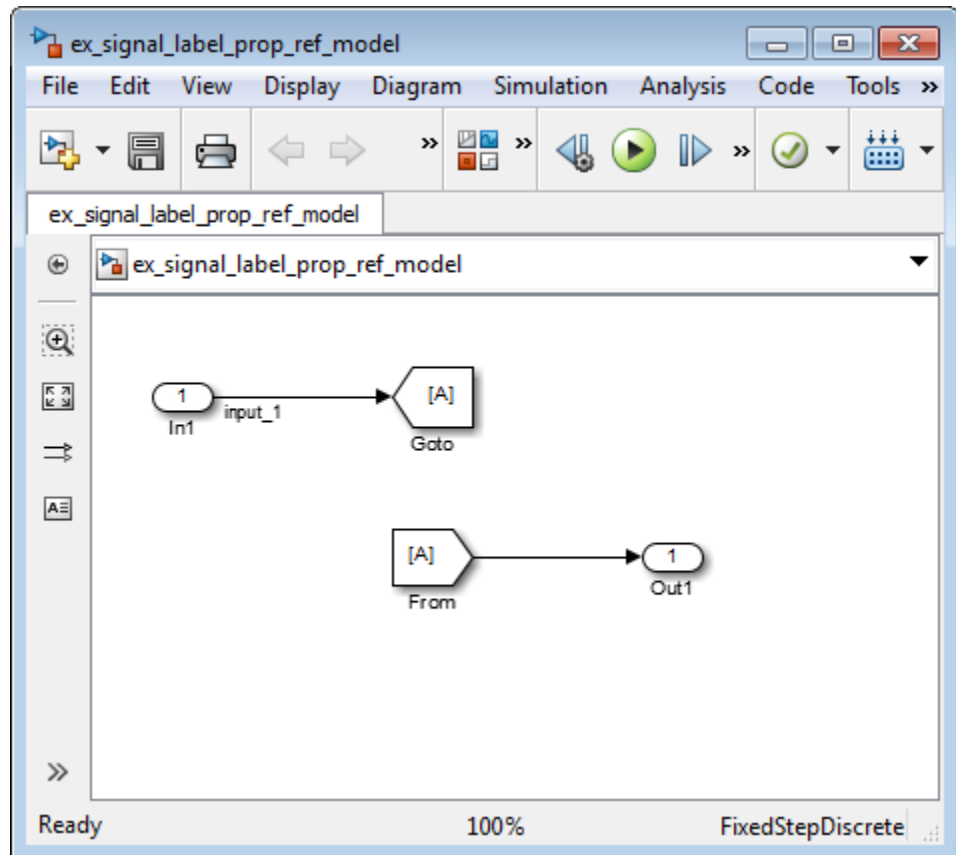
To enable signal label propagation for referenced models, in addition to the steps described in “Display Propagated Signal Labels” on page 50-25, enable the **Model Configuration Parameters > Model Referencing > Propagate all signal labels out of the model** parameter.

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 model reference variants 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.

Signal Dimensions

In this section...

“About Signal Dimensions” on page 50-34

“Simulink Blocks that Support Multidimensional Signals” on page 50-35

About Signal Dimensions

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. To determine the signal dimensionality of a particular block, see the block documentation. See “Determine Output Signal Dimensions” on page 50-36 for information on what determines the dimensions of output signals for blocks that can output nonscalar 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 52-2.

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 Simulink editor, from the **Help** menu, click **Simulink > Block Data Types & Code Generation Support > All Tables**.

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 **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 Output Signal Dimensions

In this section...

“About Signal Dimensions” on page 50-36

“Determining the Output Dimensions of Source Blocks” on page 50-36

“Determining the Output Dimensions of Nonsource Blocks” on page 50-37

“Signal and Parameter Dimension Rules” on page 50-37

“Scalar Expansion of Inputs and Parameters” on page 50-38

About Signal Dimensions

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.

Determining 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’s **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 a source block’s output value parameter(s) and **Interpret vector parameters as 1-D** parameter 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’s 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.

Determining 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 50-37).

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 50-38).

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 50-38).
- 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 a block's inputs and parameters.

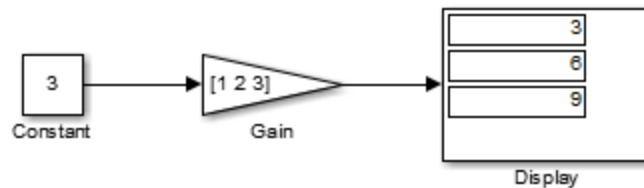
Scalar expansion of inputs refers to the expansion of scalar inputs to match the dimensions of other nonscalar inputs or nonscalar parameters. When the input to a block is a mix of scalar and nonscalar signals, Simulink expands the scalar inputs into nonscalar signals having the same dimensions as the 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 50-40.
Nonscalar	Scalar	Scalar parameter expanded to match number of elements of input. See "Nonscalar Input and Scalar Parameter" on page 50-40.
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 50-40.

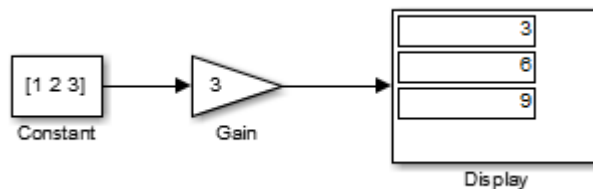
Scalar Input and Nonscalar Parameter

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.



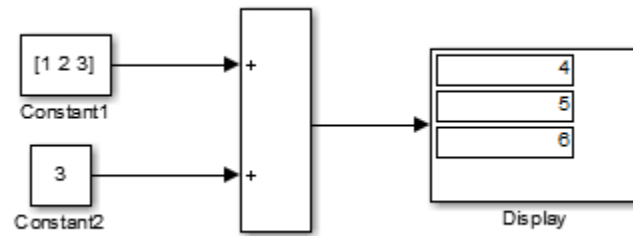
Nonscalar Input and Scalar Parameter

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].



Display Signal Sources and Destinations

In this section...
“About Signal Highlighting” on page 50-42
“Highlighting Signal Sources” on page 50-42
“Highlighting Signal Destinations” on page 50-43
“Removing Highlighting” on page 50-44
“Resolving Incomplete Highlighting to Library Blocks” on page 50-44

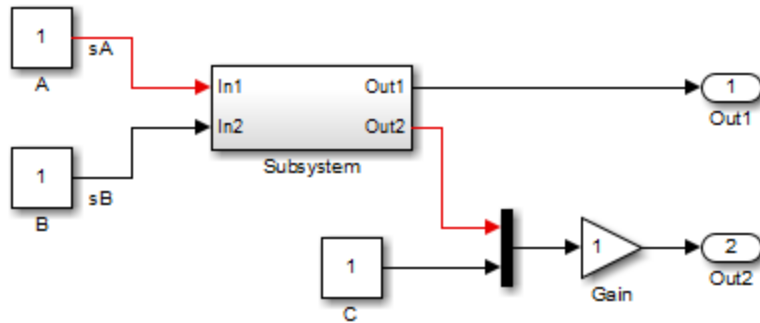
About Signal Highlighting

You can highlight a signal and its source or destination block(s), then remove the highlighting once it has served its purpose. Signal highlighting crosses subsystem boundaries, allowing you to trace a signal across multiple subsystem levels. Highlighting does not cross the boundary into or out of a referenced model. If a signal is composite, all source or destination blocks are highlighted. (See “Composite Signals” on page 51-3.)

Highlighting Signal Sources

To display the source block(s) of a signal, select the **Highlight Signal to Source** option from the context menu for the signal. This option highlights:

- All branches of the signal anywhere in the model
- All virtual blocks through which the signal passes
- The nonvirtual block(s) that write the value of the signal

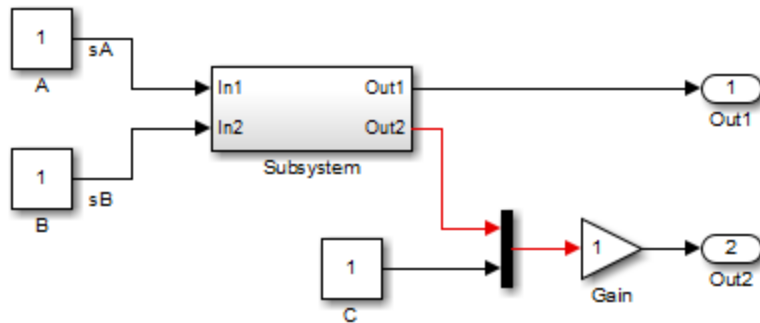


Highlighting Signal Destinations

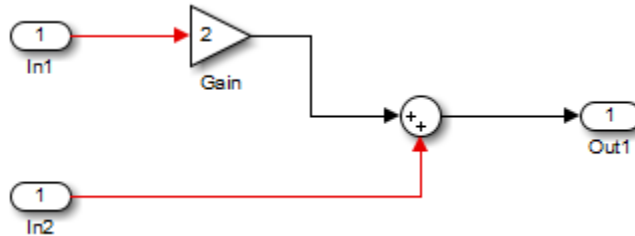
To display the destination blocks of a signal, select the **Highlight Signal to Destination** option from the context menu for the signal. This option highlights:

- All branches of the signal anywhere in the model
- All virtual blocks through which the signal passes
- The nonvirtual block(s) 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 signal highlights the Gain block as the destination block.



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 import blocks.



Removing Highlighting

To remove all highlighting, select **Remove Highlighting** from the model's context menu, or select **Display > Remove Highlighting**.

Resolving 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, first update the diagram (for example, by pressing **Ctrl+D**). The update of the diagram resolves all library references and displays the complete path to a destination block or from a source block.

Signal Ranges

In this section...
“About Signal Ranges” on page 50-45
“Blocks That Allow Signal Range Specification” on page 50-45
“Specifying Ranges for Signals” on page 50-46
“Checking for Signal Range Errors” on page 50-47

About Signal Ranges

Many Simulink blocks allow you to specify a range of valid values for their output signals. Simulink provides a diagnostic that you can enable to detect when blocks generate signals that exceed their specified ranges during simulation. See the sections that follow for more information.

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

Specifying Ranges for Signals

In general, use the **Output minimum** and **Output maximum** parameters that appear on a block parameter dialog box 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 50-45 for a list of applicable blocks.

When specifying minimum and maximum values that constitute a range, enter only expressions that evaluate to a scalar, real number with `double` data type. The default values for the minimum and maximum are `[]` (unspecified). The scalar values that you specify are subject to expansion, for example, when the block inputs are nonscalar or bus signals (see “Scalar Expansion of Inputs and Parameters” on page 50-38).

Note You cannot specify the minimum or maximum value as NaN, inf, or -inf.

Specifying Ranges for Complex Numbers

When you specify an **Output minimum** and/or **Output maximum** for a signal that is a complex number, 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 $(\text{sqrt}(a^2+b^2))$

Checking for 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 “Specifying Ranges for Signals” on page 50-46) and the block data type. That is, Simulink performs the following check:

$$\text{DataTypeMin} \quad \text{MinValue} \quad \text{VALUE} \quad \text{MaxValue} \quad \text{DataTypeMax}$$

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.

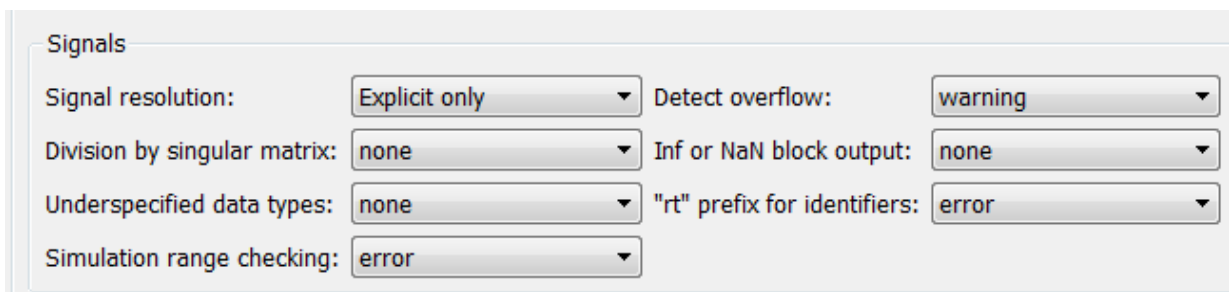
Enabling Simulation Range Checking

To enable the **Simulation range checking** diagnostic:

- 1 In your model window, select **Simulation > Model Configuration Parameters**.

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.

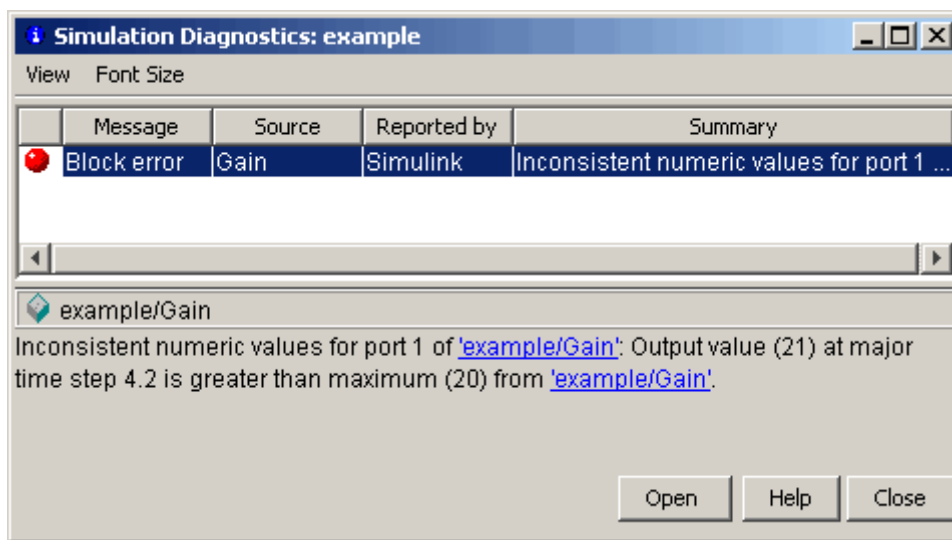
See “Simulation range checking” for more information.

Simulating Models with Simulation Range Checking

To check for signal range errors or warnings:

- 1 Enable the **Simulation range checking** diagnostic for your model (see “Enabling Simulation Range Checking” on page 50-48).
- 2 In your model window, select **Simulation > 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 displays an error message:



Otherwise, if a signal exceeds its specified range when the **Simulation range checking** diagnostic specifies warning, Simulink displays a warning message in the MATLAB Command Window:

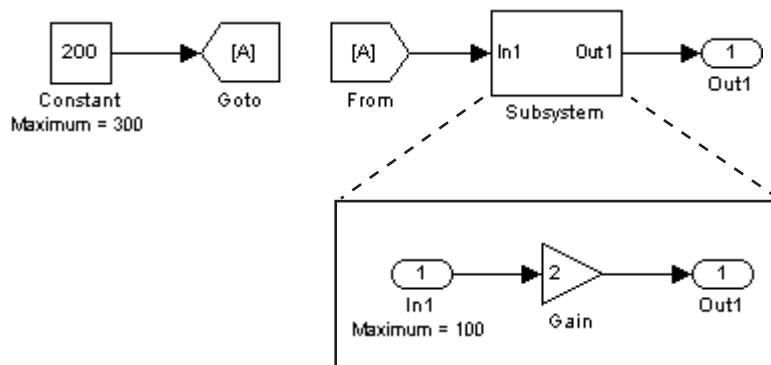
```
Warning: Inconsistent numeric values for port 1
of 'example/Gain': Output value (21) at major
time step 4.2 is greater than maximum (20) from
'example/Gain'.
```

Each message identifies the block whose output signal exceeds its specified range, and the time step at which this violation occurs.

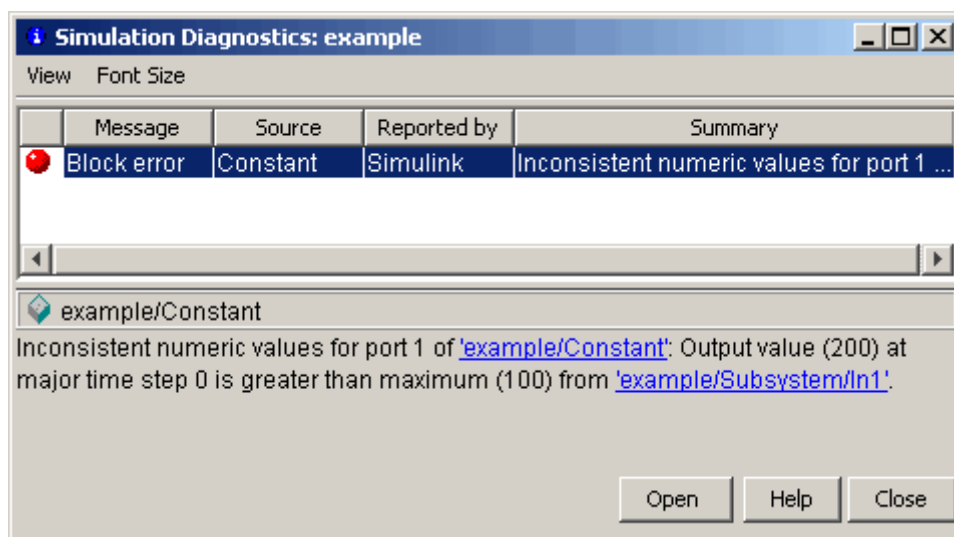
Signal Range Propagation for Virtual Blocks

Some virtual blocks (see “Virtual Blocks” on page 23-2) 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 displays the following error message:



Initialize Signals and Discrete States

In this section...

“About Initialization” on page 50-52

“Using Block Parameters to Initialize Signals and Discrete States” on page 50-53

“Using Signal Objects to Initialize Signals and Discrete States” on page 50-53

“Using Signal Objects to Tune Initial Values” on page 50-54

“Example: Using a Signal Object to Initialize a Subsystem Output” on page 50-56

“Initialization Behavior Summary for Signal Objects” on page 50-57

About Initialization

Note For information about initializing bus signals, see “Specify Initial Conditions for Bus Signals” on page 51-69.

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.

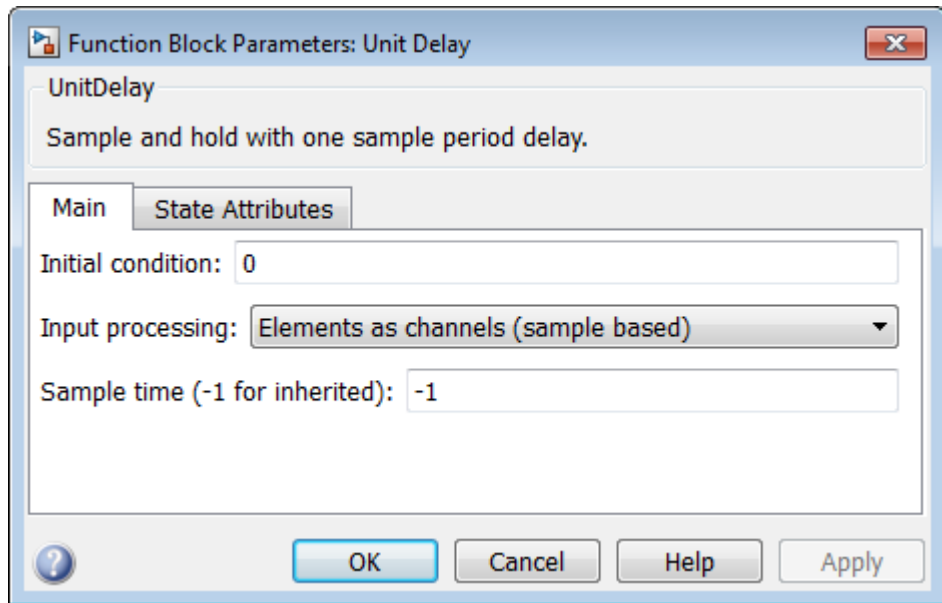
When you specify a signal object for signal or discrete state initialization, or a variable as the value of a block parameter, Simulink resolves the name that you specify to an appropriate object or variable, as described in “Symbol Resolution” on page 4-83.

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.



Using Signal Objects to Initialize Signals and Discrete States

To use a signal object to specify an initial value:

- 1 Create the signal object in the MATLAB workspace, as explained in “Data Objects” on page 46-35.

The name of the signal object must be the same as the name of the signal or discrete state that the object is initializing.

Note Consider also setting the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. This setting ensures consistency between signal objects in the MATLAB workspace and the signals that appear in your model.

- 2 Set the signal object’s storage class to a value other than 'Auto' or 'SimulinkGlobal'.
- 3 Set the signal object’s `Initial` value property to the initial value of the signal or state. For details on what you can specify, see the description of `Simulink.Signal`.

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 null (`[]`) or to the same value as the initial value of the signal object. If you set the parameter value to null, 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.

Using Signal Objects to Tune Initial Values

Simulink allows you to use signal objects as an alternative to parameter objects (see) 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’s 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.

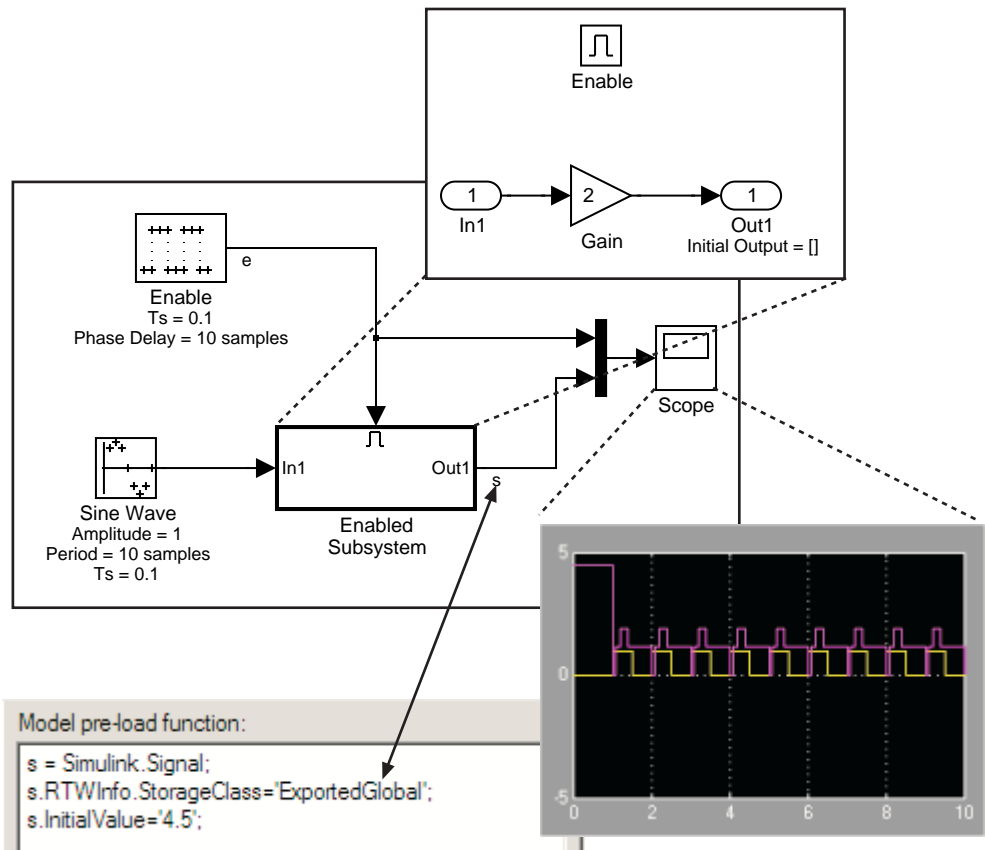
For example, suppose you want to tune the initial value of a Memory block state named `M1`. To do this, you might create a signal object named

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's 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**).

Note To be tunable via a signal object, a signal or state's corresponding initial condition parameter must be tunable, e.g., the inline parameter optimization for the model containing the signal or state must be off or the parameter must be declared tunable in the Model Parameter Configuration dialog box. For more information, see "Tunable Parameters" on page 3-9 and "Tunable Parameters" on page 24-14.

Example: Using a Signal Object to Initialize a Subsystem Output

The following example shows a signal object specifying the initial output of an enabled subsystem.

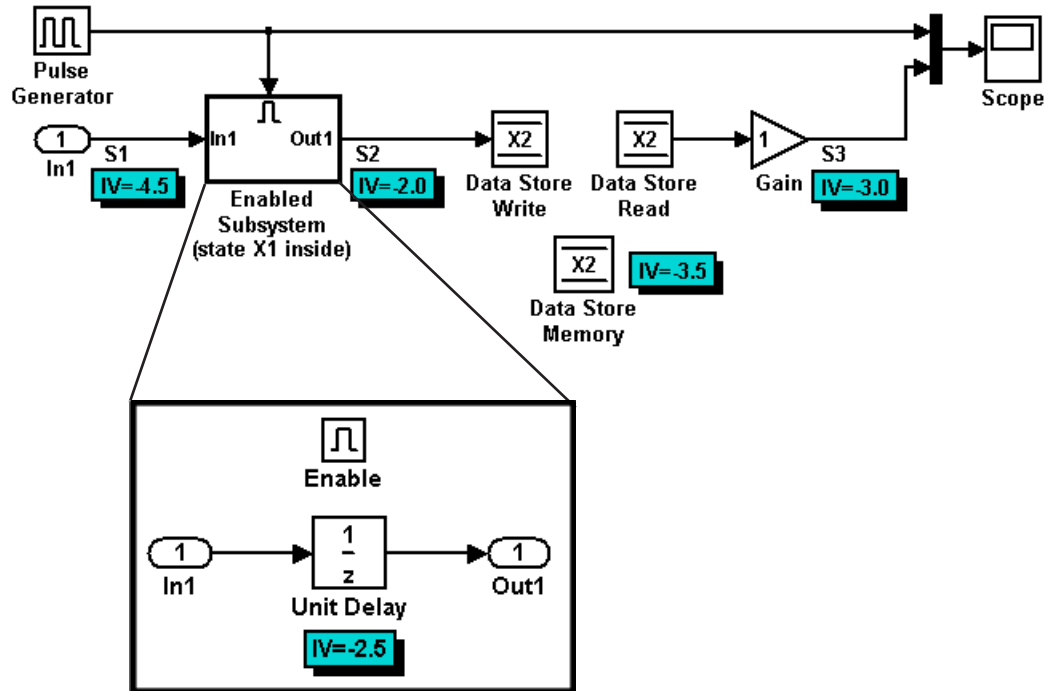


Signal `s` is initialized to 4.5. To avoid a consistency error, the initial value of the enabled subsystem's Output block must be `[]` or 4.5.

If you need a signal object and its initial value setting to persist across Simulink sessions, see "Creating Persistent Data Objects" on page 46-45.

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 inport	<ul style="list-style-type: none"> • Initialized to <code>S1.InitialValue</code>. • 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	<ul style="list-style-type: none"> • Initialized to <code>X1.InitialValue</code>. • Simulink checks whether <code>X1.InitialValue</code> matches the initial condition specified for the block and displays an error if a mismatch occurs.

Signal or Discrete State	Description	Behavior
	state that has an initial condition	<ul style="list-style-type: none"> • At first write, the output equals <code>X1.InitialValue</code> and the state equals <code>S1</code>. • At each time step after the first write, the output equals the state and the state is updated to equal <code>S1</code>. • If the block is inside an enabled subsystem, you can use the initial value as a reset value if the subsystem's Enable block parameter States when enabling is set to reset.
X2	Data Store Memory block	<ul style="list-style-type: none"> • Data type work (DWork) vector initialized to <code>X2.InitialValue</code>. For information on work vectors, see "DWork Vector Basics". • Simulink checks whether <code>X2.InitialValue</code> matches the initial condition specified for the block, and displays an error if a mismatch occurs. • Data Store Write blocks overwrite the value.
S2	Output of an enabled subsystem	<ul style="list-style-type: none"> • Initialized to <code>S2.InitialValue</code> or the value of the Output 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's Enable block parameter States when enabling or Output block parameter Output when disabled is set to reset.
S3	Persistent signals	<ul style="list-style-type: none"> • Initialized to <code>S3.InitialValue</code>. • The output value is reset by the block at each time step. • Affects code generation only. For simulation, setting the initial value for <code>S3</code> is irrelevant because the values are overwritten at the model's simulation start time.

Test Points

In this section...

“What Is a Test Point?” on page 50-59

“Designating a Signal as a Test Point” on page 50-59

“Displaying Test Point Indicators” on page 50-60

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”) 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 Loops” on page 3-39).

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 “Signals with Test Points”.

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 48-21.

Designating a Signal as a Test Point

Use one of the following ways to designate a signal as a test point:

- Open the signal’s **Signal Properties** dialog and check **Test Point** in the **Logging and accessibility** section.

- Resolve the signal to a base workspace `Simulink.Signal` object whose storage class is `SimulinkGlobal`.

The second method is more convenient when you want to control test pointing without having to alter the model.

Model Referencing Limitation

Simulink might not log all signals configured for signal logging in a referenced model, if *all* of these conditions exist:

- The referenced model sets the **Model Configuration Parameters > Data Import/Export > Signal logging format** parameter to `ModelDataLogs`.
- The referenced model uses a library and you make a change that affects the set of test points in a library, or that changes the set of models that a library references.

To ensure proper signal logging for the referenced model:

- 1 Open the referenced model.
- 2 Perform an update diagram on the referenced model (for example, by pressing **Ctrl+D**).
- 3 Save the referenced model.

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 48-21 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, select or clear **Display > Signals & Ports > Testpoint & Logging Indicators**.

Display Signal Attributes

In this section...

- “Ports & Signals Menu” on page 50-62
- “Port Data Types” on page 50-63
- “Design Ranges” on page 50-63
- “Signal Dimensions” on page 50-64
- “Signal to Object Resolution Indicator” on page 50-65
- “Wide Nonscalar Lines” on page 50-66

Ports & Signals Menu

The **Display > Signals & Ports** submenu of the Simulink Editor offers the following options for displaying signal properties on the block diagram:

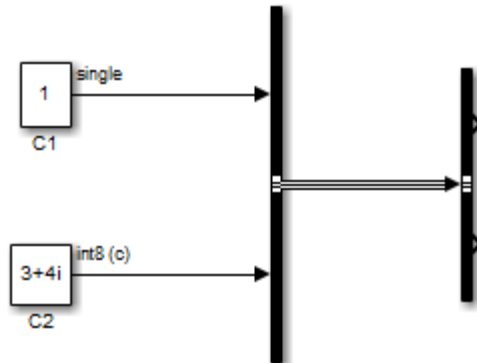
- Linearization Indicators
- Port Data Types (See “Port Data Types” on page 50-63)
- Design Ranges (See “Design Ranges” on page 50-63)
- Signal Dimensions (See “Signal Dimensions” on page 50-64)
- Storage Class
- Testpoint/Logging Indicators
- Signal Resolution Indicators (See “Signal to Object Resolution Indicator” on page 50-65)
- Viewer Indicators
- Wide Nonscalar Lines (See “Wide Nonscalar Lines” on page 50-66)

In addition, you can display sample time information. If you first select **Display > Sample Time**, a submenu provides the choices of **Colors**, **Annotations** and **All**. 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 **Annotations** option provides black codes on the signal lines which indicate the type of sample time. **All** causes both the colors and the

annotations to 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 turned 'on', color codes also appear in the legend. The same is true if **Annotations** are turned 'on'.

Port Data Types

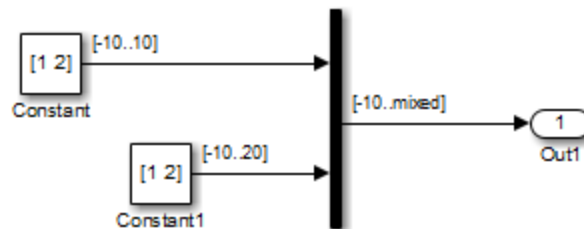
Displays the data type of a signal next to the output port that emits the signal.



The notation (c) following the data type of a signal indicates that the signal is complex.

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.

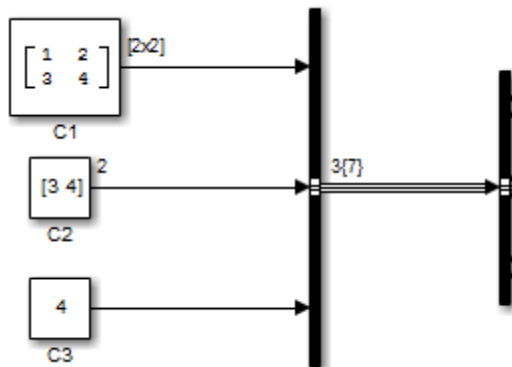


Ranges are displayed in the format [min..max]. In the above example, the design range at the output port of the Mux block is displayed as [-10..mixed], because the two signals the Mux block combines have the same design minimum but different design maximums.

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 $[N_1 \times N_2]$ 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 $\{M\}$.

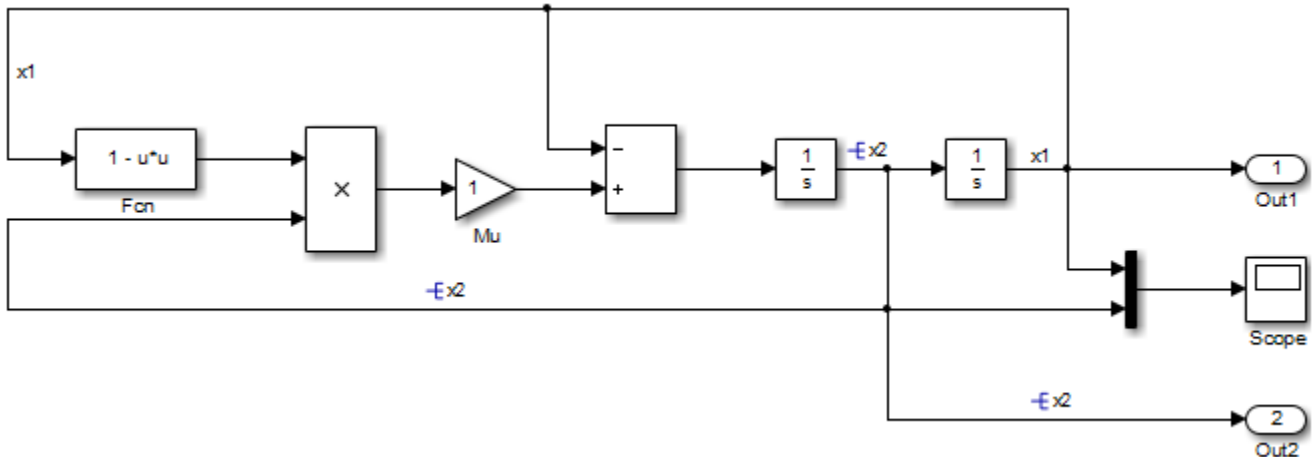
Signal to Object Resolution Indicator

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 a signal's **Signal name must resolve to signal object** property 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:



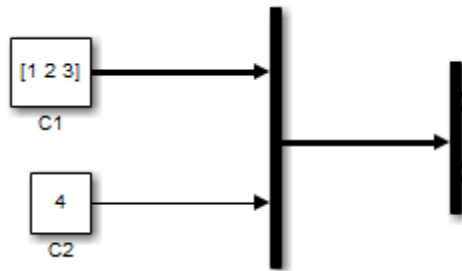
To suppress the display of signal resolution icons, in the model window deselect **Display > Signals & Ports > Signal to Object Resolution Indicator**, which is selected by default. To restore signal resolution icons,

reselect **Signal to Object Resolution Indicator**. 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 4-83
- “Initialize Signals and Discrete States” on page 50-52
- Simulink.Signal

Wide Nonscalar Lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.



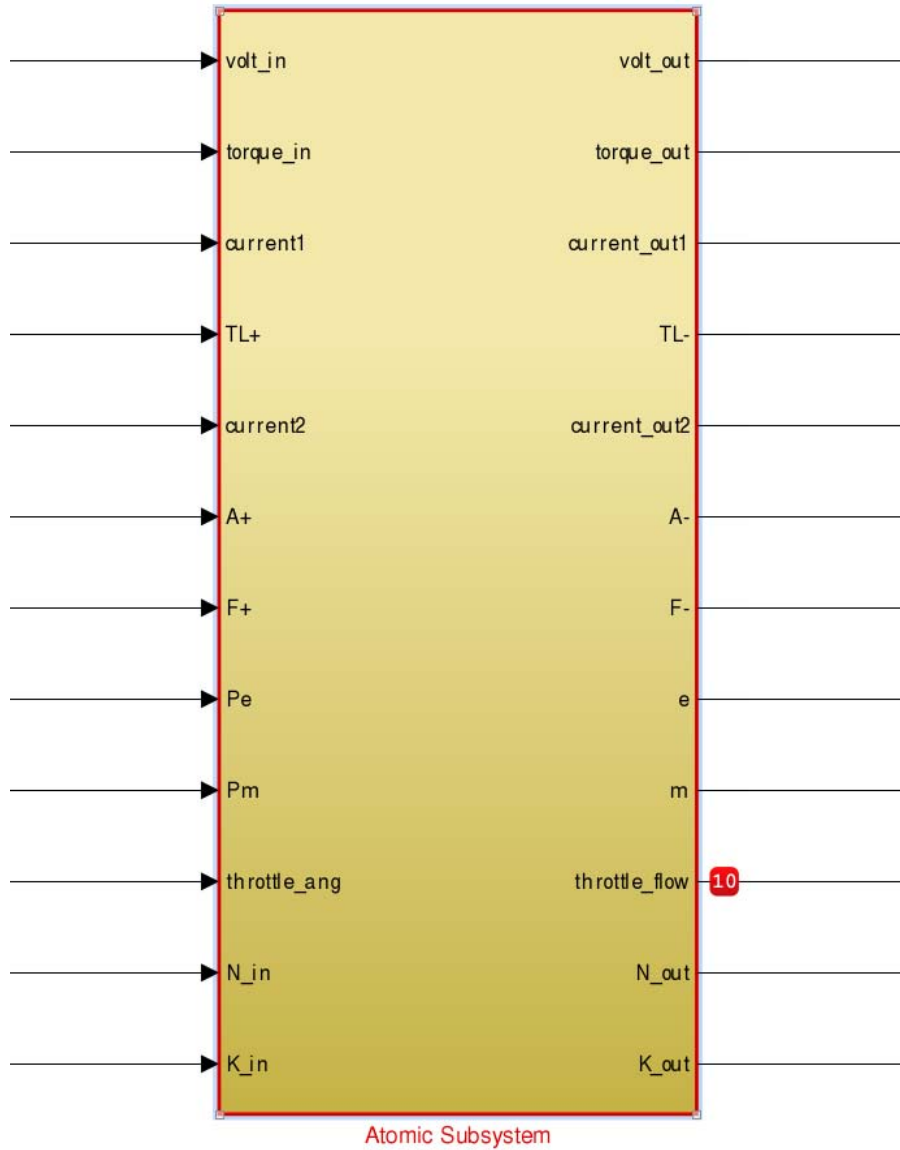
See “Composite Signals” on page 51-3 for more information about vector and matrix signals.

Display Port Numbers When Addressing Errors

Some error messages highlight the block in which the error occurs. To display the number of a port in the highlighted block, hover directly over that port. The port can be any of the following types of port:

- Input data port
- Output data port
- Connection port for a Physical Modeling product, SimEvents, or SimRF™

The port number appears in green, next to the port. For example, hovering over the `throttle_flow` port shows that its port number is 10.



Signal Groups

In this section...

“About Signal Groups” on page 50-69

“Signal Builder Window” on page 50-69

“Creating Signal Group Sets” on page 50-83

“Editing Waveforms” on page 50-113

“Signal Builder Time Range” on page 50-119

“Exporting Signal Group Data” on page 50-120

“Printing, Exporting, and Copying Waveforms” on page 50-120

“Simulating with Signal Groups” on page 50-121

“Simulation Options Dialog Box” on page 50-122

About Signal Groups

The Signal Builder block displays and allows you to create or edit interchangeable groups of signal sources and quickly switch the groups into and out of a model.

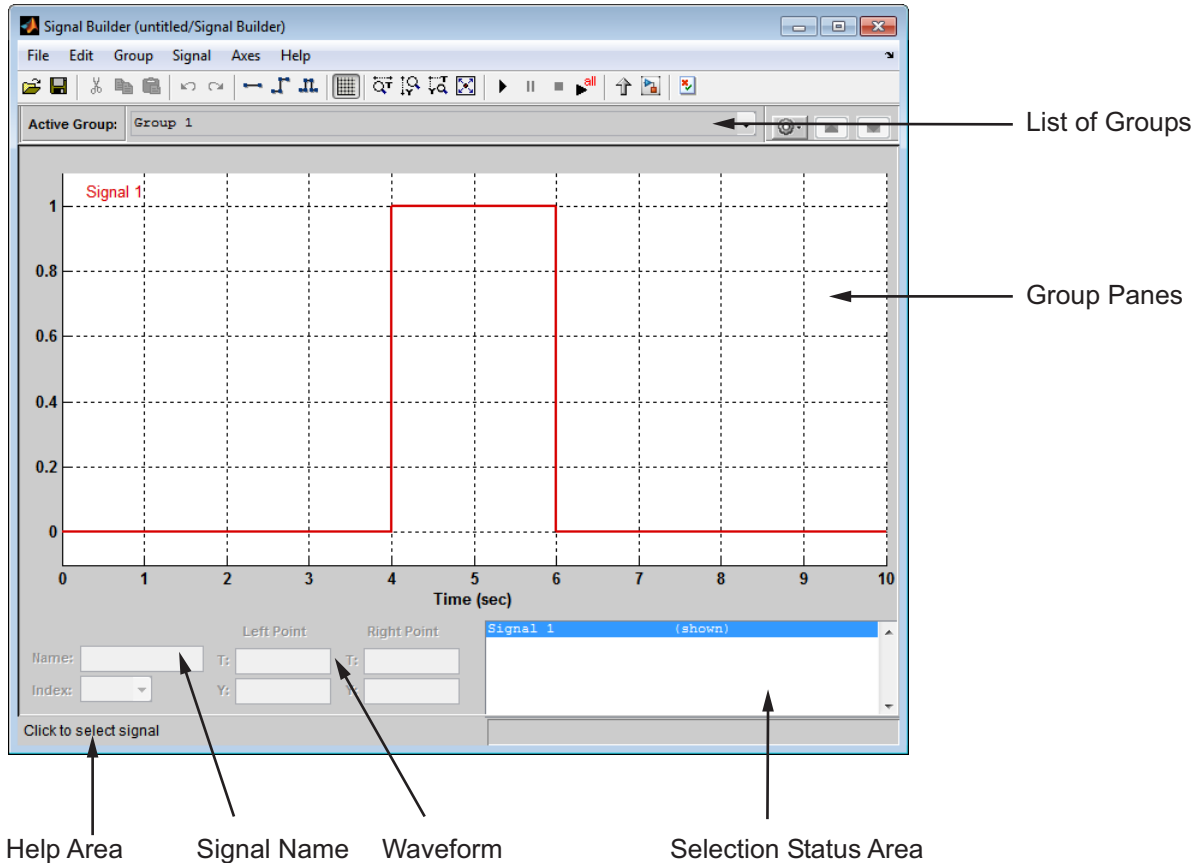
Signal groups can greatly facilitate testing a model, especially when you use them with conjunction with Simulink Assertion blocks and the Model Coverage Tool from the Simulink Verification and Validation. For a description of the Model Coverage Tool, see “Model Coverage Collection Workflow”.

Model Configuration Parameter **Solver** pane settings can affect the Signal Builder block output. See “Solvers” on page 3-21 for a description of how solvers affect simulation.

Signal Builder Window

The Signal Builder block window allows you to define the shape of the signals (waveform) output by the block. You can specify any waveform that is piecewise linear.

To open the window, double-click the block. The Signal Builder window appears.



The Signal Builder window allows you to create and modify signal groups represented by a Signal Builder block. The Signal Builder window includes the following controls.

Group Pane

Displays the set of interchangeable signal source groups represented by the block. The pane for each group displays an editable representation of each waveform in the group. The name of the group appears at the top of the pane.

Only one pane is visible at a time. To display a group that is not visible, from the list, select the group name. The block outputs the group of signals whose pane is currently visible. Each pane occupies a pane in the Signal Builder block dialog box.

Signal Axes

The signals appear on separate axes that share a common time range (see “Signal Builder Time Range” on page 50-119). This presentation allows you to compare the relative timing of changes in each signal. The Signal Builder automatically scales the range of each axis to accommodate the signal that it displays. Use the Signal Builder **Axes** menu to change the time (T) and amplitude (Y) ranges of the selected axis.

Signal List

Displays the names and visibility (see “Editing Signals” on page 50-73) of the signals that belong to the currently selected signal group. Clicking an entry in the list selects the signal. Double-clicking a signal entry in the list hides or displays the waveform on the group pane.

Selection Status Area

Displays the name of the currently selected signal and the index of the currently selected waveform segment or point.

Waveform Coordinates

Displays the coordinates of the currently selected waveform segment or point. You can change the coordinates by editing the displayed values (see “Editing Waveforms” on page 50-113).

Name

Name of the currently selected signal. You can change the name of a signal by editing this field (see “Renaming a Signal” on page 50-79).

Index

Index of the currently selected signal. The index indicates the output port at which the signal appears. An index of 1 indicates the topmost output port, 2

indicates the second port from the top, and so on. You can change the index of a signal by editing this field (see “Changing a Signal Index” on page 50-83).

Help Area

Displays context-sensitive tips on using Signal Builder window features.

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**.

A dialog box appears.

- 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 (described in the table).

Waveform	Description	Inputs
Constant	Constant waveform	None.
Step	Step waveform	None.
Pulse	Pulse waveform	None.
Square	Square waveform	<ul style="list-style-type: none"> • Frequency Waveform frequency, in hertz • Amplitude Waveform amplitude • Offset Waveform vertical offset • % Duty cycle Percent of the period the signal is positive (a value between 0 and 100)

Waveform	Description	Inputs
Triangle	Triangle waveform	<ul style="list-style-type: none"> • Frequency Waveform frequency, in hertz. • Amplitude Waveform amplitude • Offset Waveform vertical offset
Sampled Sin	Sampled sinewave waveform	<ul style="list-style-type: none"> • Frequency (Hz) Waveform frequency, in hertz • Amplitude Waveform amplitude • 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	<ul style="list-style-type: none"> • Frequency 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
Pseudorandom Noise	Pseudorandom noise waveform based on a binomial distribution with upper and lower values at input frequency	<ul style="list-style-type: none"> • Frequency 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 <p>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</p>

Waveform	Description	Inputs
		<p>resulting vectors must be of equal length.</p> <hr/> <p>Note Signal Builder displays a warning if you add a custom waveform with a large number of data points (100,000,000 or more).</p> <hr/>

- 3 Select the waveform you want to add.
- 4 Specify the inputs (in 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.

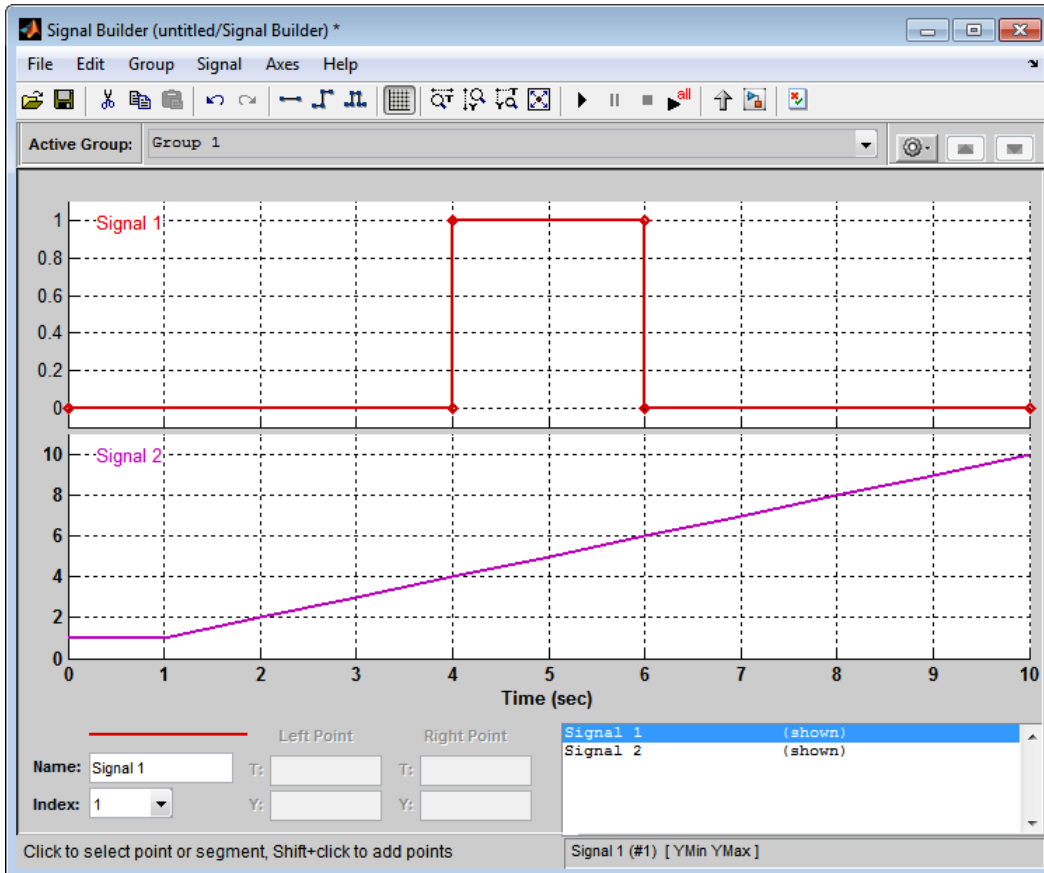
- 1 In the MATLAB Command Window, create data for two variables, t and y .

```
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 non-virtual 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, the 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.

Waveform	Description	Inputs
Square	Square waveform.	<ul style="list-style-type: none"> • Frequency Waveform frequency, in Hertz. • Amplitude Waveform amplitude. • Offset Waveform vertical offset. • % Duty cycle Percent of the period in which the signal is positive. Enter a value between 0 and 100.
Triangle	Triangle waveform.	<ul style="list-style-type: none"> • Frequency Waveform frequency, in Hertz. • Amplitude Waveform amplitude • Offset Waveform vertical offset.
Sampled Sin	Sampled sinewave waveform.	<ul style="list-style-type: none"> • Frequency (Hz) Waveform frequency, in Hertz. • Amplitude Waveform amplitude • Offset Waveform vertical offset. • Samples Per Period

Waveform	Description	Inputs
		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.	<ul style="list-style-type: none"> • Frequency 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.	<ul style="list-style-type: none"> • Frequency 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

Waveform	Description	Inputs
		<p>The initial seed value for the random number generator</p>
<p>Poisson Random Noise</p>	<p>Poisson random noise waveform that alternates between 0 and 1.</p>	<ul style="list-style-type: none"> • 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
<p>Custom</p>	<p>Custom piecewise linear waveform. Custom values must fit within the display area.</p>	<ul style="list-style-type: none"> • 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. <p>The entries in either field can be any MATLAB expression that evaluates to a vector. The resulting vectors must be of equal length.</p>

Waveform	Description	Inputs
		<hr/> <p>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.</p> <hr/>

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, 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 (see “Signal List” on page 50-71).

Creating Signal Group Sets

You can create signal groups in the Signal Builder block by:

- “Creating Signal Group Sets Manually” on page 50-84
- “Importing Signal Group Sets” on page 50-85
- “Importing Data with Custom Formats” on page 50-111

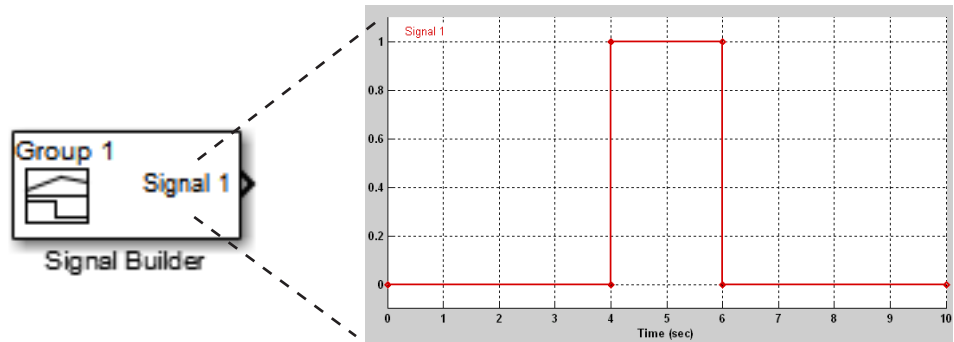
You can also use the `signalbuilder` function to populate the Signal Builder block.

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 50-85.

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 (see “Signal Builder Window” on page 50-69) 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 50-121 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 50-85 for a description of the data formats that the Signal Builder block accepts. The procedures in the following topics use the file `matlabroot\help\toolbox\simulink\ug\examples\signals\3Grp_3Sig.xls`.

See “Data Import and Logging Workflow” for a description of the Signal Builder block in a simulation workflow.

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)

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 must be first column

1	Time	DC In	Trigger	AC In
2		0	1	2
3		1	2	3
4		2	3	4
5		3	4	5
6		4	5	6
7		5	6	7
8		6	7	8
9		7	8	9
10		8	9	10
11		9	10	11
12		10	11	12
13		11	12	13
14		12	13	14
15		13	14	15
16		14	15	16
17		15	16	17
18		16	17	18

Signal names (optional)

Worksheets - equivalent to signal group

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:
 - `time`
 - `data`
 - `sigName`
 - `groupName`

`sigName` and `groupName` are optional.

- For backwards compatibility, the Signal Builder block supports logged data from the `Simulink.ModelDataLogs` object and interprets this data as a single group. The `ModelDataLogs` format will be removed in a future release.
- Signal Builder block does not support:
 - Simulink output as only a structure
 - Simulink output as only an array

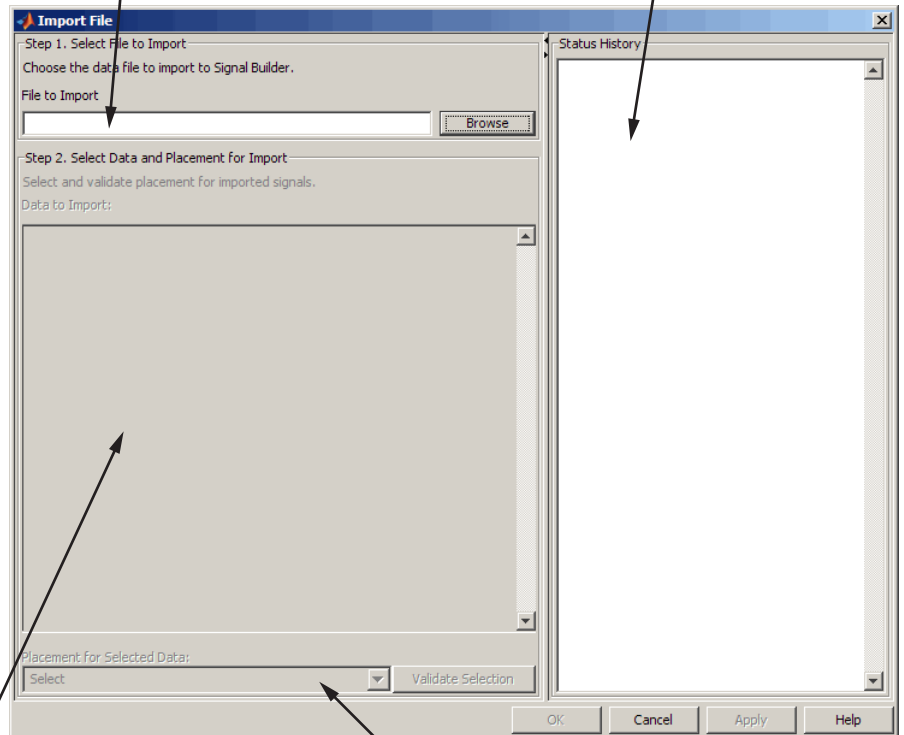
Note Signal Builder returns a warning if you import a large number of data points (100,000,000 or more). You can then cancel the action.

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.

Signal Builder Block Import File Dialog Box. The Signal Builder Import File dialog box allows you to import existing signal data files into the Signal Builder block.

Signal data file to import

Repository of data import status messages



Tree view of signal data file contents

Drop-down list of import actions for signal data

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**.

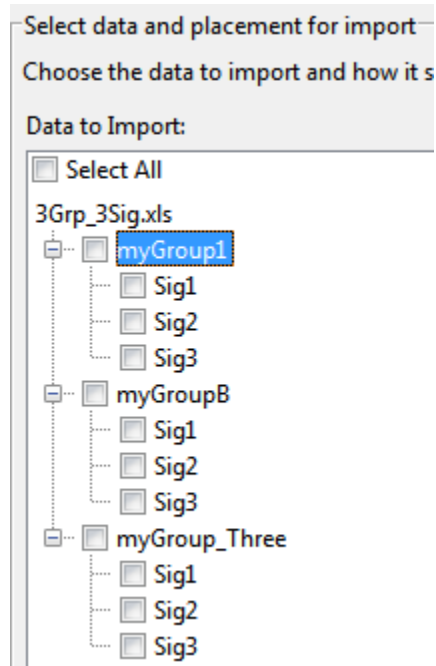
The file browser appears.

- 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

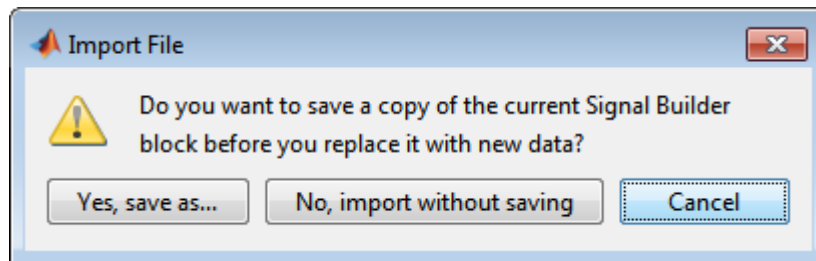
(Names may have been renamed for uniqueness.)

.....

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.

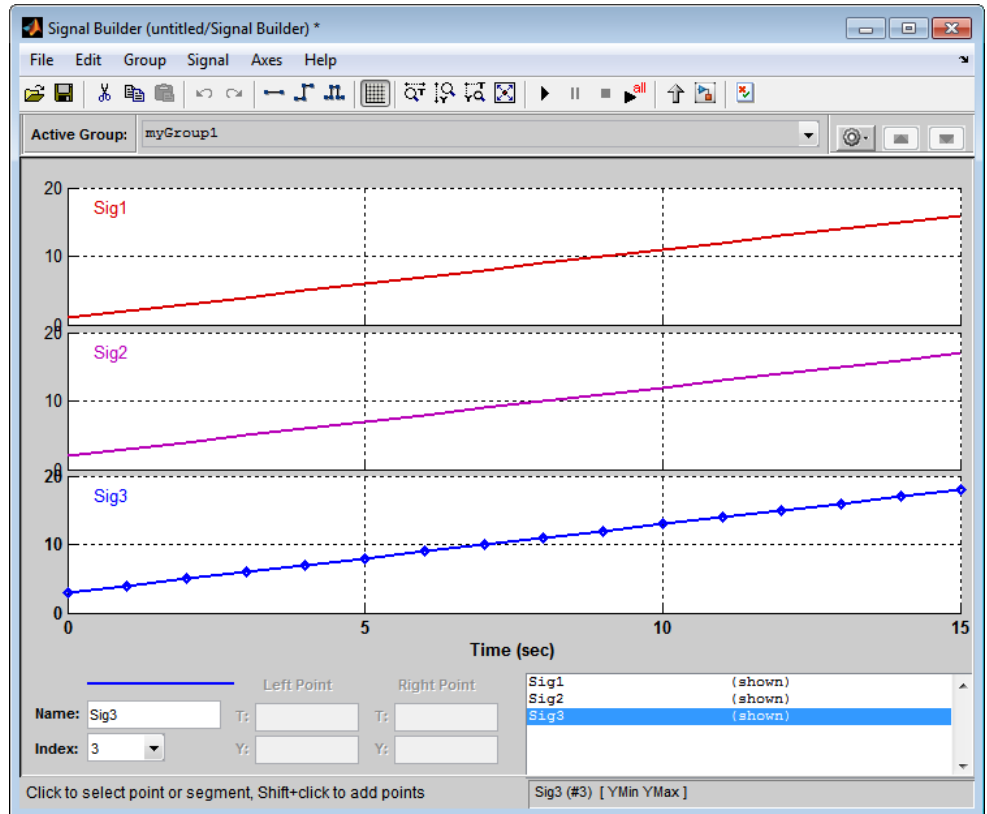


- 10 Click a button, as follows:

To...	Click...
<p>Save the contents of the Signal Builder block before replacing it with the new signal data.</p> <hr/> <p>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.</p> <hr/>	Yes, save as
<p>Replace the contents of the Signal Builder block without saving them first.</p>	No, import without saving
<p>Stop the replacement process.</p>	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 50-90.

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.

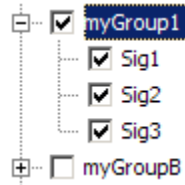
5 If you select the file browser, navigate to and select a signal data file. For example, select `3Grp_3Sig.xls`.

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.

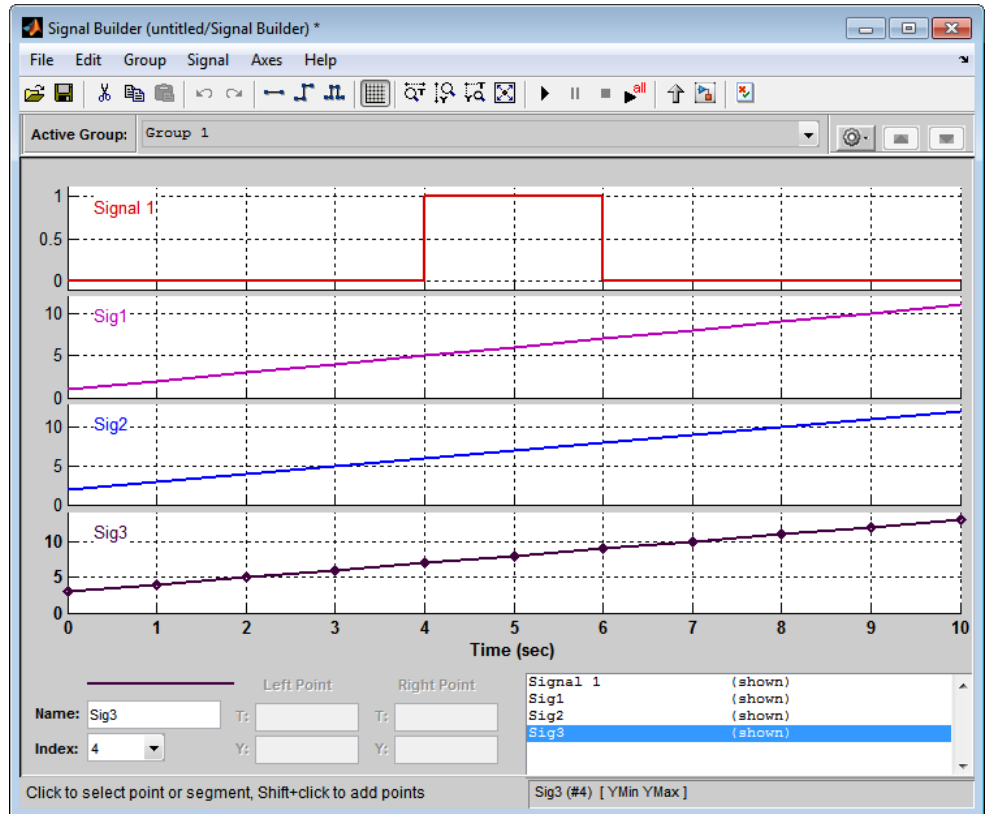
If the requested action is a valid one, the Status History pane displays messages to indicate the state. For example:

```
3 signal(s) will be appended to each group.  
Selected signal name(s):  
Before:  
  Sig1  
  Sig2  
  Sig3  
  
After:  
  Sig4  
  Sig5  
  Sig6  
  
Signal name(s) in the block:  
Before:  
  Sig1  
  Sig2  
  Sig3  
  
After:  
  Sig1  
  Sig2  
  Sig3  
  Sig4  
  Sig5  
  Sig6  
  
(Names may have been renamed for uniqueness.)  
.....
```

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.

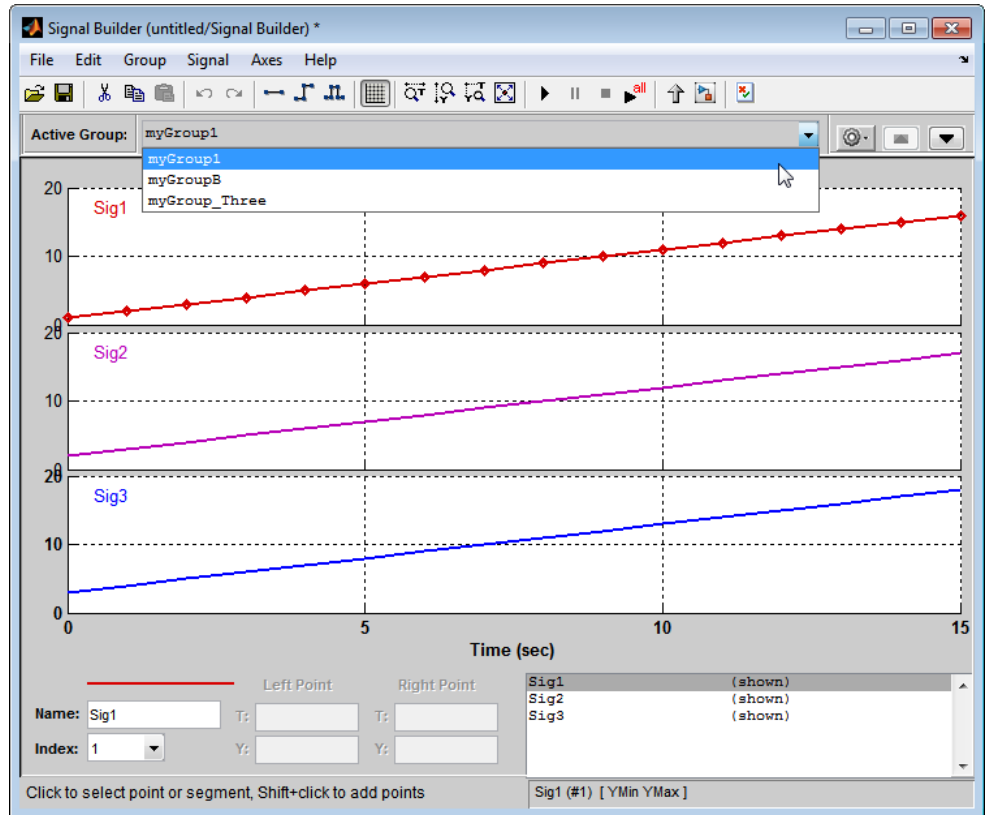
This topic uses `signalbuilder1` from the procedure in “Replacing All Signal Data with Selected Data” on page 50-90.

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. For example, this Signal Builder block has three groups, `myGroup1`, `myGroupB`, and `myGroup_Three`.



4 Double-click the block.

The Import File dialog box appears.

5 In the **File to Import** field, enter a signal data file name or click **Browse**.

The file browser appears.

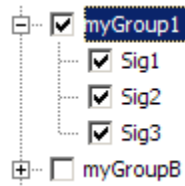
6 If you select the file browser, navigate to and select a signal data file. For example, select 3Grp_3Sig.xls.

Note If you try to import an improperly formatted signal data file, an error message popup window. 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:

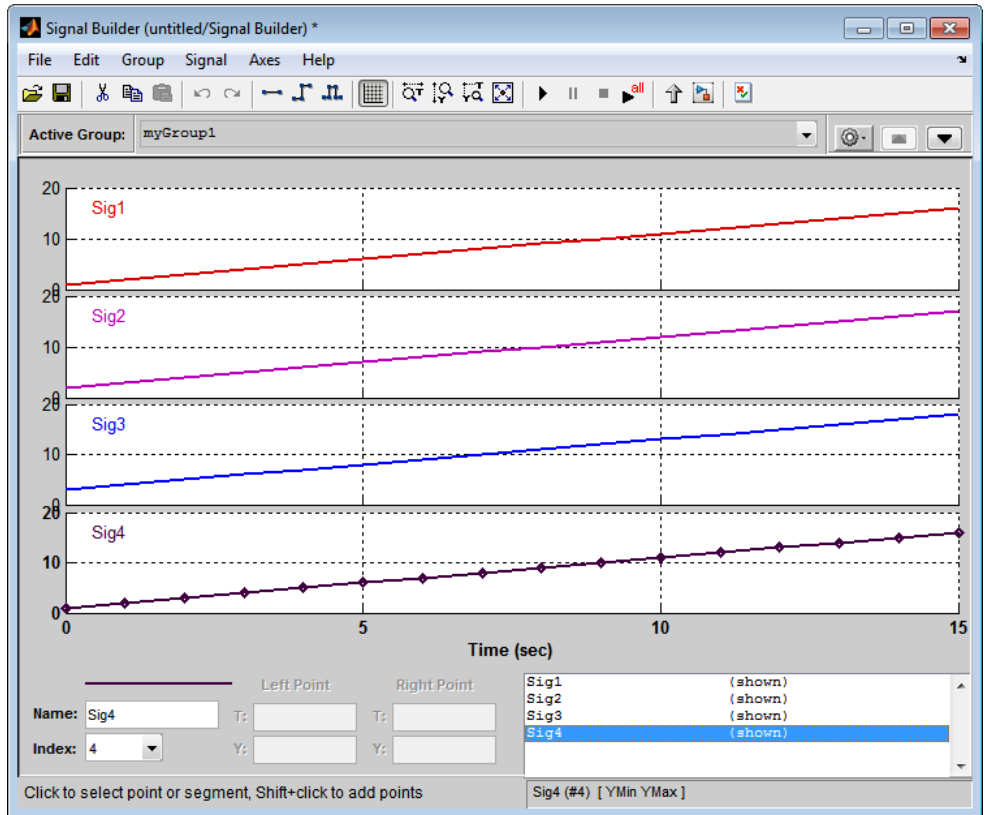
Sig4

The confirmation also enables the **OK** and **Apply** buttons.

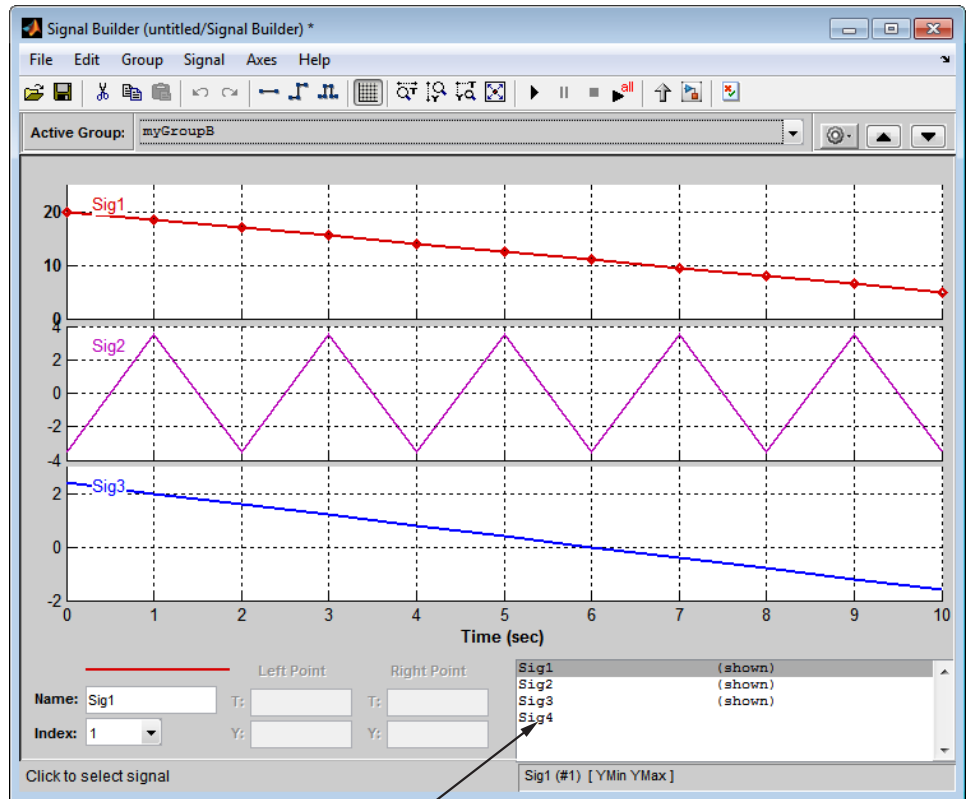
- 10** If you are satisfied with the status message, click **Apply** to append the signals.

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.

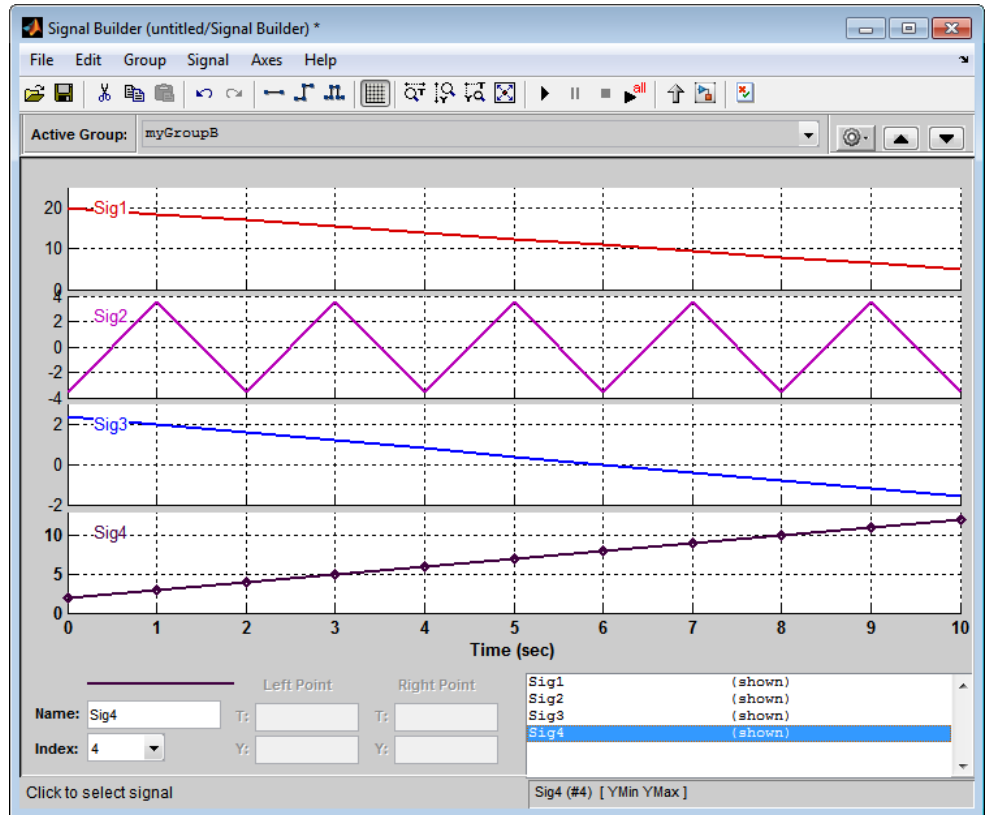


- 11 Click another group name, for example, myGroupB. Notice that Sig4 exists for the group, hidden by default.



Sig4 appears in signal list, but does not appear in group pane

- 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.

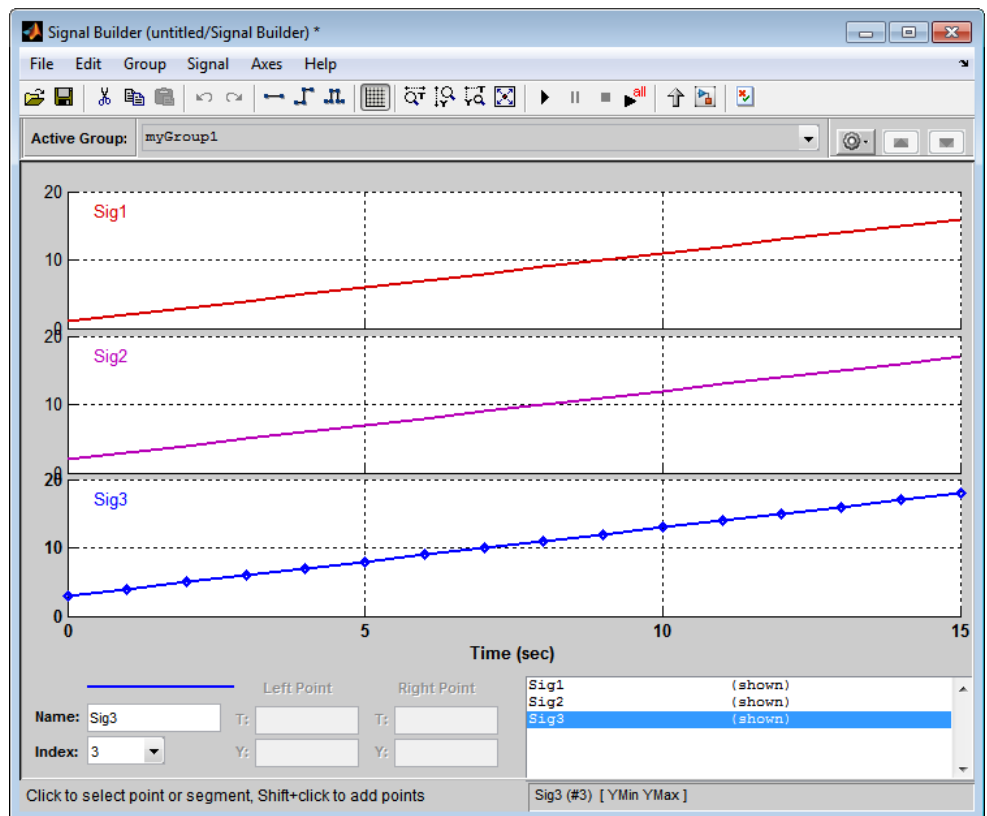
Appending Signal Groups to Existing Groups. You can append one or more signal groups to the end of the list of existing signal groups. If the block already has a signal group with the same name as the one you are adding, the software increments the group name by 1 or higher until it is unique before 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 50-90.

- 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.

The Import File dialog box appears.

- 5 In the **File to Import** text field, enter a signal data file name or click **Browse**.

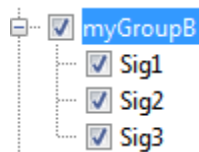
The file browser appears.

- 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

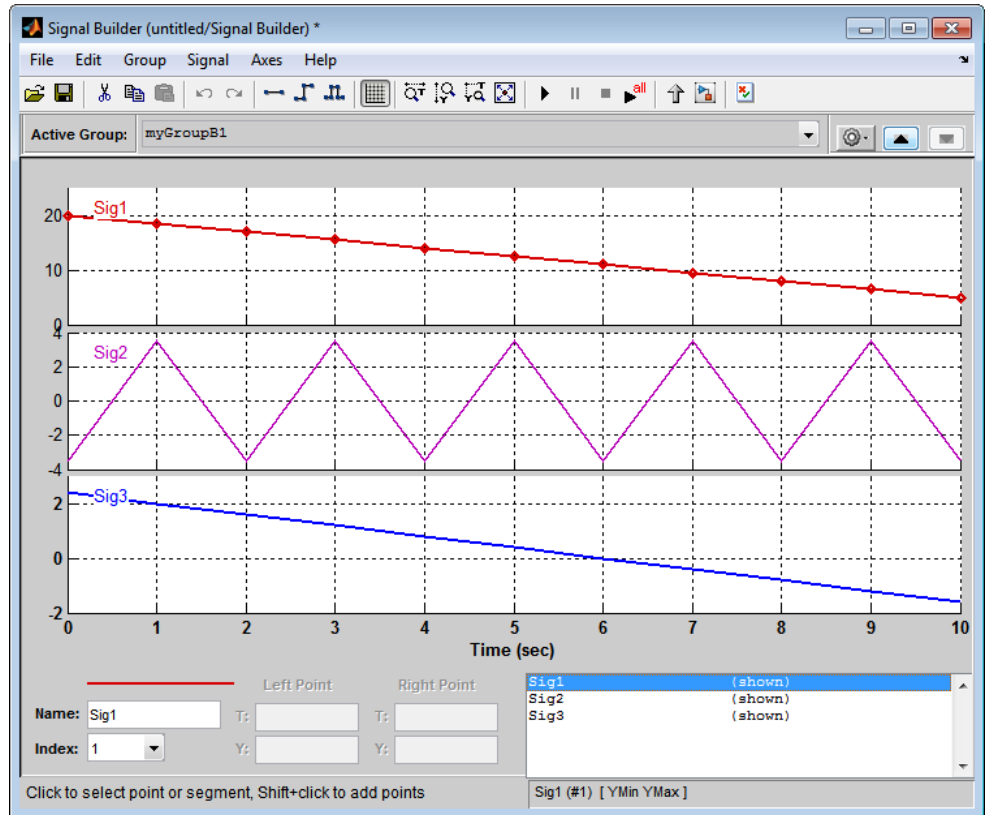
Sig3

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.



- 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 a signal data from a file whose contents you want to import. See “Importing Signal Groups from Existing Data Sets” on page 50-85 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 `matlabroot\help\toolbox\simulink\examples`, 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 strings with number of rows = 1, number of columns = number of groups.
- *sigNames* — Cell array that contains signal name strings 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 [1xN] or [Nx1], 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 50-85).

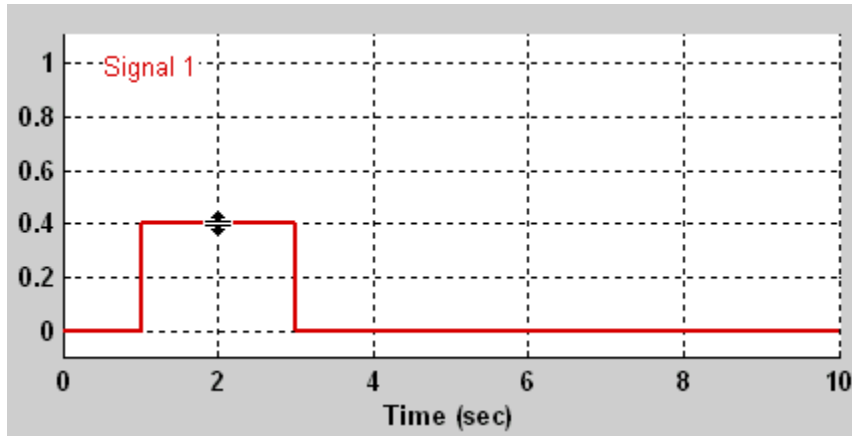
Editing Waveforms

Signal Builder allows you to change the shape, color, and line style and thickness of the waveforms output by a group.

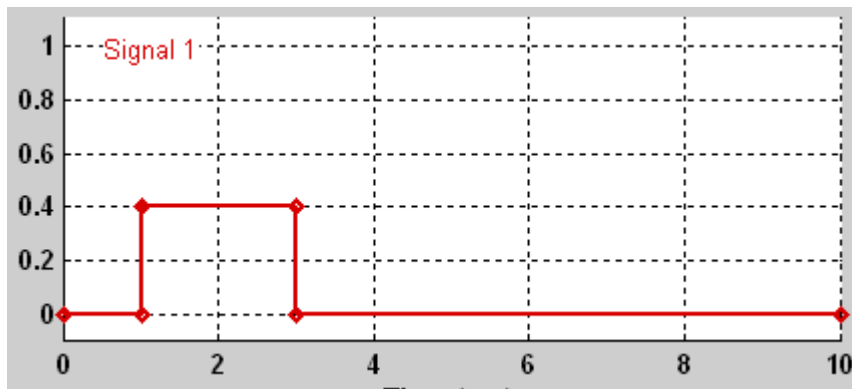
Reshaping a Waveform

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.

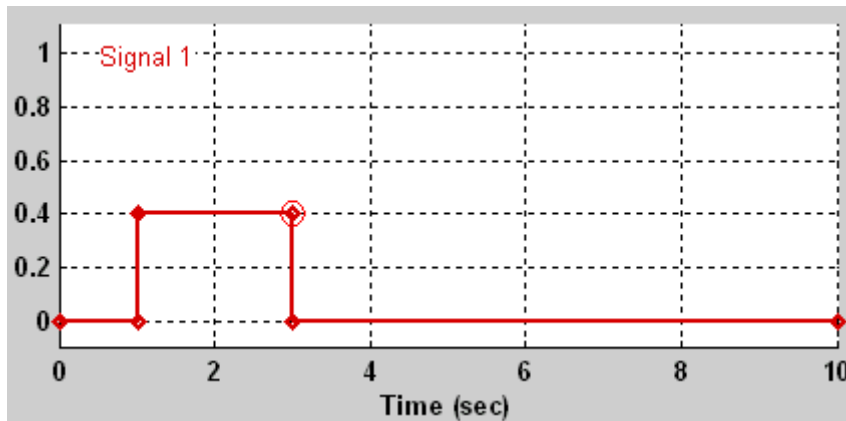


To deselect a waveform, left-click any point on the waveform axis that is not on the waveform itself or press the **Esc** key.

Selecting Points. To select a point of 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.

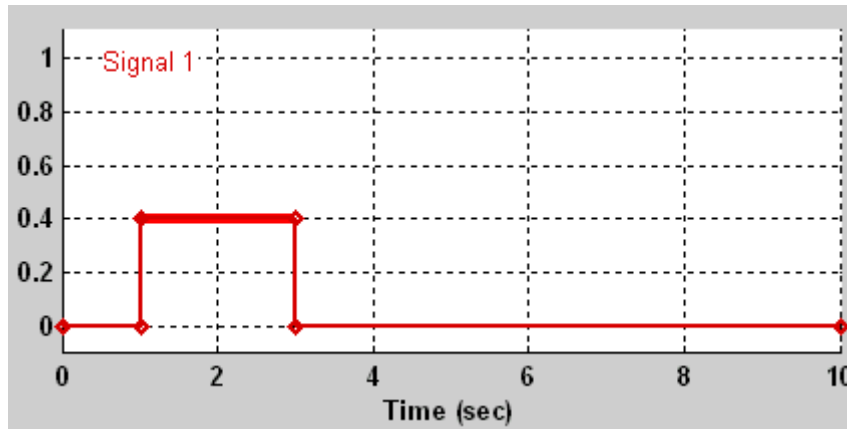


Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate your selection.



To deselect the point, press the **Esc** key.

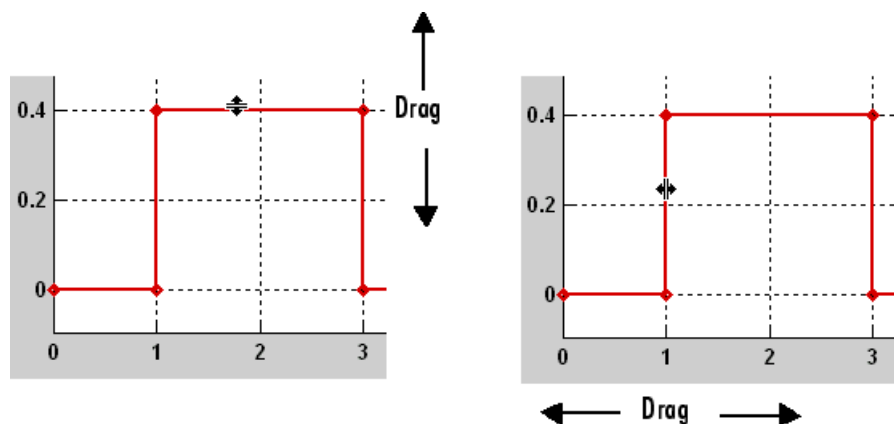
Selecting Segments. To select a line segment, first select the waveform that contains it. Then left-click the segment. The Signal Builder thickens the segment to indicate that it is selected.



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 50-117) 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. A menu of line styles pops up. 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. A dialog box appears with the line current thickness. Edit the thickness value and click **OK**.

Signal Builder Time Range

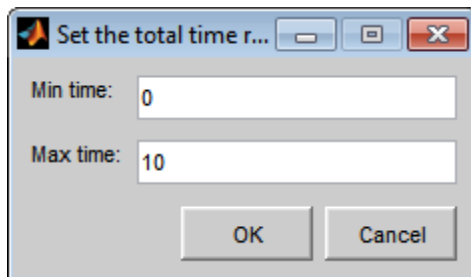
The Signal Builder time range determines the span of time over which its output is explicitly defined. By default, the time range runs from 0 to 10 seconds. You can change both the beginning and ending times of a block time range (see “Changing a Signal Builder Time Range” on page 50-119).

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 50-122 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

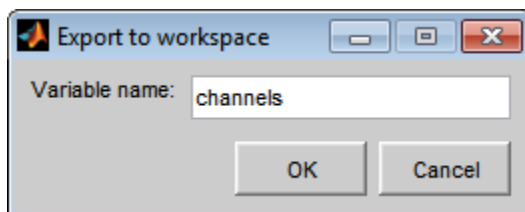
To change the time range, select **Change Time Range** from the Signal Builder **Axes** menu. A dialog box appears.



Edit the **Min time** and **Max time** fields as necessary to reflect the beginning and ending times of the new time range, respectively. Click **OK**.

Exporting Signal Group Data

To export the data that define a Signal Builder block signal groups to the MATLAB workspace, select **Export to Workspace** from the block **File** menu. A dialog box appears.



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. Click **OK**. 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. You can access the coordinate values defining signals associated with other signal groups from the structure **allXData** and **allYData** fields.

Printing, Exporting, and Copying Waveforms

Signal Builder allows you to print, export, and copy the waveforms visible in the active signal group.

To print the waveforms to a printer, select **Print** from the block **File** menu.

You can also export the waveforms to other destinations by using the **Export** option from the block **File** menu. From this submenu, select one of the following destinations:

- **To File** — Converts the current view to a graphics file.

Select the format of the graphics file from the **Save as type** drop-down list on the resulting **Export** dialog box.

- **To Figure** — Converts the current view to a MATLAB figure window.

To copy the waveforms to the system clipboard for pasting into other applications, select **Copy Figure To Clipboard** from the block **Edit** menu.

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** command in the Signal Builder window (see “Running All Signal Groups” on page 50-121).

If you want to capture inputs and outputs that the **Run all** command 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, etc., 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 **Run all** button



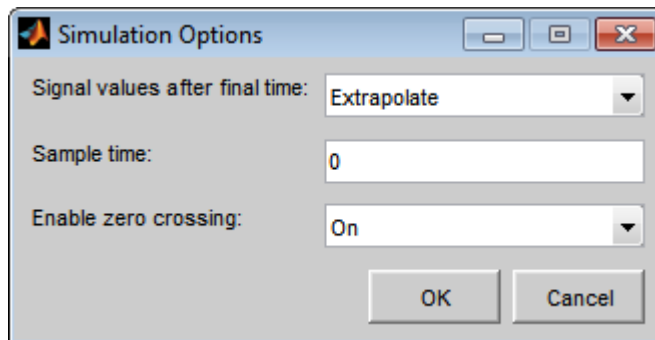
from the Signal Builder toolbar. The **Run all** command runs a series of simulations, one for each signal group defined by the block. If you installed Simulink Verification and Validation on your system and are using the Model Coverage Tool, the **Run all** command configures the tool to collect and 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 Options Dialog Box

The **Simulation Options** dialog box allows you to specify simulation options pertaining to the Signal Builder. To display the dialog box, select **Simulation Options** from the **File** menu of the Signal Builder window. The dialog box appears.



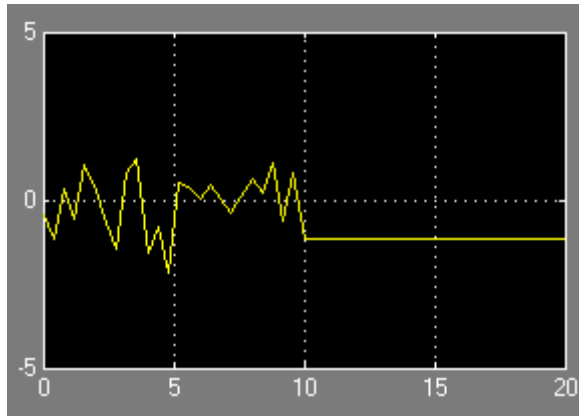
The dialog box allows you to specify the following options.

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. The options are

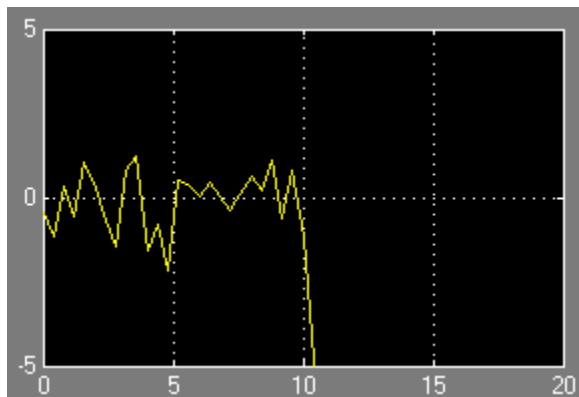
- Hold final value

Selecting this option causes the Signal Builder block to output the last defined value of each signal in the currently active group for the remainder of the simulation.



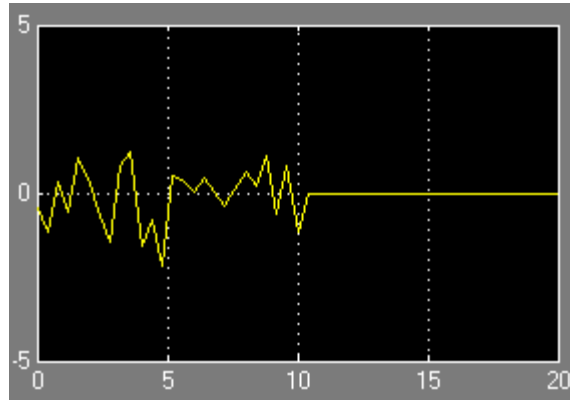
- Extrapolate

Selecting this option causes the Signal Builder block to output values extrapolated from the last defined value of each signal in the currently active group for the remainder of the simulation.



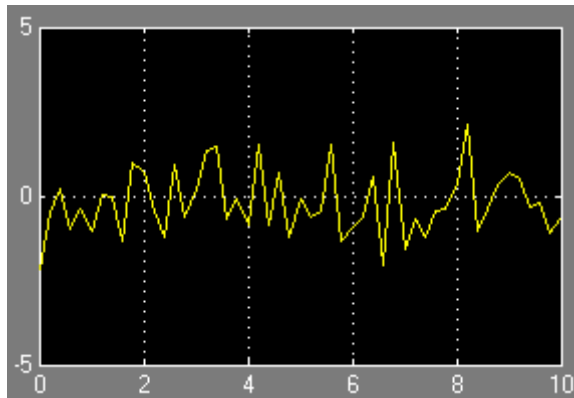
- Set to zero

Selecting this option causes the Signal Builder block to output zero for the remainder of the simulation.

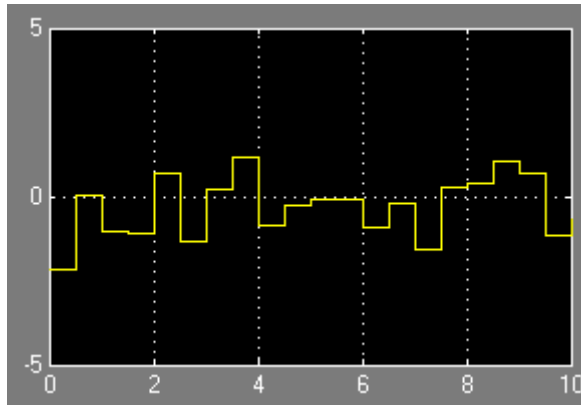


Sample time

Determines whether the Signal Builder block outputs a continuous (the default) or a discrete signal. If you want the block to output a continuous signal, enter 0 in this field. 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.



If you want the block to output a discrete signal, enter the sample time of the signal in this field. 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

Specifies whether the Signal Builder block detects zero-crossing events (enabled by default). For more information, see “Zero-Crossing Detection” on page 3-24.

Using Composite Signals

- “Composite Signals” on page 51-3
- “Buses” on page 51-6
- “Virtual and Nonvirtual Buses” on page 51-13
- “Create and Access a Bus” on page 51-18
- “Nest Buses” on page 51-20
- “Bus-Capable Blocks” on page 51-22
- “Bus Objects” on page 51-24
- “Bus Object API” on page 51-27
- “Manage Bus Objects with the Bus Editor” on page 51-28
- “Store and Load Bus Objects” on page 51-50
- “Map Bus Objects to Models” on page 51-52
- “Filter Displayed Bus Objects” on page 51-53
- “Customize Bus Object Import and Export” on page 51-59
- “Connect Buses to Inports and Outports” on page 51-65
- “Specify Initial Conditions for Bus Signals” on page 51-69
- “Combine Buses into an Array of Buses” on page 51-81
- “Arrays of Buses in Models” on page 51-87
- “Convert Models to Use Arrays of Buses” on page 51-94
- “Code Generation for Arrays of Buses” on page 51-97
- “Bus Data Crossing Model Reference Boundaries” on page 51-98
- “Buses and Libraries” on page 51-99

- “Prevent Bus and Mux Mixtures” on page 51-100
- “Correct Mux Blocks That Create Bus Signals” on page 51-104
- “Correct Buses Used as Muxes” on page 51-109
- “Buses in Generated Code” on page 51-114
- “Composite Signal Limitations” on page 51-115

Composite Signals

In this section...

“What is a Composite Signal?” on page 51-3

“Techniques for Combining Signals” on page 51-3

What is a Composite Signal?

A *composite signal* is a signal that is composed of other signals. The constituent signals originate separately and join to form the composite signal. You can then extract individual signals from the composite signal downstream and use the signal as if it had never been part of a composite signal.

Composite signals can reduce visual complexity in models by grouping signals that run in parallel over some or all of their courses. Composite signals offer other benefits, as described in the documentation for each technique for combining signals.

Techniques for Combining Signals

Simulink provides several techniques for combining signals into a composite signal, depending on your modeling requirements.

Composite Signal Technique	When to Use
Virtual bus	Combine signals of any type and dimension, without affecting the memory layout. Use for graphical convenience. See “Choosing Between Virtual and Nonvirtual Buses” on page 51-14.
Nonvirtual bus	Combine signals of different types and dimensions such that they are contiguous in memory. Produces a structure in the generated code. See “Choosing Between Virtual and Nonvirtual Buses” on page 51-14.

Composite Signal Technique	When to Use
Mux block	Graphically combine signals of same type. See “Buses and Muxes” on page 51-4.
Vector Concatenate and Matrix Concatenate block	Create a vector or matrix, for example to be used in mathematical operations. The resulting vector is stored in contiguous memory.
Array of buses	Combine multiple buses with identical properties. Array of buses is equivalent to an array of structures in MATLAB. See “Combine Buses into an Array of Buses” on page 51-81.

Buses and Muxes

If all signals in a bus are the same type, you may be able to use a contiguous vector or a virtual vector (mux) instead of a bus. For more information, see “Mux Signals” on page 50-11.

In the context of a model, do use bus and mux mixtures. A bus and mux mixture occurs when some blocks treat a signal as a mux, while other blocks treat that same signal as a bus. For more information, see “Prevent Bus and Mux Mixtures” on page 51-100.

Bus Code

The various techniques for defining buses are essentially equivalent for simulation, but the techniques used can make a significant difference in the efficiency, size, and readability of generated code. If you intend to generate production code for a model that uses buses, for information about the best techniques to use, in the Embedded Coder documentation, see “Buses”.

Function-Call Signals

To combine function-call signals to call a single system, use a Mux block.

If you use a Bus Creator block to group function-call signals into one bus, then use a Bus Selector block to separate out specific function-call signals.

Buses

In this section...
“What is a Bus?” on page 51-6
“Types of Simulink Buses” on page 51-7
“Bus Objects” on page 51-7
“View Information about Buses” on page 51-7

Tip Simulink provides several techniques for combining signals into a composite signal. For a comparison of techniques, see “Techniques for Combining Signals” on page 51-3.

What is a Bus?

A Simulink composite signal is called a *bus signal*, or just a *bus*. A Simulink bus is analogous to a bundle of wires held together by tie wraps. Simulink implements a bus as a name-based hierarchical structure. A Simulink bus should not be confused with a hardware bus, like the bus in the backplane of many computers. It is more like a programmatic structure defined in a language like C.

The signals that constitute a bus are called *elements*. The constituent signals retain their separate identities within the bus and can be of any type or types, including other buses nested to any level. The elements of a bus can be any of the following:

- Mixed data type signals (e.g. double, integer, fixed point)
- Mixture of scalar and vector elements
- Buses as elements
- N-D signals
- Mixture of Real and Complex signals

Some requirements and limitations apply when you connect buses to blocks or to nonvirtual subsystems. See “Bus-Capable Blocks” on page 51-22, “Connect

Buses to Inports and Outports” on page 51-65, and “Composite Signal Limitations” on page 51-115 for more information.

Types of Simulink Buses

A bus can be either *virtual* or *nonvirtual*. Both virtual and nonvirtual buses provide the same visual simplification, but their implementations are different.

- Virtual buses exist only graphically. They have no functional effects and do not appear in generated code; only the constituent signals appear. See “Virtual Signals” on page 50-11 for details. Simulink implements virtual buses with pointers, so virtual buses add no data copying overhead and do not affect performance.
- Nonvirtual buses may have functional effects. They appear as structures in generated code, which can simplify the code and clarify its correspondence with the model. Simulink implements nonvirtual buses by copying data from the source signals to the bus, which can affect performance.

The two types of buses are interchangeable for many purposes, but some situations require a nonvirtual bus. See “Virtual and Nonvirtual Buses” on page 51-13 for more information.

Bus Objects

A bus can have an associated *bus object*, which can both provide and validate bus properties. A bus object is an instance of class `Simulink.Bus` that is defined in the base workspace. The object defines the structure of the bus and the properties of its elements, such as nesting, data type, and size. Bus objects are optional for virtual buses and required for nonvirtual buses. See “Bus Objects” on page 51-24 for more information. You can create bus objects programmatically or by using the Simulink Bus Editor, which facilitates bus object creation and management. See “Manage Bus Objects with the Bus Editor” on page 51-28 for more information.

View Information about Buses

To view information about buses, use one of the following approaches:

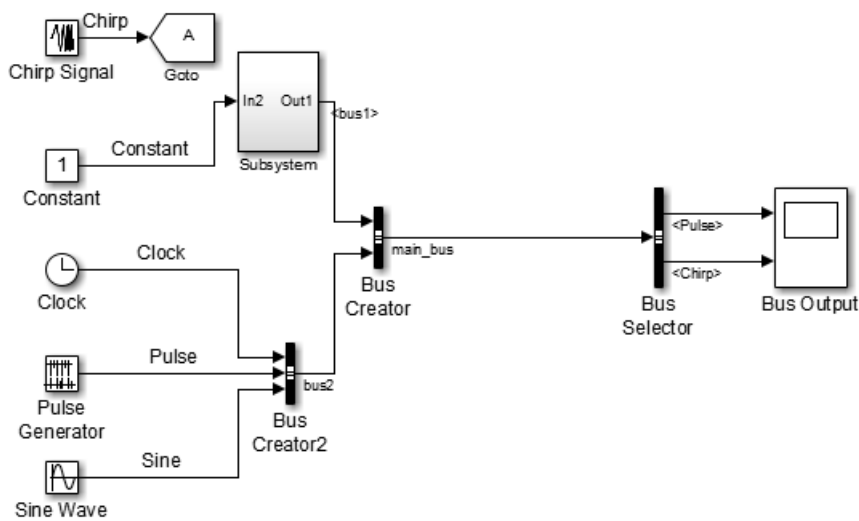
- Use the Signal Hierarchy Viewer to interactively display bus hierarchy (for bus signals)
- From the MATLAB command line, display the type and hierarchy of a bus signal in a compiled model. For details, see “CompiledBusType and SignalHierarchy Parameters” on page 51-10.

Signal Hierarchy Viewer

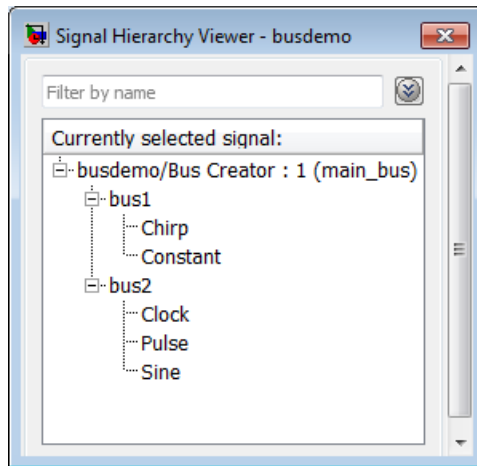
Use the Signal Hierarchy Viewer to interactively display information about a signal. For a bus signal, the Signal Hierarchy Viewer displays the bus hierarchy.

- 1 Check that the **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** parameter is set to error.
- 2 Right-click a signal line.
- 3 Select the **Signal Hierarchy** option. The Signal Hierarchy Viewer dialog box appears.

For example, open the busdemo model.



Right-click the `main_bus` signal (output signal for the Bus Creator block), and select **Signal Hierarchy**. The following information appears:




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.

You can also open the Signal Hierarchy Viewer in the Simulink Editor.

- 1 Select **Diagram > Signals & Ports > Signal Hierarchy**.
- 2 Select a signal

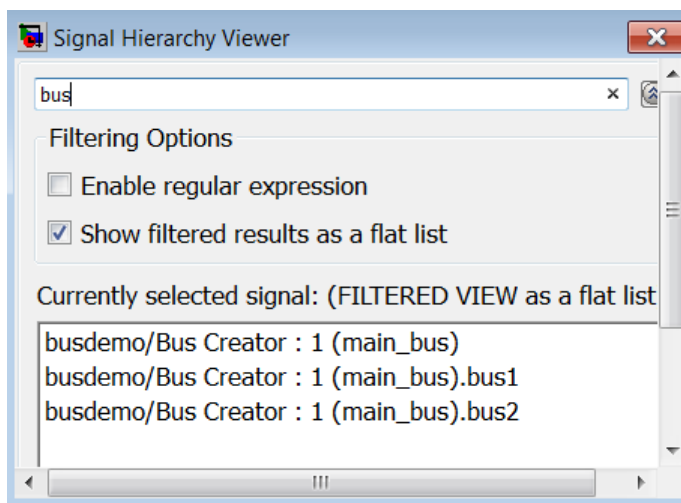
Note To produce accurate results at edit time, the Signal Hierarchy Viewer requires that the model compiles successfully.

To filter the displayed signals, click the **Options** button on the right-hand side of the **Filter by name** edit box (.

- To use MATLAB regular expressions for filtering signal names, select **Enable regular expression**. For example, entering `r$` in the **Filter by**

name edit box displays all signals whose names end with a lowercase `r` (and their immediate parents). For details, see “Regular Expressions”.

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



CompiledBusType and SignalHierarchy Parameters

To get information about the type and hierarchy of a bus signal in a compiled model, use these parameters with the `get_param` command:

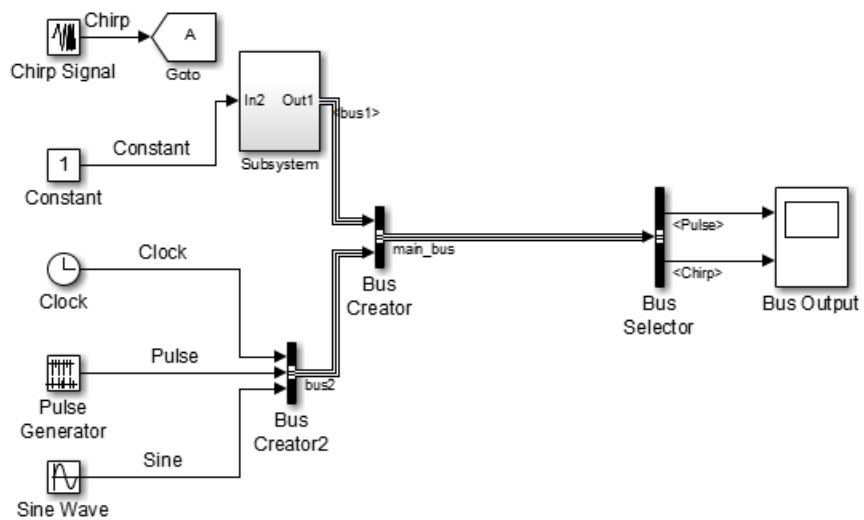
- `CompiledBusType` — For a compiled model, returns information about whether the signal connected to a port is a bus, and if so, whether the signal is a virtual or nonvirtual bus.
- `SignalHierarchy` — If the signal is a bus, returns the name and hierarchy of the signals in the bus.

Before you use these commands:

1 Set the **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** diagnostic to error.

2 Update the diagram or simulate the model.

For example, if you open and simulate the busdemo model, the model looks as shown below:



The following code illustrates how you can use the `SignalHierarchy` and `CompiledBusType` parameters:

```
mdl = 'busdemo';
open_system(mdl)
% Obtain the handle a port
ph = get_param([mdl '/Bus Creator'], 'PortHandles');
% SignalHierarchy is available at edit time
sh = get_param(ph.Outport, 'SignalHierarchy')
% Compile the model
busdemo([],[],[],'compile');
bt = get_param(ph.Outport, 'CompiledBusType')
% Terminate the model
busdemo([],[],[],'term');
```


Virtual and Nonvirtual Buses

In this section...

“About Virtual and Nonvirtual Buses” on page 51-13

“Choosing Between Virtual and Nonvirtual Buses” on page 51-14

“Creating Nonvirtual Buses” on page 51-15

“Nonvirtual Bus Sample Times” on page 51-16

“Automatic Bus Conversion” on page 51-16

“Explicit Bus Conversion” on page 51-17

About Virtual and Nonvirtual Buses

A bus signal can be *virtual*, meaning that it is just a graphical convenience that has no functional effect, or *nonvirtual*, meaning that the signal occupies its own storage.

Bus Storage

During simulation:

- A block connected to a *virtual* bus reads inputs and writes outputs by accessing the memory allocated to the component signals. These signals are typically noncontiguous, and no intermediate memory exists.
- A block connected to a *nonvirtual* bus reads inputs and writes outputs by accessing copies of the component signals. The copies are maintained in a contiguous area of memory allocated to the bus.

Compared with nonvirtual buses, virtual buses reduce memory requirements because they do not require a separate contiguous storage block, and execute faster because they do not require copying data to and from that block.

A nonvirtual bus is represented by a structure in generated code, which can be helpful when tracing the correspondence between the model and the code.

Simulation Results and Generated Code

For a *virtual* bus, simulation results and generated code are exactly the same as if the bus did not exist, which functionally it does not.

Generated code for a *nonvirtual* bus represents the bus data with a structure. The use of a structure in the generated code can be helpful when tracing the correspondence between the model and the code. For example, below is the generated code for 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

```

Choosing Between Virtual and Nonvirtual Buses

Whether to use a virtual or nonvirtual bus depends on your modeling goal. Frequently, model contain both virtual and nonvirtual buses.

Modeling Goal	Type of Bus
Use bus signals within a functional units, such as within a referenced model	Virtual
Use signals with different sample rates in a bus	Virtual (required)
Have bus data packaged as data structures in the generated code	Nonvirtual
Have bus data cross model reference, MATLAB Function block, or Stateflow chart boundaries	Nonvirtual

Not all blocks can accept buses. See “Bus-Capable Blocks” on page 51-22 for more about which blocks can handle which types of buses. Virtual buses are the default except where nonvirtual buses are explicitly required. See “Connect Buses to Inports and Outports” on page 51-65 for more information.

For additional guidelines for generated code for buses, see “About Buses and Code Generation”.

Creating Nonvirtual Buses

Bus signals do not specify whether they are virtual or nonvirtual; they inherit that specification from the block in which they originate. Every block that creates or requires a nonvirtual bus must have an associated bus object. Those blocks are:

- Bus Creator
- Inport
- Output
- Constant
- Data Store Memory

To specify that a bus is nonvirtual:

- 1** Associate the block with a bus object, as described in “Associating Bus Objects with Simulink Blocks” on page 51-25.
- 2** For the block that creates the bus, open the Block Parameters dialog box.
- 3** Set the data type to **Bus:** `<object name>` and replace `<object name>` with the bus object name.
 - For Inport, Output, and Data Store Memory blocks, use the **Data type** parameter to set the data type.
 - For Bus Creator and Constant blocks, use the **Output data type** parameter to set the data type.
- 4** For any Bus Creator, Inport, or Output blocks that have an associated bus object, enable the output as a nonvirtual bus.

- For Bus Creator and Inport blocks, use the **Output as nonvirtual bus** parameter.
- For Outport blocks, use the **Output as nonvirtual bus in parent model** parameter.

5 Click **OK** or **Apply**.

The **Signal Attributes** parameter is applicable only to root Inport and Outport blocks, and does not appear in the parameters of a subsystem Inport or Outport block. In a root Outport block, setting **Signal Attributes** > **Output as nonvirtual bus in parent model** specifies that the bus emerging in the parent model is nonvirtual. The bus that is input to the root Outport can be virtual or nonvirtual.

Nonvirtual Bus Sample Times

All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error.

All buses and signals input to a Bus Creator block that outputs a nonvirtual bus must therefore have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

Automatic Bus Conversion

When updating a diagram prior to simulation or code generation, the Simulink software automatically converts virtual buses to nonvirtual buses where the conversion is possible and prevents an error. For example, referenced models and Stateflow charts require any bus connected to them to be nonvirtual.

The conversion consists of inserting hidden Signal Conversion blocks into the model where needed.

The source block of any virtual bus that is converted to a nonvirtual bus, whether explicitly or automatically, must specify a bus object, as described in “Bus Objects” on page 51-24. Conversion to a nonvirtual bus fails if no

bus object is specified, and the Simulink software posts an error message describing the problem.

Explicit Bus Conversion

You can eliminate the need for the automatic conversion by using one of these approaches:

- Specify a nonvirtual bus in the block where the bus originates. For example, if you use a Bus Creator block to create a bus, then to specify a nonvirtual bus, set the Bus Creator block **Data type** parameter to use a `Simulink.Bus` object.
- Manually insert a Signal Conversion block. Using a Signal Conversion block can reduce memory usage, support modeling constructs such as Model blocks that require that input buses be nonvirtual, and reduce generated code. For details, see the Signal Conversion documentation.

Create and Access a Bus

The Signal Routing library provides three blocks that you can use for implementing buses:

Bus Creator

Create a bus that contains specified elements

Bus Assignment

Replace specified bus elements

Bus Selector

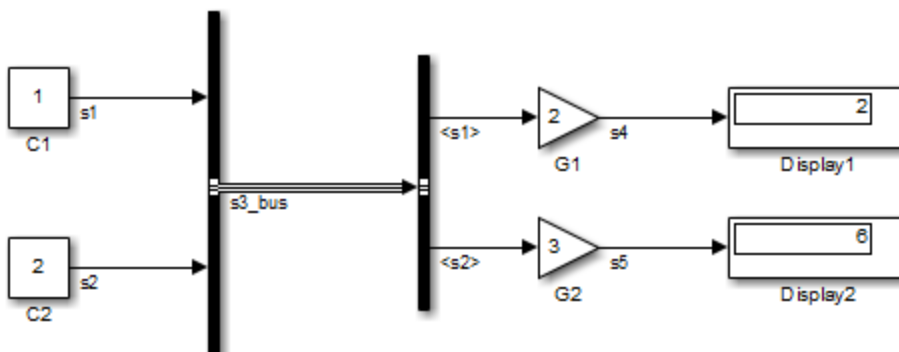
Select elements from a bus

Each of these blocks is virtual or nonvirtual depending on whether the bus that it processes is virtual or nonvirtual. The Simulink software chooses the block type, and changes it automatically if the bus type changes.

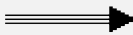

To create and access a bus signal that has default properties:

- 1 Clone a Bus Creator and Bus Selector block from the Signal Routing library.
- 2 Connect the Bus Creator, Bus Selector, and other blocks as needed to implement the desired bus.

The next figure shows two signals (s1 and s2) that are input to a Bus Creator block, transmitted as a bus signal (s3_bus) to a Bus Selector block, and output as separate signals.



The Bus Creator and Bus Selector blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks (s3_bus), representing the bus signal, is tripled because the model has been built, and the middle line is solid because the bus is virtual. The line would be dashed if the bus were nonvirtual:

Virtual Bus	
Nonvirtual Bus	

See “Signal Line Styles” on page 50-4 for more about the graphical appearance of signals. You can also display other signal characteristics graphically, as described under “Display Signal Attributes” on page 50-62.

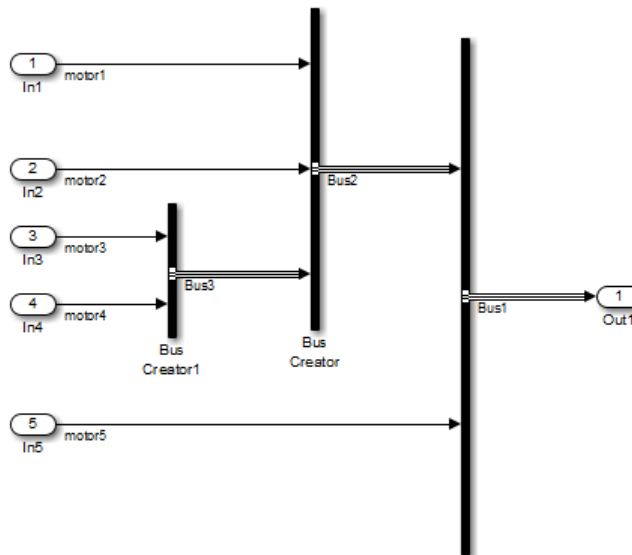
Simulink automatically labels the output signals of the Bus Selector block to reflect the name of the selected bus elements (for example, in the above model, s1 and s2).

For more information about creating and accessing buses, see the reference documentation for the Bus Creator, Bus Selector, and Bus Assignment blocks.

Nest Buses

You can nest buses to any depth. To create a nested bus, use a Bus Creator block. If one of the inputs to the Bus Creator block is a bus, then the output is a nested bus. To select a signal within a nested bus, use a Bus Selector block.

For example, in the next model, the Bus3 bus signal combines two signals, motor3 and motor4. The Bus2 signal combines the Bus3 bus signal and the motor1 and motor2 signals. The Bus1 signal combines the Bus2 bus signal and the motor5 signal.



All of the signals retain their separate identities, just as if no bus creation and selection occurred. You can use Bus Selector blocks to select individual signals from a nested bus.

The Simulink software automatically handles most of the complexities involved. For example, you can specify to have Simulink repair broken selections in the Bus Selector and Bus Assignment block parameter dialog boxes due to upstream bus hierarchy changes. To enable these automatic repairs, in the **Configuration Parameters > Diagnostics > Connectivity**

pane, set the **Repair bus selections** diagnostic to Warn and repair. The repairs occur when you update a model. To save the repairs, save the model.

Circular Bus Definitions

The ability to nest a bus as an element of another bus creates the possibility of a loop of Bus Creator blocks, Bus Selector blocks, and bus-capable blocks that inadvertently includes a bus as an element of itself. The resulting circular definition cannot be resolved and therefore causes an error.

The error message that appears specifies the location at which the Simulink software determined that the circular structure exists. The error is not really at any one location: the structure as a whole is in error. Nonetheless, the location cited in the error message can be useful for beginning to trace the definition cycle; its structure may not be obvious on visual inspection.

- 1** Begin by selecting a signal line associated with the location cited in the error message.
- 2** Choose **Highlight to Signal to Source** or **Highlight Signal to Destination** from the signal's context menu. (See "Display Signal Sources and Destinations" on page 50-42 for more information.)
- 3** Continue choosing signals and highlighting their sources and destinations until the cycle becomes clear.
- 4** Restructure the model as needed 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, in the way that an algebraic loop can be broken by inserting a Unit Delay block. No alternative exists but to restructure the model to eliminate the circular bus definition.

Bus-Capable Blocks

All virtual blocks are bus-capable. The following blocks are also bus-capable. Some blocks only support nonvirtual buses.

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant (nonvirtual buses)
- Data Store Memory (nonvirtual buses)
- Data Store Read (nonvirtual buses)
- Data Store Write (nonvirtual buses)
- From Workspace (nonvirtual buses)
- Memory
- Merge
- Multiport Switch
- Rate Transition
- Signal Conversion
- Signal Specification
- Switch
- To Workspace
- Unit Delay
- Zero-Order Hold

All signals in a nonvirtual bus input to a bus-capable block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus.

Some bus-capable blocks impose other constraints on bus propagation through them. See the documentation for specific blocks for more information.

Subsystems, model referencing and S-functions support the use of buses. See the documentation for those features for any special considerations relating to the use of buses.

Bus Objects

In this section...

“About Bus Objects” on page 51-24

“Bus Object Capabilities” on page 51-25

“Associating Bus Objects with Simulink Blocks” on page 51-25

About Bus Objects

The properties that you specify for a bus signal by using Bus Creator block parameters are inherited by all downstream blocks that use the bus. Using Bus Creator block parameters is adequate for defining virtual buses and performing limited error checking. However, to define a nonvirtual bus, or to perform complete error checking on any bus, use a *bus object* to specify additional information.

Creating a bus object establishes a composite data type whose name is the name of the bus object and whose properties are given by the object. 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 a bus. A bus object is an instance of class `Simulink.Bus` that is defined in the base workspace. A bus object serves as the root of an ordered hierarchy of *bus elements*, which are instances of class `Simulink.BusElement`. Each element completely specifies the properties of one signal in a bus: its name, data type, dimensionality, etc. The order of the elements contained in the bus object defines the order of the signals in the bus.

Referenced models, Stateflow charts, and MATLAB Function blocks that input and output buses require those buses to be defined with bus objects. Inport and Outport blocks can use bus objects to specify the structure of the bus passing through them. Root inport blocks use bus objects to specify the structure of the bus. Root outport blocks use bus objects to check the

structure of the incoming bus and to specify the structure of the bus in the parent model, if any.

Bus Object Capabilities

You can associate a bus object with several blocks. For details, see “Associating Bus Objects with Simulink Blocks” on page 51-25.

When a bus object governs a signal output by a block, the signal is a bus that has exactly the properties specified by the object. When a bus object governs a signal input by a block, the signal must be a bus that has exactly the properties specified by the object; any variance causes an error.

A bus object can also specify properties that were not defined by constituent signals, but were left to be inherited. A property specification in a bus object can either validate or provide the corresponding property in the bus. If the bus specifies a different property, an error occurs. If the bus does not specify the property, but leaves it to be inherited, the bus inherits the property from the bus object. Note again that such inheritance never includes signal values.

You can use the Simulink Bus Editor to create and manage bus objects, as described in “Manage Bus Objects with the Bus Editor” on page 51-28, or you can use the Simulink API, as described in “Bus Object API” on page 51-27. 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.

Associating Bus Objects with Simulink Blocks

You can associate a bus object with the following blocks:

- Bus Creator
- Data Store Memory
- Data Store Read
- Data Store Write
- From File
- From Workspace
- Inport

- Outport
- Permute Dimensions
- Probe
- Reshape
- Signal Conversion
- Signal Specification

In the Block Parameters dialog box for the block that you want to associate with a bus, set **Data type** to **Bus:** <object name> and replace <object name> with the bus object name.

Note Do not set the minimum and maximum values for bus data on blocks with bus object data type. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

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

Bus Object API

The Simulink software provides all Bus Editor capabilities programmatically. Many of these capabilities, like importing and exporting MATLAB code files and MAT-files, are not specific to bus objects, and are described elsewhere in the MATLAB and Simulink documentation.

The classes that implement bus objects are:

`Simulink.Bus`

Specify the properties of a signal bus

`Simulink.BusElement`

Describe an element of a signal bus

The functions that create and save bus objects are:

`Simulink.Bus.createObject`

Create bus objects for blocks, optionally saving them in a MATLAB file in a specified format

`Simulink.Bus.cellToObject`

Convert a cell array containing bus information to bus objects in the base workspace

`Simulink.Bus.objectToCell`

Convert bus objects in the base workspace to a cell array containing bus information

`Simulink.Bus.save`

Export specified bus objects or all bus objects from the base workspace to a MATLAB file in a specified format

`Simulink.Bus.createMATLABStruct`

Create MATLAB structure with same hierarchy, names, and attributes as the bus signal

In addition, when you use `Simulink.SubSystem.convertToModelReference` to convert an atomic subsystem to a referenced model, you can save any bus objects created during the conversion to a MATLAB file.

Manage Bus Objects with the Bus Editor

In this section...

“Introduction” on page 51-28
“Open the Bus Editor” on page 51-29
“Display Bus Objects” on page 51-30
“Create Bus Objects” on page 51-33
“Create Bus Elements” on page 51-36
“Nest Bus Definitions” on page 51-39
“Change Bus Entities” on page 51-42
“Export Bus Objects” on page 51-46
“Import Bus Objects” on page 51-48
“Close the Bus Editor” on page 51-48

Introduction

The Simulink Bus Editor is a tool similar to the Model Explorer, but is customized for use with bus objects. You can use the Simulink Bus Editor to:

- Create new bus objects and elements
- Navigate, change, and nest bus objects
- Import existing bus objects from a MATLAB code file or MAT-file
- Export bus objects to a MATLAB code file or MAT-file

For a description of bus objects and their use, see “Bus Objects” on page 51-24.

Base Workspace Bus Objects

All bus objects exist in the MATLAB base workspace. Bus Editor actions take effect in the base workspace immediately, and can be used by Simulink models as soon as each action is complete. The Bus Editor does not have a workspace of its own: it acts only on the base workspace. Bus Editor actions do not directly affect bus object definitions in saved MATLAB code files or

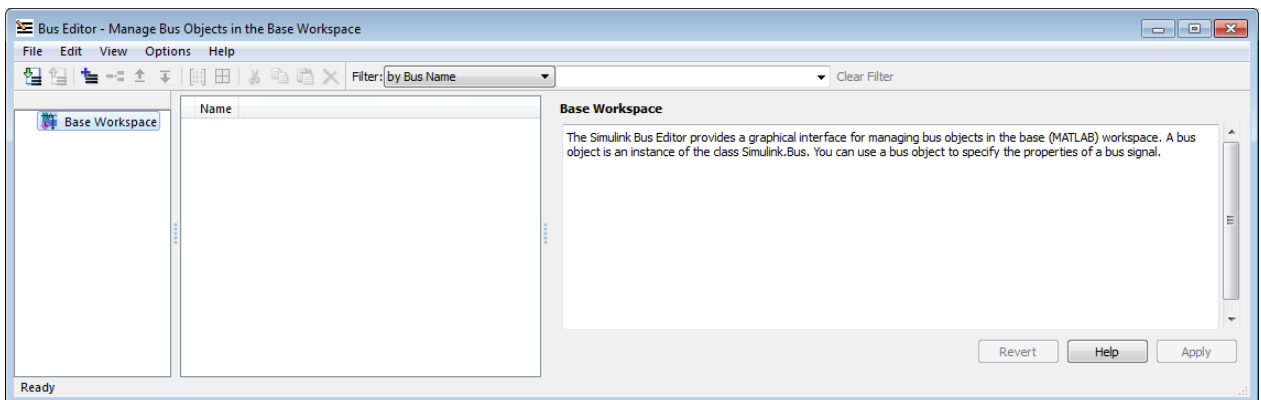
MAT-files. To save changed bus object definitions, export them from the base workspace into MATLAB code files or MAT-files, as described in “Export Bus Objects” on page 51-46.

Open the Bus Editor

You can open the Bus Editor in any of these ways:









- In the Simulink Editor, select **Edit > Bus Editor**.
- In a bus object’s dialog box in the Model Explorer, click the **Launch Bus Editor** button.
- Enter `buseditor` at the command line of the MATLAB software.

If no bus objects exist, the Bus Editor looks like the one shown here:



Bus Editor Commands

The Bus Editor provides menu choices that you can use to execute all Bus Editor commands. The editor also provides toolbar icons and keyboard shortcuts for all commonly used commands, including the standard MATLAB shortcuts for Cut, Copy, Paste, and Delete. The Toolbar Tip for each icon describes the command, and the menu entry for each command shows any shortcut. The icons for commands that are specific to the Bus Editor are:

Command	Icon	Description
Import		Import the contents of a MATLAB code file or MAT-file into the base workspace.
Export		Export all bus objects and elements to a MATLAB code file or MAT-file.
Create		Create a new bus object in the base workspace.
Insert		Add a bus element below the currently selected bus entity.
Move Up		Move the selected element up in the list of a bus object's elements.
Move Down		Move the selected element down in the list of bus object elements.
Create/Edit a Simulink.Parameter object		Create or edit a Simulink.Parameter object for the selected bus object.
Create a MATLAB structure		Create a MATLAB structure for the selected bus object.

You can use toolbar icons and keyboard shortcuts instead of menu commands whenever you prefer.

Display Bus Objects

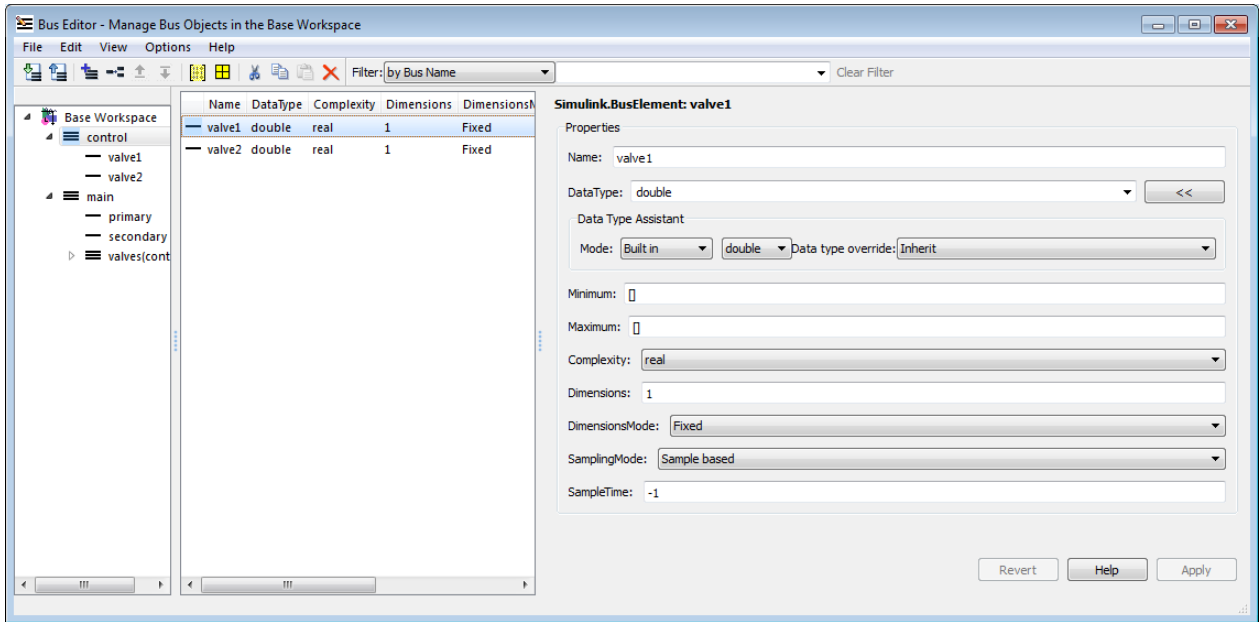
The Bus Editor is similar to the Model Explorer (which can display bus objects but cannot edit them) and uses the same three panes to display bus objects:

- **Hierarchy pane** (left) — Displays the bus objects defined in the base workspace
- **Contents pane** (center) — Displays the elements of the bus object selected in the Hierarchy pane
- **Dialog pane** (right) — Displays for editing the current selection in the Contents or Hierarchy pane

Items that appear in the Hierarchy pane or the Contents pane have Context menus that provide immediate access to the capabilities most likely to be useful with that item. The contents of an item's Context menu depend on the item and the current state within the Bus Editor. All Context menu options are also available from the menu bar and/or the toolbar. Right-click any item in the Hierarchy or Contents pane to see its Context menu.

Hierarchy Pane

If no bus objects exist in the base workspace, the Hierarchy pane shows only **Base Workspace**, which is the root of the hierarchy of bus objects. The Bus Editor then looks as shown in the previous figure. As you create or import bus objects, they appear in the Hierarchy Pane as nodes subordinate to **Base Workspace**. The bus objects appear in alphabetical order. The next figure shows the Bus Editor with two bus objects, `control` and `main`, defined in the base workspace:



The Hierarchy pane displays each bus object as an expandable node. The root of the node displays the name of the bus object, and (if the bus contains any elements) a button for expanding and collapsing the node. Expanding a bus node displays named subnodes that represent the bus's top-level elements. In the preceding figure, both bus objects are fully expanded, and control is selected.

Contents Pane

Selecting any top-level bus object in the Hierarchy Pane displays the object's elements in the Contents pane. In the previous figure, the elements of bus object control, valve1 and valve2, appear. Each element's properties appear to the right of the element's name. These properties are editable, and you can edit the properties of multiple elements in one operation, as described in "Editing in the Contents Pane" on page 51-43.

Dialog Pane

When a bus object is selected in the Hierarchy pane, or a bus object or element is selected in the Contents pane, the properties of the selected item appear in

the Dialog pane. In the previous figure, `valve1` is selected in the Contents pane, so the Dialog pane shows its properties. These properties are editable, and changes can be reverted or applied using the buttons below the Dialog pane, as described in “Editing in the Dialog Pane” on page 51-45.

Filter Boxes

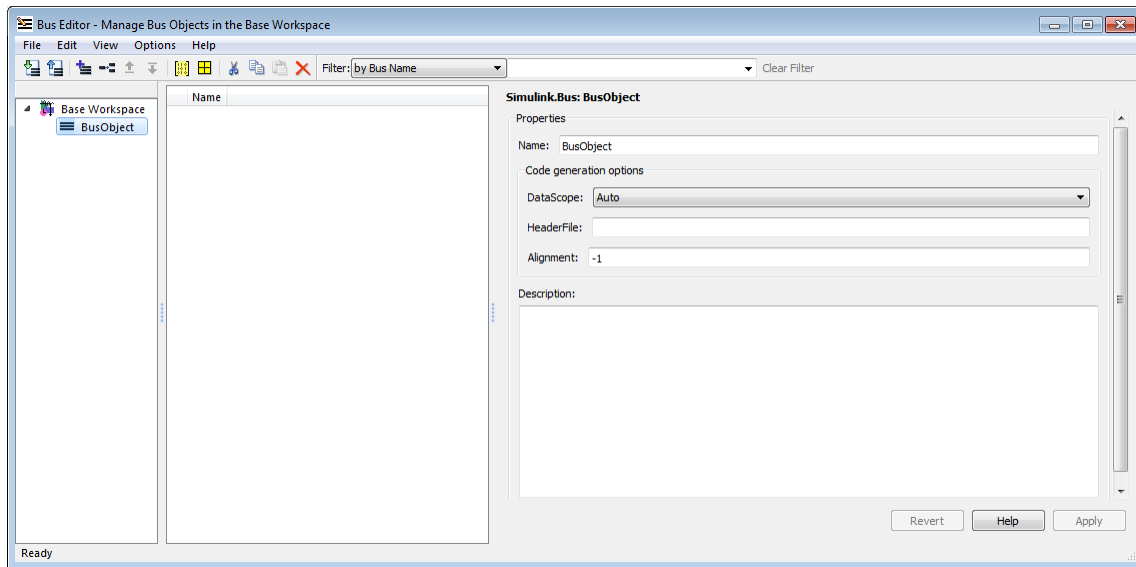
By default, the Bus Editor displays all bus objects that exist in the base workspace. Where a model contains large numbers of bus objects, seeing them all at the same time can be inconvenient. To facilitate working efficiently with large collections of bus objects, you can use the **Filter** boxes, to the right of the iconic tools in the toolbar, to show a selected subset of bus objects. See “Filter Displayed Bus Objects” on page 51-53 for details.

Create Bus Objects

To use the Bus Editor to create a new bus object in the base workspace:

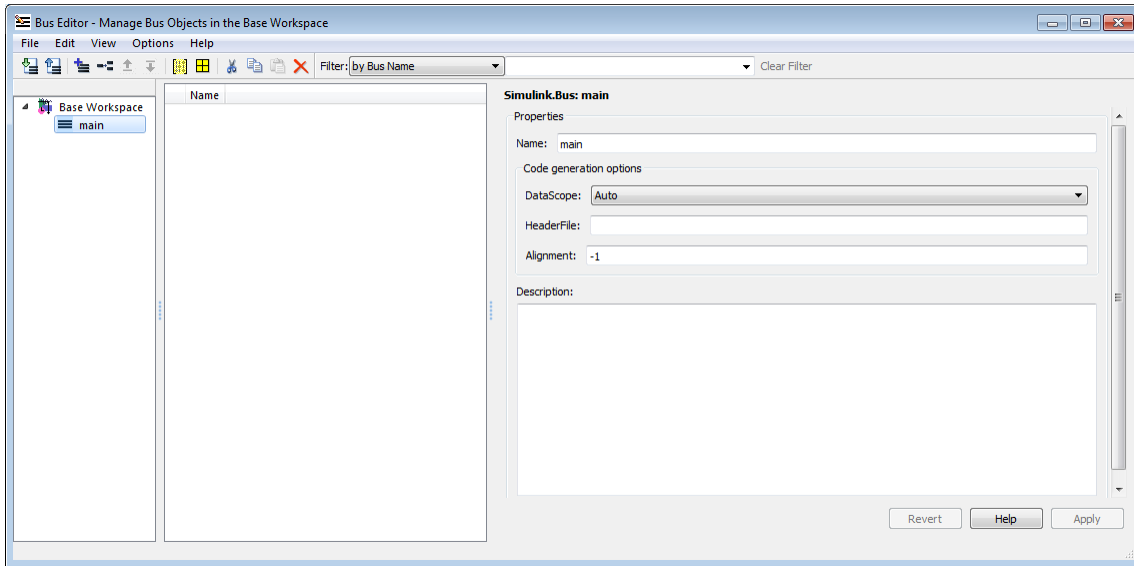
1 Choose **File > Add Bus**.

A new bus object with a default name and properties is created immediately in the base workspace. The object appears in the Hierarchy pane, and its default properties appear in the Dialog pane:

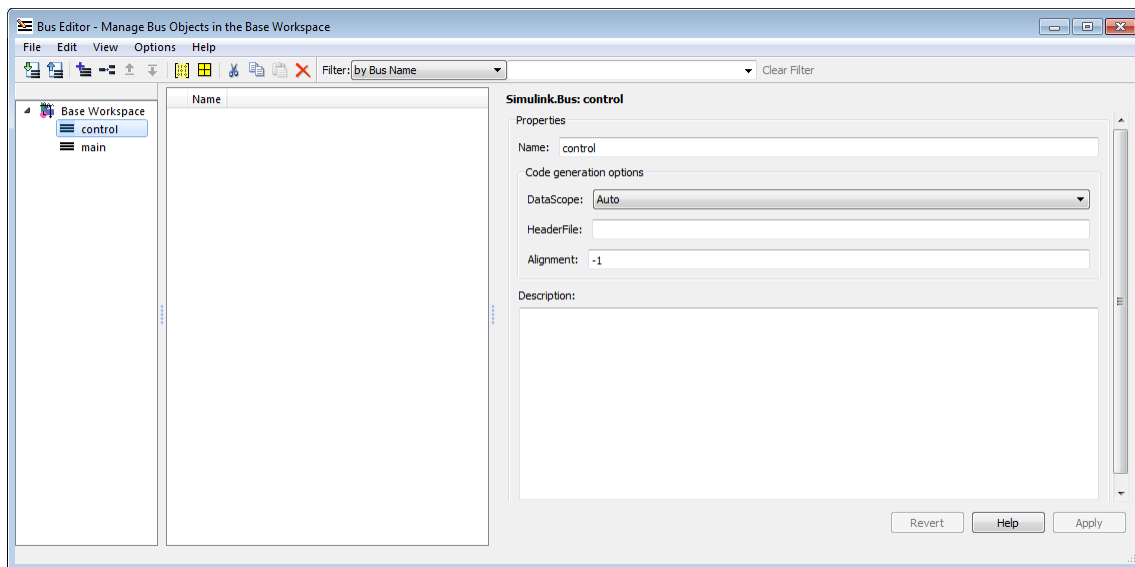


- 2 To specify the bus object name and other properties, in the Dialog pane:
 - a Specify the **Name** of the of the new bus object (or you can retain the default name). The name must be unique in the base workspace. See “Choosing a Signal Name” on page 50-19 for guidelines for signal names.
 - b Optionally, specify a **C Header file** that defines a user-defined type corresponding to this bus object. This header file has no effect on Simulink simulation; it is used only by Simulink Coder software to generate code.
 - c Optionally, specify a **Description** that provides information about the bus object to human readers. This description has no effect on Simulink simulation; it exists only for human convenience.
- 3 Click **Apply**.

The properties of the bus object on the base workspace change as specified. If you rename BusObject to main, the Bus Editor looks like this:



You can use **Add Bus** at any time to create a new bus object in the base workspace, then set the name and properties of the object as needed. You can intersperse creating bus objects and specifying their properties in any order. The hierarchy pane reorders as needed to display all bus objects in alphabetical order. If you add an additional bus object named `control`, the Bus Editor looks like this:



You can also use capabilities outside the Bus Editor to create new bus objects. Such objects do not appear in the Bus Editor until the next time its window is selected.

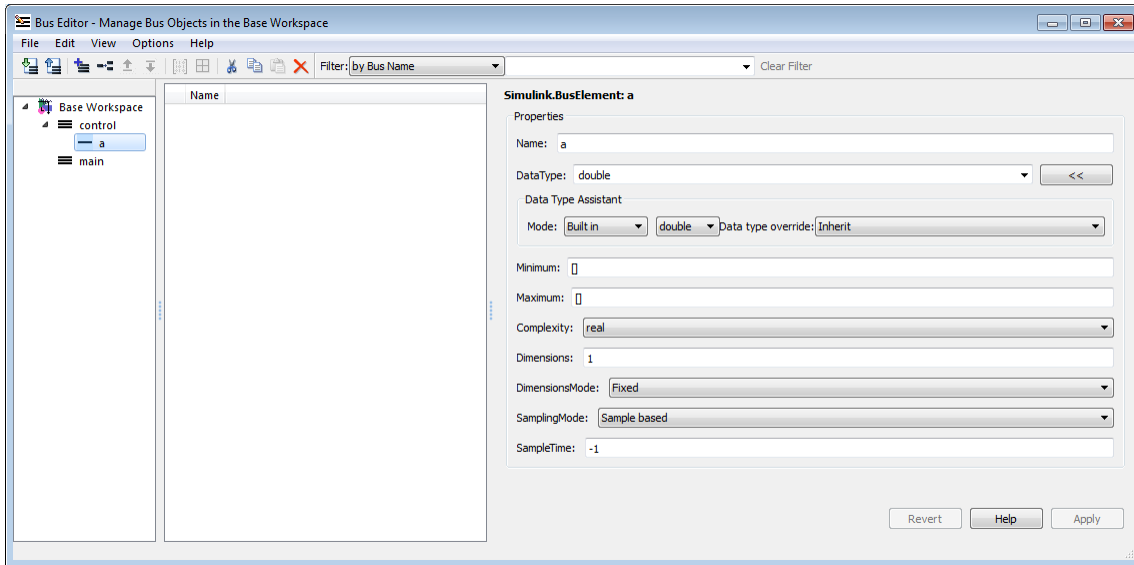
Create Bus Elements

Every bus element belongs to a specific bus. To create a new bus element:

- 1 In the Hierarchy pane, select the entity below which to create the new element. The entity can be a bus or a bus element. The new element will belong to the selected bus object, or to the bus object that contains the selected element. The previous figure shows the `control` bus object selected.

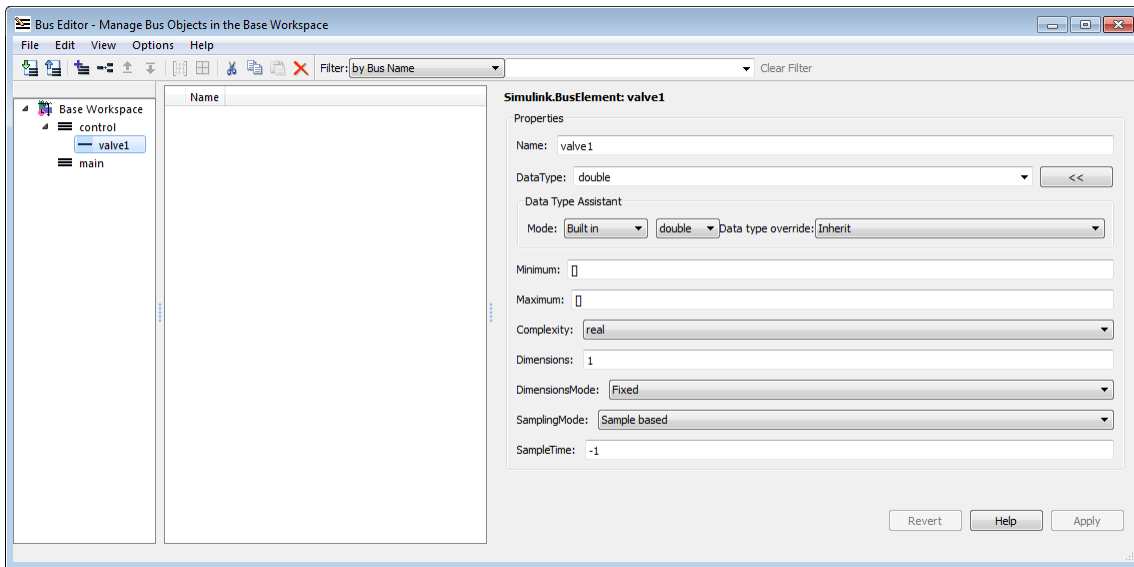
- 2 Choose **File > Add/Insert BusElement**.

A new bus element with a default name and properties is created immediately in the applicable bus object. The object appears in the Hierarchy pane immediately below the previously selected entity, and its default properties appear in the Dialog pane:

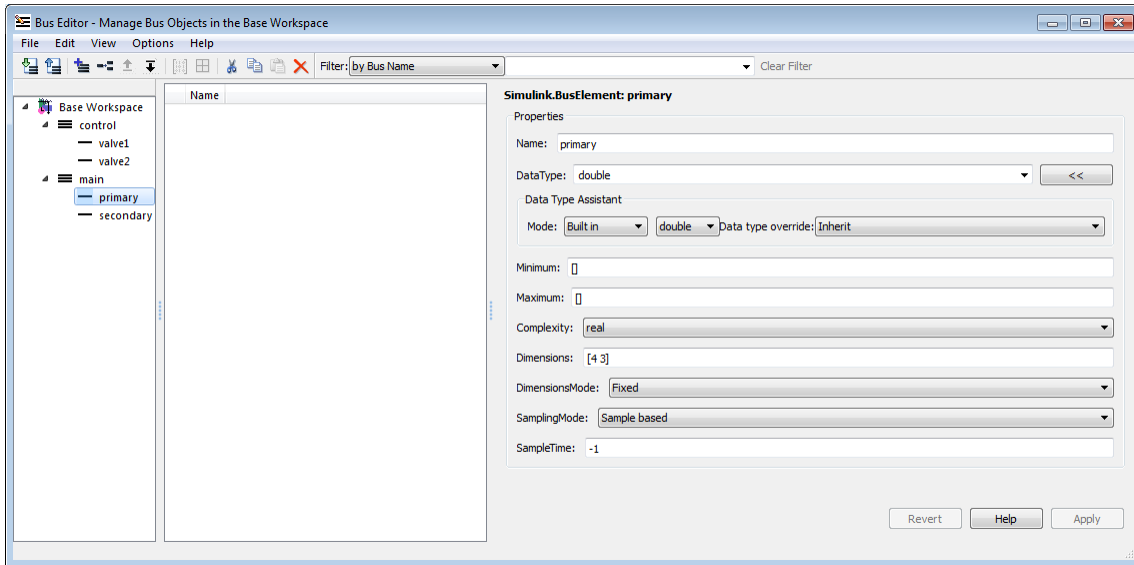


- 3 To specify the bus element name and other properties, in the Dialog pane:
 - a Specify the **Name** of the of the new bus element (or you can retain the default name). The name must be unique among the elements of the bus object. See “Signal Names” on page 50-19 for guidelines for signal names.
 - b Specify the other properties of the element. The properties must match the properties of the corresponding signal within the bus exactly, and can be anything that a legal signal might have. The Data Type Assistant appears in the Dialog pane to help specify the element’s data type. You can specify any available data type, including a user-defined data type.
- 4 Click **Apply**.

The properties of the bus element of the bus object in the base workspace change as specified. If you rename the new element a to valve1, the Bus Editor looks like this:



You can use **Add/Insert BusElement** at any time to create a new bus element in any bus object. You can intersperse creating bus objects and specifying their properties in any order. The order of the other bus elements in the bus object does not change when a new element is added. If you add element valve2 to control, and secondary and primary to main, the Bus Editor looks like this:



You can also use capabilities outside the Bus Editor to add new bus elements to a bus object. Such an addition changes an existing bus object, so any new bus element appears immediately in the Bus Editor.

Nest Bus Definitions

As described in “Nest Buses” on page 51-20, any signal in a bus can be another bus, which can in turn contain subordinate buses, and so on to any depth. Describing nested buses with bus objects requires nesting the bus definitions that the objects provide.

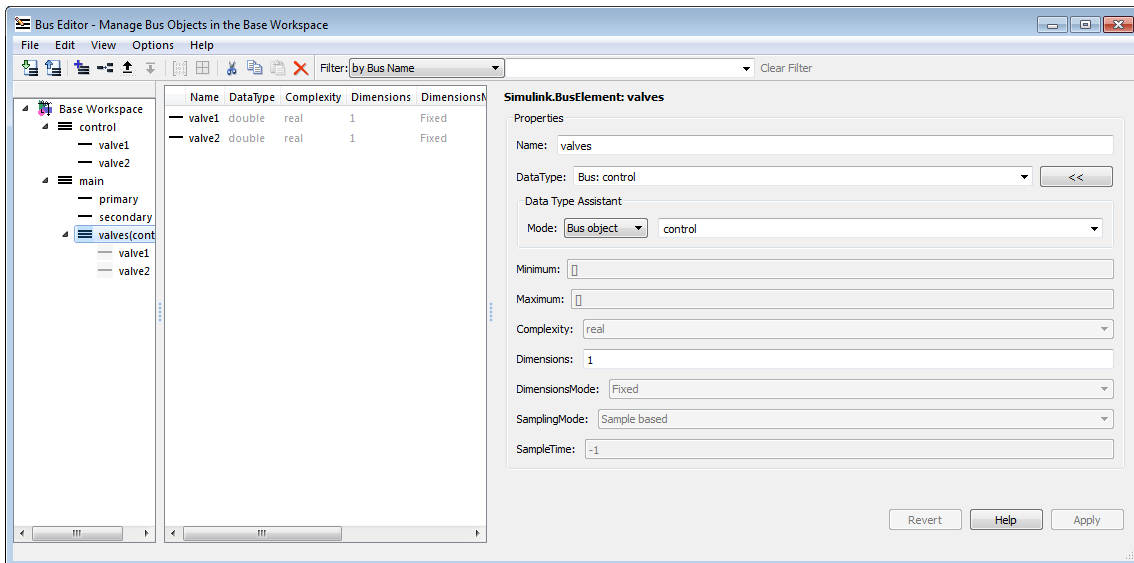
Every bus object inherently defines a data type whose properties are those specified by the object. To nest one bus definition in another, you assign to an element of one bus object a data type that is defined by another bus object. The element then represents a nested bus whose structure is given by the bus object that provides its data type.

A data type defined by a bus object is called a *bus type*. Nesting buses by assigning bus types to elements, rather than by subordinating the bus objects that define the types, allows the same bus definition to be used conveniently

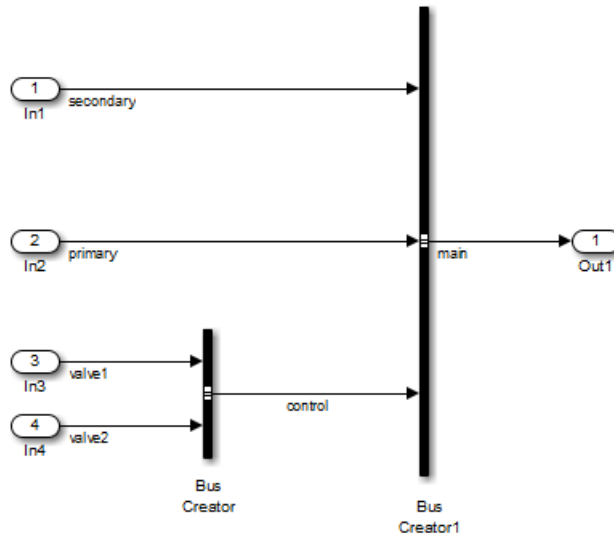
in multiple contexts without unwanted interactions. To specify that an element of a bus object represents a nested bus definition:

- 1 Create a bus element to represent the nested bus definition, in the appropriate position under the containing bus object, and give the element the desired name. (You can also use an existing element.)
- 2 Use the Dialog pane to set the data type of the element to the name of a bus object. The Data Type Assistant shows the names of all available bus types. (You can also specify a nonexistent bus type and define the object later.)

In the preceding figure, if you add to bus object main a third element named valves, set the data type of valves to be control (the name of the other defined bus object) and expand the new element valves, the Bus Editor looks like this:



The bus object main shown in the Bus Editor now defines the same structure used by the bus signal main in the next figure:



The distinction between a bus object and the bus type that it defines can be useful for initially understanding how nested bus objects work and how the Bus Editor handles them. In other contexts, the distinction is mostly an implementation detail, and describing bus objects themselves as being nested is more convenient. The rest of this chapter follows that convention.

You can nest a bus object in as many different bus objects as desired, and as many times in the same bus object as desired. You can nest bus objects to any depth, but you cannot define a circular structure by directly or indirectly nesting a bus object within itself.

If you try to define a circular structure, the Bus Editor posts a warning and sets the data type of the element that would have completed the cycle to **double**. Click **OK** to dismiss the Notice and continue using the editor.

You can use the Hierarchy pane to explore nested bus objects by expanding the objects, but you cannot change any property of a bus object anywhere that it appears in nested form. To change the properties of a nested bus object, you must change the source object, which is accessible at the top level in the Hierarchy pane. You can jump from a nested bus object to its source

object by selecting the nested object and choosing **Go to 'element'** from its Context menu.

Change Bus Entities

You can use the Bus Editor to change and delete existing bus objects and elements at any time. All three panes allow you to change the entities that they display. Changes that create, reorder, or delete entities take effect immediately in the base workspace. Changes to properties take effect when you apply them, or can be canceled, leaving the properties unchanged. The Bus Editor does not provide an Undo capability.

The Bus Editor provides comprehensive GUI capabilities for changing bus entities. You can Cut, Copy, and Paste within and between panes in any way that has a legal result. The Hierarchy and Dialog panes provide a Context menu for the current selection. Pasting a Copied entity always creates a copy, as distinct from a pointer to the original. The Bus Editor automatically changes names when needed to avoid duplication.

Changes made outside the bus editor can affect the information on display within it. Any change to an existing bus object or bus element is visible immediately in the editor. Any change that creates or deletes a bus object becomes visible in the bus editor next time its window is selected.

Editing in the Hierarchy pane

You can select the root node **Base Workspace** and perform various operations, like export, cut, copy, paste, and delete. The operation simultaneously affects all bus objects displayed in the Hierarchy pane, but does not affect any that are invisible because a filter is in effect. See “Filter Displayed Bus Objects” on page 51-53 for details.

As you use the Bus Editor, the Hierarchy pane automatically reorders the bus objects it displays to maintain alphabetical order. This behavior cannot be changed. However, the elements under a bus object can appear in any order. To change that order, cut and paste elements as needed, or move elements up and down as follows:

- 1 Select the element to be moved.

2 Choose **Edit > Move Element Up** or **Edit > Move Element Down**.

You cannot Paste one bus object under another to create a nested bus object specification. To specify a nested bus, you must change the data type of a bus element to be the type of an existing bus object, as described in “Nest Buses” on page 51-20.

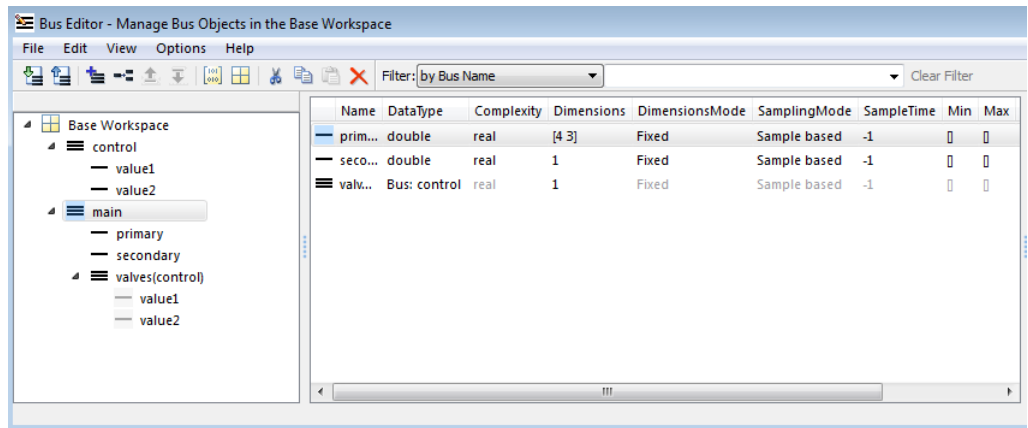
Editing in the Contents Pane

Selecting any top-level bus object in the Hierarchy pane displays the object’s elements in the Contents pane. Each element’s properties appear to the right of the element’s name, and can be edited. To change a property displayed in the Contents pane, click the value, enter a new value, then press **Return**.

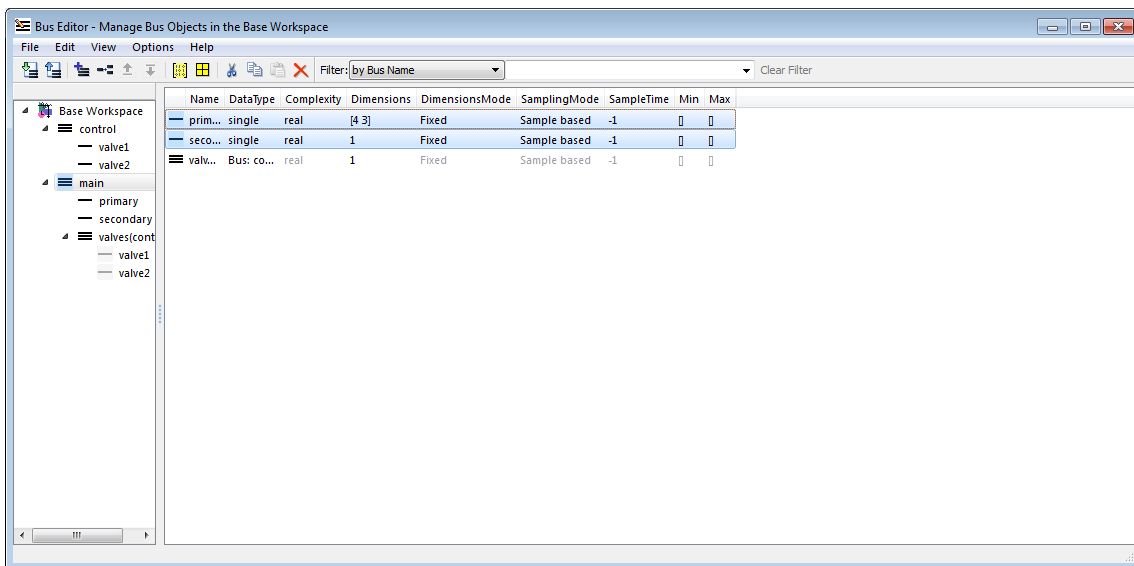
Choose **View > Dialog View** to hide the Dialog pane to provide more room to display properties in the Contents pane. Choose the command again to redisplay the Dialog pane.

You can use the mouse and keyboard to select multiple elements in the Contents pane. The selected entities need not be contiguous. You can then perform any operation that you could on a single entity selected in the pane, including operations performed with the Context menu. Clicking and editing a value in any selected element changes that value in them all.

The next figure shows the Bus Editor with **Dialog View** enabled, two elements selected in the Contents pane, and the **Data Type** property selected for editing in the second element:

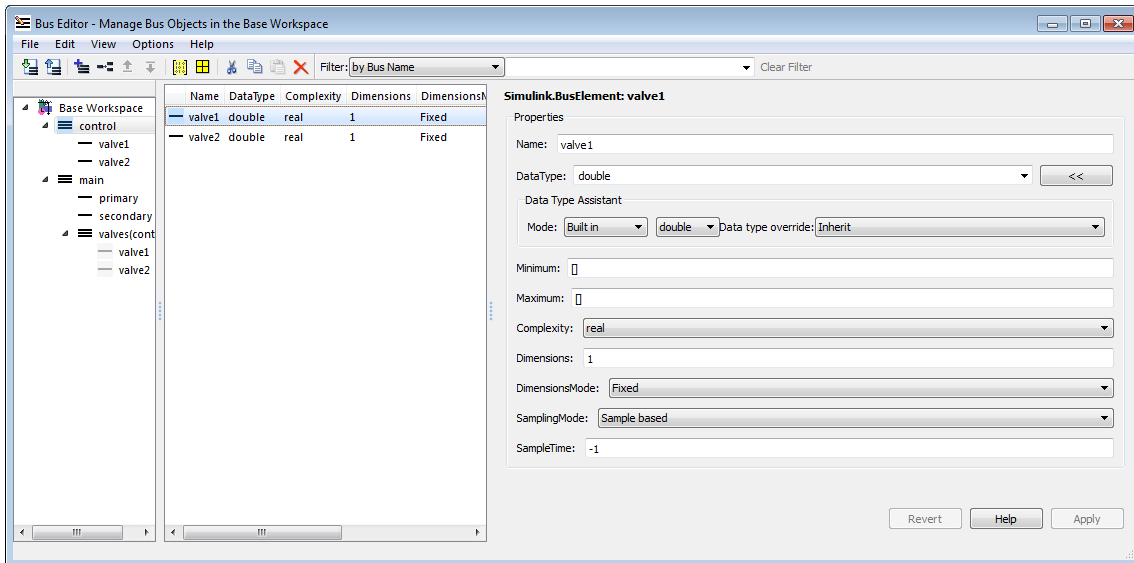


If you change the value of **DataType** to **single** and press Return, the value changes for both elements. The effect is the same no matter which element you edit in a multiple selection:

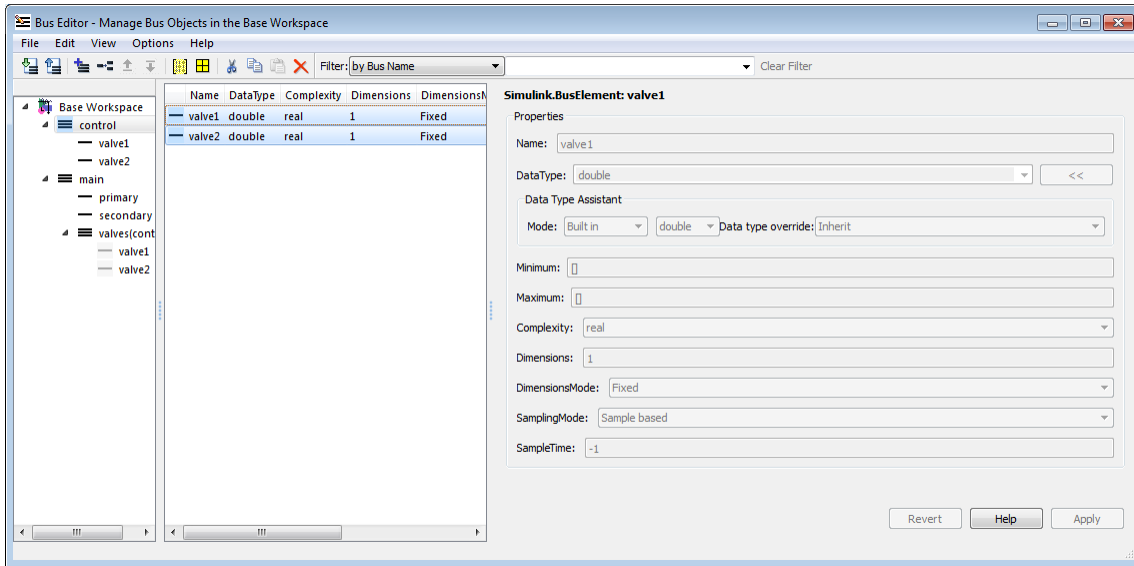


Editing in the Dialog Pane

When a bus object is selected in the Hierarchy pane, or a bus object or element is selected in the Contents pane, the properties of the selected item appear in the Dialog pane. In the next figure, `valve1` is selected in the Contents pane, so the Dialog pane shows its properties:



The properties shown in the Dialog pane are editable, and the pane includes the Data Type Assistant. Click **Apply** to save changes, or **Revert** to cancel them and restore the values that existed before any unapplied changes. You can edit only one element at a time in the Dialog pane. If multiple entities are selected in the Contents pane, all fields in the Dialog pane are grayed out:



If you use the Dialog pane to change any property of a bus entity, then navigate elsewhere without clicking either **Apply** or **Revert**, a query box appears by default. The query box asks whether to apply changes, ignore changes, or continue as if the navigation had not been tried. You can suppress this query for future operations by checking **Never ask me again** in the box, or by selecting **Options > Auto Apply/Ignore Dialog Changes**.

If you suppress the query, and thereafter navigate away from a change without clicking **Apply** or **Revert**, the Bus Editor automatically applies or discards changes, depending on which action you most recently chose in the box. You can re-enable the query box for future operations by deselecting **Options > Auto Apply/Ignore Dialog Changes**.

Export Bus Objects

Like all base workspace objects, bus objects are not saved with a model that uses them, but exist separately in a MATLAB code file or MAT-file. You can use the Bus Editor to export some or all bus objects to either type of file.

- If you export bus objects to a MATLAB code file, the Bus Editor asks whether to store them in object format or cell format (the default). Specify the desired format.
- If exporting would overwrite an existing MATLAB code file or MAT-file, a confirmation dialog box appears. Confirm the export or cancel it and try a different filename.

To export all bus objects from the base workspace to a file:

- 1** In the Bus Editor, choose **File > Export to File**.

The Export dialog box appears.

- 2** Specify the desired name and format of the export file.
- 3** Click **Save**.

All bus objects, and nothing else, are exported to the specified file in the specified format.

To export only selected bus objects from the base workspace to a file:

- 1** Select a bus object in the Hierarchy pane, or one or more bus objects in the Contents pane.
- 2** Right-click to display the Context menu.
- 3** Choose **Export to File** to export only the selected bus objects, or **Export with Related Bus Objects to File** to also export any nested bus objects used by the selected objects.
- 4** Use the Export dialog box to export the selected bus object(s).

Clicking the **Export** icon in the toolbar is equivalent to choosing **File > Export**, which exports all bus objects whether or not any are selected.

Customizing Bus Object Export

You can customize bus object export by providing a custom function that writes the exported objects to something other than the default destination, a MATLAB code file or MAT-file stored in the file system. For example, the

exported bus objects could be saved as records in a corporate database. See “Customize Bus Object Import and Export” on page 51-59 for details.

Import Bus Objects

You can use the Bus Editor to import the definitions in a MATLAB code file or MAT-file to the base workspace. Importing the file imports the complete contents of the file, not just any bus objects that it contains. If you import a file not exported by the Bus Editor, be careful that it does not contain unwanted definitions previously exported from the base workspace or created programmatically.

To import bus objects from a file to the base workspace:

- 1** Choose **File > Import into Base Workspace**.
- 2** Use the Open File dialog box to navigate to and import the desired file.

Before importing the file, the Bus Editor posts a warning that importing the file will overwrite any variable in the base workspace that has the same name as an entity in the file. Click **Yes** or **No** as appropriate. The imported bus objects appear immediately in the editor. You can also use capabilities outside the Bus Editor to import bus objects. Such objects do not appear in the Bus Editor until the next time its window becomes the current window.

Customizing Bus Object Import

You can customize bus object import by providing a custom function that imports the objects from something other than the default source, a MATLAB code file or MAT-file stored in the file system. For example, the bus objects could be retrieved from records in a corporate database. See “Customize Bus Object Import and Export” on page 51-59 for details.

Close the Bus Editor

To close the Bus Editor, choose **File > Close**. Closing the Bus Editor neither saves nor discards changes to bus objects, which remain unaffected in the base workspace. However, if you also close MATLAB without saving changes to bus objects, the changes will be lost. To save bus objects without saving other base workspace contents, use the techniques described in “Export Bus

Objects” on page 51-46. You can also save bus objects using any MATLAB technique that saves the contents of the base workspace, but the resulting file will contain everything in the base workspace, not just bus objects.

You can configure the Bus Editor so that closing it posts a reminder to save bus objects. To enable the reminder, select **Options > Always Warn Before Closing**. When this option is selected and you try to close the Bus Editor, a reminder appears that asks whether the editor should save bus objects before closing. Click **Yes** to save bus objects and close, **No** to close without saving bus objects, or **Cancel** to dismiss the reminder and continue in the Bus Editor. You can disable the reminder by deselecting **Options > Always Warn Before Closing**.

Store and Load Bus Objects

In this section...
“MATLAB Code Files” on page 51-50
“MATLAB Data Files (MAT-Files)” on page 51-51
“Database or Other External Source Files” on page 51-51

You can store bus object objects in a variety formats.

Format	When to Use
MATLAB code file	For traceability and model differencing using MATLAB code
MATLAB data file (MAT-file)	For faster bus loading and saving
Database or other external data source	For comparing bus interface information with design documents stored in an external data source.

MATLAB Code Files

You can read and save bus data with MATLAB code files.

To save all bus objects (instances of the `Simulink.Bus` class) in the MATLAB base workspace to a MATLAB code file, use *one* of the following approaches:

- Use the Bus Editor (see “Export Bus Objects” on page 51-46 and “Import Bus Objects” on page 51-48).
- From the MATLAB command line, use the `Simulink.Bus.save` command.

To save variables from the base workspace to a MATLAB code file, use the `Simulink.saveVars` command. The file containing the variables is formatted to be easily understood and editable. Running the file restores the saved variables to the base workspace.

For traceability, consider using a clearly named separate file for each model.

MATLAB Data Files (MAT-Files)

You can load bus objects in MATLAB data files (MAT-files), using *one* of the following approaches:

- Use the Bus Editor (see “Export Bus Objects” on page 51-46 and “Import Bus Objects” on page 51-48).
- From the MATLAB command line, use the `load` command.

Loading large data from MAT-files is faster than loading from MATLAB code files.

Database or Other External Source Files

You can capture bus interface information in a database or other external source, and use scripts and Database Toolbox™ functionality to read that information into MATLAB.

You can use `sl_customization.m` to customize the Bus Editor to import bus data from a database or other external source. For details, see “Customize Bus Object Import and Export” on page 51-59.

Map Bus Objects to Models

As models become complex, you need to keep track of which models use which bus objects. From any model or bus object, you should be able to tell what component it needs or is needed by.

A model must load all of its bus objects before you execute the model. For automation and consistency across models, it is important to map bus objects to models.

- By identifying all of the bus objects a model needs, 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 that bus object.

To map bus objects to models, consider:

- Using Simulink Projects. For details, see “Project Management”
- Capturing the mapping information in an external data source, such as a database.

Use a Rigorous Naming Convention

Using a rigorous and standard naming convention for bus mapping information is a straightforward approach. For example, consider the model and data required for an actuator control function. You could name the model `Actuator` and name the input and output ports `Actuator_bus_in` and `Actuator_bus_out`, respectively. This naming convention makes it clear to what models a particular bus object is related, and vice versa.

Note that this approach can cause issues if the output from one model reference is fed directly to another model reference. In this case, the naming mismatch results in an error.

Filter Displayed Bus Objects

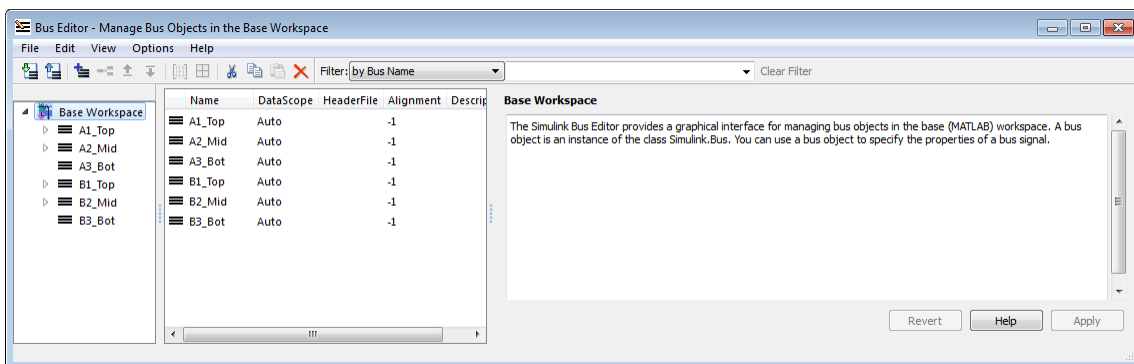
In this section...

- “Filter by Name” on page 51-54
- “Filter by Relationship” on page 51-55
- “Change Filtered Objects” on page 51-57
- “Clear the Filter” on page 51-58

By default, the Bus Editor displays all bus objects that exist in the base workspace, always in alphabetical order. When a model contains large numbers of bus objects, seeing them all at the same time can be inconvenient. To facilitate working efficiently with large collections of bus objects, you can set the Bus Editor to display only bus objects that:

- Have names that match a given string or regular expression
- Have a specified relationship to a bus object specified by name

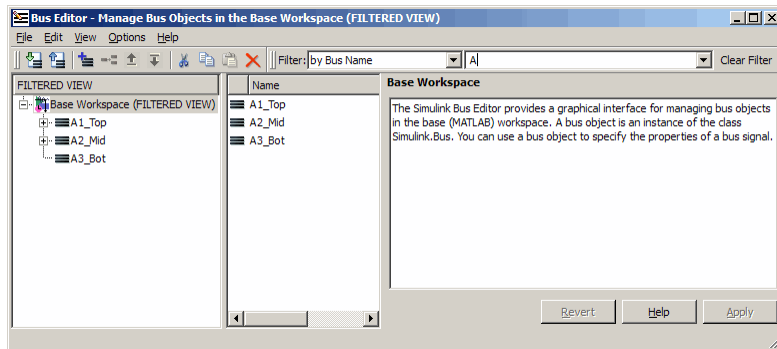
To set a filter, you specify values in the **Filter** boxes to the right of the tools in the toolbar. The left **Filter** box specifies the type of filtering. This box always appears, and is called the **Filter Type** box. Depending on the specified type of filtering, one or two boxes appear to the right of the **Filter Type** box. The next figure includes the **Filter** boxes (near the top of the dialog box) and shows that six bus objects exist in the Base Workspace:



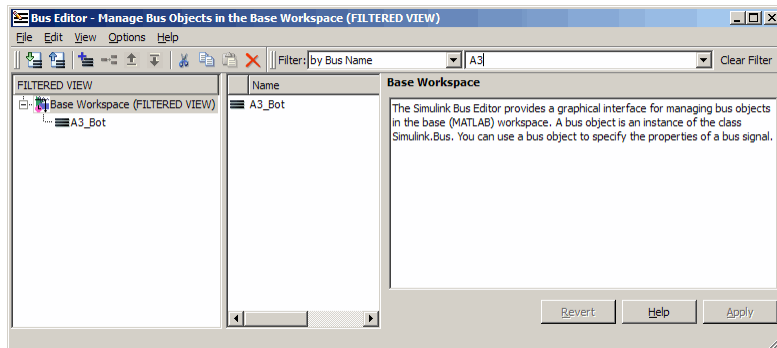
The bus objects shown form two disjoint hierarchies. A1_Top is the parent of A2_Mid, which is the parent of A3_Bot. Similarly, B1_Top > B2_Mid > B3_Bot. See “Nest Buses” on page 51-20 for information about bus object hierarchies.

Filter by Name

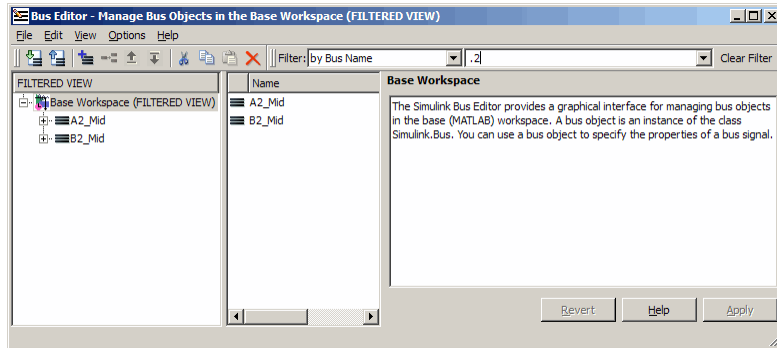
To filter bus objects by name, set the **Filter Type** box to by **Bus Name** (which is the default). The right **Filter** box is the **Object Name** box. Type any MATLAB regular expression (which can just be a string) into the **Object Name** box. As you type, the Bus Editor updates dynamically to show only bus objects whose names match the expression you have typed. The comparison is case-sensitive. For example, entering A displays:



Note that **FILTERED VIEW** appears in three locations, as shown in the preceding figure. This indicator appears whenever any filter is in effect. Entering the additional character 3 into the **Object Name** box displays:



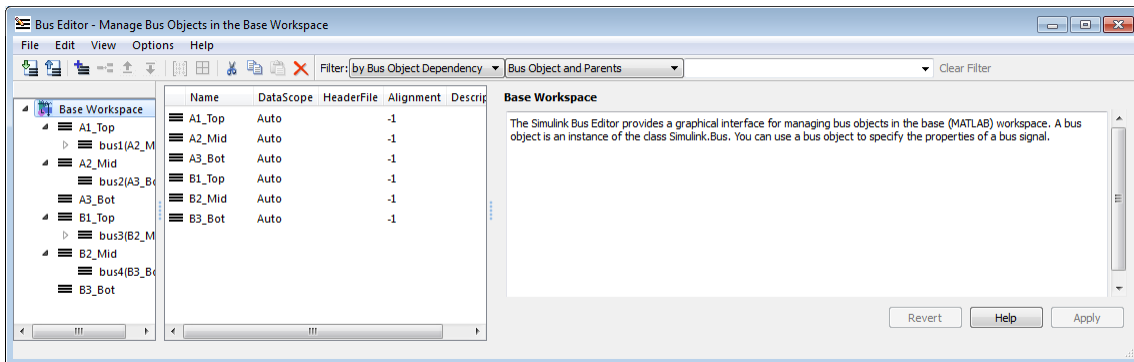
In a MATLAB regular expression, the metacharacter dot (.) matches any character, so entering .2 displays:



See “Regular Expressions” for complete information about MATLAB regular expression syntax.

Filter by Relationship

To filter bus objects by relationship, set the **Filter Type** box to by Bus Object Dependency. A third **Filter** box, called the **Relationship** box, appears between the **Filter Type** box and the **Object Name** box. You may have to widen the Bus Editor to see all three boxes:



In the **Relationship** box, select the type of relationship to display. The options are:

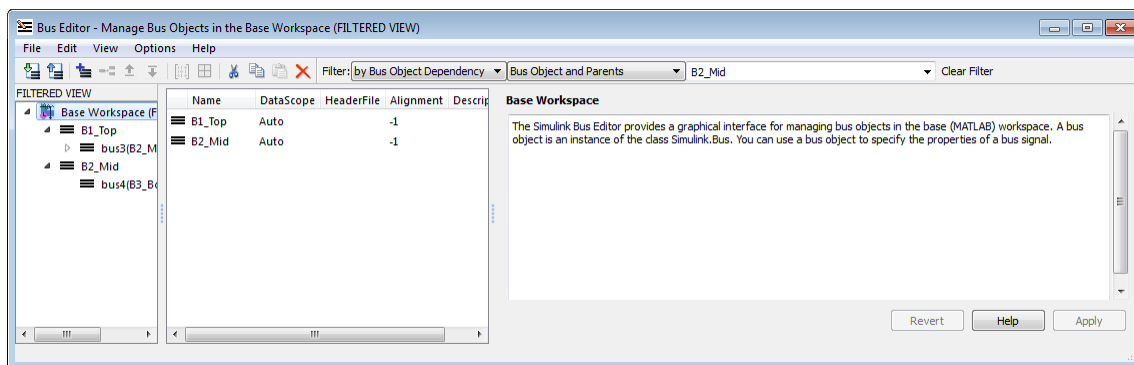
- **Bus Object and Parents** — Show a specified bus object and all superior bus objects in the hierarchy (default)
- **Bus Object and Dependents** — Show a specified bus object and all subordinate bus objects in the hierarchy
- **Bus Object and Related Objects** — Show a specified bus object and all superior and subordinate bus objects

In the **Object Name** box, specify a bus object by name. You can use the list to select any existing bus object name, or you can type a name. As you type, the editor:

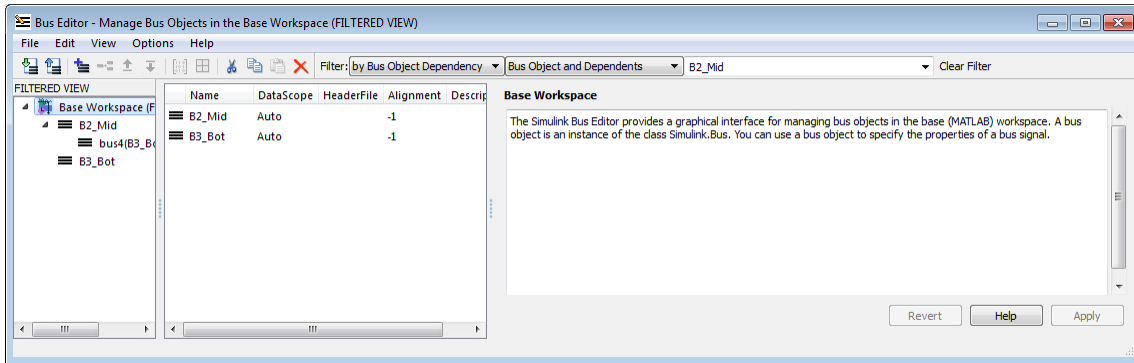
- Dynamically completes the field to indicate the first bus object that alphabetically matches what you have typed.
- Updates the display panes to show only that object and any objects that have the specified relationship to it.

When filtering by relationship, you must enter a string, not a regular expression, in the **Object Name** box. The match is case-sensitive. For example, assuming that **A1_Top** is the parent of **A2_Mid**, which is the parent of **A3_Bot** (as previously described) if you enter **B2** (or any leftmost substring that matches only **B2_Mid**) in the **Object Name** box, the Bus Editor displays the following for each of the three choices of relationship type:

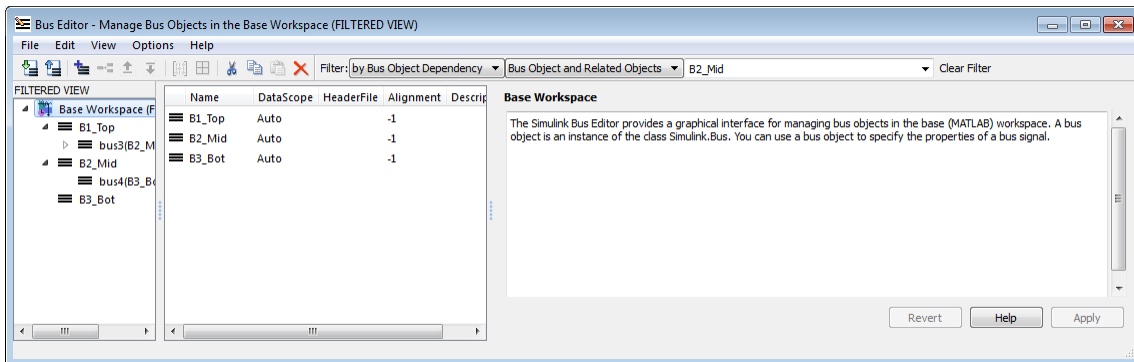
Bus Object and Parents



Bus Object and Dependents



Bus Object and Related Objects



Note that **FILTERED VIEW** appears in each the preceding figures, as it does when any filter is in effect.

Change Filtered Objects

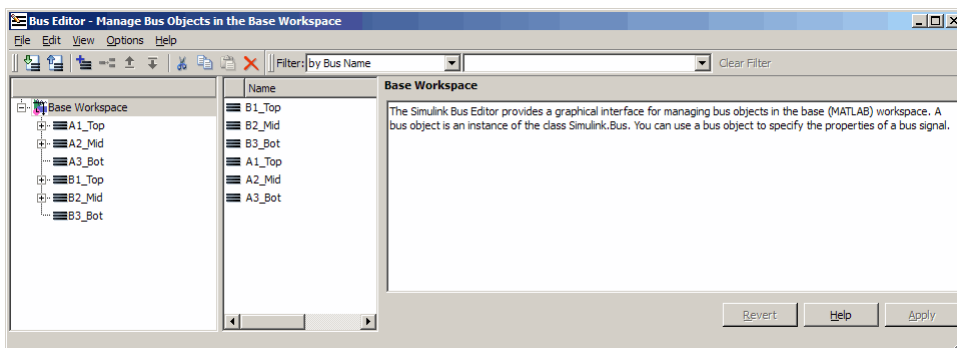
You can work with any bus object that is visible in a filtered display exactly as you could in an unfiltered display. If you change the name or dependency of an object so that it no longer passes the current filter, the object vanishes from the display. Conversely, if some activity outside the Bus Editor changes a filtered object so that it passes the current filter, the object immediately becomes visible.

A new bus object created within the Bus Editor with a filter in effect may or may not appear, depending on the filter. If you create a new bus object but do not see it in the editor, check the filter. The new object (whose name always begins with `BusObject`) may exist but be invisible. Bus objects created or imported using capabilities outside the Bus Editor are not visible until the Bus Editor window is next selected, regardless of whether a filter is in effect.

Operations performed on the root node **Base Workspace** in the Hierarchy pane, such as exporting bus objects, affect only visible objects. An object that is invisible because a filter is in effect is unaffected by the operation. If you want to export all existing bus objects, be sure to clear any filter that may be in effect before performing the export.

Clear the Filter

To clear any filter currently in effect, click the **Clear Filter** button at the right of the Filter subpane, or press the **F5** key. The subpane reverts to its default state, in which all bus objects appear:



If you jump from a nested bus object to its source object by selecting the nested object and choosing **Go to 'element'** from its Context menu, and the source object is invisible due to a filter, the Bus Editor automatically clears the filter and selects the source object. Jumping to an object that is already visible leaves the filter unchanged.

Customize Bus Object Import and Export

In this section...

“Prerequisites for Customization” on page 51-60

“Writing a Bus Object Import Function” on page 51-60

“Writing a Bus Object Export Function” on page 51-61

“Registering Customizations” on page 51-62

“Changing Customizations” on page 51-63

You can use the Bus Editor to import bus objects to the base workspace, as described in “Import Bus Objects” on page 51-48, and to export bus objects from the base workspace, as described in “Export Bus Objects” on page 51-46. By default, the Bus Editor can import bus objects only from a MATLAB code file or MAT-file, and can export bus objects only to a MATLAB code file or MAT-file, with the files stored somewhere that is accessible using an ordinary **Open** or **Save** dialog.

Alternatively, you can customize the Bus Editor’s import and export commands by writing MATLAB functions that provide the desired capabilities, and registering these functions using the Simulink Customization Manager. When a custom bus object import or export function exists, and you use the Bus Editor to import or export bus objects, the editor calls the custom import or export function rather than using its default capabilities.

A customized import or export function can have any desired effect and use any available technique. For example, rather than storing bus objects in MATLAB code files or MAT-files in the file system, you could provide customized functions that store the objects as records in a corporate database, perhaps in a format that also meets other corporate data management requirements.

This section describes techniques for designing and implementing a custom bus object import or export function, and for using the Simulink Customization Manager to register such a custom function. The registration process establishes the custom import and export functions as callbacks for the Bus Editor **Import to Base Workspace** and **Export to File** commands, replacing the default capabilities of the editor.

Customizing the Bus Editor's import and export capabilities has no effect on any MATLAB or Simulink API function: it affects only the behavior of the Bus Editor. You can customize bus object import, export, or both. You can establish, change, and cancel import or export customization at any time. Canceling import or export customization restores the default Bus Editor capabilities for that operation without affecting the other.

Prerequisites for Customization

To perform bus object import or export customization, you must understand:

- The MATLAB language and programming techniques that you will need.
- Simulink bus object syntax. See “About Bus Objects” on page 51-24, `Simulink.Bus`, `Simulink.BusElement`, and “Nest Bus Definitions” on page 51-39.
- The proprietary format into which you will translate bus objects, and all techniques necessary to access the facility that stores the objects.
- Any platform-specific techniques needed to obtain data from the user, such as the name of the location in which to store or access bus objects.

The rest of the information that you will need, including all necessary information about the Simulink Customization Manager appears in this section. For complete information about the Customization Manager, see “Simulink Environment Customization”.

Writing a Bus Object Import Function

A function that customizes bus import can use any MATLAB programming construct or technique. The function can take zero or more arguments, which can be anything that the function needs to perform its task. You can use functions, global variables, or any other MATLAB technique to provide argument values. The function can also 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.

An example of 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 could be registered with the Simulink Customization Manager. A real import function would somehow obtain a designation of where to obtain the bus object(s) to import; convert each one to a Simulink bus object; and store the object in the base workspace. The additional logic needed is enterprise-specific.

Writing a Bus Object Export Function

A callback function that customizes bus export can use any MATLAB programming construct or technique. The function must take one argument, and can take zero or more additional arguments, which can be anything that the function needs to perform its task. When the Bus Editor calls the function, the value of the first argument is a cell array containing the names of all bus objects selected within the editor to be exported. 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.

An example of 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 could be registered. It accepts a cell array of bus object names, iterates over them, and prints each name. A real export function would use each name to retrieve the corresponding bus object from the base workspace; convert the object to proprietary format; and store the converted object somewhere. The additional logic needed is enterprise-specific.

Registering Customizations

To customize bus object import or export, you provide a *customization registration function* that inputs and configures the Customization Manager whenever you start the Simulink software or subsequently 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 (or 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 will be 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 directory; or you may want to extend the search path.

A simple example of a customization registration function is:

```
function sl_customization(cm)
disp('My customization file was loaded.');
```

```
cm.BusEditorCustomizer.importCallbackFcn = @myImportCallback;
cm.BusEditorCustomizer.exportCallbackFcn = @(x)myExportCallBack(x);
```

When the Simulink software starts up, it traverses the MATLAB search path looking for files named `sl_customization.m`. The software 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 previous customization function will display a message (which an actual function probably would not) and establish 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 need not be defined when the registration occurs, but it must be defined when the Bus Editor later calls the function. The same latitude and requirement applies to any functions or global variables used to provide the values of any 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, so any changes to functions in the customization file will be ignored until one of those events occurs. By contrast, changes to other MATLAB code files on the MATLAB path take effect immediately.

Changing Customizations

You can change the handles established in the `sl_customization` function by changing the function to specify the changed handles, saving the function, then refreshing customizations by executing:

```
sl_refresh_customizations
```

The Simulink software then 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 the appropriate `BusEditorCustomizer` element to `[]` in the `sl_customization` function, 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 “Registering Customizations” on page 51-62).

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, in which case they take effect next time you refresh customizations. Keeping the callback functions in separate files usually provides more flexible and modular results.

Connect Buses to Inports and Outports

In this section...

“Connect Buses to Root Level Inports” on page 51-65

“Connect Buses to Root Level Outports” on page 51-65

“Connect Buses to Nonvirtual Inports” on page 51-65

“Connect Buses to Model, Stateflow, and MATLAB Function Blocks” on page 51-68

“Connect Multi-Rate Buses to Referenced Models” on page 51-68

Connect Buses to Root Level Inports

If you want a root level Inport of a model to produce a bus signal, in the Inport block parameters dialog box, set **Data type** to **Bus: <object name>** and replace <object name> with the name of the bus object name that defines the bus that the Inport produces. See “Bus Objects” on page 51-24 for more information.

Connect Buses to Root Level Outports

A root level Outport of a model can accept a virtual bus only if all elements of the bus have the same data type. The Outport block automatically unifies the bus to a vector having the same number of elements as the bus, and outputs that vector.

If you want a root level Outport of a model to accept a bus signal that contains mixed types, in the Outport block parameters dialog box, set **Data type** to **Bus: <object name>** and replace <object name> with the name of the bus object name that defines the bus that the Inport produces. If the bus signal is virtual, it will be converted to nonvirtual, as described in “Automatic Bus Conversion” on page 51-16. See “Bus Objects” on page 51-24 more information.

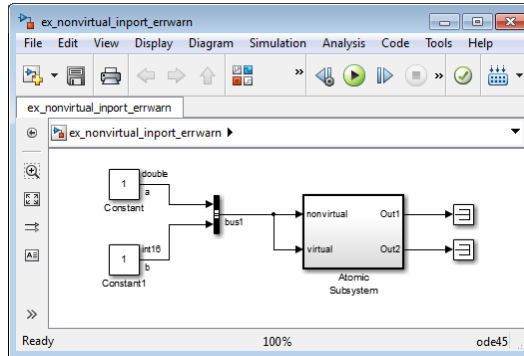
Connect Buses to Nonvirtual Inports

By default, an Inport block is a virtual block and accepts a bus as input. However, an Inport block is nonvirtual if it resides in a conditionally executed or atomic subsystem, or a referenced model, and it or any of its components

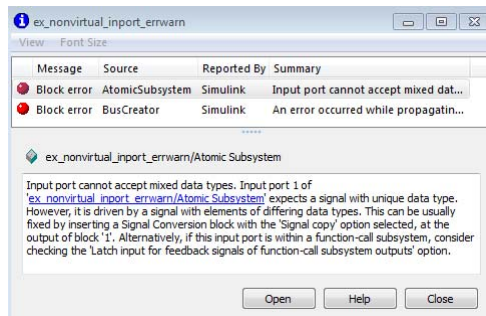
is directly connected to an output of the subsystem or model. In such a case, the Inport block can accept a bus only if all elements of the bus have the same data type.

If the components are of differing data types, attempting to simulate the model causes the Simulink software to halt the simulation and display an error message. You can avoid this problem, without changing the semantics of your model, by inserting a Signal Conversion block between the Inport block and the Outport block to which it was originally connected.

For example, the following model, which includes an atomic subsystem, does not simulate.

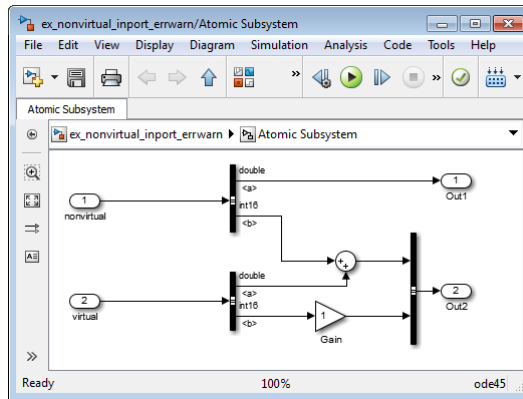


Starting the simulation generates the following error messages:

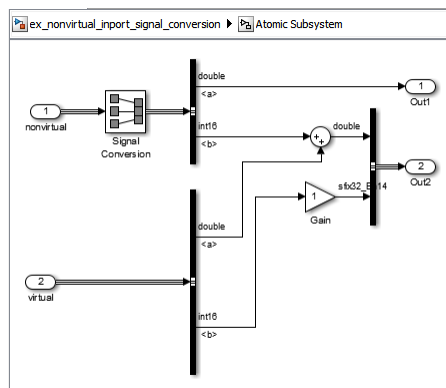


Opening the subsystem reveals that in this model, the Inport block labeled nonvirtual is nonvirtual because it resides in an atomic subsystem and one

of its components (labeled a) is directly connected to one of the subsystem's outputs. Further, the bus (bus1) connected to the subsystem's inputs has components of differing data types. As a result, you cannot simulate this model.



To break the direct connection to the subsystem's output, insert a Signal Conversion block. Set the Signal Conversion block **Output** parameter to **Signal copy**. Inserting the Signal Conversion block enables the Simulink software to simulate the model.



Connect Buses to Model, Stateflow, and MATLAB Function Blocks

Referenced models, Stateflow charts, and MATLAB Function blocks require any bus connected to them to be nonvirtual. To provide for this requirement, where possible the Simulink software automatically converts any virtual bus connected to a Model block or Stateflow chart to a nonvirtual bus. See “Automatic Bus Conversion” on page 51-16 for details.

Connect Multi-Rate Buses to Referenced Models

In a model that uses a fixed-rate solver, referenced models can input only single-rate buses. However, you can input the signals in a multi-rate bus to a referenced model by inserting blocks into the parent and referenced model as follows:

- 1 In the parent model:** Insert a Rate Transition block to convert the multi-rate bus to a single-rate bus. 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:
 - **Periodic sample time constraint** is Specified
 - **Sample time properties** contains the specified rate.
 - The Inport that accepts the 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.

Specify Initial Conditions for Bus Signals

In this section...

“Bus Signal Initialization” on page 51-69

“Create Initial Condition (IC) Structures” on page 51-71

“Three Ways to Initialize Bus Signals Using Block Parameters” on page 51-76

“Setting Diagnostics to Support Bus Signal Initialization” on page 51-80

Bus Signal Initialization

Bus signal initialization is a special kind of signal initialization. For general information about initializing signals, see “Initialize Signals and Discrete States” on page 50-52.

Bus signal initialization specifies the bus element values that Simulink uses for the first execution of a block that uses that bus signal. By default, the initial value for a bus element is the ground value (represented by 0). Bus initialization, as described in this section, involves specifying nonzero initial conditions (ICs).

You can use bus signal 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 signal values that input to a block, if that block meets both of these conditions:

- It has an initial value or initial condition block parameter
- It supports bus signals

The following blocks support bus signal initialization:

- Data Store Memory
- Memory
- Merge
- Outport (when the block is inside a conditionally executed context)
- Rate Transition
- Unit Delay

For example, the Unit Delay block is a bus-capable block and the Block Parameters dialog box for the Unit Delay block has an **Initial conditions** parameter.

Initialization Is Not Supported for Bus Signals with Variable-Size or Frame-Based Elements

You cannot initialize a bus that has:

- Variable-size signals
- Frame-based signals

Workflow for Initializing Bus Signals Using Initial Condition Structures

You need to set up your model properly to use initial condition structures to initialize bus signals. The general workflow involves the tasks listed in the following table. You can vary the order of the tasks, but before you update the diagram or run a simulation, you need to ensure your model is set up properly.

Task	Documentation
Define an IC structure	“Create Initial Condition (IC) Structures” on page 51-71
Use an IC structure to specify a nonzero initial condition.	“Three Ways to Initialize Bus Signals Using Block Parameters” on page 51-76
Set Configuration Parameters dialog box diagnostics	“Setting Diagnostics to Support Bus Signal Initialization” on page 51-80

Create Initial Condition (IC) Structures

You can create partial or full IC structures to represent initial values for a bus signal. Create an IC structure by either:

- Defining a MATLAB structure in the MATLAB base or Simulink model workspace
- Specifying an expression that evaluates to a structure for the initial condition parameter in the Block Parameters dialog box for a block that supports bus signal initialization

For information about defining MATLAB structures, see “Create a Structure Array” in the MATLAB documentation.

Full and Partial IC Structures

A full IC structure provides an initial value for every element of a bus signal. This IC structure mirrors the bus hierarchy and reflects the attributes of the bus elements.

A partial IC structure provides initial values for a subset of the elements of a bus signal. If you use a partial IC structure, during simulation, Simulink creates a full IC structure to represent all of the bus signal elements, assigning the respective ground value to each element for which the partial IC 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.

Specifying full structures during code generation offers these advantages:

- Generates more readable code
- Supports a modeling style that explicitly initializes all signals

Match IC Structure Values to Corresponding Bus Element Data Characteristics

The field that you specify in an IC structure must match the following data attributes of the bus element exactly:

- Name
- Data type
- Dimension
- Complexity

For example, if you define a bus element to be a real [2x2] double array, then in the IC structure, define the value to initialize that bus element to be a real [2x2] double array.

You must explicitly specify fields in the IC structure for every bus element that has an enumerated (enum) data type.

When you define a partial IC structure:

- Include only fields that are in the bus.
- You can omit fields that are in the bus.
- Make the field in the IC structure correspond to the nesting level of the bus element.
- Within the same nesting level in both the structure and the bus, you can specify the structure fields in a different order than the order of the elements in the bus.

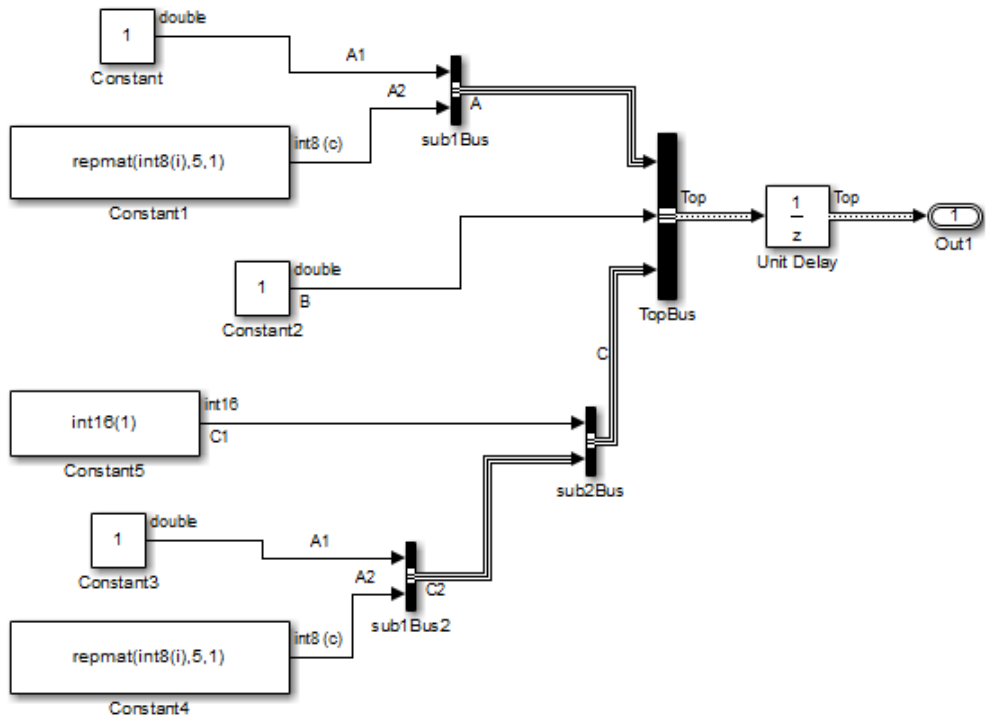
Note The value of an IC structure must lie within the design minimum and maximum range of the corresponding bus element. Simulink performs this range checking during an update diagram and when you run the model.

Examples of Partial Structures

Suppose 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 following model, `basic_example` includes the nested `Top` bus. The screen capture below shows the model after it has been updated.



The following diagram summarizes 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)

Valid partial IC structures. In the following examples, K is an IC structure specified for the initial value of the Unit Delay block. The IC structure corresponds to the Top bus in the `basic_example` model.

The following table shows valid initial condition specifications.

Valid Syntax	Description
<code>K.A.A1 = 3</code>	Bus element <code>Top.A.A1</code> is double; the corresponding structure field is <code>3</code> , which is a double.
<code>K = struct('C',struct('C1',int16(4)))</code>	Matching data types can require you to cast types. Bus element <code>Top.C.C1</code> is <code>int16</code> . The corresponding structure field explicitly specifies <code>int16(4)</code> .
<code>K = struct('B',3,'A',struct('A1',4))</code>	Bus element <code>Top.B</code> and <code>Top.A</code> 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 IC structures. In the following examples, K is an IC structure specified for the initial value of the Unit Delay block. The IC structure corresponds to the Top bus in the `basic_example` model.

These three initial condition specifications are *not* valid:

Invalid Syntax	Reason the Syntax Is Invalid
<code>K.A.A2 = 3</code>	Value data type, dimension, and complexity do not match. <code>Top.A.A2</code> is an <code>int8</code> , but <code>K.A.A2</code> is a double; <code>Top.A.A2</code> is <code>5x1</code> , but <code>K.A.A2</code> is <code>1x1</code> ; <code>Top.A.A2</code> is complex, but <code>K.A.A2</code> is real.
<code>K.C.C2 = 3</code>	You cannot use a scalar to initialize IC substructures.
<code>K = struct('B',3,'X',4)</code>	You cannot specify fields that are not in the bus (<code>X</code> does not exist in the bus).


Creating Full IC Structures Using Simulink.Bus.createMATLABStruct

Use the `Simulink.Bus.createMATLABStruct` function to streamline the process of creating a full MATLAB initial condition structure with the same hierarchy, names, and data attributes as a bus signal. This function fills all the elements that you do not specify with the ground values for those elements.

You can use several different kinds of input with the `Simulink.Bus.createMATLABStruct` function, including

- A bus object name
- An array of port handles

You can invoke the `Simulink.Bus.createMATLABStruct` function from the Bus Editor, using one of these approaches:

- Select the **File > Create a MATLAB structure** menu item.
- Select the bus object for which you want to create a full MATLAB structure and click the **Create a MATLAB structure** icon () from the toolbar.

You can then edit the MATLAB structure in the MATLAB Editor.

See the `Simulink.Bus.createMATLABStruct` documentation for details.

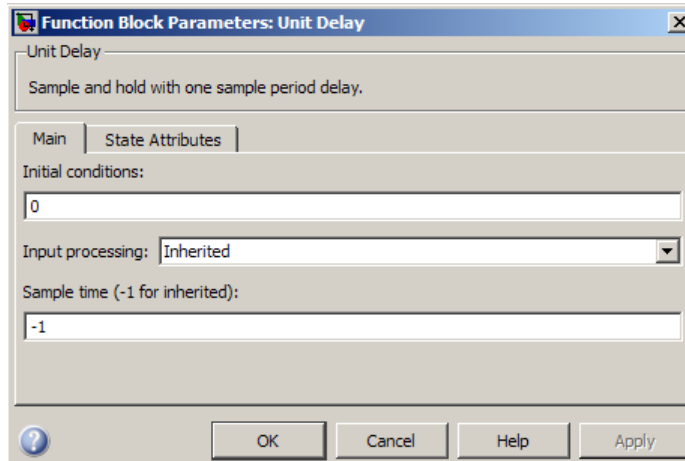
Using Model Advisor to Check for Partial Structures

To detect when structure parameters are not consistent in shape (hierarchy and names) with the associated bus signal, in the Simulink Editor, use the **Analysis > Model Advisor > By Product > Simulink “Check for partial structure parameter usage with bus signals”** check. This check identifies partial IC structures.

Three Ways to Initialize Bus Signals Using Block Parameters

Initialize a bus signal by setting the initial condition parameter for a block that receives a bus signal as input and that supports bus initialization (see “Blocks that Support Bus Signal Initialization” on page 51-69).

For example, the Block Parameters dialog box for the Unit Delay block has an **Initial conditions** parameter.



For a block that supports bus signal initialization, you can replace the default value of 0 using one of these approaches:

- “MATLAB Structure for Initialization” on page 51-78
- “MATLAB Variable for Initialization” on page 51-78
- “Simulink.Parameter for Initialization” on page 51-79

All three approaches require that you define an IC structure (see “Create Initial Condition (IC) Structures” on page 51-71). You cannot specify a nonzero scalar value or any other type of value other than 0, an IC structure, or `Simulink.Parameter` object to initialize a bus signal.

Defining an IC structure as a MATLAB variable, rather than specifying the IC structure directly in the block parameters dialog box offers several advantages, including:

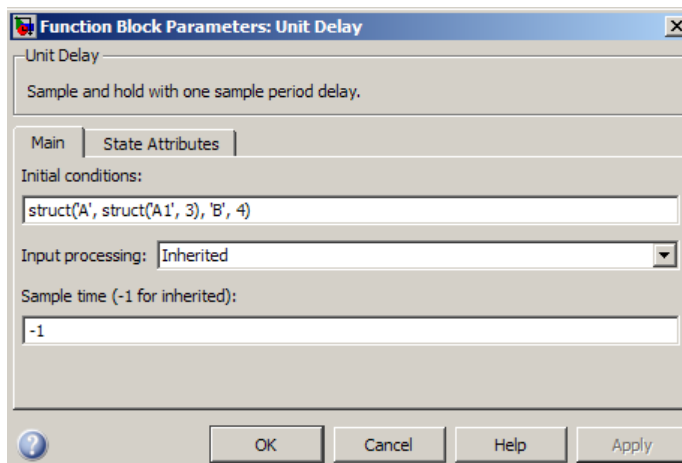
- Reuse of the IC structure for multiple blocks
- Using the IC structure as a tunable parameter during simulation

You can tune the value of a `Simulink.Parameter` object during simulation.

MATLAB Structure for Initialization

You can initialize a bus signal using a MATLAB structure that explicitly defines the initial conditions for the bus signal.

For example, in the **Initial conditions** parameter of the Unit Delay block, you could type in a structure such as shown below:



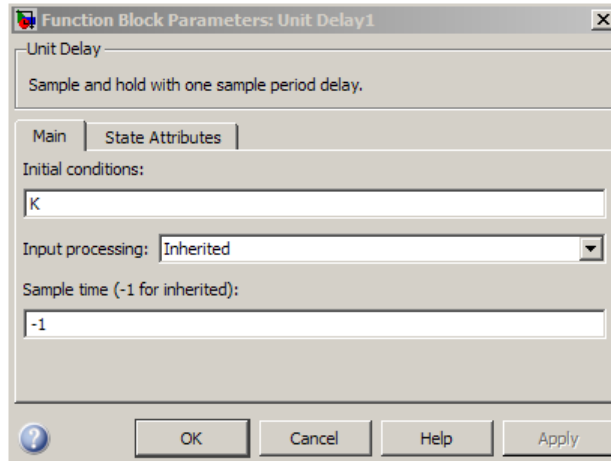
MATLAB Variable for Initialization

You can initialize a bus signal using a MATLAB variable that you define as an IC 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:



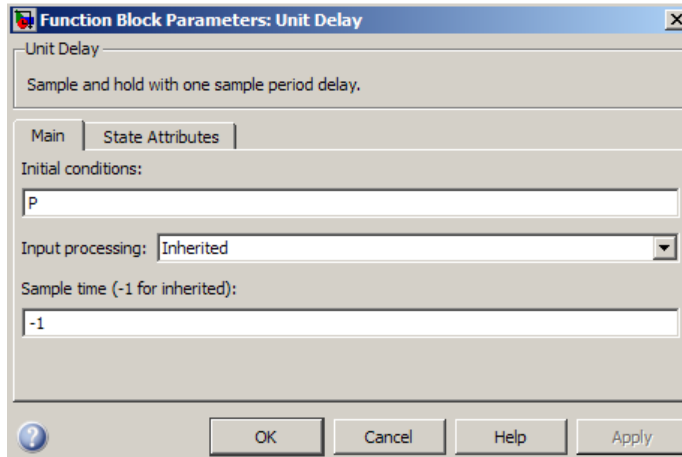
Simulink.Parameter for Initialization

You can initialize a bus signal using a `Simulink.Parameter` object that uses an IC structure for the `Value` property.

For example, you could define the partial structure `P` in the base workspace (reflecting the basic 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:



Setting Diagnostics to Support Bus Signal Initialization

To enable bus signal initialization, before you start a simulation, set the following two Configuration Parameter diagnostics as indicated:

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

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

Combine Buses into an Array of Buses

In this section...

“What Is an Array of Buses?” on page 51-81

“Benefits of an Array of Buses” on page 51-82

“Array of Buses Limitations” on page 51-83

“Define an Array of Buses” on page 51-85

“See Also” on page 51-86

Tip Simulink provides several techniques for combining signals into a composite signal. For a comparison of techniques, see “Techniques for Combining Signals” on page 51-3.

What Is an Array of Buses?

An array of buses is an array whose elements are buses. Each element in an array of buses must be nonvirtual and must have the same bus type. Each bus object has the same signal name, hierarchy, and attributes for its bus elements.

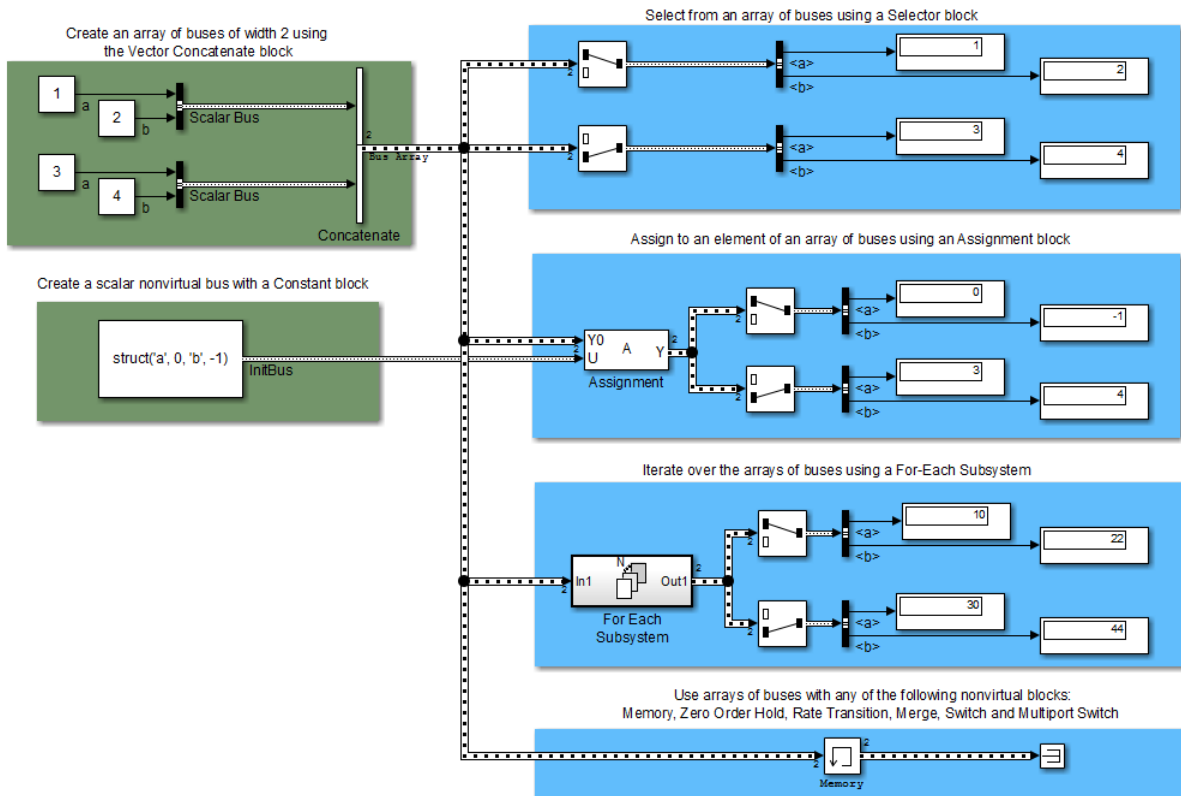
An example of using an array of buses is to model a multi-channel communication system. You can model all of the channels using the same bus object, although each of the channels could have a different value.

Using arrays of buses involves:

- Using a bus object as a data type (see “Specify a Bus Object Data Type” on page 46-27)
- Working with arrays (see “Create Numeric Arrays”)

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 bus signals. When you update the diagram, the model looks like the following figure:

Modeling Arrays of Bus Signals



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

Using an array of buses:

- Represents structured data compactly
 - Reduces model complexity
 - Reduces maintenance by centralizing algorithms used for processing multiple buses
- Streamlines iterative processing of multiple buses of the same type, for example, by using a For Each Subsystem with the array of buses
- Simplifies changing the number of buses, without your having to restructure the rest of the model or make updates in multiple places in the model
- Allows models to use built-in blocks, such as the Assignment or Selector blocks, to manipulate arrays of buses just like arrays of any other type, rather than your creating custom S-functions to manage packing and unpacking structure signals
- Supports using the combined bus data across subsystem boundaries, model reference boundaries, and into or out of a MATLAB Function block
- Allows you to keep all the logic in the Simulink model, rather than splitting the logic between C code and the Simulink model
 - Supports integrated consistency and correctness checking, maintaining metadata in the model, and avoids your keeping track of model components in two different environments
- Generates code that has an array of C structures, which you can integrate with legacy C code that uses arrays of structures
 - Simplifies indexing into an array for Simulink computations, using a for loop on indexed structures

Array of Buses Limitations

Bus Limitations

The buses combined into an array must all:

- 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

Supported Blocks

You can use arrays of buses with the following blocks:

- Virtual blocks
- Several nonvirtual blocks, such as:
 - Some signal routing blocks (for example, Data Store Memory, Merge, and Switch)
 - Rate Transition and Zero-Order Hold blocks
- Several additional blocks, such as Assignment, MATLAB Function, and Signal Conversion

For a complete list, see “Blocks That Support Arrays of Buses” on page 51-87. That section describes requirements for using the supported blocks.

Structure Parameter Limitations

The following limitations apply to using structure parameters with an array of buses.

You can use:

- 0 as an initial condition for an array of buses
- A scalar `struct` that represents the same hierarchy and names as the array of buses

You *cannot* use:

- Partial structures
- An array of structures
- A structure parameter for an array of buses that has an element that is an array of buses

For more information, see “Specify Initial Conditions for Bus Signals” on page 51-69.

Signal Logging Limitation

Simulink does not log signals inside referenced models in Rapid Accelerator mode.

Stateflow Limitations

Stateflow action language does not support arrays of buses.

Define an Array of Buses

For information about the kinds of buses that you can combine into an array of buses, see “Bus Limitations” on page 51-83.

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 column 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 bus signals.

- 1 Define *one* bus object to use for *all* of the buses that you want to combine into an array of buses. For information about defining bus objects, see “Create Bus Objects” on page 51-33.

The `sldemo_bus_arrays` model defines an `sldemo_bus_arrays_busobject` bus object, which both of the Bus Creator blocks use for the input bus signals (Scalar Bus) for the array of buses.

- 2 Add a Vector Concatenate or Matrix Concatenate block to the model and open the block's 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 of the buses that you want to be in the array of buses.

See Also

For details about working with an array of buses, see:

- “Arrays of Buses in Models” on page 51-87
- “Code Generation for Arrays of Buses” on page 51-97

For information about converting an existing model, see “Convert Models to Use Arrays of Buses” on page 51-94.

Arrays of Buses in Models

In this section...

- “Blocks That Support Arrays of Buses” on page 51-87
- “Arrays of Buses with Bus-Related Blocks” on page 51-88
- “Set Up a Model to Use Arrays of Buses” on page 51-89
- “Set Diagnostic” on page 51-92
- “Signal Line Style” on page 51-93

Blocks That Support Arrays of Buses

The following blocks support arrays of buses:

- Virtual blocks (see “Virtual Blocks” on page 23-2)
- These nonvirtual blocks:
 - Data Store Memory
 - Data Store Read
 - Data Store Write
 - Merge
 - Multiport Switch
 - Rate Transition
 - Switch
 - Zero-Order Hold
- Assignment
- MATLAB Function
- Matrix Concatenate
- Selector
- Signal Conversion
- Vector Concatenate

- Width
- Two-Way Connection (a Simscape block)

Block Parameter Settings

The following table describes the block parameter settings for blocks that support arrays of buses. This information is also in the reference pages for each of these blocks.

For usage information for bus-related blocks, see “Arrays of Buses with Bus-Related Blocks” on page 51-88.

Block	Block Parameters Settings
Memory	Initial condition — Only this parameter (which may be, but does not have to be, a structure) is scalar-expanded to match the dimensions of the array of buses.
Merge	<ul style="list-style-type: none"> • Allow unequal port widths — Clear this parameter. • Number of inputs — Set to a value of 2 or greater. • Initial condition — Only this parameter (which may be a structure) is scalar-expanded to match the dimensions of 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.

Arrays of Buses with Bus-Related Blocks

To select a signal within an array of buses:

- 1 Use a Selector block to find the appropriate bus within the array of buses.
- 2 Use a Bus Selector block to select the signal.

To assign a value to a signal within an array of buses:

- 1 Use a Bus Assignment block to assign a value to a bus element.
- 2 Use the Assignment block to assign the bus to the array of buses.

Bus Selector and Bus Assignment blocks can only accept scalar buses, not arrays of buses.

A Bus Creator block can accept an array of buses as input, but cannot have an array of buses as output.

For details, see “Set Up a Model to Use Arrays of Buses” on page 51-89.

Set Up a Model to Use Arrays of Buses

Setting up a model to use an array of buses usually involves basic tasks similar to these:

- 1 Define the array of buses (see “Define an Array of Buses” on page 51-85).
- 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.

Perform Iterative Processing

You can perform iterative processing on the bus signal 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, or on a selected subset of buses in the array of buses.

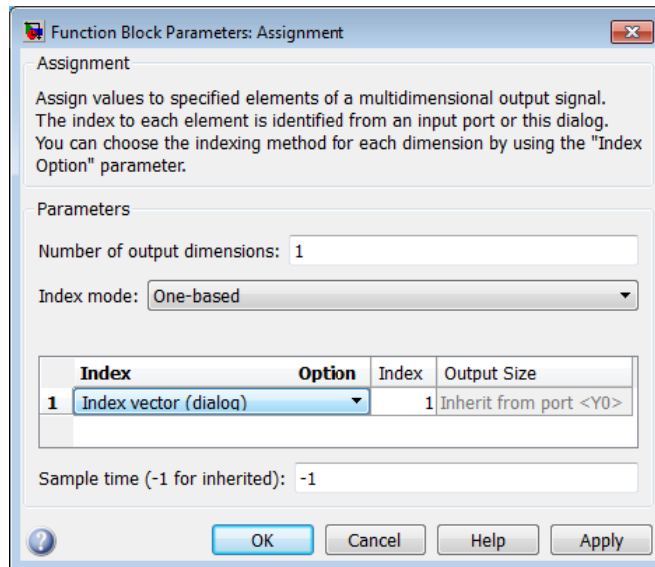
- 3 Connect the array of buses signal from the Concatenate block to the iterative processing subsystem.
- 4 Model your scalar algorithm within the iterative processing subsystem (for example, a For Each subsystem).
 - a Operate on the array of buses (using Selector and Assignment blocks).

- b Use the Bus Selector and Bus Assignment blocks to select elements from, or assign elements to, a scalar bus within the subsystem.

Assign values into an 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.

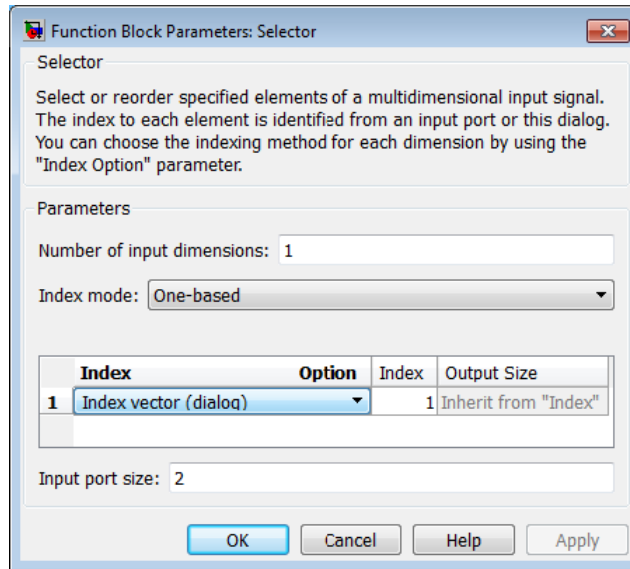


To assign bus elements within a bus signal, use the Bus Assignment block. The input for the Bus Assignment block must be a scalar bus signal.

Select bus elements from an array of buses

Use a Selector block to select elements of an array of buses. The input array of buses can have any dimension. The output bus signal 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 signal that the Assignment and For Each Subsystem blocks outputs. In this example, the block parameters dialog box for the Selector block that selects the first element looks like this:



To select bus elements within a bus signal, use the Bus Selector block. The input for the Bus Selector block must be a scalar bus signal.

- 5 Optionally, import or log array of buses data.

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 48-126.

You cannot use a From Workspace or From File block to import data for an array of buses.

Log Array of Buses Signals

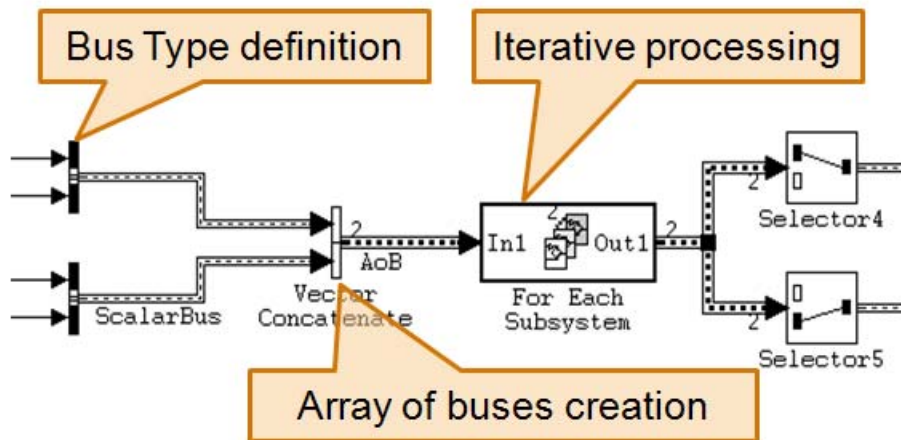
To export an array of buses signal, you must log the signal, using the Signal Properties dialog box.

Set the **Configuration Parameters > Data Import/Export > Signal logging format** to Dataset. For more information, see “Signal Logging”.

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 48-61.

The resulting model includes these components.



Set Diagnostic

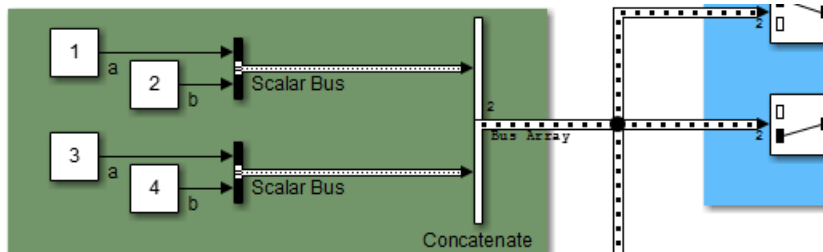
Before you run a simulation on a model that uses an array of buses, in the **Configuration Parameters > Diagnostics > Connectivity** pane, check

that the **Mux blocks used to create bus signals** parameter uses the default setting of error.

Signal Line Style

After you create an array of buses and update the diagram, the line style for the array of buses signal is a thicker version of the signal line style for a nonvirtual bus signal.

For example, in the `sldemo_bus_arrays` model, the `Scalar Bus` signal is a nonvirtual bus signal, and the `Bus Array` output signal of the `Concatenate` block is an array of buses signal.



Convert Models to Use Arrays of Buses

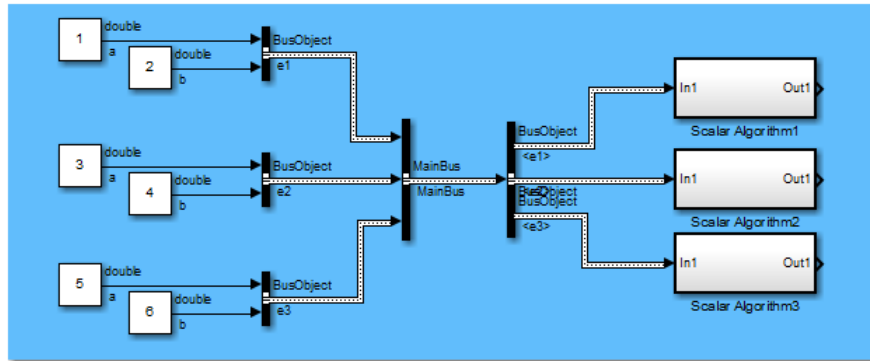
There are several reasons to convert a model to use an array of buses (see “Benefits of an Array of Buses” on page 51-82). For example:

- 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 that has grown in complexity.

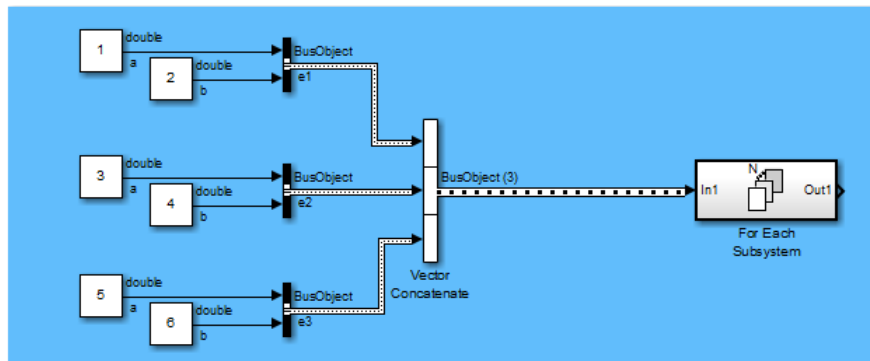
General Conversion Approach

This section presents 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 51-81 and “Arrays of Buses in Models” on page 51-87.

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 signal 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 new 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 signal is a bus of buses, where each element bus signal is of the same type.
 - The bus that you convert must be a nonvirtual bus. You can convert a virtual bus to a nonvirtual bus if all elements of the target bus have the same sample time (or if the sample time is inherited).
 - The target bus cannot have variable-dimensioned and frame-based elements.
- 2** Use a Concatenate block to convert the original bus of buses signal 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 signal is `BusObject`. The new model eliminates the wrapper bus signal (`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 signal 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 signal and concatenate the output bus signal.

The scalar algorithm within the For-Each subsystem cannot have continuous states. For additional limitations, see the For Each Subsystem block documentation.

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) are 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;          /* Block signals (auto storage) */  
} BusObject;  
typedef struct {  
    BusObject ForEachSubsystem_IterInp_0[2];  
} BlockIO_acb1;
```

Bus Data Crossing Model Reference Boundaries

A model reference boundary refers to the boundary between a model that contains a Model block and the referenced model. If you have bus data in a model that is passed to a Model block, then that data crosses the boundary to the referenced model.

To have bus data cross model reference boundaries:

- 1** Use a nonvirtual bus (required). Using a nonvirtual bus provides a well-defined data interface for code generation.

Use a bus object (`Simulink.Bus`) to define the bus. For details, see “Creating Nonvirtual Buses” on page 51-15.

- 2** Consider stripping out unneeded data from bus objects crossing model reference boundaries.

In large models, bus objects can become quite large and have several levels of hierarchy. Often referenced models need some, but not all, of the data contained in large buses. Passing unneeded data across model reference boundaries impacts performance negatively. The interface definition for a model should specify exactly what data the model uses.

Buses and Libraries

When you define a library block, the block can input, process, and output buses just as an ordinary subsystem can.

You need to provide the appropriate input bus signal if:

- You have a bus routing block (Bus Creator, Bus Selector, or Bus Assignment) block that is in a library.
- That block depends on signals that are input to the library.

To change that block in a library, perform these steps. For details about modifying library links, see “Work with Library Links” on page 28-8.

- 1** Copy the library block that uses that block to a model that connects an input bus.
- 2** Disable the link to the library block in this model.
- 3** Edit the bus routing block within the context of the outside model.
- 4** Resolve the link to the library.
- 5** In the Link Tool, in **Push/Restore Mode**, select Push to place the edited content into the library.
- 6** Save the library.

Alternatively, to configure the library to supply an appropriate bus signal, use a bus object to lock in the data type at the interface of a library subsystem block .

Prevent Bus and Mux Mixtures

In this section...

“What Are Bus and Mux Signal Mixtures?” on page 51-100

“Why Avoid Mixing Bus and Mux Signals?” on page 51-101

“When to Configure a Model to Prevent Bus and Mux Mixtures?” on page 51-102

“Two Upgrade Procedures” on page 51-103

What Are Bus and Mux Signal Mixtures?

A Simulink mux signal is a virtual signal that graphically combines two or more scalar or vector signals into one signal line. All of the signals in a mux signal must have the same data type and complexity. For more information, see “Mux Signals” on page 50-11.

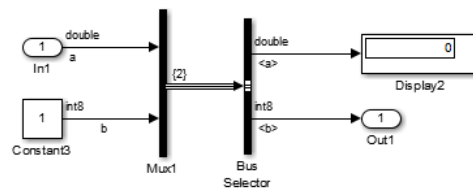
Bus signals can contain signals of different data types and complexities.

A bus and mux mixture occurs when some blocks treat a signal as a mux, while other blocks treat that same signal as a bus, as described below.

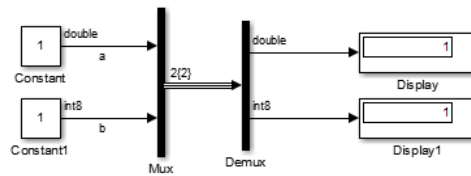
Mux Block That Creates a Virtual Bus

One way that a model can mix bus and mux signals is when a Mux block creates a virtual bus. For example:

- A signal from a Mux block connects to a Bus Selector block.



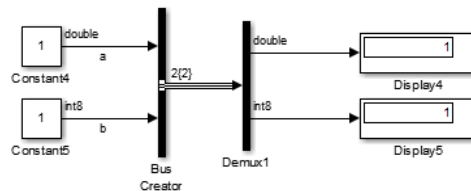
- Mux and Demux blocks combine a signal with different attributes (for example, data type or complexity).



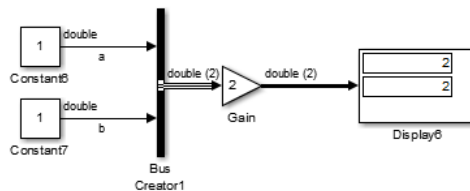
Bus Signal Treated as a Mux Signal

A second way that a model can mix bus and mux signals is when the model treats a bus signal as if it is a mux signal. For example:

- Bus Creator block that creates a signal to be used as a vector.



- Bus Creator block that creates a signal to be used as a vector.



Why Avoid Mixing Bus and Mux Signals?

Mixing bus and mux signals in a model causes your model to be less robust. Configuring your model to prevent bus and mux mixtures:

- Improves loop handling
- Produces clear error messages

- Contributes to consistent edit and compile time behavior

Configuring your model to prevent bus and mux mixtures not only makes your model more robust, it also allows you to update your model to take advantage of several features that you could not otherwise use, including:

- Nonzero initialization of bus signals
- Bus support for blocks such as Constant, Data Store Memory, From File, From Workspace, To File, and To Workspace
- Signal Hierarchy Viewer
- Signal label propagation enhancements
- Arrays of buses

When to Configure a Model to Prevent Bus and Mux Mixtures?

Configure a model to prevent bus and mux mixtures if *all* of the following conditions apply — the model:

- Was created before R2013b
- Contains one of the following: Bus Creator, Bus Selector, or Bus Assignment block, or a bus object
- Has **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** set to none or warning

Even if a model created before R2013b does not require that you configure it to prevent bus and mux mixtures, you may want to update it anyway. Doing so facilitates making future modifications to the model to use buses or features that require that the model is configured to prevent bus and mux mixtures.

Starting in R2013b, when you create a new model, Simulink automatically configures the model to avoid bus and mux mixtures. Simulink sets the **Mux blocks used to create bus signals** parameter to error. Then, when you do a model update or simulate the model, Simulink reports an error if you introduce a bus and mux mixture.

Two Upgrade Procedures

Configuring a model to prevent bus and mux mixtures involves performing these procedures:

- “Correct Mux Blocks That Create Bus Signals” on page 51-104
- “Correct Buses Used as Muxes” on page 51-109

The procedure to correct muxes that create bus signals not only configures your model to identify when you introduce a bus and mux mixture, it also automatically changes your model to correct many of the instances of such mixtures.

Correct Mux Blocks That Create Bus Signals

In this section...

“Choose the Appropriate Procedure” on page 51-104

“Models Without Model Referencing” on page 51-104

“Models With Model Referencing” on page 51-105

“Address Compatibility Issues After Running Upgrade Advisor” on page 51-106

Choose the Appropriate Procedure

The procedure that you perform depends on whether or not a model uses model referencing. Use the appropriate procedure, as described in:

- “Models Without Model Referencing” on page 51-104
- “Models With Model Referencing” on page 51-105

Models Without Model Referencing

- 1 Save a copy of the existing model and simulation results for the model.

Because you might need to make several changes to your model during the conversion process, it can help to have the original model for reference and for comparing simulation results.

- 2 Open the Upgrade Advisor. In the Simulink Editor, select **Analysis > Model Advisor > Upgrade Advisor**.
- 3 Select **Check for Mux blocks used to create bus signals**.
- 4 Click **Run this check**.
- 5 Click **Modify**. The Upgrade Advisor replaces Mux blocks with Bus Creator blocks and sets the **Mux blocks used to create bus signals** configuration parameter to error.

The Upgrade Advisor closes any models that it checks that contain library links.

- 6 If necessary, reopen the model. Rerun the check to confirm that the model passes.

If the check fails, go to step 7.

- 7 Update your model to address the applicable modeling patterns described in “Address Compatibility Issues After Running Upgrade Advisor” on page 51-106 and rerun the check.

Models With Model Referencing

In a model that uses model referencing, the Upgrade Advisor upgrades the top model, but not the referenced model. To make the upgrade process more efficient for models that use model referencing, use *one* of the following approaches:

- For models that reference a small number of models (for example, five), in the Upgrade Advisor, run the **Analyze model hierarchy for upgrade issues** check. This check addresses bus and mux signal mixture issues for each referenced model.

The steps for using the Upgrade Advisor are the same as described in “Models Without Model Referencing” on page 51-104.

- For models that have many referenced models (for example, more than five), use `find_mdrefs` and `sreplace_mux` to find Mux blocks and replace them with Bus Creator blocks. You can create a script based on the following pattern.

```
[refMdl,~] = find_mdrefs mdl, true);

% run through reference models and top model
for i = 1:length(refMdl)
    load_system(refMdl{i});

    % Add code if any model uses referenced configuration sets

    sreplace_mux(refMdl{i},false) % reportonly = false;

    save_system(refMdl{i});
    close_system(refMdl{i});
end
```

If a model does not compile, address the applicable modeling patterns described in “Address Compatibility Issues After Running Upgrade Advisor” on page 51-106 and rerun the **Analyze model hierarchy for upgrade issues** check or, for models with many referenced models, the script.

Address Compatibility Issues After Running Upgrade Advisor

For many models, running the Upgrade Advisor modifies your model so that Mux blocks no longer create bus signals. However, for some models you might encounter compatibility issues even after running the check. You need to modify your model manually to address those issues.

Also, after you compile the model using Upgrade Advisor, the Simulink Editor sometimes indicates that the model needs to be saved (that is, that the model is dirty), even though you did not make any changes. Save the model to prevent this issue from reoccurring for this model.

Modeling Pattern	Issue	Solution
Demux block with Bus selection mode enabled	You cannot use a Demux block to select bus signals.	Replace the Demux block with a Bus Selector block. If a Demux block has input from a Mux block, change the Mux block to a Bus Creator.
Data Store Memory block with Data Type set to Inherit: auto	A Data Store Memory block whose associated Data Store Read or Data Store Write blocks read or write bus signal data must use a bus object.	In the Data Store Memory block, set the Data Type signal attribute to Bus: <BusObject> .

Modeling Pattern	Issue	Solution
Signal Conversion block Output parameter matches input bus type	<p>A Signal Conversion block whose Output parameter is set to Nonvirtual bus requires a virtual bus input.</p> <p>A Signal Conversion block whose Output parameter is set to Virtual bus requires a nonvirtual bus input.</p>	To create a copy of the input signal, set Output to Signal copy .
Merge, Switch, or Multiport Switch block with multiple bus inputs	Merge, Switch, and Multiport Switch blocks with multiple bus inputs require all of those inputs to have the same names and hierarchy.	<p>Reconfigure the model so that the bus inputs all have the same names and hierarchy.</p> <p>To view the names and hierarchy of the bus signals, use the “Signal Hierarchy Viewer” on page 51-8 .</p>
Root Inport block outputting a virtual bus and specifying a value for Port dimensions	A root Inport block that outputs to a virtual bus must inherit the dimensions.	Set the Inport block Port dimensions signal attribute to 1 or -1 (inherit).
Mux block with nonvirtual bus inputs	A Mux block cannot accept nonvirtual bus signals.	<p>To treat the output as an array, replace the Mux block with a Vector Concatenate block.</p> <p>If you want a virtual bus output, use a Bus Creator block to combine the signals.</p>

Modeling Pattern	Issue	Solution
Bus to Vector block without a virtual bus signal input	A nonbus signal does not need a Bus to Vector block.	Remove the Bus to Vector block.
Assignment block with virtual bus inputs	The Upgrade Advisor converts the Assignment block Y0 port bus input to a vector.	Add a Bus to Vector block before the Assignment block.
S-function using a nonvirtual bus	An S-function that is not a Level-2 C S-function does not support nonvirtual bus signals.	Change the S-function to be a Level-2 C S-function. Consider using an S-Function Builder block to create a Level-2 C S-function.
Stateflow chart with parameterized data type	In a Stateflow chart, you cannot parameterize the data type of an input or output in terms of another input or output if the data type is a bus object.	For the parameterized port, set Data Type to Bus: <object name>.
Ground block used as a bus source	The output signal of a Ground block cannot be a source for a bus.	Use a Constant block with Constant value set to 0 and the Output data type signal attribute set to Bus: <object name>.

Correct Buses Used as Muxes

In this section...
“Three Approaches” on page 51-109
“Use the Model Advisor” on page 51-109
“Explicitly Add Bus to Vector Blocks” on page 51-110
“Reorganize the Model” on page 51-111
“Bus to Vector Block Compatibility Issues” on page 51-112

Three Approaches

The three approaches that you can use to correct bus signals used as muxes are:

- “Use the Model Advisor” on page 51-109
- “Explicitly Add Bus to Vector Blocks” on page 51-110
- “Reorganize the Model” on page 51-111

Generally, using the Model Advisor is the most efficient approach.

Use the Model Advisor

Before you use the Model Advisor to correct bus signals used as a muxes, perform the steps described in “Correct Mux Blocks That Create Bus Signals” on page 51-104.

- 1** Open the Model Advisor from the Upgrade Advisor, or in the Simulink Editor, select **Analysis > Model Advisor > Model Advisor**.
- 2** Select and run the **Simulink > Check bus usage** check.

The Model Advisor reports any cases of bus signals treated as muxes.

- 3** Follow the Model Advisor suggestions to correct any errors reported by the check.

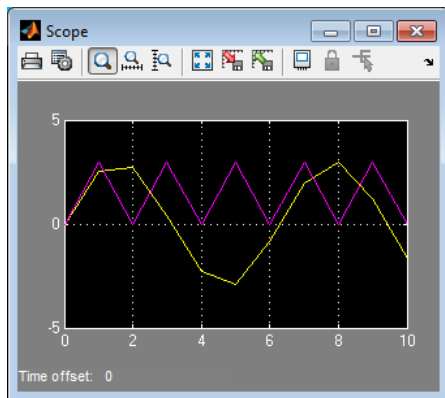
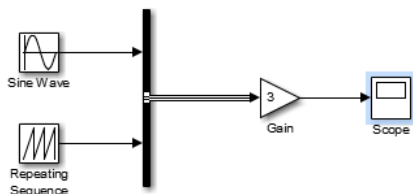
For additional information about using the Model Advisor, see “Consult the Model Advisor” on page 4-88.

Explicitly Add Bus to Vector Blocks

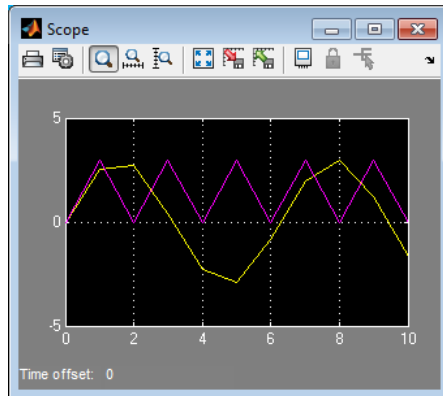
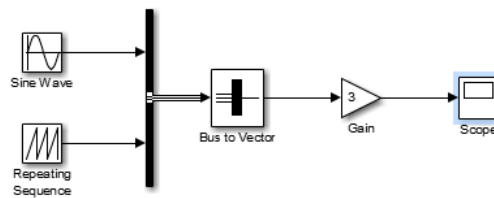
You can explicitly add Bus to Vector blocks to convert the bus signal to a mux (vector), using one of these approaches:

- Insert the Bus to Vector block into any bus used implicitly as a mux to explicitly convert the bus to a mux (vector).
- Use the `Simulink.BlockDiagram.addBusToVector` function, which automatically inserts Bus to Vector blocks wherever needed.

For example, this model uses a bus signal as a mux signal by using the bus as an input to a Gain block. The Scope block shows the simulation results.



This figure shows the same model, rebuilt after inserting a Bus to Vector block after the Bus Creator block.

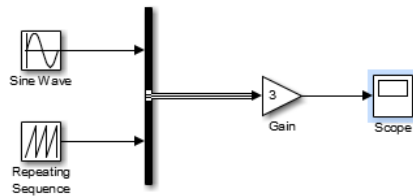


The results of simulation are the same in either case. The Bus to Vector block is virtual, and does not affect simulation results, code generation, or performance.

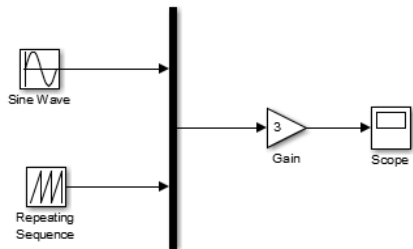
Reorganize the Model

You can replace blocks manually to avoid using bus signals as muxes. Change the sources for a block that require vector inputs to avoid feeding a bus signal into a block that requires vector input.

For example, in the following model, the Gain block requires a vector signal. However, input signal is a bus created by a Bus Creator block.



Change the Bus Creator block to a Mux block to provide the required vector signal for the Gain block.



Challenges with reorganizing the model manually include:

- Identifying all of the occurrences in a model. (The Model Advisor check identifies all occurrences in the model and helps you to correct them.)
- Dealing with many occurrences in a model is time-consuming and error-prone.
- Reorganizing the model to address this issue can interfere with other aspects of the model.

Bus to Vector Block Compatibility Issues

If you use **Save As** to save a model in a version of the Simulink product before R2007a (V6.6), Simulink:

- Sets the `StrictBusMsg` parameter to `error` if its value is `WarnOnBusTreatedAsVector` or `ErrorOnBusTreatedAsVector`.

- Replaces each Bus to Vector block in the model with a null subsystem that outputs nothing.

The resulting model specifies strong type checking for Mux blocks used to create buses. Before you can use the model, you must reconnect or otherwise correct each signal that contained a Bus to Vector block but is now interrupted by a null subsystem.

Buses in Generated Code

If you have a Simulink Coder license, the various techniques for defining buses are essentially equivalent for simulation, but the techniques used 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, below is the generated code for 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, and a separate copy of any algorithm that manipulates the bus exists for each element.

Using buses properly results in efficient code and visually clean models. If you intend to generate production code for a model that uses buses, see “Buses” for information about the best techniques to use.

Composite Signal Limitations

- Buses that contain signals of enumerated data types cannot pass through a block that requires a nonzero scalar initial value (such as a Unit Delay block).
- Root level bus outputs cannot be logged using the **Configuration Parameters > Data Import/Export > Save to Workspace > Output** option. Use standard signal logging instead, as described in “Export Signal Data Using Signal Logging” on page 48-21.
- 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.

Working with Variable-Size Signals

- “Variable-Size Signal Basics” on page 52-2
- “Simulink Models Using Variable-Size Signals” on page 52-6
- “S-Functions Using Variable-Size Signals” on page 52-21
- “Simulink Block Support for Variable-Size Signals” on page 52-24
- “Variable-Size Signal Limitations” on page 52-28

Variable-Size Signal Basics

In this section...

“About Variable-Size Signals” on page 52-2

“Creating Variable-Size Signals” on page 52-2

“How Variable-Size Signals Propagate” on page 52-3

“Empty Signals” on page 52-4

“Subsystem Initialization of Variable-Size Signals” on page 52-4

“See Also” on page 52-5

About Variable-Size Signals

A Simulink signal can be a scalar, vector (1-D), matrix (2-D), or N-D. For information about these types of signals, see “Signal Basics” on page 50-2 in the *Simulink User’s Guide*.

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.

Creating Variable-Size Signals

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:

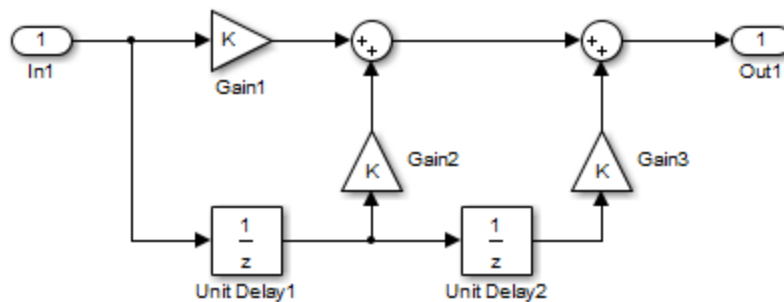
- **At every step of model execution.**

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 52-14 shows how you can use the two ways of propagating variable-size signals in a complementary fashion to model complex systems.

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.

Subsystem Initialization of Variable-Size Signals

The initial signal size from an Output 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.
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 Output 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 Output 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.

See Also

- “Simulink Models Using Variable-Size Signals” on page 52-6
- “S-Functions Using Variable-Size Signals” on page 52-21
- “Simulink Block Support for Variable-Size Signals” on page 52-24
- “Variable-Size Signal Limitations” on page 52-28

Simulink Models Using Variable-Size Signals

In this section...

“Variable-Size Signal Generation and Operations” on page 52-6

“Variable-Size Signal Length Adaptation” on page 52-10

“Mode-Dependent Variable-Size Signals” on page 52-14

“See Also” on page 52-19

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 52-24.

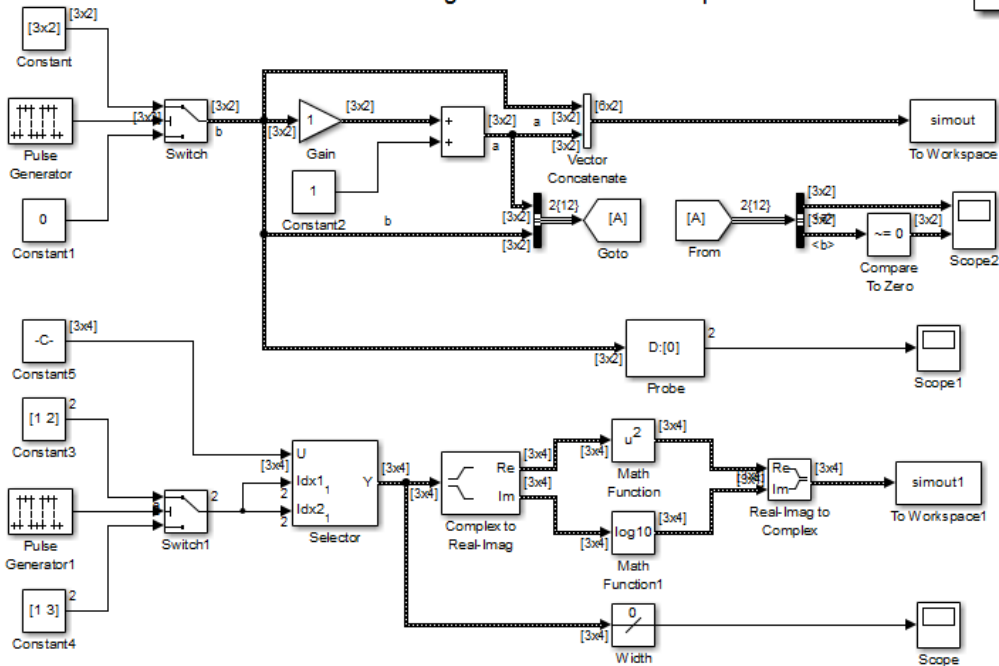
1 In the MATLAB Command Window, type

```
sldemo_varsize_basic
```

2 In the Simulink Editor, select **Display > Signals & Ports > Signal Dimensions**. Run a simulation or press Ctrl-D.

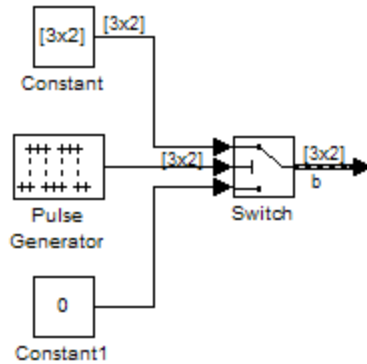
The Simulink Editor displays the signal dimensions and line styles. See “Signal Basics” on page 50-2 for an interpretation of signal line styles.

Variable-Size Signal Generation and Operations



Creating 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 3×2 . 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.

Saving 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
```

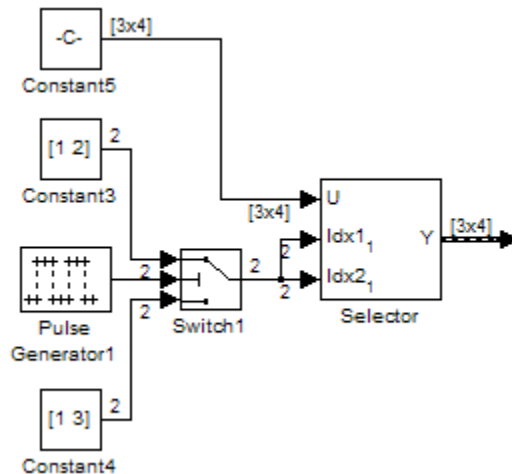
The signal dimensions for the first three time steps are shown.

```
ans =
```

```
    3    2
    3    2
    1    1
```

Creating 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.



Viewing 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.

Processing Variable-Size Signals

The remainder of the model shows various operations that are possible with variable-size signals. Operations include using the Gain block, the Sum block, the Math Function block, the Matrix Concatenate block. You can connect variable-size signals with the From, Goto, Bus Assignment, Bus Creator, and Bus Selector blocks.

Variable-Size Signal Length Adaptation

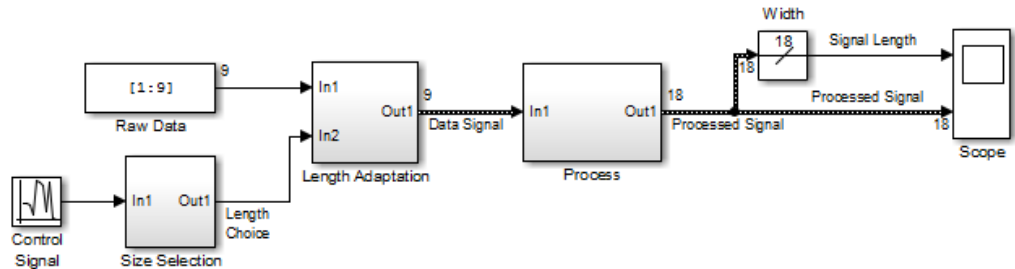
This example model corresponds to a hypothetical system where the model adapts the length of a signal over time. Length adaptation is based on the value of a control signal. When the control signal falls within one of three

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 52-24.

To open the example model, in the MATLAB Command Window, type:

```
sldemo_varsize_dataLengthAdapt
```



Creating a Variable-Size Signal by Adapting the Length of a Data Signal

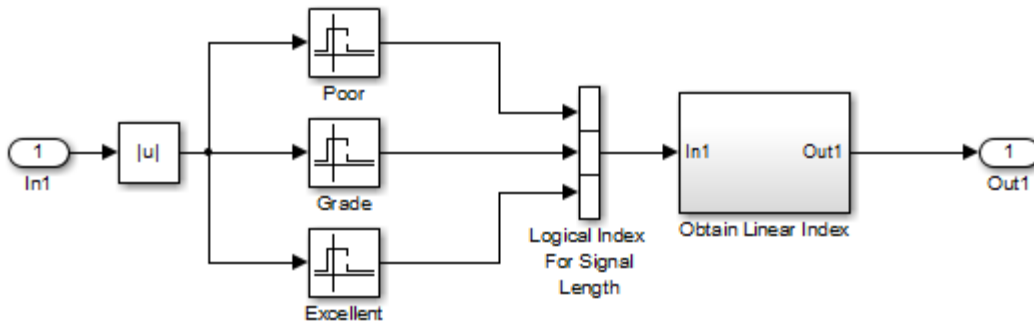
This model generates a data signal and converts the signal to a variable-size signal. The size of the signal depends on the value of a control signal. The raw data signal is a column vector with values from 1 to 9.

```
[1:9].'
```

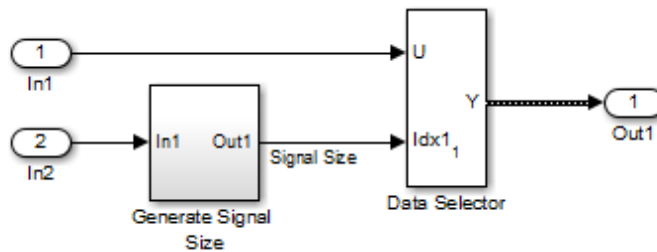
```
ans =
     1
     2
     3
     4
     5
     6
     7
```

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.

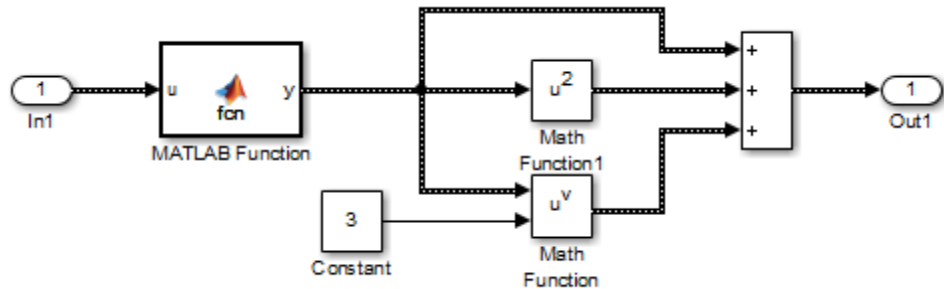


In the Length Adaptation subsystem, the Signal Size subsystem generates an index based on the quality value from the Size Selection subsystem (In2). The Data Selector block uses the starting and ending indices to adapt the length of the data signal (In1) and output a variable-size signal.



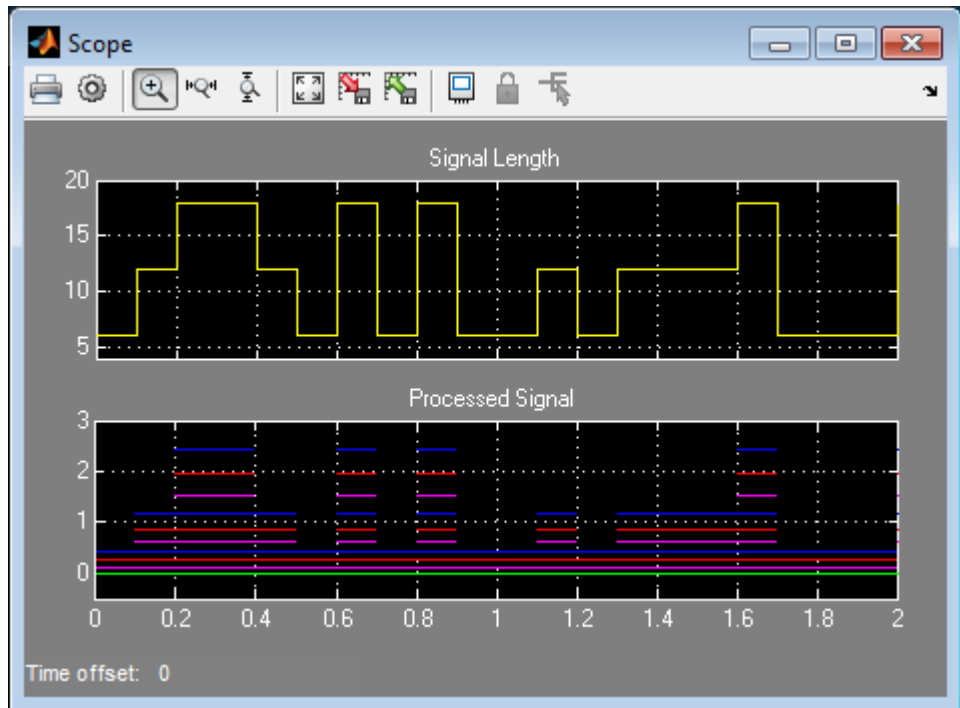
Processing a Variable-Size Signal

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.



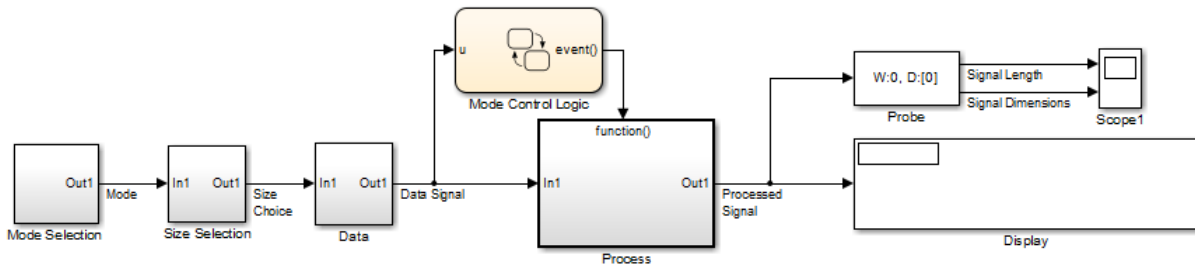
Mode-Dependent Variable-Size Signals

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.

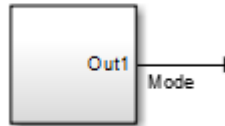
To open the model, In the MATLAB Command Window, type:

```
sldemo_varsize_multimode
```

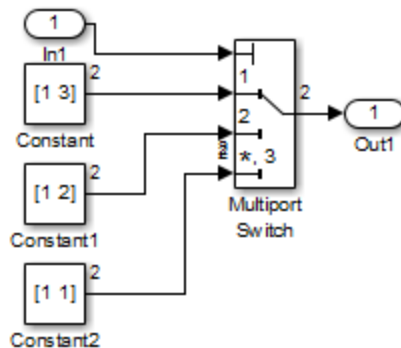


Creating a Variable-Size Signal Based on Mode

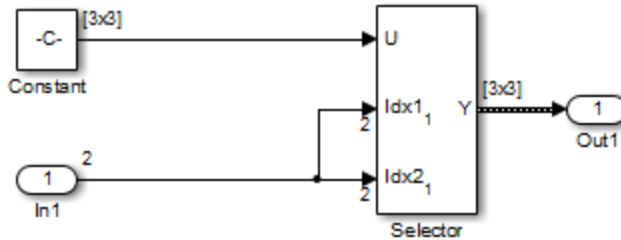
The Mode Selection subsystem determines the mode for processing a data signal and outputs a mode value (1, 2, or 3). This value helps to select the length of the data signal using the Size Selection and Data subsystems.



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

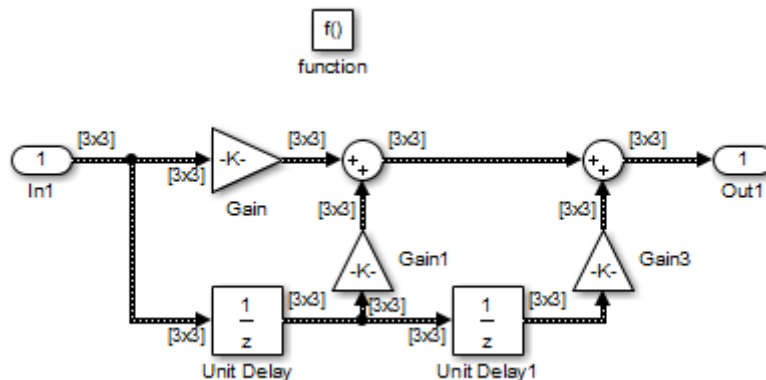
ans(:, :, 3) =

1	4	7
2	5	8
3	6	9

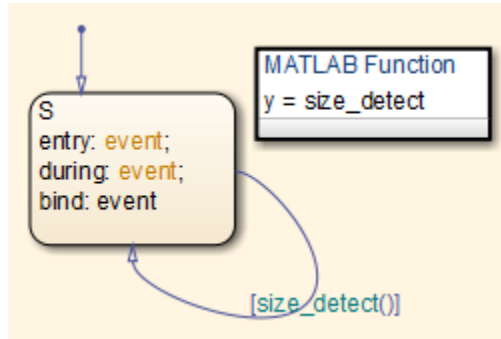
Processing a Variable-Size Signal with a Conditionally Executed Subsystem

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 52-3.

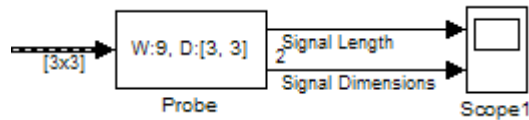


The Stateflow chart determines if there is a change in the size of the signal. The function `size_detect` calculates the width of the variable-size signal at each time step, and compares the current width to the previous width. If there is a change in signal size, the chart outputs a function-call output event that resets and repropagates the signal sizes within the Process subsystem.

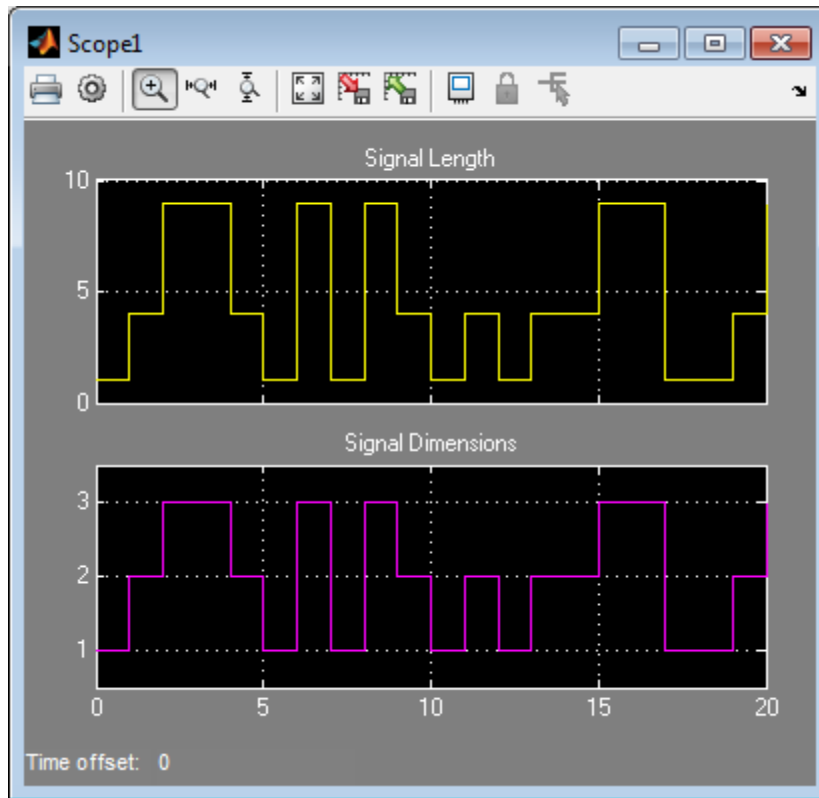


Visualizing Data

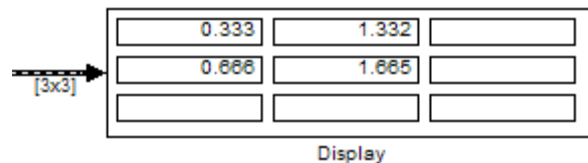
Use the Probe block to visualize signal size and signal dimension.



Since the signals are $n \times n$ 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.



See Also

- “Variable-Size Signal Basics” on page 52-2

- “S-Functions Using Variable-Size Signals” on page 52-21
- “Simulink Block Support for Variable-Size Signals” on page 52-24
- “Variable-Size Signal Limitations” on page 52-28

S-Functions Using Variable-Size Signals

In this section...

“Level-2 MATLAB S-Function with Variable-Size Signals” on page 52-21

“C S-Function with Variable-Size Signals” on page 52-22

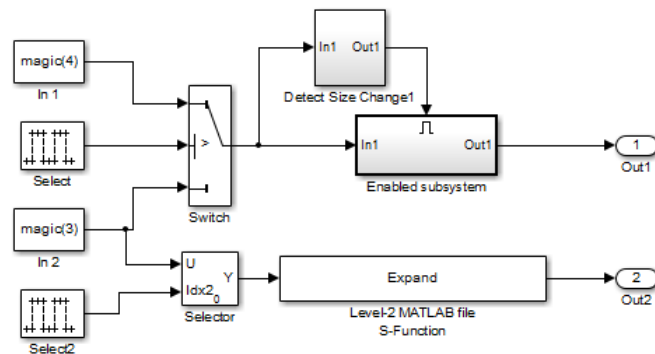
“See Also” on page 52-23

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:

```
msfcdemo_varsize
```



```
matlabroot\toolbox\simulink\simdemos\simfeatures\msfcn_varsize_expand.m
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_c\msfcn_varsize_expand.tlc
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\msfcn_varsize_holdStatesUntilReset.m
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_c\msfcn_varsize_holdStatesUntilReset.tlc
```

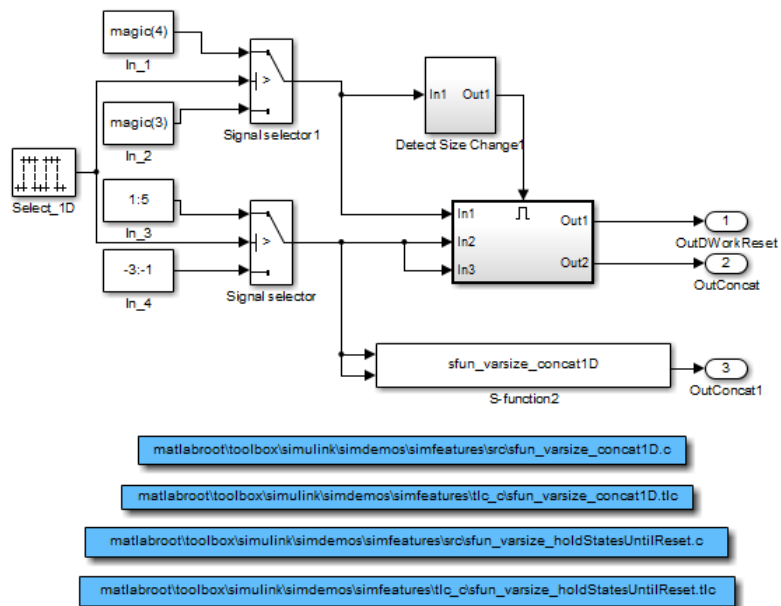
The Enabled subsystem includes a Level-2 MATLAB S-Function which shows how to implement a block that holds its states until reset. Because this block contains states and delays the input signal, the input size can change only when a reset occurs.

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:

```
sfcn_demo_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

- “Variable-Size Signal Basics” on page 52-2
- “Simulink Models Using Variable-Size Signals” on page 52-6
- “Simulink Block Support for Variable-Size Signals” on page 52-24
- “Variable-Size Signal Limitations” on page 52-28

Simulink Block Support for Variable-Size Signals

In this section...

“Simulink Block Data Type Support” on page 52-24

“Conditionally Executed Subsystem Blocks” on page 52-24

“Switching Blocks” on page 52-25

“See Also” on page 52-27

Simulink Block Data Type Support

The Simulink Block Data Type Support table includes a complete list of blocks that support variable-size signals.

To view the table:

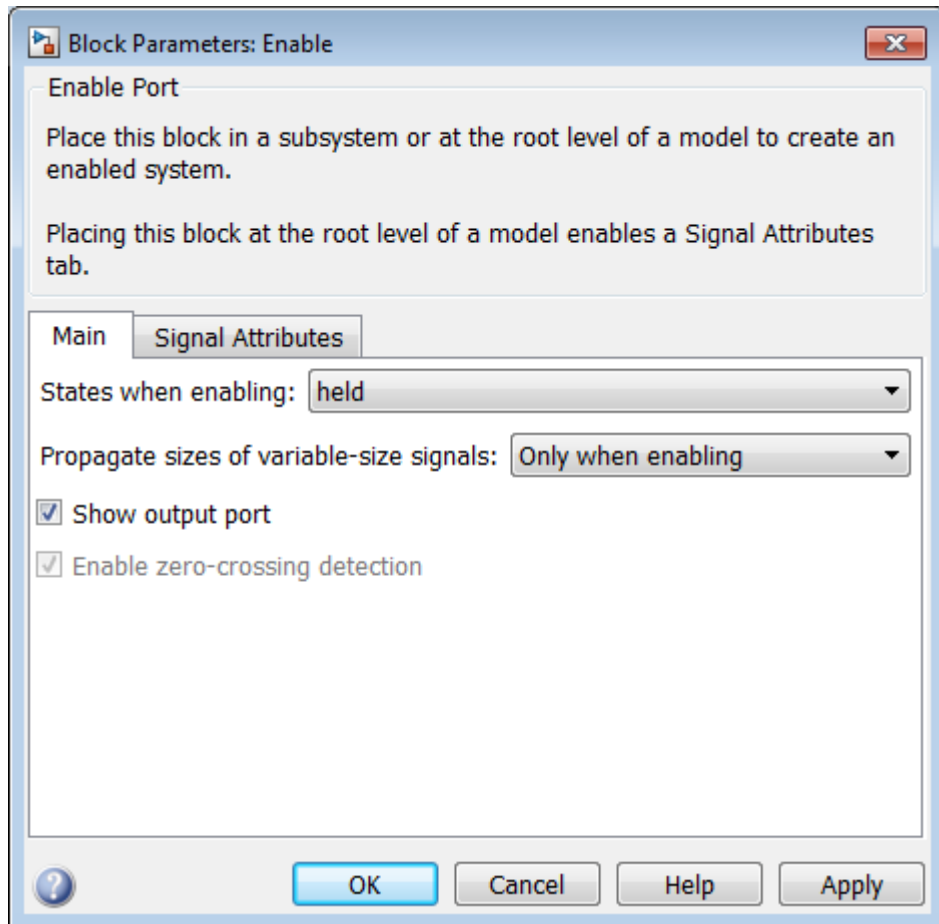
- 1 Open a Simulink model.
- 2 Select **Help > Simulink > Block Data Types & Code Generation Support > Simulink**.

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.

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)**.

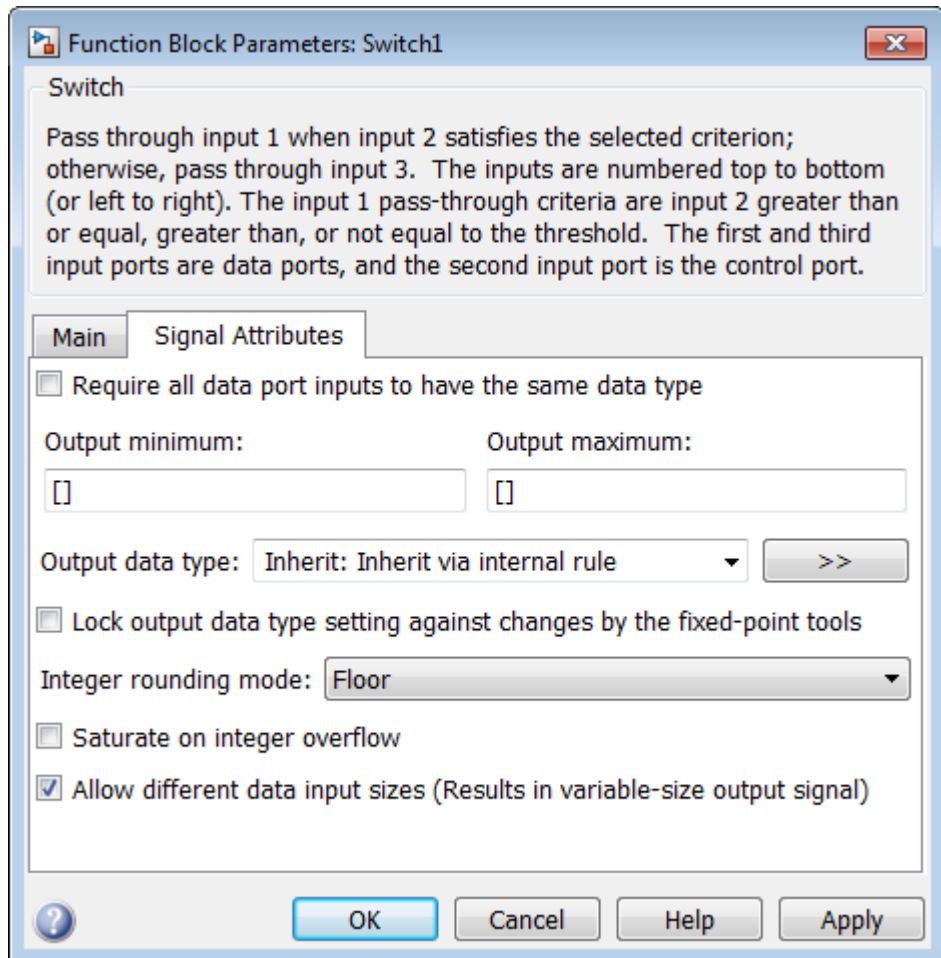


- Action Port
- Enable
- Trigger — **Trigger type** set to function-call

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

See Also

- “Variable-Size Signal Basics” on page 52-2
- “Simulink Models Using Variable-Size Signals” on page 52-6
- “S-Functions Using Variable-Size Signals” on page 52-21
- “Variable-Size Signal Limitations” on page 52-28

Variable-Size Signal Limitations

The following table is a list of known limitations and workarounds.

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 Output 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.	—

See Also

- “Variable-Size Signal Basics” on page 52-2
- “Simulink Models Using Variable-Size Signals” on page 52-6
- “S-Functions Using Variable-Size Signals” on page 52-21
- “Simulink Block Support for Variable-Size Signals” on page 52-24

Customizing Simulink Environment and Printed Models

- Chapter 53, “Customizing the Simulink User Interface”
- Chapter 54, “Frames for Printed Models”

Customizing the Simulink User Interface

- “Add Items to Model Editor Menus” on page 53-2
- “Disable and Hide Model Editor Menu Items” on page 53-16
- “Disable and Hide Dialog Box Controls” on page 53-18
- “Customize the Library Browser” on page 53-24
- “Registering Customizations” on page 53-27

Add Items to Model Editor Menus

In this section...

“About Adding Items” on page 53-2

“Code for Adding Menu Items” on page 53-2

“Define Menu Items” on page 53-4

“Register Menu Customizations” on page 53-10

“Callback Info Object” on page 53-11

“Debugging Custom Menu Callbacks” on page 53-11

“Menu Tags” on page 53-11

About Adding Items

You can add commands and submenus to the following menu locations for the Simulink Editor and Stateflow Editor:

- The end of top-level menus
- The menu bar
- The beginning or end of a context menu

To add an item to an editor menu:

- For each item, create a function, called a *schema function*, that defines the item (see “Define Menu Items” on page 53-4).
- 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 53-10).
- 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’s **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 53-4
- “Defining Custom Submenus” on page 53-8

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 53-11) and create and return an action schema object (see “Action Schema Object” on page 53-5) that specifies the item’s 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's label.
    schema.label = 'My Item 1';
    schema.userdata = 'item1';
    %% Specify the menu item's 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 string that identifies this action, for example, so that it can be referenced by a filter function.
- **label**
String 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**

String specifying text to appear in the editor's status bar when the user selects the menu item that triggers this action.

- **userdata**

Data that you specify. May be of any type.

- **accelerator**

String specifying a keyboard shortcut that a user may use to trigger this action. The string must be of the form 'Ctrl+K', where *K* is the shortcut key. For example, 'Ctrl+T' specifies that the user may invoke this action by holding down the **Ctrl** key and pressing the **T** key.

- **callback**

String 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 string that identifies this toggle action, for example, so that it can be referenced by a filter function.

- **label**
String 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**
String that specifies the state of this toggle action. Valid values are 'Enabled' (default), 'Disabled', and 'Hidden'.
- **statustip**
String specifying text to appear in the editor's 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**
String specifying a keyboard shortcut that a user may use to trigger this action. The string must be of the form 'Ctrl+K', where *K* is the shortcut key. For example, 'Ctrl+T' specifies that the user may invoke this action by holding down the **Ctrl** key and pressing the **T** key.
- **callback**
String 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 53-8) 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 53-4.

```
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’s 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 string that identifies this submenu.
- `label`
String specifying the submenu’s label.
- `state`
String that specifies the state of this submenu. Valid values are 'Enabled' (the default), 'Disabled', and 'Hidden'.
- `statustip`
String specifying text to appear in the editor’s 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 53-4)

- 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 53-11) 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 53-27) to perform this task. In particular, for each menu that you want to customize, your system’s `sl_customization` function must invoke the customization manager’s `addCustomMenuFcn` method (see “Customization Manager” on page 53-27). Each invocation should pass the tag of the menu (see “Menu Tags” on page 53-11) 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 53-10). 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 53-4). 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 53-11) 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. Properties of these objects include

- `uiObject`

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`

Handle to the model being displayed in the editor window.

- `userdata`

User data. The value of this field 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 is a string that identifies a Simulink Editor or the Stateflow Editor menu bar or menu. You need to know a menu's tag to add custom items to it (see "Register Menu Customizations" on page 53-10). You can configure the editor to display all (see "Displaying Menu Tags" on page 53-13) but the following tags:

Tag	Add...
Simulink tags	
Simulink:MenuBar	Menu to Simulink Editor's menu bar
Simulink:PreContextMenu	Item to the beginning of Simulink Editor's context menu
Simulink:ContextMenu	Item to the end of Simulink Editor's context menu
Simulink:FileMenu	Item near the end of the Simulink Editor's File menu, but before the Exit MATLAB item
Simulink:EditMenu	Item to the end of the Simulink Editor's Edit menu
Simulink:ViewMenu	Item to the end of the Simulink Editor's View menu
Simulink:DisplayMenu	Item to the end of the Simulink Editor's Display menu
Simulink:DiagramMenu	Item to the end of the Simulink Editor's Diagram menu
Simulink:SimulationMenu	Item to the end of the Simulink Editor's Simulation menu
Simulink:AnalysisMenu	Item to the end of the Simulink Editor's Analysis menu
Simulink:CodeMenu	Item to the end of the Simulink Editor's Code menu
Simulink:ToolsMenu	Item to the end of the Simulink Editor's Tools menu
Simulink:HelpMenu	Item to the end of Simulink Editor's Help menu
Stateflow tags	
Stateflow:MenuBar	Menu to Stateflow Editor's menu bar
Stateflow:PreContextMenu	Item to the beginning of Stateflow Editor's context menu.

Tag	Add...
Stateflow:ContextMenu	Items to the end of Stateflow Editor's context menu.
Stateflow:FileMenu	Item near the end of the Stateflow Editor's File menu, but before the Exit MATLAB item
Stateflow>EditMenu	Item to the end of Stateflow Editor's Edit menu
Stateflow:ViewMenu	Item to the end of the Stateflow Editor's View menu
Stateflow:DisplayMenu	Item to the end of the Stateflow Editor's Display menu
Stateflow:ChartMenu	Item to the end of the Stateflow Editor's Chart menu
Stateflow:SimulationMenu	Item to the end of the Stateflow Editor's Simulation menu
Stateflow:AnalysisMenu	Item to the end of the Stateflow Editor's Analysis menu
Stateflow:CodeMenu	Item to the end of the Stateflow Editor's Code menu
Stateflow:ToolsMenu	Item to the end of the Stateflow Editor's Tools menu
Stateflow:HelpMenu	Item to the end of the Stateflow Editor's Help menu

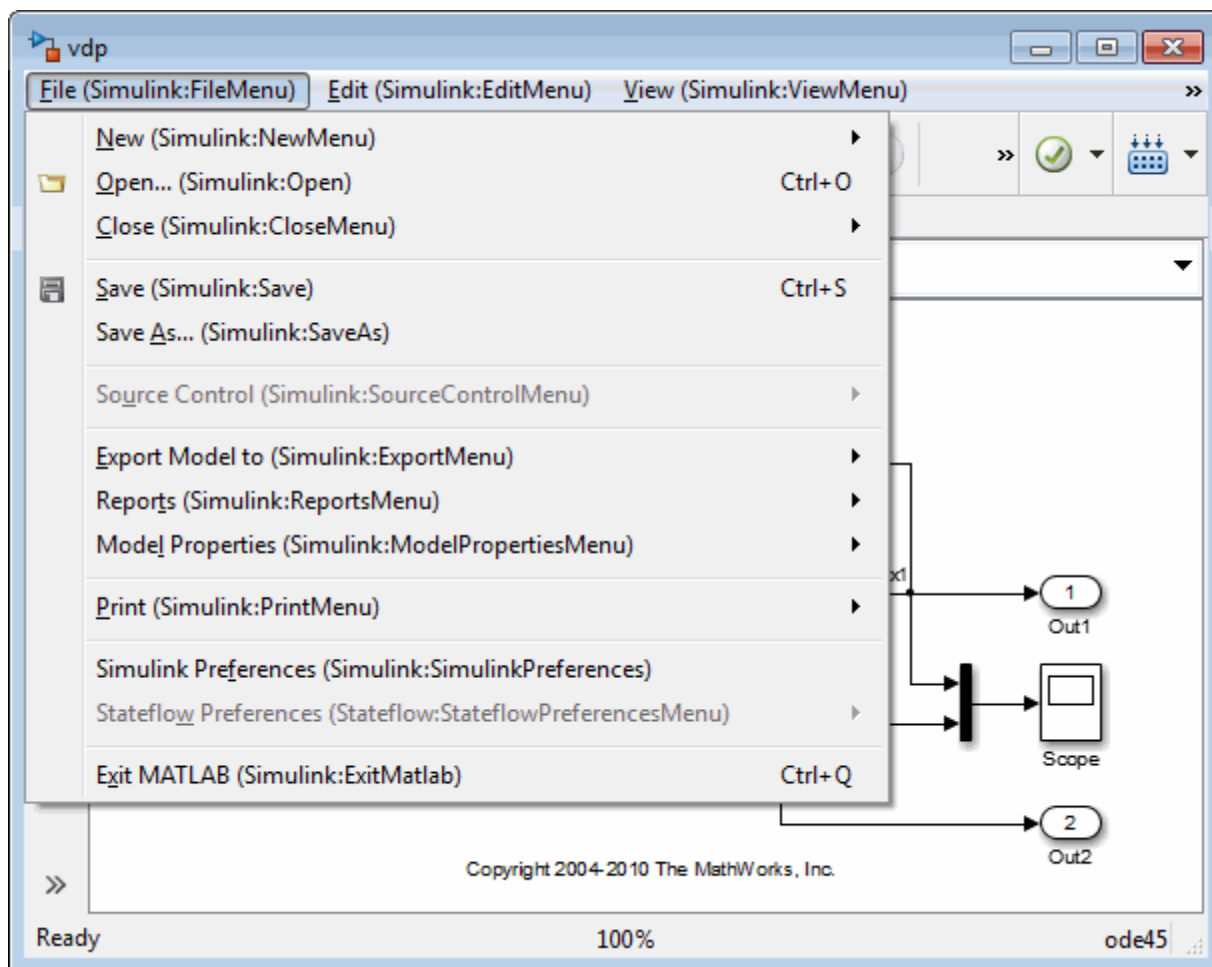
Displaying Menu Tags

You can configure the Simulink and Stateflow software to display the tag for a menu item next to the item's label, allowing you to determine at a glance the tag for a menu. The `Simulink:TagName` customizations appear only if the current editor is the Simulink Editor. The `Stateflow:TagName` customizations appear only if the current editor is the Stateflow Editor.

To configure the editor to display menu tags, at the MATLAB command line, set the customization manager's `showWidgetIdAsToolTip` property to `true`. For example:

```
cm = sl_customization_manager;  
cm.showWidgetIdAsToolTip=true;
```

The tag of each menu item appears next to the item's label on the menu:



To turn off tag display, enter the following command at the command line:

```
cm.showWidgetIdAsToolTip=false;
```

Note Some menu items may not work while menu tag display is enabled. To ensure that all items work, turn off menu tag display before using the menus.

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.

Disable and Hide Model Editor Menu Items

In this section...

“About Disabling and Hiding Model Editor Menu Items” on page 53-16

“Example: Disabling the New Model Command on the Simulink Editor’s File Menu” on page 53-16

“Creating a Filter Function” on page 53-16

“Registering a Filter Function” on page 53-17

About Disabling and Hiding Model Editor Menu Items

You can disable or hide items that appear on the Simulink model editor menus. To disable or hide a menu item, you must:

- Create a filter function that disables or hides the menu item (see “Creating a Filter Function” on page 53-16).
- Register the filter function with the customization manager (see “Registering a Filter Function” on page 53-17).

For more information on Model Editor menu items, see:

Example: Disabling the New Model Command on the Simulink Editor’s File Menu

```
function sl_customization(cm)
    cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end

function state = myFilter(callbackInfo)
    state = 'Disabled';
end
```

Creating a Filter Function

Your filter function must accept a callback info object and return a string that specifies the state that you want to assign to the menu 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 the item, it is 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. The 'Disabled' state is of middling strength. It overrides 'Enabled' but is itself overridden by 'Hidden'. For example, if any filter function or Simulink or Stateflow assigns 'Disabled' to a menu item and none assigns 'Hidden' to the item, the item is disabled.

Note The Simulink software does not allow you to filter some menu items, for example, the **Exit MATLAB** item on the Simulink **File** menu. An error message is displayed if you attempt to filter a menu item that you are not allowed to filter.

Registering a Filter Function

Use the customization manager's `addCustomFilterFcn` method to register a filter function. The `addCustomFilterFcn` method takes two arguments: a tag that identifies the menu or menu item to be filtered (see “Displaying Menu Tags” on page 53-13) 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 **File** menu.

```
function sl_customization(cm)
    cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end
```

Disable and Hide Dialog Box Controls

In this section...

“About Disabling and Hiding Controls” on page 53-18

“Disable a Button on a Dialog Box” on page 53-19

“Write Control Customization Callback Functions” on page 53-20

“Dialog Box Methods” on page 53-20

“Dialog Box and Widget IDs” on page 53-21

“Register Control Customization Callback Functions” on page 53-22

About Disabling and Hiding Controls

The Simulink product 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 of its dialog boxes. The customization API allows you to disable or hide controls on an entire class of dialog boxes, for example, parameter dialog boxes via a single method call.

Before attempting to customize a Simulink dialog box or class of dialog boxes, you must first ensure 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 standalone dialog box (i.e., one that does not appear in Model Explorer) is customizable, open the dialog box, enable dialog and widget ID display (see “Dialog Box and Widget IDs” on page 53-21), and position the mouse over a widget. If a widget ID appears, the dialog box is customizable.

Once you have determined that a dialog box or class of dialog boxes is customizable, you must 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 53-20) and registering the callback functions via an object called the customization manager (see “Register Control Customization Callback Functions” on page 53-22). Simulink then invokes

the callback functions to disable or hide the controls whenever a user opens the dialog boxes.

For more information on Dialog Box controls, see:

Disable a Button on a Dialog Box

The following `sl_customization.m` file disables the **Build** 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.
cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)
% Disable for Configuration Parameters dialog box that appears in
% the Model Explorer.
cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end

function disableRTWBuildButton(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({'Simulink.RTWCC.Build'})
    end
end

end
```

To test this customization:

- 1** Put the preceding `sl_customization.m` file on the path.
- 2** Register the customization by entering `sl_refresh_customizations` at the command line or by restarting the MATLAB software (see “Registering Customizations” on page 53-27).

- 3 Open the `sldemo_engine` model, for example, by entering the command `sldemo_engine` at the command line.

Write Control Customization Callback Functions

A callback function for disabling or hiding controls on a dialog box should accept 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, the following callback function uses these objects to disable the **Build** button on the **Code Generation** pane of the Configuration Parameters dialog box displayed in Model Explorer for any model whose name contains “engine.”

```
function disableRTWBuildButton(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({'Simulink.RTWCC.Build'})
    end
```

Dialog Box Methods

Dialog box objects provide the following methods for enabling, disabling, and hiding controls:

- `disableWidgets(widgetIDs)`
- `hideWidgets(widgetIDs)`

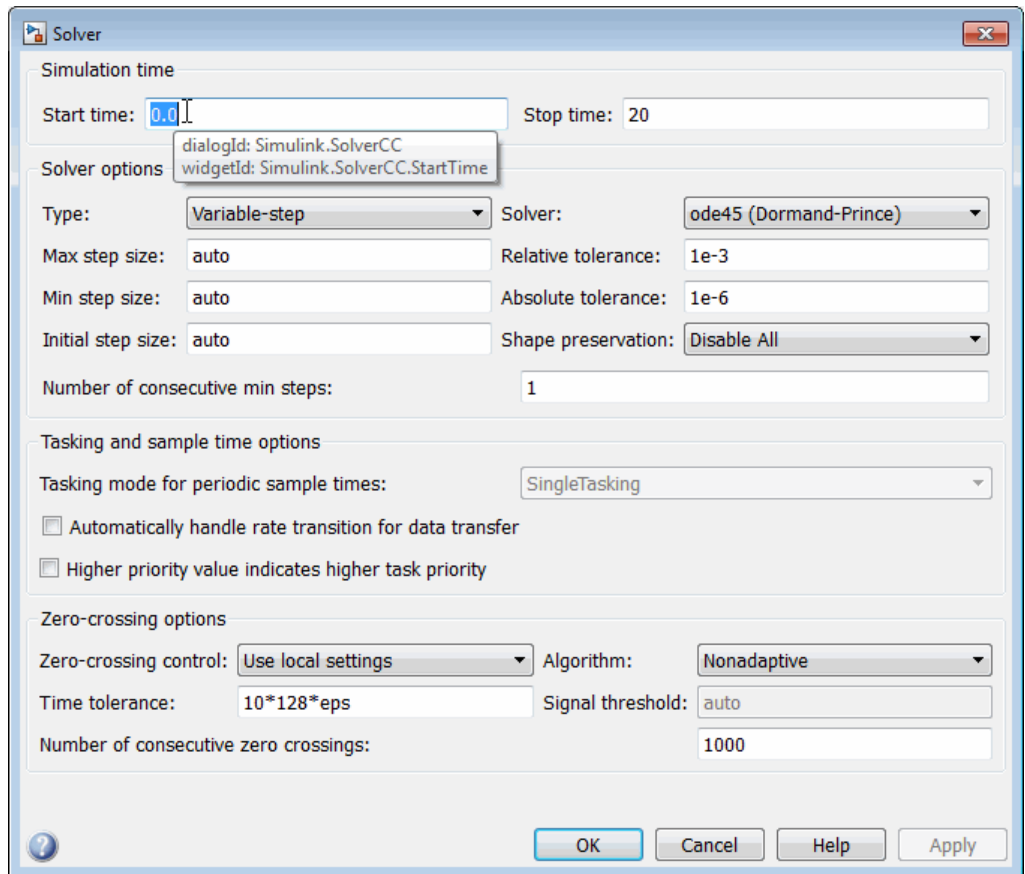
where `widgetIDs` is a cell array of widget identifiers (see “Dialog Box and Widget IDs” on page 53-21) that specify the widgets to be disabled or hidden.

Dialog Box and Widget IDs

Dialog box and widget IDs are strings that identify a control on a Simulink dialog box. To determine the dialog box and widget ID for a particular control, execute the following code at the command line:

```
cm = sl_customization_manager;  
cm.showWidgetIdAsToolTip = true
```

Then, open the dialog box that contains the control and move the mouse cursor over the control to display a tooltip listing the dialog box and the widget IDs for the control. For example, moving the cursor over the **Start time** field on the **Solver** pane of the Configuration Parameters dialog box reveals that the dialog box ID for the **Solver** pane is `Simulink.SolverCC` and the widget ID for the **Start time** field is `Simulink.SolverCC.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 a particular installation of the Simulink product, include code in the installation’s `sl_customization.m` file (see “Registering Customizations” on page 53-27) that invokes the customization manager’s `addDlgPreOpenFcn` on the callbacks.

The `addDlgPreOpenFcn` takes two arguments. The first argument is a dialog box ID (see “Dialog Box and Widget IDs” on page 53-21) and the second is a pointer to the callback function to be registered. Invoking this method causes the registered function to be invoked for each dialog box of the type specified by the dialog box ID. The function is invoked before the dialog box is opened, allowing the function to perform the customizations before they become visible to the user.

The following example registers a callback that disables the **Build** button on the **Code Generation** pane of the Configuration Parameters dialog box (see “Write Control Customization Callback Functions” on page 53-20).

```
function sl_customization(cm)

    % Disable for standalone Configuration Parameters dialog box.
    cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)

    % Disable for Configuration Parameters dialog box that appears in
    % the Model Explorer
    cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end
```

Note Registering a customization callback causes the Simulink software to invoke the callback for every instance of the class of dialog boxes specified by the method’s dialog box ID argument. This allows you to 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. This is because most built-in block dialog boxes are instances of the same dialog box super class.

Customize the Library Browser

In this section...

“Reorder Libraries” on page 53-24

“Disable and Hide Libraries” on page 53-24

“Customize the Library Browser Menu” on page 53-25

Reorder Libraries

The order in which a library appears in the Library Browser is determined by its name and its sort priority. Libraries appear in the Library Browser’s tree view in ascending order of priority, with all blocks having the same priority sorted alphabetically. The Simulink library has a sort priority of -1 by default; all other libraries, a sort priority of 0. This guarantees that the Simulink library is by default the first library displayed in the Library Browser. You can reorder libraries by changing their sort priorities. To change library sort priorities, insert a line of code of the following form in an `sl_customization.m` file (see “Registering Customizations” on page 53-27) on the MATLAB path:

```
cm.LibraryBrowserCustomizer.applyOrder( {'LIBNAME1', PRIORITY1, ...
                                         'LIBNAME2', 'PRIORITY2, ...
                                         .
                                         .
                                         'LIBNAMEN', PRIORITYN} );
```

where `LIBNAMEn` is the name of the library or its model file and `PRIORITYn` is an integer indicating the library’s sort priority. For example, the following code moves the Simulink Extras library to the top of the Library Browser’s tree view.

```
cm.LibraryBrowserCustomizer.applyOrder( {'Simulink Extras', -2} );
```

Disable and Hide Libraries

To disable or hide libraries, sublibraries, or library blocks, insert code of the following form in an `sl_customization.m` file (see “Registering Customizations” on page 53-27) on the MATLAB path:

```
cm.LibraryBrowserCustomizer.applyFilter( {'PATH1', 'STATE1', ...
                                         'PATH2', 'STATE2', ...
                                         .
                                         .
                                         'PATHN', 'STATEN'} );
```

where PATH_n is the path of the library, sublibrary, or block to be disabled or hidden and 'STATE_n' is 'Disabled' or 'Hidden'. For example, the following code hides the Simulink Sources sublibrary and disables the Sinks sublibrary.

```
cm.LibraryBrowserCustomizer.applyFilter({'Simulink/Sources','Hidden'});
cm.LibraryBrowserCustomizer.applyFilter({'Simulink/Sinks','Disabled'});
```

Customize the Library Browser Menu

You can perform the same kinds of customizations to the Library Browser menu as you can to the model and Stateflow editor menus. Simply use the corresponding Library Browser menu tags to perform the customizations.

- LibraryBrowser:FileMenu
- LibraryBrowser>EditMenu
- LibraryBrowser:ViewMenu
- LibraryBrowser:HelpMenu

For example, the following code adds a menu item to the Library Browser's file menu:

```
%Menu customization:
% Add items to the Library Browser File menu
cm.addCustomMenuFcn('LibraryBrowser:FileMenu',@getMyMenuItems)

%% Define the custom menu function.
function schemaFcns = getMyMenuItems(callbackInfo)
schemaFcns = {@myBasic};
end

%% Define the schema function for first menu item.
function schema = myBasic(callbackInfo)
disp('1');
```

```
schema = sl_action_schema;  
schema.label = 'Display 1';  
schema.userdata = 'item one';  
schema.tag = 'LibraryBrowser:ItemOne';  
end
```

Registering Customizations

In this section...

“About Registering User Interface Customizations” on page 53-27

“Customization Manager” on page 53-27

About Registering User Interface Customizations

You must register your user interface customizations using a MATLAB function called `sl_customization.m`. This is located on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function should accept one argument: a handle to a customization manager object. For example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering menu and control customizations (see “Customization Manager” on page 53-27). Your instance of the `sl_customization` function should use these methods to register customizations specific to your application. For more information, see the following sections on performing customizations.

- “Add Items to Model Editor Menus” on page 53-2
- “Disable and Hide Model Editor Menu Items” on page 53-16
- “Disable and Hide Dialog Box Controls” on page 53-18

The `sl_customization.m` file is read when the Simulink software starts. If you subsequently change the `sl_customization.m` file, you must restart the Simulink software or enter the following command at the command line to effect the changes:

```
sl_refresh_customizations
```

Customization Manager

The customization manager includes the following methods:

- `addCustomMenuFcn(stdMenuTag, menuSpecsFcn)`

Adds the menus specified by `menuSpecsFcn` to the end of the standard Simulink menu specified by `stdMenuTag`. The `stdMenuTag` argument is a string that specifies the menu to be customized. For example, the `stdMenuTag` for the Simulink editor's **Tools** menu is `'Simulink:ToolsMenu'` (see “Displaying Menu Tags” on page 53-13 for more information). The `menuSpecsFcn` argument is a handle to a function that returns a list of functions that specify the items to be added to the specified menu. See “Add Items to Model Editor Menus” on page 53-2 for more information.

- `addCustomFilterFcn(stdMenuItemID, filterFcn)`

Adds a custom filter function specified by `filterFcn` for the standard Simulink model editor menu item specified by `stdMenuItemID`. The `stdMenuItemID` argument is a string that identifies the menu item. For example, the ID for the **New Model** item on the Simulink editor's **File** menu is `'Simulink:NewModel'` (see “Displaying Menu Tags” on page 53-13 for more information). The `filterFcn` argument is a pointer to a function that hides or disables the specified menu item. See “Disable and Hide Model Editor Menu Items” on page 53-16 for more information.

Frames for Printed Models

- “Print Frames” on page 54-2
- “Create a Print Frame” on page 54-6
- “Add Rows and Cells to Print Frames” on page 54-7
- “Add Content to Print Frame Cells” on page 54-9
- “Print Using Print Frames” on page 54-13

Print Frames

In this section...

“What are Print Frames?” on page 54-2

“PrintFrame Editor” on page 54-3

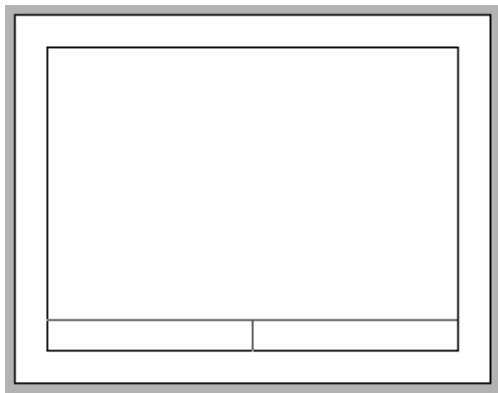
“Single Use or Multiple Use Print Frames” on page 54-5

“Text and Variable Content” on page 54-5

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:

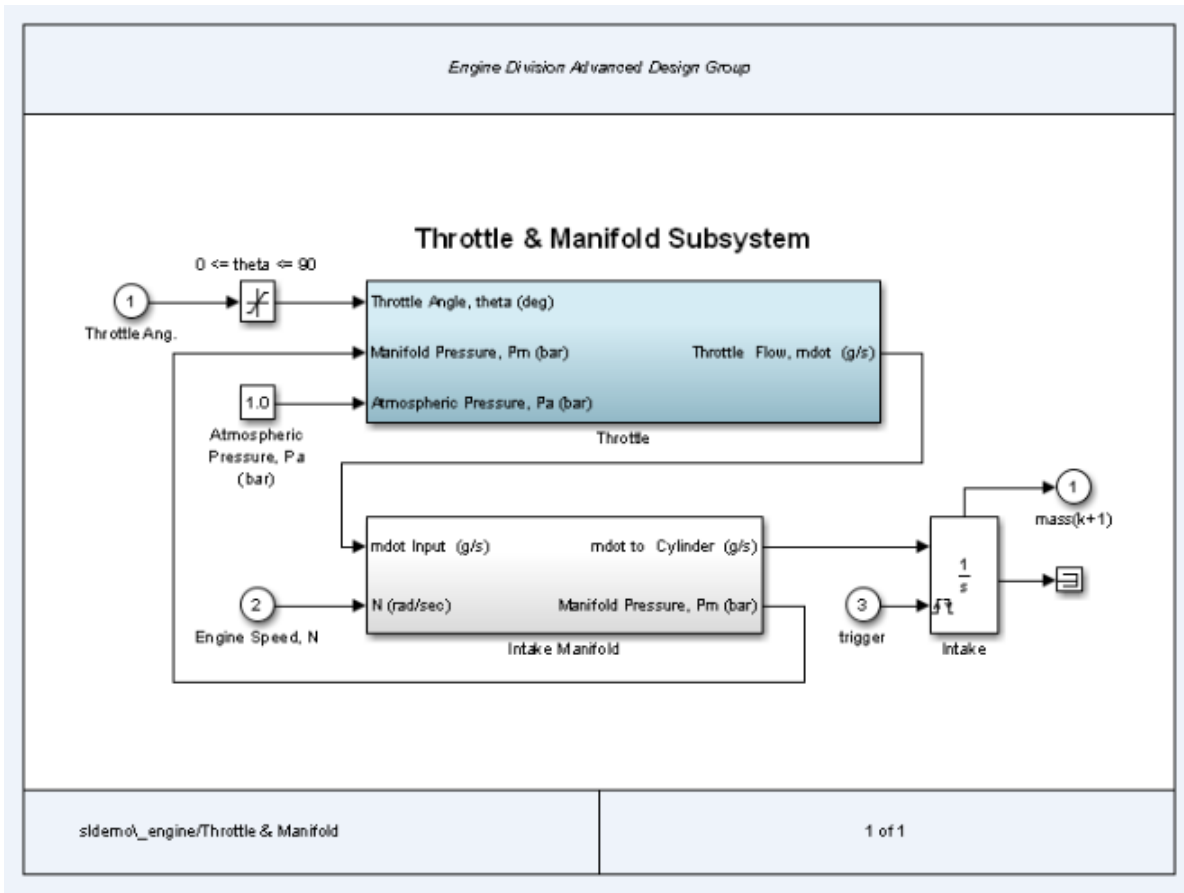


Rows contain one or more cells. You can add content entries to cells. You can also add new rows and cells.

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

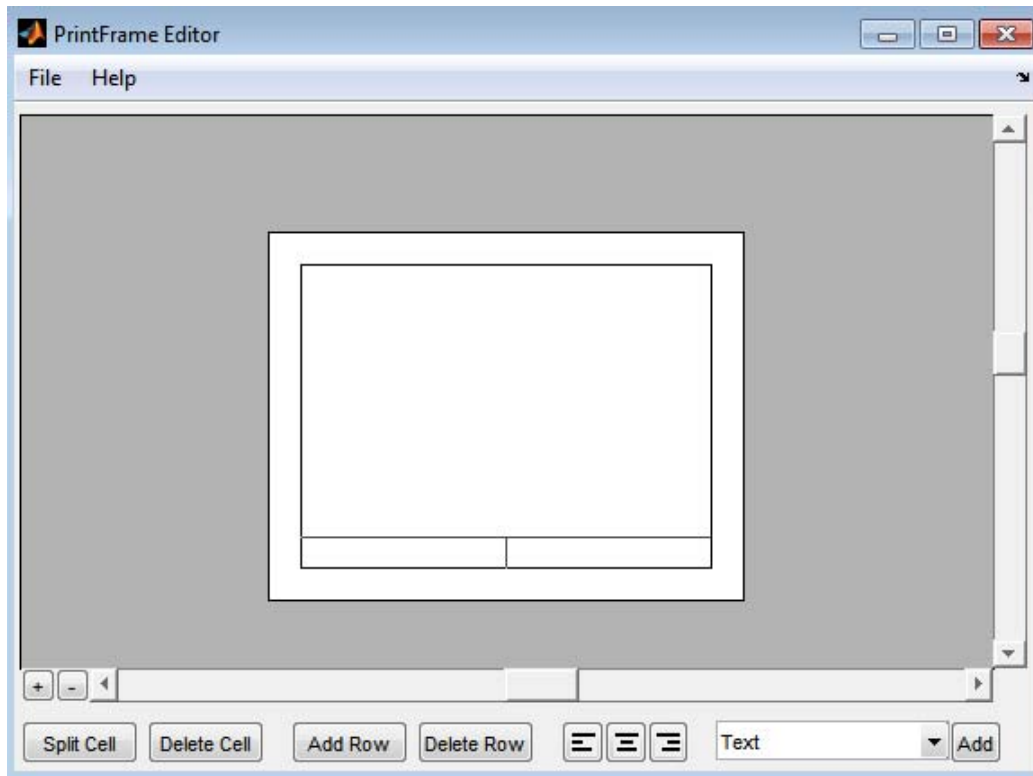
- A bottom row, in which one cell has the path to the subsystem and another cell has the page number



PrintFrame Editor

Use the PrintFrame Editor to create and edit print frames.

To open the PrintFrame Editor, at the MATLAB command line, enter the `framedit` 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).

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 54-7
 - “Add Content to Print Frame Cells” on page 54-9
 - 4 Save the print frame as a `.fig` file. Select **File > Save As**.

Add Rows and Cells to Print Frames

In this section...
“Add and Remove Rows” on page 54-7
“Add and Remove Cells” on page 54-7
“Resize Rows and Cells” on page 54-8

Tip Specify the print frame page setup before you create rows and cells or add content (see “Create a Print Frame” on page 54-6).

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 54-6.

Add Content to Print Frame Cells

In this section...
“Types of Content” on page 54-9
“Add Content to Cells” on page 54-9
“Block Diagram” on page 54-10
“Variables” on page 54-11
“Text” on page 54-12
“Format Content in Cells” on page 54-12

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 54-10
- “Variables” on page 54-11
- “Text” on page 54-12

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 54-12.

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 54-10.
- **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 Properties).

Format Content in Cells

You can align cell contents using the left, center, and right alignment buttons. (Block diagrams are always center aligned.)

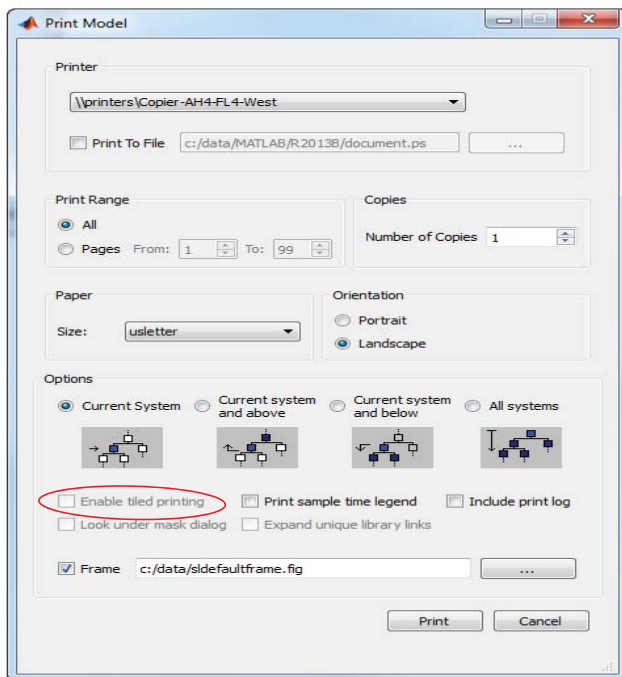
You can change font properties, such as size or style (for example, italics or bold). To change font properties, select the cell, then right-click the contents and use the context menu to format the text.

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 54-6.

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

- 1 In the Simulink Editor or Stateflow Editor, select **File > Print > 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.

Running Models on Target Hardware

- Chapter 55, “About Run on Target Hardware Feature”
- Chapter 56, “Arduino Hardware”
- Chapter 57, “BeagleBoard Hardware”
- Chapter 58, “Gumstix Overo Hardware”
- Chapter 59, “LEGO MINDSTORMS NXT Hardware”
- Chapter 60, “PandaBoard Hardware”
- Chapter 61, “Raspberry Pi Hardware”

About Run on Target Hardware Feature

- “What is “Run on Target Hardware?”” on page 55-2
- “Open Examples for This Support Package” on page 55-4
- “Support Packages and Support Package Installer” on page 55-10
- “Install This Support Package on Other Computers” on page 55-12
- “Tune and Monitor Models Running on Target Hardware” on page 55-14
- “Troubleshoot Running Models on Target Hardware” on page 55-19

What is “Run on Target Hardware”?

Run on Target Hardware is a feature of the Simulink product that runs a Simulink model on *target hardware*.

At the time of publication, Simulink software supports the following target hardware:

- Arduino® Mega 2560, Arduino Uno, and Arduino Nano
- BeagleBoard Xm, BeagleBoard Bx, and BeagleBoard Cx
- Gumstix® Overo
- LEGO™ MINDSTORMS® NXT
- PandaBoard
- Raspberry Pi

You can use Support Package Installer to install support for your target hardware, such as:

- Simulink block libraries
- Parameters on the **Run on Target Hardware** pane in the Configuration Parameters dialog
- Third-party software such as compiler toolchains and software libraries
- Examples for learning how to use Simulink with the target hardware

For more information about support packages, see “Support Packages and Support Package Installer” on page 55-10.

For more information about using Simulink software with your target hardware, see:

- “Arduino Hardware”
- “BeagleBoard Hardware”
- “Gumstix Overo® Hardware”
- “LEGO MINDSTORMS NXT Hardware”

- “PandaBoard Hardware”
- “Raspberry Pi™ Hardware”

After installing the appropriate support package, you can run a model on your target hardware using the following menu items under **Tools > Run on Target Hardware**:

- **Prepare to Run** — Configure your model to run on a specific type of target hardware. Clicking **Prepare to Run** automatically opens the Configuration Parameters dialog, changes the **System target file** parameter to `realtime.tlc`, and displays the **Run on Target Hardware** pane. Then, use the **Target hardware** parameter to specify the type of target hardware you are using, and save the changes to your model. Clicking **Prepare to Run** changes the model Configuration Parameters. If you are working with a model that has a configuration you want to preserve, create a backup copy of your model before using clicking **Prepare to Run**.
- **Options** — This menu item opens the Run on Target Hardware pane in the model Configuration Parameters dialog. You can use this pane to specify the target hardware and configure optional settings, such as External mode.
- **Install/Update Support Package** — Open Support Package Installer with options for installing support packages.
- **Update Firmware** — Open Support Package Installer with options for updating firmware and setting up third-party software.

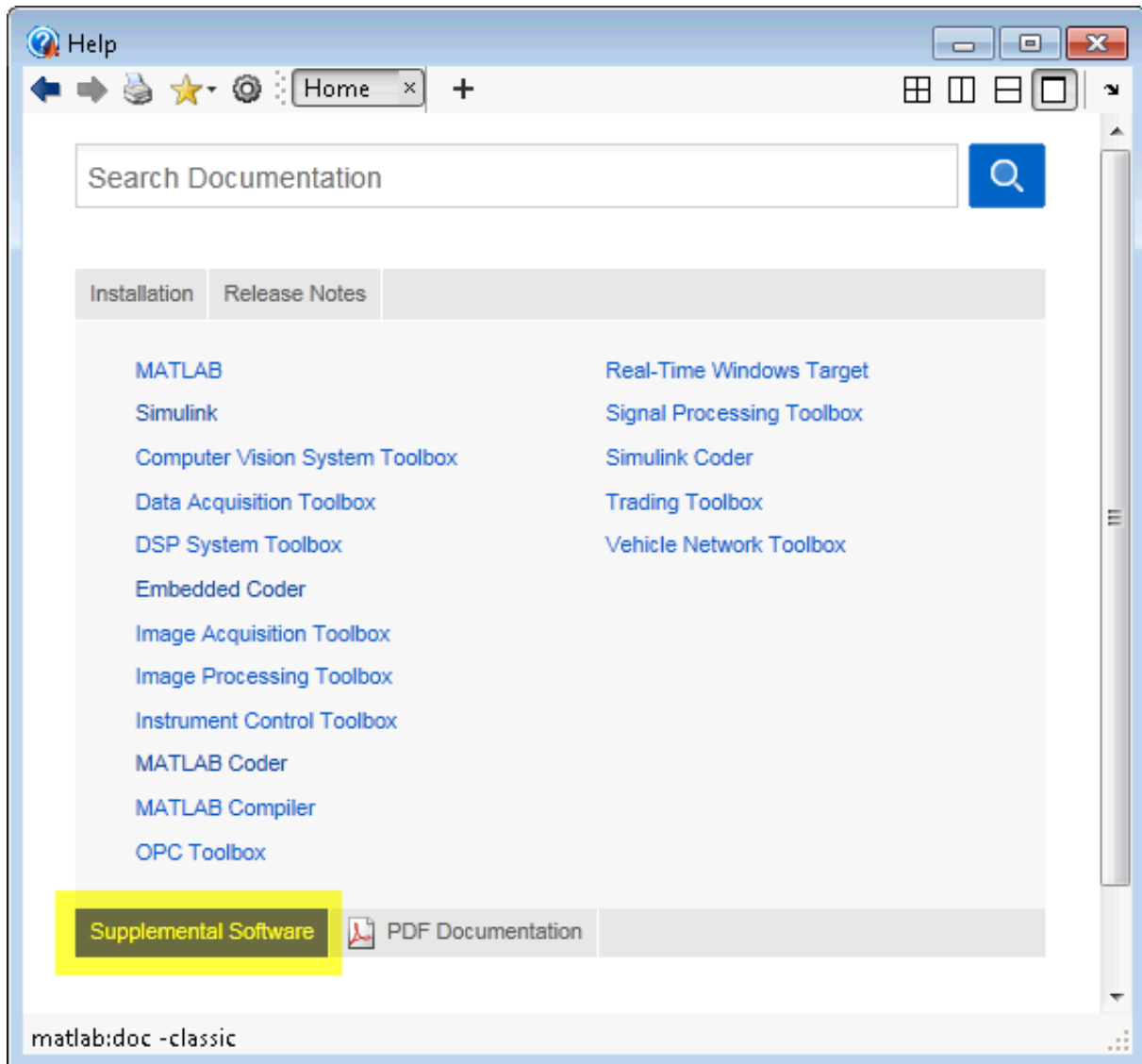
Open Examples for This Support Package

In this section...
“Using the Help Browser” on page 55-4
“Using the Block Library” on page 55-6
“Using Support Package Installer” on page 55-7

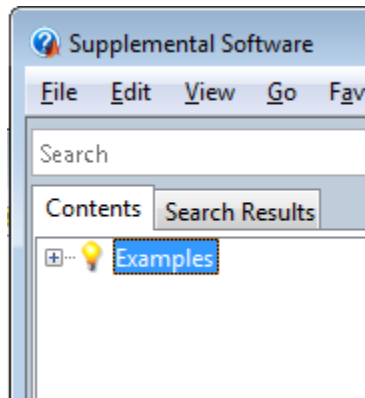
Using the Help Browser

You can open support package examples from the Help browser:

- 1 Enter `doc` in the MATLAB Command Window.
- 2 In the Help browser, click **Supplemental Software** in the lower left corner of the Home page.



3 In Supplemental Software, double-click **Examples**.



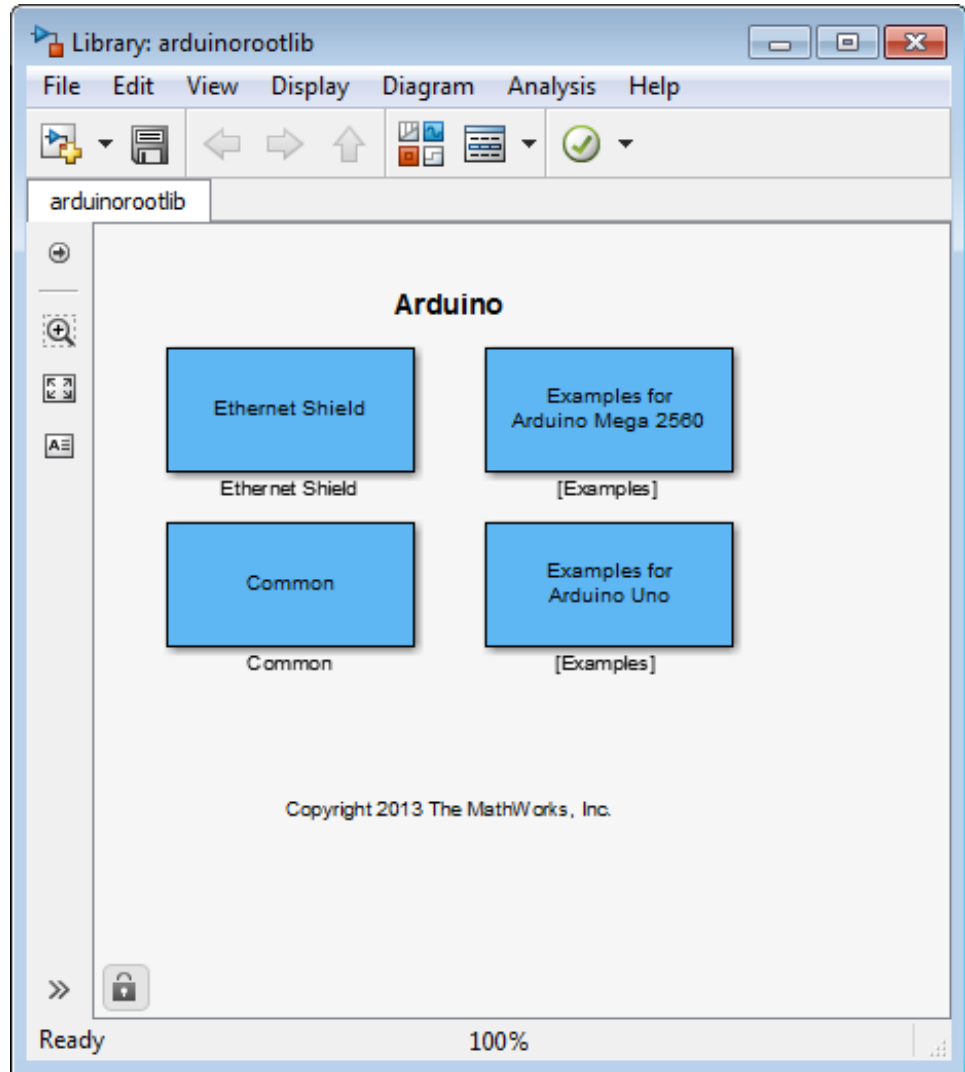
- 4 Select the examples for your support package.

Note For other types of examples, open the Help browser and search for your product name followed by “examples”.

Using the Block Library

To open support package from the support package block library:


- 1 Enter `simulink` in the MATLAB Command Window.
- 2 In Simulink Library Browser, open the support package block library.
- 3 In the block library, double-click the Examples block.



Using Support Package Installer

Support Package Installer (supportPackageInstaller) automatically displays the support package examples when you complete the process of installing and setting up a support package.

On the last screen in Support Package Installer, leave **Show support package examples** enabled and click **Finish**.

 Support Package Installer

Support package setup complete

You have completed the setup tasks.

Show support package examples

Support Packages and Support Package Installer

What Is a Support Package?

A *support package* is an add-on that enables you to use a MathWorks product with specific third-party hardware and software.

Support packages can include:

- Simulink block libraries
- MATLAB functions, classes, and methods
- Firmware updates for the third-party hardware
- Automatic installation of third-party software
- Examples and tutorials

A *support package file* has a *.zip extension. This type of file contains MATLAB files, MEX files, and other supporting files required to install the support package. Use Support Package Installer to install these support package files.

A *support package installation file* has a *.mlpkginstall extension. You can double click this type of file to start Support Package Installer, which preselects a specific support package for installation. You can download these files from MATLAB Central File Exchange and use them to share support packages with others.

What Is Support Package Installer?

Support Package Installer is a wizard that guides you through the process of installing support packages.

You can use Support Package Installer to:

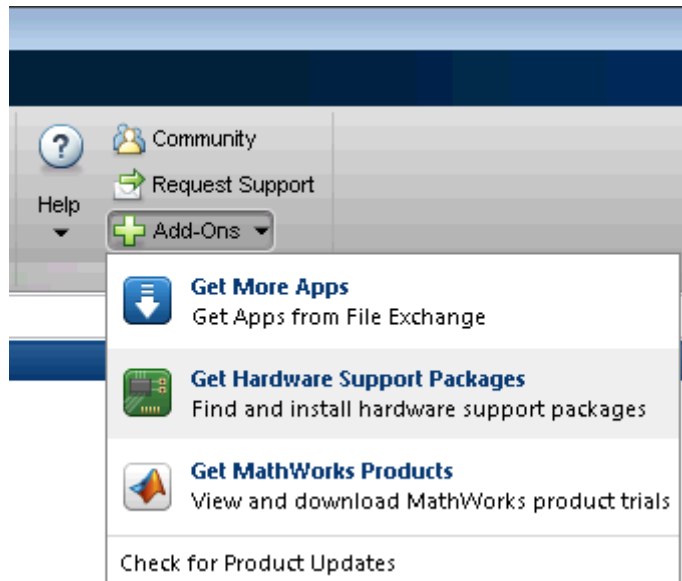
- Display a list of available, installable, installed, or updatable support packages
- Install, update, download, or uninstall a support package.
- Update the firmware on specific third-party hardware.

- Provide your MathWorks software with information about required third-party software.

If third-party software is included, Support Package Installer displays a list of the software and licenses for you to review before continuing.

You can start Support Package Installer in one of the following ways:

- On the MATLAB toolstrip, click **Add-Ons > Get Hardware Support Packages**.



- In the MATLAB Command Window, enter `supportPackageInstaller`.
- Double-click a support package installation file (*.mlpkginstall).

Install This Support Package on Other Computers

You can download a support package to one computer, and then install it on other computers. You can use this approach to:

- Save time when installing support packages on multiple computers.
- Install support packages on computers that are not connected to the Internet.

Before starting, select a computer to use for downloading. This computer must have the same base product license and platform as the computers upon which you are installing the support package. For example, suppose you want to install a Simulink support package on a group of computers that are running 64-bit Windows. To do so, you must first download the support package using a computer that has a Simulink license and is running 64-bit Windows.

Download the support package to one computer:

- 1** In the MATLAB Command Window, enter `supportPackageInstaller`.
- 2** In Support Package Installer, on the **Select an action** screen, choose **Install from Internet** or **Download from Internet**. Click **Next**.
- 3** On the following screen, select only one support package.

Notice the path of the **Download folder**. For example, `C:\MATLAB\SupportPackages\R2013b\downloads`.

- 4** Using the file manager on your computer, open the downloads folder and observe its contents.
- 5** Using Support Package Installer, complete the installation or download process.

This process creates a folder within the **Download folder**. In some cases, if the support package requires another support package, this process creates an additional folder.

Prepare and share the support package files:

- 1 In the file manager, check how many folders were created during the installation or download process.
- 2 If more than one folder was created, combine the contents of the folders into the folder named after the support package.

For example,

C:\MATLAB\SupportPackages\R2013b\downloads*support_package_name*.

- 3 Make that folder available to other computers by sharing it on the network, or copying it to portable media, such as a USB flash drive.

Note Some support packages require that you install third-party software separately before completing the support package installation process. In that case, also make the third-party software available for installation on the other computers.

Install the support package on the other computers:

- 1 Run Support Package Installer on the other computer or computers.
- 2 On the **Install or update support package** screen, select the **Folder** option.
- 3 Use **Browse** to specify the location of the support package folder on the network or portable media.
- 4 Complete the instructions provided by Support Package Installer.

Tune and Monitor Models Running on Target Hardware

In this section...

“Overview of Using External Mode” on page 55-14

“Run Your Simulink Model in External Mode” on page 55-15

“Stop External Mode” on page 55-16

“External Mode Control Panel” on page 55-17

Overview of Using External Mode

You can use External mode to tune parameters and monitor a model running on your target hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on model results in real time. Doing so helps you find the optimal values to achieve desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to rerun the model each time you change parameters. You can also use External mode to develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

This workflow lists the tasks usually required to tune parameters with External mode:

- 1** In the model on your host computer, enable External mode.
- 2** (Optional) Place one or more sink blocks in your model. For example, use Display or Scope blocks to visualize data, or use a To File block to log signal data.
- 3** Give the Simulink software command to run the model on the target hardware.
- 4** (Optional) Observe the data External mode sends from the target hardware to sink blocks in the model on the host computer.

- 5 Change and apply parameter values in the model on your host computer.
- 6 Find the optimal parameter values by making adjustments and observing the results.
- 7 Save the new parameter values, disable External mode, and save the model.

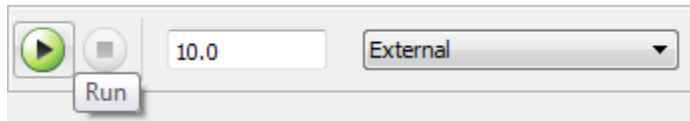
Run Your Simulink Model in External Mode

Note If you have the Embedded Coder or Simulink Coder software, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

- 1 Connect the target hardware to your host computer.

Different types of target hardware can use different types of connections. Check the External mode topic for your target hardware to determine which type of connection to use.

- 2 On the model toolbar, set **Simulation mode** to External mode.



- 3 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 4 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
```

```
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\sl_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

5 Change parameter values in the model on your host computer.

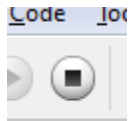
Observe the corresponding changes in the model running on the hardware.

Any Simulink Sinks blocks in your model receive data from the hardware and display it on your host computer.

Note External mode increases the processing burden of the model running on the board, and can cause overruns.

Stop External Mode

To stop the model running in External mode, click the black square Stop button located on the model toolbar, as shown here.



This action stops the process for the model running on the target hardware, and stops the model simulation running on your host computer.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

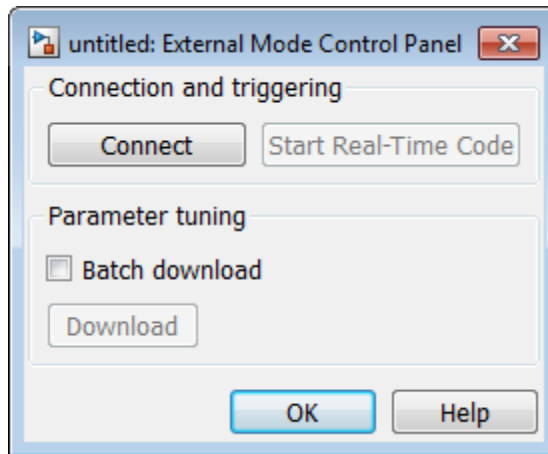
When you are finished using External mode, set **Simulation mode** back to Normal.

External Mode Control Panel

Using External Mode Control Panel provides additional control of External mode operations, including:

- Connect or disconnect the model on the host computer to/from the model running on the target hardware.
- Start and stop the model running on the target hardware.
- Gather changes to parameter values in a batch before applying them concurrently to the model running on the target hardware.

To open the External Mode Control Panel dialog box, in the model window, select **Code > External Mode Control Panel**.



The **Connect/Disconnect** button connects or disconnects the model on your host computer to/from the model running on the target hardware. If the model is not running on the target hardware, clicking **Connect** automatically deploys and runs the model on the target hardware.

- The **Start Real-Time Code/Stop Real-Time Code** button starts or stops the model running on the target hardware.
- **Batch download** enables you to gather changes before using the **Download** button to simultaneously apply those changes to the model running on the hardware:

- While **Batch download** is disabled, clicking **OK** or **Apply** in a block dialog box sends updated block parameter values from the block to the model running on the target hardware.
- When you enable **Batch download**, clicking **OK** or **Apply** in a block dialog box stores updated block parameter values on the host computer. You can complete a set of changes before clicking **Download** to simultaneously send all of the updated values to the model running on the target hardware. This feature is useful for avoiding error conditions when a model contains blocks whose parameter values must be changed concurrently.

External Mode Control Panel displays `Parameter changes pending...` to the right of the **Download** button until the model running on the target hardware has applied the new parameter values.

Troubleshoot Running Models on Target Hardware

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.

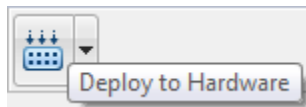
This is the expected behavior.

For example, in a model, if you select **Simulation > Mode > Normal** and then select **Simulation > 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, select **Tools > Run on Target Hardware > Prepare to Run**. Then, click the Deploy to Hardware button.



To use External mode, select **Simulation > Mode > External mode**. Then, select **Simulation > Run**.

Arduino Hardware

- “Install Support for Arduino Hardware” on page 56-2
- “Open Block Libraries for Arduino Hardware” on page 56-3
- “Connect Arduino Ethernet Shield to Arduino Hardware” on page 56-8
- “Configure Network Settings for Arduino Ethernet Shield” on page 56-10
- “Run Model on Arduino Hardware” on page 56-12
- “Stop or Restart a Model on the Arduino Hardware” on page 56-13
- “Tune and Monitor Model Running on Arduino Mega 2560 Hardware” on page 56-14
- “Use Serial Communications with Arduino Hardware” on page 56-18
- “Detect and Fix Task Overruns on Arduino Hardware” on page 56-21
- “Troubleshoot Running Models on Arduino Hardware” on page 56-23
- “Configure Host COM Port Manually” on page 56-26

Install Support for Arduino Hardware

This example shows how to add support for Arduino hardware to the Simulink product. After you complete this process, you can run Simulink models on your Arduino hardware.

To install support for Arduino hardware:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2 Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

The installation process adds the following items to your host computer:

- Third-party software development tools, such as the Arduino software
- A Simulink block library for configuring and accessing Arduino sensors, actuators, and communication interfaces
- Examples for getting started and learning about specific features

To check for updates, repeat this process when a new version of MATLAB software is released. You can also check for updates between releases.

Open Block Libraries for Arduino Hardware

You can open the block libraries for your Arduino hardware from the MATLAB Command Window or from the Simulink Library Browser.

The blocks in these block libraries provide support for various peripherals available on the Arduino hardware.

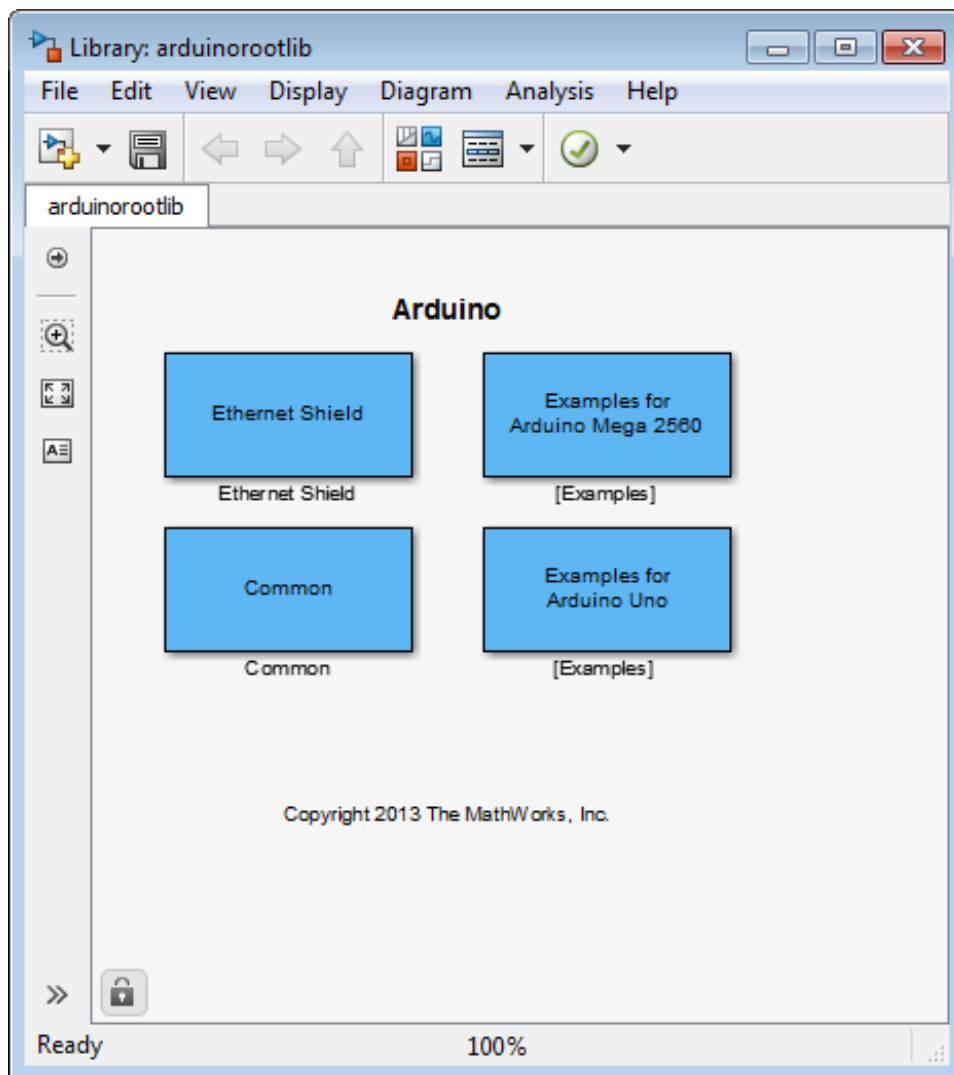
From the Command Line

After installing support for your Arduino hardware, you can open its block library from the MATLAB Command Window.

After installing support for Arduino hardware, enter:

```
arduinorootlib
```

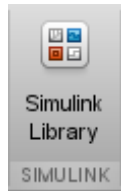
The software opens the corresponding block library.



From the Simulink Library Browser

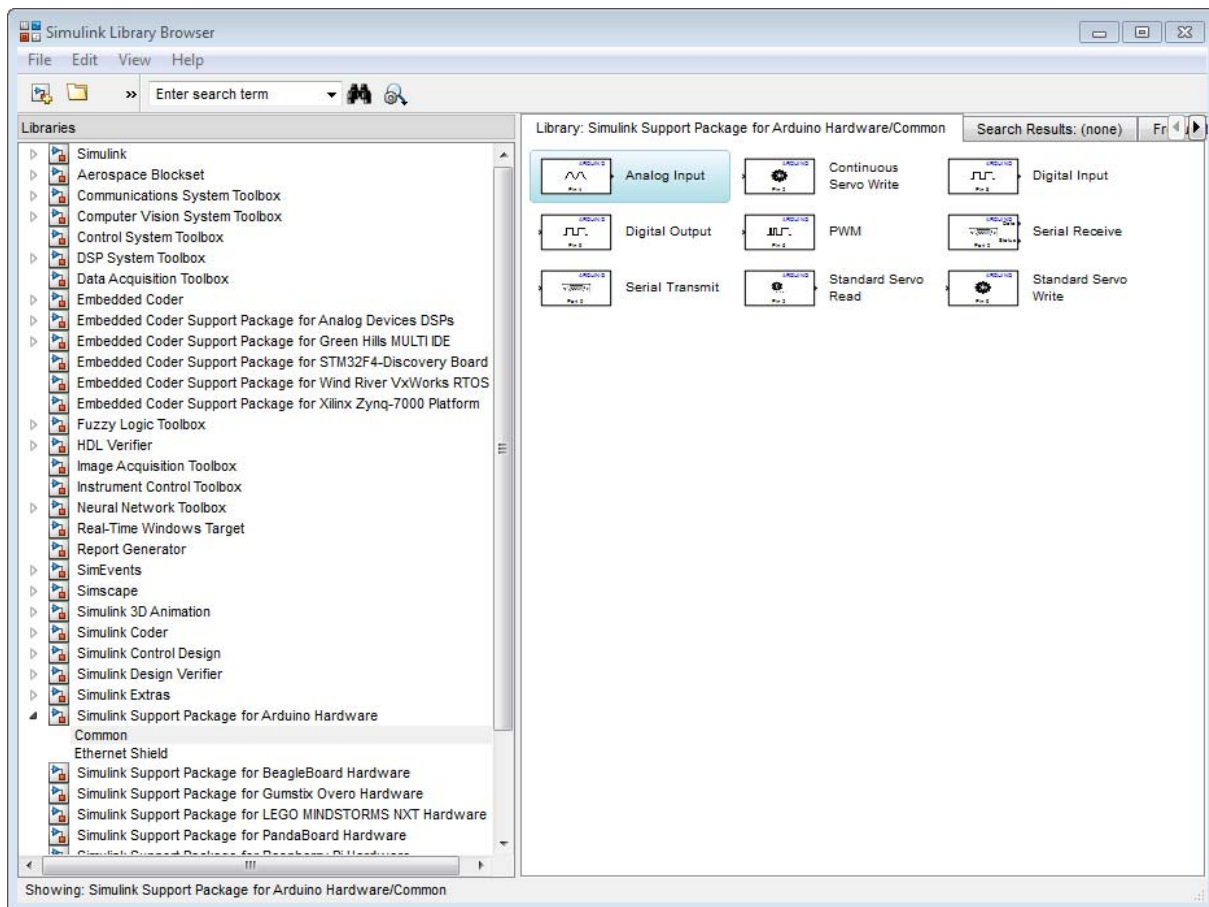
To open the block library from the Simulink Library Browser:

Enter `simulink` in the MATLAB Command Window, or click the following icon on the MATLAB toolbar.



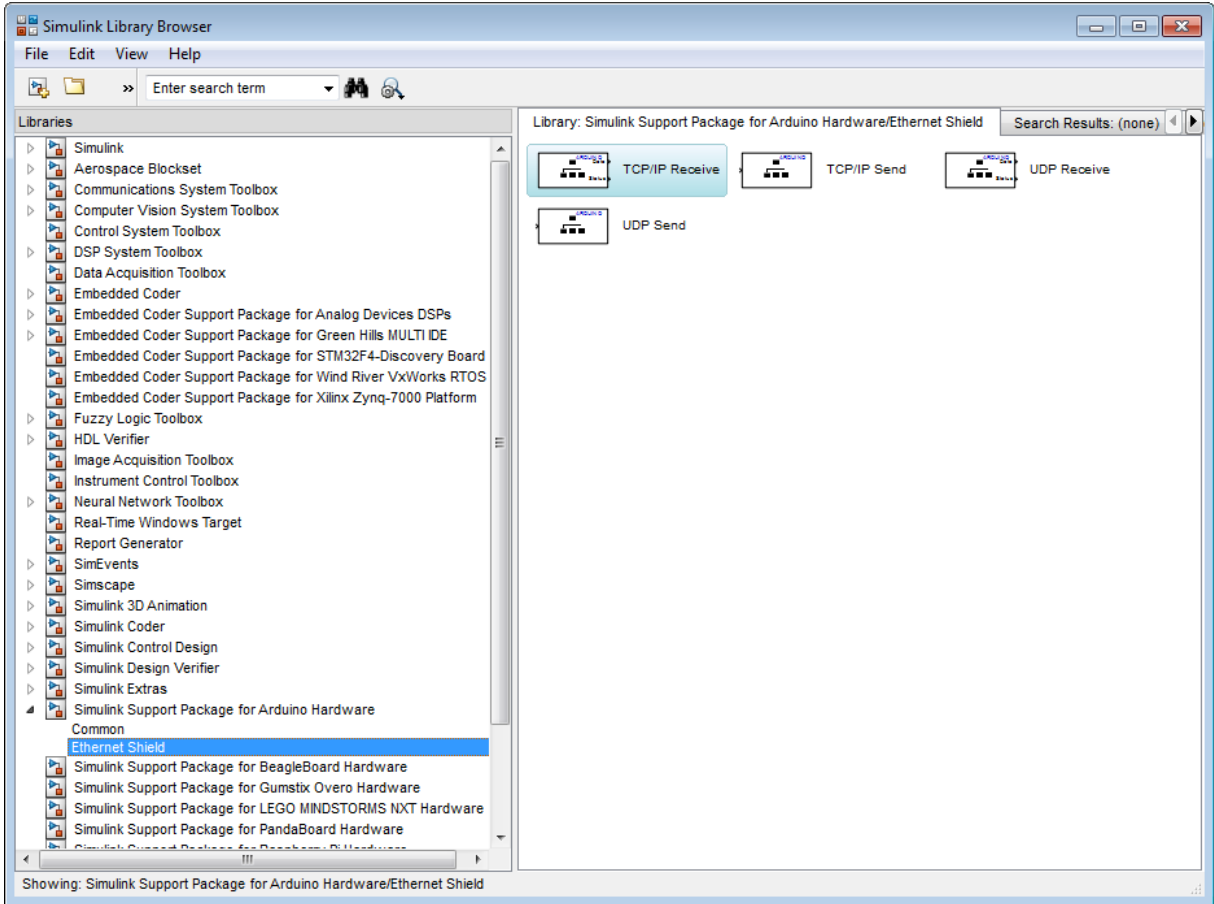
In the Simulink Library Browser, click **Simulink Support Package for Arduino Hardware >Common**.

Simulink Library Browser displays the corresponding block library.



In the Simulink Library Browser, click **Simulink Support Package for Arduino Hardware >Ethernet Shield**.

Simulink Library Browser displays the corresponding block library.

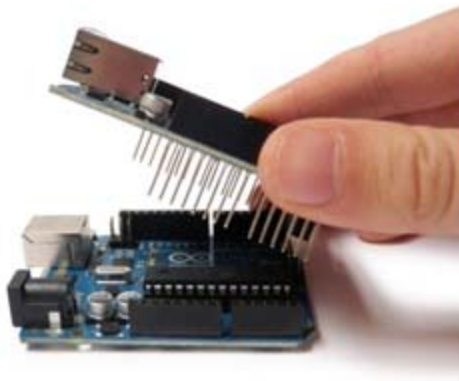


Connect Arduino Ethernet Shield to Arduino Hardware

The Arduino Ethernet shield stacks on top of the Arduino using the shield interface, which you can find on the Arduino Uno, Arduino Mega, and the other variants of the Arduino hardware.

To connect the Arduino Ethernet shield to Arduino hardware and your PC:

- 1** Insert the Ethernet shield firmly on the Arduino hardware.



An Ethernet shield stacked on the Arduino hardware is shown in the following figure.

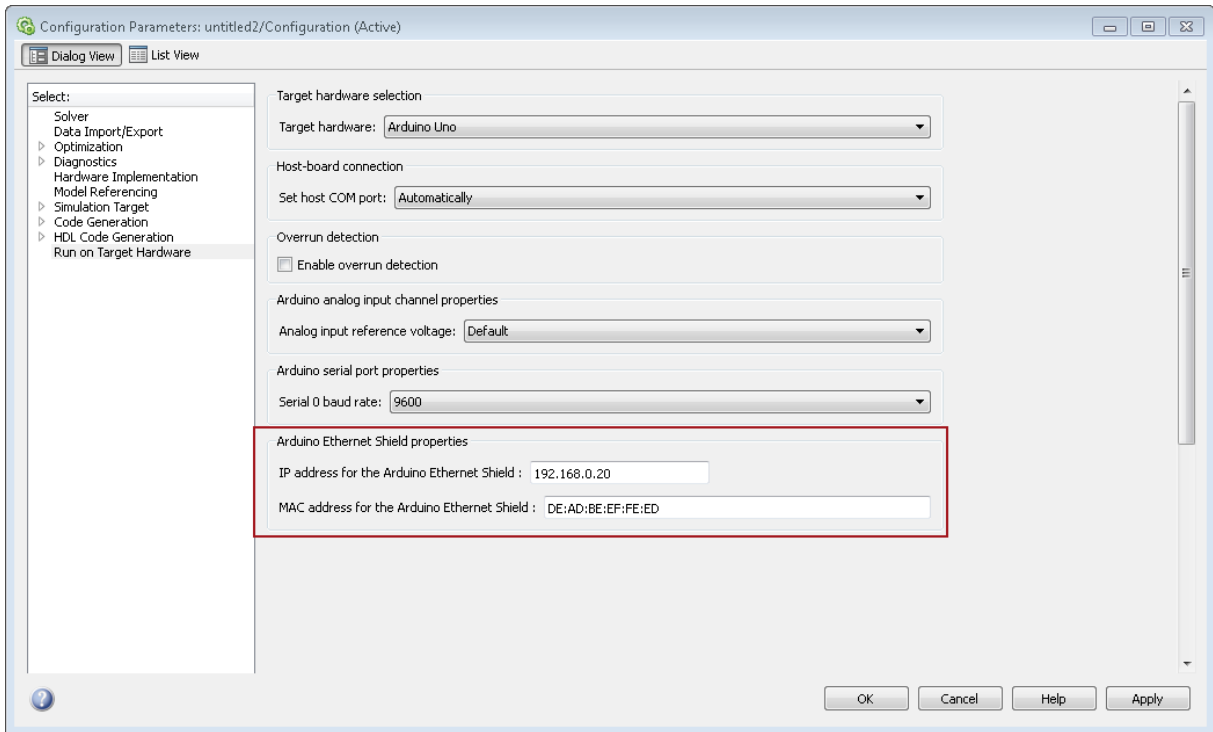


- 2** Connect the Ethernet shield to the network router, or to PC, using a RJ45 cable.



Configure Network Settings for Arduino Ethernet Shield

You can configure the network settings for Arduino Ethernet shield on Configuration Parameters dialog box. To do so, select **Arduino Ethernet Shield properties**.



To configure the network settings for the Arduino Ethernet Shield:

- 1 Open **Configuration Parameters** dialog box. Under Arduino Ethernet Shield properties, specify the **IP address for the Arduino Ethernet Shield**

When you specify the Ethernet IP address, follow these guidelines:

- IP address must be in the 4 octet format. For example, 192.168.0.20.

- When connected to a PC the IP addresses of the Arduino Ethernet Shield and the PC must be in the same range so they can communicate with each other.
 - IP address is not the same as the PC to which the Ethernet shield is connected.
 - IP address must be in the valid range of the supported IP address, if connected to a router.
 - If the Arduino hardware is connected to a router, the IP address must not be in the filtered list for that router.
- 2** Specify the MAC (Media Access Control address), the physical network address for the Arduino Ethernet shield in **MAC address for the Arduino Ethernet Shield**.

When you specify the Ethernet MAC address, follow these guidelines:

- IP address must be in the 6 octet format. For example, DE:AD:BE:EF:ED
- If the Arduino hardware is connected to a router, the IP address must not be in the filtered list for that router.

Run Model on Arduino Hardware

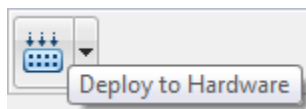
You can prepare, configure, and run a model on your Arduino hardware.

Before starting this procedure:

- Connect your Arduino hardware to the host computer using a USB cable.
- Create or open a Simulink model.

To prepare and run the model:

- 1** Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 2** In your model, select **Tools > Run on Target Hardware > Prepare to Run**. This action changes the model Configuration Parameters.
- 3** When the Run on Target Hardware pane opens, use the **Target hardware** parameter to the specific board you are using.
- 4** Click the Deploy to Hardware button.



This action automatically downloads and runs your model on the board.

Stop or Restart a Model on the Arduino Hardware

To stop a model running on Arduino hardware, you can:

- Disconnect the power from the hardware. When you reconnect the power, the hardware starts running the model again.
- Run a new or updated model on the hardware. This action automatically stops and erases the previous model running on the Arduino hardware.

To restart the model running on the Arduino hardware, press the RESET button on the board.

Tune and Monitor Model Running on Arduino Mega 2560 Hardware

In this section...

“About External Mode” on page 56-14

“Run Your Model in External Mode” on page 56-15

“Stop External Mode” on page 56-17

About External Mode

You can use External mode to tune parameters in, and monitor data from, your model while it is running on the Arduino Mega 2560 hardware. This capability is not available with Arduino Uno hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on the model results in real-time, in order to find the optimal values to achieve the desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to re-run the model each time you change parameters. External mode also lets you develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

The following list provides an overview of the parameter tuning process with External mode:

- In the model on your host computer, you enable External mode in the Configuration Parameters.
- In the model on your host computer, you configure Simulink software to run your model on the target hardware.
- You use the model on the host computer as a user interface for interacting with the model running on the target hardware:

- When you open blocks and apply new parameter values on the host computer, External mode updates the corresponding values in the model running on the target hardware.
- If your model contains blocks for viewing data, such as Scope or Display blocks, External mode sends the corresponding data from the target hardware to those blocks on the host computer.
- You determine the optimal parameter settings by adjusting parameter values on the host computer and observing data/outputs from the target hardware.

When you have finished tuning a model, you can disable External mode and run the tuned model on your hardware.

Some limitations apply while you are using External mode:

- Do not configure Serial Receive or Serial Transmit blocks in your model to use serial port 0. External mode uses serial port 0.
- Do not use the following Arduino® servo blocks: Standard Servo Read, Standard Servo Write, and Continuous Servo Write.

Run Your Model in External Mode

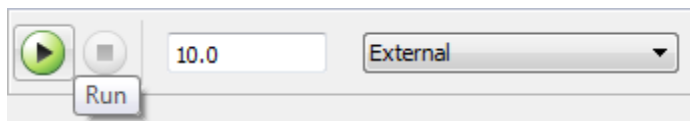
Before starting:

- Connect your Arduino Mega 2560 hardware to the host computer using a USB cable.
- Create or open a Simulink model located on a local drive or a mapped network drive that has a drive letter assigned to it.

The software generates an error message if the location of the model contains a UNC path. For example, `\\server-00\user$\MATLAB\`

To prepare and run the model:

- 1** On the model toolbar, set **Simulation mode** to External mode.



- 2 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 3 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\s1_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

- 4 While the model is running in External mode, you can change tunable parameter values in the model on your host computer and observe the corresponding changes in the model running on the hardware.

If your model contains blocks from the Simulink Sinks block library, the sink blocks in the model on your host computer display the values generated by the model running on the hardware.

If your model does not contain a sink block to which External mode can send data, the MATLAB Command Window displays a “No data has been selected for uploading” warning. You can disregard this warning, or you can add a sink block to the model and rerun your model.

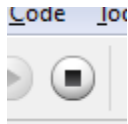
When you have finished tuning and monitoring your model, you can disable External mode.

To deploy model to your hardware without using External mode. See “Run Model on Arduino Hardware” on page 56-12.

Note External mode increases the processing burden of the model running on the hardware. If the software reports an overrun, you can apply the solutions described in “Detect and Fix Task Overruns on Arduino Hardware” on page 56-21.

Stop External Mode

To stop the model running in External mode, click the black square Stop button located on the model toolbar, as shown here.



This action stops the process for the model running on the target hardware, and stops the model simulation running on your host computer.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

Use Serial Communications with Arduino Hardware

In this section...
“Hardware” on page 56-18
“Transmit Serial Data” on page 56-19
“Receive Serial Data” on page 56-20

Arduino hardware has serial ports, also known as UARTs, that can communicate with other devices that have serial interfaces.

Hardware

The Arduino Uno board has one serial port, serial port 0, connected to:

- The digital pins marked TX 1 (transmit) and RX 0 (receive).
- The USB port, through a serial-to-USB converter.

The Arduino Nano board has

- The digital pins marked TX 1 (transmit) and RX 0 (receive).

The Arduino Mega 2560 board has four serial ports:

- Serial port 0 is connected to Communication pins marked TX0 1 (transmit) and RX0 0 (receive). Serial port 0 is also connected to the USB port through a converter.
- Serial port 1 is connected to Communication pins marked TX1 18 (transmit) and RX1 19 (receive).
- Serial port 2 is connected to Communication pins marked TX2 16 (transmit) and RX2 17 (receive).
- Serial port 3 is connected to Communication pins marked TX3 14 (transmit) and RX3 15 (receive).

You can use serial port 0 to communicate with other devices that have serial ports, or to communicate with a computer over the USB port.

Each serial port supports one Serial Transmit and one Serial Receive block, one block per pin.

If you intend to use External mode with Arduino Mega 2560 hardware, use serial ports 1 through 3 for serial communications. Serial port 0 is not available for serial communications because it is connected to the USB port, which External mode uses to communicate with the host computer. This restriction does not apply to Arduino Uno hardware, because External mode is not supported. For more information, see “Tune and Monitor Model Running on Arduino Mega 2560 Hardware” on page 56-14.

Serial communications are not supported in models that also use the Arduino Standard Servo Read, Standard Servo Write, and Continuous Servo Write blocks.

Warning Only connect serial port pins to devices that use 5 Volt TTL logic. Do not connect these pins to an RS-232 serial interface, such as the DE-9M connector on a computer, without limiting the voltage. The RS-232 standard allows higher voltages that can damage your hardware. For details, read the documentation for your Arduino hardware.

Transmit Serial Data

To transmit data through a serial port or USB port on the Arduino hardware:

- 1 Add the Arduino Serial Transmit block to your model.
- 2 Connect a data source to the block input on the Serial Transmit block.

If the data type is not uint8, use a Data Type Conversion block to convert it to uint8.

- 3 In the Arduino Serial Transmit block, select a **Port number**.
- 4 Click the **Tools** menu in the model, and select **Run on Target Hardware > Options**.

In the Configuration Parameters dialog that opens, on the Run on Target Hardware pane, set the baud rate for the serial port you selected in the Arduino Serial Transmit block.

- 5** Connect the appropriate digital transmit pin to the hardware that receives the data.
- 6** Run the model, as described in “Run Model on Arduino Hardware” on page 56-12.
- 7** If your model uses the Arduino USB port (Serial port 0) to transmit data to a device that is not your host computer, reconnect the USB cable to that device and press the RESET button.

Receive Serial Data

To receive data through a serial port or USB port on the Arduino hardware:

- 1** Add the Arduino Serial Receive block to your model.
- 2** On the Arduino Serial Receive block, connect the **Data** block output to a block that uses the data.
- 3** Open the Arduino Serial Receive block and specify the **Port number**.
- 4** Click the **Tools** menu in the model, and select **Run on Target Hardware > Options**.

In the Configuration Parameters dialog that opens, on the Run on Target Hardware pane, set the baud rate for the serial port you selected in the Arduino Serial Receive block.

- 5** Connect the digital receive pin to the hardware that transmits the data.
- 6** Run the model, as described in “Run Model on Arduino Hardware” on page 56-12.
- 7** If your model uses the Arduino USB port (Serial port 0) to receive data from a device that is not your host computer, reconnect the USB cable to that device and press the RESET button.

Detect and Fix Task Overruns on Arduino Hardware

You can configure a Simulink model running on the target hardware to detect and notify you when a task overrun occurs. A task overrun occurs if the target hardware is still performing one instance of a task when the next instance of that task is scheduled to begin. You can fix overruns by decreasing the frequency with which tasks are scheduled to run, and/or by reducing the number of tasks defined by your model.

To enable overrun detection:

- 1 Click the **Tools** menu in the model, and select **Run on Target Hardware** and **Options**.
- 2 In the Run on Target Hardware pane that opens, select the **Enable overrun detection** check box.
- 3 Use the **Digital output to set on overrun** parameter to specify the pin number of a digital output.
- 4 Click **OK**.

To create a visual overrun indicator for your board, connect an appropriate resistor in series with an LED between the GND and the hardware pin specified by the **Digital output to set on overrun** parameter. Orient the LED so the longer leg (positive) is connected to the digital output pin.

When a task overrun occurs:

- The state of the digital output pin specified by the **Digital output to set on overrun** parameter changes from low (0 Volts) to high (5 Volts).
- The model continues running, but the effective sample time will be longer than specified.

To fix an overrun condition, reduce the processing burden of the model by applying one or more of the following solutions:

- Increase the sample times for the model. For example, increase the values of the **Sample time** parameters in all of your data source blocks.

- Simplify the model.

If you are using External mode, and the preceding solutions do not fix the task overrun condition, consider disabling External mode. External mode adds a lightweight server to the model running on the target hardware. This server increases the processing burden upon the target hardware, which can contribute to a task overrun condition.

Troubleshoot Running Models on Arduino Hardware

In this section...

““Could not automatically set host COM port”” on page 56-23

““Invalid port name. COM port must start with”” on page 56-24

“Could not automatically set host COM port”

If you try to run a model on your Arduino hardware and Simulink generates an error message similar to this one: “The call to `realtime_make_rtw_hook`, during the entry hook generated the following error: Could not automatically set host COM port for your Arduino hardware. This may be due to a disconnected or unrecognized board. If the board is not connected to your host computer, connect it and let the operating system install the board driver.”

Resolve Connection Issues

To resolve connection issues:

- 1 Verify that your Arduino hardware is powered on and connected to your host computer.
- 2 Try running the model again on your Arduino hardware.

Resolve Driver Issues

If you get the error message while your board is powered on and connected to your host computer, resolve any issues with Arduino drivers in Windows:

- 1 In the Windows Start menu, select **Devices and Printers**.
- 2 If you find an **Unknown Device** under **Other Devices** or **COM Ports**, double click the **Unknown Device**.
- 3 In the **Unknown Device Properties** dialog box that opens, click the **Hardware** tab, and click **Properties**.
- 4 In the **Unknown device Properties** dialog box that opens, click **Update Driver**.

- 5 In the **Update Driver Software - Unknown Device** dialog box that opens, click **Browse my computer for driver software**.
- 6 Select the **Include subfolders** checkbox and click **Browse**.
- 7 Navigate to the **Installation folder** that Support Package Installer used when you installed support for your Arduino hardware, and then click **Next**. By default, this folder location is `C:\MATLAB\Targets\releasenumbr\arduino-version`. For example:
`C:\MATLAB\Targets\R2012a\arduino-1.0`.
- 8 If prompted by Windows Security, choose **Install this driver software anyway**, and let Windows complete the process of installing the driver.
- 9 Try running the model again on your Arduino hardware.

If you get the error message after resolving issues with Arduino drivers, set the host COM port and baud rate manually. The drivers for some Arduino board revisions do not identify the board as an Arduino device in Windows. In that case, set the **COM port number** and **Serial 0 baud rate** manually, as described in “Configure Host COM Port Manually” on page 56-26

“Invalid port name. COM port must start with”

Some Apple Mac OS X platforms require you to install a driver before you can connect to Arduino hardware.

Suppose you cannot download and run a model on the Arduino hardware because the COM port is not set correctly, and trying to change the COM port number produces an error message such as: Invalid port name. COM port must start with '/dev/cu.usb' like /dev/cu.usbmodemfa131.

To resolve this issue:

- 1 Visit <http://www.ftdichip.com/Drivers/VCP.htm>.
- 2 Download and install the driver that matches your Apple Mac OS X hardware.
- 3 Update the COM port number.

4 Try running the model on the target hardware again.

Configure Host COM Port Manually

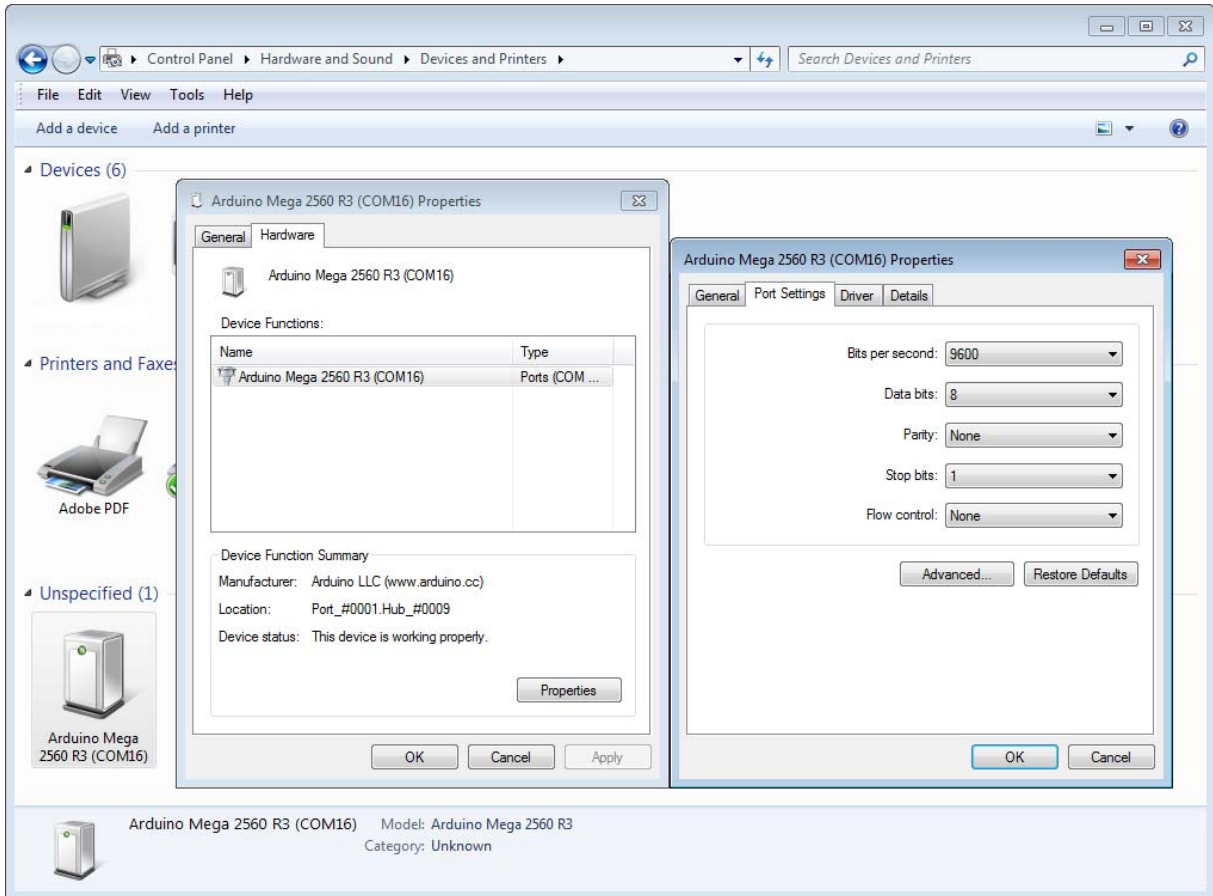
Simulink software automatically detects the COM port settings of the USB connection between your host computer and the Arduino hardware. Optionally, you can also configure these settings manually.

Configure COM Port on Windows

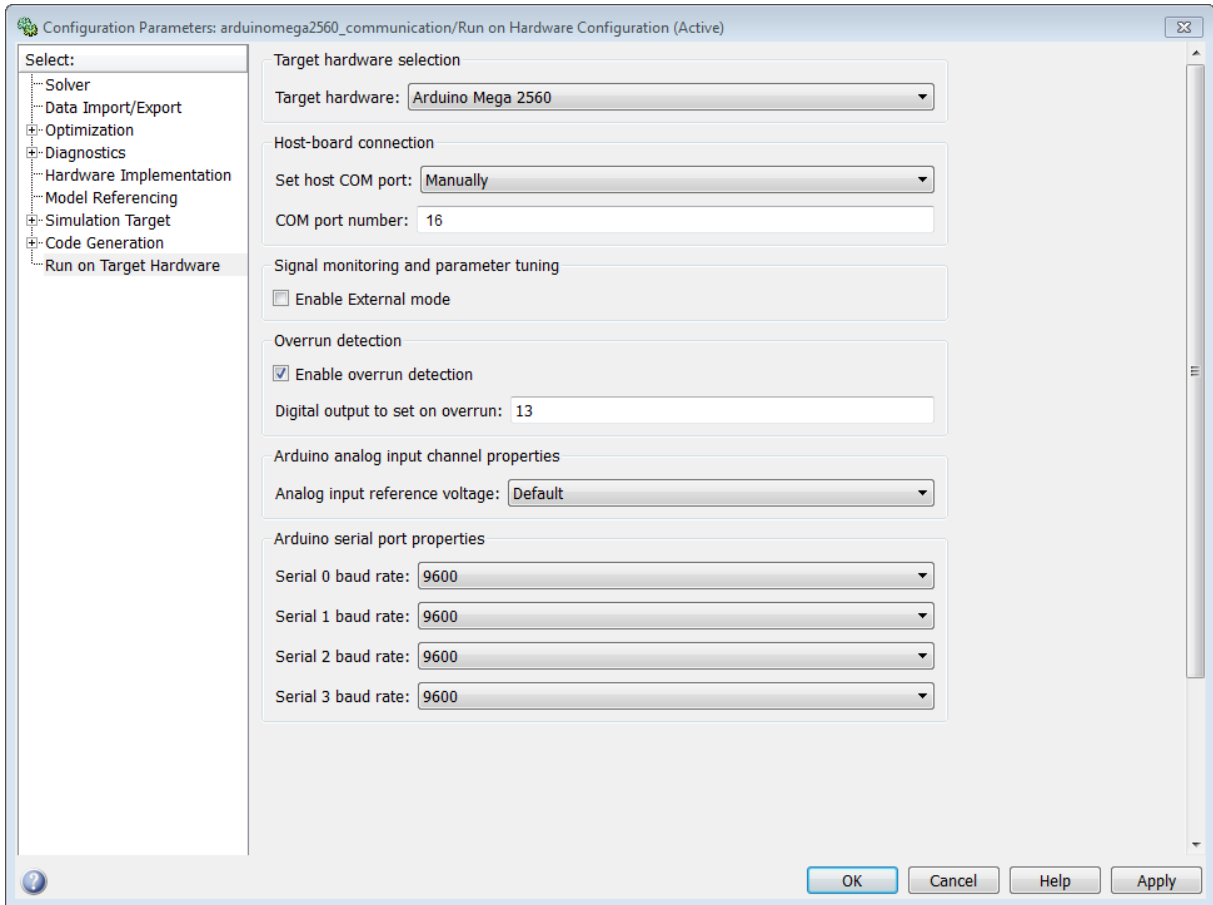
To configure the COM port settings manually on Windows:

- 1** Click the **Tools** menu in the model, and select **Run on Target Hardware > Options**.
- 2** In the Run on Target Hardware pane, change the **Set host COM port** parameter to **Manually** and leave the Configuration Parameters dialog open.
- 3** Open **Devices and Printers** in Windows.
- 4** Double-click **Arduino Uno** or **Arduino Mega 2560** or **Arduino Nano** device.
- 5** In the device properties dialog, click the **Hardware** tab, and then click the **Properties** button.
- 6** Click the **Port Settings** tab.

For example, the following image shows an Arduino device in Devices and Printers, the Hardware tab, and the Port Settings tab.



7 In Configuration Parameters, update **COM port number** and **Serial 0 baud rate** to match the Arduino device in Windows.



8 Click **OK** and save the model.

Configure COM Port on Mac OS X

To configure the COM port settings manually on Mac OS X:

1 Click the **Tools** menu in the model, and select **Run on Target Hardware > Options**.

2 In the Run on Target Hardware pane, change the **Set host COM port** parameter to **Manually** and leave the Configuration Parameters dialog open.

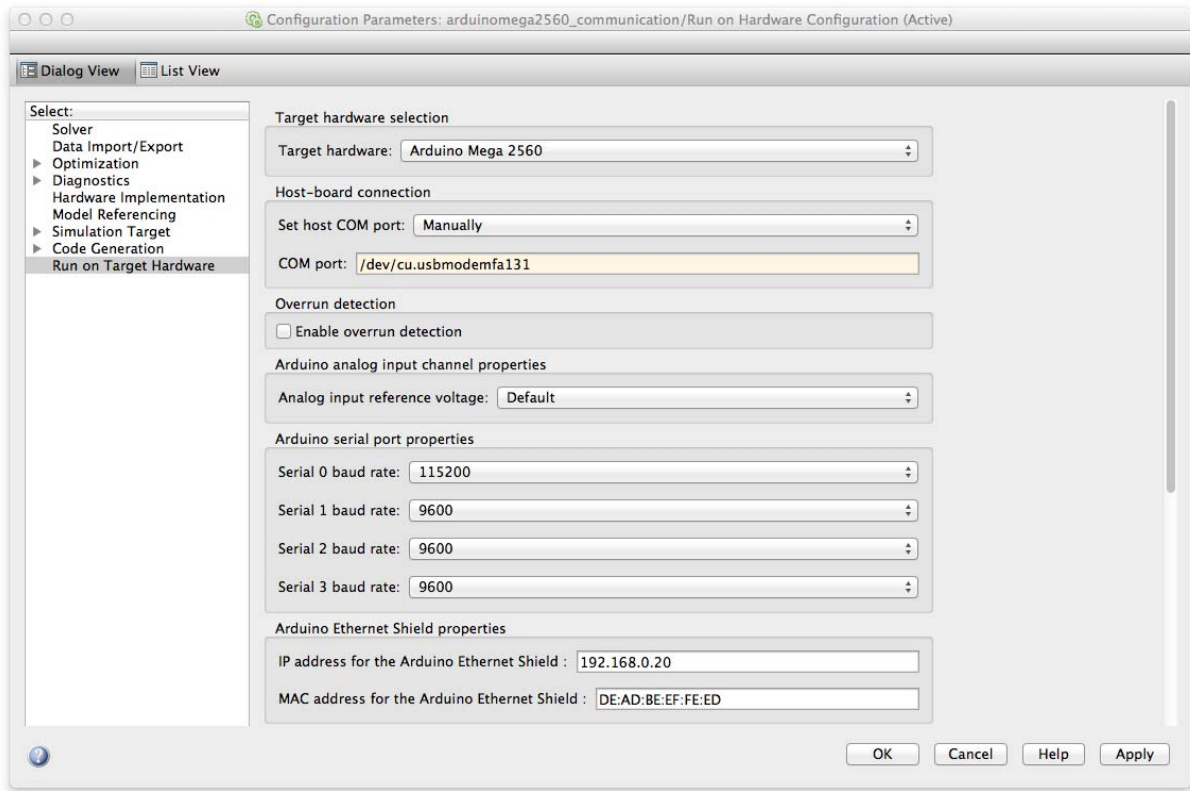
3 On Mac OS X, open a terminal window and enter the following command:

```
ls /dev/cu.usb*
```

This command lists the path and file name of the USB device connected to the Arduino hardware. For example:

```
/dev/cu.usbmodemfa131
```

4 In Configuration Parameters, update **COM port** with the path and file name of the USB device. For example, enter `/dev/cu.usbmodemfa131`.



5 Click **OK** and save the model.

BeagleBoard Hardware

- “Install Support for BeagleBoard Hardware” on page 57-2
- “Replace Firmware on BeagleBoard Hardware” on page 57-3
- “Choose the Type of Serial Cable” on page 57-19
- “Connect to Serial Port on BeagleBoard Hardware” on page 57-20
- “Configure Network Connection with BeagleBoard Hardware” on page 57-25
- “Get IP Address of BeagleBoard Hardware” on page 57-29
- “Open Block Library for BeagleBoard Hardware” on page 57-31
- “Run Model on BeagleBoard Hardware” on page 57-34
- “Tune and Monitor Model Running on BeagleBoard Hardware” on page 57-36
- “Detect and Fix Task Overruns on BeagleBoard Hardware” on page 57-40

Install Support for BeagleBoard Hardware

This example shows how to add support for BeagleBoard hardware to the Simulink product. After you complete this process, and replace the firmware on the BeagleBoard hardware, you can run a Simulink model on your BeagleBoard hardware.

To install support for BeagleBoard hardware:

- 1** In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2** Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

This process downloads and installs the following items on your host computer:

- Third-party software development tools
- A Simulink block library called **Simulink Support Package for BeagleBoard Hardware**.
- Examples

For convenience, this document occasionally refers to BeagleBoard hardware as a “board” or as “target hardware”.

To check for updates, repeat this process when a new version of MATLAB software is released. You can also check for updates between releases.

Replace Firmware on BeagleBoard Hardware

This example shows how to replace the firmware on the BeagleBoard hardware with Ubuntu® Linux firmware that can run Simulink models.

Before replacing the firmware, install the support package on your host computer.

After replacing the firmware, you can run a Simulink model on the BeagleBoard hardware.

The following steps provide an overview of the firmware replacement process:

- 1** Support Package Installer locates a firmware image on your host computer or downloads new one.
- 2** Support Package Installer uses the host computer to write the firmware image to a microSD or SD memory card.
- 3** You transfer the microSD or SD memory card to the BeagleBoard hardware.
- 4** Support Package Installer applies the IP settings you choose to the firmware on the BeagleBoard hardware.

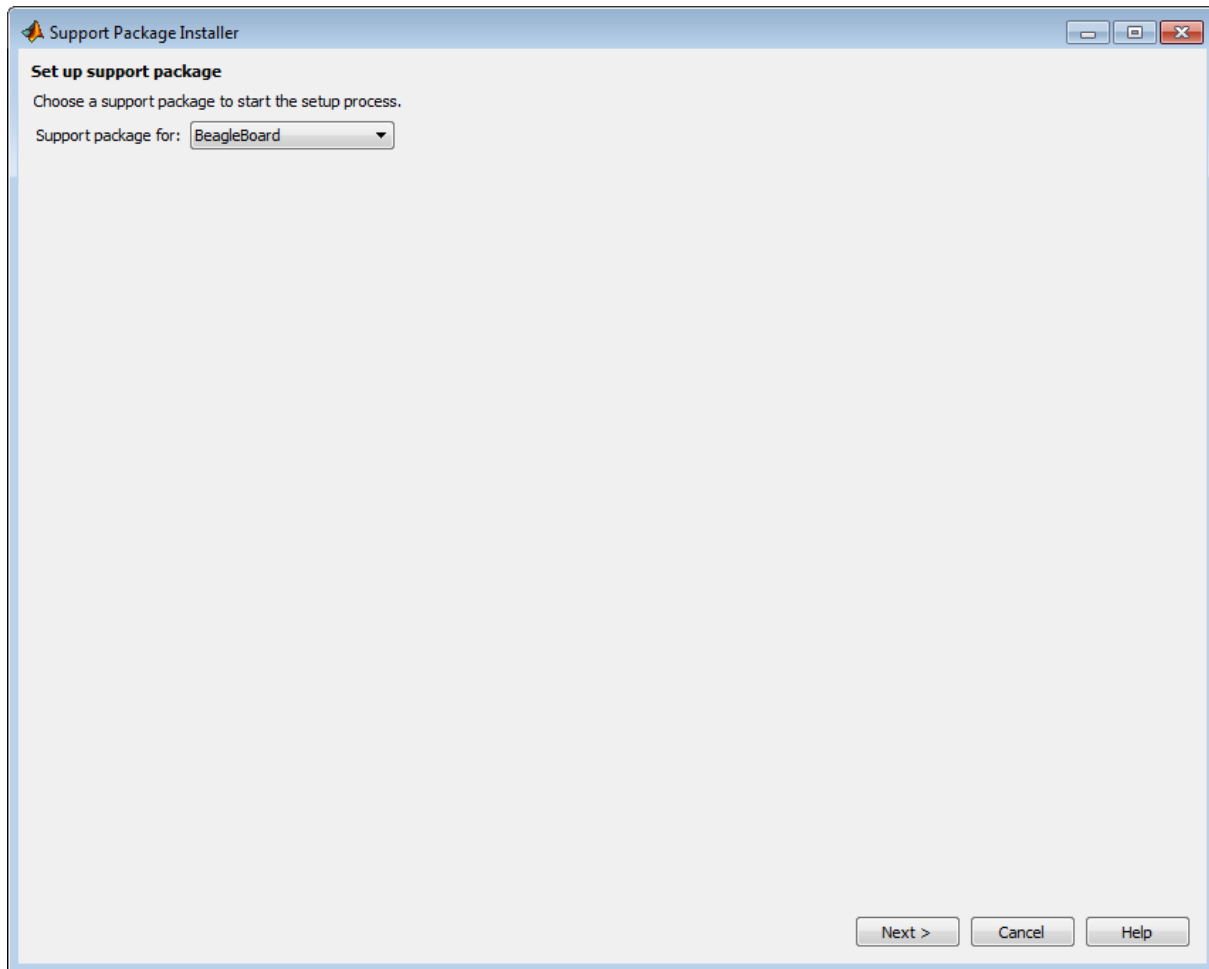
Support Package Installer does not use the BeagleBoard hardware to write the firmware image to the memory card.

To replace the firmware on your BeagleBoard hardware:

Open the **Update firmware** screen in Support Package Installer using one of the following methods:

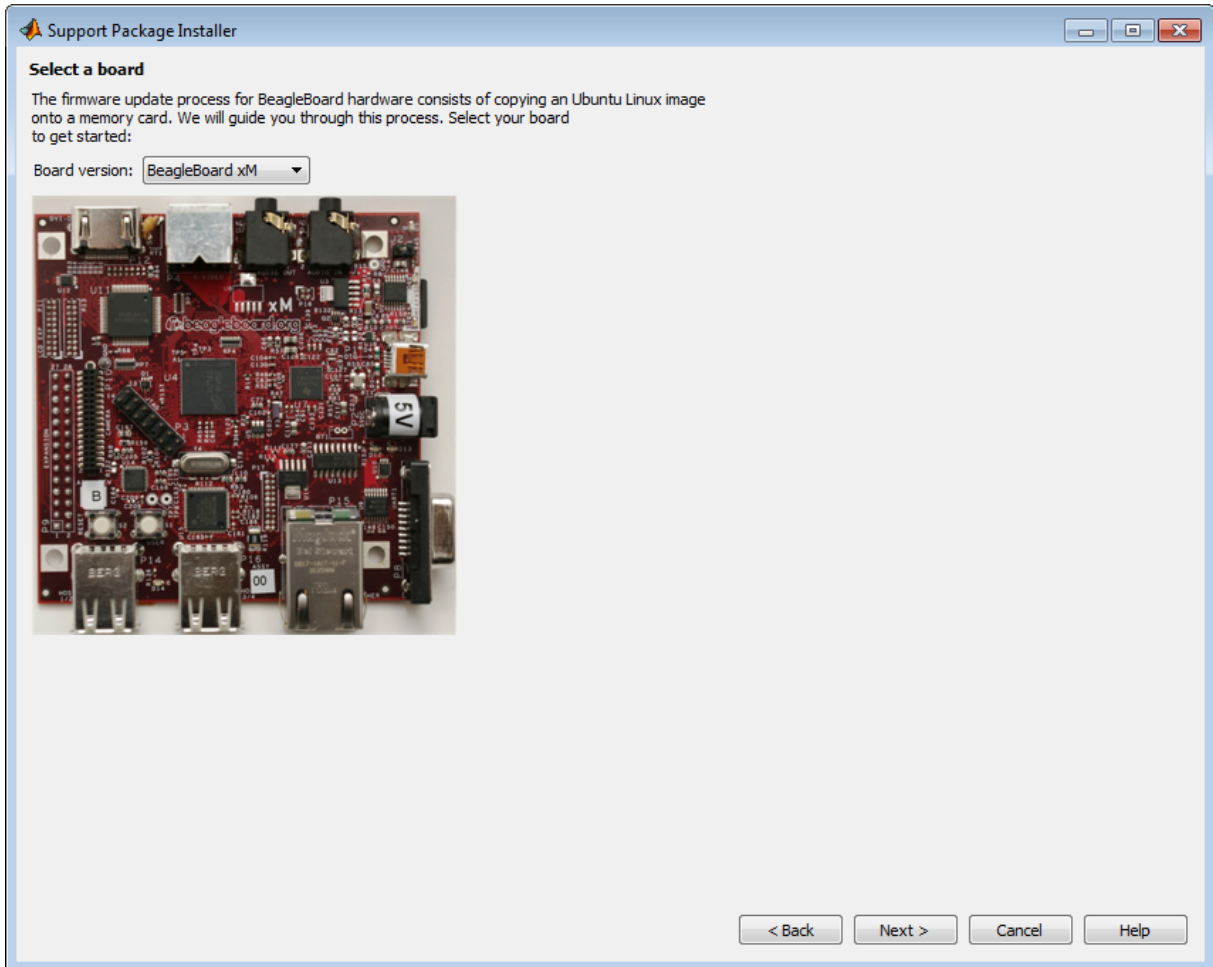
- Click **Continue** in the **Install/update complete** screen of Support Package Installer.
- In a model, select **Tools > Run on Target Hardware > Update firmware**.
- In a MATLAB Command Window, enter `targetupdater`.

Update firmware screen: Choose the BeagleBoard option and click **Next**.

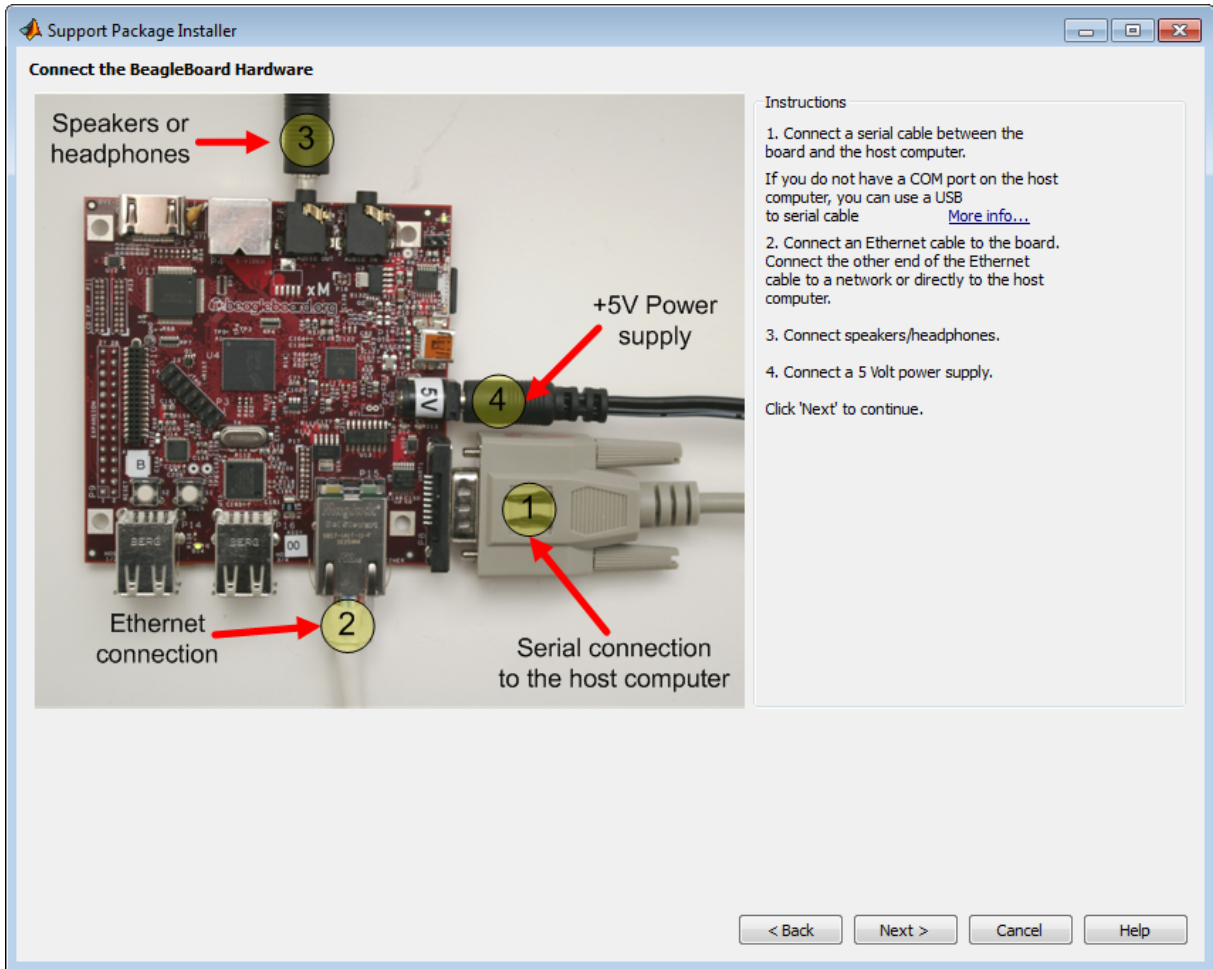


Choose your board version and click **Next**. This action enables Support Package Installer to display instructions that are specific to your hardware.

Note The images in this topic only apply to the BeagleBoard xM hardware. If you are using the BeagleBoard Bx or Cx hardware, refer to the images in Support Package Installer.



Make the connections shown and click **Next**.



Choose to get the firmware image from the Internet or from a folder.

Note The file size of the firmware image is approximately 1 GB. Depending on your connection, downloading the firmware can take from 2 to 60 minutes, or more.

Internet: This option is selected by default. When you click **Download**, Support Package Installer checks the **Download folder** for a valid firmware image:

- If a firmware image is not present, Support Package Installer downloads a firmware image from the Internet, and saves it to the download folder.
- If a firmware image is present, Support Package Installer uses the firmware image already present in the download folder, and does not download a new firmware image from the Internet.

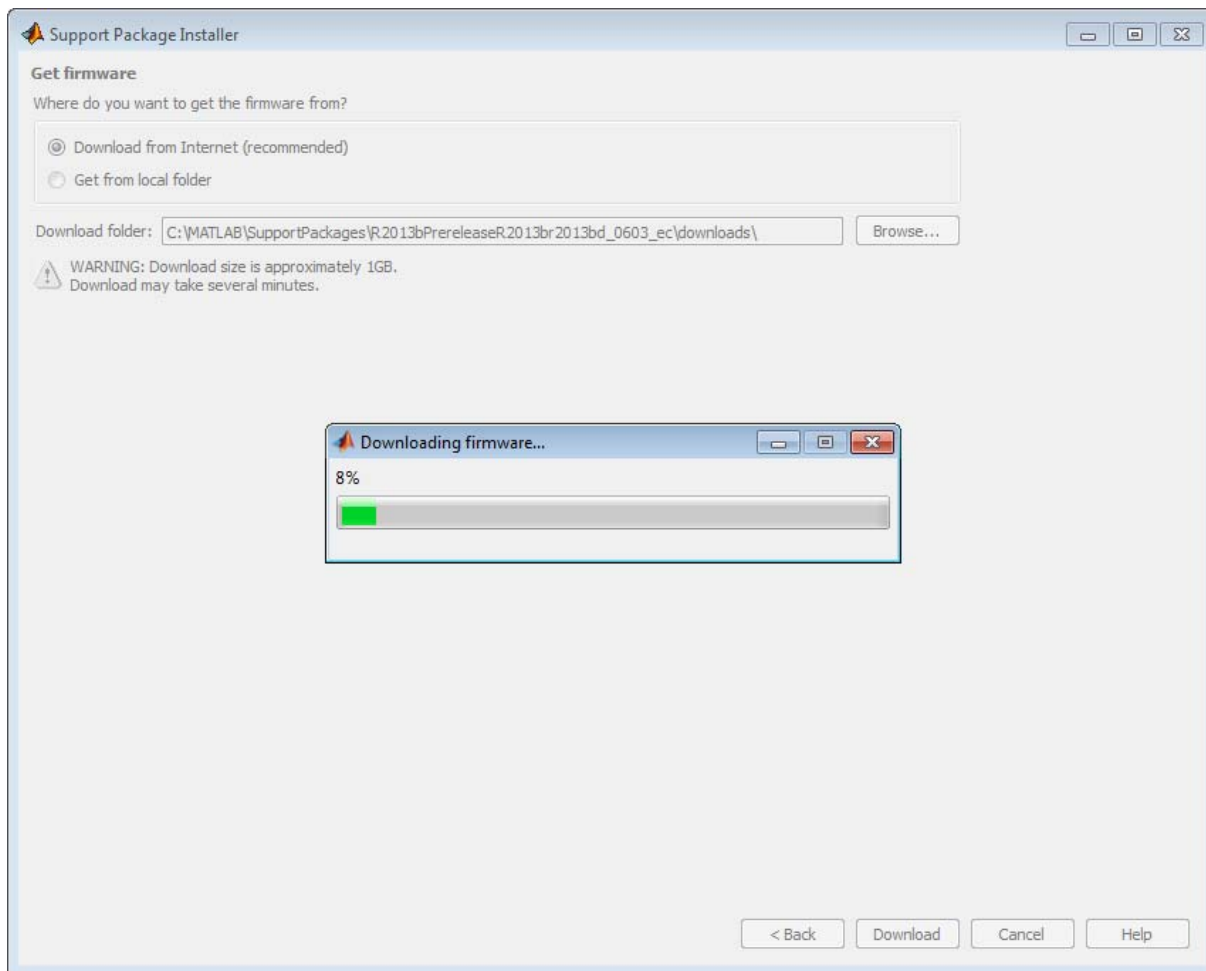
Folder: When you click **Download**, Support Package Installer checks the **Download folder** for a valid firmware image:

- If a firmware image is not present, Support Package Installer displays an error message that the image file is missing. To solve this issue, copy the firmware image from another location to the download folder, or choose the **Internet** option instead.
- If a firmware image is present, Support Package Installer continues the firmware installation process.

You must have write privileges for the download folder. If you use the default download folder, having write privileges is typically not an issue. If you change to a new download folder for which you do not have write permissions, such as a shared folder on a network, Support Package Installer generates an error message: “Error: Download the firmware. The download folder is not writable. Choose a folder for which you have write permissions”.

To solve this issue, copy the firmware image to a folder for which you have write privileges. For example, copy the firmware image from the shared folder on the network to the C:\Users\username\AppData\Local\Temp folder. Then, update the **Download location** to the same folder, and click **Download** again.

To locate the firmware image in a folder, search for a filename that begins with beagleboard_ubuntu. For example: beagleboard_ubuntu_11_04_r4_12_08_2011.img.7z

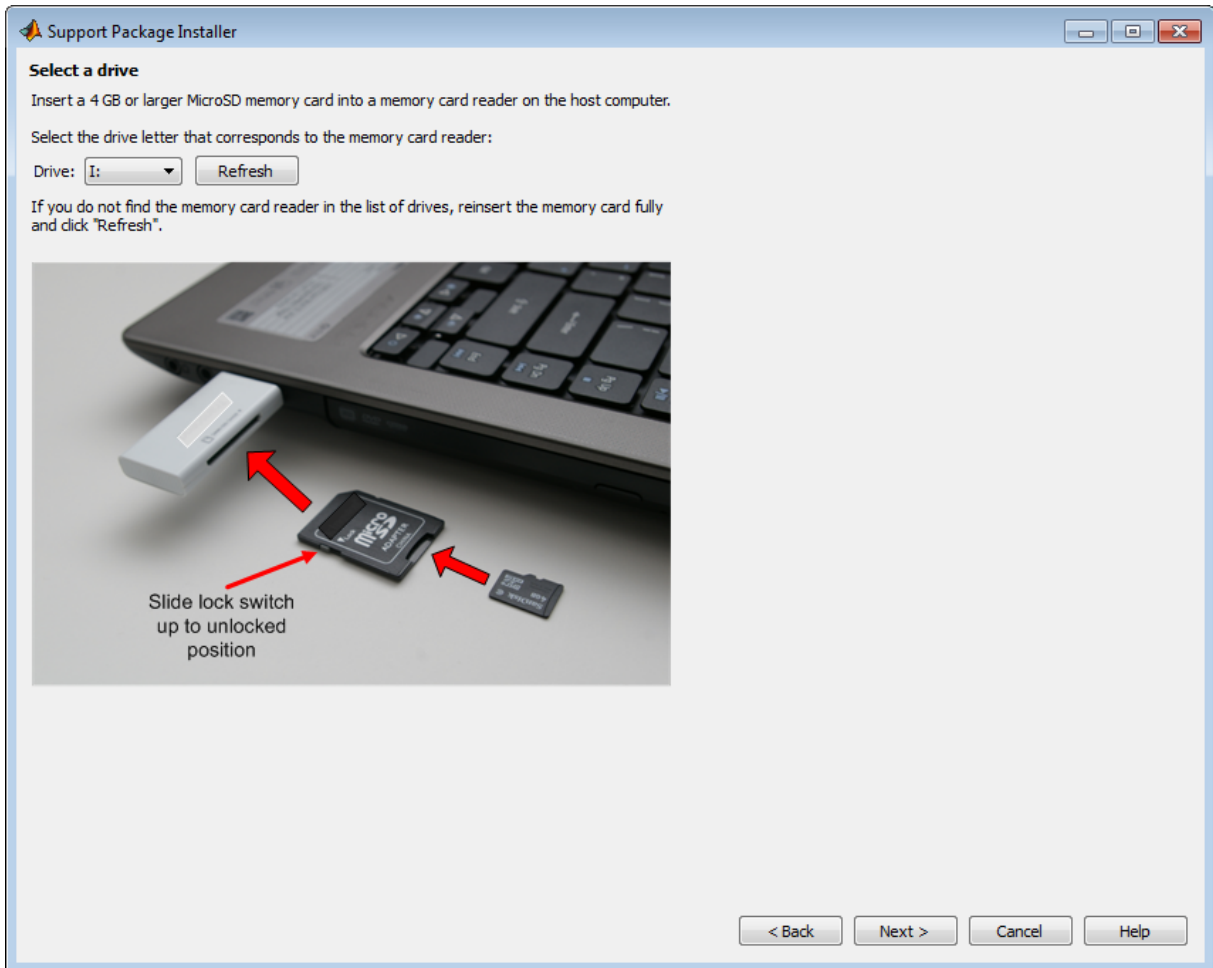


Insert the microSD or SD memory card into a media card reader connected to your host computer. Windows assigns a drive letter to the memory card.

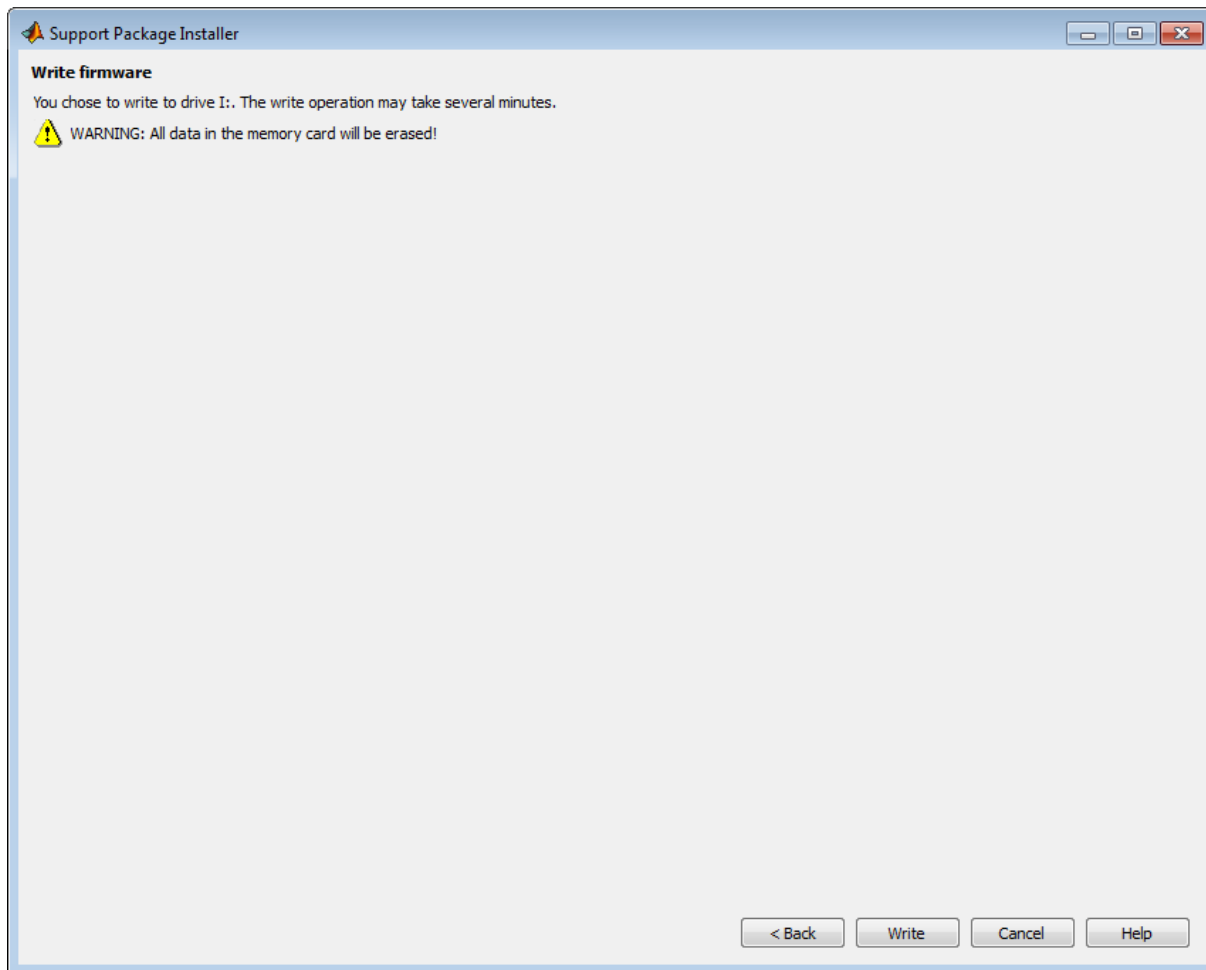
Support Package Installer does not automatically detect the drive letter of the memory card. It displays a drive letter for each device with removable storage.

- If only one drive letter is available, click **Next**.

- If no drive letters are available, check that the memory card is fully inserted, and click **Refresh**.
- If multiple drive letters are available, open the Windows Start menu, choose **Computer**, and look for the memory card under **Devices with Removable Storage**.

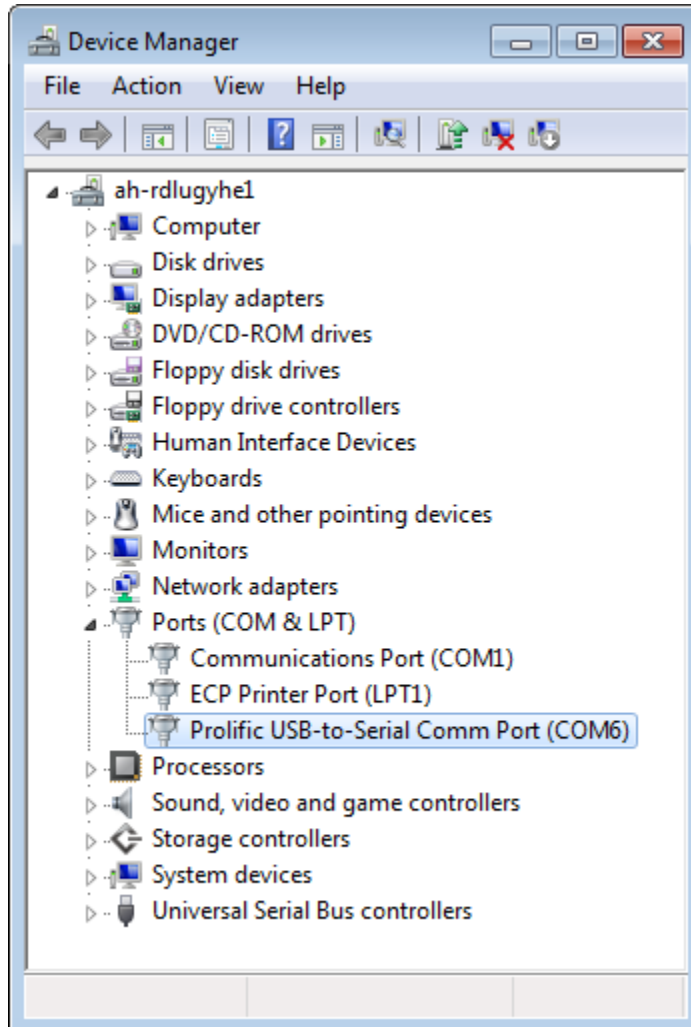


Click **Write**. Support Package Installer overwrites all previous data on the memory card with the firmware. This process takes several minutes to complete.



Get the COM port of the serial connection in Windows. From the Windows Start menu, choose search programs and files for “Device Manager”. Open Device Manager, expand **Ports (COM & LPT)**, and identify the COM port of the serial connection to the BeagleBoard hardware.

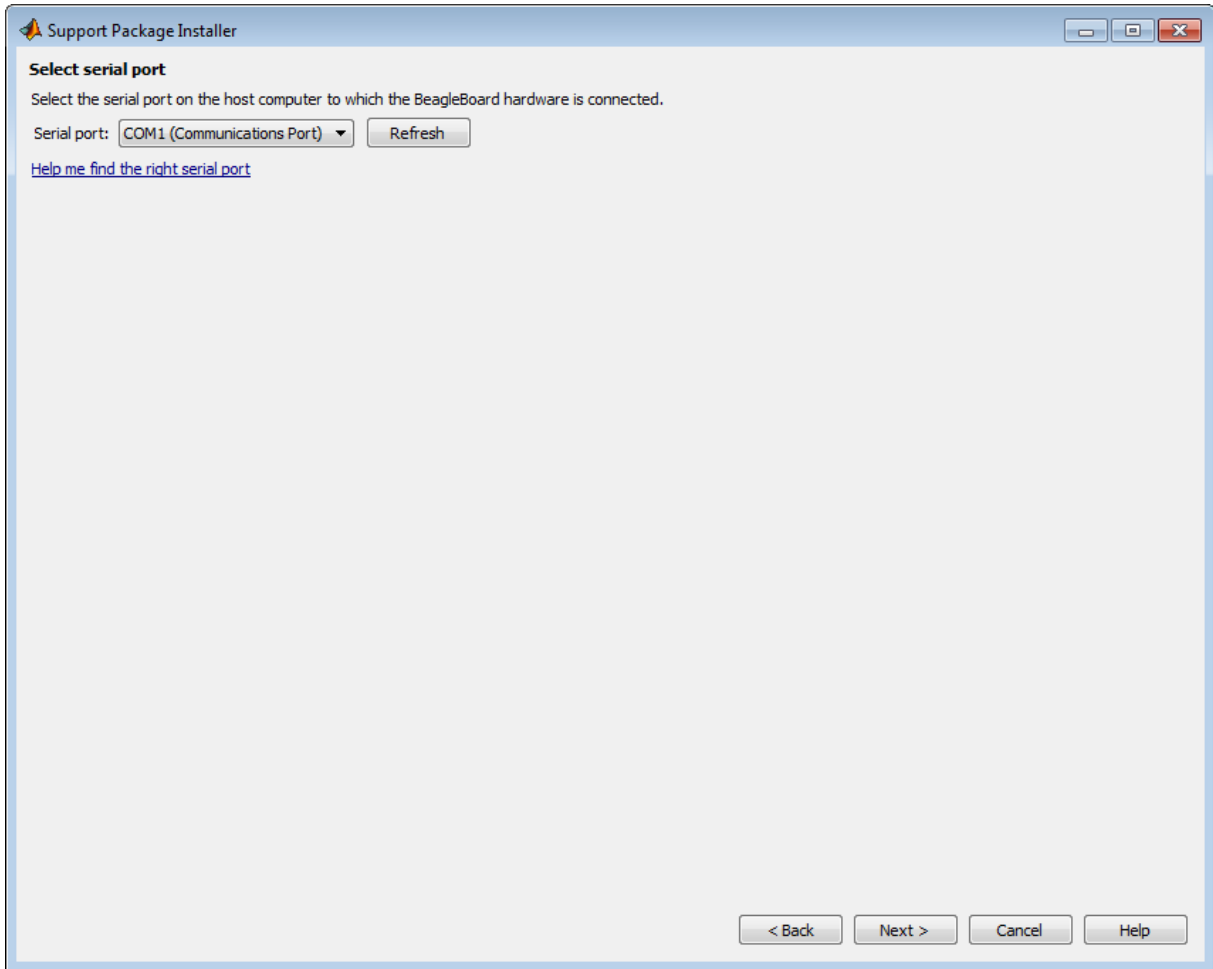
For example, the following image shows a DB9 serial port called “Communications Port” using COM1, and a USB-to-serial adapter called “Prolific USB-to-Serial Comm Port” using COM6.



Note Some USB-to-serial adapters do not appear in the list of serial connections immediately after you install the software drivers. To solve this issue, disconnect/reconnect the adapter, or reboot your host computer.

Support Package Installer does not automatically choose the COM port of the serial connection. After you get the COM port of the serial connection from Windows Device Manager, return to Support Package Installer, set **Serial port** to the COM port, and click **Next**.

The following illustration shows **COM 1 (Communications Port)**, which is a likely setting if you are connecting the serial port on your host computer to the serial port on the board.

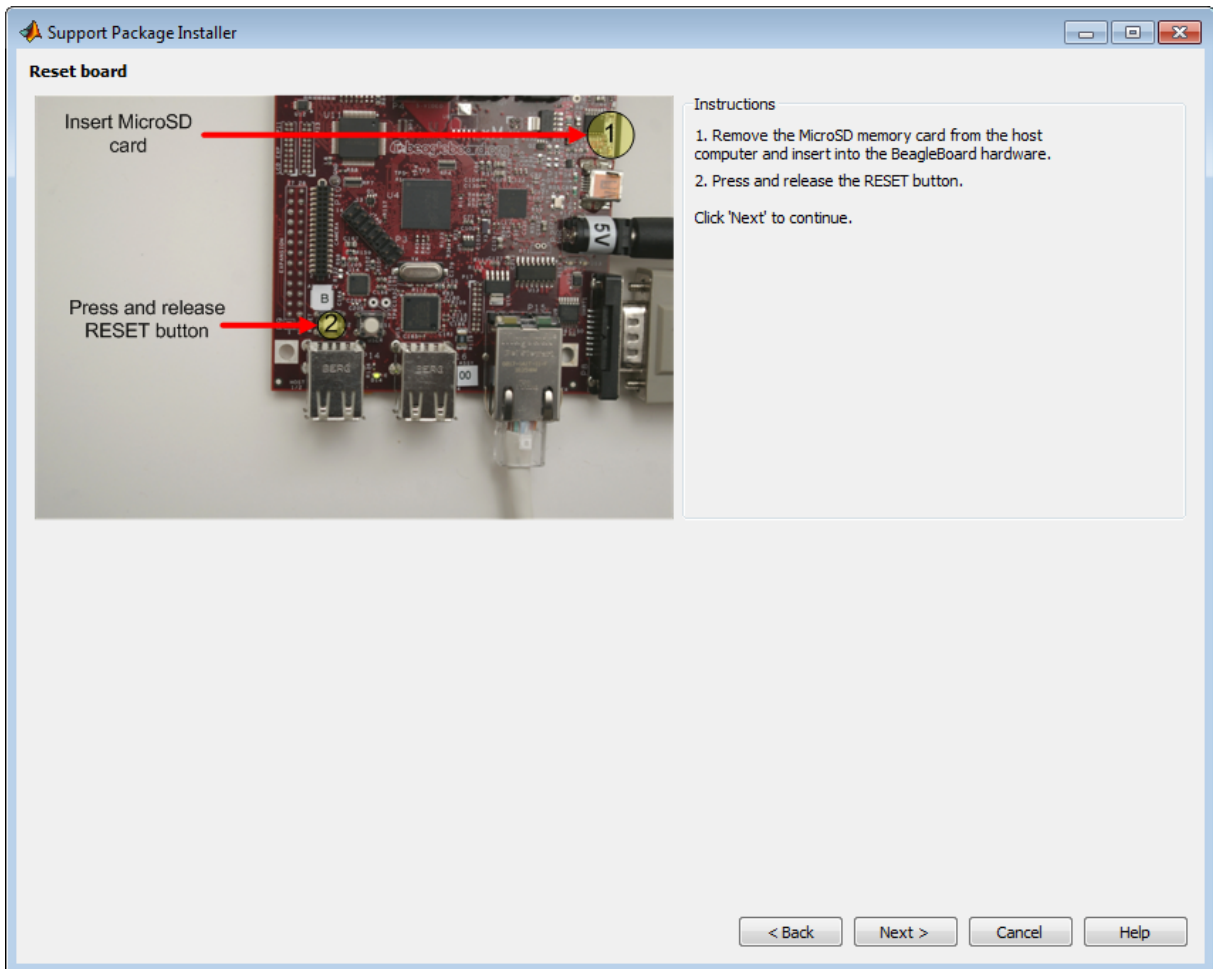


Insert the memory card into the BeagleBoard hardware, and follow the instructions for resetting your board:

- For the BeagleBoard xM hardware, press and release the RESET button.
- For the BeagleBoard Bx/Cx hardware, hold down the USER button while you press and release the RESET button.

Click **Next**. When Support Package Installer detects a reset, it displays progress booting the board.

If Support Package Installer does not display any progress, click **Back**. On the **Select serial port** screen. Verify that you selected the correct COM port, and close any other applications, such as PuTTY, that might using the serial connection.



If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet, select **Automatically get IP address**, give the board a unique name, and click **Configure**. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.

If your board is directly connected to an Ethernet port on your computer, or connected to an isolated network without DHCP services, select **Manually enter IP address**, give the board a unique name, enter static IP settings for the board, and click **Configure**.

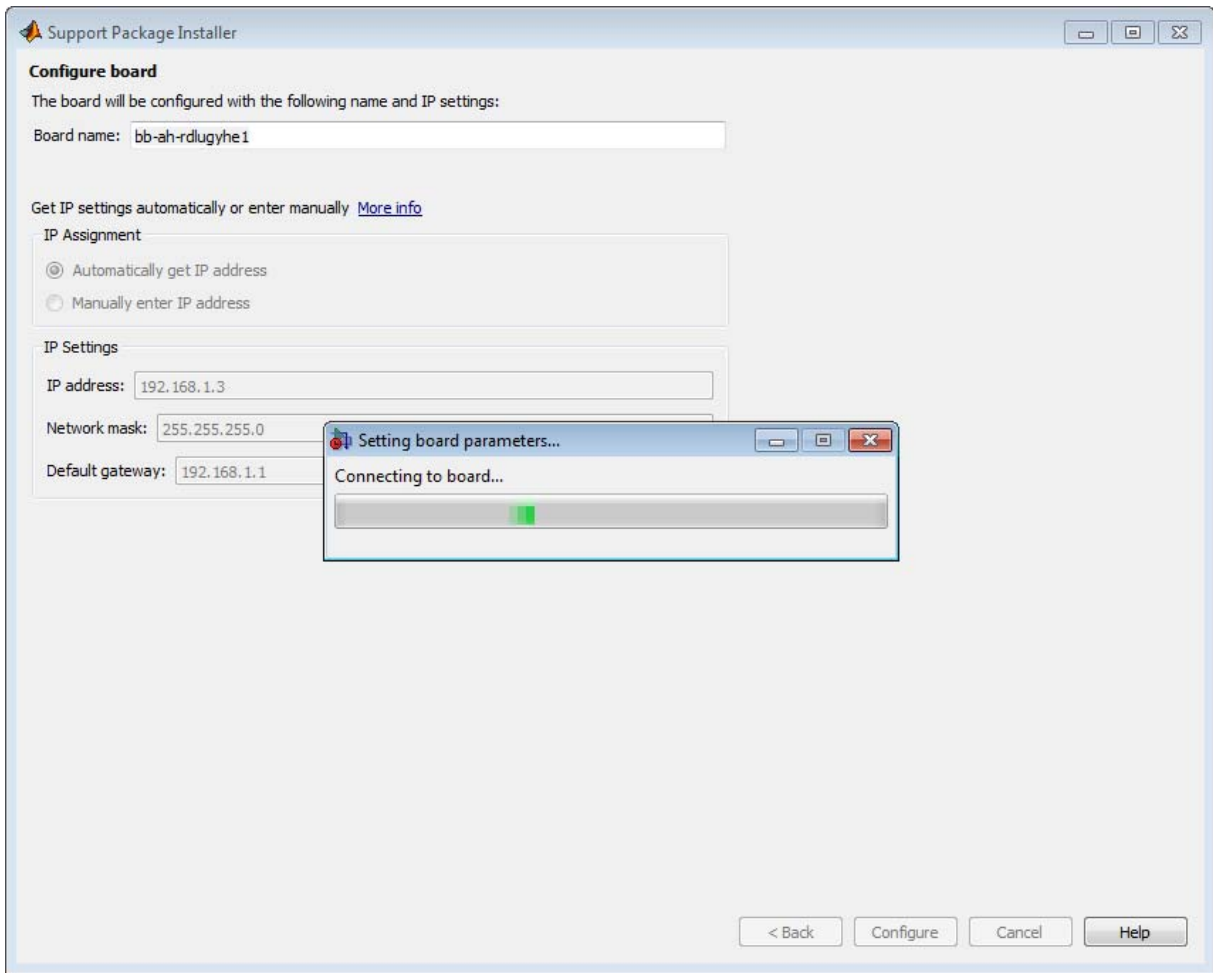
To enter static IP settings for your board, follow these guidelines:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

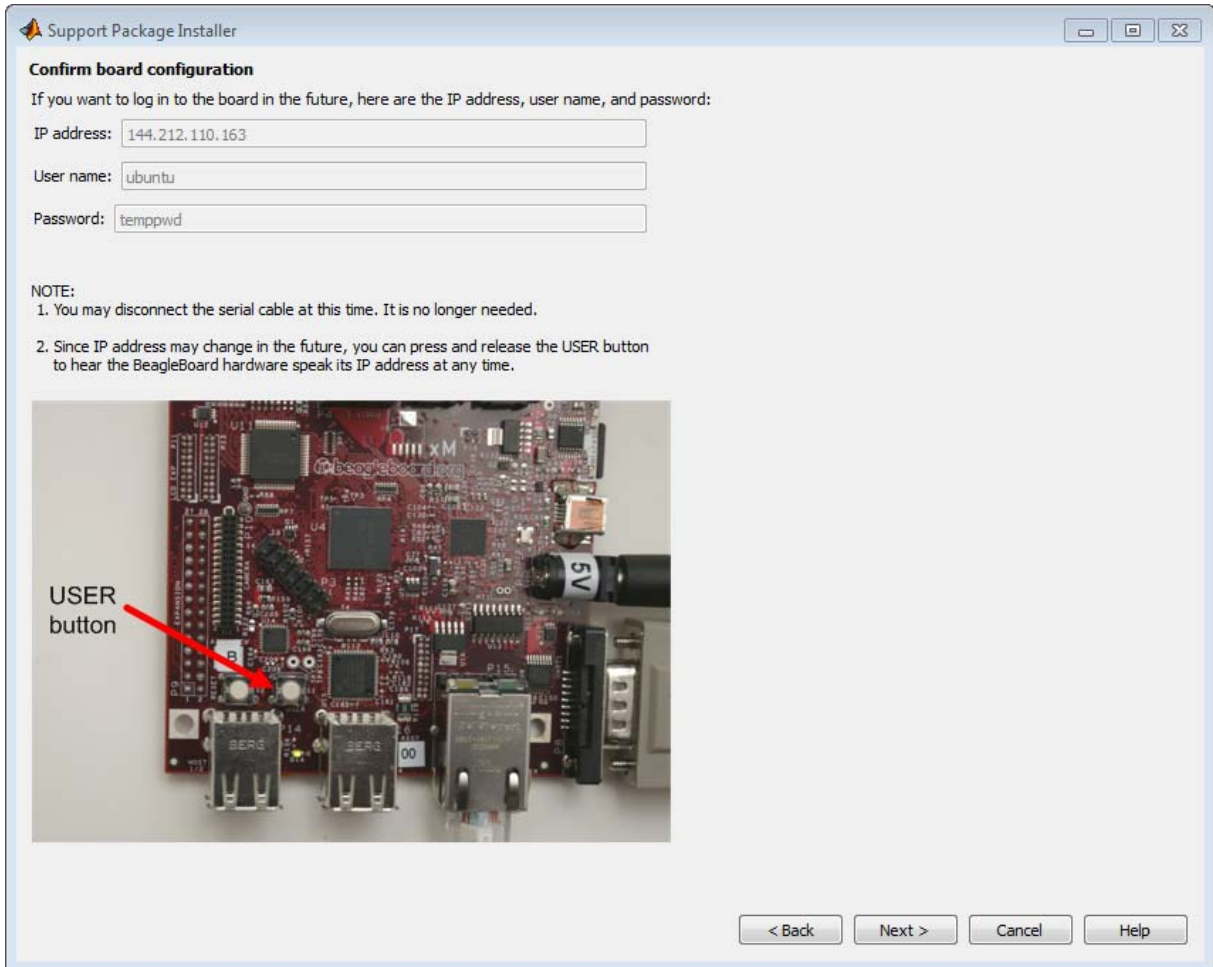
For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- **Network mask** to use the same network mask value, 255.255.255.0.
- **IP address** to an unused IP address, between 192.168.1.2 and 192.168.1.254.

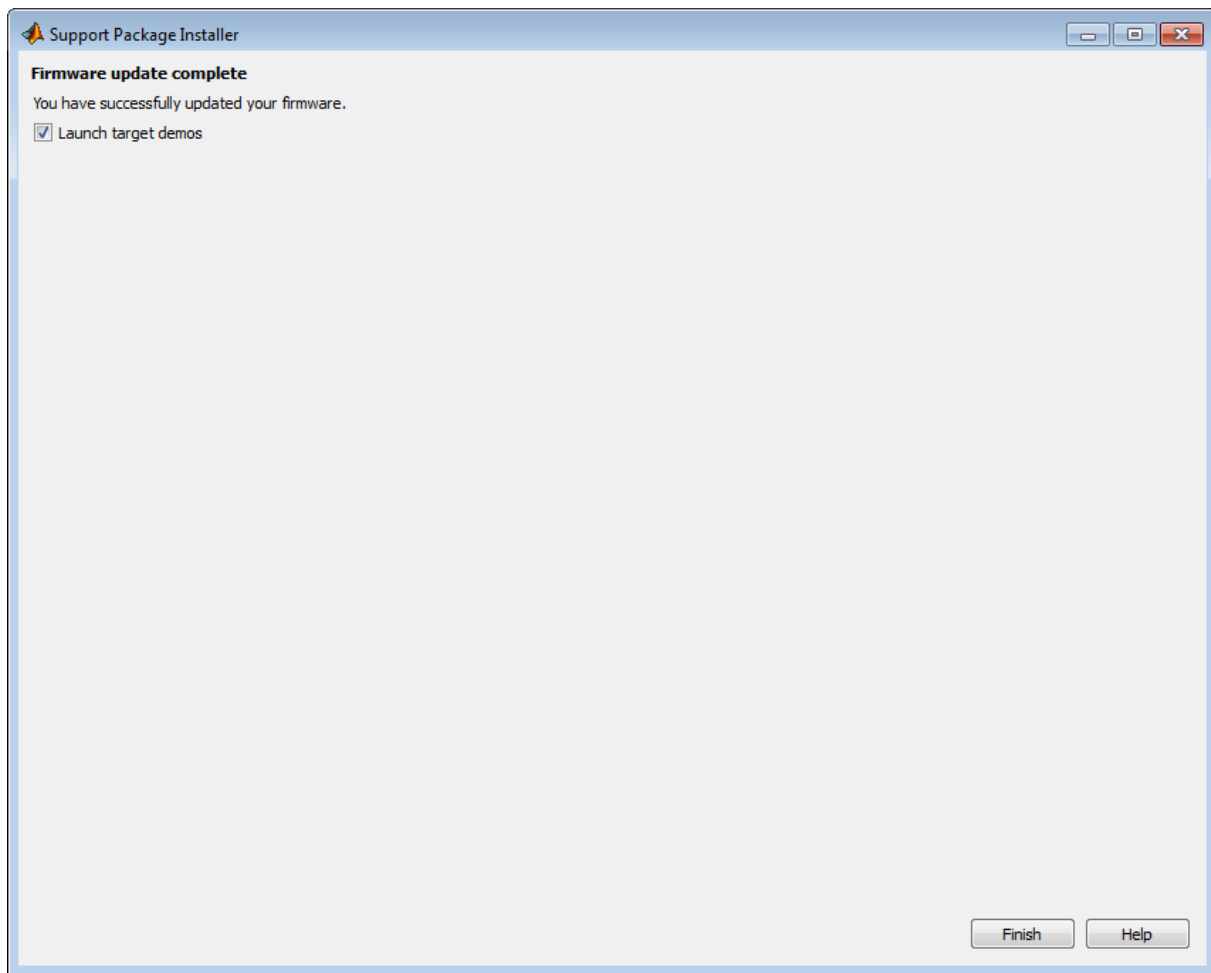
When you click **Configure**, Support Package Installer opens a serial connection and applies the settings to the board.



Make a note of the board information. Then, click **Next**.



Click **Finish**. If **Show support package examples** is enabled, Support Package Installer opens the example page for BeagleBoard hardware.



Choose the Type of Serial Cable

This topic shows you how to choose a serial cable for connecting your host computer to the BeagleBoard hardware.

Two types of cables are available for this purpose:

- DB9 null modem M/F cable
- USB to DB9 male adapter cable

DB9 null modem M/F cable:

- If your host computer has a DB9 male serial connector similar to the one in the following image, you can use a DB9 null modem M/F cable.



- To find the appropriate cable online, search for “DB9 low profile null modem M-F cable”.

USB to DB9 male adapter cable:

- If your host computer has a USB port, you can use a USB to DB9 male adapter cable.
- To find the appropriate cable online, search for "USB to RS-232 DB9 Serial Adapter”.
- Some USB-to-serial adapters do not appear in the list of serial connections immediately after you install the software drivers. To solve this issue, disconnect and reconnect the adapter, or reboot your host computer.
- To avoid issues with the software/driver quality, we recommend choosing an adapter that has good customer feedback ratings, or using the DB9 null modem M/F cable instead.

Connect to Serial Port on BeagleBoard Hardware

This topic shows you how to configure and open a command line session with the target hardware.

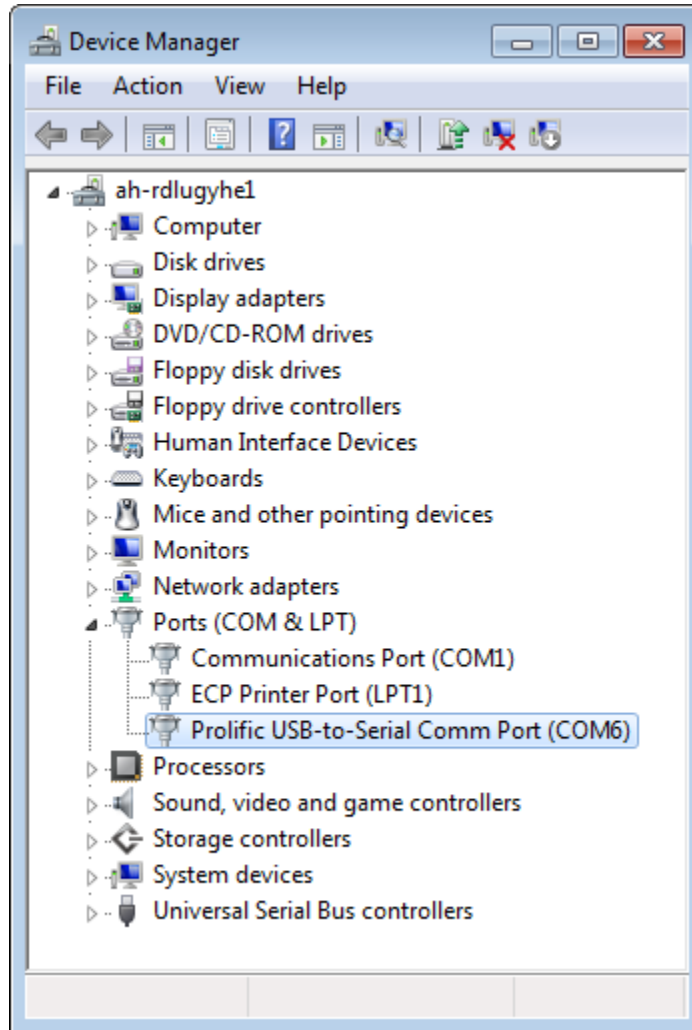
This is an optional task that you can use to:

- View the standard output while the board boots.
- Get or set the IP address, as described in “Get IP Address of BeagleBoard Hardware” on page 57-29 topic.

To open a serial connection to your board:

- 1** Connect a serial cable from your host computer to the female DB9 connector on the board.
- 2** Identify the COM port for your serial connection. In Windows 7, use the “search programs and files” feature in the Start menu to find “Device Manager”. Open Device Manager, and expand **Ports (COM & LPT)** to see the list of serial connections.

For example, the following image shows that the DB9 serial port called “Communications Port” uses COM1. Similarly, the image shows that the USB-to-serial adapter called “Prolific USB-to-Serial Comm Port” uses COM6.

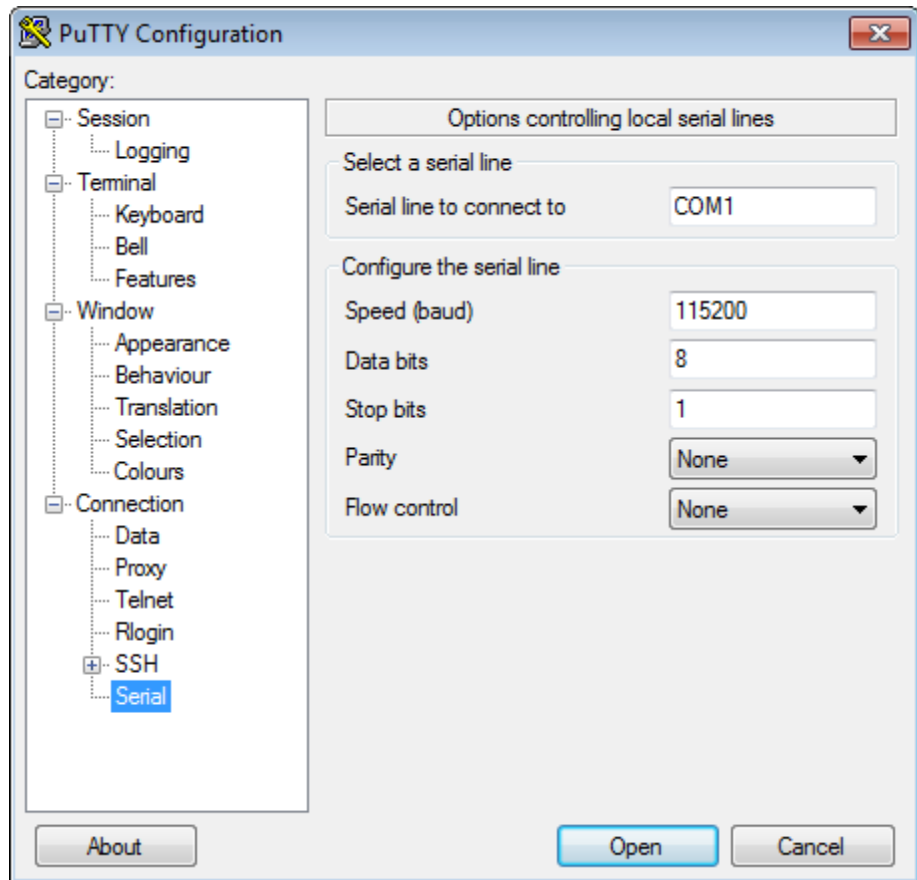


Note If you have trouble locating the COM port number, search the Windows “Help and Support” for “Device Manager”.

3 In the MATLAB Command Window, start PuTTY by entering:

```
h = beagle;  
h.openShell('serial')
```

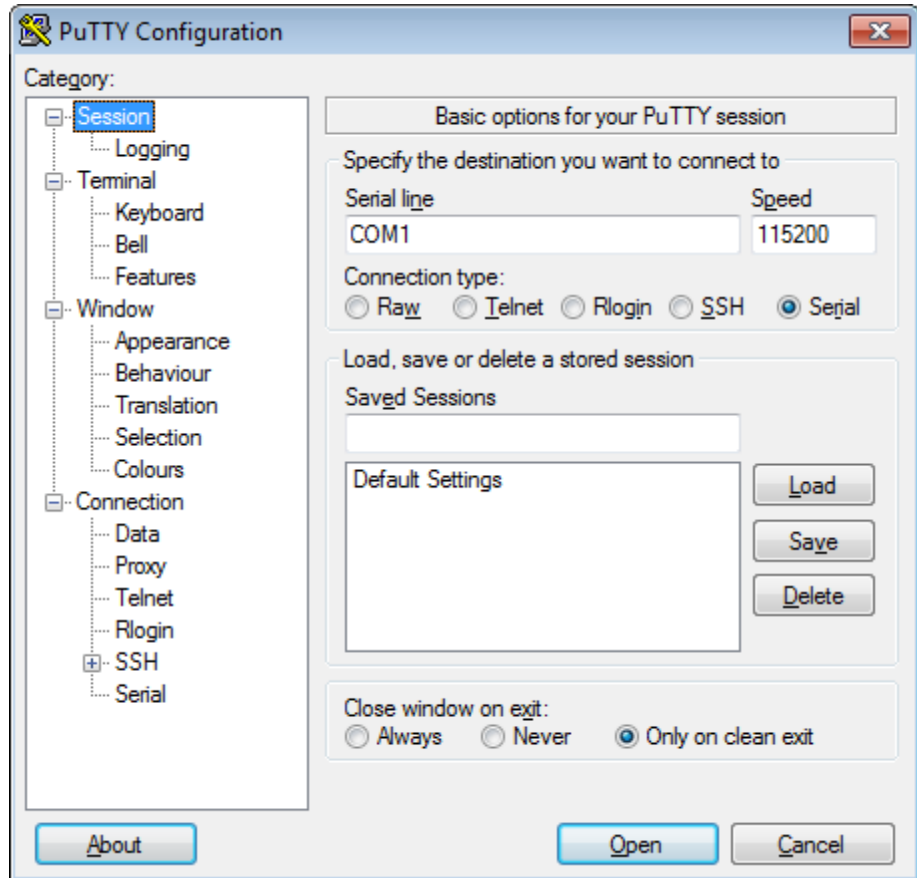
- 4 In the PuTTY dialog box that opens, select the **Serial** category.



- 5 Set the following values:

- **Serial line to connect to:** If you are using your computer's built-in serial port, try COM1. Otherwise, use the Device Manager in Windows to determine the COM port number.
- **Speed (baud):** 115200

- **Flow control:** None
- 6 In the PuTTY dialog box, select the **Session** category.



- 7 Set the **Connection type** parameter to **Serial**, enter a new name for **Saved Sessions**, click **Save**, and click **Open**.

Note Do not save the serial settings to **Default Settings**. The **Default Settings** must remain configured to use SSH in order to run your Simulink model on the BeagleBoard hardware. Otherwise, trying to run the model on BeagleBoard hardware produces an error message.

- 8** When a terminal window opens, press the **Enter** key on your keyboard. The terminal window displays the Linux command line.
- 9** When you are finished, enter `logout` on the Linux command line, and close PuTTY to end the serial connection.

See Also

- “Get IP Address of BeagleBoard Hardware” on page 57-29
- For more information about PuTTY, see:
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

Configure Network Connection with BeagleBoard Hardware

You can configure the IP settings of the BeagleBoard hardware by running Linux shell commands directly on the BeagleBoard hardware.

To inspect and reconfigure the IP settings on a board that already has the new firmware, follow the procedure in this section that starts with “To configure the board to use DHCP or static IP settings”.

To configure the IP settings while you are replacing the firmware on your BeagleBoard hardware, see “Replace Firmware on BeagleBoard Hardware” on page 57-3.

You may need to reconfigure the IP settings if your board:

- Has unknown IP settings.
- Is unreachable using a network connection.
- Is being moved to a network or direct Ethernet connection that uses static IP settings.
- Is being moved from a network that used static IP settings to one that uses DHCP services.

When you complete the procedure in this topic, the board should be able to communicate over the network to which it is connected.

Before starting the procedure in this topic, it helps to understand the conditions under which networks use DHCP or static IP settings:

- If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet, configure the board to use DHCP services. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.
- If your board is directly connected to an Ethernet port on your computer, or connected to an isolated network without DHCP services, configure the board to use static IP settings.

To configure the board to use DHCP or static IP settings:

- 1** Open a serial command line session, as described in “Connect to Serial Port on BeagleBoard Hardware” on page 57-20

Alternatively, if your board is connected to a monitor, keyboard, and mouse, you can log in to the Ubuntu desktop and open a terminal window.

- 2** Display the contents of the `/etc/network/interfaces` file. Enter:

```
cat /etc/network/interfaces
```

If the board is configured to use DHCP services (the default configuration), `dhcp` appears at the end of the following line:

```
iface eth0 inet dhcp
```

If the board is configured to use static IP settings, `static` appears at the end of the following line:

```
iface eth0 inet static
```

- 3** Create a backup of the `/etc/network/interfaces` file. Enter:

```
sudo cp /etc/network/interfaces /etc/network/interfaces.backup
```

Enter the root password when prompted (default: `temppwd`).

- 4** Edit `interfaces` using a simple editor called `nano`. Enter:

```
sudo nano /etc/network/interfaces
```

- 5** Edit the last word of line that starts with `iface eth0 inet`.

To use DHCP services, edit the line to say:

```
iface eth0 inet dhcp
```

To use static IP settings, edit the line to say:

```
iface eth0 inet static
```

- 6** For static IP settings, add lines for address, netmask, and gateway. For example:

```
iface eth0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    gateway 192.168.1.1
```

Note For static IP settings:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- netmask to use the same network mask value, 255.255.255.0.
 - address to an unused IP address, between 192.168.1.2 and 192.168.1.254.
-

- 7** Tell nano to exit and save the changes:

Press **Ctrl+X**.

Enter Y to save the modified buffer.

For “File Name to Write: interfaces”, press Enter.

The nano editor confirms that it “Wrote # lines” and returns control to the command line.

- 8** Reboot your board for the settings to take effect:

```
sudo shutdown -r now
```

- 9** Test the IP settings by logging in to the board over a telnet session.

Note You can use the `ifconfig` command to temporarily change the IP settings. Rebooting the board removes the `ifconfig` settings and restores the `/etc/network/interfaces` settings.

To change the IP settings temporarily, open a Linux command line, and enter `ifconfig`, the device id, a valid IP address, netmask, and the appropriate network mask. For example:

```
ifconfig eth0 192.168.45.12 netmask 255.255.255.0
```

Related Examples

- “Run Model on BeagleBoard Hardware” on page 57-34
- “Get IP Address of BeagleBoard Hardware” on page 57-29

Get IP Address of BeagleBoard Hardware

You can get the IP address of the BeagleBoard hardware by listening to the audio output, or by using the Linux command line. This is an optional task that you can use to:

- Preparing a model to run on a board that is different from the previous one you used.
- Opening a telnet session with a board whose IP settings you do not know.
- Configuring the UDP Send block.

When you prepare a model to run on a board, your model Configuration Parameters reuse the IP settings of the previous board you used. If you are working with a different board on the same network, get the IP address of the your new board, and update the **Host name** parameter in the Configuration Parameters. For more information, see “Run Model on BeagleBoard Hardware” on page 57-34.

If Ethernet port on the BeagleBoard hardware board has an IP address, the audio output on the board can read the IP address out loud. For example the board can say “My IP address is one hundred and forty four point two one two point one one zero point two zero six.”

To listen your board’s IP address using headphones or speakers:

- 1** Plug headphones or speakers into the AUDIO OUT jack on the board.
- 2** Press the USER button on the board.
- 3** Write down the IP address as the board reads it out over the headphones or speakers.

The Linux command line on the board can provide the IP address of the Ethernet port.

To get your board’s IP address command line session:

- 1** Open a serial command line session on your board, as described in “Connect to Serial Port on BeagleBoard Hardware” on page 57-20.

If your board is connected to a monitor, keyboard, and mouse, you can also log in to the Ubuntu desktop and open a terminal window.

2 At the Linux command line, enter `ifconfig`.

3 Locate the IP address following `inet addr` in the command line output. For example, locate `inet addr:144.212.110.206` on line 9 of the following output:

```
omap login: ubuntu
Password:
Last login: Wed Nov 23 15:40:36 EST 2011 from ah-rdlugyhe1.dhcp.mathworks.com on pts/1
Welcome to Ubuntu 11.04 (GNU/Linux 3.0.4-x3 armv7l)

* Documentation:  https://help.ubuntu.com/
ubuntu@omap:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr d2:a7:db:b3:c4:81
          inet addr:144.212.110.206  Bcast:144.212.110.255  Mask:255.255.255.0
          inet6 addr: fe80::d0a7:dbff:feb3:c481/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1488  Metric:1
          RX packets:125537 errors:0 dropped:0 overruns:0 frame:0
          TX packets:76 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:18103491 (18.1 MB)  TX bytes:7670 (7.6 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:240 (240.0 B)  TX bytes:240 (240.0 B)

ubuntu@omap:~$
```

Related Examples

- “Connect to Serial Port on BeagleBoard Hardware” on page 57-20
- “Run Model on BeagleBoard Hardware” on page 57-34
- “Configure Network Connection with BeagleBoard Hardware” on page 57-25

Open Block Library for BeagleBoard Hardware

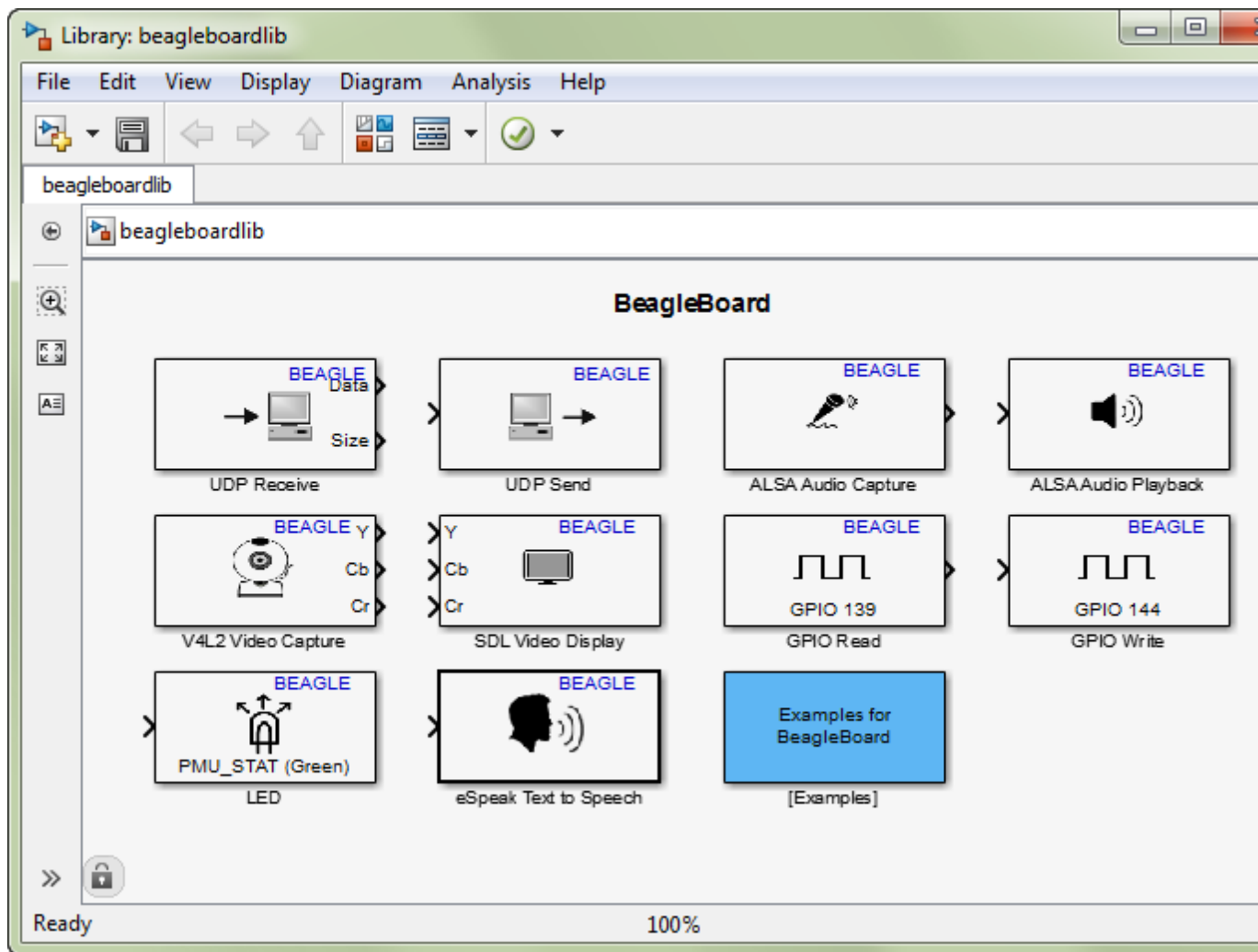
This example shows how to open the block library for BeagleBoard hardware.

This block library provides support for protocols and APIs available on BeagleBoard hardware.

To open the block library from the MATLAB Command Window, enter:

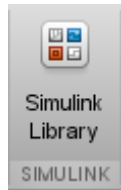
```
beagleboardlib
```

The following block library opens.

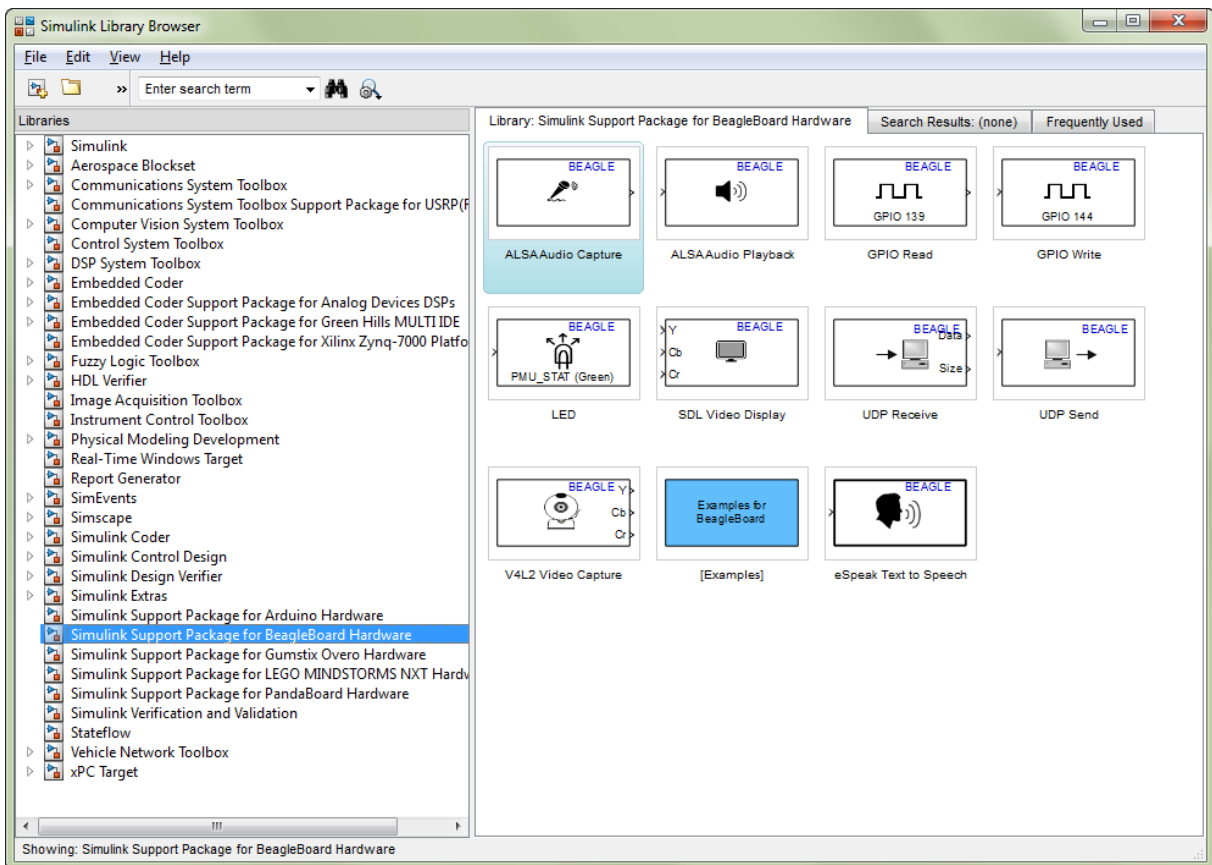


To open the block library from the Simulink Library Browser:

Open the Simulink Library Browser by entering `simulink` in the MATLAB Command Window, or by clicking the following icon on the MATLAB toolbar.



In the Simulink Library Browser, click **Simulink Support Package for BeagleBoard Hardware**.



Run Model on BeagleBoard Hardware

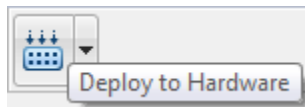
This example shows how to prepare, configure, and run a simple model on your BeagleBoard hardware.

Before starting this procedure:

- Connect your board to the network and to a power supply.
- Create or open a Simulink model.

To prepare and run the model:

- 1** Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 2** In your model, select **ToolsRun on Target HardwarePrepare to Run**. This action changes the model Configuration Parameters.
- 3** In the Run on Target Hardware pane that opens, set the **Target hardware** parameter to **BeagleBoard**.
- 4** If you have changed boards since the last time you updated the firmware or ran a model, update the **Host name**, **User name**, and **Password** parameters.
- 5** Click the Deploy to Hardware button.



This action automatically downloads and runs your model on the board.

Note Pressing RESET or cycling the power on the BeagleBoard hardware during this step can cause the SSH utility to hang.

Running a new or updated model on the board:

- Automatically stops a running model with the same name.
- Does not stop running models that have other names.

To stop a model running on the board, enter the following commands in the MATLAB Command Window:

```
h=beagle;  
h.stop('modelName');
```

For example, to stop the sumdiff model, enter:

```
h=beagle;  
h.stop('sumdiff');
```

To restart a model that was previously running on the board, or to run multiple instances of a model, enter the following commands in the MATLAB Command Window:

```
h=beagle;  
h.run('modelName');
```

Note You do not need to enter `h=beagle;` multiple times if a previous instance of `h` is available in the MATLAB Workspace.

For example, to restart the sumdiff model you stopped in the previous example, enter:

```
h.run('sumdiff');
```

See Also

- “Create the Simple Model”
- “Configure Network Connection with BeagleBoard Hardware” on page 57-25
- “Tune and Monitor Model Running on BeagleBoard Hardware” on page 57-36
- “Overview of Model Referencing” on page 6-2

Tune and Monitor Model Running on BeagleBoard Hardware

In this section...

“About External Mode” on page 57-36

“Run Your Simulink Model in External Mode” on page 57-37

“Stop External Mode” on page 57-38

About External Mode

You can use External mode to tune parameters and monitor a model running on your target hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on model results in real-time, in order to find the optimal values to achieve desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to re-run the model each time you change parameters. External mode also lets you develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

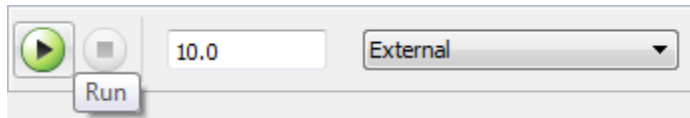
The following list provides an overview of the parameter tuning process with External mode:

- In the model on your host computer, you enable External mode.
- In the model on your host computer, you tell Simulink software to run your model on the target hardware.
- Simulink software runs the model on the target hardware.
- You use the model on the host computer as a user interface for interacting with the model running on the target hardware:
 - When open blocks and apply new parameter values on the host computer, External mode updates the corresponding values on target hardware.

- If your model contains blocks for viewing data, such as Scope or Display blocks, External mode sends the corresponding data from the target hardware to those blocks on the host computer.
- You determine the optimal parameter values by adjusting parameter values on the host computer and observing data/outputs from the target hardware.
- When you are finished, you save the new parameter values, disable External mode, and save the model.

Run Your Simulink Model in External Mode

- 1 Create a network connection between the BeagleBoard hardware and your host computer. See “Configure Network Connection with BeagleBoard Hardware” on page 57-25
- 2 On the model toolbar, set **Simulation mode** to External mode.



- 3 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 4 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\s1_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

- 5 While the model is running in External mode, you can change parameter values in the model on your host computer and observe the corresponding changes in the model running on the hardware.

If your model contains blocks from the Simulink Sinks block library, the sink blocks in the model on your host computer display the values generated by the model running on the hardware.

If your model does not contain a sink block to which External mode can send data, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2012a Student1\toolbox\
  realtime\realtime\+realtime\extModeAutoConnect.p>
  extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2012a Student1\toolbox\
  realtime\realtime\sl_customization.p>myRunCallback at 149
```

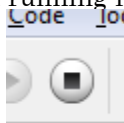
You can disregard this warning, or you can add a sink block to the model.

Note External mode increases the processing burden of the model running on the board. If the software reports an overrun, disable External mode.

Note If you have the Embedded Coder or Simulink Coder products, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

Stop External Mode

To stop the model running in External mode, click the Stop button located on



the model toolbar.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

Detect and Fix Task Overruns on BeagleBoard Hardware

You can configure a model running on the target hardware to detect and notify you of when task overrun occurs.

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. When a task is required to perform extra processing and takes longer than normal to execute, it may be scheduled to execute before a previous instance of the same task has completed. The result is a task overrun.

To enable overrun detection:

- 1** In your model, click **Tools > Run on Target Hardware > Options**.
- 2** In the Run on Target Hardware pane that opens, select the **Enable overrun detection** check box.
- 3** Click **OK**.

When a task overrun occurs, the command prompt on the host machine repeatedly prints an “Overrun” error message, such as “Overrun — rate for subrate task 1 is too fast”, until the model stops.

To fix an overrun condition:

- Simplify the model.
- Increase the sample times for the model and the blocks in it. For example, change the **Sample time** parameter in all of your data source blocks, such as blocks for input devices, from 0.1 to 0.2.

Note External mode increases the processing burden of the model running on your board. If the software reports an overrun, disable External mode.

Gumstix Overo Hardware

- “Install Support for Gumstix® Overo® Hardware” on page 58-2
- “Replace Firmware on Gumstix® Overo® Hardware” on page 58-3
- “Open a Serial Command-line Session with Gumstix® Overo® Hardware” on page 58-20
- “Configure Network Settings on Gumstix® Overo® Hardware” on page 58-25
- “Get IP Address of Gumstix® Overo® Hardware” on page 58-29
- “Open Block Library for Gumstix® Overo® Hardware” on page 58-31
- “Run Model on Gumstix® Overo® Hardware” on page 58-33
- “Stop or Restart a Model Running on Gumstix® Overo® Hardware” on page 58-35
- “Tune and Monitor Model Running on Gumstix® Overo® Hardware” on page 58-36
- “Detect and Fix Task Overruns on Gumstix® Overo® Hardware” on page 58-39
- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 58-40
- “Access the Linux Desktop Remotely Using VNC” on page 58-41

Install Support for Gumstix Overo Hardware

You can add support for Overo hardware to the Simulink product. When you complete this process, and replace the firmware on the Overo hardware, you can run Simulink models on Overo hardware, as described in

To install support for Overo hardware:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2 Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

During this process, Support Package Installer installs:

- Third-party software development tools
- A Simulink block library called *Simulink Support Package for Gumstix Overo Hardware*.
- Examples

These instructions occasionally refer to Overo hardware as a “board” or as “target hardware”.

Replace Firmware on Gumstix Overo Hardware

Replace the firmware on the Overo hardware with a customized Linux distribution.

Before replacing the firmware, install the support package.

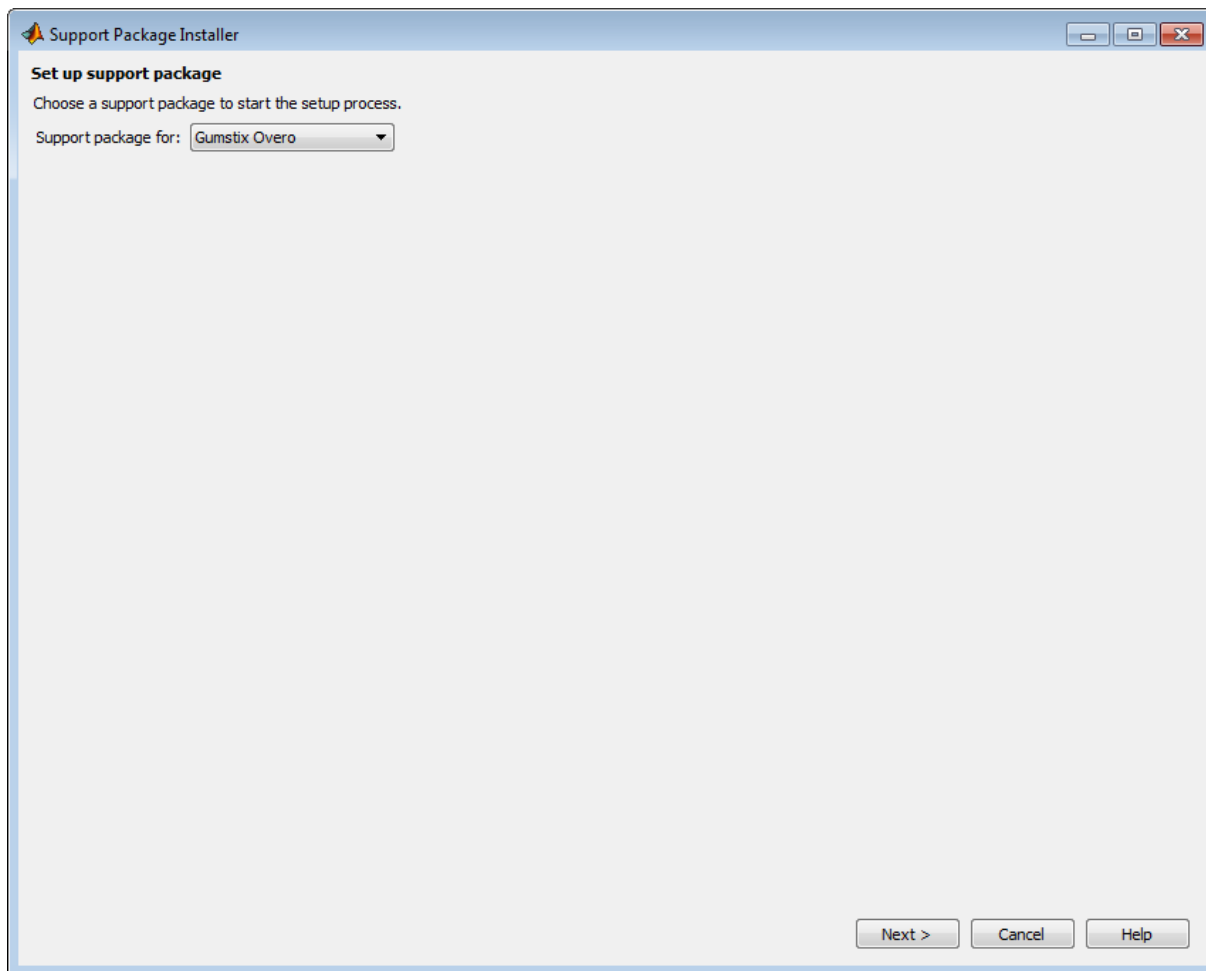
After replacing the firmware, you can run a Simulink model on the Overo hardware.

The following overview summarizes the firmware replacement process:

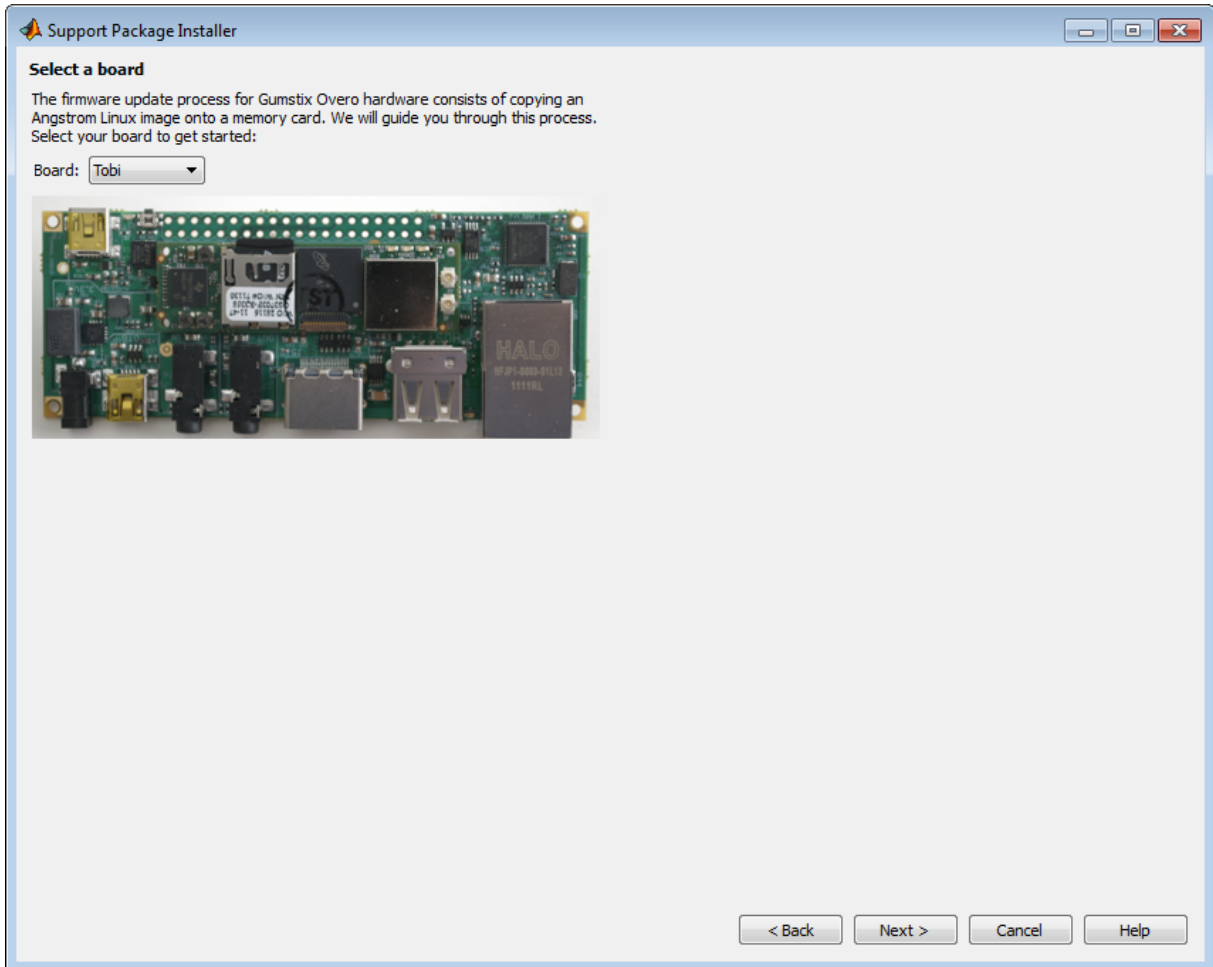
- 1** Support Package Installer locates a firmware image on your host computer or downloads new one.
- 2** Support Package Installer uses the host computer, not the Overo hardware, to write the firmware image to a microSD card.
- 3** You transfer the microSD card to the Overo hardware.
- 4** Support Package Installer configures the Overo hardware.

To replace the firmware on your Overo hardware:

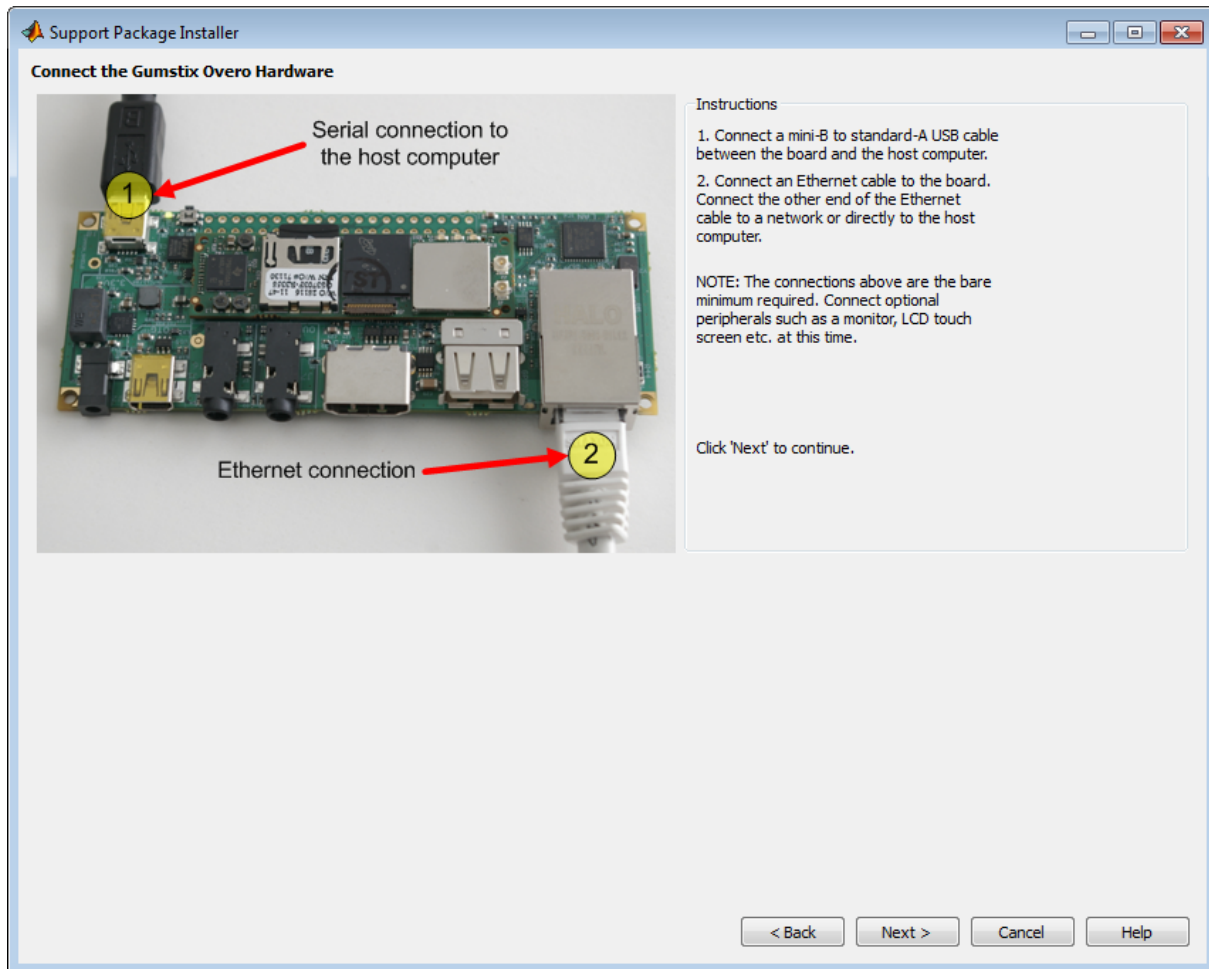
- 1** Open the **Update firmware** screen in Support Package Installer using one of the following methods:
 - Click **Continue** in the **Install/update complete** screen of Support Package Installer.
 - In a model, select **Tools > Run on Target Hardware > Update firmware**.
 - In a MATLAB Command Window, enter `targetupdater`.
- 2** Choose the Gumstix Overo option, and click **Next**.



3 Select the Overo board you are using, and click **Next**.



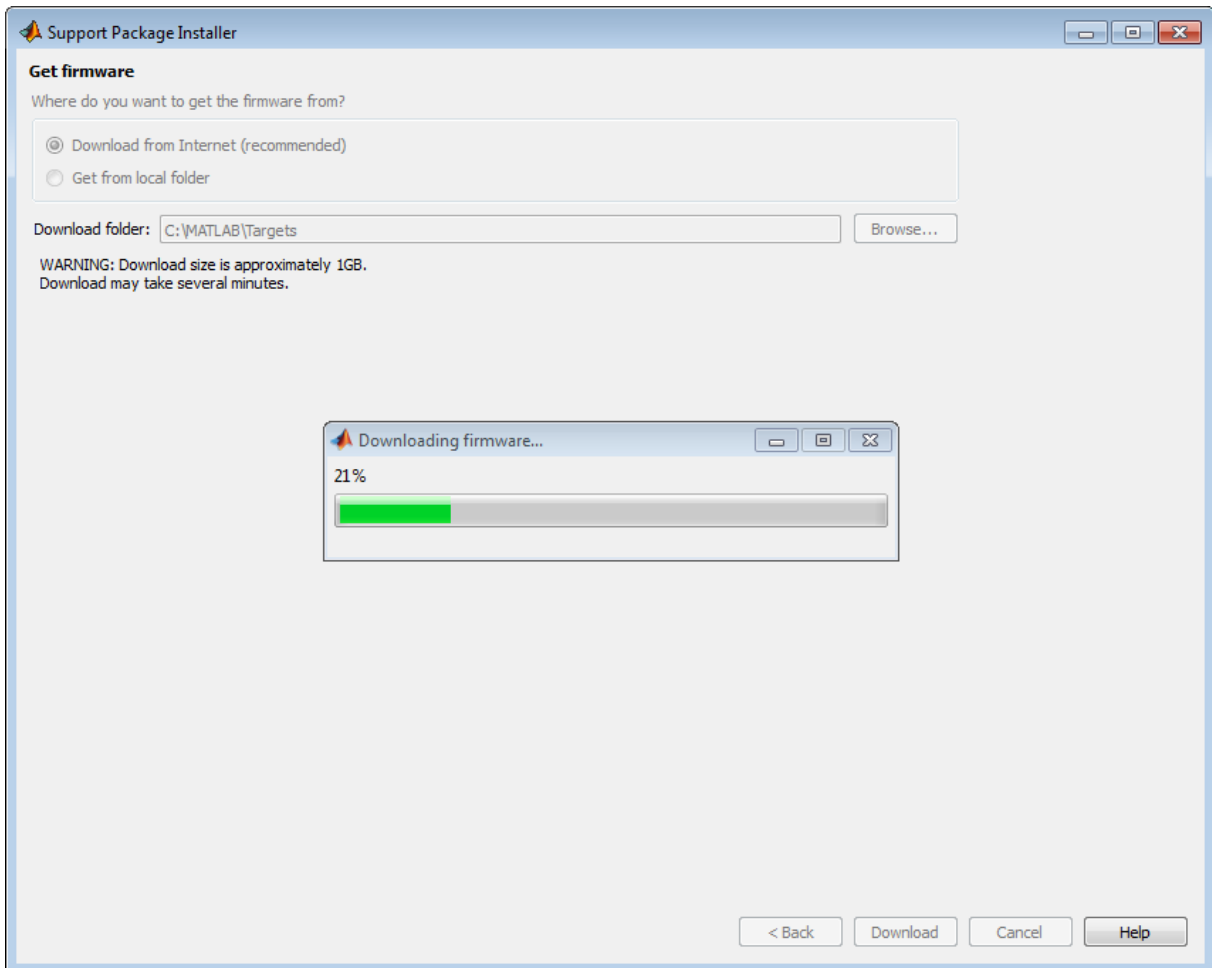
4 Make the connections shown in the following figure, and click **Next**.



- 5** Leave **Download from Internet (recommended)** selected and click **Download**. The length of time required to download the firmware depends on the speed of your connection.


If the correct firmware image is already present in the **Download folder**, the Support Package Installer automatically skips re-downloading the firmware image.

The file size of the firmware image is approximately 1 GB. The length of time required to download the firmware depends on the speed of your connection.



- 6 Insert the microSD card into a media card reader connected to your host computer. Windows assigns a drive letter to the memory card.
- 7 Click **Refresh** and select the drive letter of the microSD card:

- If only one drive letter is available, click **Next**.
- If multiple drive letters are available, open the Windows Start menu, choose **Computer**, and look under **Devices with Removable Storage** to identify the letter Windows assigned to the microSD card.

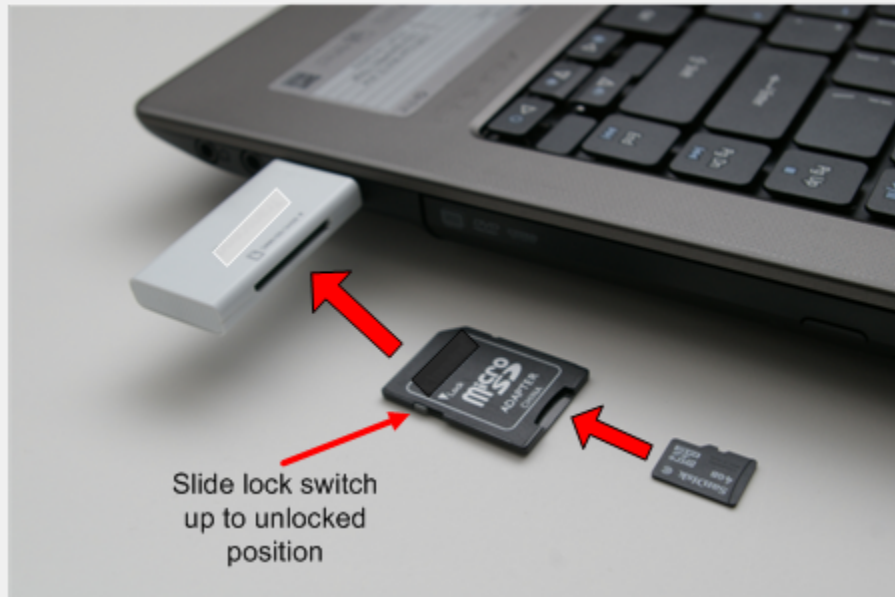
 Support Package Installer**Select a drive**

Insert a 4 GB or larger MicroSD memory card into a memory card reader on the host computer.

Select the drive letter that corresponds to the memory card reader:

Drive:


If you do not find the memory card reader in the list of drives, reinsert the memory card fully and click "Refresh".



< Back

Next >

- 8** Click **Write**. Support Package Installer overwrites all previous data on the memory card with the firmware. This process takes several minutes to complete.

 Support Package Installer

Write firmware

You chose to write to drive I:. The write operation may take several minutes.

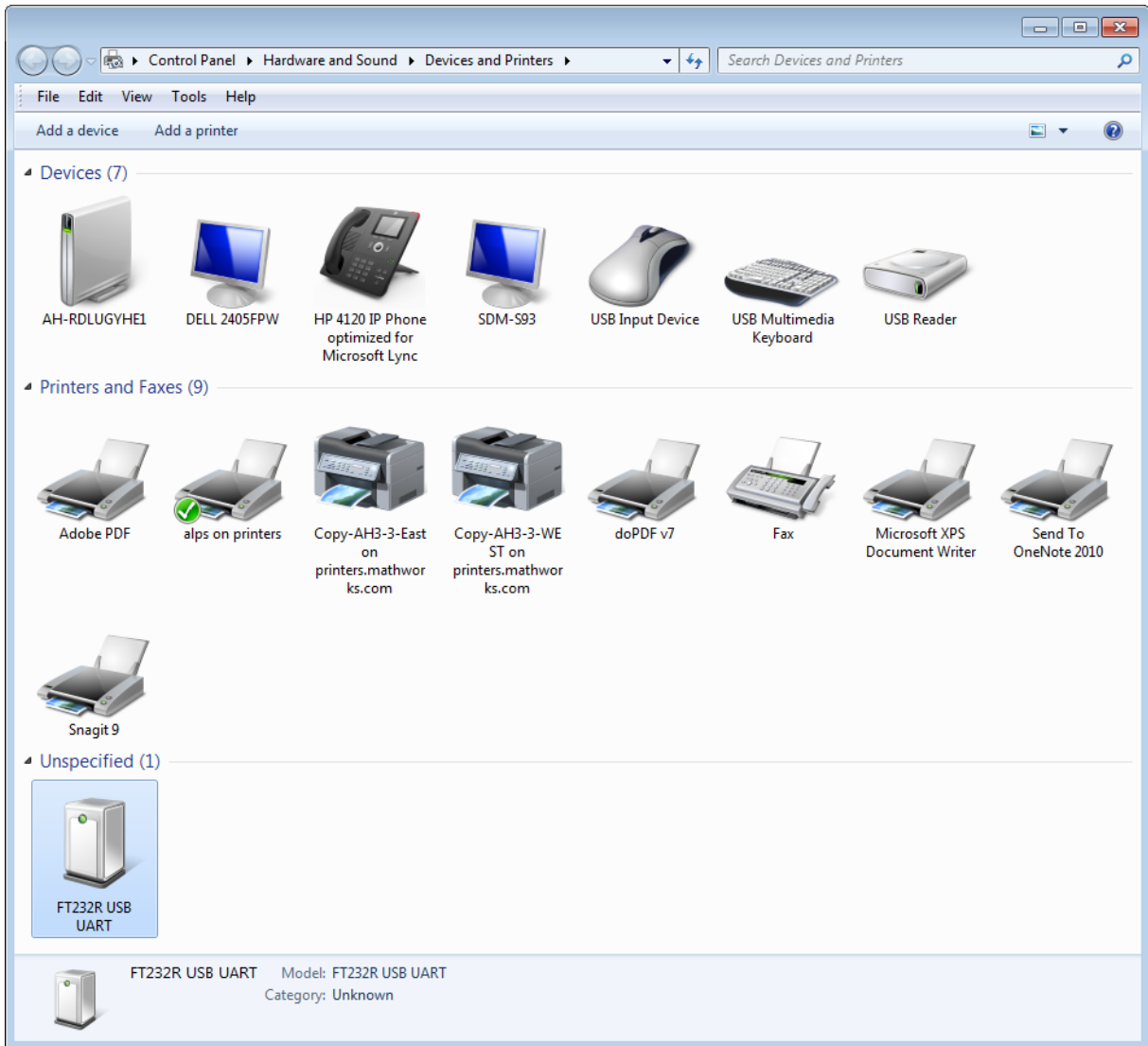


WARNING: All data in the memory card will be erased!

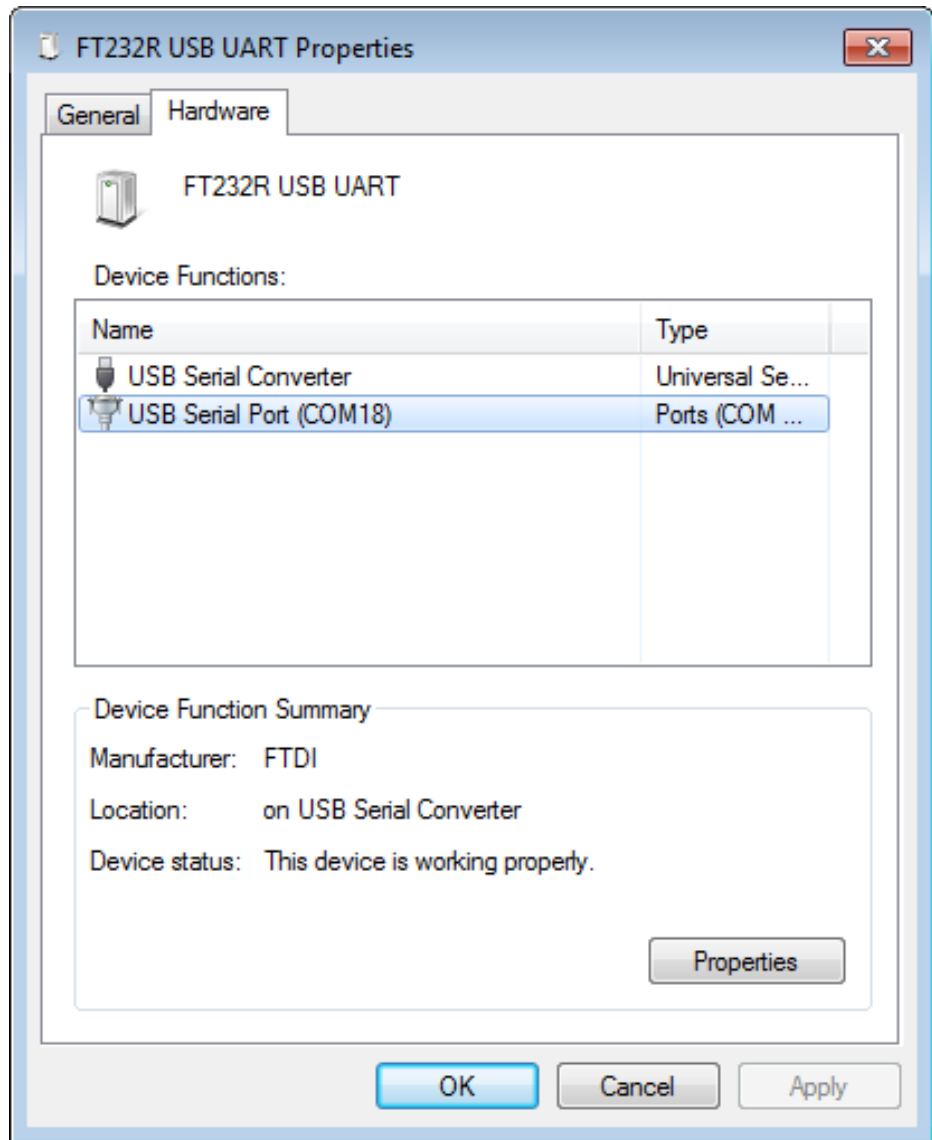
< Back

Write

9 In Microsoft Windows, open the **Devices and Printers**, and locate the Overo hardware device.

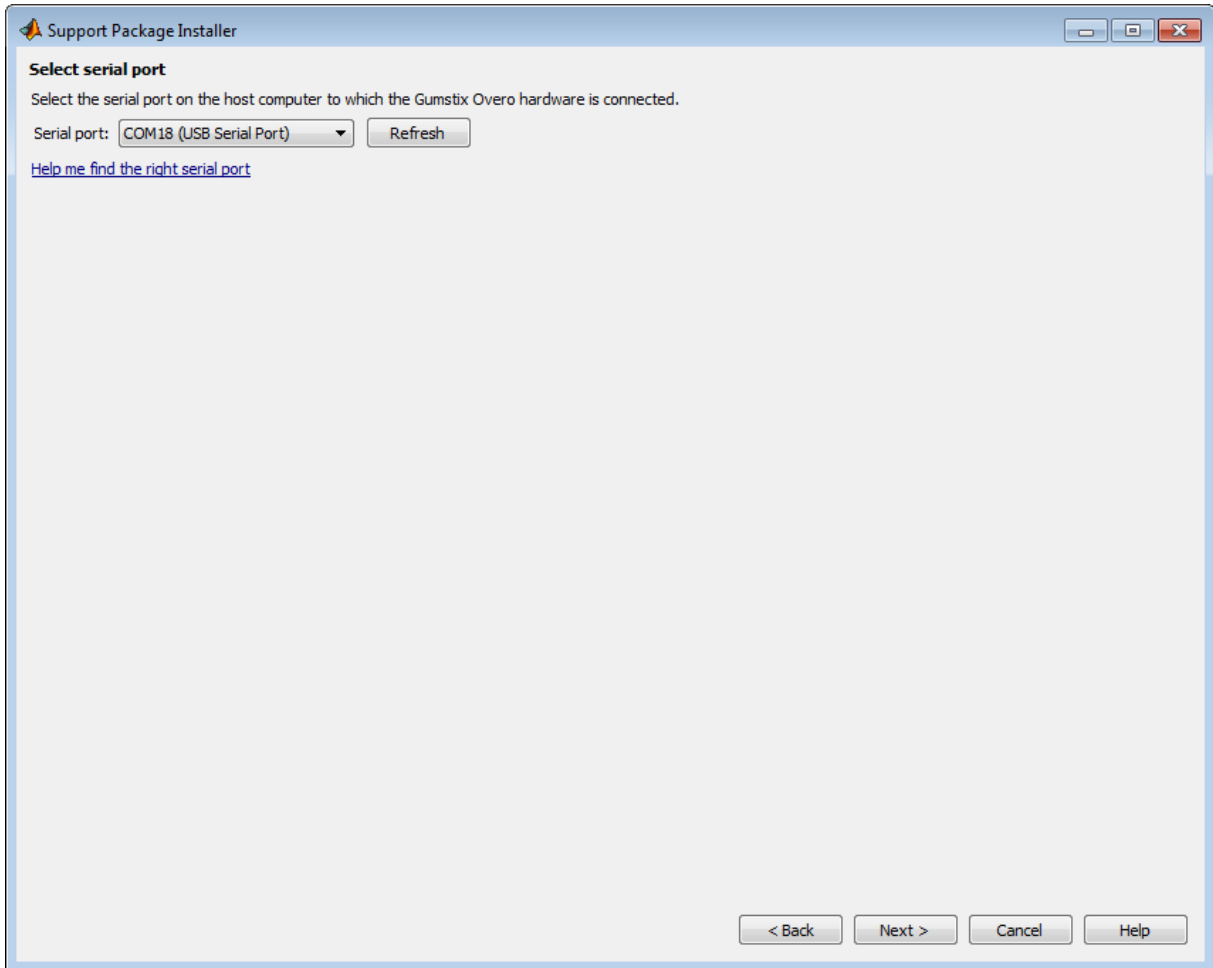


- 10 Get the COM port number from the **Hardware** tab of the Overo hardware device.

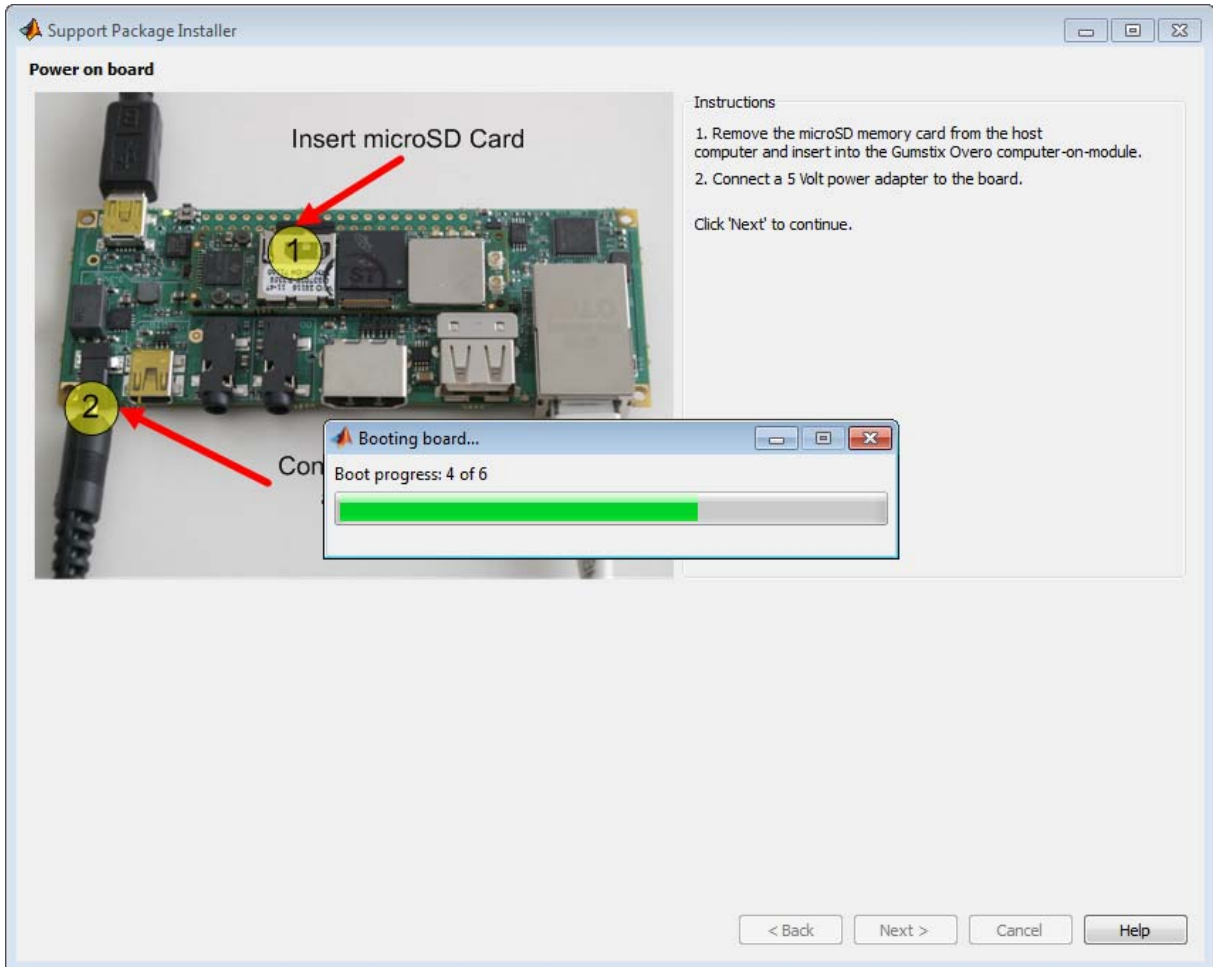


- 11 Set **Serial port** to the COM port number of the Overo hardware, and click **Next**.

If you just connected the Overo hardware using the USB cable, click **Refresh** to update the list of options.



- 12 Insert the microSD card into the Overo hardware. Then, connect the board to a 5 volt power supply, and click **Next**.



13

Review the automatically generated value of **Board name**.

Review the settings under **IP Assignment**:

- If the Overo hardware is connected to a network with DHCP services, leave **Automatically get IP address** selected and click **Configure**. A LAN or an Internet router typically provide DHCP services.

- If your board is directly connected to an Ethernet port on your computer or connected to an isolated network without DHCP services, select **Manually enter IP address**. In the dialog box that appears, give the board a unique name, enter static IP settings for the board, and click **Configure**.

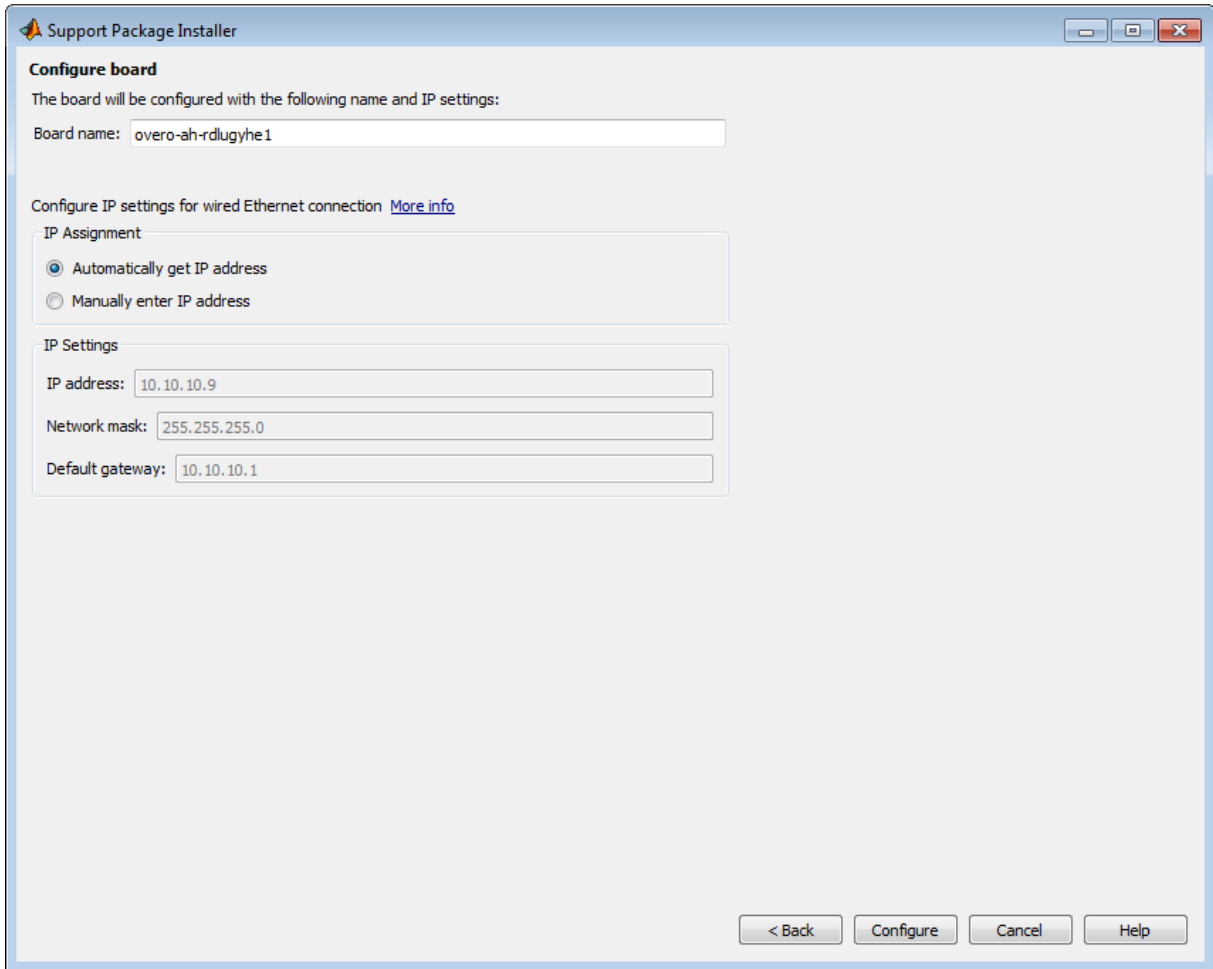
Note To enter static IP settings for your board, follow these guidelines:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- **Network mask** to use the same network mask value, 255.255.255.0.
 - **IP address** to an unused IP address, between 192.168.1.2 and 192.168.1.254.
-

When you click **Configure**, Support Package Installer opens a serial connection and applies the settings to the board.

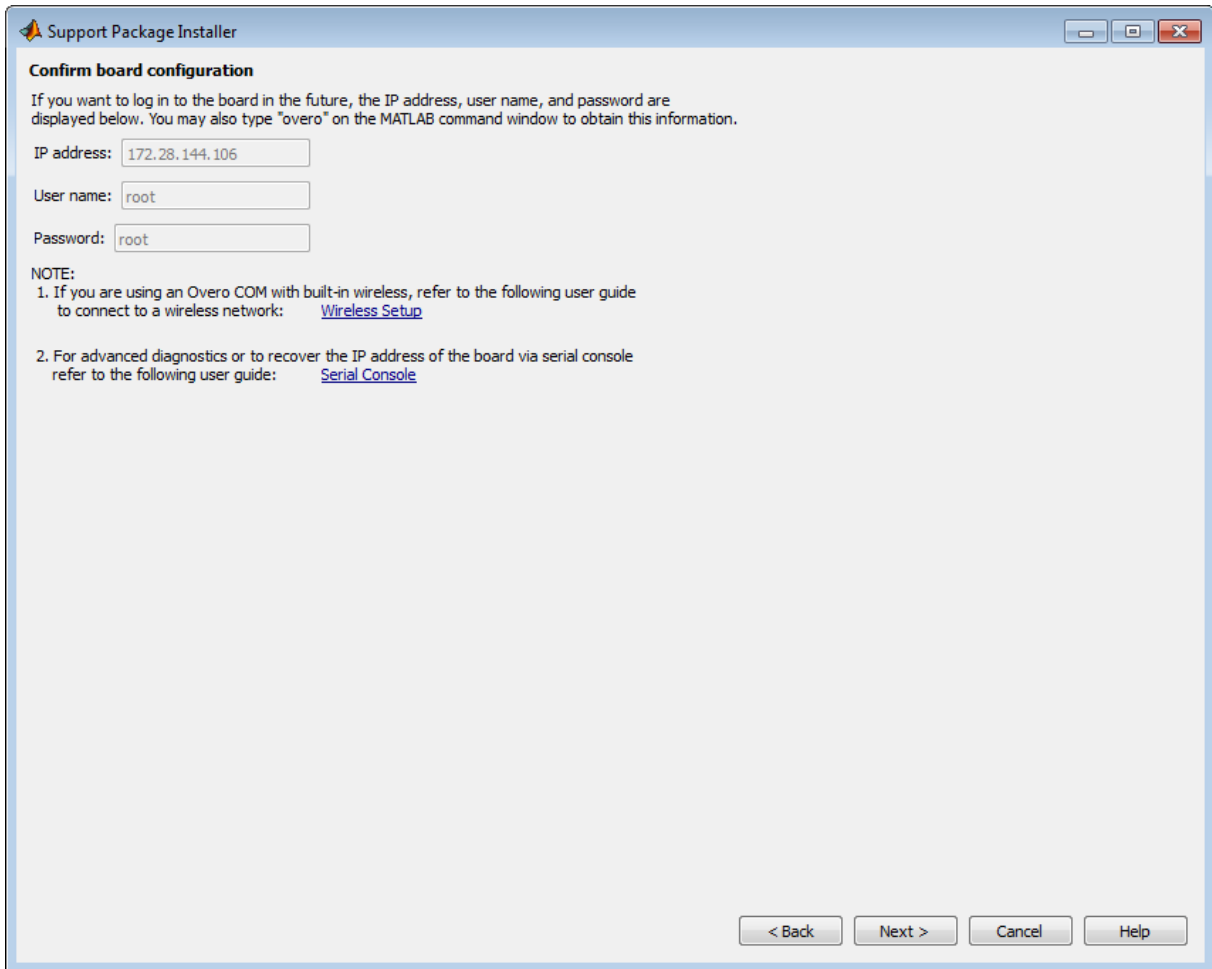


- 14** Make a note of the IP address, User name, and Password for use later on. Then, click **Next**.

Some Overo COMs have a built-in Wi-Fi module. To configure these wireless settings, see http://wiki.gumstix.org/index.php?title=Overo_Wifi.

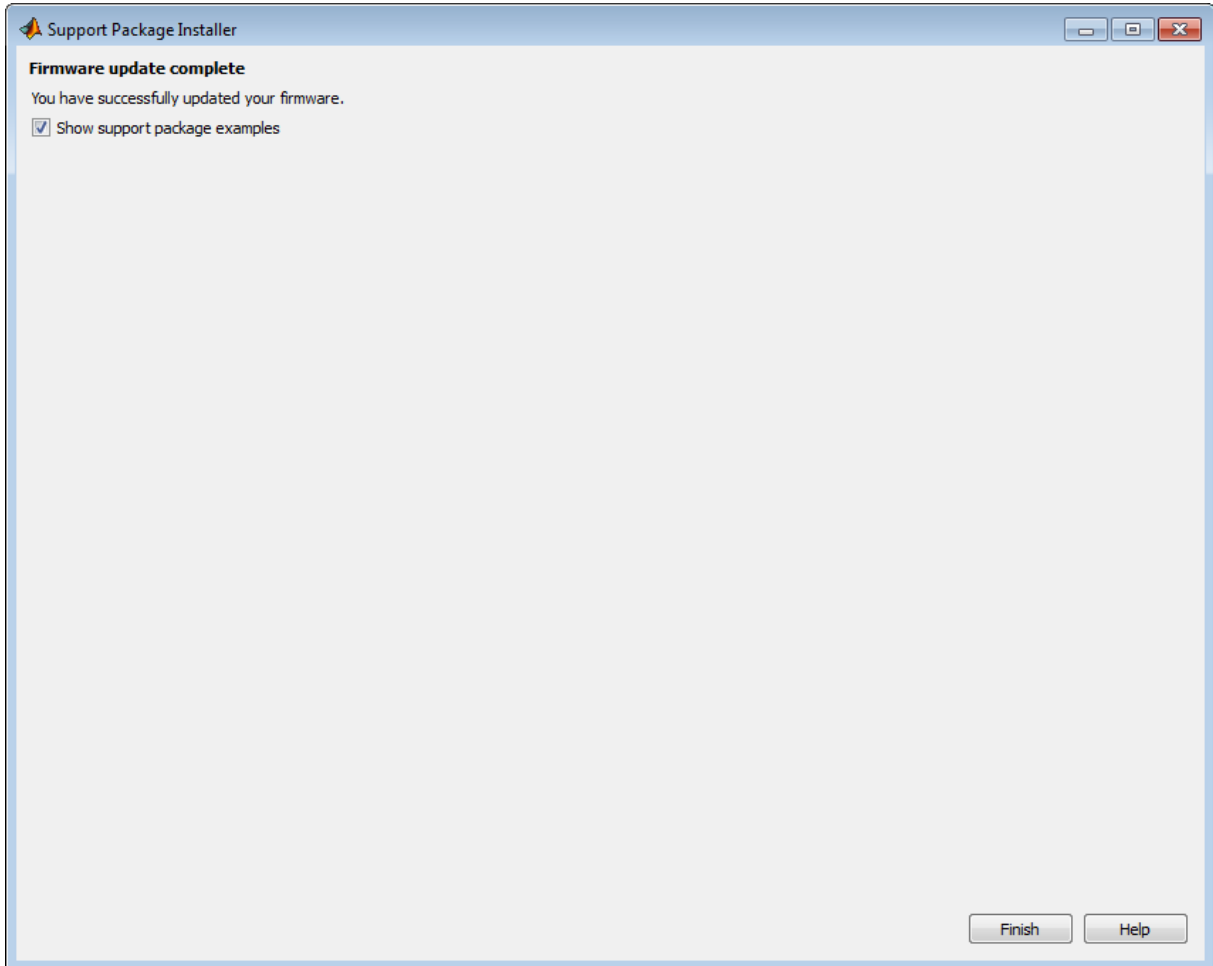
For advanced diagnostics, or to recover the IP address via a serial console, see:

- “Open a Serial Command-line Session with Gumstix® Overo® Hardware” on page 58-20
- “Get IP Address of Gumstix® Overo® Hardware” on page 58-29
- “Configure Network Settings on Gumstix® Overo® Hardware” on page 58-25



15 Click **Finish**. If you leave **Show support package examples** enabled, Support Package Installer opens the example page for Overo hardware.

To reopen these examples later on, see “Open Examples for This Support Package” on page 55-4.



Open a Serial Command-line Session with Gumstix Overo Hardware

This topic shows you how to configure and open a command-line session with the target hardware.

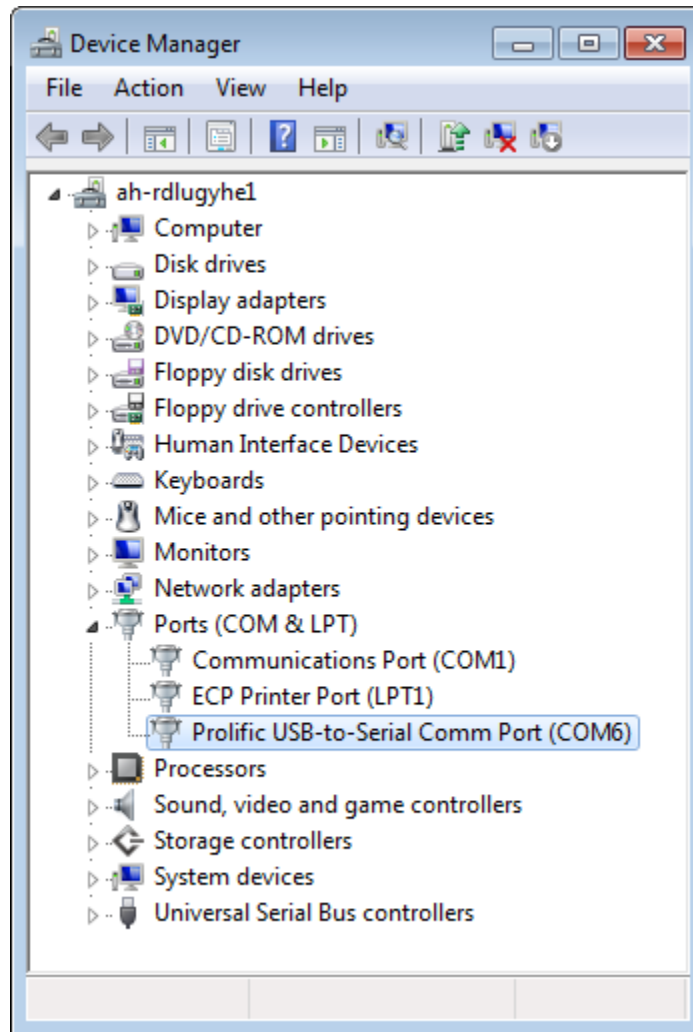
This is an optional task that you can use to:

- View the standard output while the board boots.
- Get or set the IP address, as described in “Get IP Address of Gumstix® Overo® Hardware” on page 58-29 topic.

To open a serial connection to your board:

- 1** Connect a USB cable from your host computer to the board.
- 2** Identify the COM port of your USB connection.
 - a** In Windows 7, use the “search programs and files” feature in the Start menu to find “Device Manager”.
 - b** Open Device Manager, and expand **Ports (COM & LPT)** to see the list of serial connections.

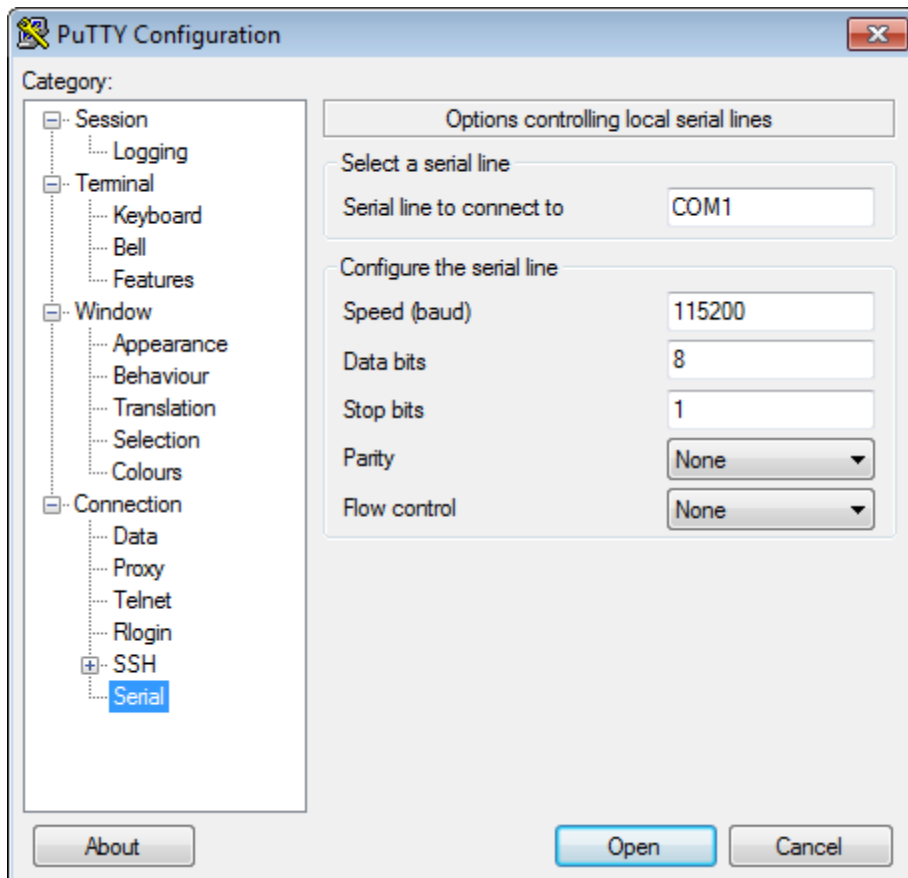
For example, the following image shows that the USB connection uses COM6.



3 In the MATLAB Command Window, start PuTTY by entering:

```
h = overo;  
h.openShell('serial');
```

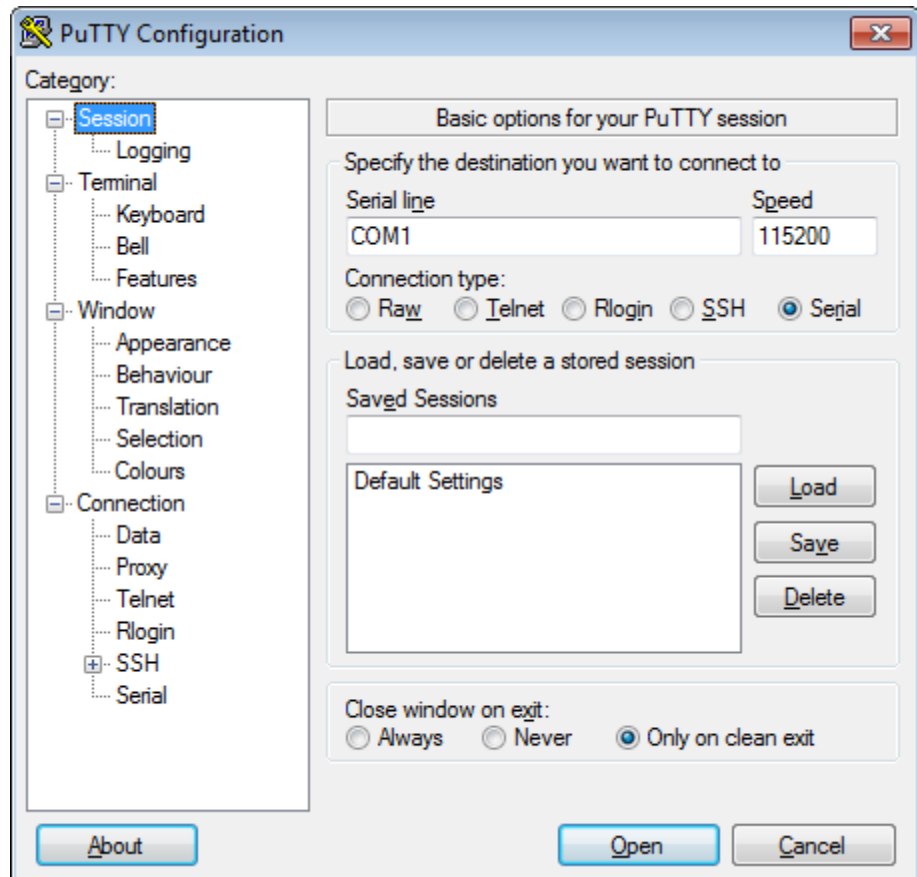
4 When the PuTTY dialog box opens, select the **Serial** category.



5 Set the following values:

- **Serial line to connect to:** Enter the COM port number of the USB connection.
- **Speed (baud):** 115200
- **Flow control:** None

6 In the PuTTY dialog box, select the **Session** category.



- 7 Set the **Connection type** parameter to **Serial**, enter a new name for **Saved Sessions**, click **Save**, and click **Open**.

Note Do not save the serial settings to **Default Settings**. In order to run your Simulink model on the target hardware, **Default Settings** must remain configured to use SSH. Otherwise, trying to run the model on target hardware produces an error message.

8 When a terminal window opens, press the **Enter** key on your keyboard. The terminal window displays the Linux command line.

9 In the PuTTY window, log in by entering root twice. For example:

```
overo-boardname login: root  
Password: root
```

10 When you are finished, enter `logout` on the Linux command line, and close PuTTY to end the serial connection.

See Also

- “Get IP Address of Gumstix® Overo® Hardware” on page 58-29
- For more information about PuTTY, see:
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

Configure Network Settings on Gumstix Overo Hardware

You can configure the IP settings of the Overo hardware by running Linux shell commands directly on the Overo hardware. Doing so allows the board to communicate over the network to which it is connected.

To inspect and reconfigure the IP settings on a board that already has the new firmware, follow the procedure in this section.

To configure the IP settings while you are replacing the firmware on your Overo hardware, see “Replace Firmware on Gumstix® Overo® Hardware” on page 58-3.

You may need to reconfigure the IP settings if your board:

- Has unknown IP settings
- Is unreachable using a network connection
- Is being moved to a network or direct Ethernet connection that uses static IP settings
- Is being moved from a network that used static IP settings to one that uses DHCP services

There are several conditions under which networks use DHCP or static IP settings:

- Use DHCP services — If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.
- Use static IP settings — If your board is directly connected to an Ethernet port on your computer or connected to an isolated network without DHCP services.

To configure the board to use DHCP or static IP settings:

- 1 Open a serial command-line session, as described in “Open a Serial Command-line Session with Gumstix® Overo® Hardware” on page 58-20.

Alternatively, you can use a terminal window after accessing the Linux desktop as described in:

- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 58-40
- “Access the Linux Desktop Remotely Using VNC” on page 58-41

2 Display the contents of the `/etc/network/interfaces` file. Enter:

```
cat /etc/network/interfaces
```

If the board is configured to use DHCP services (the default configuration), `dhcp` appears at the end of the following line:

```
iface eth0 inet dhcp
```

If the board is configured to use static IP settings, `static` appears at the end of the following line:

```
iface eth0 inet static
```

3 Create a backup of the `/etc/network/interfaces` file. Enter:

```
sudo cp /etc/network/interfaces /etc/network/interfaces.backup
```

If prompted, enter the root password (default: `root`).

4 Edit `interfaces` using a simple editor called `nano`. Enter:

```
sudo nano /etc/network/interfaces
```

5 Edit the last word of line that starts with `iface eth0 inet`.

To use DHCP services, change the line to:

```
iface eth0 inet dhcp
```

To use static IP settings, change the line to:

```
iface eth0 inet static
```

6 For static IP settings, add lines for address, netmask, and gateway. For example:

```
iface eth0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    gateway 192.168.1.1
```

For static IP settings:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- `netmask` to use the same network mask value, 255.255.255.0.
- `address` to an unused IP address, between 192.168.1.2 and 192.168.1.254.

7 Tell nano to exit and save the changes:

- a** Press **Ctrl+X**.
- b** Enter **Y** to save the modified buffer.
- c** For “File Name to Write: /etc/network/interfaces”, press **Enter**.
- d** The nano editor confirms that it “Wrote # lines” and returns control to the command line.

8 Reboot your board for the settings to take effect:

```
sudo shutdown -r now
```

9 Test the IP settings by logging in to the board over a telnet session.

Note You can use the `ifconfig` command to temporarily change the IP settings. Rebooting the board removes the `ifconfig` settings and restores the `/etc/network/interfaces` settings.

To change the IP settings temporarily, open a Linux command line. Enter `ifconfig`, the device id, a valid IP address, `netmask`, and the appropriate network mask. For example:

```
ifconfig eth0 192.168.45.12 netmask 255.255.255.0
```

Related Examples

- “Run Model on Gumstix® Overo® Hardware” on page 58-33
- “Get IP Address of Gumstix® Overo® Hardware” on page 58-29

Get IP Address of Gumstix Overo Hardware

You can get the IP address of the Overo hardware using the Linux command line.

This is an optional task that you can use to:

- Run a model on a different board from the previous one you used. (Update the **Host name** parameter as described in “Run Model on Gumstix® Overo® Hardware” on page 58-33.).
- Open a telnet or SSH session with a board whose IP settings you do not know.
- Configure the UDP Send block.

To get your board’s IP address using a command-line session:

- 1 Open a serial command-line session on your board, as described in “Open a Serial Command-line Session with Gumstix® Overo® Hardware” on page 58-20.

Alternatively, you can use a terminal window after accessing the Linux desktop as described in:

- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 58-40
- “Access the Linux Desktop Remotely Using VNC” on page 58-41

- 2 At the Linux command line, enter `ifconfig`.

- 3 Locate the `eth0` device, and get the value of `inet addr` in the command-line output. For example, `172.28.144.36` in the following output:

```
root@overo-ah-rd1ugyhe1:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:15:C9:28:F2:FE
          inet addr:172.28.144.106  Bcast:172.28.144.255  Mask:255.255.255.0
          inet6 addr: fe80::215:c9ff:fe28:f2fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14379 errors:0 dropped:0 overruns:0 frame:0
          TX packets:200 errors:0 dropped:0 overruns:0 carrier:0
```

```
collisions:0 txqueuelen:1000  
RX bytes:1368932 (1.3 MiB) TX bytes:25329 (24.7 KiB)  
Interrupt:80
```

Related Examples

- “Open a Serial Command-line Session with Gumstix® Overo® Hardware” on page 58-20
- “Run Model on Gumstix® Overo® Hardware” on page 58-33
- “Get IP Address of Gumstix® Overo® Hardware” on page 58-29

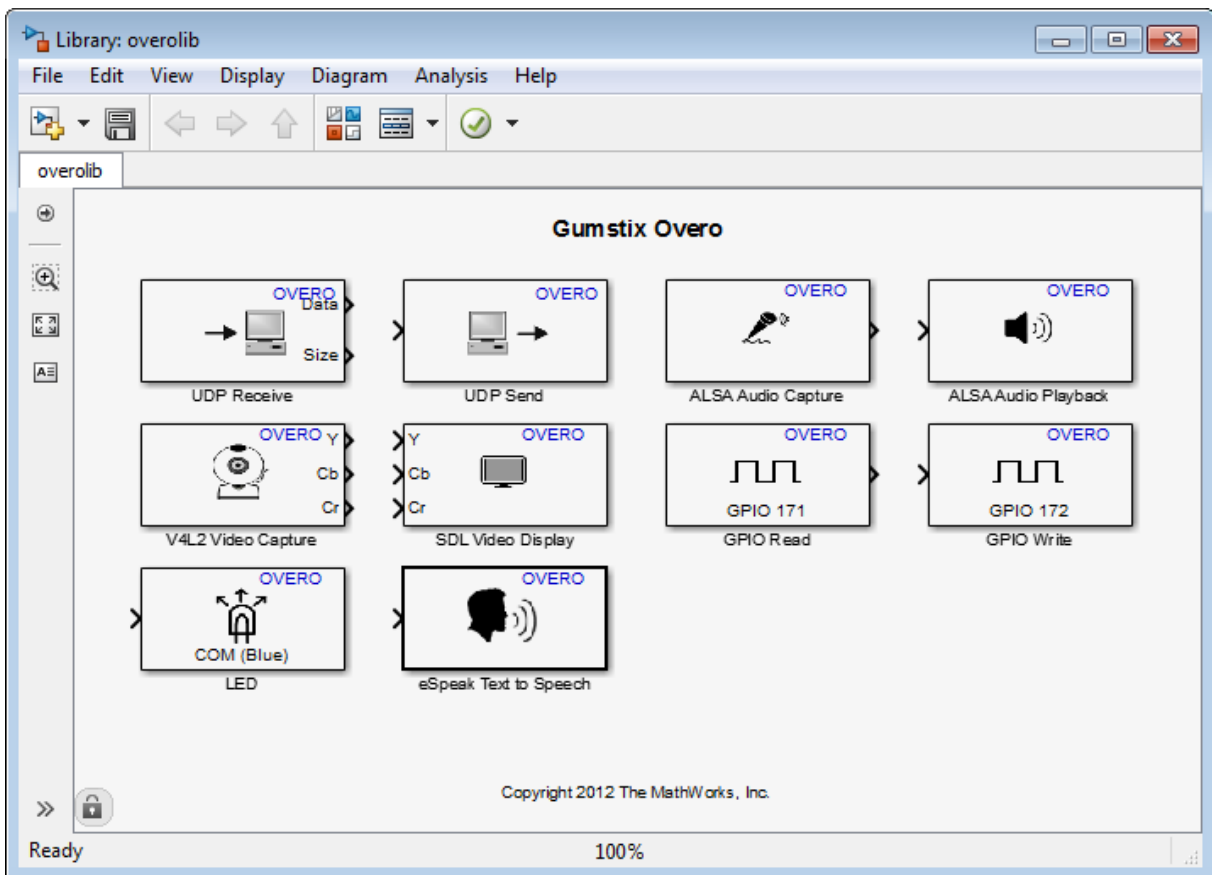
Open Block Library for Gumstix Overo Hardware

This block library provides support for protocols and APIs available through Overo hardware.

To open the block library from the MATLAB Command Window, enter:

```
overolib
```

The following block library opens.

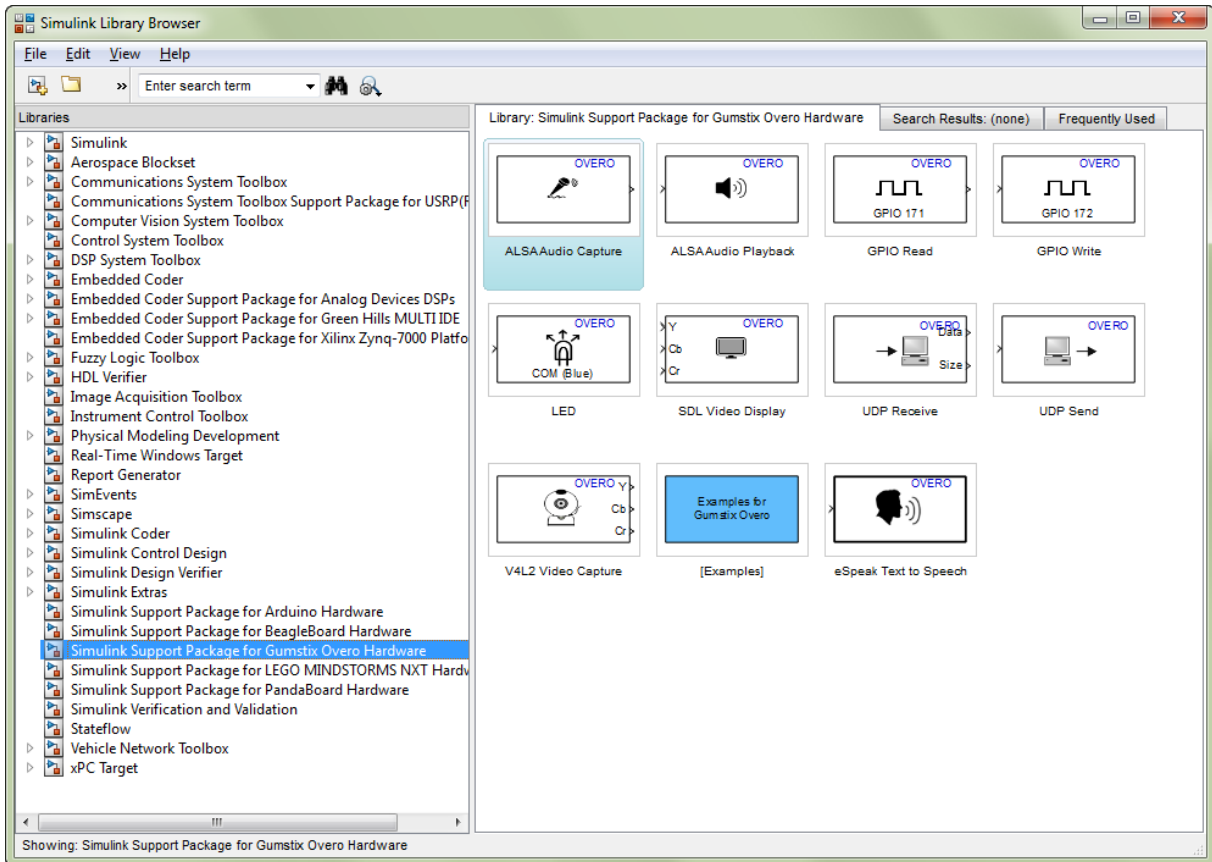


To open the block library from the Simulink Library Browser:

- 1 Enter `simulink` in the MATLAB Command Window, or click the following icon on the MATLAB toolbar.



- 2 When the Simulink Library Browser opens, click **Simulink Support Package for Gumstix Overo Hardware**.



Run Model on Gumstix Overo Hardware

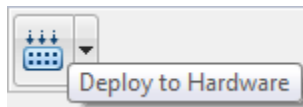
This example shows how to prepare, configure, and run a simple model on your Overo hardware.

Before starting this procedure:

- Connect your board to the network and to a power supply.
- Create or open a Simulink model.

To prepare and run the model:

- 1** Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 2** In your model, select **Tools > Run on Target Hardware > Prepare to Run**. This action changes the model Configuration Parameters.
- 3** When the Run on Target Hardware opens, set the **Target hardware** parameter to Gumstix Overo.
- 4** If you have changed boards since the last time you updated the firmware or ran a model, update the **Host name**, **User name**, and **Password** parameters.
- 5** Click the Deploy to Hardware button.



This action automatically downloads and runs your model on the board.

Note Pressing RESET or cycling the power on the Overo hardware during this step can cause the ssh utility to hang.

Running a new or updated model on the board:

- Automatically stops a running model with the same name.
- Does not stop running models that have other names.

Caution Avoid using the RESET button to stop a running model and reboot the board. Doing so can corrupt operating system and program files. To fix corrupt files, replace the firmware as described in “Replace Firmware on Gumstix® Overo® Hardware” on page 58-3.

Note If you have the Embedded Coder or Simulink Coder software, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

See Also

- “Create the Simple Model”
- “Configure Network Settings on Gumstix® Overo® Hardware” on page 58-25
- “Tune and Monitor Model Running on Gumstix® Overo® Hardware” on page 58-36
- “Overview of Model Referencing” on page 6-2

Stop or Restart a Model Running on Gumstix Overo Hardware

Caution Avoid using the RESET button to stop a running model and reboot the board. Doing so can corrupt operating system and program files. To fix corrupt files, replace the firmware as described in “Replace Firmware on Gumstix® Overo® Hardware” on page 58-3.

To stop a model running on the board, enter the following commands in the MATLAB Command Window:

```
h=overo;  
h.stop('modelName');
```

For example, to stop the sumdiff model, enter:

```
h=overo;  
h.stop('sumdiff');
```

You can also restart a model that was previously running on the board, or to run multiple instances of a model. To do so, enter the following commands in the MATLAB Command Window:

```
h=overo;  
h.run('modelName');
```

Note You do not need to enter `h=overo;` multiple times if a previous instance of `h` is available in the MATLAB Workspace.

For example, to restart the sumdiff model you stopped in the previous example, enter:

```
h.run('sumdiff');
```

Tune and Monitor Model Running on Gumstix Overo Hardware

In this section...

“Overview of Using External Mode” on page 58-36

“Run Your Simulink Model in External Mode” on page 58-37

“Stop External Mode” on page 58-38

Overview of Using External Mode

You can use External mode to tune parameters and monitor a model running on your target hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on model results in real time. Doing so helps you find the optimal values to achieve desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to rerun the model each time you change parameters. You can also use External mode to develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

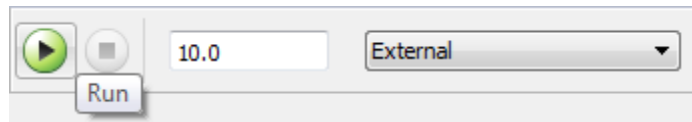
This workflow lists the tasks usually required to tune parameters with External mode:

- 1** In the model on your host computer, enable External mode.
- 2** Optional: Place one or more sink blocks in your model. For example, use Display or Scope blocks to visualize data, or use a To File block to log signal data.
- 3** Give Simulink software the command to run the model on the target hardware.
- 4** Optional: Observe data External mode sends from the target hardware to sink blocks in the model on the host computer.

- 5 Change and apply parameter values in the model on your host computer.
- 6 Find the optimal parameter values by making adjustments and observing the results.
- 7 Save the new parameter values, disable External mode, and save the model.

Run Your Simulink Model in External Mode

- 1 Create a network connection between the Overo hardware and your host computer. See “Configure Network Settings on Gumstix® Overo® Hardware” on page 58-25.
- 2 On the model toolbar, set **Simulation mode** to External mode.



- 3 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 4 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\sl_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

- 5 Change parameter values in the model on your host computer.

Observe the corresponding changes in the model running on the hardware.

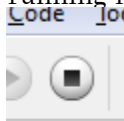
If your model contains blocks from the Simulink Sinks block library, the sink blocks in the model on your host computer display the values generated by the model running on the hardware.

Note External mode increases the processing burden of the model running on the board. If the software reports an overrun, disable External mode.

Note If you have the Embedded Coder or Simulink Coder software, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

Stop External Mode

To stop the model running in External mode, click the Stop button located on



the model toolbar.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

Detect and Fix Task Overruns on Gumstix Overo Hardware

You can configure a model running on the target hardware to detect and notify you of task overruns.

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. A task overrun occurs when a task is scheduled to execute before a previous instance of the same task has completed.

To enable overrun detection:

- 1** In your model, click **Tools > Run on Target Hardware > Options**.
- 2** When the Run on Target Hardware opens, select the **Enable overrun detection** check box.
- 3** Click **OK**.

When a task overrun occurs, the command prompt on the host machine repeatedly displays an “Overrun” error message until the model stops. For example, a message might read “Overrun — rate for subrate task 1 is too fast”,

When an overrun condition occurs, you can correct it using one or both of the following approaches:

- Simplify the model.
- Increase the sample times for the model and the blocks in it. For example, change the **Sample time** parameter in all of your data source blocks, such as blocks for input devices, from 0.1 to 0.2.

Note External mode increases the processing burden of the model running on your board. If the software reports an overrun, disable External mode.

Access the Linux Desktop Directly Using Desktop Computer Peripherals

You can access the Linux desktop on the Overo hardware using hardware peripherals such as a monitor, keyboard, and mouse.

If you do not have hardware peripherals, see “Access the Linux Desktop Remotely Using VNC” on page 58-41 instead.

To connect the Overo hardware to your hardware peripherals:

- 1** Connect a computer monitor to the HDMI or LCD video output on the Overo hardware.
- 2** Connect a powered USB hub to the USB port on the Overo hardware.
- 3** Connect a mouse and keyboard to the powered USB hub.
- 4** Connect computer speakers to either of the two 3.5 mm audio connectors on the Overo hardware.
- 5** Connect a network cable to the Ethernet port.
- 6** Connect the Overo hardware to the power supply.
- 7** Log in to the Linux desktop using the username and password. The default user name and password is root.

Access the Linux Desktop Remotely Using VNC

You can remotely access the Linux desktop on the Overo hardware from another computer. This capability uses a client-server graphical desktop sharing system called Virtual Network Computing (VNC).

The Overo firmware distributed by MathWorks has a built-in VNC server called *x11vnc*. To remotely interact with the Overo hardware, you can run a VNC client on a remote computer, open a VNC connection with *x11vnc* on the Overo hardware.

To open a VNC session from MATLAB:

- 1** Create a physical network connection between your host computer and the Overo hardware.
- 2** In the MATLAB Command Window, create a connection to the Overo hardware by entering:

```
h = overo
```

The command line responds with information about the Overo hardware. For example:

```
h =
```

```
LinuxServices with properties:
```

```
HostName: '172.28.144.106'  
UserName: 'root'  
Password: 'root'  
BuildDir: '/home/root'
```

Enable the VNC server on the Overo hardware by entering:

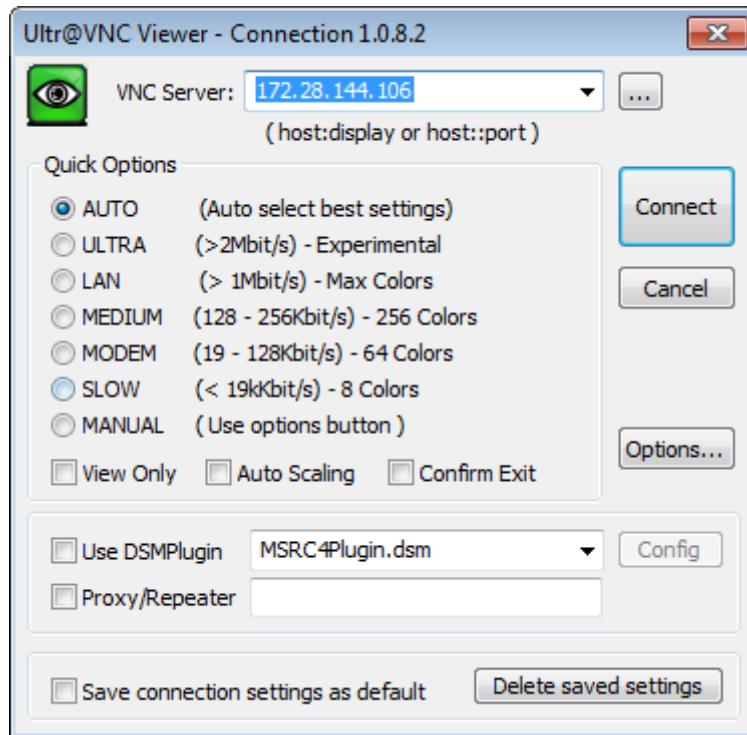
```
h.execute('export DISPLAY=:0.0; xhost+; x11vnc -bg')
```

When the VNC server is running, the command line displays:

```
ans =
```

0

- 3 Open a VNC viewer on your host system, and connect to the VNC server on the Overo hardware.



- 4 Log in to the operating system. The default user name and password is root.

LEGO MINDSTORMS NXT Hardware

- “Install Support for LEGO MINDSTORMS NXT Hardware” on page 59-2
- “Replace Firmware on NXT Brick” on page 59-3
- “Open Block Library for LEGO MINDSTORMS NXT Hardware” on page 59-7
- “Run Model on NXT Brick” on page 59-11
- “Tune Parameters and Monitor Data in a Model Running on NXT Brick” on page 59-13
- “Detect and Fix Task Overruns on NXT Brick” on page 59-19
- “Exchange Data Between Two NXT Bricks” on page 59-20
- “Set Up A Bluetooth Connection” on page 59-29

Install Support for LEGO MINDSTORMS NXT Hardware

This example shows how to add support for LEGO MINDSTORMS NXT hardware to the Simulink product. After you complete this process, you can run Simulink models on your LEGO MINDSTORMS NXT hardware.

To install support for LEGO MINDSTORMS NXT hardware:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2 Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

This process downloads and installs:

- Third-party software development tools on your host computer.
- A Simulink block library called **Simulink Support Package for LEGO MINDSTORMS NXT Hardware**.
- Examples

For convenience, this document occasionally refers to LEGO MINDSTORMS NXT hardware as an “NXT brick” or as “target hardware”.

To check for updates, repeat this process when a new version of MATLAB software is released. You can also check for updates between releases.

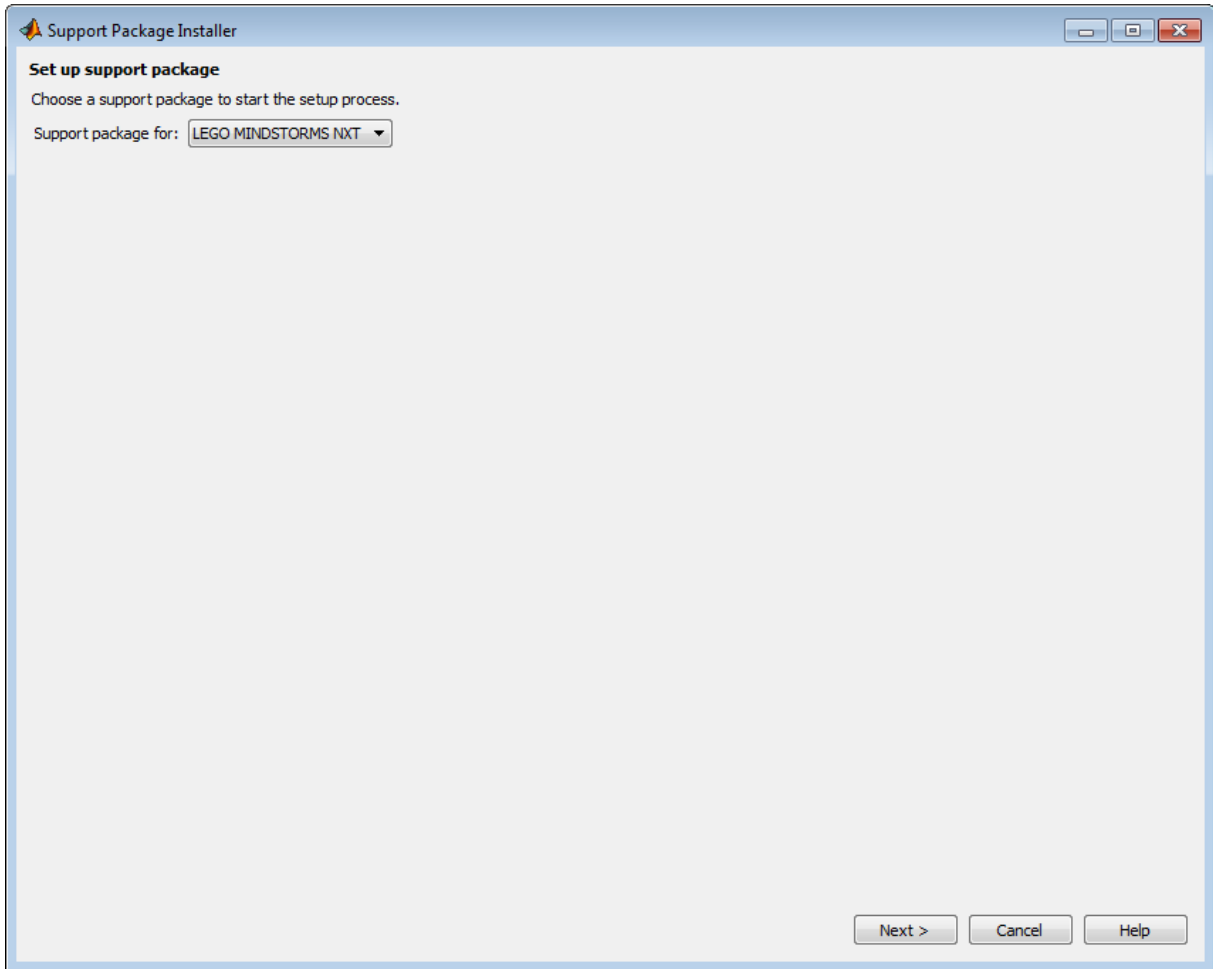
Replace Firmware on NXT Brick

This example shows how to replace the firmware on the NXT brick with enhanced firmware that can run Simulink models. The enhanced firmware can run

When you complete this process, and install the support package for LEGO MINDSTORMS NXT hardware, you can run a Simulink model on NXT brick.

Warning This process completely erases the memory on the NXT brick.

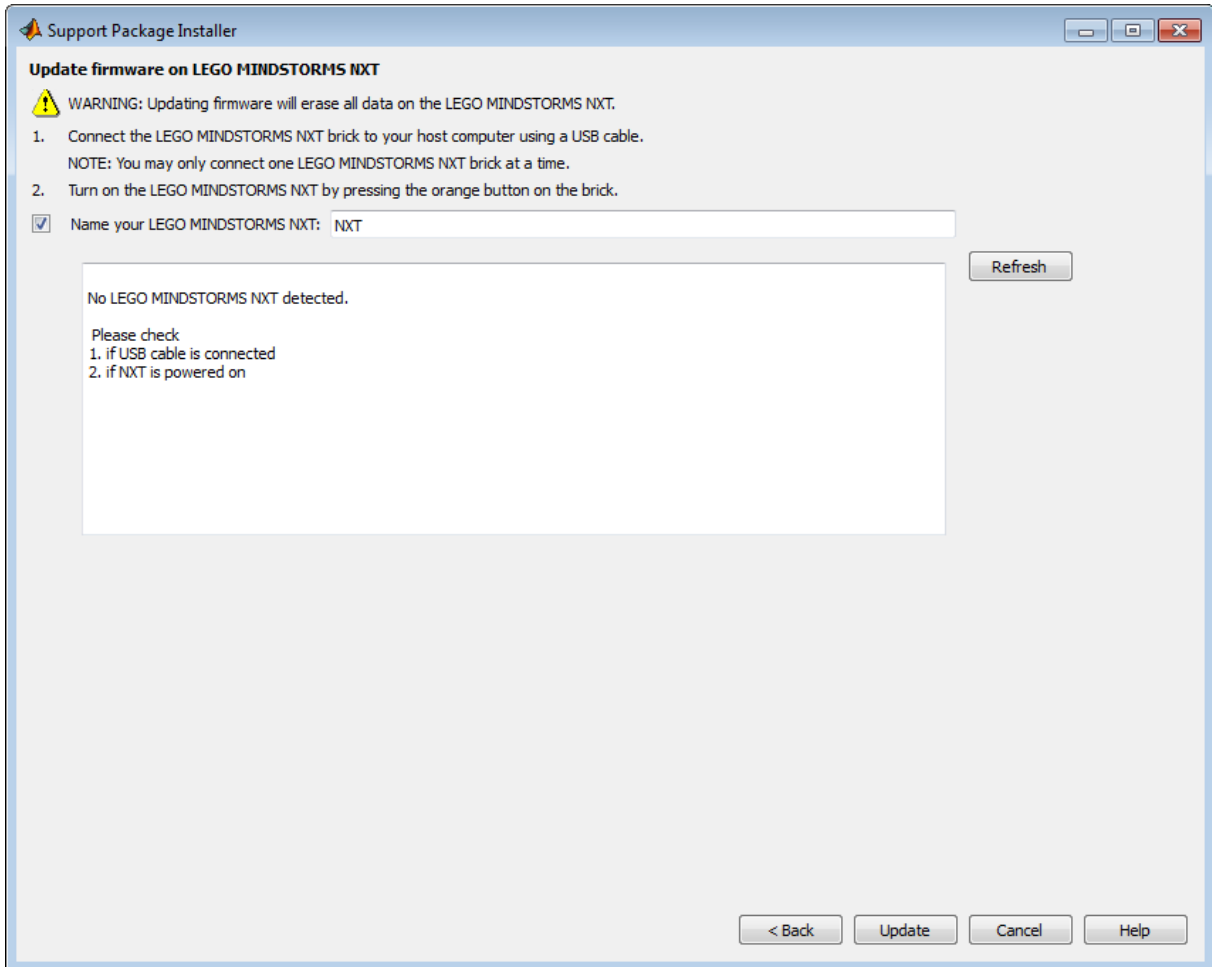
- 1** Open the **Update firmware** screen in Support Package Installer using one of the following methods:
 - Click **Continue >** in the **Install/update complete** screen of Support Package Installer.
 - In a model, select **Tools > Run on Target Hardware > Update firmware**.
 - In a MATLAB Command Window, enter `targetupdater`.
- 2** Choose LEGO MINDSTORMS NXT and click **Next**.



3 Connect the NXT brick to your host computer with a USB cable.

Turn the NXT brick on by pressing its orange button.

Enter a unique name for the **Name your LEGO MINDSTORMS NXT** parameter, and click **Update**.

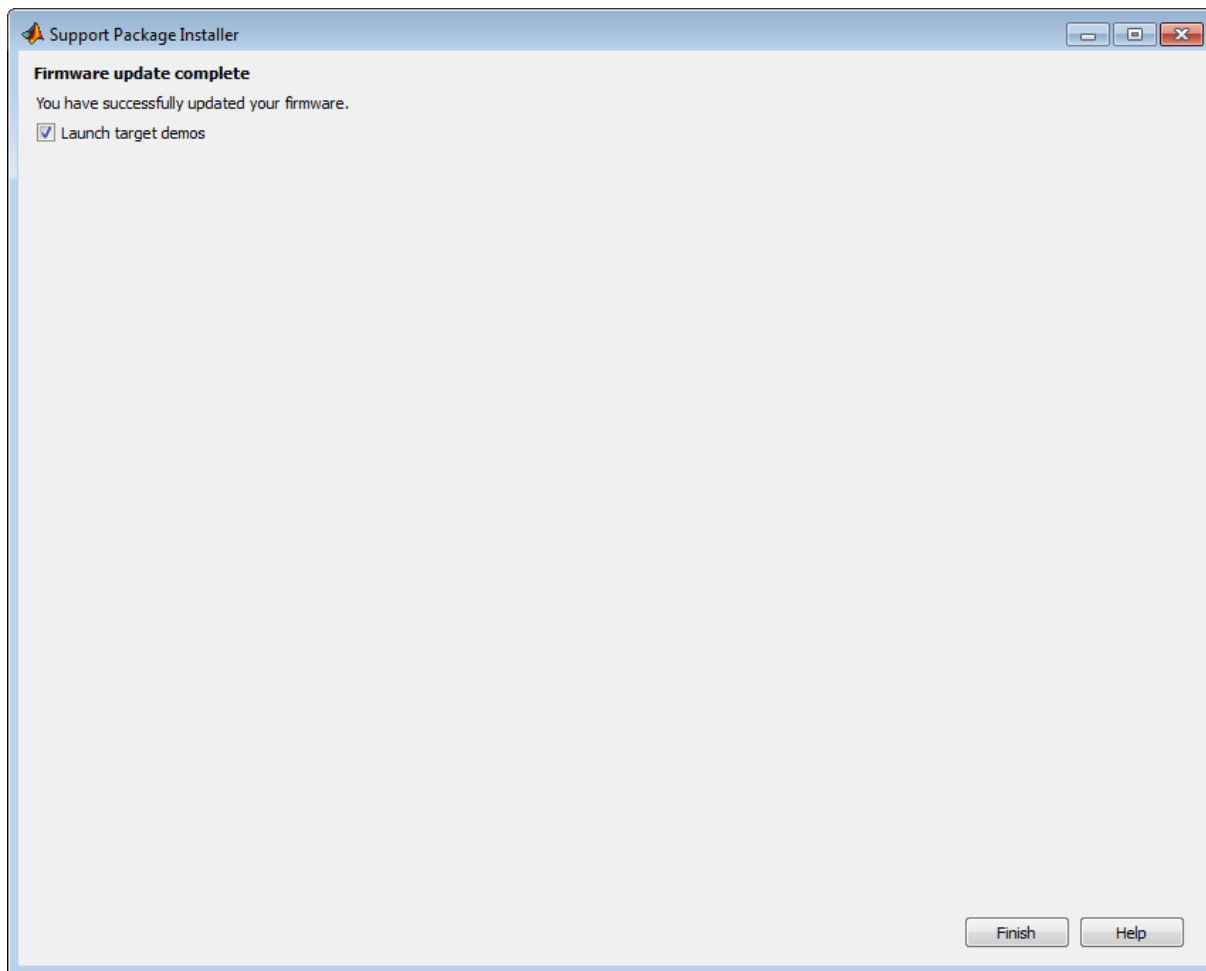


During the firmware update, the NXT brick emits a faint ticking sound and the LCD goes blank. The USB connection to the host computer can disconnect and reconnect multiple times.

When the firmware process is complete, the NXT brick displays the default screen including My Files.

4 Click **Finish** and review the examples.

To reopen these examples later on, see “Open Examples for This Support Package” on page 55-4.



Open Block Library for LEGO MINDSTORMS NXT Hardware

The block library for LEGO MINDSTORMS NXT hardware is a collection of blocks that provides drivers for LEGO MINDSTORMS NXT sensors and hardware.

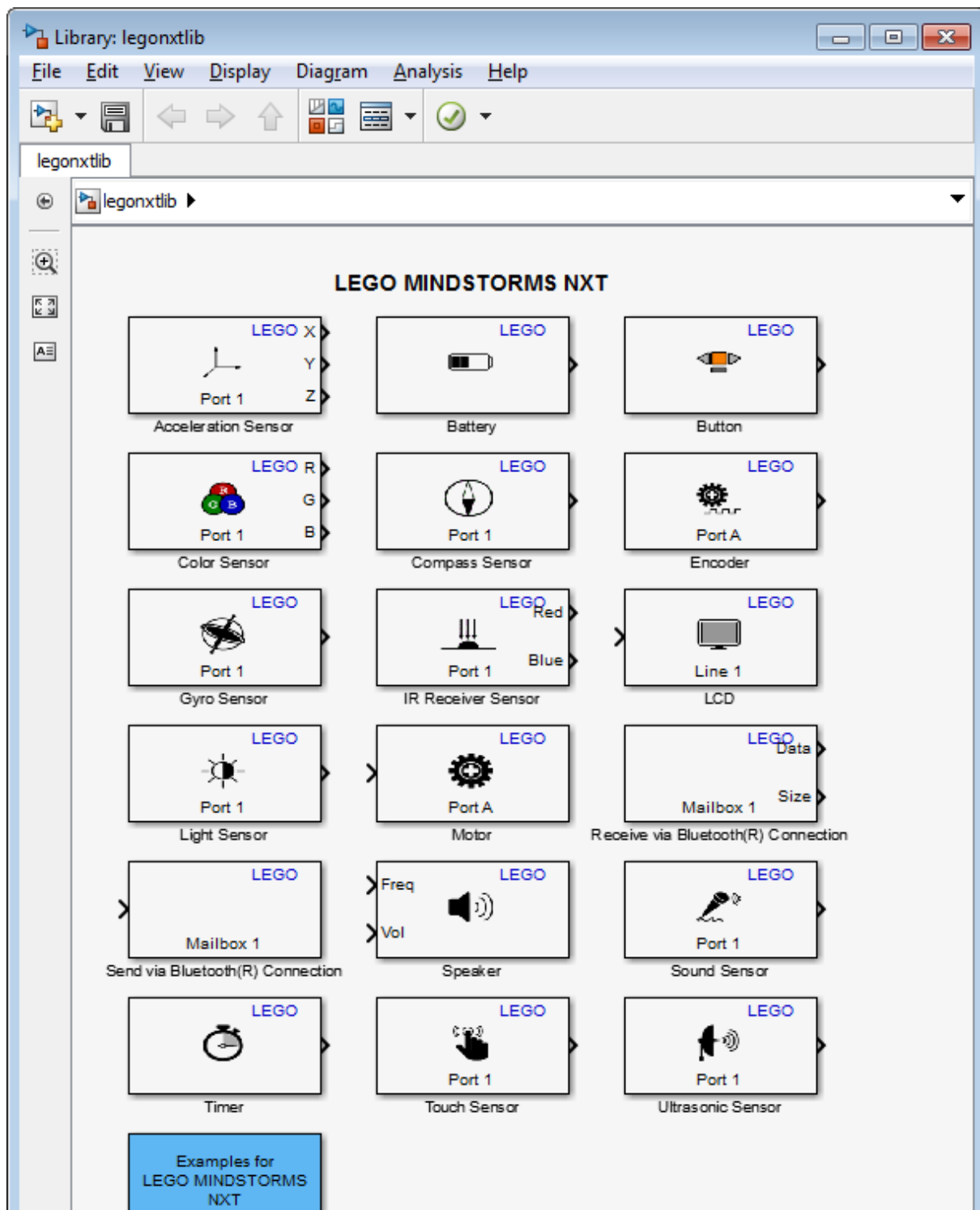
There are two ways to open the block library for LEGO MINDSTORMS NXT hardware:

- The MATLAB Command Window. This approach is quick, but doesn't display the other Simulink block libraries that are available.
- The Simulink Library Browser. This approach takes a few more steps, but displays all of the Simulink block libraries that are available.

1 In the MATLAB Command Window, enter:

```
legonxtlib
```

The block library for LEGO MINDSTORMS NXT hardware opens.



2 In the MATLAB Command Window, enter:

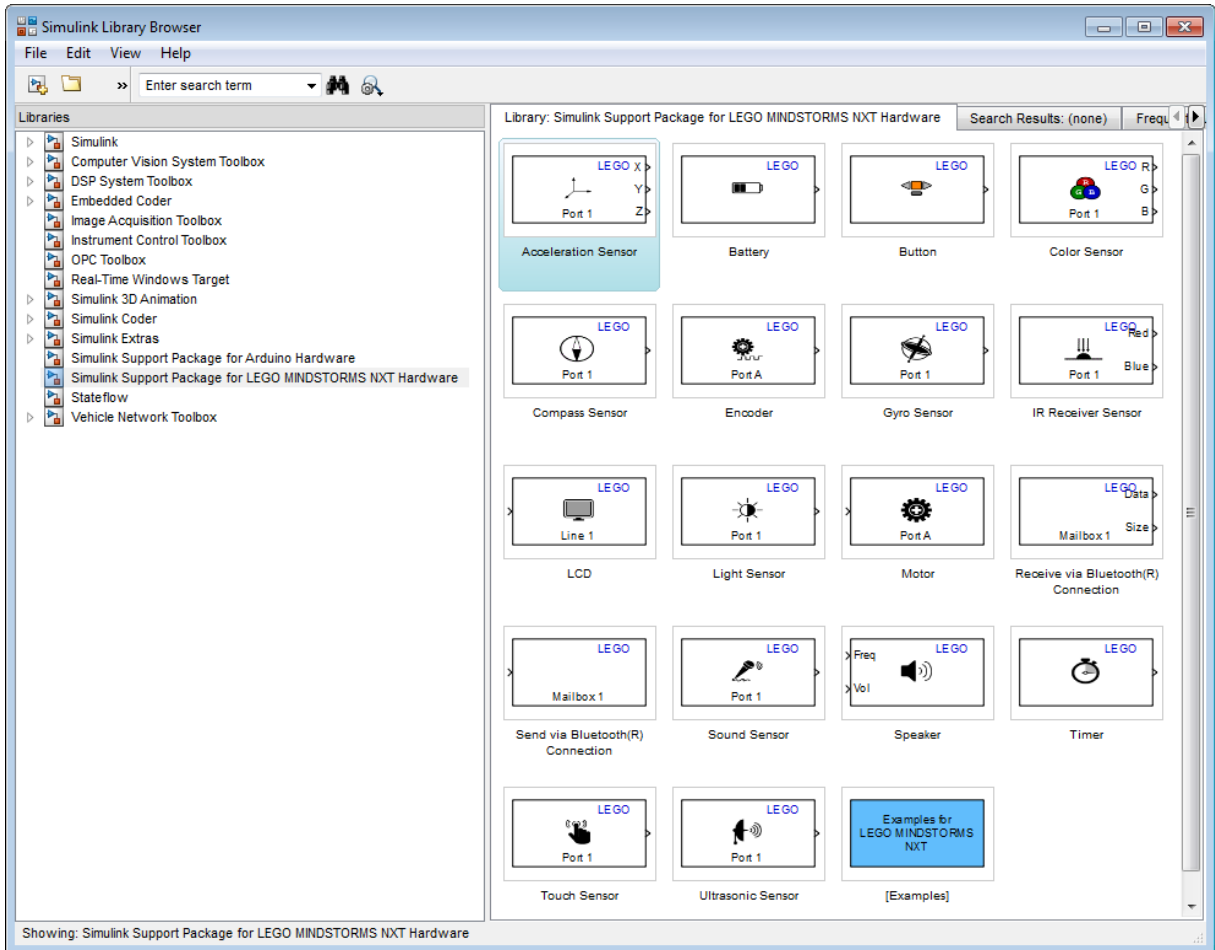
```
simulink
```

Or, in the MATLAB Toolbar, click the following icon.



Simulink Library Browser opens.

3 In the Simulink Library Browser, click **Simulink Support Package for LEGO MINDSTORMS NXT Hardware**.



Run Model on NXT Brick

This example shows how to configure and run a model on NXT brick.

Before starting this procedure, create or open a Simulink model.

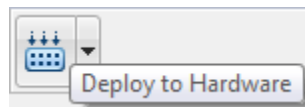
- 1 Connect your NXT brick to the host computer using a USB cable or a Bluetooth® device. Simulink software uses this connection to download the model to the NXT hardware when you run the model.

For more information about using a Bluetooth device, see “Set Up A Bluetooth Connection” on page 59-29.

- 2 Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 3 In your model, select **Tools > Run on Target Hardware > Prepare to Run**. This action changes the model Configuration Parameters.
- 4 In the Run on Target Hardware pane that opens, set the **Target hardware** parameter to LEGO MINDSTORMS NXT.
- 5 Set the **Connection type** parameter to USB connection or Bluetooth connection.
- 6 Turn the NXT brick on by pressing the orange button. The LCD displays LEGO and MINDSTORMS splash screens, and then **My Files**.

Note The NXT brick automatically powers down after several minutes of inactivity. If this happens, turn the power back on.

- 7 Click the Deploy to Hardware button.



This action automatically downloads and runs your model on the NXT brick.

When the NXT hardware starts running the model, the LCD displays “I am running”, or it displays the input values and labels of any LCD blocks in the model.

Note To restart a model while it is running, press the left-arrow button on the NXT brick.

To restart a model that is stopped, use the orange button on the NXT brick to select My Files > Software Files, and then select your model.

To stop the model, press the grey button on the NXT brick.

Tune Parameters and Monitor Data in a Model Running on NXT Brick

In this section...

“About External Mode” on page 59-13

“Run Your Simulink Model in External Mode” on page 59-14

“Stop External Mode” on page 59-15

“Set COM Port Number Manually” on page 59-15

About External Mode

This topic explains how the *External mode* feature enables you to tune parameters and monitor data in a model running on your target hardware.

For example, in a model, if you select **Simulation > Mode > External mode** and then select **Simulation > Run**, Simulink software embeds lightweight client/server code in your model and then runs your model on the target hardware. If you tune parameters in the model on your host computer, External mode updates the corresponding parameter values in the model running on the target hardware. You can use sink blocks, such as Display or Scope blocks, to monitor data from the model running on the target hardware. External mode sends data from the sink blocks on the target hardware to the corresponding sink blocks on your host computer, where you can view or log the data.

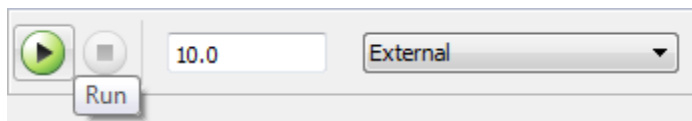
External mode accelerates parameter tuning. When you develop an algorithm, you must typically try a range of parameter values in order to find the optimal settings. Without External mode, you must re-run the model on the target hardware to evaluate the effects of each change. With External mode, you can adjust parameter values, monitor data, and evaluate the results until you find the optimal value, without re-running the model.

External mode increases the processing burden of the model running on the NXT brick. If the model running on the hardware reports an overrun, you can disable External mode or fix the issue as described in “Detect and Fix Task Overruns on NXT Brick” on page 59-19.

Run Your Simulink Model in External Mode

Before running your model in External mode, set up a Bluetooth connection between the NXT hardware and your Windows computer, as described in “Set Up A Bluetooth Connection” on page 59-29

- 1 On the model toolbar, set **Simulation mode** to External mode.



- 2 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 3 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
  realtime\realtime\+realtime\extModeAutoConnect.p>
  extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
  realtime\realtime\sl_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

- 4 While the model is running in External mode, you can change parameter values in the model on your host computer and observe the corresponding changes in the model running on the hardware.

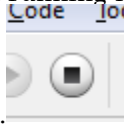
If your model contains blocks from the Simulink Sinks block library, the sink blocks in the model on your host computer display the values generated by the model running on the hardware.

Note External mode has low priority on the NXT brick. Under heavy loads, External mode sends data to the host computer less frequently.

Note If you have the Embedded Coder or Simulink Coder products, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

Stop External Mode

To stop the model running in External mode, click the Stop button located on



the model toolbar.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

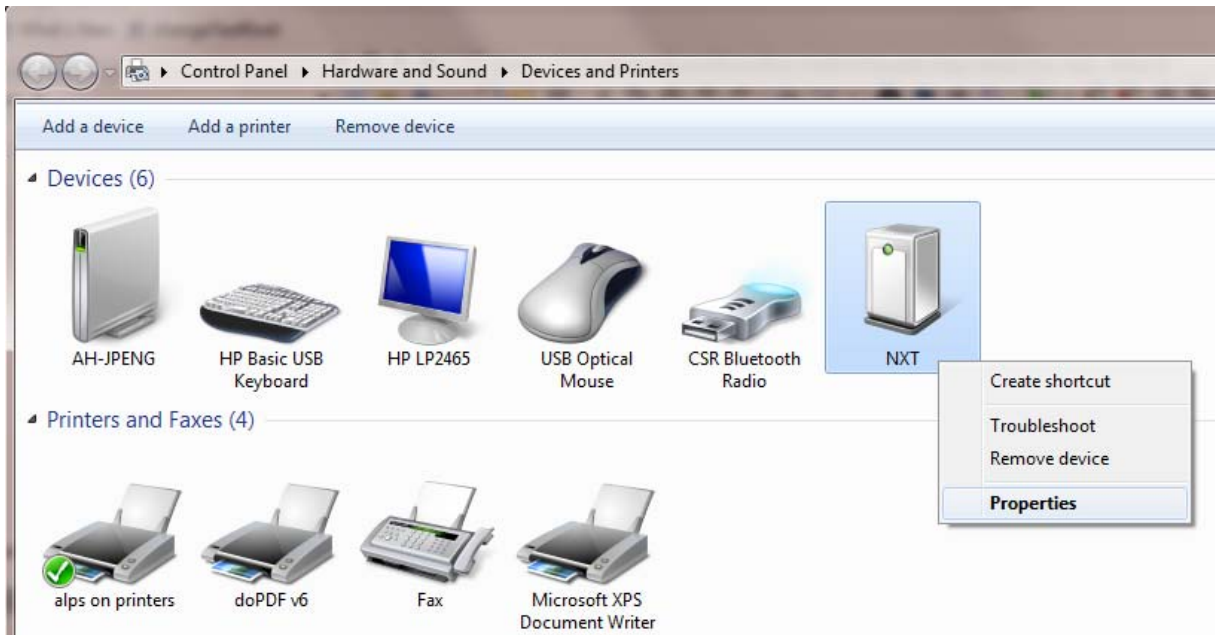
Set COM Port Number Manually

By default, Run on Target Hardware automatically gets the COM port number of the NXT brick when you use External mode. However, you have the option of setting the COM port number manually.

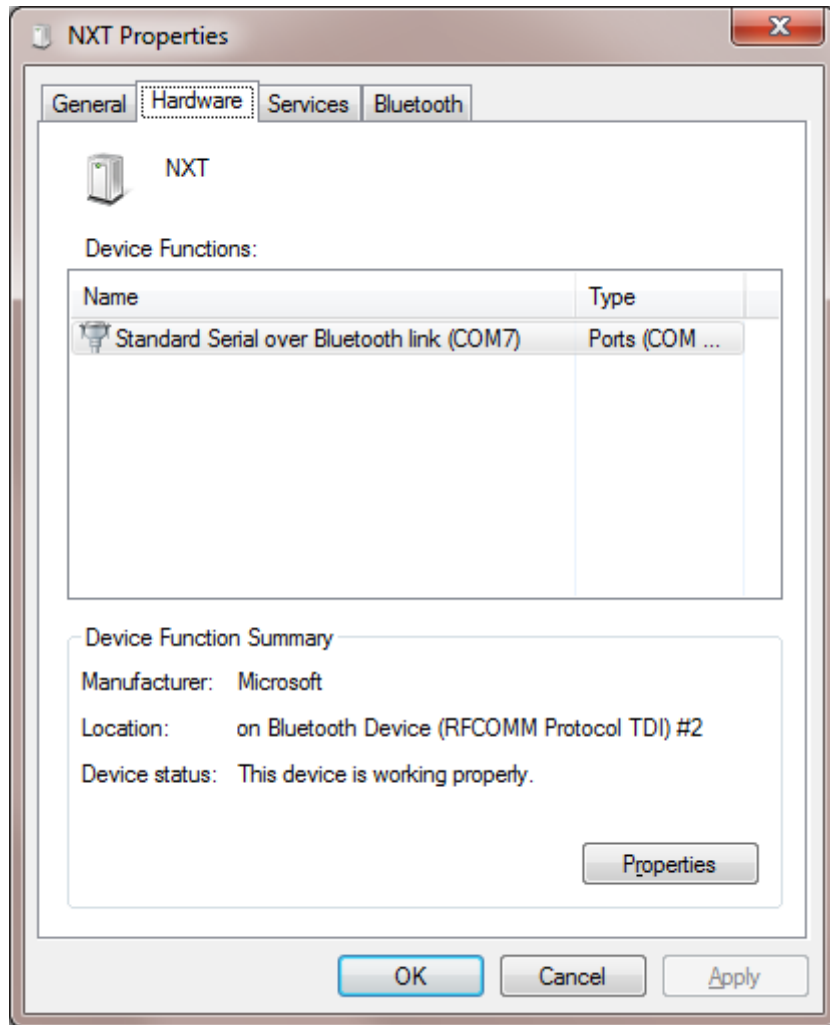
- 1 In Windows, locate the NXT brick under **Control Panel > Hardware and Sound > Devices and Printers**.

The NXT brick uses the name you assigned to it during the firmware update process. By default, that name is “NXT”. If you changed that name during the firmware update, the Windows device label shows that new name.

- 2 Right-click the NTX brick and select **Properties**.



- 3 In the Properties dialog box, select the Hardware tab and make a note of the COM port number.



- 4 In your model, select the **Tools > Run on Target Hardware > Options** menu item.
- 5 In the Run on Target Hardware pane that opens, change the **Set host COM port** parameter to Manually.

- 6** In the **COM port number** parameter, enter the COM port number you noted earlier.

Detect and Fix Task Overruns on NXT Brick

You can configure a model running on the target hardware to detect and notify you of when task overrun occurs.

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. When a task is required to perform extra processing and takes longer than normal to execute, it may be scheduled to execute before a previous instance of the same task has completed. The result is a task overrun.

To enable overrun detection:

- 1** In your model, click **Tools, Run on Target Hardware** and **Options**.
- 2** In the Run on Target Hardware pane that opens, select the **Enable overrun detection** check box.
- 3** Click **OK**.

When a task overrun occurs, the LCD on the NXT brick displays an “Overrun!” error message until you stop the model.

To fix an overrun condition:

- Simplify the model.
- Increase the sample times for the model and the blocks in it. For example, change the **Sample time** parameter in all of your data sources, such as the sensor blocks, from 0.1 to 0.2.

Note External mode increases the processing burden of the model running on the NXT brick. If the software reports an overrun, disable External mode.

Exchange Data Between Two NXT Bricks

In this section...

“Tips for Use Bluetooth Blocks” on page 59-20

“Add Bluetooth Blocks to your Models” on page 59-20

“Configure Bluetooth Slave” on page 59-24

“Configure Bluetooth Master” on page 59-26

This topic explains how to exchange data between two separate NXT Intelligent Bricks (“NXT bricks”) over a Bluetooth connection.

If, instead, you are looking for information about exchanging data between your host computer and an NXT brick, see “Tune Parameters and Monitor Data in a Model Running on NXT Brick” on page 59-13.

Tips for Use Bluetooth Blocks

Here are some tips for using the Send via Bluetooth Connection block and Receive via Bluetooth Connection block:

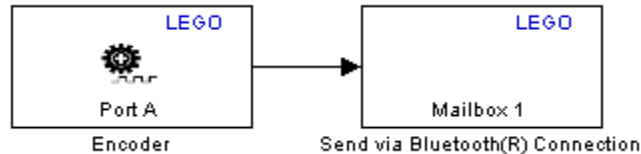
- Use two separate models, one for each NXT brick.
- You can only use the Bluetooth blocks for connecting two NXT bricks.
- For each Send via Bluetooth Connection block in one model, use a corresponding Receive via Bluetooth Connection block in the other model. Set the **Mailbox** parameter in both blocks to the same number.
- For each Receive via Bluetooth Connection block in a model, use a unique Mailbox number.
- A model can contain no more than five Receive via Bluetooth Connection blocks.

Add Bluetooth Blocks to your Models

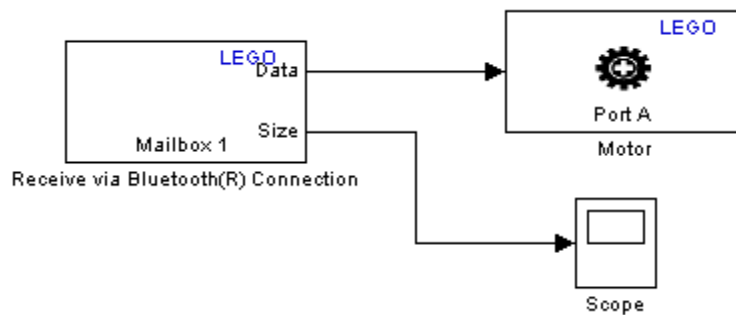
Add Bluetooth blocks to your models and configure them:

- 1 Open `legonxtlib`.

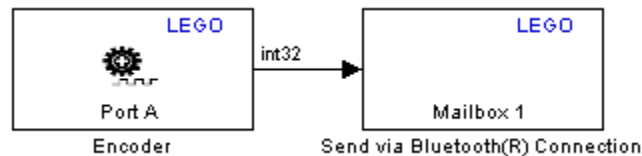
- 2 Open two models. One for each NXT brick.
- 3 Add a Send via Bluetooth Connection block to a model, and connect it to the data source, as shown in the following example.



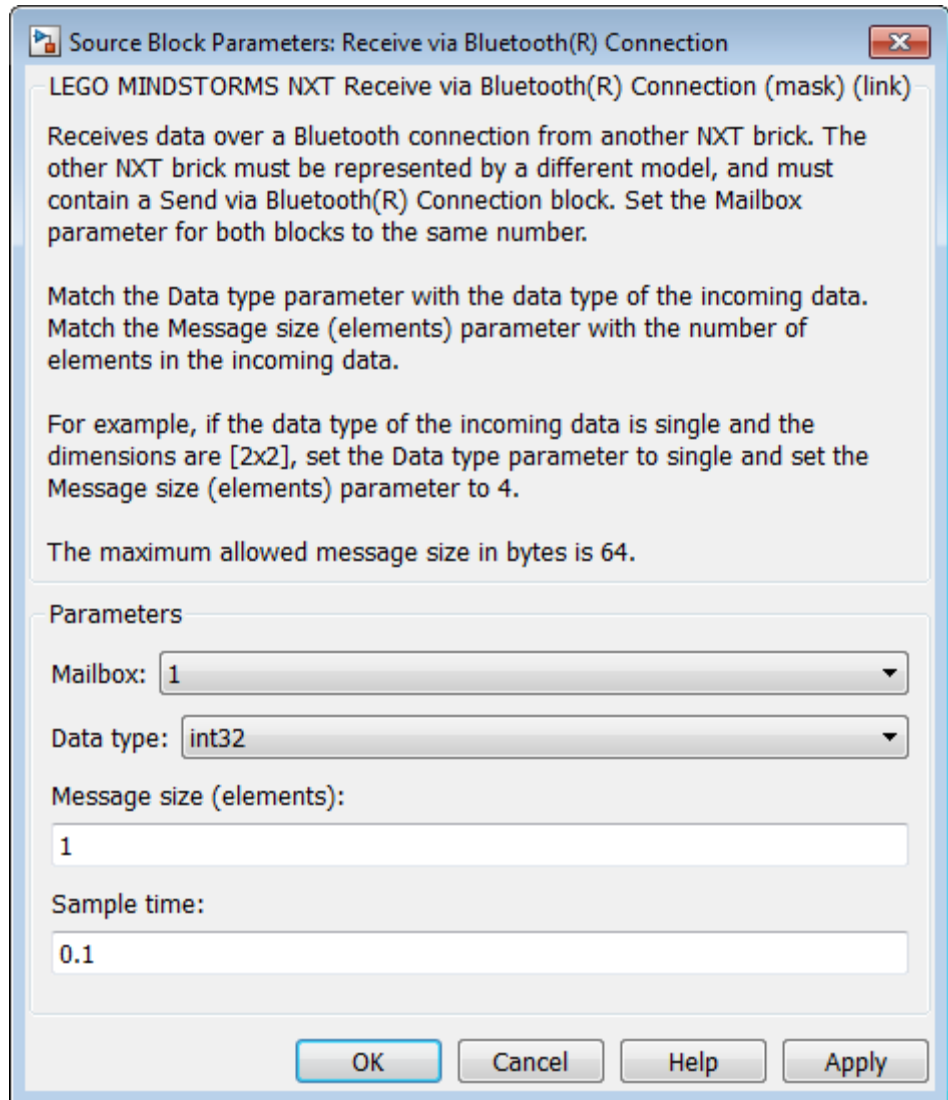
- 4 Add a Receive via Bluetooth Connection block to the other model, and connect it to the data sinks, as shown in the following example.



- 5 In the model that contains the Send via Bluetooth Connection block, select the **Display > Signals & Ports > Port Data Types** menu item. Then, select **Simulation > Update Diagram Ctrl+D**. Observe the data type that appears above the connector attached to the Send via Bluetooth Connection block. For example “int32” in the following example.



- 6** Open the Receive via Bluetooth Connection block in the other model. Apply the data type from the previous step to the **Data type** parameter in the Receive via Bluetooth Connection block.
- 7** In the model that contains the Send via Bluetooth Connection block, select the **Display > Signals & Ports > Signal Dimensions** menu item. Then, select **Simulation > Update Diagram Ctrl+D**.
 - If the signal dimensions are greater than 1, the connector attached to the Send via Bluetooth Connection block displays the signal dimension. Multiply the signal dimensions to get the value you will apply to the **Message size** parameter. For example, if the signal dimensions are 2x10, the set **Message size** to 10.
 - If the signal is one-dimensional, the connector attached to the Send via Bluetooth Connection block does not display a signal dimension. For a one-dimensional signal, set **Message size** to 1.
- 8** In the model that contains the Receive via Bluetooth Connection block, apply the signal dimension from the previous step to the **Signal dimension** parameter in the Receive via Bluetooth Connection block.



- 9 Save all of your changes and leave the models open.

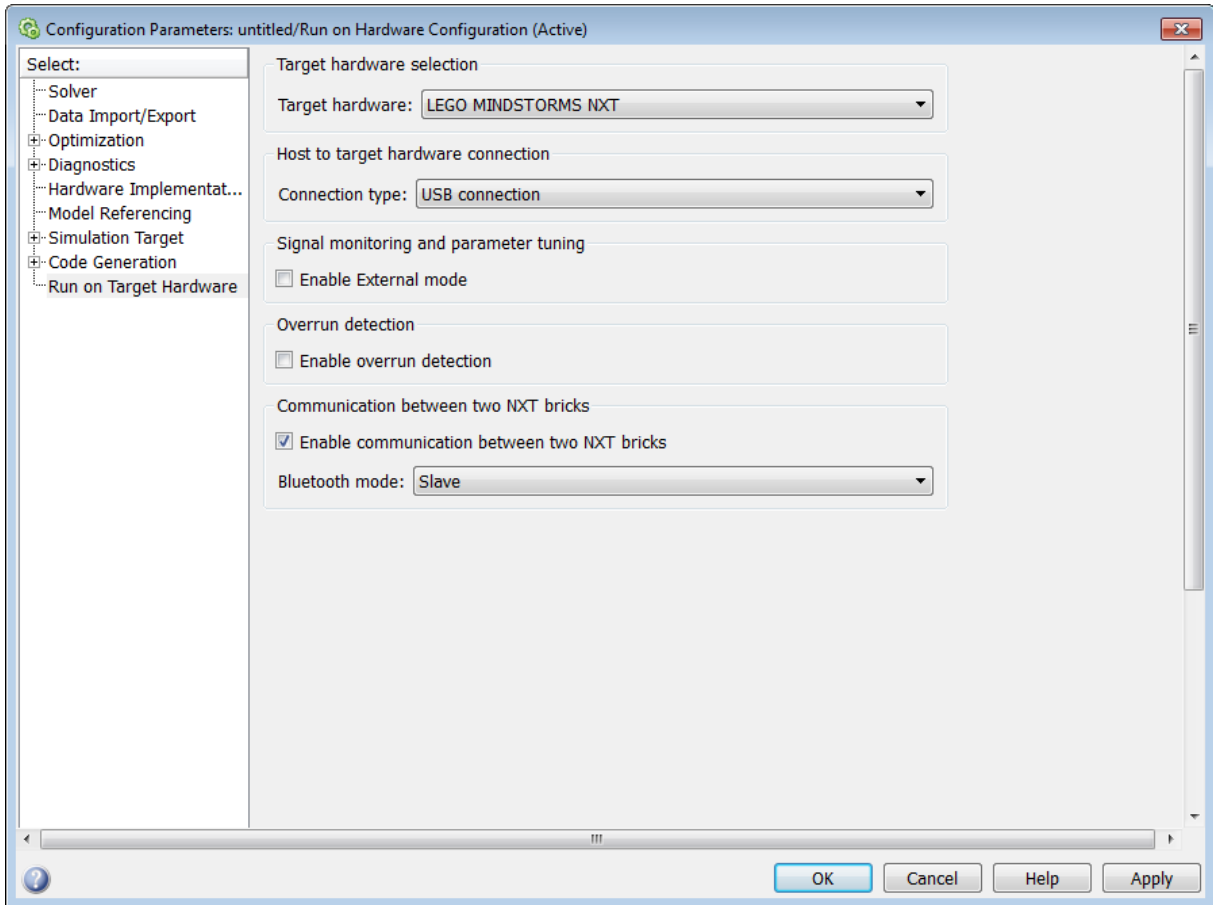
Configure Bluetooth Slave

Using your models, configure the NXT brick to operate as a slave Bluetooth node. You can make this choice arbitrarily, without regard to which device is sending or receiving data.

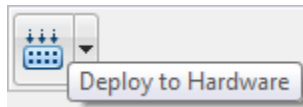
To configure an NXT brick as a slave Bluetooth node:

- 1** Open one model and select **Tools > Run on Target Hardware > Options**.
- 2** In the Run on Target Hardware pane that opens, select the **Enable communication between two NXT bricks** check box, and set **Bluetooth mode** to Slave.

Note This action disables the **External mode** parameter. Enabling communication between two NXT bricks requires exclusive use of the Bluetooth device on the NXT brick.



- 3 Click **OK**. This saves and closes the Configuration Parameters.
- 4 In your model, click the Deploy to Hardware button to run your model on the NXT brick.



This process can take a couple of minutes. When the download completes, the NXT brick automatically runs the application.

- 5 When Simulink software finishes downloading the application to the slave NXT brick, the LCD displays “Waiting”.

Observe the number on the second line of the LCD. The first 12 digits of this number are the *Slave Bluetooth address*, which you will use as you configure the Bluetooth master in the next procedure.

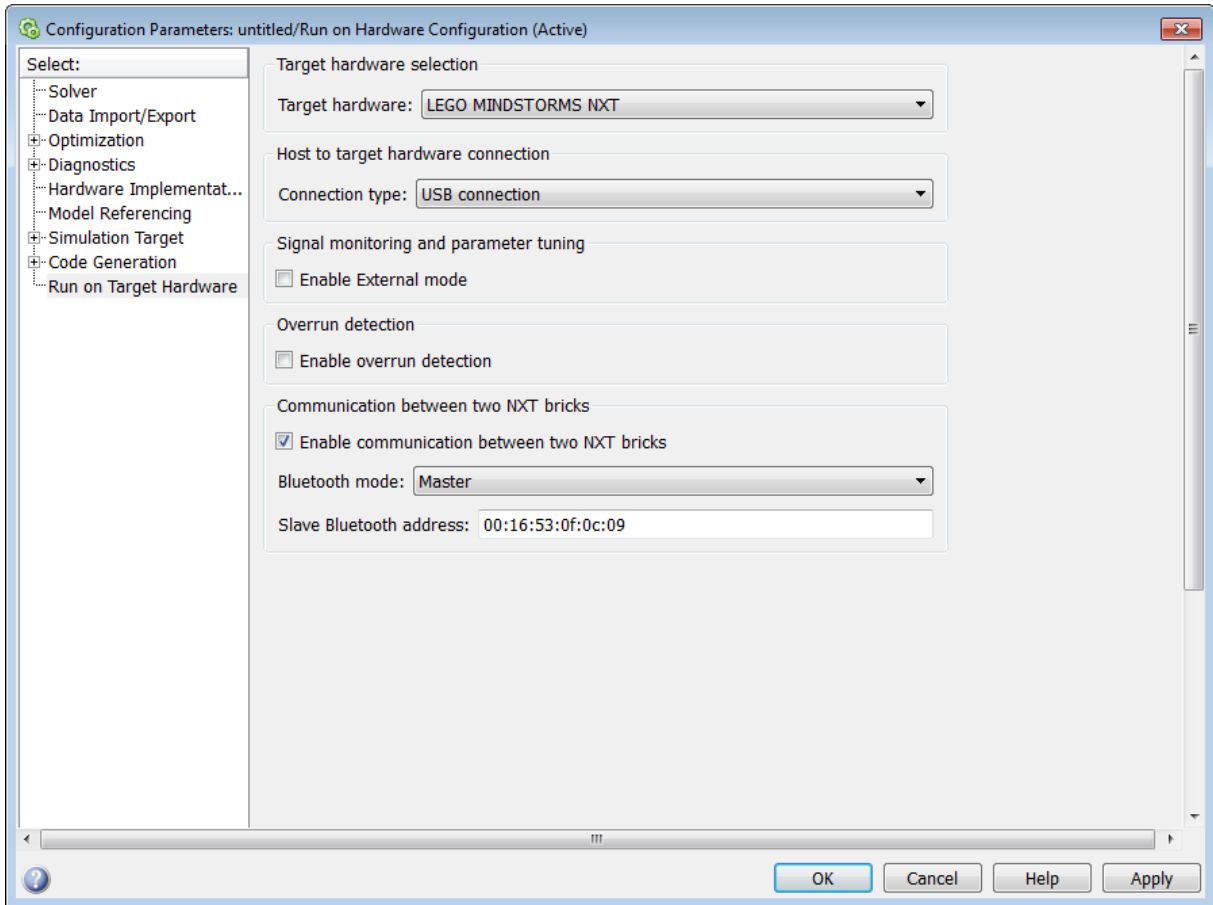
Configure Bluetooth Master

Configure the other NXT brick as a master Bluetooth node:

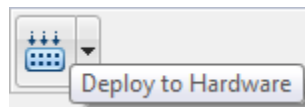
- 1 Open one model and select **Tools > Run on Target Hardware > Options**.
- 2 In the Run on Target Hardware pane that opens, select the **Enable communication between two NXT bricks** check box, and set **Bluetooth mode** to Master.

Note This action disables the **External mode** parameter. Enabling communication between two NXT bricks requires exclusive use of the Bluetooth device on the NXT brick.

- 3 Enter the 12-digit **Slave Bluetooth address**, inserting a colon between each pair of digits. For example, if the slave NXT brick displays 00165312926900, enter 00:16:53:12:92:69. Disregard the last two digits, 00.



- 4 Click **OK**. This saves and closes the Configuration Parameters.
- 5 In your model, click the Deploy to Hardware button to run your model on the NXT brick.



This process can take a couple of minutes. When the download completes, the NXT brick automatically runs the application and connects to the Bluetooth slave.

- 6 On the Bluetooth master, turn the wheel back and forth. Observe the direction and speed of the wheel on the Bluetooth slave.

Set Up A Bluetooth Connection

This topic shows you how to set up a Bluetooth connection between your Windows host computer and the NXT hardware.

You may need to create a Bluetooth connection between your Windows host computer and the NXT hardware for either of these purposes:

- To enable External mode, as described in “Tune Parameters and Monitor Data in a Model Running on NXT Brick” on page 59-13.
- To download the model to the NXT hardware. You do so by setting the **Connection type** to **Bluetooth connection**, as described in “Run Model on NXT Brick” on page 59-11.

This procedure requires a Bluetooth device on your host computer that is compatible with the NXT's built-in Bluetooth device. Details for adding a Bluetooth device vary from one version of Windows to another. If needed, search Microsoft help for additional information.

Note Some Bluetooth devices do not support connections with the NXT brick. For this reason, MathWorks recommends using the 9847 Bluetooth Dongle specified on the LEGO MINDSTORMS NXT Web site.

Set up a Bluetooth on your Windows host:

- 1** Connect a supported Bluetooth dongle to your Windows host computer. Let Windows automatically find and install the appropriate drivers.
- 2** Turn the NXT brick on, and verify that the small Bluetooth icon is visible in the upper-left corner of the LCD. The icon looks like a capital-B composed of two triangles.

If the Bluetooth symbol is not visible, press the right arrow on the NXT brick four times. Then, select **Bluetooth > On/Off > On**.

- 3** In Windows 7, open **Control Panel > Devices and Printers** to check whether the NXT brick has already been added to Devices.

- 4 If the NXT brick is not among the devices, click **Add a device**.
- 5 In the **Add a device** wizard that opens, select your NXT Bluetooth device, and click **Next**.

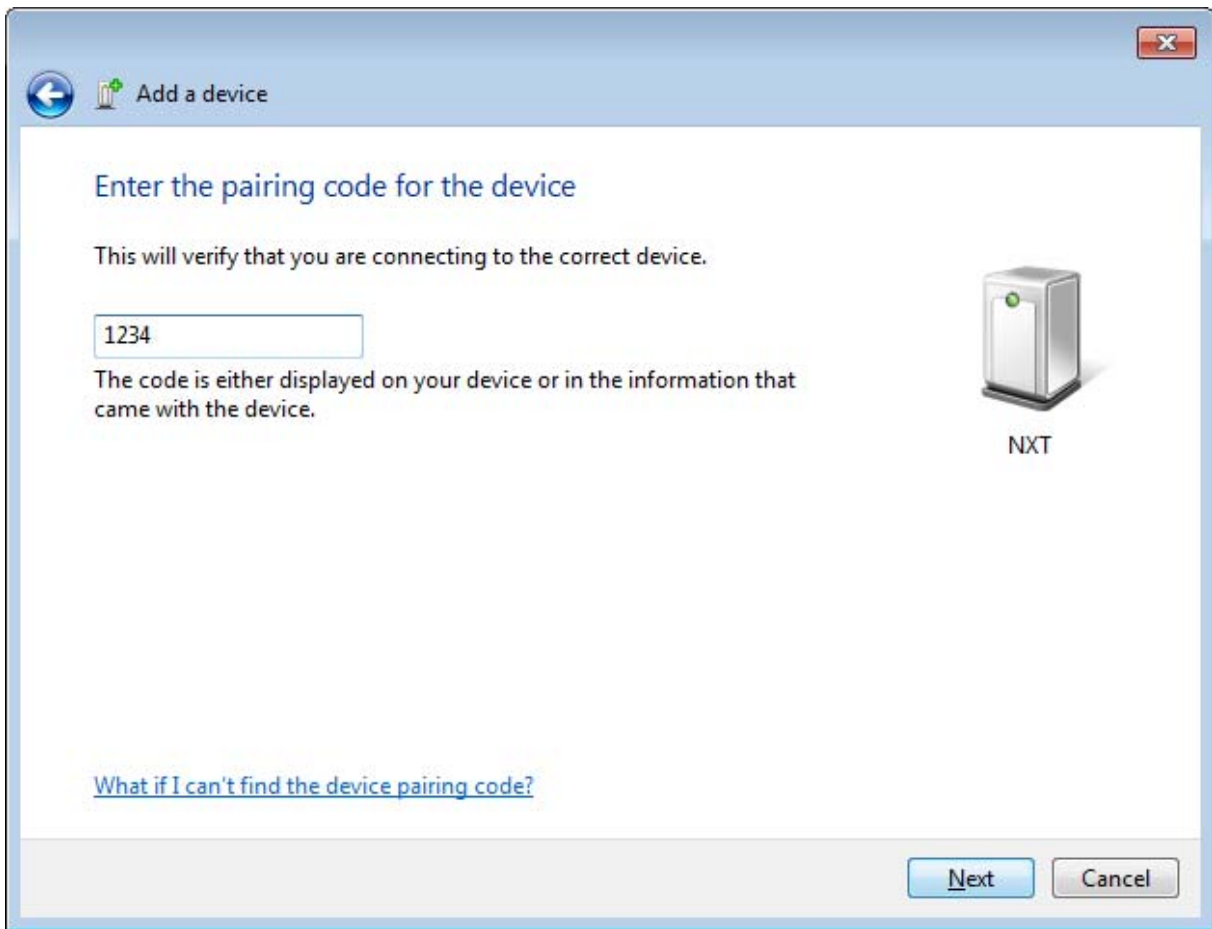


Note All of the NXT bricks operating within Bluetooth range appear in the list of devices. To avoid confusion, assign a unique name to your NXT brick, as described in “Replace Firmware on NXT Brick” on page 59-3.

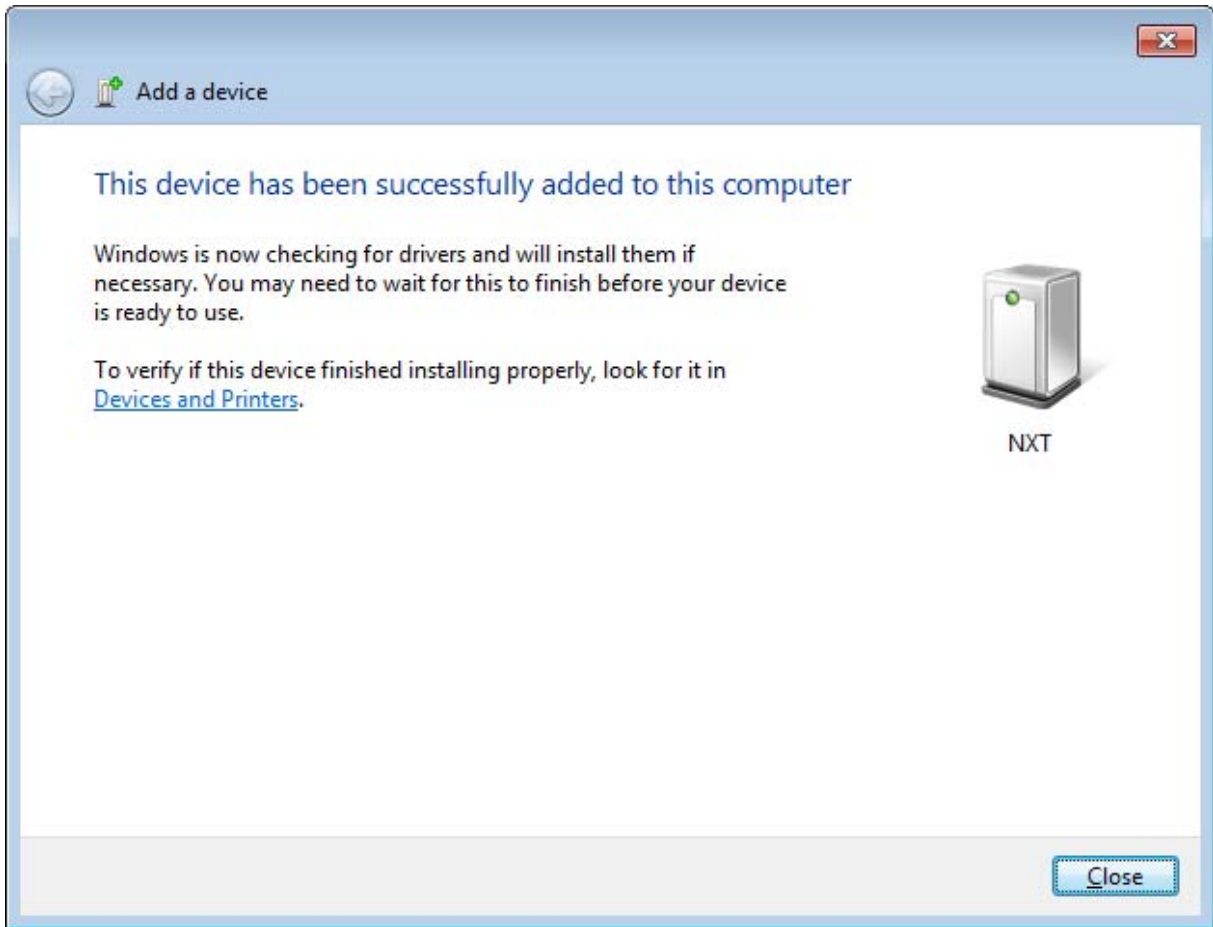
- 6 When the LCD on the NXT brick displays the default passkey, 1234, promptly press the orange button on the NXT brick.

Note If the NXT times out during this step, return to Windows, select the NXT Bluetooth device, and click **Next**.

- 7 In the **Add a device** dialog box, enter the passkey, 1234, as the pairing code. Click **Next**.



- 8 When you see confirmation that the device has been added, click **Close**. The next time you turn on both devices, the Bluetooth connection between your host computer and the NXT brick will be available.



Tip If you have trouble with the Bluetooth connection, try removing the device from Devices and Printers and repeating this process.

PandaBoard Hardware

- “Install Support for PandaBoard Hardware” on page 60-2
- “Replace Firmware on PandaBoard Hardware” on page 60-3
- “Choose the Type of Serial Cable” on page 60-18
- “Connect to Serial Port on PandaBoard Hardware” on page 60-19
- “Configure Network Connection with PandaBoard Hardware” on page 60-24
- “Get IP Address of PandaBoard Hardware” on page 60-28
- “Open Block Library for PandaBoard Hardware” on page 60-31
- “Run Model on PandaBoard Hardware” on page 60-34
- “Tune and Monitor Model Running on PandaBoard Hardware” on page 60-37
- “Detect and Fix Task Overruns on PandaBoard Hardware” on page 60-40
- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 60-41
- “Access the Linux Desktop Remotely Using VNC” on page 60-43
- “Configure Wi-Fi on PandaBoard Hardware” on page 60-46

Install Support for PandaBoard Hardware

You can add support for PandaBoard hardware to Simulink product, using the following process. After you complete this process, and replace the firmware on the PandaBoard hardware, you can run a Simulink model on your PandaBoard hardware.

To install support for PandaBoard hardware:

- 1** In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2** Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

Using this installation process, you download and install the following items on your host computer:

- Third-party software development tools
- A Simulink block library called Simulink Support Package for PandaBoard Hardware.
- Examples

For convenience, these instructions occasionally refer to PandaBoard hardware as a “board” or as “target hardware”.

Replace Firmware on PandaBoard Hardware

You can replace the firmware on the PandaBoard hardware (the "board") using the following process. To run Simulink models, you must replace the original firmware with the distribution provided by the Support Package Installer.

Before replacing the firmware, install the support package for PandaBoard.

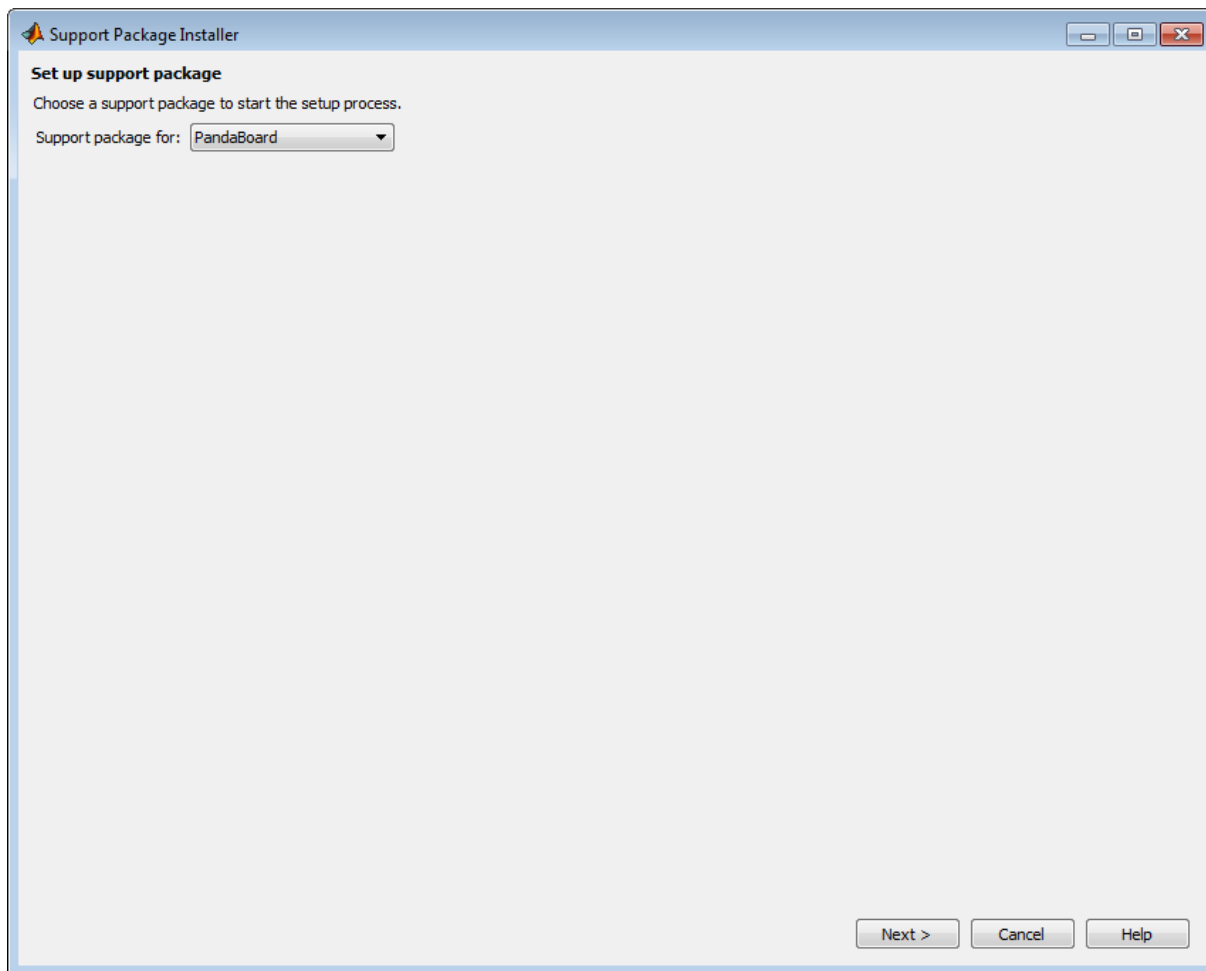
After replacing the firmware, you can run a Simulink model on the PandaBoard hardware.

The following steps provide an overview of the firmware replacement process:

- 1** Support Package Installer locates a firmware image on your host computer or downloads new one.
- 2** Support Package Installer uses the host computer to write the firmware image to a microSD or SD memory card.
- 3** You transfer the microSD or SD memory card to the PandaBoard hardware.
- 4** Support Package Installer applies the IP settings you choose to the firmware on the PandaBoard hardware.

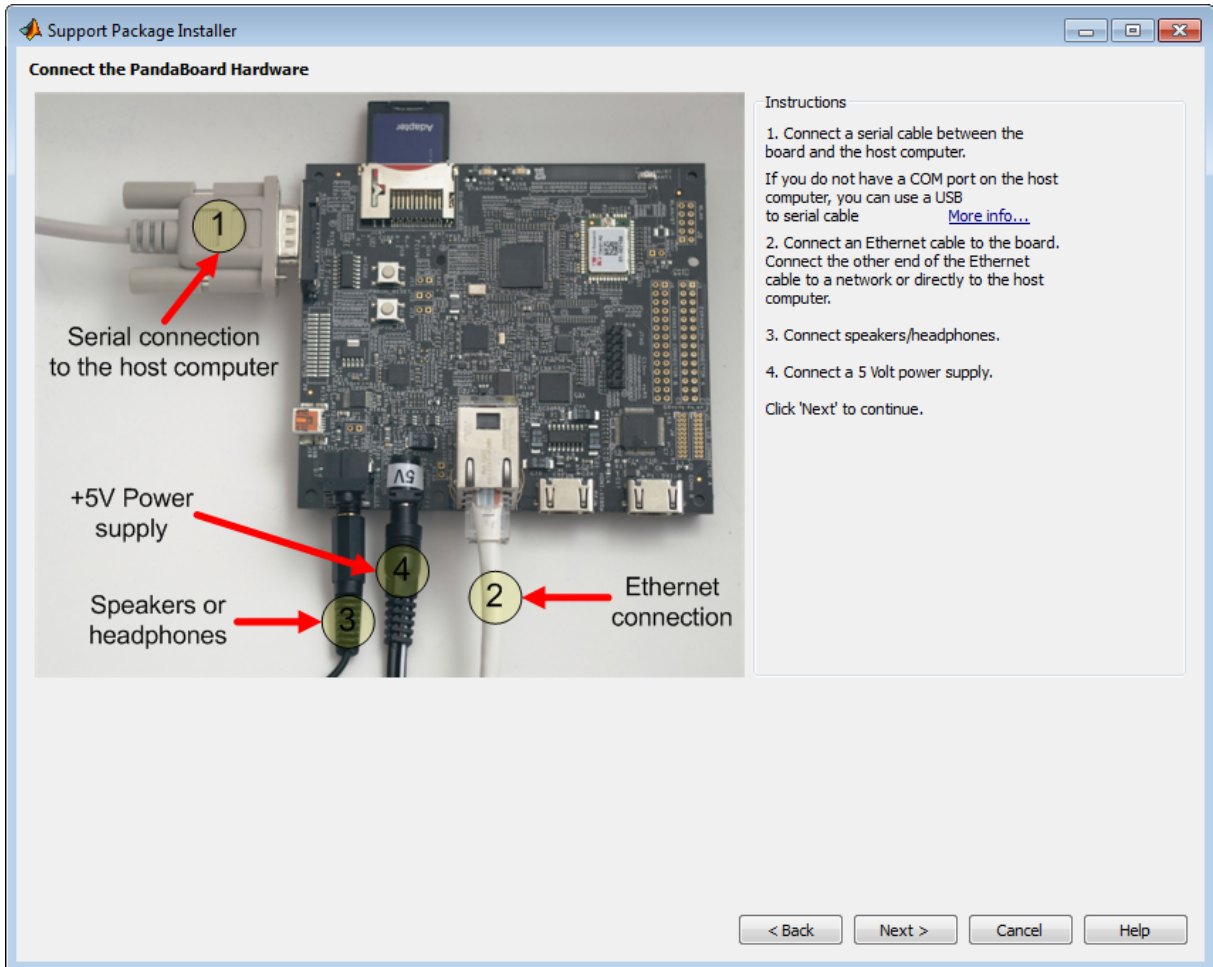
Support Package Installer does not use the PandaBoard hardware to write the firmware image to the memory card.

- 1** To replace the firmware on your PandaBoard hardware:
- 2** Open the **Update firmware** screen in Support Package Installer using one of the following methods:
 - Click **Continue** in the **Install/update complete** screen of Support Package Installer.
 - In a model, select **Tools > Run on Target Hardware > Update firmware**.
 - In a MATLAB Command Window, enter `targetupdater`.
- 3** Choose the PandaBoard option and click **Next**.



4 Make the connections shown and click **Next**.

Note For “3. Speakers or headphones”, use the lower jack (closest to the board).



- 5 Choose to get the firmware image from the Internet or from a folder.

Note The file size of the firmware image is approximately 1 GB. Depending on your connection, downloading the firmware can take from 2 to 60 minutes, or more.

Internet: This option is selected by default. When you click **Download**, Support Package Installer checks the **Download folder** for a valid firmware image:

- If a firmware image is not present, Support Package Installer downloads a firmware image from the Internet, and saves it to the download folder.
- If a firmware image is present, Support Package Installer uses the firmware image already present in the download folder, and does not download a new firmware image from the Internet.

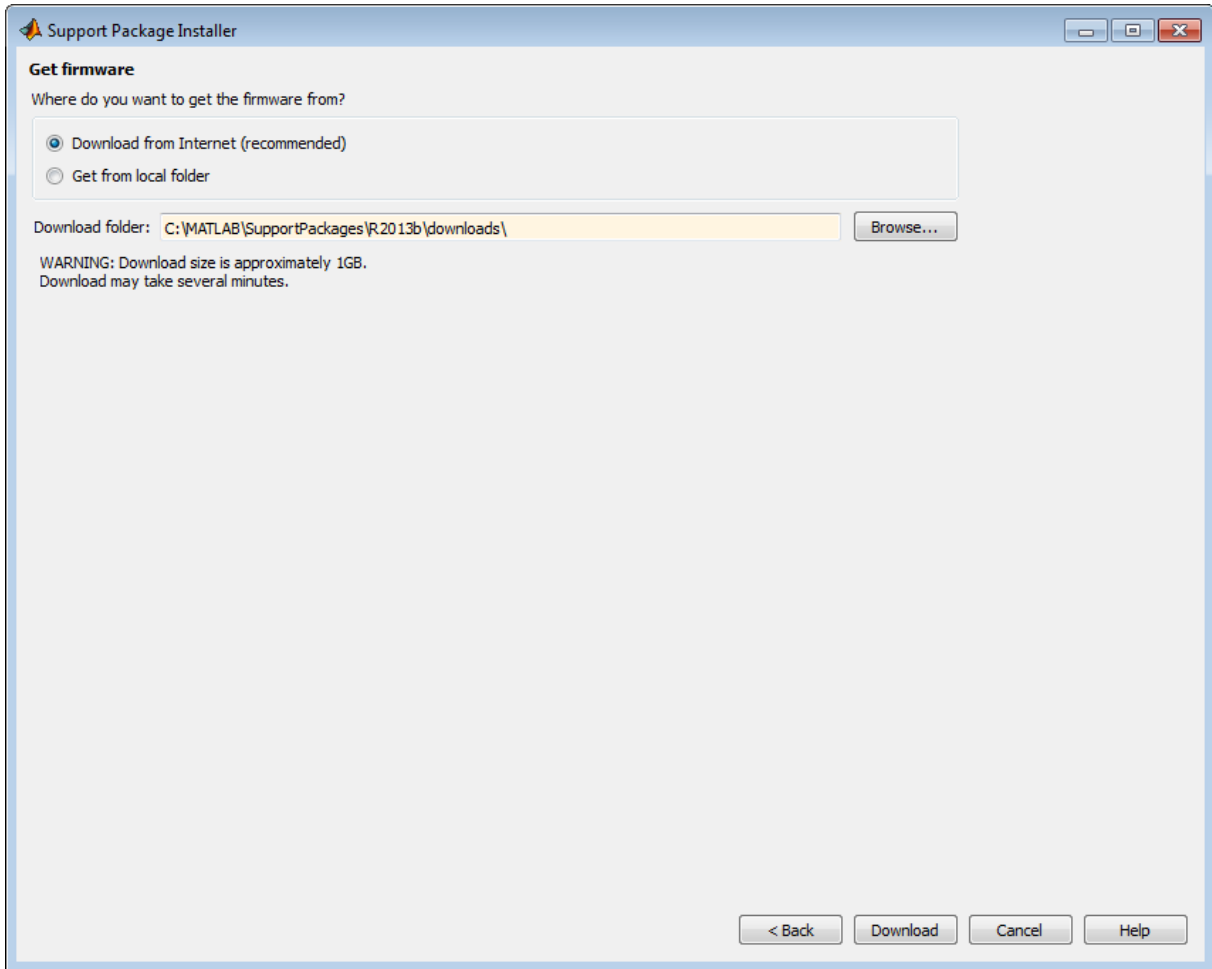
Folder: When you click **Download**, Support Package Installer checks the **Download folder** for a valid firmware image:

- If a firmware image is not present, Support Package Installer displays an error message that the image file is missing. To solve this issue, copy the firmware image from another location to the download folder, or choose the **Internet** option instead.
- If a firmware image is present, Support Package Installer continues the firmware installation process.

You must have write privileges for the download folder. If you use the default download folder, having write privileges is typically not an issue. If you change to a new download folder for which you do not have write permissions, such as a shared folder on a network, Support Package Installer generates an error message: “Error: Download the firmware. The download folder is not writable. Choose a folder for which you have write permissions”.

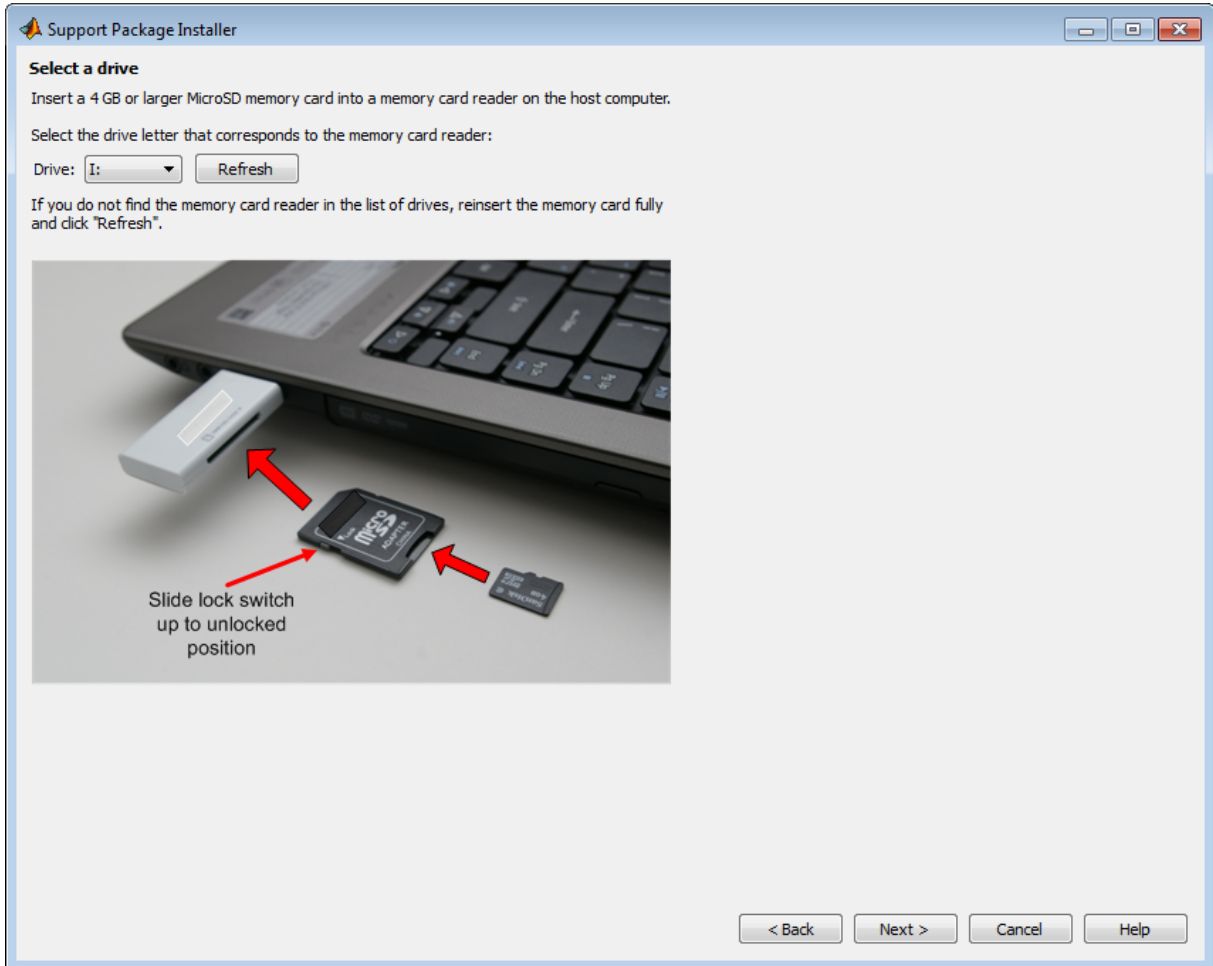
To solve this issue, copy the firmware image to a folder for which you have write privileges. For example, copy the firmware image from the shared folder on the network to the C:\Users\username\AppData\Local\Temp folder. Then, update the **Download location** to the same folder, and click **Download** again.

To locate the firmware image in a folder, search for a filename that begins with `pandaboard` and ends with `.img` or `.7z`. For example: `pandaboard_ubuntu_11_10_3_1_1_04_14_2012.img`

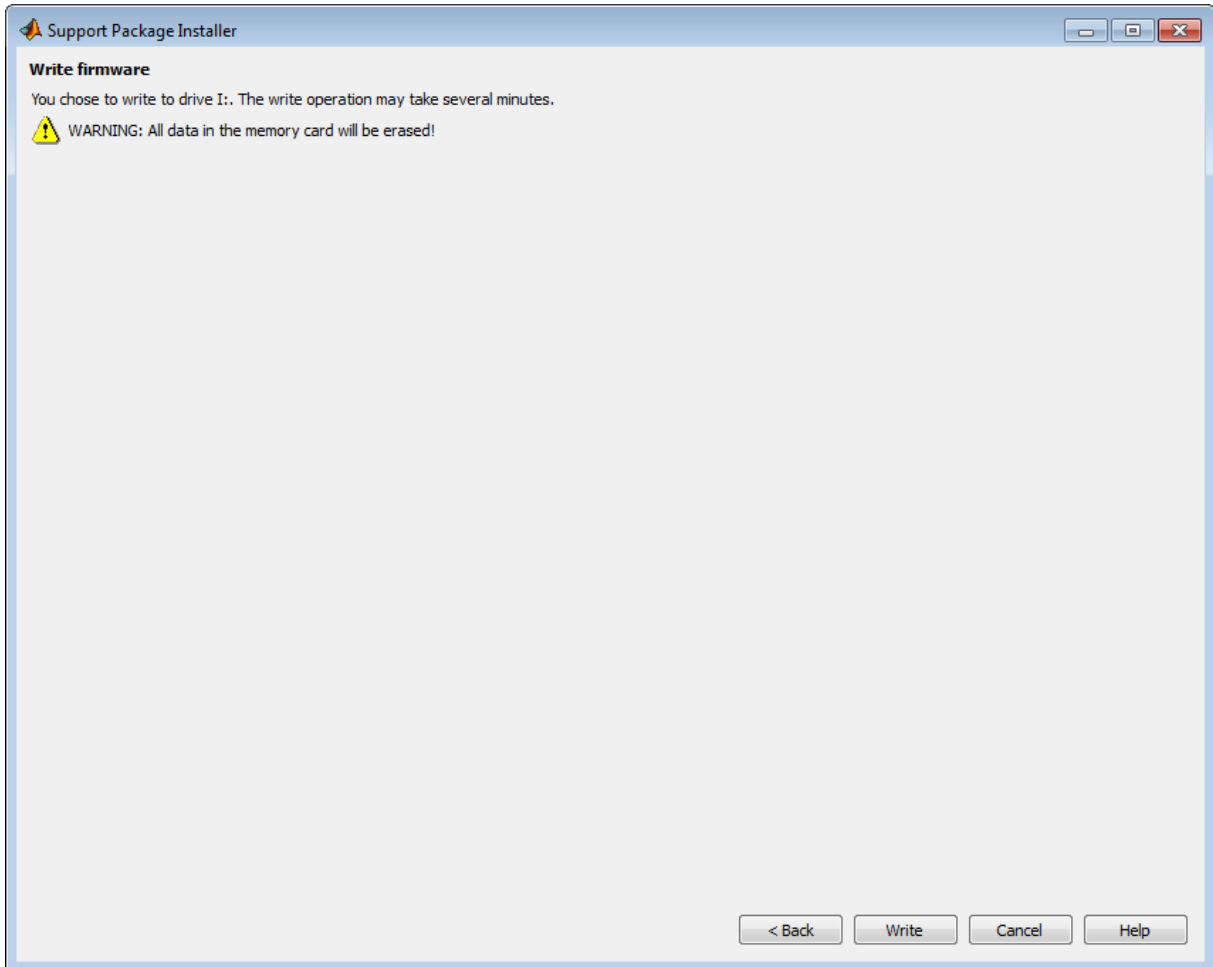


- 6 Insert the microSD or SD memory card into a media card reader connected to your host computer. Windows assigns a drive letter to the memory card.
- 7 Support Package Installer does not automatically detect the drive letter of the memory card. It displays a drive letter for each device with removable storage.
 - If only one drive letter is available, click **Next**.

- If no drive letters are available, check that the memory card is fully inserted, and click **Refresh**.
- If multiple drive letters are available, open the Windows Start menu, choose **Computer**, and look for the memory card under **Devices with Removable Storage**.

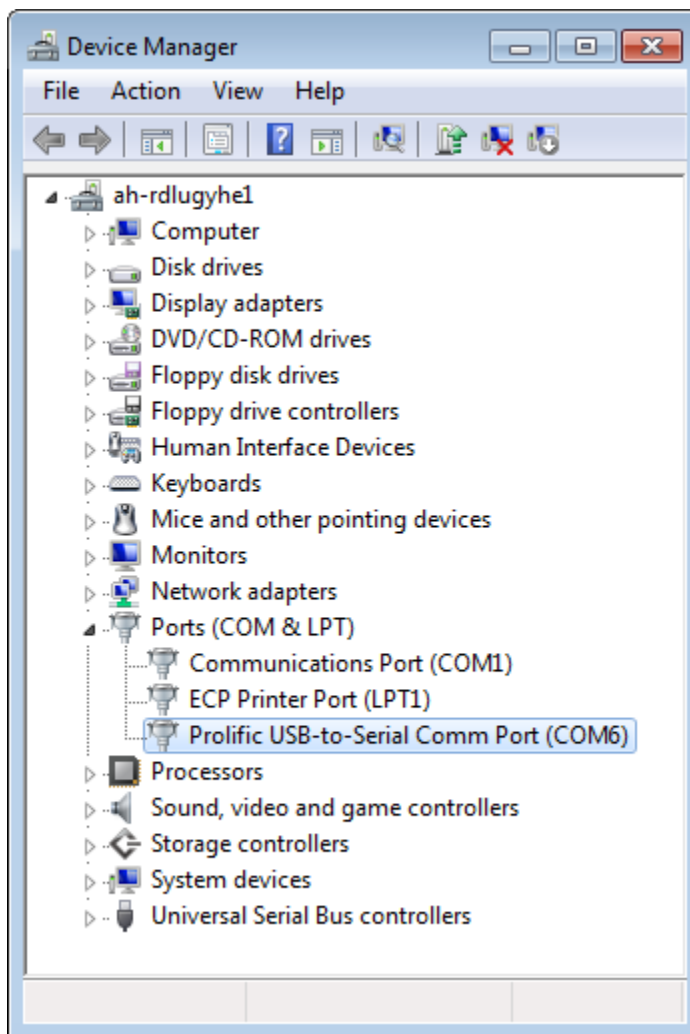


- 8 Click **Write**. Support Package Installer overwrites all previous data on the memory card with the firmware. This process takes several minutes to complete.



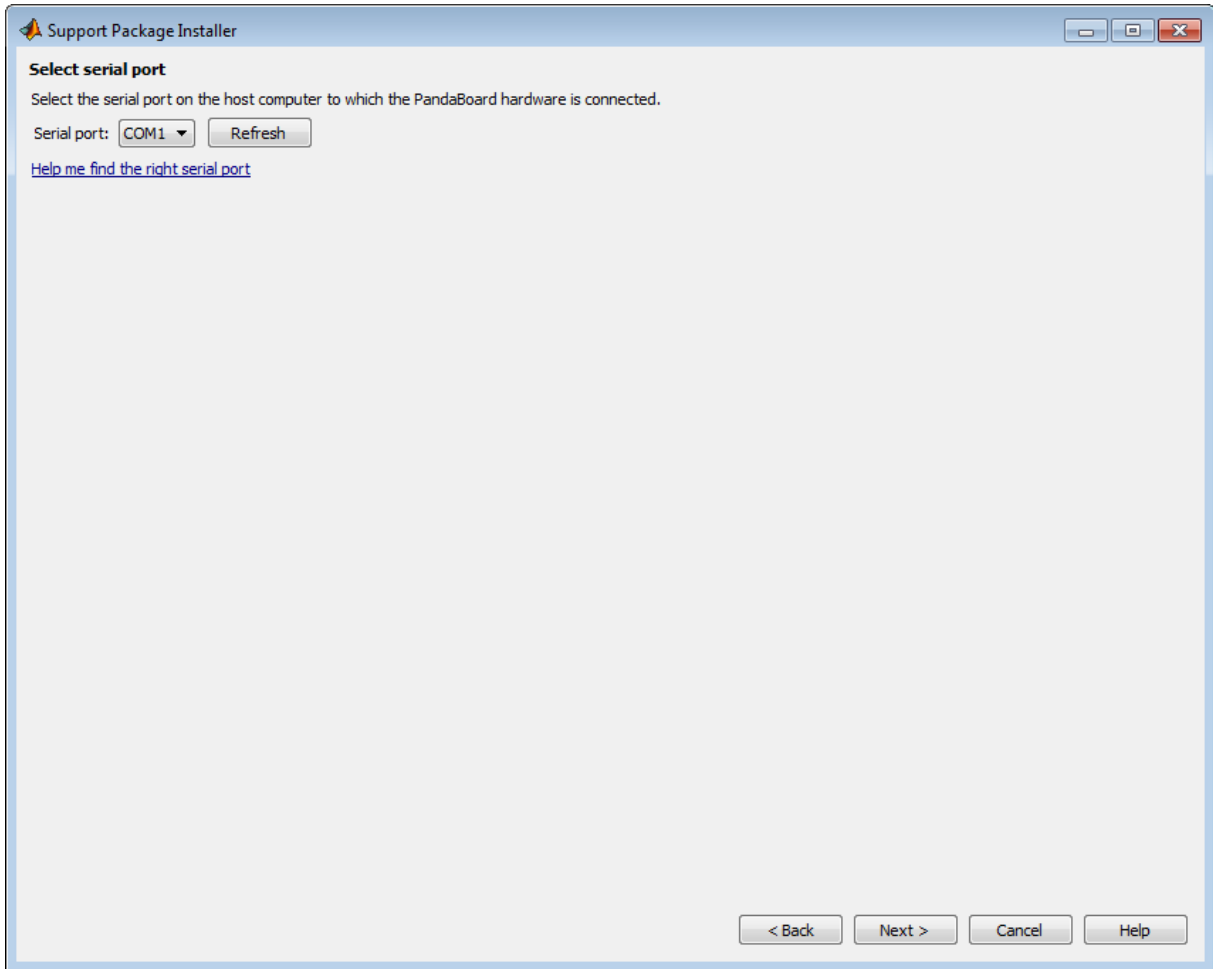
- 9 Get the COM port of the serial connection in Windows. From the Windows Start menu, choose search programs and files for "Device Manager". Open Device Manager, expand **Ports (COM & LPT)**, and identify the COM port of the serial connection to the PandaBoard hardware.

For example, the following image shows a DB9 serial port called "Communications Port" using COM1, and a USB-to-serial adapter called "Prolific USB-to-Serial Comm Port" using COM6.



Note Some USB-to-serial adapters do not appear in the list of serial connections immediately after you install the software drivers. To solve this issue, disconnect and then reconnect the adapter, or reboot your host computer.

- 10** Support Package Installer does not automatically choose the COM port of the serial connection. After you get the COM port of the serial connection from Windows Device Manager, return to Support Package Installer. Set **Serial port** to the COM port, and click **Next**.

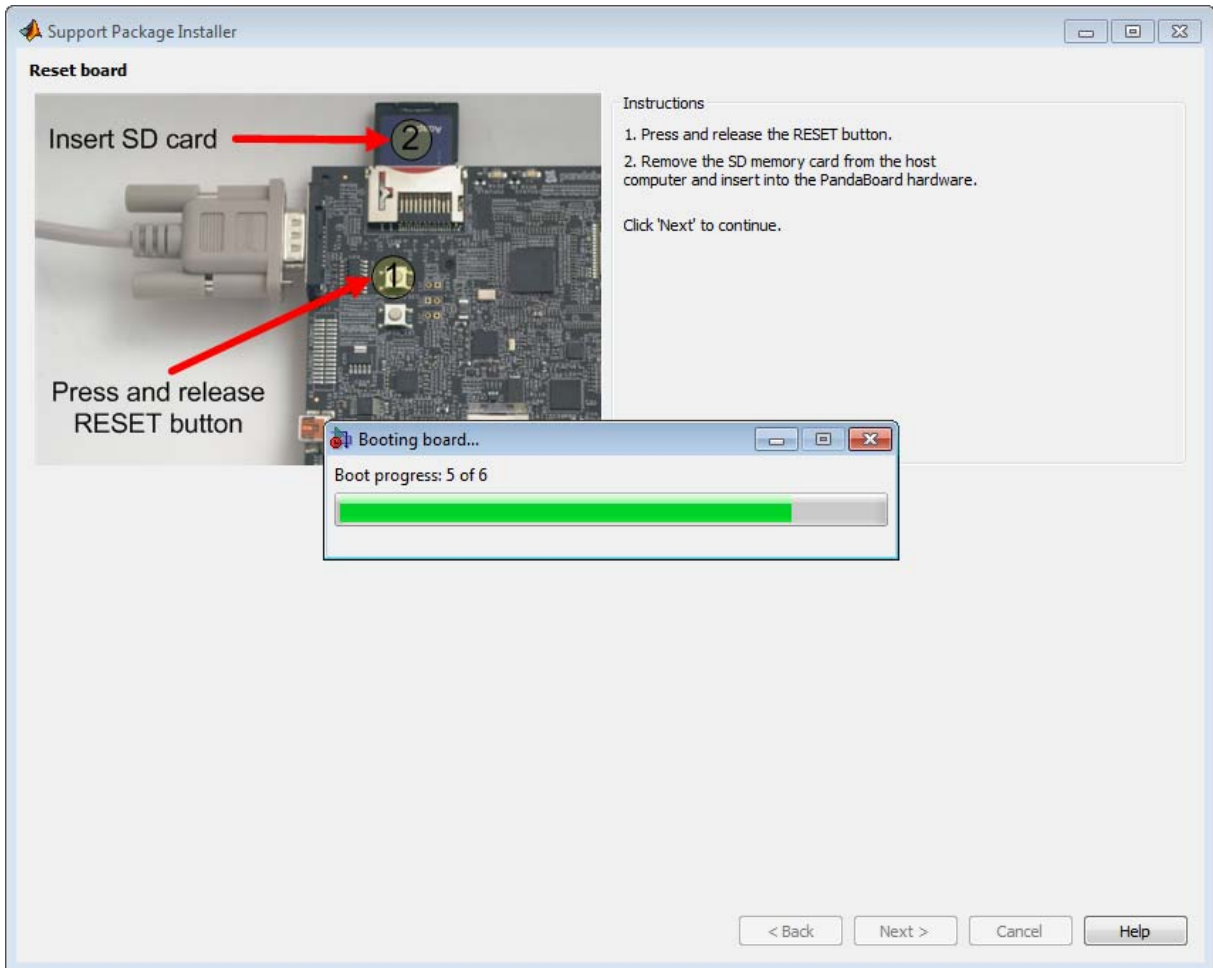


- 11 Press and release the RESET button. Then, insert the memory card into the PandaBoard hardware.

Click **Next**. When Support Package Installer detects a reset, it displays progress booting the board.

If Support Package Installer does not display any progress, click **< Back**. On the **Select serial port** screen. Verify that you selected the correct

COM port, and close any other applications, such as PuTTY, that might using the serial connection.



- 12** If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet, select **Automatically get IP address**, give the board a unique name, and click **Configure**. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.

If your board is directly connected to an Ethernet port on your computer, or connected to an isolated network without DHCP services, select **Manually enter IP address**, give the board a unique name, enter static IP settings for the board, and click **Configure**.

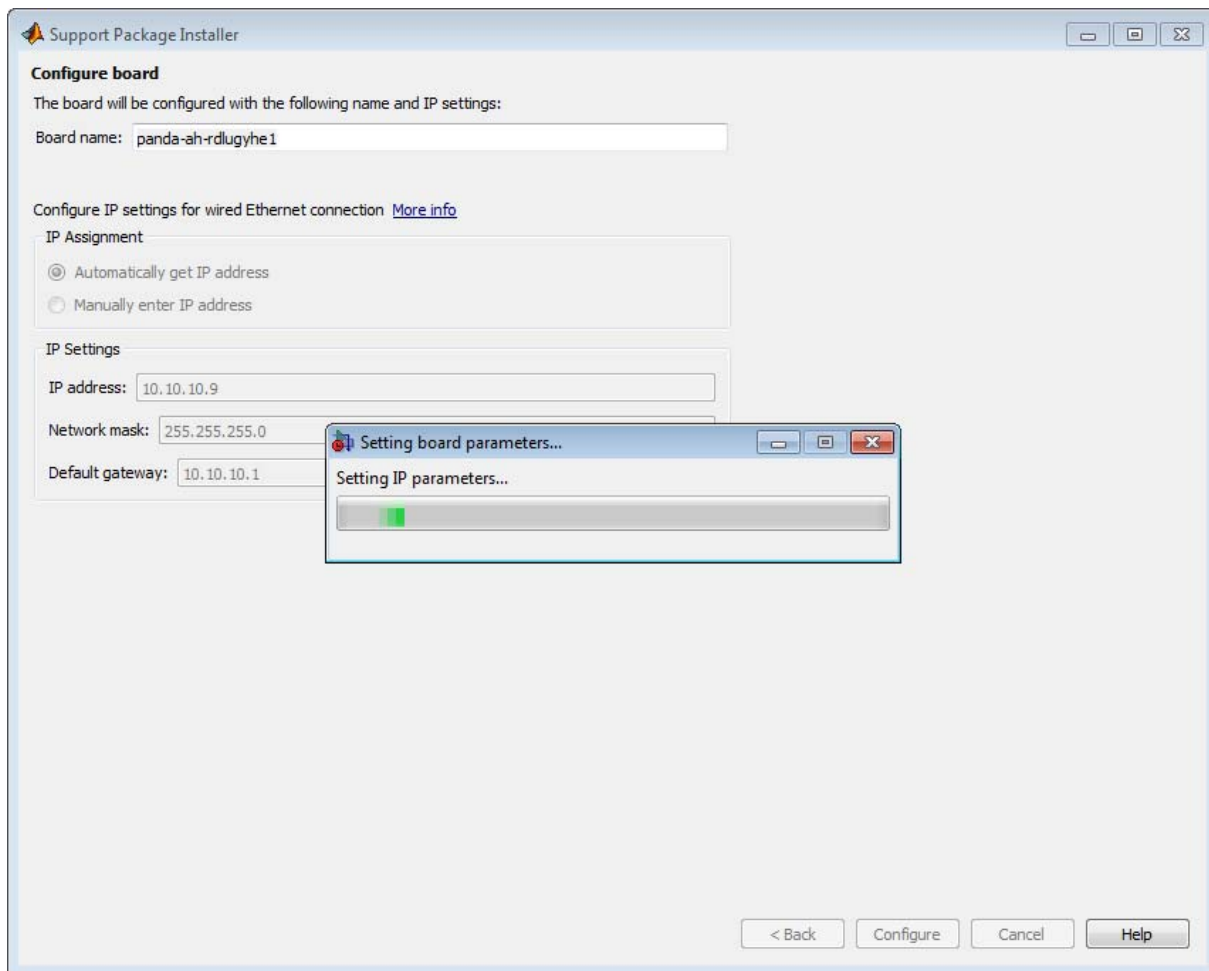
To enter static IP settings for your board, follow these guidelines:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

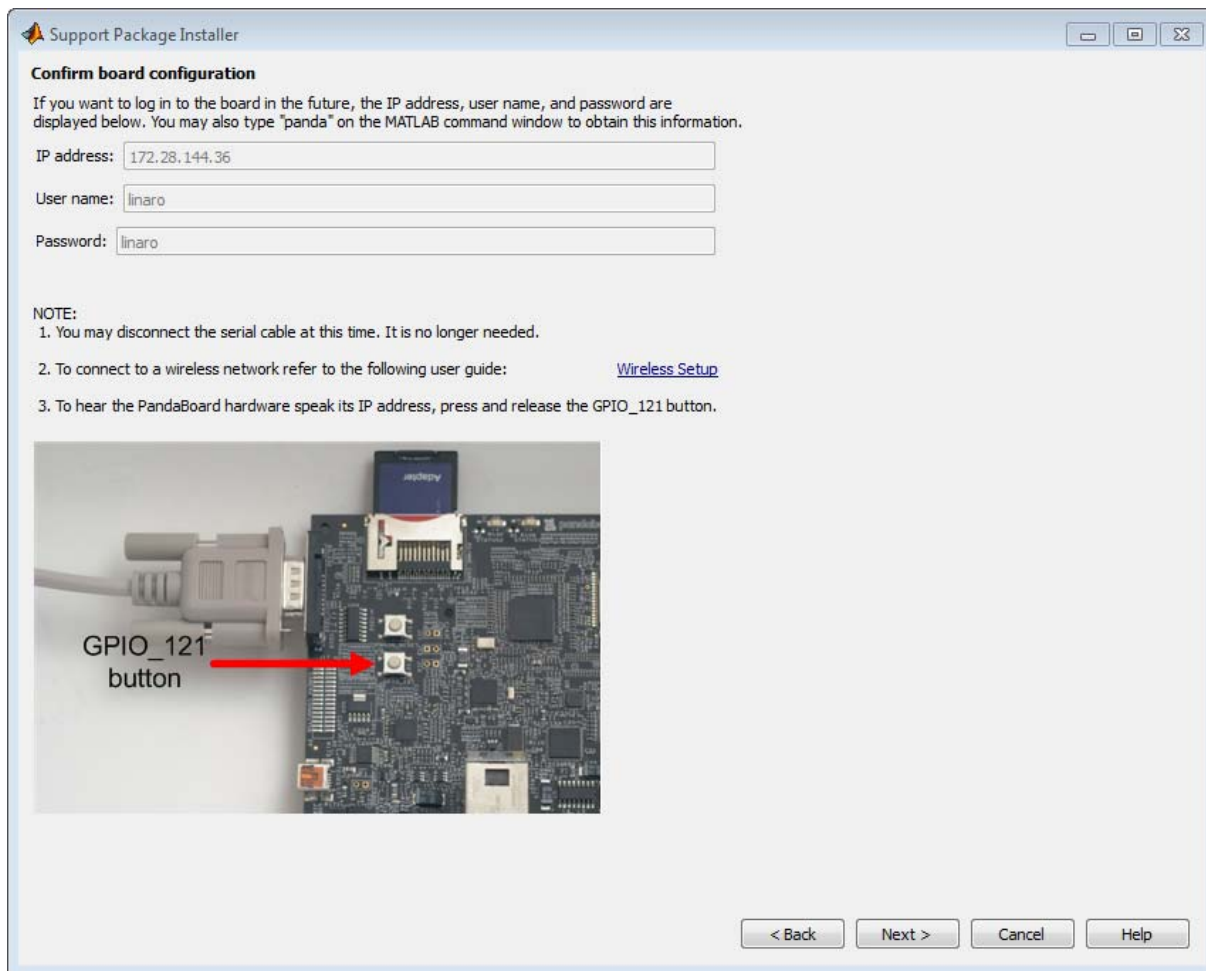
For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- **Network mask** to use the same network mask value, 255.255.255.0.
- **IP address** to an unused IP address, between 192.168.1.2 and 192.168.1.254.

When you click **Configure**, Support Package Installer opens a serial connection and applies the settings to the board.

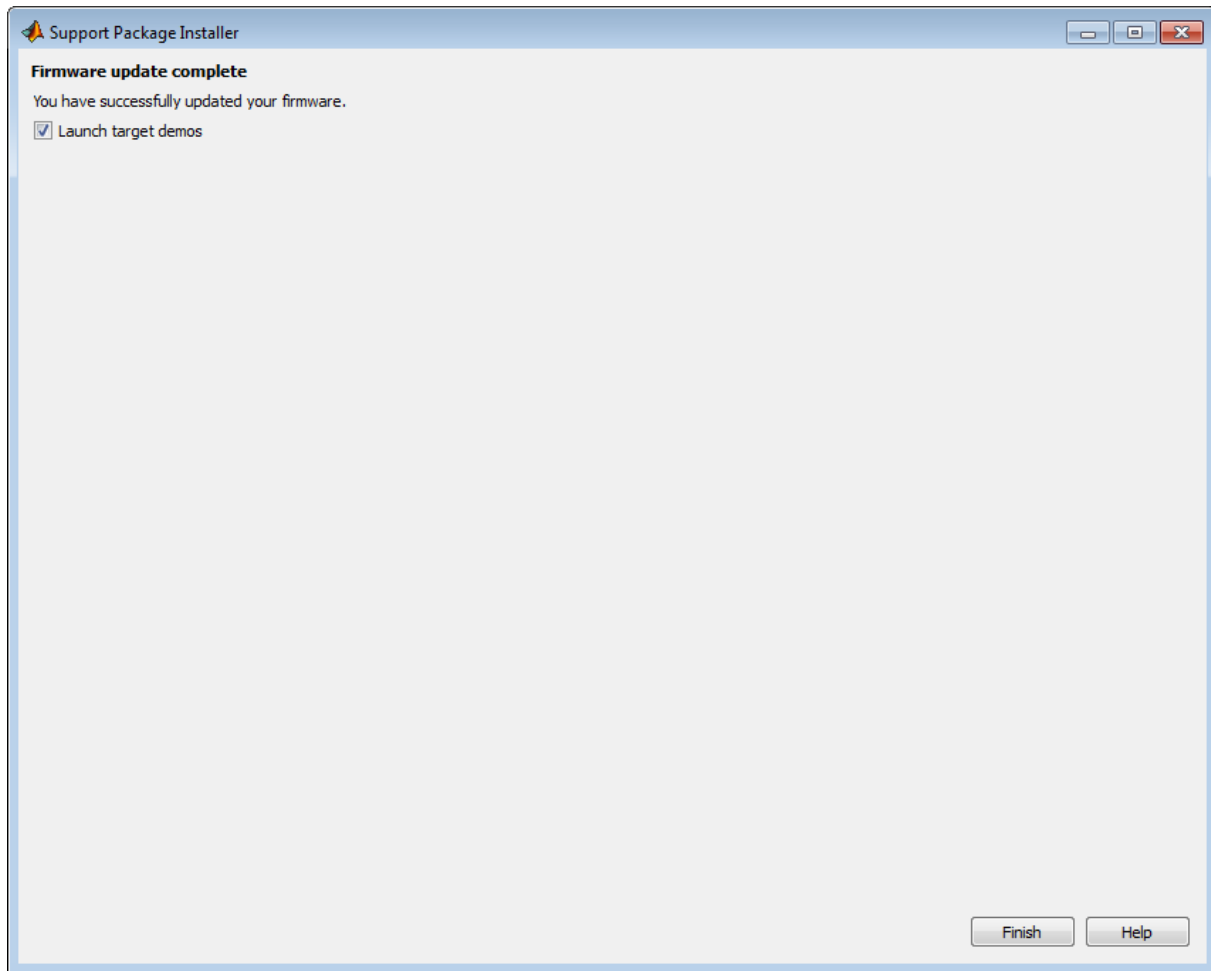


13 Make a note of the values for **IP address**, **User name**, and **Password** for use later on. Then, click **Next**.



14 Click **Finish**. If **Show support package examples** is enabled, Support Package Installer opens the example page for PandaBoard hardware.

To reopen these examples later on, see "Open Examples for This Support Package" on page 55-4.



Choose the Type of Serial Cable

This topic shows you how to choose a serial cable for connecting your host computer to the PandaBoard hardware.

Two types of cables are available for this purpose:

- DB9 null modem M/F cable
- USB to DB9 male adapter cable

DB9 null modem M/F cable:

- If your host computer has a DB9 male serial connector similar to the one in the following image, you can use a DB9 null modem M/F cable.



- To find the appropriate cable online, search for “DB9 low profile null modem M-F cable”.

USB to DB9 male adapter cable:

- If your host computer has a USB port, you can use a USB to DB9 male adapter cable.
- To find the appropriate cable online, search for "USB to RS-232 DB9 Serial Adapter”.
- Some USB-to-serial adapters do not appear in the list of serial connections immediately after you install the software drivers. To solve this issue, disconnect and reconnect the adapter, or reboot your host computer.
- To avoid issues with the software/driver quality, we recommend choosing an adapter that has good customer feedback ratings, or using the DB9 null modem M/F cable instead.

Connect to Serial Port on PandaBoard Hardware

This topic shows you how to configure and open a command line session with the target hardware.

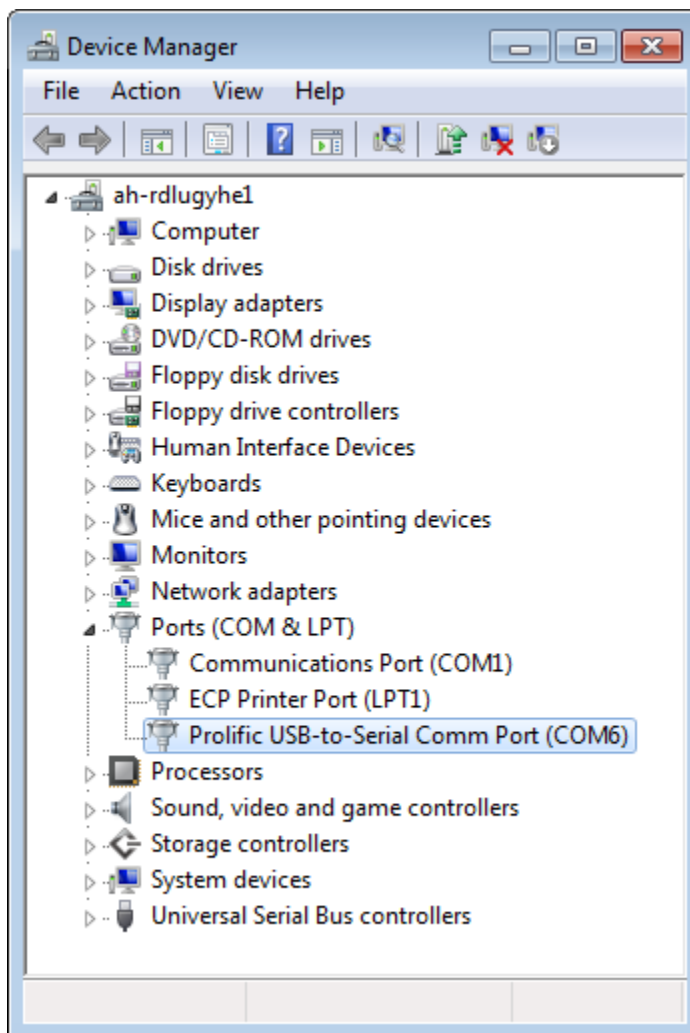
This is an optional task that you can use to:

- View the standard output while the board boots.
- Get or set the IP address, as described in “Get IP Address of PandaBoard Hardware” on page 60-28 topic.

To open a serial connection to your board:

- 1** Connect a serial cable from your host computer to the female DB9 connector on the board.
- 2** Identify the COM port for your serial connection. In Windows 7, use the “search programs and files” feature in the Start menu to find “Device Manager”. Open Device Manager, and expand **Ports (COM & LPT)** to see the list of serial connections.

For example, the following image shows that the DB9 serial port called “Communications Port” uses COM1. Similarly, the image shows that the USB-to-serial adapter called “Prolific USB-to-Serial Comm Port” uses COM6.

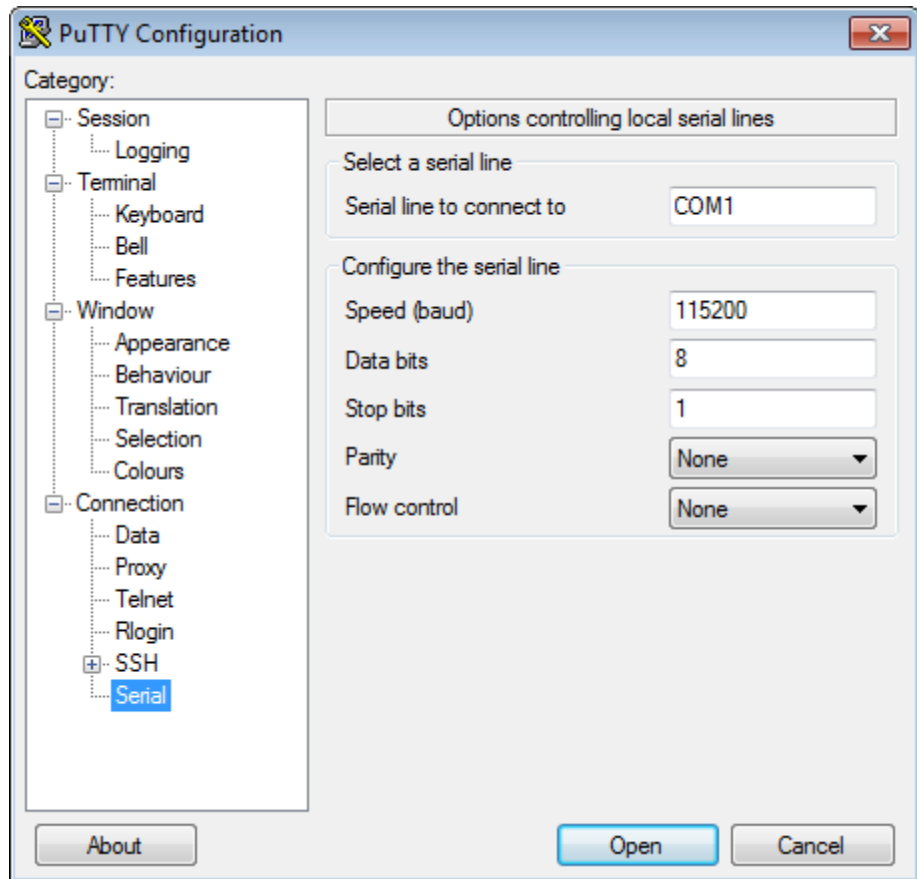


Note If you have trouble locating the COM port number, search the Windows “Help and Support” for “Device Manager”.

3 In the MATLAB Command Window, start PuTTY by entering:


```
h = panda;  
h.openShell('serial')
```

- 4 In the PuTTY dialog box that opens, select the **Serial** category.

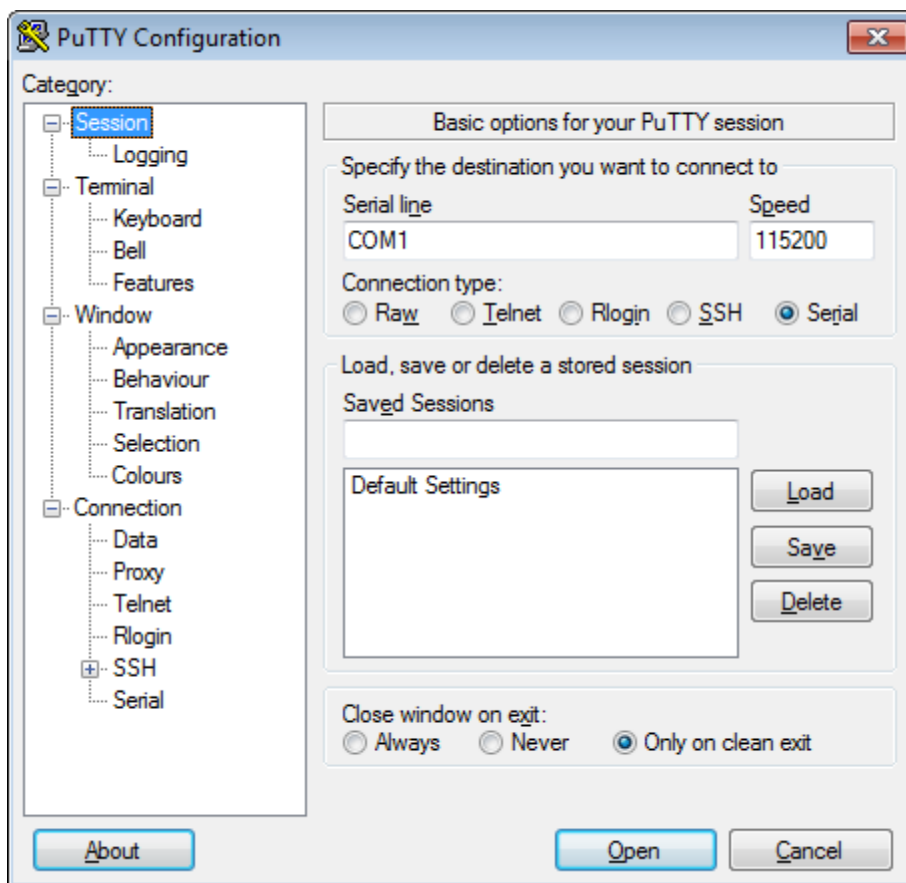


- 5 Set the following values:

- **Serial line to connect to:** If you are using your computer's built-in serial port, try COM1. Otherwise, use the Device Manager in Windows to determine the COM port number.
- **Speed (baud):** 115200

- **Flow control:** None

6 In the PuTTY dialog box, select the **Session** category.



7 Set the **Connection type** parameter to **Serial**, enter a new name for **Saved Sessions**, click **Save**, and click **Open**.

Note Do not save the serial settings to **Default Settings**. The **Default Settings** must remain configured to use SSH in order to run your Simulink model on the BeagleBoard hardware. Otherwise, trying to run the model on BeagleBoard hardware produces an error message.

- 8 When a terminal window opens, press the **Enter** key on your keyboard. The terminal window displays the Linux command line.
- 9 When you are finished, enter `logout` on the Linux command line, and close PuTTY to end the serial connection.

See Also

- “Get IP Address of PandaBoard Hardware” on page 60-28
- For more information about PuTTY, see:
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

Configure Network Connection with PandaBoard Hardware

You can configure the IP settings of the PandaBoard hardware by running Linux shell commands directly on the PandaBoard hardware.

To inspect and reconfigure the IP settings on a board that already has the new firmware, follow the procedure in this section that starts with “To configure the board to use DHCP or static IP settings”.

To configure the IP settings while you are replacing the firmware on your PandaBoard hardware, see “Replace Firmware on PandaBoard Hardware” on page 60-3.

You may need to reconfigure the IP settings if your board:

- Has unknown IP settings.
- Is unreachable using a network connection.
- Is being moved to a network or direct Ethernet connection that uses static IP settings.
- Is being moved from a network that used static IP settings to one that uses DHCP services.

When you complete the procedure in this topic, the board should be able to communicate over the network to which it is connected.

Before starting the procedure in this topic, it helps to understand the conditions under which networks use DHCP or static IP settings:

- If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet, configure the board to use DHCP services. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.
- If your board is directly connected to an Ethernet port on your computer, or connected to an isolated network without DHCP services, configure the board to use static IP settings.

To configure the board to use DHCP or static IP settings:

- 1** Open a serial command line session, as described in “Connect to Serial Port on PandaBoard Hardware” on page 60-19

Alternatively, you can use a terminal window after accessing the Linux desktop as described in:

- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 60-41
- “Access the Linux Desktop Remotely Using VNC” on page 60-43

- 2** Display the contents of the `/etc/network/interfaces` file. Enter:

```
cat /etc/network/interfaces
```

If the board is configured to use DHCP services (the default configuration), `dhcp` appears at the end of the following line:

```
iface eth0 inet dhcp
```

If the board is configured to use static IP settings, `static` appears at the end of the following line:

```
iface eth0 inet static
```

- 3** Create a backup of the `/etc/network/interfaces` file. Enter:

```
sudo cp /etc/network/interfaces /etc/network/interfaces.backup
```

If prompted, enter the root password (default: `linaro`).

- 4** Edit `interfaces` using a simple editor called `nano`. Enter:

```
sudo nano /etc/network/interfaces
```

- 5** Edit the last word of line that starts with `iface eth0 inet`.

To use DHCP services, edit the line to say:

```
iface eth0 inet dhcp
```

To use static IP settings, edit the line to say:

```
iface eth0 inet static
```

- 6** For static IP settings, add lines for address, netmask, and gateway. For example:

```
iface eth0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    gateway 192.168.1.1
```

Note For static IP settings:

- The value of the subnet mask must be the same for all devices on the network.
- The value of the IP address must be unique for each device on the network.

For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- netmask to use the same network mask value, 255.255.255.0.
 - address to an unused IP address, between 192.168.1.2 and 192.168.1.254.
-

- 7** Tell nano to exit and save the changes:

Press **Ctrl+X**.

Enter Y to save the modified buffer.

For “File Name to Write: interfaces”, press Enter.

The nano editor confirms that it “Wrote # lines” and returns control to the command line.

- 8** Reboot your board for the settings to take effect:

```
sudo shutdown -r now
```

- 9** Test the IP settings by logging in to the board over a telnet session.

Note You can use the `ifconfig` command to temporarily change the IP settings. Rebooting the board removes the `ifconfig` settings and restores the `/etc/network/interfaces` settings.

To change the IP settings temporarily, open a Linux command line, and enter `ifconfig`, the device id, a valid IP address, `netmask`, and the appropriate network mask. For example:

```
ifconfig eth0 192.168.45.12 netmask 255.255.255.0
```

Related Examples

- “Run Model on PandaBoard Hardware” on page 60-34
- “Get IP Address of PandaBoard Hardware” on page 60-28

Get IP Address of PandaBoard Hardware

You can get the IP address of the PandaBoard hardware by listening to the audio output, or by using the Linux command line. This is an optional task that you can use to:

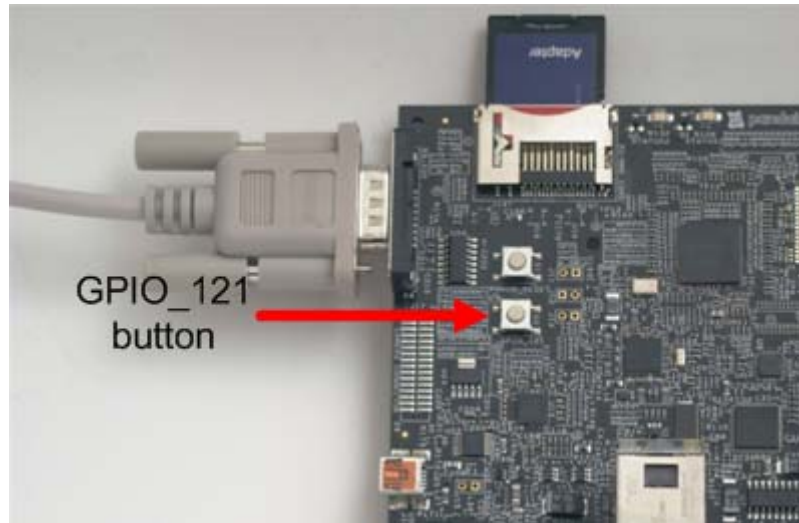
- Preparing a model to run on a board that is different from the previous one you used.
- Opening a telnet session with a board whose IP settings you do not know.
- Configuring the UDP Send block.

When you prepare a model to run on a board, your model Configuration Parameters reuse the IP settings of the previous board you used. If you are working with a different board on the same network, get the IP address of the your new board, and update the **Host name** parameter in the Configuration Parameters. For more information, see “Run Model on PandaBoard Hardware” on page 60-34.

If Ethernet port on the PandaBoard hardware board has an IP address, the audio output on the board can read the IP address out loud. For example the board can say “My IP address is one hundred and forty four point two one two point one one zero point two zero six.”

To listen your board’s IP address using headphones or speakers:

- 1** Plug headphones or speakers into the AUDIO OUT jack on the board.
- 2** Press the GPIO_121 button on the board.



- 3** Write down the IP address as the board reads it out over the headphones or speakers.

The Linux command line on the board can provide the IP address of the Ethernet port.

To get your board's IP address using a command line session:

- 1** Open a serial command line session on your board, as described in “Connect to Serial Port on PandaBoard Hardware” on page 60-19.

Alternatively, you can use a terminal window after accessing the Linux desktop as described in:

- “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 60-41
- “Access the Linux Desktop Remotely Using VNC” on page 60-43

- 2** At the Linux command line, enter `ifconfig`.

- 3** Locate the IP address following `inet addr` in the command line output. For example, locate `inet addr:172.28.144.36` on line 3 of the following output:

```
root@panda-ah-rdlugyhe1:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 0a:40:27:be:3b:0a
          inet addr:172.28.144.36  Bcast:172.28.144.255  Mask:255.255.255.0
          inet6 addr: fe80::840:27ff:febe:3b0a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1488  Metric:1
          RX packets:15137 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1080 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5118973 (5.1 MB)  TX bytes:144806 (144.8 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

wlan0     Link encap:Ethernet  HWaddr 0a:00:27:be:3b:0a
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@panda-ah-rdlugyhe1:~#
```

Related Examples

- “Connect to Serial Port on PandaBoard Hardware” on page 60-19
- “Run Model on PandaBoard Hardware” on page 60-34
- “Get IP Address of PandaBoard Hardware” on page 60-28

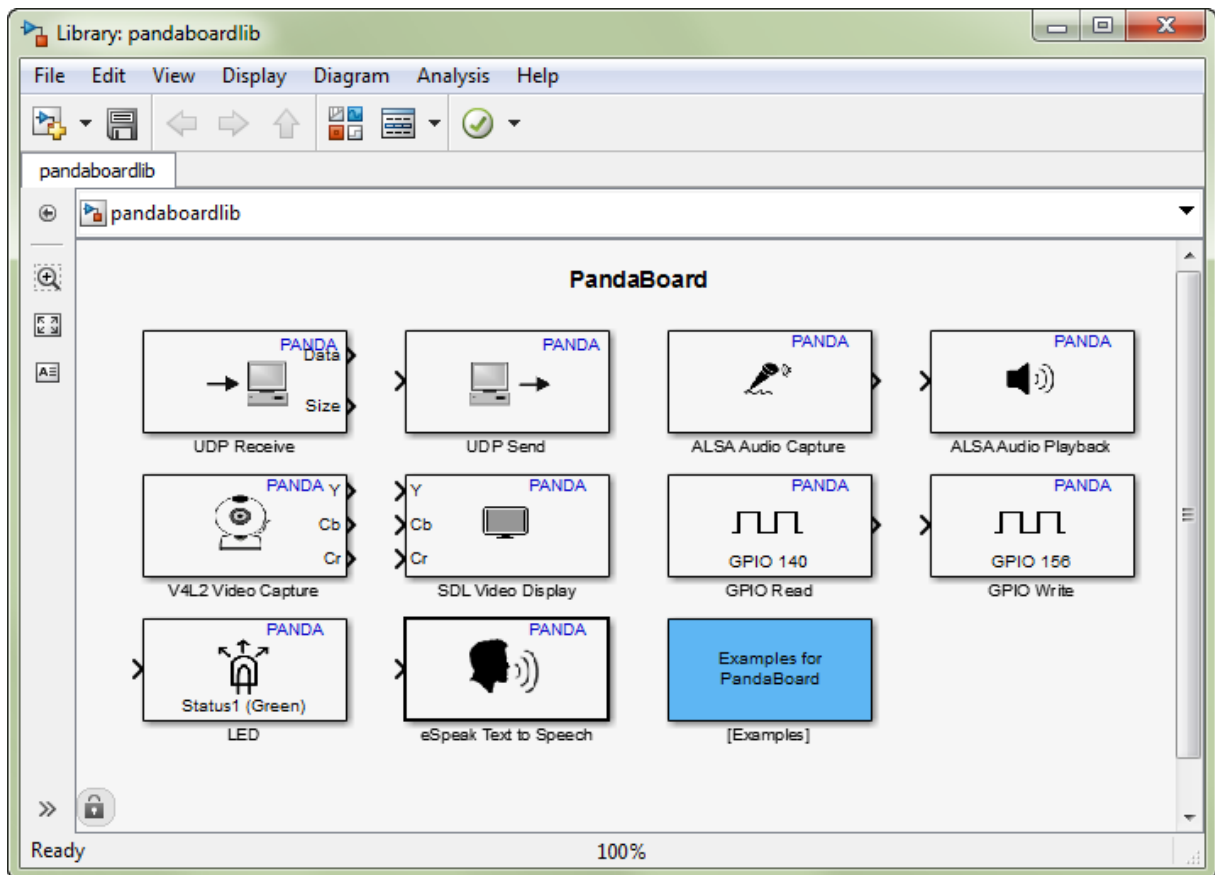
Open Block Library for PandaBoard Hardware

This block library provides support for protocols and APIs available on PandaBoard hardware.

To open the block library from the MATLAB Command Window, enter:

```
pandaboardlib
```

The following block library opens.

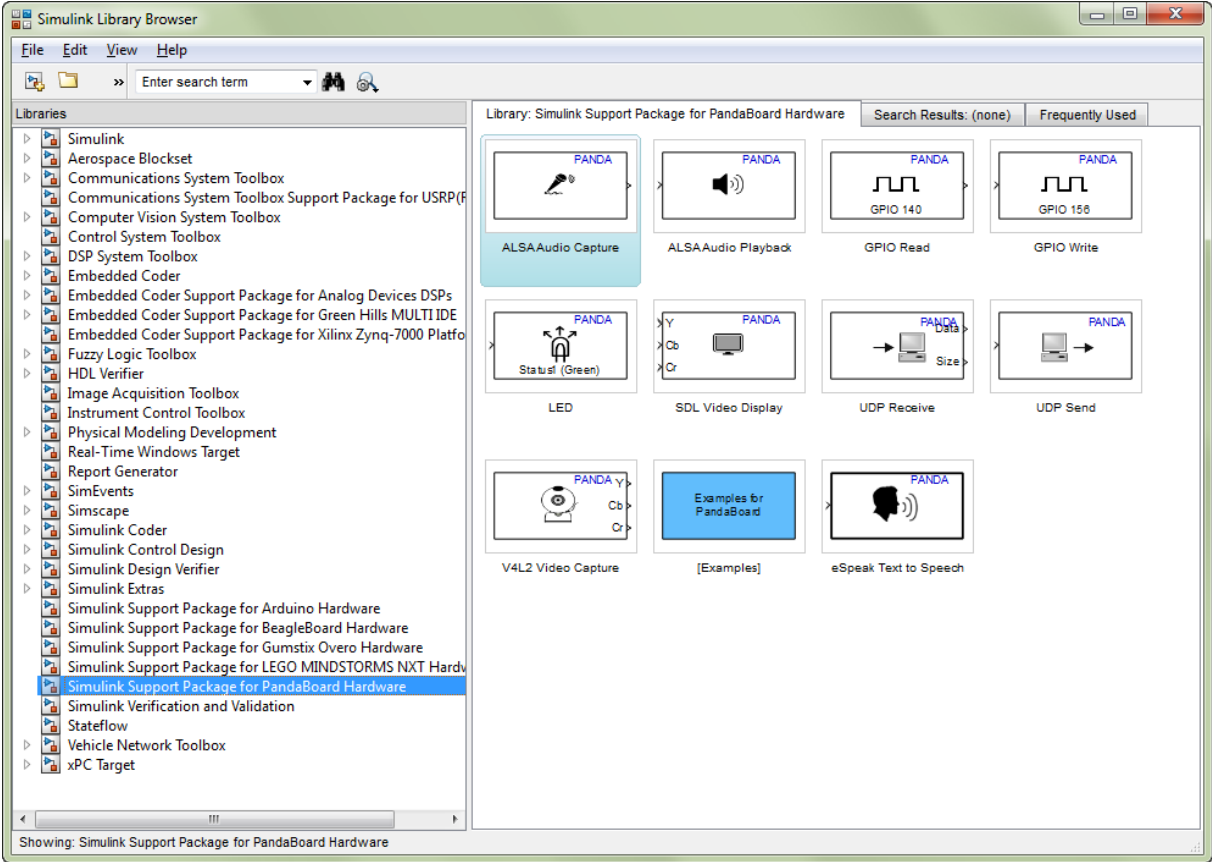


To open the block library from the Simulink Library Browser:

- 1** Open the Simulink Library Browser by entering `simulink` in the MATLAB Command Window, or by clicking the following icon on the MATLAB toolbar.



- 2** In the Simulink Library Browser, click **Simulink Support Package for PandaBoard Hardware**.



Run Model on PandaBoard Hardware

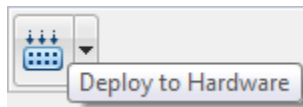
This example shows how to prepare, configure, and run a simple model on your PandaBoard hardware.

Before starting this procedure:

- Connect your board to the network and to a power supply.
- Create or open a Simulink model.

To prepare and run the model:

- 1** Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 2** In your model, select **Tools > Run on Target Hardware > Prepare to Run**. This action changes the model Configuration Parameters.
- 3** In the Run on Target Hardware pane that opens, set the **Target hardware** parameter to PandaBoard.
- 4** If you have changed boards since the last time you updated the firmware or ran a model, update the **Host name**, **User name**, and **Password** parameters.
- 5** Click the Deploy to Hardware button.



This action automatically downloads and runs your model on the board.

Note Pressing RESET or cycling the power on the PandaBoard hardware during this step can cause the ssh utility to hang.

Warning Avoid using the RESET button to stop a running model and reboot the board. Doing so can corrupt operating system and program files. To fix corrupt files, replace the firmware as described in “Replace Firmware on PandaBoard Hardware” on page 60-3.

Running a new or updated model on the board:

- Automatically stops a running model with the same name.
- Does not stop running models that have other names.

To stop a model running on the board, enter the following commands in the MATLAB Command Window:

```
h=panda;  
h.stop('modelName');
```

For example, to stop the sumdiff model, enter:

```
h=panda;  
h.stop('sumdiff');
```

To restart a model that was previously running on the board, or to run multiple instances of a model, enter the following commands in the MATLAB Command Window:

```
h=panda;  
h.run('modelName');
```

Note You do not need to enter `h=panda;` multiple times if a previous instance of `h` is available in the MATLAB Workspace.

For example, to restart the sumdiff model you stopped in the previous example, enter:

```
h.run('sumdiff');
```

See Also

- “Create the Simple Model”

- “Configure Network Connection with PandaBoard Hardware” on page 60-24
- “Tune and Monitor Model Running on PandaBoard Hardware” on page 60-37
- “Overview of Model Referencing” on page 6-2

Tune and Monitor Model Running on PandaBoard Hardware

In this section...

“About External Mode” on page 60-37

“Run Your Simulink Model in External Mode” on page 60-38

“Stop External Mode” on page 60-39

About External Mode

You can use External mode to tune parameters and monitor a model running on your target hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on model results in real-time, in order to find the optimal values to achieve desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to re-run the model each time you change parameters. External mode also lets you develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

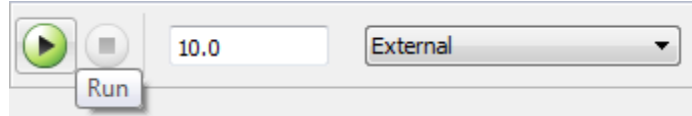
The following list provides an overview of the parameter tuning process with External mode:

- In the model on your host computer, you enable External mode in the Configuration Parameters.
- In the model on your host computer, you tell Simulink software to run your model on the target hardware.
- Simulink software runs the model on the target hardware.
- You use the model on the host computer as a user interface for interacting with the model running on the target hardware:

- When open blocks and apply new parameter values on the host computer, External mode updates the corresponding values on target hardware.
- If your model contains blocks for viewing data, such as Scope or Display blocks, External mode sends the corresponding data from the target hardware to those blocks on the host computer.
- You determine the optimal parameter values by adjusting parameter values on the host computer and observing data/outputs from the target hardware.
- When you are finished, you save the new parameter values, disable External mode, and save the model.

Run Your Simulink Model in External Mode

- 1 Create a network connection between the PandaBoard hardware and your host computer. See “Configure Network Connection with PandaBoard Hardware” on page 60-24
- 2 On the model toolbar, set **Simulation mode** to External mode.



- 3 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 4 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\sl_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

- 5 While the model is running in External mode, you can change parameter values in the model on your host computer and observe the corresponding changes in the model running on the hardware.

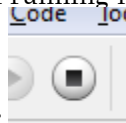
If your model contains blocks from the Simulink Sinks block library, the sink blocks in the model on your host computer display the values generated by the model running on the hardware.

Note External mode increases the processing burden of the model running on the board. If the software reports an overrun, disable External mode.

Note If you have the Embedded Coder or Simulink Coder products, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

Stop External Mode

To stop the model running in External mode, click the Stop button located on



the model toolbar.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

Detect and Fix Task Overruns on PandaBoard Hardware

You can configure a model running on the target hardware to detect and notify you of when task overrun occurs.

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. When a task is required to perform extra processing and takes longer than normal to execute, it may be scheduled to execute before a previous instance of the same task has completed. The result is a task overrun.

To enable overrun detection:

- 1** In your model, click **Tools > Run on Target Hardware > Options**.
- 2** In the Run on Target Hardware pane that opens, select the **Enable overrun detection** check box.
- 3** Click **OK**.

When a task overrun occurs, the command prompt on the host machine repeatedly prints an “Overrun” error message, such as “Overrun — rate for subrate task 1 is too fast”, until the model stops.

To fix an overrun condition:

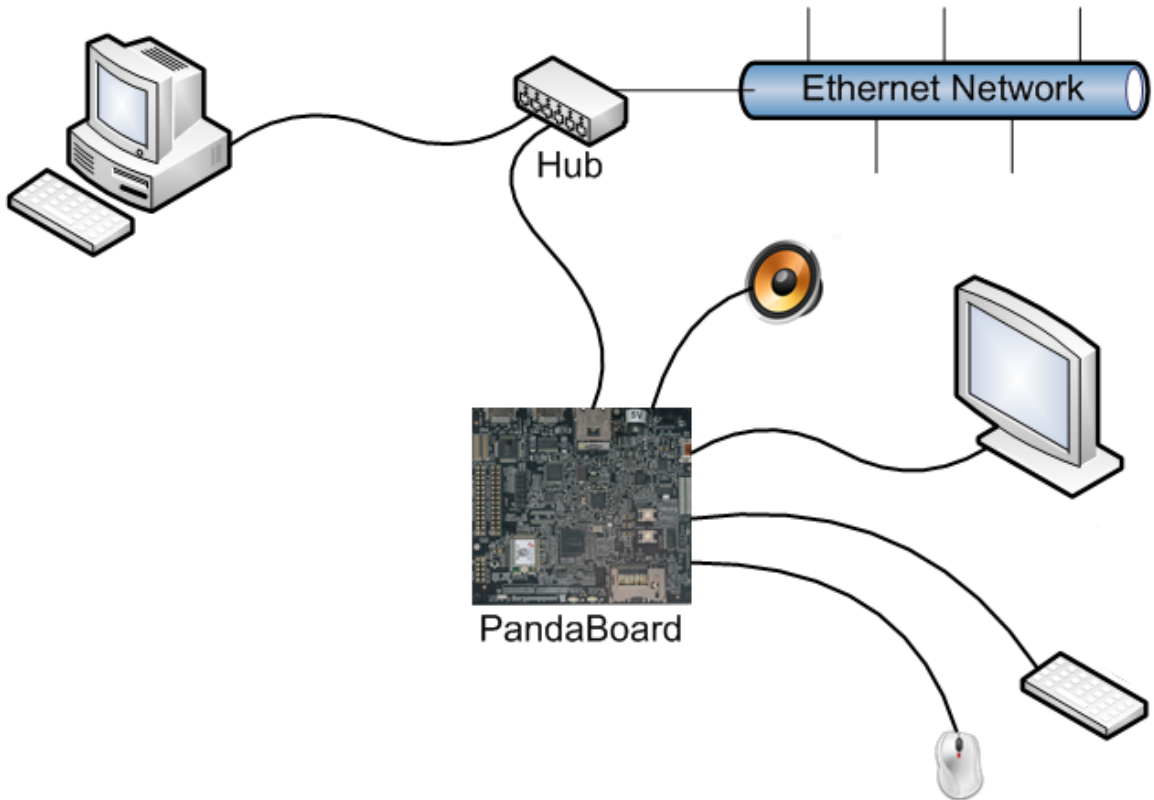
- Simplify the model.
- Increase the sample times for the model and the blocks in it. For example, change the **Sample time** parameter in all of your data source blocks, such as blocks for input devices, from 0.1 to 0.2.

Note External mode increases the processing burden of the model running on your board. If the software reports an overrun, disable External mode.

Access the Linux Desktop Directly Using Desktop Computer Peripherals

You can access the Linux desktop on the PandaBoard hardware using hardware peripherals such as a monitor, keyboard, and mouse.

You can use this approach to connect the PandaBoard hardware to a wireless network, as described in “Configure Wi-Fi on PandaBoard Hardware” on page 60-46.



If you do not have hardware peripherals, you can access the Linux desktop as described in “Access the Linux Desktop Remotely Using VNC” on page 60-43.

To connect to the PandaBoard hardware to the hardware peripherals:

- 1** Connect a computer monitor to the HDMI output, labelled P2 on the PandaBoard hardware.
- 2** Connect the USB keyboard and USB mouse to the USB ports on the PandaBoard hardware.

Note To connect additional USB devices, use a powered USB hub.

- 3** You can connect computer speakers to the lower of the two 3.5 mm audio connectors, labelled J16 on the PandaBoard hardware.
- 4** You can connect a network cable to the Ethernet port.
- 5** Connect the PandaBoard hardware to the power supply.
- 6** Log in to the Linux desktop using the username and password. The default user name and password is `linaro`.

Access the Linux Desktop Remotely Using VNC

You can remotely access the Linux desktop on the PandaBoard hardware from another computer. This capability uses a client-server graphical desktop sharing system called Virtual Network Computing (VNC).

The PandaBoard firmware distributed by MathWorks has a built-in VNC server called *x11vnc*. To remotely interact with the PandaBoard hardware, you can run a VNC client on a remote computer, open a VNC connection with *x11vnc* on the PandaBoard hardware.

To open a VNC session from MATLAB:

- 1 Create a physical network connection between your host computer and the PandaBoard hardware.
- 2 In the MATLAB Command Window, create a connection to the PandaBoard hardware by entering:

```
h = panda
```

The command line responds with information about the PandaBoard hardware. For example:

```
h =  
  
realtime.internal.LinuxServices  
Package: realtime.internal  
  
Properties:  
  HostName: '172.28.144.36'  
  UserName: 'linaro'  
  Password: 'linaro'  
  BuildDir: '/home/linaro'  
  PutilsFolder: [1x66 char]  
  
Methods
```

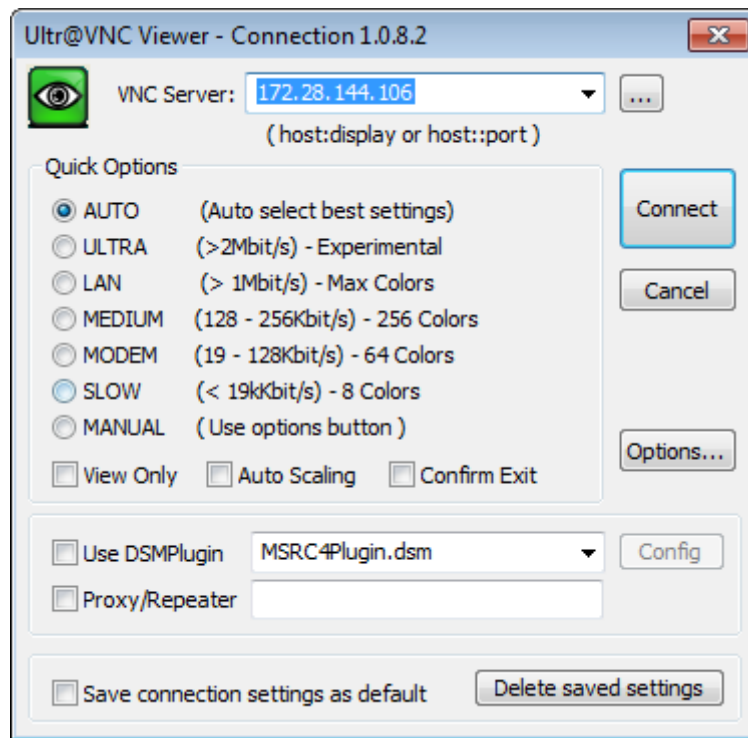
Enable the VNC server on the PandaBoard hardware by entering:

```
h.execute('export DISPLAY=:0.0; xhost+; x11vnc -bg')
```

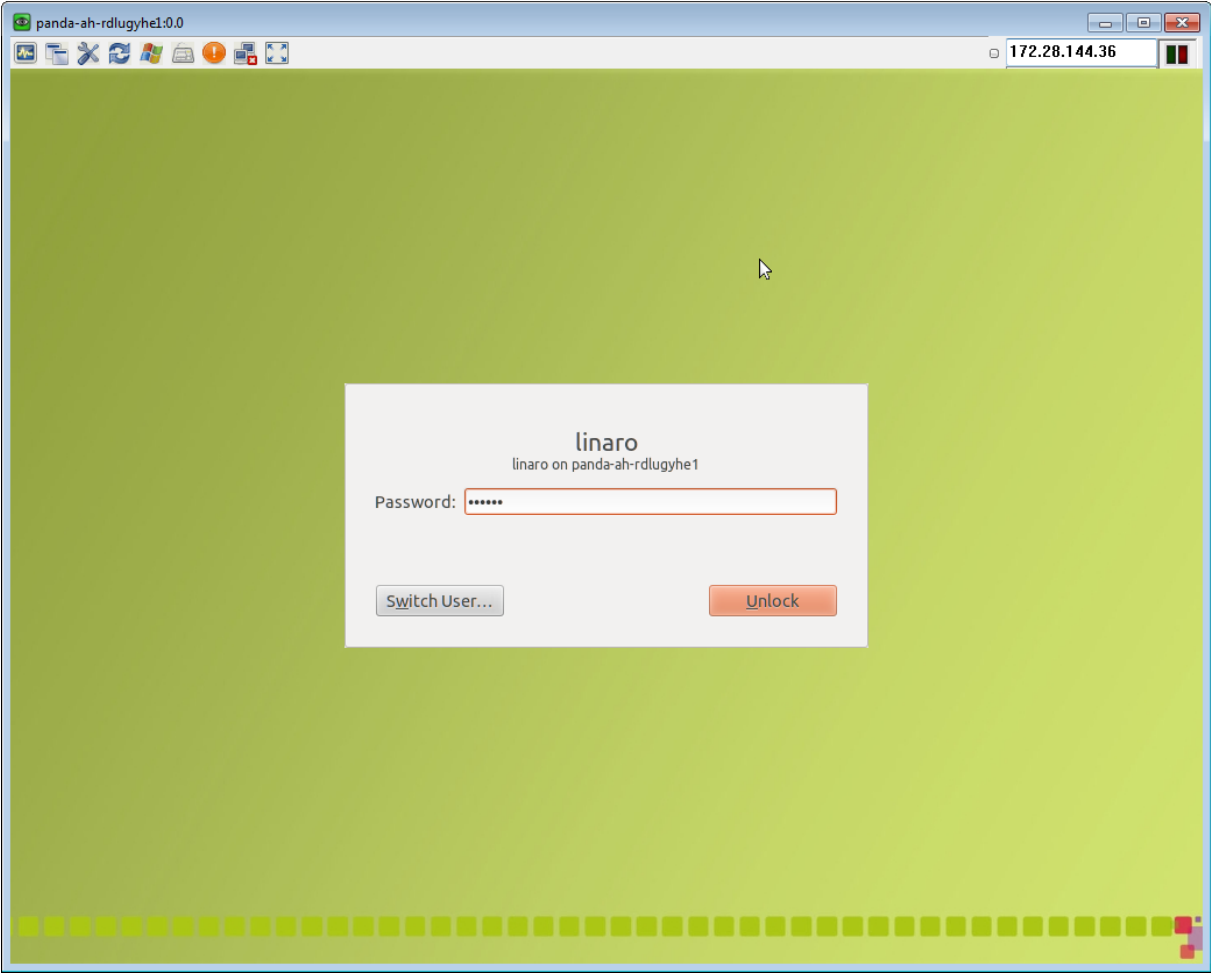
When the VNC server is running, the command line displays:

```
ans =  
  
0
```

- 3 Open a VNC viewer on your host system, and connect to the VNC server on the PandaBoard hardware.



- 4 Log in to the operating system. The default user is `linaro`, and the default password is `linaro`.

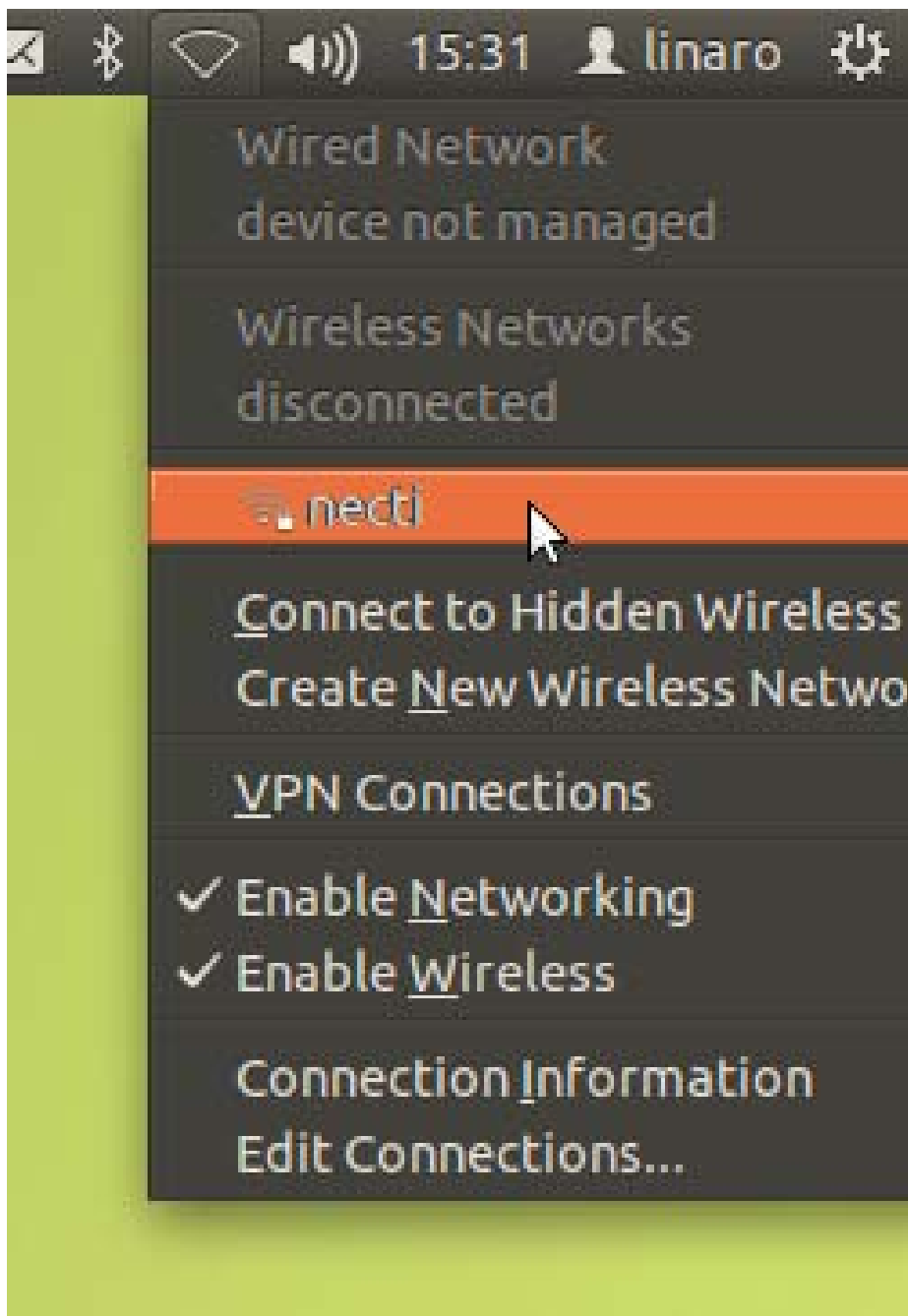


Configure Wi-Fi on PandaBoard Hardware

PandaBoard hardware has built-in 802.11b/g/n wireless networking.

To configure the PandaBoard hardware for wireless network connection, use the Network Manager in the Linux desktop:

- 1** Access the Linux desktop running on the PandaBoard hardware as described in either of the following topics:
 - “Access the Linux Desktop Directly Using Desktop Computer Peripherals” on page 60-41
 - “Access the Linux Desktop Remotely Using VNC” on page 60-43
- 2** On the Linux desktop, open the Network manager, and select a wireless network.



3 If prompted, enter the wireless network password, and click **Connect**.



Raspberry Pi Hardware

- “Install Support for Raspberry Pi™ Hardware” on page 61-2
- “Replace Firmware on Raspberry Pi Hardware” on page 61-3
- “Open a Command-Line Session with Raspberry Pi Hardware” on page 61-16
- “Get IP Address of Raspberry Pi Hardware” on page 61-18
- “Configure Raspberry Pi Hardware to Mail IP Address Changes” on page 61-20
- “Configure Network Settings on Raspberry Pi Hardware” on page 61-23
- “Open Block Library for Raspberry Pi Hardware” on page 61-26
- “Run Model on Raspberry Pi Hardware” on page 61-28
- “Stop or Restart a Model Running on Raspberry Pi Hardware” on page 61-30
- “Tune and Monitor Model Running on Raspberry Pi Hardware” on page 61-31
- “Detect and Fix Task Overruns on Raspberry Pi Hardware” on page 61-34
- “Access the Linux Desktop Using Computer Peripherals” on page 61-35
- “Extend the Linux Partition on the SD Card” on page 61-36
- “Shut Down Raspberry Pi Hardware” on page 61-38

Install Support for Raspberry Pi Hardware

You can add support for Raspberry Pi hardware to the Simulink product. When you complete this process, and replace the firmware, you can run Simulink models on Raspberry Pi hardware.

To install support for Raspberry Pi hardware:

- 1** In a MATLAB Command Window, enter `supportPackageInstaller`. This starts the Support Package Installer.
- 2** Follow the instructions and default settings provided by Support Package Installer to complete the installation. For more information about the options on a particular screen, click the **Help** button.

During this process, Support Package Installer installs:

- Third-party software development tools
- A Simulink block library called *Simulink Support Package for Raspberry Pi Hardware*.
- Examples

These instructions occasionally refer to Raspberry Pi hardware as a *board* or as *target hardware*.

Replace Firmware on Raspberry Pi Hardware

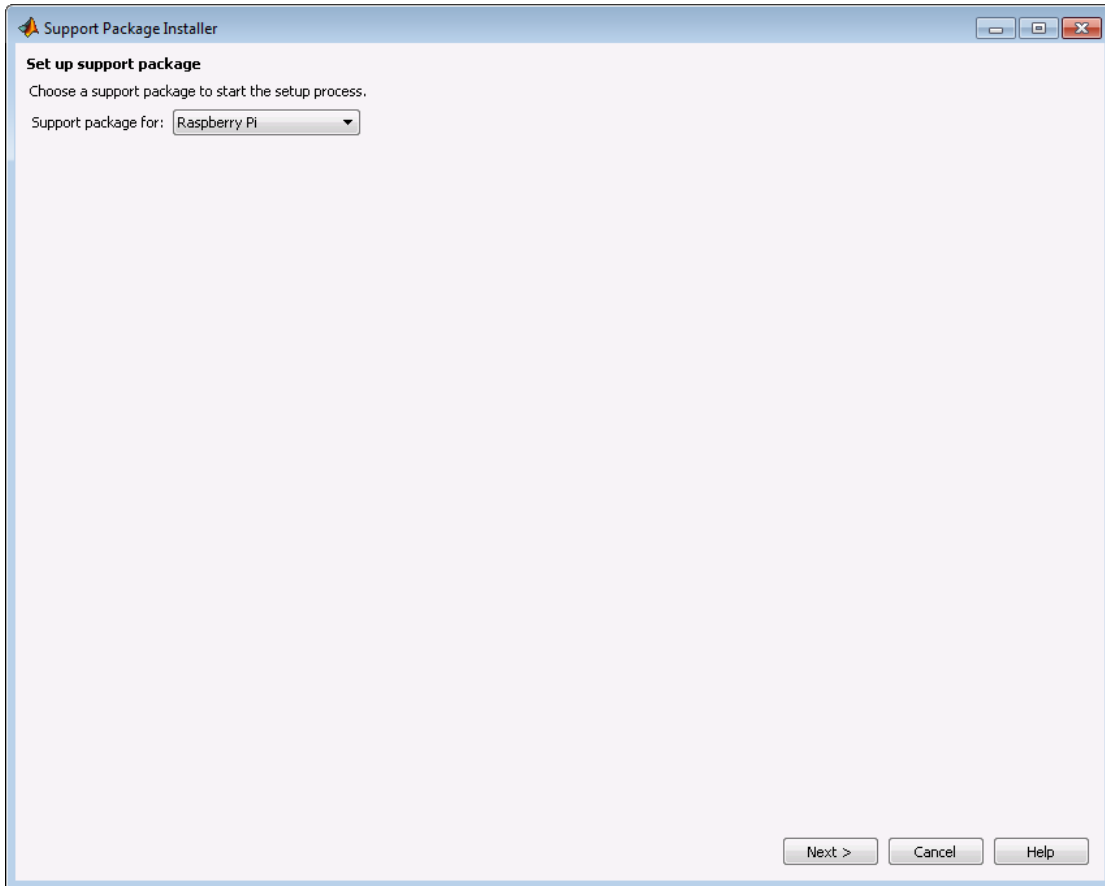
After installing the support package for Raspberry Pi hardware, use Support Package Installer to replace the firmware on the Raspberry Pi hardware.

This process replaces the current firmware with a customized version of Raspbian Wheezy Linux that can run Simulink models, as described in the following topics:

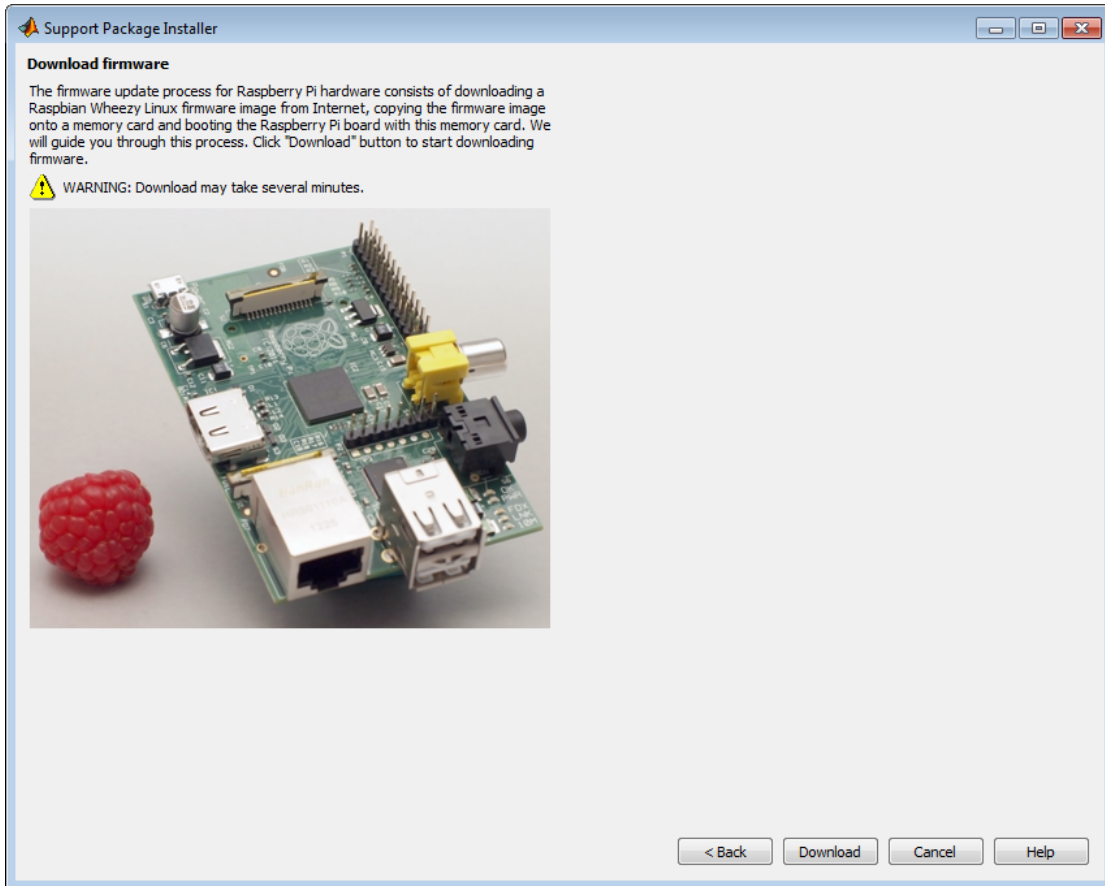
- “Run Model on Raspberry Pi Hardware” on page 61-28
- “Tune and Monitor Model Running on Raspberry Pi Hardware” on page 61-31

To replace the firmware on your Raspberry Pi hardware:

- 1** Open the **Update firmware** screen in Support Package Installer:
 - Click **Continue** in the **Install/update complete** screen of Support Package Installer.
 - In a model, select **Tools > Run on Target Hardware > Update firmware**.
 - In a MATLAB Command Window, enter `targetupdater`.
- 2** Choose the Raspberry Pi option, and click **Next**.

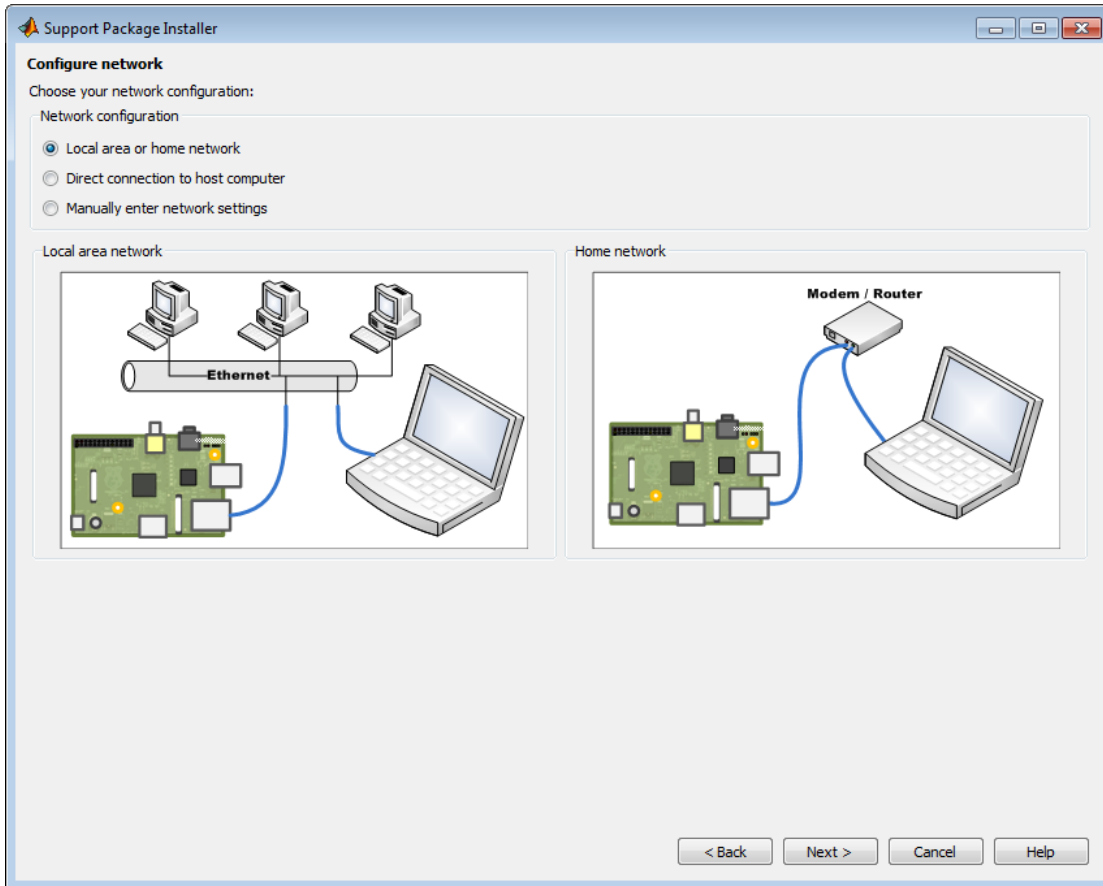


- 3 Review the information, and click **Download**. The firmware download starts automatically and takes several minutes.

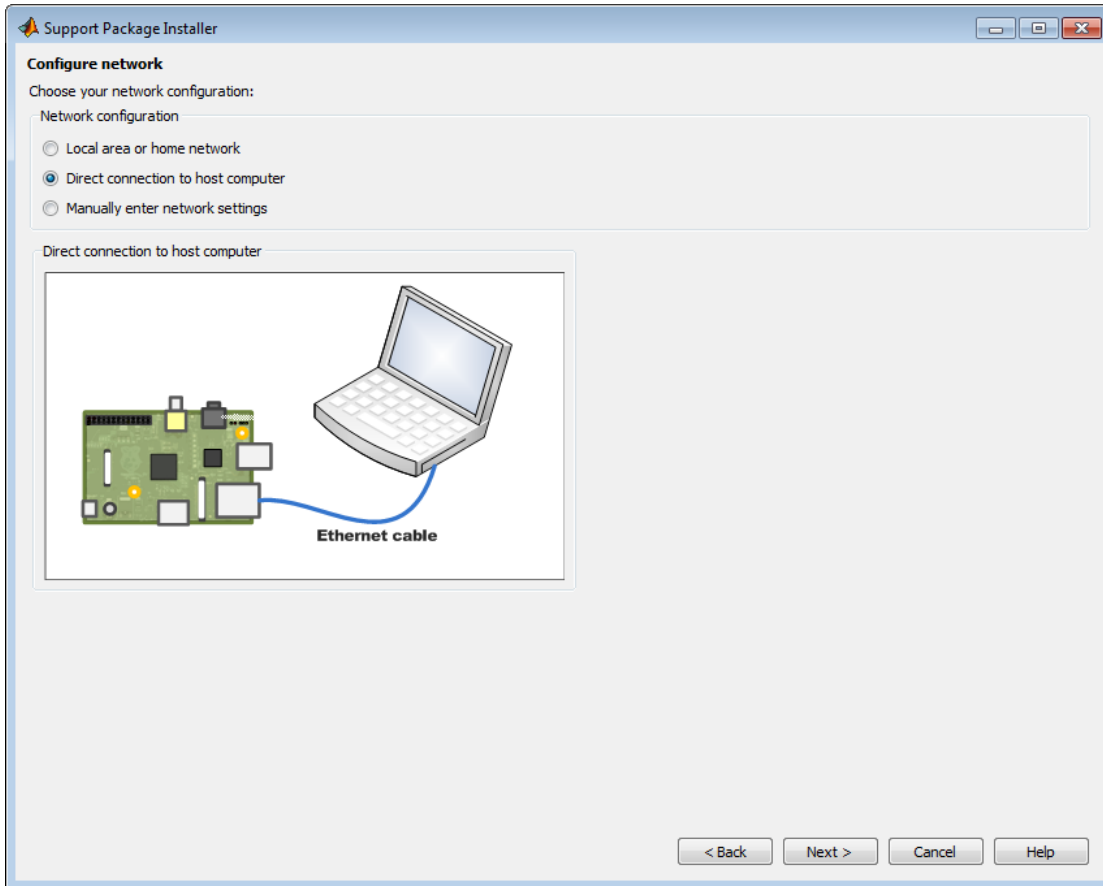


4 Choose your network configuration:

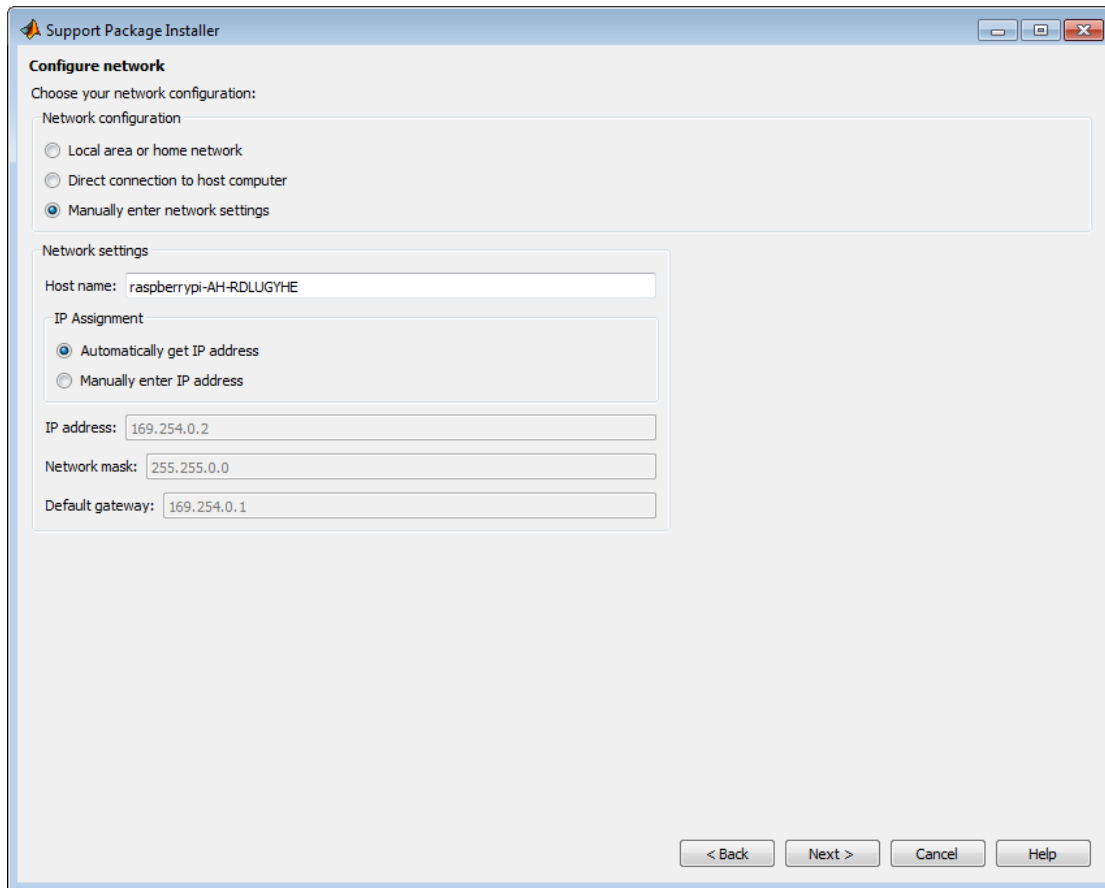
- **Local area or home network** — Confirm that the Raspberry Pi hardware is connected to your host computer using a local area network (LAN) or home network similar to the ones displayed by Support Package Installer. Then, click **Next**. This option automatically applies dynamic network settings provided by a DNS service on the network.



- **Direct connection to host computer** — Confirm that the Raspberry Pi hardware is connected to your host computer using a direct connection similar to the one displayed by Support Package Installer. Then, click **Next**. This option automatically applies static network settings based upon the network settings of the host computer.



- **Manually enter network settings** — To manually configure the network settings, select this option.



Selecting **Manually enter network settings** displays the following options:

- **Host name** — This parameter displays an automatically-generated network host name that will be assigned to the Raspberry Pi hardware. If multiple boards are connected to the network, edit the host name to make it unique.
- **Automatically get IP address** — Like **Local area or home network**, this option automatically applies dynamic network settings provided by a DNS service on the network.

- **Manually enter IP address** — Use this option to edit the **IP address**, **Network mask**, and **Default gateway** settings.

Guidelines for manually entering IP settings on the Raspberry Pi hardware:

- The value of the **IP address** must be unique for each device on the network.
- The value of the **Network mask** must be the same for all devices on the network. This value is also known as **Subnet mask**.
- The value of the **Default gateway** is usually the same for all devices on the network.

For example, to manually configure IP settings, enter `ipconfig` on the command line of your host computer. This command displays network settings of Ethernet adapters on the host computer. Suppose that, for the Ethernet adapter on the host computer that is connected to the Raspberry Pi hardware, the values are:

```
IPv4 Address . . . . . : 192.168.1.2
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.1
```

In that case, for the Raspberry Pi hardware:

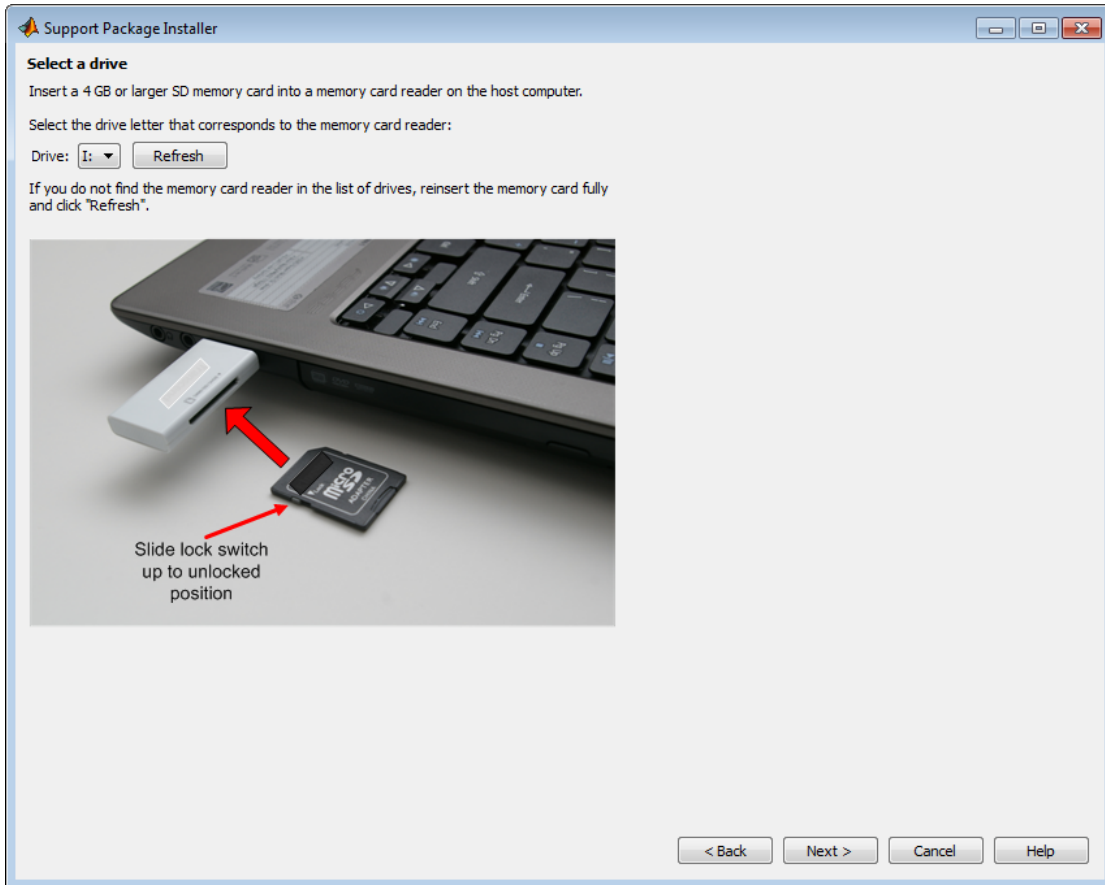
- Set **IP address** to an unused IP address, between 192.168.1.3 and 192.168.1.254
- Set **Network mask** to use the same network mask value, 255.255.255.0

For more information, see “Get IP Address of Raspberry Pi Hardware” on page 61-18 and “Configure Network Settings on Raspberry Pi Hardware” on page 61-23.

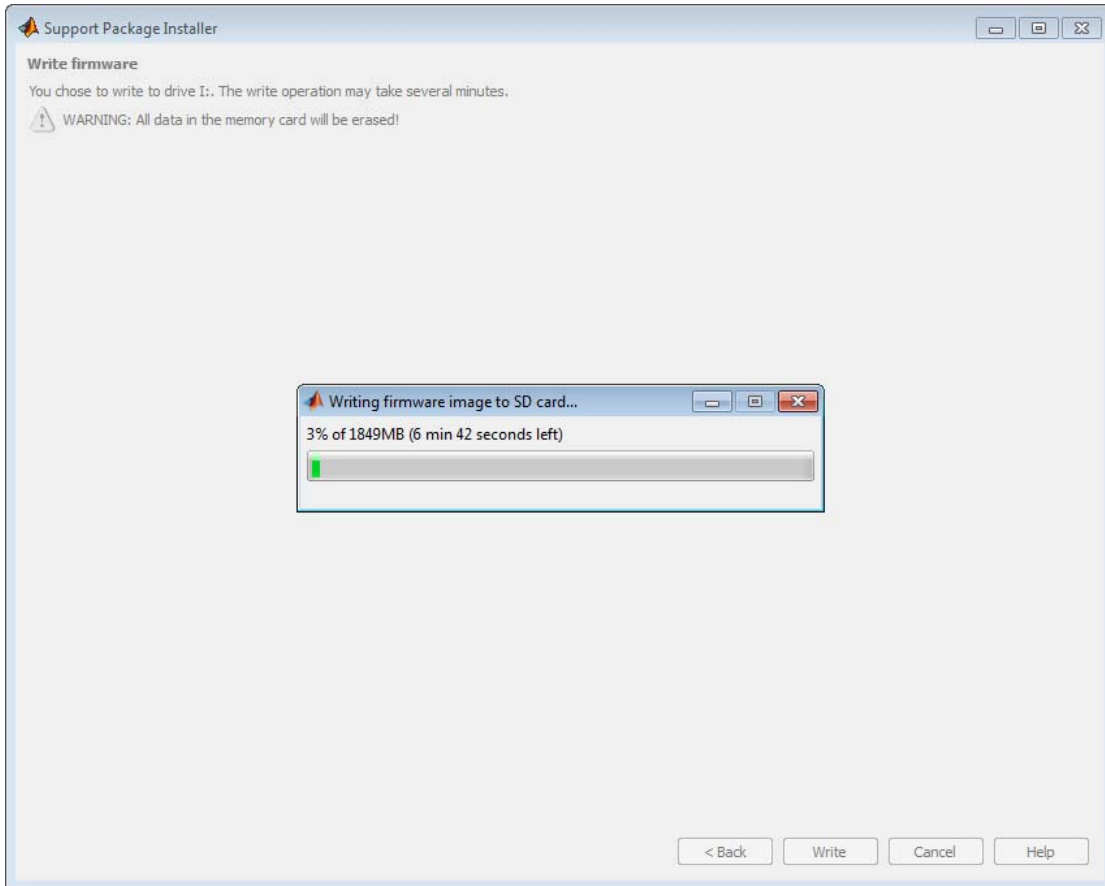
- 5 Insert the SD card into a media card reader connected to your host computer. Windows assigns a drive letter to the memory card.

Click **Refresh**, and select the drive letter of the SD card and click **Next**.

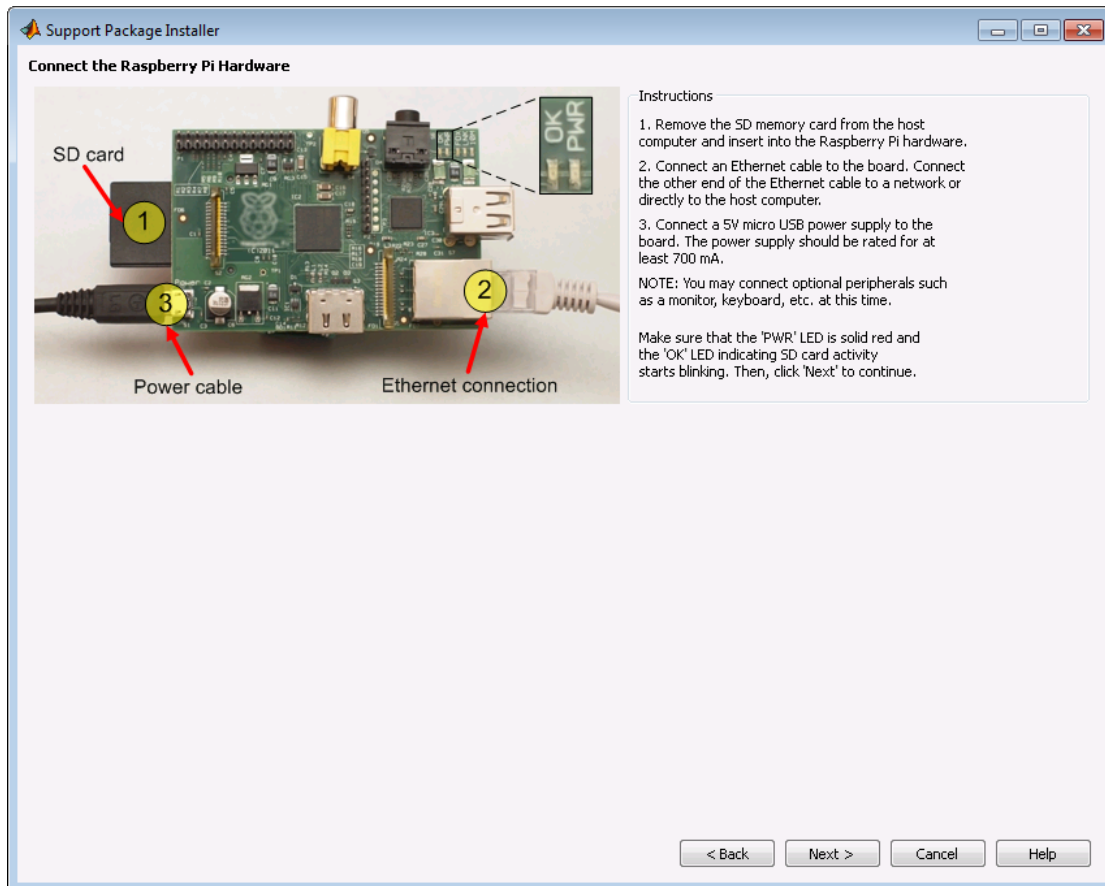
Tip If multiple drive letters are available, identify the drive letter of the media card reader. Open the Windows Start menu, choose **Computer**, and review the list of **Devices with Removable Storage**



- 6 Click **Write**. Support Package Installer overwrites all previous data on the memory card with the firmware. This process takes several minutes to complete.



- 7 Make the connections shown in the figure, and check the LEDs labelled OK or ACT, and PWR. Then, click **Next**.



8 Click **Test Connection**.

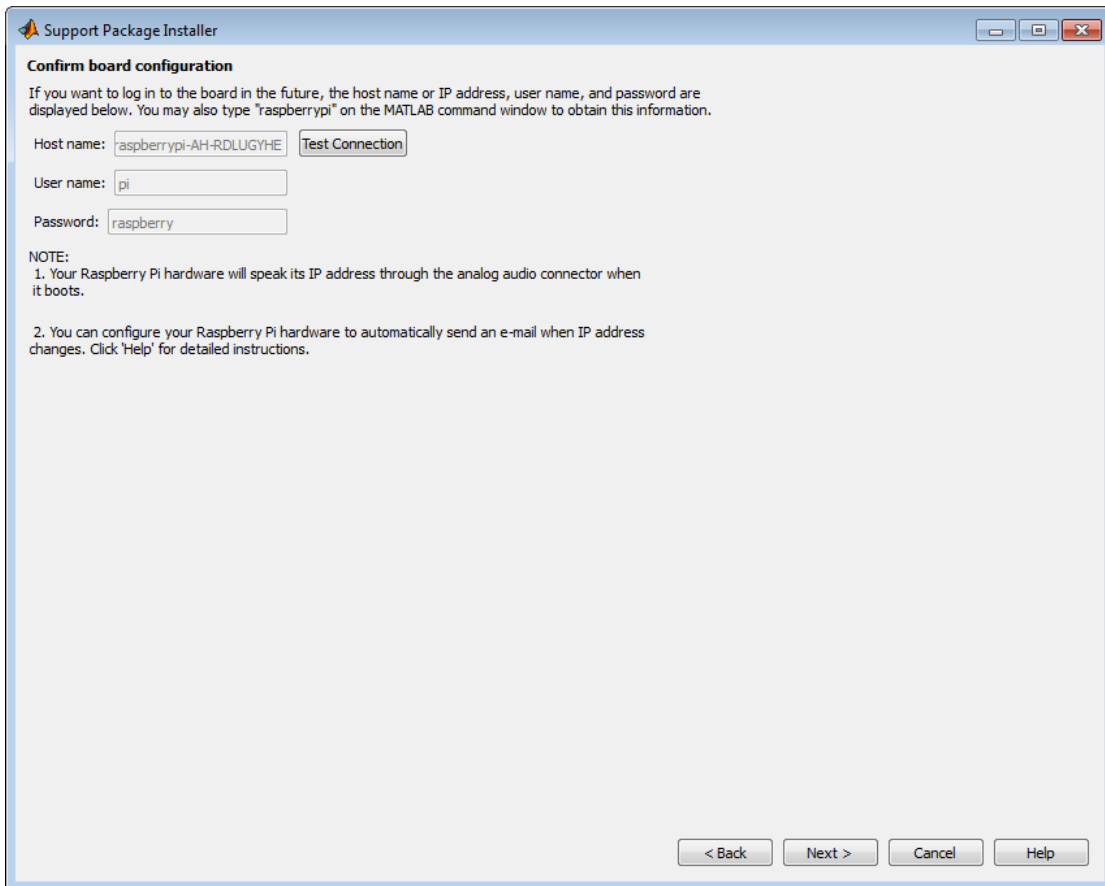
Support Package Installer uses SSH to create a test connection to the board using the host name, user name, and password shown.

- If the test fails, the software displays a message that it cannot connect to the hardware, and includes the error reported by PuTTY. You can click **Back** or use the `targetupdater` function to revisit the network settings. Also see “Configure Network Settings on Raspberry Pi Hardware” on page 61-23.

- If the test succeeds, the software displays a “Connection successful” message on the progress bar for a few moments before closing the progress bar.

Make a note of the host name, user name, and password so you can log in to the board later on. Then, click **Next**.

If the IP address of the Raspberry Pi hardware changes, the board can send you email notification of the new IP address. For more information, see “Configure Raspberry Pi Hardware to Mail IP Address Changes” on page 61-20.



The screenshot shows a window titled "Support Package Installer" with a standard Windows-style title bar (minimize, maximize, close buttons). The main content area is titled "Confirm board configuration" and contains the following text:

If you want to log in to the board in the future, the host name or IP address, user name, and password are displayed below. You may also type "raspberrypi" on the MATLAB command window to obtain this information.

Host name:

User name:

Password:

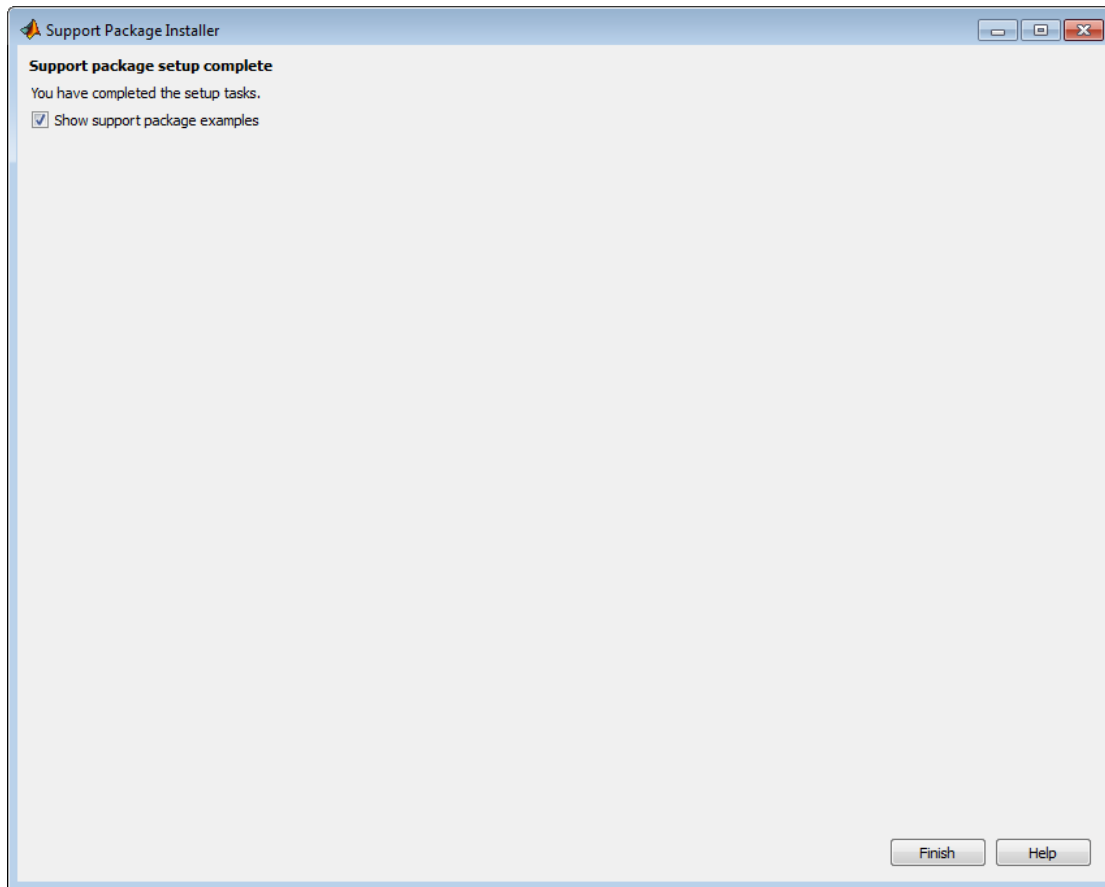
NOTE:

1. Your Raspberry Pi hardware will speak its IP address through the analog audio connector when it boots.
2. You can configure your Raspberry Pi hardware to automatically send an e-mail when IP address changes. Click 'Help' for detailed instructions.

At the bottom of the window, there are four buttons: "< Back", "Next >", "Cancel", and "Help".

9 Click **Finish**.

If you leave **Show support package examples** enabled, Support Package Installer opens the example page for Raspberry Pi hardware.



10 For hands-on experience using Simulink models with Raspberry Pi hardware, complete the examples.

To reopen these examples later on, see “Open Examples for This Support Package” on page 55-4.

The screenshot shows a web browser window displaying the 'Simulink Support Package for Raspberry Pi Hardware EXAMPLES' page. The page title is 'Simulink Support Package for Raspberry Pi Hardware EXAMPLES'. Below the title, there is a brief description: 'Simulink® lets you design and run models on Raspberry Pi® hardware. With this capability, you can assess and optimize algorithms in the classroom or lab as they execute in real-time on Raspberry Pi hardware with physical I/O.' A link to the 'Product page at mathworks.com' is provided.

The page is organized into two main sections: 'Tutorials' and 'Examples'.

Tutorials:

- Getting Started with Raspberry Pi Hardware:** Includes a small image of a Raspberry Pi and a 'Model' icon.
- Communicating with Raspberry Pi Hardware:** Includes a small image of a Raspberry Pi and a 'Model' icon.
- Remote Execution of Linux Commands:** Includes a small image of a Raspberry Pi and a 'Script' icon.

Examples:

- Parametric Audio Equalizer:** Includes a block diagram of an audio equalizer and the text 'Parametric Audio Equalizer Uses: DSP System Toolbox'. It has a 'Model' icon.
- Image Inversion:** Includes a block diagram of an image inversion process and the text 'Image Inversion'. It has a 'Model' icon.
- Motion Sensor:** Includes a block diagram of a motion sensor and the text 'Motion Sensor'. It has a 'Model' icon.

Open a Command-Line Session with Raspberry Pi Hardware

This topic shows you how to configure and open a command-line session with the target hardware.

To open a connection to your board:

- If you configured your Raspberry Pi to automatically obtain an IP address using DHCP, communicate with your board using the host name you assigned to the board during the firmware update process.

For example, if the board is named `raspberrypi-ah`, enter the following in a MATLAB Command Window:

```
h = raspberrypi('raspberrypi-ah')
h.openShell('ssh')
```

- If you configured your Raspberry Pi to use a static IP address during the firmware update process, use that IP address to communicate with your board.

For example, if the board's IP address is `192.168.1.2`, enter the following in a MATLAB Command Window:

```
h = raspberrypi('192.168.1.2')
h.openShell('ssh')
```

1 Connect an Ethernet cable from your host computer to the board.

2 Communicate with the board:

- If you configured your Raspberry Pi to automatically obtain an IP address using DHCP, communicate with your board using the host name you assigned to the board during the firmware update process.

For example, if the board is named `raspberrypi-ah`, enter the following in a MATLAB Command Window:

```
h = raspberrypi(`raspberrypi-ah')
h.openShell('ssh')
```

- If you configured your Raspberry Pi to use a static IP address during the firmware update process, use that IP address to communicate with your board.

For example, if the board's IP address is 192.168.1.2, enter the following in a MATLAB Command Window:

```
h = raspberrypi(`192.168.1.2')
h.openShell('ssh')
```

- 3** When a PuTTY terminal window opens, press the **Enter** key on your keyboard. The terminal window displays the Linux command line.
- 4** At the Linux command line, log in by entering the user name and password. For example:

```
raspberrypi-boardname login: pi
Password: raspberry
```

- 5** When you are finished, enter `logout` on the Linux command line, and close PuTTY to end the serial connection.

See Also

- “Get IP Address of Raspberry Pi Hardware” on page 61-18
- For more information about PuTTY, see:
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

Get IP Address of Raspberry Pi Hardware

You can use your Raspberry Pi hardware in one of the following configurations:

- Monitor and keyboard attached
- Headless (no monitor or keyboard)

If your Raspberry Pi hardware is connected to a keyboard and monitor, you can find the IP address of the board. To do so, open a Linux shell, and enter:

```
ifconfig
```

The `ifconfig` command lists the network information for all Ethernet interfaces on your board. For example:

```
eth0      Link encap:Ethernet  HWaddr b8:27:eb:63:40:b8
          inet addr:172.28.145.190  Bcast:172.28.145.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:27224 errors:0 dropped:0 overruns:0 frame:0
          TX packets:733 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2801074 (2.6 MiB)  TX bytes:107019 (104.5 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

The wired Ethernet connection on your Raspberry Pi hardware corresponds to `eth0` network interface. The IP address of `eth0` is listed in the `inet addr` field.

If you are using your Raspberry Pi hardware in headless configuration, there are several ways to find the IP address:

- Raspberry Pi configured to automatically obtain IP address using DHCP
 - You can communicate with your board using the host name you assigned

to the board during the firmware update process. This approach does not work if you configured your board to use a static IP address.

For example, if the board is named `raspberrypi-ah`, enter the following in a MATLAB Command Window:

```
h = raspberrypi(`raspberrypi-ah')  
h.openShell(`ssh')
```

- A feature called speak IP, which MathWorks software adds to Raspian Wheezy. This feature speaks the IP address of the Raspberry Pi hardware at boot time, just after acquiring an IP address. To hear the IP address, connect headphones to the Audio Output port when booting the board.
- Configure Raspberry Pi hardware to use Dynamic DNS (DDNS) — Dynamic DNS is a method of updating the Domain Name System to point to a changing IP address on the Internet. Dynamic DNS requires a DDNS server. DDNS services, such as the ones provided by dnsdynamic.org, are available from the Internet. Instructions for setting up a dynamic DNS client on Raspberry Pi hardware can be found on Internet.
- Configure the board to send you an email message when the IP address changes — See “Configure Raspberry Pi Hardware to Mail IP Address Changes” on page 61-20.

Configure Raspberry Pi Hardware to Mail IP Address Changes

If the IP address of the Raspberry Pi hardware changes, the board can send you an email notification of the new IP address. To enable this feature, add your email address to the `ssmtp.conf` file on the hardware.

Before performing this process, connect to the Linux desktop on the Raspberry Pi hardware using a monitor, keyboard, and mouse, as described in “Access the Linux Desktop Using Computer Peripherals” on page 61-35.

The Raspbian Wheezy image is configured to include the `ssmtp` package, which enables you to send email from the Raspberry Pi hardware. The `ssmtp` package is a simple mail transfer agent (MTA) that requires an authenticated email server. For simplicity, `ssmtp.conf` is configured to use a Gmail™ server. The configuration file for the `ssmtp` is located in `/etc/ssmtp/ssmtp.conf` file on the Raspberry Pi hardware:

```
#
# Config file for sSMTP sendmail
#
# The person who gets all mail for userids < 1000
# Make this empty to disable rewriting.
root=postmaster

# The place where the mail goes. The actual machine name is required no
# MX records are consulted. Commonly mailhosts are named mail.domain.com
mailhub=smtp.gmail.com:587

# Where will the mail seem to come from?
rewriteDomain=gmail.com

# The full hostname
hostname=gmail.com

# Are users allowed to set their own From: address?
# YES - Allow the user to specify their own From: address
# NO - Use the system generated From: address
FromLineOverride=YES
```



```
AuthUser=pi
```

```
AuthPass=raspberry
```

```
UseSTARTTLS=YES
```

To update the `ssmtp.conf` file:

1 Open a command line session, as described in “Open a Command-Line Session with Raspberry Pi Hardware” on page 61-16.

2 Use a simple editor called nano to edit `ssmtp.conf`. Enter:

```
sudo nano /etc/ssmtp/ssmtp.conf
```

3 Modify the `AuthUser` and `AuthPass` entries in `ssmtp.conf` with your Gmail account user name and the password.

4 Save the changes and exit nano:

a Press **Ctrl+X**.

b Enter **Y** to save the modified buffer.

c Respond to File Name to Write: `/etc/ssmtp/ssmtp.conf` by pressing **Enter**.

The nano editor displays Wrote # lines and returns control to the command line.

5 Edit the `mailip` file in nano by entering:

```
sudo nano /etc/network/if-up.d/mailip
```

6 Update the `RECIPIENTADDR` entry in `mailip` with your email address.

7 Save the changes and exit nano:

a Press **Ctrl+X**.

b Enter **Y** to save the modified buffer.

- c Respond to File Name to Write: `/etc/network/if-up.d/mailip` by pressing **Enter**.

The nano editor displays `Wrote # lines` and returns control to the command line.

- 8 Enable the auto-email feature by entering the following command on a Linux shell on your Raspberry Pi hardware:

```
sudo chmod ugo+x /etc/network/if-up.d/mailip
```

- 9 Test the auto-email feature by entering the following on a Linux shell on your Raspberry Pi hardware:

```
sudo /etc/network/if-up.d/mailip
```

If the test is successful, auto-email sends you a message with the IP address of the Raspberry Pi hardware on the subject line. The message body includes the same network information one gets from using `ifconfig`.

To use a mail server other than the Gmail server, update the `mailhub`, `rewriteDomain`, `hostname`, and `UseSTARTTLS` entries in `ssmtp.conf`. Specify the values required by the new email server.

Configure Network Settings on Raspberry Pi Hardware

You can resolve network connectivity issues by inspecting and editing the IP configuration of the Raspberry Pi Ethernet port.

To configure these settings when you are first setting up your Raspberry Pi hardware, see “Replace Firmware on Raspberry Pi Hardware” on page 61-3.

You may need to reconfigure the IP settings if your board:

- Has unknown IP settings
- Is unreachable using a network connection
- Is being moved to a network or direct Ethernet connection that uses static IP settings
- Is being moved from a network that used static IP settings to one that uses DHCP services

There are several conditions under which networks use DHCP or static IP settings:

- Use DHCP services — If your board is connected to a network with DHCP services, such as an office LAN or a home network connected to the Internet. DHCP is a network service that automatically configures the IP settings of Ethernet devices connected to a network.
- Use static IP settings — If your board is directly connected to an Ethernet port on your computer or connected to an isolated network without DHCP services.

To configure the board to use DHCP or static IP settings:

- 1** You can use a terminal window after accessing the Linux desktop. See “Access the Linux Desktop Using Computer Peripherals” on page 61-35.
- 2** Display the contents of the `/etc/network/interfaces` file. Enter:

```
cat /etc/network/interfaces
```

If the board is configured to use DHCP services (the default configuration), `dhcp` appears at the end of the following line:

```
iface eth0 inet dhcp
```

If the board is configured to use static IP settings, `static` appears at the end of the following line:

```
iface eth0 inet static
```

- 3** Create a backup of the `/etc/network/interfaces` file. Enter:

```
sudo cp /etc/network/interfaces /etc/network/interfaces.backup
```

If prompted, enter the root password.

- 4** Edit `interfaces` using a simple editor called `nano`. Enter:

```
sudo nano /etc/network/interfaces
```

- 5** Edit the last word of line that starts with `iface eth0 inet`.

To use DHCP services, change the line to:

```
iface eth0 inet dhcp
```

To use static IP settings, change the line to:

```
iface eth0 inet static
```

- 6** For static IP settings, add lines for address, netmask, and gateway. For example:

```
iface eth0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    gateway 192.168.1.1
```

For static IP settings:

- The value of the subnet mask must be the same for all devices on the network.

- The value of the IP address must be unique for each device on the network.

For example, if the Ethernet port on your host computer has a network mask of 255.255.255.0 and a static IP address of 192.168.1.1, set:

- `netmask` to use the same network mask value, 255.255.255.0.
- `address` to an unused IP address, between 192.168.1.2 and 192.168.1.254.

7 Save the changes and exit nano:

- a** Press **Ctrl+X**.
- b** Enter **Y** to save the modified buffer.
- c** For “File Name to Write: /etc/network/interfaces”, press **Enter**.
- d** The nano editor confirms that it “Wrote # lines” and returns control to the command line.

8 Reboot the board. In MATLAB Command Window, enter:

```
h = raspberrypi
h.execute('sudo shutdown -r now')
```

9 Test the IP settings by logging in to the board over a telnet session.

Tip You can use the `ifconfig` command to temporarily change the IP settings. Rebooting the board removes the `ifconfig` settings and restores the `/etc/network/interfaces` settings.

To change the IP settings temporarily, open a Linux command line. Enter `ifconfig`, the device id, a valid IP address, `netmask`, and the appropriate network mask. For example:

```
ifconfig eth0 192.168.45.12 netmask 255.255.255.0
```

Related Examples

- “Run Model on Raspberry Pi Hardware” on page 61-28
- “Get IP Address of Raspberry Pi Hardware” on page 61-18

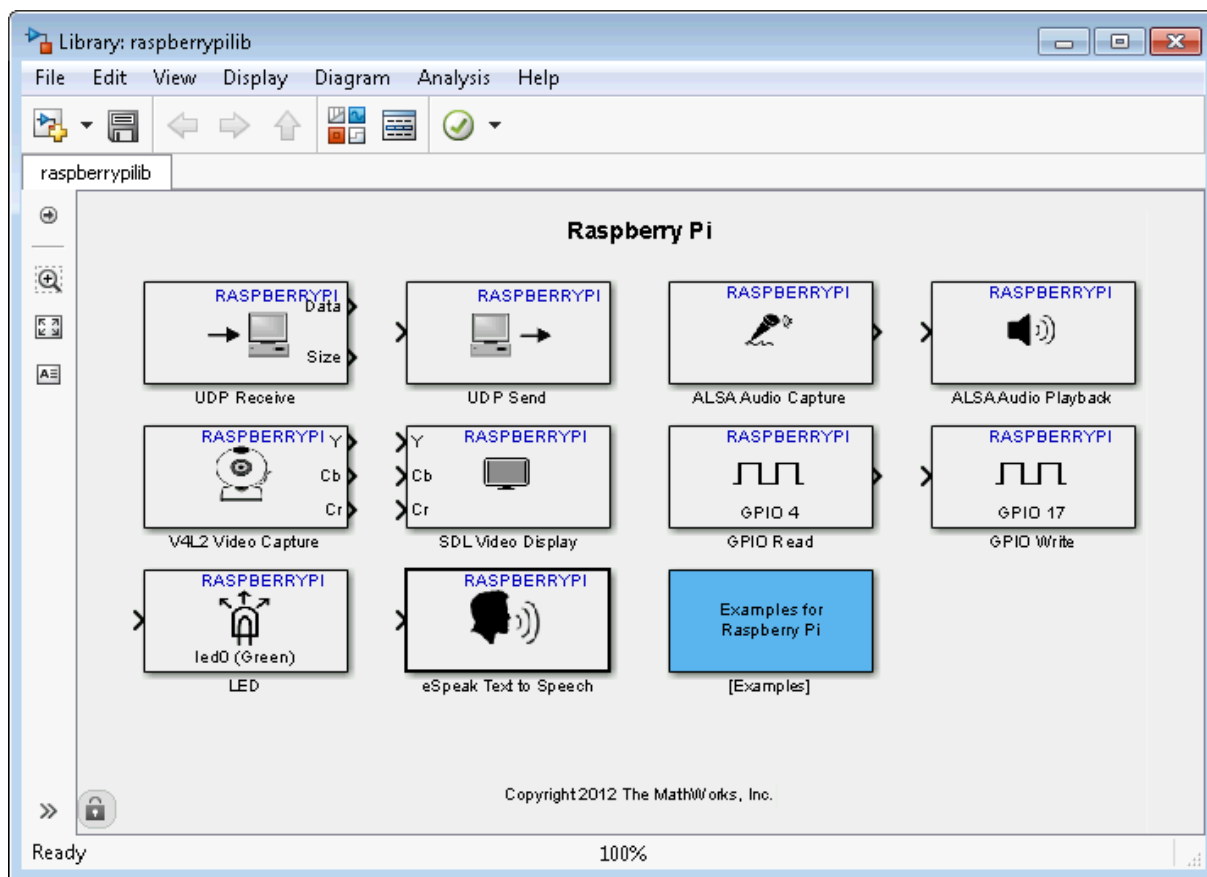
Open Block Library for Raspberry Pi Hardware

This block library provides support for protocols and APIs available through Raspberry Pi hardware.

To open the block library from the MATLAB Command Window, enter:

```
rasberrypilib
```

The following block library opens.



To open the block library from the Simulink Library Browser:

- 1 Enter `simulink` in the MATLAB Command Window, or click the following icon on the MATLAB toolbar.



- 2 When the Simulink Library Browser opens, click **Simulink Support Package for Raspberry Pi Hardware**.

Run Model on Raspberry Pi Hardware

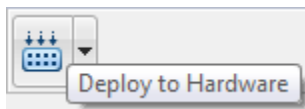
You can prepare, configure, and run a model on your Raspberry Pi hardware.

Before starting this procedure:

- Connect your board to the network and to a power supply.
- Create or open a Simulink model.

To prepare and run the model:

- 1** Use **File > Save As** to create a working copy of your model. Keep the original model as a backup copy.
- 2** In your model, select **Tools > Run on Target Hardware > Prepare to Run** to change the model Configuration Parameters.
- 3** When the Run on Target Hardware pane opens, set the **Target hardware** parameter to **Raspberry Pi**.
- 4** If you have changed boards since the last time you updated the firmware or ran a model, update the **Host name**, **User name**, and **Password** parameters.
- 5** Click the **Deploy to Hardware** button.



This action automatically downloads and runs your model on the board.

Note Pressing RESET or cycling the power on the Raspberry Pi hardware during this step can cause the ssh utility used for downloading the model to hang.

You press RESET during this stage and are subsequently unable to download your model, shut down the hung ssh process using your host computer's task manager, or reboot your host computer.

Running a new or updated model on the board:

- Automatically stops a running model with the same name.
- Does not stop running models that have other names.

Caution Do not use the RESET button to stop a running model and reboot the board. Doing so can corrupt operating system and program files. To repair corrupt files, replace the firmware as described in “Replace Firmware on Raspberry Pi Hardware” on page 61-3.

See Also

- “Create the Simple Model”
- “Configure Network Settings on Raspberry Pi Hardware” on page 61-23
- “Tune and Monitor Model Running on Raspberry Pi Hardware” on page 61-31
- “Overview of Model Referencing” on page 6-2

Stop or Restart a Model Running on Raspberry Pi Hardware

Caution Do not use the RESET button to stop a running model and reboot the board. Doing so can corrupt operating system and program files. To repair corrupt files, replace the firmware as described in “Replace Firmware on Raspberry Pi Hardware” on page 61-3.

To stop a model running on the board, in the MATLAB Command Window, enter:

```
h=raspberrypi;  
h.stop('modelName');
```

For example, to stop the sumdiff model, enter:

```
h=raspberrypi;  
h.stop('sumdiff');
```

To restart a model that was previously running on the board or run multiple instances of a model, in the MATLAB Command Window, enter:

```
h=raspberrypi;  
h.run('modelName');
```

Note You do not need to enter `h=raspberrypi;` multiple times if a previous instance of `h` is available in the MATLAB Workspace.

For example, to restart the sumdiff model you stopped in the previous example, enter:

```
h.run('sumdiff');
```

Tune and Monitor Model Running on Raspberry Pi Hardware

In this section...

“Overview of Using External Mode” on page 61-31

“Run Your Simulink Model in External Mode” on page 61-32

“Stop External Mode” on page 61-33

Overview of Using External Mode

You can use External mode to tune parameters and monitor a model running on your target hardware.

External mode enables you to tune model parameters and evaluate the effects of different parameter values on model results in real time. Doing so helps you find the optimal values to achieve desired performance. This process is called *parameter tuning*.

External mode accelerates parameter tuning because you do not have to rerun the model each time you change parameters. You can also use External mode to develop and validate your model using the actual data and hardware for which it is designed. This software-hardware interaction is not available solely by simulating a model.

This workflow lists the tasks usually required to tune parameters with External mode:

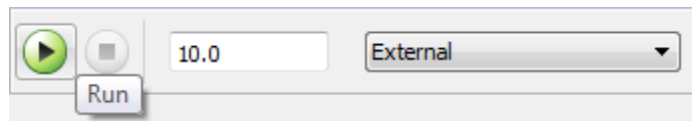
- 1** In the model on your host computer, enable External mode.
- 2** (Optional) Place one or more sink blocks in your model. For example, use Display or Scope blocks to visualize data, or use a To File block to log signal data.
- 3** Give the Simulink software command to run the model on the target hardware.
- 4** (Optional) Observe the data External mode sends from the target hardware to sink blocks in the model on the host computer.

- 5 Change and apply parameter values in the model on your host computer.
- 6 Find the optimal parameter values by making adjustments and observing the results.
- 7 Save the new parameter values, disable External mode, and save the model.

Run Your Simulink Model in External Mode

Note If you have the Embedded Coder or Simulink Coder software, you can use External mode with a model that contains Model blocks (uses the “Model reference”).

- 1 Create a network connection between the Raspberry Pi hardware and your host computer. See “Configure Network Settings on Raspberry Pi Hardware” on page 61-23.
- 2 On the model toolbar, set **Simulation mode** to External mode.



- 3 Set the **Simulation stop time** parameter, located to the left of **Simulation mode** on the model toolbar. The default value is 10.0 seconds. To run the model for an indefinite period, enter `inf`.
- 4 Click the **Run** button.

If your model does not contain a sink block, the MATLAB Command Window displays a warning message. For example:

```
Warning: No data has been selected for uploading.
> In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\+realtime\extModeAutoConnect.p>
extModeAutoConnect at 17
In C:\Program Files (x86)\MATLAB\R2013a Student1\toolbox\
realtime\realtime\sl_customization.p>myRunCallback at 149
```

You can disregard this warning or add a sink block to the model.

After several minutes, Simulink starts running your model on the board.

5 Change parameter values in the model on your host computer.

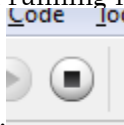
Observe the corresponding changes in the model running on the hardware.

Any Simulink Sinks blocks in your model receive data from the hardware and display it on your host computer.

Note External mode increases the processing burden of the model running on the board. If the software reports an overrun, disable External mode.

Stop External Mode

To stop the model running in External mode, click the Stop button located on



the model toolbar.

If the **Simulation stop time** parameter is set to a specific number of seconds, External mode stops when that time elapses.

When you are finished using External mode, set **Simulation mode** back to Normal.

Detect and Fix Task Overruns on Raspberry Pi Hardware

You can configure a model running on the target hardware to detect and notify you of task overruns.

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. A task overrun occurs when a task is scheduled to execute before a previous instance of the same task has completed.

To enable overrun detection:

- 1** In your model, click **Tools > Run on Target Hardware > Options**.
- 2** When the Run on Target Hardware opens, select the **Enable overrun detection** check box.
- 3** Click **OK**.

When a task overrun occurs, the command prompt on the host machine repeatedly displays an error message until the model stops. For example, such a message might read “Overrun — rate for subrate task 1 is too fast”.

When an overrun condition occurs, you can correct it using one or both of the following approaches:

- Simplify the model.
- Increase the sample times for the model and the blocks in it. For example, change the **Sample time** parameter in all of your data source blocks, such as blocks for input devices, from 0.1 to 0.2.

Note External mode increases the processing burden of the model running on your board. If the software reports an overrun, disable External mode.

Access the Linux Desktop Using Computer Peripherals

You can access the Linux desktop on the Raspberry Pi hardware using hardware peripherals such as a monitor, keyboard, and mouse.

To connect to the Raspberry Pi hardware to your hardware peripherals:

- 1** Connect a computer monitor to the HDMI output on the Raspberry Pi hardware.
- 2** Connect a powered USB hub to the USB port on the Raspberry Pi hardware.
- 3** Connect a mouse and keyboard to the powered USB hub.
- 4** Connect computer speakers to either of the two 3.5 mm audio connectors on the Raspberry Pi hardware.
- 5** Connect a network cable to the Ethernet port.
- 6** Connect the Raspberry Pi hardware to the power supply.
- 7** Log in to the Linux desktop using the user name and password.

Extend the Linux Partition on the SD Card

The firmware update for Raspberry Pi hardware burns a Raspbian Wheezy Linux image on an SD card. This Linux image uses approximately 3.5 GB of the SD card. If you use an SD card whose size is greater than 3.5 GB, some unused space remains on the SD card. To use the full capacity of the SD card, you can use the “raspi-config” program available with Raspbian Wheezy.

- 1 Enter the following commands in the MATLAB Command Window:

```
h = raspberrypi
h.openShell('ssh')
```

This command sequence launches a PuTTY SSH shell to your Raspberry Pi hardware.

- 2 Log in to the Raspberry Pi hardware with your user name and password.

- 3 Enter the following command in the SSH command shell:

```
sudo raspi-config
```

- 4 On the raspi-config menu, select the move `expand_rootfs` option, and press **Enter**.

- 5 Press **Enter** when the menu displays: Root partition has been resized. The filesystem will be enlarged upon the next reboot. <OK>

- 6 On the raspi-config menu, select the Finish option, and press **Enter**.

- 7 Enter the following command in the SSH command shell:

```
sudo shutdown -r now
```

- 8 Press **Enter** when the menu displays: Would you like to reboot now? <Yes> <No>

- 9 When the board finishes rebooting, raspi-config automatically resizes the Linux partition on the SD card to occupy the remaining space.

The boot process takes longer than usual for larger SD cards.

Shut Down Raspberry Pi Hardware

Your Raspberry Pi hardware is a mini-Linux computer. Disconnecting the power cable without properly shutting down can corrupt the SD card and cause Linux boot failures. To avoid this problem, shut down your Raspberry Pi hardware using Linux commands. Using the MATLAB Command Window, enter:

```
h = raspberrypi
h.execute('sudo shutdown -h now')
```

When the Raspberry Pi shuts down, all LEDs except for the PWR LED turn off. Power recycle your Raspberry Pi hardware when you want to boot into Linux kernel again.

To reboot your Raspberry Pi hardware, execute the following command:

```
h.execute('sudo shutdown -r now')
```

A

- Abs block
 - zero crossings 3-35
- absolute tolerance
 - definition 14-22
 - simulation accuracy 19-11
- accelbuild command
 - building Accelerator MEX-file 22-27
- Acceleration
 - code regeneration 22-7
 - compilation overhead 19-5
 - debugger advantages 22-30
 - decision tree 22-14
 - designing for 22-16
 - how to run debugger with 22-30
 - inhibiting 22-20
 - numerical precision 22-17
 - verses normal mode 22-17
 - trade-offs 22-11
- Accelerator
 - description 22-2
 - determining why it rebuilds 22-8
 - how it works 22-3
 - making run time changes 22-26
 - Simulink blocks whose performance is not improved by 22-16
 - switching back to normal mode 22-31
 - using with Simulink debugger 22-30
- Accelerator and Rapid Accelerator
 - choosing between 22-11
 - comparing 22-11
- Accelerator mode
 - keywords 22-21
- Accelerators
 - customizing build process 22-23
 - interacting programmatically 22-27
 - running 22-24
- Action Port block
 - in subsystem 4-47
- activating
 - configuration references 10-21
- Adams-Bashforth-Moulton PECE solver 14-19
- Add button 54-9
- adding
 - cells 54-7
 - rows 54-7
 - text to cells 54-9
 - variable content to cells 54-9
- Algebraic loop** diagnostic 3-43
- algebraic loop solver
 - limitations 3-51
 - line-search algorithm 3-50
 - operations 3-49
 - trust-region algorithm 3-50
- algebraic loops
 - artificial
 - atomic subsystems causing 3-55
 - bundled signals causing 3-58
 - eliminating using Simulink 3-69
 - problems caused by 3-43
 - avoiding discontinuities in 3-51
 - changing block priorities effect on 3-55
 - definition 3-40
 - direct feedthrough blocks 3-39
 - displaying 3-43 to 3-45
 - highlighting 21-36
 - identifying blocks in 21-33
 - mathematical definition of 3-41
 - physical meaning of 3-42
 - problems caused by 3-43
 - simulation speed 19-3
 - using Algebraic Constraint blocks to
 - remove 3-54
 - using IC blocks to remove 3-54
- aligning blocks 23-12
- annotations
 - changing font 4-26
 - creating 4-23
 - definition 4-23
 - deleting 4-27

- editing 4-25
- moving 4-24
- using symbols and Greek letters in 4-33
- using TeX formatting commands in 4-33
- using to document models 12-7
- arguments
 - limit on number for code generation from MATLAB 44-19
- artificial algebraic loops
 - atomic subsystems causing 3-55
 - bundled signals causing 3-58
- Assignment block
 - and For Iterator block 4-53
- associating
 - bus objects
 - with model entities 51-25
- asynchronous sample time 5-19
- atomic subsystem 3-12
- attaching
 - configuration references 10-18
 - to additional models 10-38
- attributes format string 4-21
- AttributesFormatString block
 - parameter 23-21

B

- Backlash block
 - zero crossings 3-35
- backpropagating sample time 5-35
- Backspace key
 - deleting blocks 23-13
 - deleting labels 50-23
- Band-Limited White Noise block
 - simulation speed 19-3
- block
 - simulation terminated or suspended 14-5
- block callback parameters 4-64
- block callbacks
 - adding custom functionality 27-38
- Block data tips 23-2
- block diagram
 - updating 1-32
- block diagrams
 - printing 1-34
- block libraries
 - adding to Library Browser 28-32
 - creating 28-20
 - locking 28-21
 - modifying 28-21
 - searching 4-5
- block methods
 - invoking 23-36
- block names
 - changing location 23-28
 - copied blocks 23-10
 - editing 23-27
 - flipping location 23-28
 - generated for copied blocks 23-10
 - hiding and showing 23-29
 - location 23-27
 - rules 23-27
- block parameters
 - about 24-1
 - displaying beneath a block 23-50
 - modifying during simulation 14-31
 - scalar expansion 50-38
 - setting 24-4
- block priorities
 - assigning 23-50
 - effect on algebraic loops 3-55
 - rules 23-51
- Block Properties dialog box 23-15
- blocks
 - aligning 23-12
 - assigning priorities 23-50
 - associating user data with 46-59
 - autoconnecting 4-12
 - bus-capable 51-22
 - callbacks 4-56

- changing font 23-28
- changing font names 23-28
- changing location of names 23-28
- checking connections 3-18
- connecting automatically 4-12
- connecting manually 4-15
- copying from Library Browser 4-5
- copying into models 23-10
- custom, Simulink 27-2
 - categories 27-3
 - code generation requirements 27-8
 - designing 27-18
 - examples 27-44
 - incorporating legacy code 27-6
 - modeling requirements 27-7
 - simulation requirements 27-8
 - tutorial 27-18
 - using block callbacks 27-38
 - using MATLAB functions 27-3
 - using S-functions 27-4
 - using Subsystems 27-4
- deleting 23-13
- disconnecting 4-21
- displaying sorted order on 23-36
- drop shadows 23-26
- duplicating 23-9
- grouping to create subsystem 4-38
- hiding block names 23-29
- input ports with direct feedthrough 3-39
- invoking methods 23-36
- moving between windows 23-11
- moving in a model 23-11
- names
 - editing 23-27
- orientation 23-23
- resizing 23-25
- reversing signal flow through 12-8
- rotating 23-23
- rules for priorities 23-51
- showing block names 23-29
 - updating 3-18
- <>blocks 23-27
 - See also* block names
- borders
 - creating 54-7
 - important note about 54-7
- bounding box
 - grouping blocks for subsystem 4-38
 - selecting objects 4-7
- branch lines 4-16
- Break Library Link menu item 28-18
- breaking links to library block 28-18
- breakpoints
 - setting 21-22
 - setting at end of block 21-25
 - setting at timesteps 21-25
 - setting on nonfinite values 21-26
 - setting on step-size-limiting steps 21-26
 - setting on zero crossings 21-26
- Browser 9-80
- build information
 - updating 29-245
- building models
 - tips 12-6
- Bus Editor 51-24
 - opening 51-29
- bus objects
 - associating
 - with model entities 51-25
 - creating
 - with the API 51-27
 - with the Bus Editor 51-33
 - using 51-24
- Bus to Vector block
 - backward compatibility 51-112
- bus-capable blocks 51-22
 - sorted order and 23-47
- buses 51-65
 - connecting to inports 51-65
 - mixing with muxes 51-100

- nesting 51-20
- busses used as muxes
 - correcting 51-109

C

C compiler

- set up for code generation 29-213
- set up for code generation from MATLAB 29-185

C/C++ code generation for supported functions 34-1

callback tracing 4-73

callbacks 4-56

callbacks, referencing mask parameters in 4-69

cells

- adding 54-7
- adding text to 54-12
- adding variable content to 54-11
- deleting 54-7
- resizing 54-8
- selecting 54-7
- splitting 54-7

changing

- values using configuration references 10-23
10-39

characters, special 54-12

classes

- enumerated 47-1

Clear menu item 23-13

Clipboard block callback parameter 4-66

Clock block

- example 18-3

clone method 30-14

CloseFcn block callback parameter 4-69

CloseFcn model callback parameter 4-62

code generation for MATLAB

- comparing to MATLAB S-functions 27-5

code generation from MATLAB

- benefits of 33-2

- best practices for working with variables 36-3

- calling local functions 44-9

- calling MATLAB functions 44-11

- calling MATLAB functions using feval 44-16

- characters 37-6

- communications system toolbox System objects 35-7

- compilation directive `%#codegen` 44-8

- computer vision system toolbox System objects 35-2

- converting mxArray to known types 44-18

- declaring MATLAB functions as extrinsic functions 44-12

- defining persistent variables 36-10

- defining variables 36-2

- defining variables by assignment 36-3

- dsp system toolbox System objects 35-13

- eliminating redundant copies of function inputs 45-7

- eliminating redundant copies of uninitialized variables 36-7

- how it resolves function calls 44-2

- image acquisition toolbox System objects 35-23

- initializing persistent variables 36-10

- inlining functions 45-9

- limit on number of function arguments 44-19

- phased array system toolbox System objects 35-19

- pragma 44-8

- resolving extrinsic function calls during simulation 44-16

- resolving extrinsic function calls in generated code 44-17

- rules for defining uninitialized variables 36-7

- setting properties of indexed variables 36-6

- supported toolbox functions 44-10

- using type cast operators in variable

- definitions 36-6

- variables, complex 37-4
- when not to use 33-2
- when to use 33-2
- which features to use 33-4
- working with mxArray 44-17
- Code Generation from MATLAB
 - C compiler set up 29-185 29-213
 - getting started tutorial prerequisites 29-179 29-206
- `coder.ExternalDependency` 29-239
 - best practices 29-243
- `coder.extrinsic` 44-12
 - using to debug MATLAB code 29-215
- `coder.nullcopy`
 - uninitialized variables 36-7
- `coder.updateBuildInfo` 29-245
- colors for sample times 5-35
- command line debugger for MATLAB Function block 29-28
- commands
 - undoing 1-23
- communications system toolbox System objects supported for code generation from MATLAB 35-7
- Compare To Constant block
 - zero crossings 3-35
- Compare To Zero block
 - zero crossings 3-35
- compiled sample time 5-21
- Compiled Size property for MATLAB Function block variables 29-75
- compilers
 - supported for MATLAB Function blocks (code generation) 29-12
 - supported for MATLAB Function blocks (simulation) 29-12
- composite signals 51-3
- computer vision system toolbox System objects supported for code generation from MATLAB 35-2
- conditional execution behavior 7-50
- conditionally executed subsystem 3-11
- conditionally executed subsystems 7-2
- configurable subsystem 4-137
- Configuration Parameters dialog box
 - increasing Accelerator performance 19-9
- configuration references
 - activating 10-21
 - and building models 10-25
 - and generating code 10-25
 - attaching 10-18
 - to additional models 10-38
 - changing values with 10-23 10-39
 - creating 10-18
 - limitations 10-31
 - obtaining handles 10-40
 - obtaining values with 10-39
- configuration sets
 - copying 10-36
 - reading from an M-file 10-36
 - referencing 10-18
 - using 10-12
- connecting
 - buses to inports 51-65
 - buses to root-level inports 51-65
- connecting blocks 4-15
- ConnectionCallback
 - port callback parameters 4-72
- constant sample time 5-17
- content
 - adding to cells 54-9
 - in print frames 54-5
 - multiple entries in a cell 54-10
- content list box 54-9
- ContinueFcn model callback parameter 4-63 4-68
- continuous sample time 5-16
- control flow diagrams
 - do-while 4-51
 - for 4-52

- if-else 4-46
- switch 4-48
- while 4-49
- control flow subsystem 7-2
- control input 7-2
- control signal 7-2
- Control System Toolbox
 - linearization 18-6
- controlling run-time checks
 - MATLAB Function block 29-176
- Copy menu item 23-10
- CopyFcn block callback parameter 4-66
- copying
 - blocks 23-10
 - configuration sets 10-36
 - signal labels 50-23
- correcting
 - buses used as muxes 51-109
- Created model parameter 4-127
- creating
 - bus objects
 - with the API 51-27
 - with the Bus Editor 51-33
 - configuration references 10-18
 - custom Simulink blocks 27-2
 - freestanding configuration sets 10-36
- Creator model parameter 4-127
- custom blocks
 - creating 27-2
- Customizing Accelerator Build
 - AccelVerboseBuild 22-29
 - SimCompilerOptimization 22-29
- Cut menu item 23-11

D

- data
 - enumerated 47-1
- data range checking
 - MATLAB Function blocks 29-30
- data types
 - displaying 46-28
 - enumerated 47-1
 - propagation definition 46-29
 - specifying 46-8
- data types of MATLAB Function variables 29-70
- data, adding to MATLAB Function blocks 29-43
- Dead Zone block
 - zero crossings 3-35
- debugger
 - highlighting algebraic loops using 3-45
 - running incrementally 21-17
 - setting breakpoints 21-22
 - setting breakpoints at time steps 21-25
 - setting breakpoints at zero crossings 21-26
 - setting breakpoints on nonfinite values 21-26
 - setting breakpoints on step-size-limiting steps 21-26
 - skipping breakpoints 21-20
 - starting 21-10
 - stepping by time steps 21-20
- debugging
 - breakpoints in MATLAB Function block function 29-25
 - display variable values in MATLAB Function block function 29-28
 - displaying MATLAB Function block variables in MATLAB 29-28
 - MATLAB Function block example 29-24
 - MATLAB Function block function 29-24
 - operations for debugging MATLAB functions 29-30
 - stepping through MATLAB Function block function 29-26
- decimation factor
 - saving simulation output 48-15 to 48-16
- default print frame 54-2
- defining uninitialized variables
 - rules 36-7
- defining variables

- for C/C++ code generation 36-3
- Delete key
 - deleting annotations 4-27
 - deleting blocks 23-13
 - deleting signal labels 50-23
- DeleteChildFcn block callback parameter 4-70
- DeleteFcn block callback parameter 4-67
- deleting
 - cells 54-7
 - rows 54-7
- dependency analysis 9-83 13-159
 - best practices 13-178
 - comparing manifests 13-172
 - editing manifests 13-169
 - exporting manifests 13-173
 - file manifests 13-175
 - generating manifests 13-160
 - viewing dependencies 9-83
- Derivative block
 - linearization 18-9
- Description model parameter 4-128
- Description property
 - MATLAB Function blocks 29-43
- design considerations
 - when writing MATLAB Code for code generation 33-7
- designing
 - custom Simulink blocks 27-18
- DestroyFcn block callback parameter 4-67
- diagnosing simulation errors 14-39
- diagonal line segments 4-17
- direct feedthrough blocks 3-39
- direct-feedthrough ports
 - sorted order and 23-47
- disabled subsystem
 - output 7-20
- disconnecting blocks 4-21
- discrete blocks
 - in enabled subsystem 7-24
 - in triggered systems 7-36

- discrete sample time 5-15
- discrete states
 - initializing 50-52
- discretization methods 4-134
- discretizing a Simulink model 4-131
- dlinmod function
 - extracting linear models 18-5
- do-while control flow diagram 4-51
- Docking Scope Viewer 16-15
- Document link property
 - MATLAB Function blocks 29-43
- drop shadows 23-26
- dsp system toolbox System objects
 - supported for code generation from MATLAB 35-13
- duplicating blocks 23-9

E

- editing
 - MATLAB Function block function code 29-34
 - mode 54-9
- editing look-up tables 25-28
- editor 1-19
- either trigger event 7-33
- eliminating redundant copies of function
 - inputs 45-7
- Enable block
 - creating enabled subsystems 7-18
 - outputting enable signal 7-23
 - states when enabling 7-22
 - zero crossings 3-35
- enabled subsystems 7-17
 - initializing output of 50-56
 - setting states 7-22
- ending Simulink session 1-58
- enumerated data types 47-1
- Enumerations 47-1
- error checking
 - MATLAB Function blocks 29-12

- error tolerance 14-22
 - simulation accuracy 19-11
 - simulation speed 19-3
- ErrorFcn block callback parameter 4-71
- examples
 - Clock block 18-3
 - continuous system 12-8
 - converting Celsius to Fahrenheit 12-15
 - equilibrium point determination 18-11
 - linearization 18-5
 - multirate discrete model 5-26
 - Outport block 18-3
 - return variables 18-3
 - structures in a MATLAB Function block 29-81
 - To Workspace block 18-3
 - Transfer Function block 12-9
 - working with frame-based signals in MATLAB Function blocks 29-131
- execution context
 - defined 7-52
 - displaying 7-54
 - propagating 7-52
- Exit MATLAB menu item 1-58
- external code interface
 - encapsulating 29-239
- External mode
 - External Mode Control Panel 55-17
- extrinsic functions 44-12
 - calling in MATLAB Function block functions 29-6
- F**
- falling trigger event 7-33
- Fcn block
 - simulation speed 19-3
- figure file 54-6
- file name
 - maximum length 1-11
- files
 - writing to 14-5
- fixed in minor step 5-16
- fixed-point data 46-4
 - properties 29-71
 - properties in Simulink model 46-20
- fixed-step solvers
 - definition 3-22
- Flip Block Name menu item 23-28
- floating Display block 14-31
- floating Scope block 14-31
- font
 - annotations 4-26
 - block 23-28
 - block names 23-28
 - special characters 54-12
 - symbols 54-12
- Font menu item
 - changing block name font 23-28
- font size, setting for Model Explorer 9-6
- font size, setting for Simulink dialog boxes 9-6
- for control flow diagram 4-52
- For Iterator block
 - and Assignment block 4-53
 - in subsystem 4-52
 - output iteration number 4-53
 - specifying number of iterations 4-53
- frame-based signals
 - adding frame-based data to MATLAB Function blocks 29-130
 - examples of use in MATLAB Function blocks 29-131
 - using in MATLAB Function blocks 29-129
- frameedit
 - starting 54-6
- frames 54-2
- freestanding configuration sets
 - creating 10-36
- From File block
 - zero crossings 3-36

- From Workspace block
 - zero crossings 3-36
 - function call outputs, adding to MATLAB
 - Function blocks 29-51
 - Function-Call Split blocks
 - sorted order and 23-46
 - function-call subsystems
 - sorted order and 23-44
 - functions
 - limit on number of arguments for code generation 44-19
 - Functions supported for C/C++ code
 - generation 34-1
 - alphabetical list 34-2
 - arithmetic operator functions 34-82
 - bit-wise operation functions 34-82
 - casting functions 34-83
 - Communications System Toolbox
 - functions 34-83
 - complex number functions 34-83
 - Computer Vision System Toolbox
 - functions 34-84
 - data and file management functions 34-86
 - data type functions 34-86
 - derivative and integral functions 34-87
 - discrete math functions 34-87
 - error handling functions 34-88
 - exponential functions 34-88
 - filtering and convolution functions 34-89
 - Fixed-Point Designer functions 34-89
 - histogram functions 34-98
 - Image Processing Toolbox functions 34-98
 - input and output functions 34-100
 - interpolation and computational geometry
 - functions 34-101
 - linear algebra functions 34-101
 - logical operator functions 34-101
 - MATLAB Compiler functions 34-102
 - MATLAB Desktop environment
 - functions 34-102
 - matrix/array functions 34-102
 - nonlinear numerical methods 34-106
 - Phased Array System Toolbox
 - functions 34-107
 - polynomial functions 34-110
 - programming utilities 34-110
 - relational operator functions 34-110
 - rounding and remainder functions 34-111
 - set functions 34-111
 - signal processing functions 34-112
 - Signal Processing Toolbox functions 34-112
 - special value functions 34-117
 - specialized math functions 34-117
 - statistical functions 34-118
 - Statistics Toolbox functions 34-118
 - string functions 34-124
 - structure functions 34-125
 - trigonometric functions 34-125
 - Functions supported for MEX and C/C++ code
 - generation
 - categorized list 34-80
 - fundamental sample time 14-11
- G**
- Generator
 - performing common tasks 16-21
 - removing 16-22
 - get_param command
 - checking simulation status 15-8
 - getDiscreteState method 30-14
 - getNumInputs method 30-13
 - getNumOutputs method 30-13
 - Go To Library Link menu item 28-7
 - Greek letters 54-12
 - using in annotations 4-33
 - grouping blocks 4-37

H

- handles on selected object 4-6
- held output of enabled subsystem 7-20
- held states of enabled subsystem 7-22
- Hide Port Labels menu item 4-43
- hiding block names 23-29
- hierarchy of model
 - advantage of subsystems 12-7
 - replacing virtual subsystems 3-18
- highlighting
 - requirements in a model 9-97
- Hit Crossing block
 - notification of zero crossings 3-37
 - zero crossings 3-36
- how to disable run-time checks
 - MATLAB Function block 29-177
- hybrid systems
 - integrating 5-29

I

- If block
 - connecting outputs 4-47
 - data input ports 4-47
 - data output ports 4-47
 - zero crossings
 - and Disable zero crossing detection option 3-36
- if-else control flow diagram 4-46
- image acquisition toolbox System objects
 - supported for code generation from MATLAB 35-23
- importing initial states 48-146
- indexed variables
 - setting properties for code generation from MATLAB 36-6
- info method 30-14
- inherited sample time 5-16
- inheriting MATLAB Function block variable size 29-75
- inheriting MATLAB Function variable types 29-69
- InitFcn block callback parameter 4-68
- InitFcn model callback parameter 4-63
- initial conditions
 - specifying 48-142
- Initial State check box 48-146
- initial states
 - loading 48-146
- initial step size
 - simulation accuracy 19-11
- initial values
 - tuning 50-54
- initialization
 - persistent variables 36-10
- inlining S-functions using the TLC and Accelerator performance 22-17
- Inport block
 - in subsystem 4-38
 - linearization 18-6
 - supplying input to model 48-72
- inports
 - connecting to buses 51-65
 - root-level
 - connecting to buses 51-65
- input triggers, adding to MATLAB Function blocks 29-49
- inputs
 - loading from a workspace 48-72
 - mixing vector and scalar 50-39
 - scalar expansion 50-38
- Integer Delay blocks
 - use in removing algebraic loops 3-51
- integer overflow and underflow
 - and simulation targets in MATLAB Function blocks 29-41
 - in MATLAB Function blocks 29-40 to 29-41
- Integrator block
 - example 12-8
 - sample time colors 5-35

- simulation speed 19-3
- zero crossings 3-36
- Interpreted MATLAB Function block
 - simulation speed 19-3
- invalid loops, avoiding 12-2
- invalid loops, detecting 12-3
- isDone method 30-14
- isLocked method 30-14

J

- Jacobian matrices 14-21

K

- keyboard actions summary 1-59
- Keywords
 - acceleration 22-21

L

- labeling signals 50-22
- labeling subsystem ports 4-43
- LastModifiedBy model parameter 4-128
- LastModifiedDate model parameter 4-128
- libinfo command 28-7
- libraries 12-17
- library blocks
 - breaking links to 28-18
 - finding 28-7
 - getting information about 28-16
- Library Browser
 - adding libraries to 28-32
 - copying blocks from 4-5
- library links
 - disabling 28-11
 - displaying 28-8
 - self-modifying 28-7
 - showing in Model Browser 9-82
 - status of 28-16
 - unresolved 28-19

- line segments 4-17
 - diagonal 4-19
 - moving 4-18
- line vertices
 - moving 4-20
- line-search algorithm
 - Simulink algebraic loop solver 3-50
- linear models
 - extracting
 - example 18-5
- linearization 18-5
- lines
 - branch 4-16
 - connecting blocks 4-12
 - dragging 54-8
 - moving 23-11
 - selecting 54-7
 - signals carried on 14-31
- linked blocks
 - about 28-4
 - creating 28-4
 - modifying 28-6
 - updating 28-5
- links
 - breaking 28-18
- LinkStatus block parameter 28-16
- linmod function
 - example 18-5
- LoadFcn block callback parameter 4-65
- loading from a workspace 48-72
- local functions
 - in MATLAB Function block functions 29-6
- location of block names 23-27
- Lock Editor property
 - MATLAB Function blocks 29-41
- logging signals 48-21
- look-up tables, editing 25-28
- Lookup Table Editor 25-28
- lookup tables
 - blocks 25-5

- components 25-4
- data characteristics 25-18
- data entry 25-11
- definition 25-2
- estimation 25-23
- examples for Prelookup and Interpolation
 - Using Prelookup blocks 25-67
- extrapolation 25-24
- interpolation 25-23
- rounding 25-25
- selection guidelines 25-7
- terminology 25-78

loops, algebraic. *See* algebraic loops

loops, avoiding invalid 12-2

loops, detecting invalid 12-3

M

manifest 13-175

masked blocks

- parameters
 - referencing in callbacks 4-69
- showing in Model Browser 9-82

masked subsystems

- showing in Model Browser 9-82

mathematical symbols 54-12

- using in annotations 4-33

MATLAB

- features not supported for code generation 33-14
- terminating 1-58

MATLAB file S-functions

- simulation speed 19-3

MATLAB for code generation

- variable types 36-18

MATLAB function block

- MATLAB Function report keyboard shortcuts 29-64

MATLAB Function block

- calling MATLAB code 29-215
- controlling run-time checks 29-176
 - how to disable run-time checks 29-177
 - resolving signal objects 29-78
 - using 29-187 29-214
 - when to disable run-time checks 29-176

MATLAB Function Block Editor

- description 29-10

MATLAB Function block fimath

- MATLAB Function blocks 29-41

MATLAB Function blocks 29-53

- adding data using the Ports and Data Manager 29-43
 - adding frame-based data 29-130
 - adding function call outputs using the Ports and Data Manager 29-51
 - adding input triggers using the Ports and Data Manager 29-49
 - and embedded applications 29-8
 - and standalone executables 29-8
- breakpoints in function 29-25
- built-in data types 29-70
- calling extrinsic functions 29-6
- calling MATLAB functions 29-6
- creating model with 29-9
- data range checking 29-30
- debugging 29-24
 - debugging example 29-24
 - debugging function for 29-24
 - debugging operations 29-30
- description 29-53
- Description property 29-43
- diagnostic errors 29-12
- display variable value 29-28
- displaying variable values in MATLAB 29-28
- Document link property 29-43
- example containing structures 29-81
- example model with 29-9
- example program 29-10
- inherited data types and sizes 29-8
- inheriting variable size 29-75

- local functions 29-6
- Lock Editor property 29-41
- MATLAB Function Block Editor 29-10 29-34
- MATLAB Function block fimath 29-41
- MATLAB run-time library of functions 29-6
- Name property 29-39
- parameter arguments 29-77
- Ports and Data Manager 29-38
- Saturate on integer overflow property 29-40
- setting properties 29-39
- simulating function 29-24
- Simulink input signal properties 29-41
- sizing variables 29-75
- sizing variables by expression 29-76
- stepping through function 29-26
- supported compilers for code
 - generation 29-12
- supported compilers for simulation 29-12
- typing variables 29-67
- typing with other variables 29-71
- Update method property 29-39
- variable type by inheritance 29-69
- variables for 29-14
- why use them? 29-8
- working with frame-based signals 29-129
- MATLAB Function report keyboard shortcuts
 - MATLAB Function report 29-64
- MATLAB Function reports 29-53
 - description 29-53
- MATLAB functions
 - and generating code for mxArray 44-17
 - calling in MATLAB Function block
 - functions 29-6
- MATLAB run-time library functions 29-6
- mdl files 1-11
- Memory block
 - simulation speed 19-3
- memory issues 12-7
- Minimize algebraic loop occurrences**
 - parameter 3-63
- Minimize algebraic loop** parameter 3-63
- MinMax block
 - zero crossings 3-36
- mixed continuous and discrete systems 5-29
- mixing
 - muxes, and buses 51-100
- model
 - editor 1-19
- Model Browser 9-80
 - showing library links in 9-82
 - showing masked subsystems in 9-82
- model callback parameters 4-58
- model configuration preferences 9-11
- model dependencies 9-83 13-159
 - best practices 13-178
 - comparing manifests 13-172
 - editing manifests 13-169
 - exporting manifests 13-173
 - file manifests 13-175
 - generating manifests 13-160
 - viewing 9-83
- model dependency viewer 9-83
- model discretization
 - configurable subsystems 4-137
 - discretizing a model 4-131
 - overview 4-130
 - specifying the discretization method 4-134
 - starting the model discretizer 4-133
- Model Explorer
 - Apply Changes 9-71
 - Auto Apply/Ignore Dialog Changes 9-71
 - Dialog pane 9-70
 - font size 9-6
- model file name, maximum size of 1-11
- model files
 - mdl file 1-11
- model navigation commands 4-40
- model referencing 12-17
- ModelCloseFcn block callback parameter 4-69
- modeling strategies 12-7

- modeling techniques 12-17
 - models
 - callbacks 4-56
 - creating 4-2
 - creating change histories for 4-125
 - creating templates 4-2
 - editing 1-5
 - highlighting requirements in 9-97
 - navigating 4-40
 - navigating to requirements documents
 - from 9-100
 - organizing and documenting 12-7
 - printing 1-34
 - properties of 4-118
 - saving 1-10
 - selecting entire 4-7
 - tips for building 12-6
 - ModelVersion model parameter 4-129
 - ModelVersionFormat model parameter 4-129
 - ModifiedBy model parameter 4-128
 - ModifiedByFormat model parameter 4-128
 - ModifiedComment model parameter 4-128
 - ModifiedDateFormat model parameter 4-128
 - ModifiedHistory> model parameter 4-128
 - Monte Carlo analysis 15-2
 - mouse actions summary 1-59
 - MoveFcn block callback parameter 4-65
 - multiple entries in a cell 54-10
 - multirate discrete systems
 - example 5-27
 - muxes
 - correcting busses used as 51-109
 - mixing with buses 51-100
 - mxArrays
 - converting to known types 44-18
 - for code generation from MATLAB 44-17
- N**
- Name property
 - MATLAB Function blocks 29-39
 - NameChangeFcn block callback parameter 4-65
 - names
 - blocks 23-27
 - copied blocks 23-10
 - navigating
 - from model to requirements documents 9-100
 - nesting
 - buses 51-20
 - New menu item 4-2
 - nonvirtual buses
 - compared with virtual 51-13
 - specifying 51-15
 - nonvirtual subsystem 3-11
 - numerical differentiation formula 14-21
 - numerical integration 3-19
- O**
- objects
 - selecting more than one 4-6
 - selecting one 4-6
 - obtaining
 - configuration reference handles 10-40
 - values using configuration references 10-39
 - ode113 solver
 - hybrid systems 3-8
 - ode113 solver
 - advantages 14-19
 - Memory block
 - and simulation speed 19-3
 - ode15s solver
 - advantages 14-21
 - and stiff problems 19-3
 - hybrid systems 3-8
 - Memory block
 - and simulation speed 19-3
 - unstable simulation results 19-11
 - ode23 solver
 - hybrid systems 3-8

- ode23s solver
 - advantages 14-21
 - simulation accuracy 19-11
- ode45 solver
 - hybrid systems 3-8
- Open menu item 1-5
- OpenFcn block callback parameter
 - purpose 4-65
- opening
 - PrintFrame Editor 54-6
 - Subsystem block 4-39
 - the Bus Editor 51-29
- optimizing generated code
 - unrolling for-loops 45-41
- orientation of blocks 23-23
- Output block
 - example 18-3
 - in subsystem 4-38
 - linearization 18-6
- output
 - additional 48-18
 - between trigger events 7-36
 - disabled subsystem 7-20
 - enable signal 7-23
 - smoother 48-17
 - specifying for simulation 48-19
 - subsystem initial output 7-18
 - trajectories
 - viewing 18-2
 - trigger signal 7-36
 - writing to file
 - when written 14-5
 - writing to workspace
 - when written 14-5
- output ports
 - Enable block 7-23
 - Trigger block 7-36

P

- PaperOrientation model parameter 1-49
- PaperPosition model parameter 1-49
- PaperPositionMode model parameter 1-49
- PaperType model parameter 1-49
- parameter arguments for MATLAB Function blocks 29-77
- Parameter values
 - checking
 - using get_param 24-8
- parameters
 - block 24-1
 - setting values of 24-4
 - tunable
 - definition 3-9
- ParentCloseFcn block callback parameter 4-71
- Paste menu item 23-10
- PauseFcn model callback parameter 4-63 4-68
- performance
 - comparing Acceleration to Normal Mode 19-7
- persistent variables
 - defining for code generation from MATLAB 36-10
 - initializing for code generation from MATLAB 36-10
- phased array system toolbox System objects
 - supported for code generation from MATLAB 35-19
- ports
 - default port rotation 23-24
 - labeling in subsystem 4-43
 - physical port rotation 23-24
- Ports and Data Manager
 - adding data to MATLAB Function blocks 29-43
 - adding function call outputs to MATLAB Function blocks 29-51
 - adding input triggers to MATLAB Function blocks 29-49

- Ports and Data Manager, MATLAB Function Block 29-38
 - PostLoadFcn model callback parameter 4-62
 - PostSaveFcn block callback parameter 4-68
 - PostSaveFcn model callback parameter 4-62
 - PreCopyFcn block callback parameter 4-66
 - PreDeleteFcn block callback parameter 4-66
 - preferences, model configuration 9-11
 - PreLoadFcn model callback parameter 4-61
 - PreSaveFcn block callback parameter 4-68
 - PreSaveFcn model callback parameter 4-62
 - print command 1-34
 - print frames
 - defined 54-2
 - Print menu item 1-34
 - PrintFrame Editor
 - interface for 54-6
 - starting 54-6
 - printing with print frames
 - block diagrams 54-13
 - multiple use 54-5
 - Priority block parameter 23-50
 - produce additional output option 48-18
 - produce specified output only option 48-19
 - profiler
 - enabling 22-34
 - Profiler
 - how it works 22-32
 - properties
 - setting for MATLAB Function blocks 29-39
 - properties of Scope Viewer 16-16
 - purely discrete systems 5-26
- R**
- Random Number block
 - simulation speed 19-3
 - Rapid Accelerator
 - description 22-2
 - determining why it rebuilds 22-8
 - how to run 22-23
 - Simulink blocks not supported 22-16 to 22-17
 - visualization 22-19
 - Rapid Accelerator mode
 - how to run 22-23
 - keywords 22-21
 - rate transitions 5-32
 - reading
 - configuration sets from an M-file 10-36
 - Redo menu item 1-23
 - referencing
 - configuration sets 10-18
 - refine factor
 - smoothing output 48-17
 - Relational Operator block
 - zero crossings 3-36
 - relative tolerance
 - definition 14-22
 - simulation accuracy 19-11
 - Relay block
 - zero crossings 3-36
 - release method 30-13
 - requirements
 - features available in Simulink 9-96
 - features requiring Simulink Verification and Validation license 9-96
 - highlighting 9-97
 - in subsystems 9-97
 - navigating to 9-100
 - navigating to, from System Requirements block 9-101
 - viewing details 9-99
 - Requirements Management Interface (RMI)
 - Simulink Verification and Validation license required for 9-96
 - Reserved
 - accelerator mode keywords 22-21
 - reset
 - output of enabled subsystem 7-20

- states of enabled subsystem 7-22
- reset method 30-13
- resizing blocks 23-25
- resizing rows and cells 54-8
- resolving
 - signal objects for MATLAB Function blocks 29-78
- return variables
 - example 18-3
- reversing direction of signal flow 12-8
- rising trigger event 7-33
- root-level inports
 - connecting to buses 51-65
- Rosenbrock formula 14-21
- Rotate & Flip>Clockwise menu item 23-23
- Rotate & Flip>Counter-clockwise menu item 23-23
- rotating a block 23-23
- rows
 - adding 54-7
 - deleting 54-7
 - resizing 54-8
 - selecting 54-7
 - splitting 54-7
- RTW.BuildInfo
 - updating 29-245

S

- sample time
 - backward propagating 5-35
 - colors 5-35
 - fundamental 14-11
 - simulation speed 19-3
- Sample Time Legend 5-10
- sample time propagation 5-33
- saturate
 - on integer overflow in MATLAB Function blocks 29-40
- Saturate on integer overflow property

- MATLAB Function blocks 29-40
- saturation block
 - zero crossings
 - how used 3-37
- Saturation block
 - zero crossings 3-36
- Save As menu item 1-11
- Save menu item 1-11
- save_system command
 - breaking links 28-18
- scalar expansion 50-38
- Scope block
 - example of a continuous system 12-9
- Scope blocks and viewers
 - differences between 16-3
- Scope Viewer 16-1
 - adding multiple signals to 16-8
 - attaching 16-6
 - axes 16-16
 - data markers 16-17
 - how to display 16-9
 - legends 16-11
 - limiting data 16-18
 - performance 16-20
 - performing common tasks 16-6
 - properties dialog 16-16
 - refresh 16-19
 - saving axes settings
 - gui 16-14
 - scroll 16-17
 - signal logging 16-18
 - simulation speed 19-4
 - toolbar 16-14
- Scopes
 - using with Rapid Accelerator 22-19
- Second-Order Integrator, zero crossings 3-36
- Select All menu item 4-7
- selecting cells, rows, and lines 54-7
- Set Font dialog box 23-28
- set_param command

- breaking link 28-18
- controlling model execution 22-27
- running a simulation 14-3 15-8
- setting block parameters within MATLAB
 - S-functions 27-34
- setting simulation mode 22-27
- setting breakpoints 21-22
- shadowed files 12-4
- Shampine, L. F. 14-21
- Shape preservation
 - improving solver accuracy 3-23
- show output port
 - Enable block 7-23
 - Trigger block 7-36
- showing block names 23-29
- Sign block
 - zero crossings 3-36
- Signal Builder
 - snap grid 50-117
- Signal Builder block
 - zero crossings 3-36
- Signal Builder time range
 - about 50-119
 - changing 50-119
- Signal Builder window 50-70
- signal groups 50-69
 - activating 50-121
 - copying and pasting 50-78
 - creating a custom waveform in 50-73 50-79
 - creating a set of 50-84
 - creating and deleting 50-72
 - creating signals in 50-72
 - deleting 50-79
 - discrete 50-124
 - editing 50-69
 - exporting to workspace 50-120
 - final values 50-122
 - hiding waveforms 50-73
 - moving 50-72
 - outputting 50-77
 - renaming 50-72
 - renaming signals in 50-121
 - running all 50-121
 - simulating with 50-121
 - specifying final values for 50-122
 - specifying sample time of 50-124
 - time range of 50-119
- signal labels
 - copying 50-23
 - creating 50-22
 - deleting 50-23
 - editing 50-22
 - moving 50-22
 - using to document models 12-7
- signal logging, enabling 48-24
- signal objects
 - resolving for MATLAB Function blocks 29-78
 - using to initialize signals 50-53
- signal processing functions
 - for C/C++ code generation 34-112
- signals
 - composite 51-3
 - initializing 50-52
 - labeling 50-22
 - labels 50-22
 - names 50-19
 - reversing direction of flow 12-8
 - virtual 50-11
- signals, creating 50-3
- signals, logging 48-21
- sim command
 - comparing performance 19-7
 - simulating an accelerated model 22-28
 - syntax 15-3
- simulation
 - accuracy 19-11
 - checking status of 15-8
 - displaying information about
 - algebraic loops 21-30
 - block execution order 21-33

- block I/O 21-28
 - debug settings 21-36
 - integration 21-32
 - nonvirtual blocks 21-35
 - nonvirtual systems 21-34
 - system states 21-31
 - zero crossings 21-36
- execution phase 3-19
- MATLAB Function block 29-24
- parameters
 - specifying 14-39
- running incrementally 21-17
- running nonstop 21-20
- speed 19-3
- stepping by breakpoints 21-22
- stepping by time steps 21-20
- unstable results 19-11
- Simulation Diagnostics Viewer 14-39
- simulation errors
 - diagnosing 14-39
- Simulation Options dialog box 50-122
- simulation time
 - compared to clock time 14-8
- Simulink
 - custom blocks, creating 27-2
 - ending session 1-58
 - icon 1-3
 - starting 1-3
 - terminating 1-58
- Simulink block library. *See* block libraries
- simulink command
 - starting Simulink 1-3
- Simulink dialog boxes
 - font size 9-6
- Simulink input signal properties
 - MATLAB Function blocks 29-41
- Simulink Profiler
 - purpose 22-32
- Simulink, printing with print frames 54-2
- Simulink.BlockDiagram.getChecksum command
 - determining why the Accelerators rebuilds
 - with 22-8
- size of
 - cells 54-8
 - rows 54-8
- size of block
 - changing 23-25
- sizing MATLAB Function block variables by
 - expression 29-76
- sizing MATLAB Function block variables by
 - inheritance 29-75
- sizing MATLAB Function variables 29-75
- smart guides 23-12
- snap grid, Signal Builder 50-117
- solvers
 - fixed-step
 - definition 3-22
 - ode113
 - advantages 14-19
 - and simulation speed 19-3
 - ode15s
 - advantages 14-21
 - and simulation speed 19-3
 - and stiff problems 19-3
 - simulation accuracy 19-11
 - ode23s
 - advantages 14-21
 - simulation accuracy 19-11
- sorted order
 - bus-capable blocks and 23-47
 - direct-feedthrough ports and 23-47
 - displaying 23-36
 - Function-Call Split blocks and 23-46
 - function-call subsystems and 23-44
 - how Simulink® determines 23-47
 - notation 23-37
 - virtual blocks and 23-43
- special characters 54-12
- specifying
 - nonvirtual buses 51-15

- speed of simulation 19-3
- splitting rows and cells 54-7
- stairs function 5-28
- Start menu item 12-16
- start time 14-8
- StartFcn block callback parameter 4-68
- StartFcn model callback parameter 4-62
- starting
 - PrintFrame Editor 54-6
- starting Simulink 1-3
- starting the model discretizer 4-133
- states
 - between trigger events 7-36
 - importing initial 48-146
 - when enabling 7-22
- states, discrete
 - initializing 50-52
- static content in cells 54-5
- status
 - checking simulation 15-8
- Step block
 - zero crossings 3-37
- step method 30-13
- step size
 - simulation speed 19-3
- stiff problems 14-21
- stiff systems
 - simulation speed 19-3
- stop time 14-8
- StopFcn block callback parameter 4-68
- StopFcn model callback parameter 4-63
- streaming data
 - using System objects 30-12
- subscript 54-12
- subsystem
 - atomic 3-12
 - conditionally executed 3-11
 - initial output 7-18
 - nonvirtual 3-11
 - virtual 3-11
- Subsystem block
 - adding to create subsystem 4-38
 - opening 4-39
- Subsystem Examples block library 12-2
- subsystem ports
 - labeling 4-43
- subsystem sample time
 - sample time 5-24
- subsystems 12-17
 - controlling access to 4-43
 - creating 4-37
 - highlighting requirements in 9-97
 - labeling ports 4-43
 - model hierarchy 12-7
 - opening 4-40
 - triggered and enabled 7-42
 - underlying blocks 4-39
 - undoing creation of 4-40
- summary of mouse and keyboard actions 1-59
- superscript 54-12
- Switch block
 - zero crossings 3-37
- Switch Case block
 - zero crossings
 - and Disable zero-crossing detection option 3-37
- switch control flow diagram 4-48
- SwitchCase block
 - adding cases 4-48
 - connecting to Action subsystem 4-49
 - data input 4-48
- symbols 54-12
- System object
 - clone method 30-14
 - getDiscreteState method 30-14
 - getNumInputs method 30-13
 - getNumOutputs method 30-13
 - info method 30-14
 - isDone method 30-14
 - isLocked 30-14

- methods 30-12
- release method 30-13
- reset method 30-13
- step method 30-13
- using with MATLAB code generation 30-5
- System Requirements block 9-101

T

- terminating MATLAB 1-58
- terminating Simulink 1-58
- terminating Simulink session 1-58
- test point icons 48-32 50-60
- test points 50-59
- TeX commands
 - using in annotations 4-33
- TeX sequences 54-12
- text
 - adding to cells 54-12
 - editing mode 54-9
 - special characters 54-12
- tic command
 - comparing performance 19-7
- time interval
 - simulation speed 19-3
- time range
 - of a Signal Builder block 50-119
- tips for building models 12-6
- To Workspace block
 - example 18-3
- toc command
 - comparing performance 19-7
- Transfer Fcn block
 - example 12-9
- Transport Delay block
 - linearization 18-9
- Trigger and Enabled Subsystem
 - zero crossings 3-37
- Trigger block
 - creating triggered subsystem 7-35

- outputting trigger signal 7-36
 - showing output port 7-36
 - zero crossings 3-37
- triggered and enabled subsystems 7-42
- triggered sample time 5-19
- triggered subsystems 7-33
- triggers
 - control signal
 - outputting 7-36
 - either 7-33
 - events 7-33
 - falling 7-33
 - input 7-33
 - rising 7-33
 - type parameter 7-36
- trust-region algorithm
 - Simulink algebraic loop solver 3-50
- tunable parameters
 - definition 3-9
- tutorial
 - creating a customized saturation block 27-18
- type cast operators
 - using in variable definitions 36-6
- typing MATLAB Function block variables with
 - other variables 29-71
- typing MATLAB Function variables 29-67

U

- Undo menu item 1-23
- UndoDeleteFcn block callback parameter 4-67
- undoing commands 1-23
- uninitialized variables
 - eliminating redundant copies in generated code 36-7
- unstable simulation results 19-11
- Update Diagram menu item
 - fixing bad link 28-19
 - out-of-date linked block 28-5
- Update method property

- MATLAB Function blocks 29-39
- updating a block diagram 1-32
- updating a diagram programmatically 15-8
- user
 - specifying current 4-118
- user data 46-59
- UserData 46-59
- UserDataPersistent 46-59
- using
 - bus objects 51-24

V

- variable content
 - adding to cells 54-11
 - defined 54-5
- variable sample time 5-18
- variable types supported for code generation
 - from MATLAB 36-18
- variables
 - creating for MATLAB Function blocks 29-14
 - eliminating redundant copies in C/C++ code
 - generated from MATLAB 36-7
- Variables
 - defining by assignment for code generation
 - from MATLAB 36-3
 - defining for code generation from
 - MATLAB 36-2
- vector length
 - checking 3-18
- vertices
 - moving 4-20
- Viewers
 - using with Rapid Accelerator 22-19
- Viewers and generators
 - when to use 16-4
- viewing output trajectories 18-2
- viewing sample time information
 - Sample Time Display 5-9
- virtual blocks 23-2

- sorted order and 23-43
- virtual buses
 - compared with nonvirtual 51-13
- virtual signals 50-11
- virtual subsystem 3-11
- visualization
 - Rapid Accelerator limitations 22-19
- Visualizing
 - simulation results 16-1

W

- when to disable run-time checks
 - MATLAB Function block 29-176
- while control flow diagram 4-49
- While Iterator block
 - changing to do-while 4-51
 - condition input 4-51
 - in subsystem 4-50
 - initial condition input 4-51
 - iterator number output 4-51
- workspace
 - loading from 48-72
 - writing to
 - simulation terminated or suspended 14-5

Z

- zero
 - zero-crossing threshold 3-34
- zero crossing detection 3-24
- zero crossings
 - disabled by non-double data types 46-30
 - saturation block 3-37
- zero-crossing
 - adaptive 3-33
 - algorithms 3-33
 - blocks that register 3-35
 - demonstrating effects 3-24

missing events 3-30
nonadaptive 3-33
preventing excessive 3-30

threshold 3-34
zero-crossing detection 3-24
zero-crossing slope method 7-18